Friedrich-Alexander-Universität
Erlangen-Nürnberg

Master Thesis in Computer Science

A Framework for Defining Structure-Preserving Diagram Operators

Navid Roux

Advisors: PD Dr. habil. Florian Rabe, Prof. Dr. Michael Kohlhase

Erlangen, 1st April 2022

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, 1. April 2022 [illegible]

A Framework for Defining Structure-Preserving Diagram Operators

Navid Roux
FAU Erlangen-Nuremberg, Germany

Abstract

Formal systems such as interactive or automated theorem provers, specification systems, and deduction systems all enable the mechanized verification of developments and proofs in formal sciences. Moreover, they can serve the inevitable need to represent and categorize the ever-increasing amount of knowledge; a task that humans fail at large scales. Importantly, the utility of every formal system hinges on the availability of a comprehensive and well-designed standard library of concepts and reasoning techniques. Designing, creating, and maintaining such libraries is a very labor-intensive task.

We advance the theory of meta-programming in formal systems, helping automate entire developments in libraries. We focus on the very general case given by diagrams of formalizations consisting of theories and theory morphisms, where theories are lists of typed constants and theory morphisms are compositional translations between theories. We identify a class of meta-level operators on such diagrams that yields a framework for library developers to easily specify & implement operators. At the same time, these operators are still expressive enough to allow for a wide variety of applications in practice.

Concretely, we consider functorial operators that are defined declaration-wise on input theories and morphisms. This yields further valuable properties such as preservation of includes between theories. Thus, it is possible to apply them to entire diagrams at once, often in a way that the output diagram mimics any morphisms or modular structure of the input diagram.

Our results are worked out using the MMT language for structured theories instantiated with the Edinburgh Logical Framework LF for basic theories, but they can be easily transferred to other languages. We give numerous examples. Logic-independent functors are applicable fairly generally to many developments, and here we present the pushout functor, some refactoring-inspired operators, and a functor for representing certain meta theorems on theory morphisms (logical relations). We compose these operators in a culminating case study to get the powerful operator that systematically translates formalizations of type theory from intrinsic to extrinsic style. Among logic-dependent operators, we investigate operators for universal algebra automating universal constructions such as homomorphisms, substructures, and congruences, among others. Applied to an input diagram (e.g., some algebraic hierarchy), these operators yield diagrams of corresponding theories of homomorphisms, substructures, and congruences.

We implement our framework and all logic-independent operators in the MMT system, as part of a larger case study that was previously published by Rabe and Roux ("Systematic Translation of Formalizations of Type Theory from Intrinsic to Extrinsic Style", *Proceedings of the Workshop on Logical Frameworks: Meta-Theory and Practice* 2021).

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

Keywords and phrases meta programming,formal systems,theory graphs,formalizations,functors,structure preservation,mathematical knowledge management

Acknowledgements Submitted as a draft for personal reasons.

Submitted YYYY-MM-DD-TODO. Last updated 2022-03-23.

Contents

https://www.researchgate.net/publication/2384408_The_Tecton_Concept_Library#read

Notation	Usage
Σ	signatures, i.e., lists of declarations (“anonymous theories”)
σ	signature morphisms, i.e., lists of defined declarations
$\Delta_\Sigma(c\colon A\,[=t])$	a linear functor’s action on a constant c over signature S, Σ where S is s
$\delta_\Sigma(c\colon A)$	a linear connector’s action on a constant c over signature S, Σ where S i
Γ	contexts (usually over some designated signature, mostly Σ)
$\Gamma \vdash_\Sigma \ldots$	an LF judgement in signature Σ and context Γ
$\Gamma \vdash_\Sigma^M \ldots$	in principle same as $\Gamma \vdash_{M,\Sigma} \ldots$; connotation: Σ is an M-extension
f, g, h, m	morphisms
x	variables
t	terms (either MMT or `SFOL` ones)
A, B	LF types or kinds
c	identifiers of constants
$f\ t$	a term of function application
$\Pi\, x\colon A.\ B$	dependent function type
$\lambda x\colon A.\ B$	abstraction
id_T	identity morphism on theory T
$\sigma \circ \sigma'$, $f \circ g$, $m \circ n$	(signature) morphism composition (in which order?)
$\phi(x)$	LF term with one free variable, ϕ is a meta-level function
$\forall\, x\colon \mathtt{tm}\ T.\ \phi(x)$	$\forall\ T\ \lambda x\colon \mathtt{tm}\ A.\ \phi(x)$
$\exists$	analogously to forall...
X, Y	theory identifiers
v, w	morphism identifiers
n	theory and morphism identifiers (when we intend to make no distinctio
O	linear functors
$O(\Sigma), O(X), O(Y)$	linear functor O applied to flat theories Σ, X, Y
$O(\sigma)$	linear functor O applied to flat morphism σ
C	linear connectors
$\mathbb{LF}$	category of flat theories and morphisms, see Def. ...
$\mathbb{LF}^S$	for a structured theory S the category of flat S-extension and -extensio
`SFOL`	the `SFOL`MMT/LF-theory from ...
SFOL/DFOL/PFOL(-declarations/-theories)	the meta concept, not referring to any specific MMT/LF-theory,
type	either LF- or SFOL-type (usually clear from context)
`SFOL`-extension	all possible extensions of `SFOL` induced by MMT/LF (i.e., theories inclu
SFOL-/PFOL-/DFOL-/PDFOL-theories	a subset of certain well-patterned `SFOL`-extensions, see Definition TODO
$s \doteq_T,\ s \doteq t$	equality (type at which the equality is taken is usually omitted for read

make extra subtable for just sfol notations

1 Introduction

1.1 Motivation

There is wide variety of tools for representing and reasoning with formal knowledge on the computer [Naw+19]:

- automatic theorem provers that try to prove or refute statements (e.g., E [SCV19], Vampire [RV01], LEO-III [SB18])
- interactive theorem provers that let humans compose and verify proofs (e.g., Coq [BC04], Isabelle/HOL [NPW02], Agda [Nor09])
- specification systems that enable requirements analysis and system design on a high level (e.g., OBJ [Gog+93], CASL [Ast+02])
- deduction systems that allow defining logics and their proof calculi themselves (e.g., MMT [RK13], Twelf [PS], IMPS [FGT93], Specware [SJ95], Beluga [PD10])

Let us call such tools **formal systems** whenever they are based on a logical foundation. Formal systems have a number of uses. Primarily, they enable the verification of developments and proofs in formal sciences such as mathematics, logic, and computer science. Moreover, they serve the inevitable need to represent and categorize the ever-increasing amount of knowledge; a task that humans spectactularly fail at large scales [Car+21].

In practice, the utility of every formal system hinges on the **availability of a comprehensive standard library** of *theories* (i.e., concepts) and reasoning techniques. Standard libraries allow users to focus on representing their domain problem instead of spending time reinventing the wheel, e.g., by needing to formalize ubiquitous mathematical structures. Even trivial domain problems quickly **require standard libraries of huge sizes**. Already basic datastructures such as boolean, integers, lists, sets, trees, etc. including corresponding operations and reasoning schemes (e.g., filtering operations and induction) make up non-trivial portions of standard libraries. Another major source of ubiquitous theories is our **running example: the algebraic hierarchy** consisting of monoids, groups, rings, etc. And again, for those theories to be useful we almost surely need corresponding theories of homomorphisms, substructures, congruence structures, etc., too.

Importantly, there are **more metrics** than mere size of a standard library determining its quality (and the amount of manual labor needed to create and maintain it). Three quality metrics are particularly important for our endeavor. First, libraries need to feature a **consistent style**, e.g., in choice of identifiers, notations, argument orders, and packaging conventions. Only if these traits are consistently applied throughout a library, they become predictable for users, making it actually convenient for them to write formalizations without constantly looking up source code or documentation. In our running example, users would in particular expect all theories of the algebraic hierarchy to be packaged in the same way (e.g., into a theory, class, record, struct, etc. – whatever the language of choice offers). Second, libraries should be **feature complete**, i.e., all things that users might reasonably expect to be present should actually be present. For example, if the notion of homomorphisms is developed for the theory of groups, then by users' expectation that notion should also be available for monoids, rings, and all other theories of the algebraic hierarchy. Ideally, libraries fulfill this metric even recursively, e.g., if a library formalizes notions of quotient- and subgroups, then users might reasonably expect the notion of homomorphisms between quotiened subgroups, too. Third, a library should offer **rich interrelations**, so that users can transport knowledge or proof across theories. For example, libraries should offer converting every group homomorphism to a monoid homomorphism, and all theorems on monoid homomorphisms should be inherited by all group homomorphisms. Again, this property applies recursively, e.g., all homomorphisms on quotient groups should be convertible to homomorphisms of quotient monoids.

It is **very labor-intensive to design, create, and maintain a standard library** with those properties. Therefore, it is desirable to reduce the portion of the library that needs to be humanly written and maintained **in favor of parts that can be automatically generated**. A key observation motivating this research is that many theories (such as the universal constructions of homomorphisms and substructures) are inherently automatable in the sense that they do not require particular human intelligence to write down, but only require a particular systematic operation to be applied consistently everywhere.

1.2 Related Work

Metaprogramming As a means of automation and library design, meta programming has been used for decades in programming languages (see [LS19] for a recent survey). In contrast, in the formal systems community, the technique of meta programming has mainly been used for two independent purposes: **proof automation and self-verification**. Proof automation [Rin+19] is a standard technique in interactive theorem provers to outsource dull and repeating proof patterns into complex *tactics*, considerably cutting proof sizes and making proofs more robust wrt. changes in surrounding formalizations. As such, proof automation has substantially contributed to feasibility of fully formalizing big theorems (e.g., the Kepler conjecture [Hal+17], the Odd Order Theorem [Gon+], or the Four-Color Theorem [Gon08]). For Coq a number of tactic languages have been developed including Ltac [Del00; Péd19], Mtac [Kai+18], and SSReflect [GM10]. Meta programming facilities built into Lean [Ebn+17] and Agda [WS; Che20; KS15] have also been mainly used for proof automation.

For self-verification the syntax and semantics of a formal system are represented in itself, allowing the verification of meta theorems that would otherwise only be verifiable in other formal systems or on paper. The MetaCoq project [Soz+20] is following this avenue and has led to a correctness proof of Coq's typechecking in Coq itself [Soz+19]. There have been efforts in a similar direction for other systems, mostly based on dependent type theory, e.g., for Lean [Ebn+17] and Idris [CB16].

In contrast with these successes, both proof automation and self-verification paid relatively **little attention to use meta-programming to generate entire libraries**. One notable exception is Haskell – arguably closer to a programming language than formal system – offering Template Haskell [SP02] and the *deriving via* mechanism [BLS18], which can both be used to derive complete program fragments (e.g., function and datatype declarations).

Metaprogramming is not always needed. Sometimes we can circumvent it by using deep embeddings, however, at the cost of usability and conciseness of the resulting formalizations. For example, representing algebra theories with a deep embedding allows to give programmatic definitions of universal constructions without any metaprogramming and has been done in multiple settings, e.g., in Coq [Cap99; SW11] and Agda [DeM21; GGP18]. We give a detailed account of shallow vs. deep embeddings for universal algebra in Section 5.2.

Diagrams of Theories & Theory Morphisms

To be more concrete, let us from now on understand a *theory* as a structured list of declarations, and as theory interrelations let us use *theory morphisms*, i.e., compositional translations between theories. Diagrams of theories and theory morphisms have long been used successfully to build large networks of theories both in algebraic specification languages [**asl**; **specificationarbitrary**; **caslmanual**] and in deduction systems [**imps**; **isabelle_locales**; **RK:mmt:10**; SJ95], and are the right abstraction for general research of knowledge representation and processing [Koh14].

In parallel to the present work, [CFS20] also expands on the development of diagram operators. That work has a similar focus and even includes some of the same examples as ours. It focuses on programmatically generating many theories derived from the algebraic hierarchy, whereas we focus on structure preservation.

[HR20] introduces syntactic theory functors in the setting where theories are pairs of a signature and a set of axioms. Because signatures are kept abstract, the setting cannot be directly compared to ours, but their treatment of axioms corresponds to ours, and several of their concrete examples fit our framework. They also consider what we will call include-preservation but do not consider morphisms.

Following [RS19], while our diagrams are formalized in MMT, our diagram operators are implemented as self-contained objects in the programming language underlying MMT.

Our operators are defined within the framework for MMT diagram operators developed in [RS19]. We specialize our presentation to the LF logical framework [HHP93] as defined inside MMT. This choice is made for the sake of simplicity and concreteness, and our results can be easily transferred to other formal systems. More precisely, our results are applicable to any formal system whose syntax is described by a category of theories in which theories consist of lists of named declarations. In fact, MMT was introduced specifically as an abstract definition capturing exactly those formal systems [Rab17a]. Moreover, as LF is a logical framework designed specifically for representing the syntax and proof theory of formal systems, even the restriction to MMT/LF subsumes diagram operators for many important logics and type theories (see [Cod+b] for examples). Our implementation is already applicable to any framework defined in MMT with LF as used here being just one example (see [MR19] for others).

1.3 Challenges & Objectives

Meta-programming is most powerful whenever meta-programs are allowed to operate on abstract syntax trees. And this presents our main challenge, which is to combine two conflicting goals. On the one hand, it must be **easy to define and implement** new meta-programs acting on formalizations. This is critical for scalability as we conjecture many meta-programs to be contributed by library developers (and not developers of the formal system itself), who may not be perfectly familiar with the overall framework and therefore need an interface as easy as possible. Moreover, it is critical for correctness: meta-programs tend to be very difficult to correctly specify and implement. On the other hand, meta-programs shine when applied to large formalizations, and **large formalizations are inevitably built with complex language features**. This applies both to the base logic, e.g., adding nodes to diagrams that require more expressive logics than originally envisioned, and to the structuring features for building larger theories out of smaller ones.

We want to develop **a general framework for specifying and implementing meta-programs** that work on structured collections of formalizations. Concretely, our framework must meet the following objectives:

i) Library Users should be able to apply translations

  a) to structured diagrams with complex language features
  b) to build large and interrelated structured diagrams

   c) with little manual effort in writing and maintaining formalization code
   d) with intuitive and predictable outcomes in structure of diagrams and contents

ii) Library Developers should be able to easily specify, verify, and implement translations as meta-programs

iii) System Developers should be able to efficiently apply those meta-programs to large diagrams and cache their results

1.4 Contribution

We identify a class of *diagram operators*, for which these conflicting objectives can largely be achieved. The general design uses two steps. First, our diagram operators are **defined and implemented for flat theories** only. In particular, complex language features remain transparent to library developers specifying and implementing diagram operators. Second, every such definition is automatically **lifted to an operator on structured diagrams**. This lifting preserves diagram structures, and in particular it maps hierarchies of theories built out of includes to analogous ones. We give an account of corresponding the meta theory and, e.g., state verification criteria that are easy to apply. Overall, we work with the MMT language [RK13] of theories and theory morphisms, providing us a fairly general setting. This makes our contributions applicable to many declarative and typed languages, incl. many logics, type theories, and set theories.

Regarding structuring features, we limit attention to the three simplest and arguably most important features: definitions, inclusions, and morphisms. This choice is made for brevity, and our results generalize to the more complex structuring features supported by MMT such as qualified inclusions.

We present several important operators as exemplary uses of our framework. First, we consider **logic-independent operators**, i.e., operators that are applicable to formalizations over very weak logics, thus are rather domain-agnostic and of widespread use. As a canonical example, we show the pushout operator which allows to apply many logic translations occurring in practice to entire diagrams. For example, assuming a theory morphism from some formalized type theory to some variant of set theory, entire developments over that type theory, i.e., diagrams, can be translated to set theory using the pushout operator. Moreover, we consider proofs by logical relations (i.e., certain complex meta-theorems) and give an operator to internalize them. Thus, we also use our framework to develop operators that extend the underlying formal system with new features. We consider two refactoring-inspired operators: one that abstracts every theory declaration over a new type, and one that removes parameters of function declarations. While simple in nature, these operators occur widely and can be composed with other operators to realize powerful operators: as a culminating case study, as a composition of previous logic-independent operators we realize the operator that systematically translates formalizations of type theory from intrinsic to extrinsic style (following [RR21b]).

Second, as logic-dependent operators we consider the large example base of **operators for universal algebra**. We phrase common universal constructions such as homomorphisms, substructures, and congruence structures as operators in our framework. Thus, when applied to a diagram D consisting of algebra theories, these operators yield corresponding diagrams $\texttt{Hom}(D)$, $\texttt{Sub}(D)$, and $\texttt{Cong}(D)$. For example, $\texttt{Hom}(D)$ contains a theory $\texttt{Hom}(T)$ formalizing the theory of homomorphisms between T-models for every theory T in D. Even though the underlying constructions are well-known, the level of formality and generality in our setting forces us to invest considerable effort and novelty to generalize them.

We have contributed an implementation of our framework and all logic-independent operators to the MMT system [MMTb]. We touch on and refer to an extensive case study [RR21b] for the logic-independent operators conducted, to the LATIN atlas [LATIN2] of modular formalizations of formal systems and logics [Cod+a]. (The implementation in the case study has been solely developed by the author.)

Overview As preliminaries in Section 2, we recap MMT and logical relations for a logical framework. In Section 3 we motivate and develop the heart of this thesis: a framework of structure-preserving diagram operators. We then proceed to give examples of logic-independent operators in Section 4. Concretely, we give pushout in Section 4.1, the refactoring-inspired operators to abstract declarations and remove parameters in Sections 4.2 and 4.3, and finally the operator to represent logical relations in Section 4.4. We present our culminating case study of logic-independent operators in Section 4.5. In Section 5 we present our logic-dependent operators for universal algebra with an introduction in Section 5.1, an elaborate account of approaches for representing algebra theories to begin with in Section 5.2, the actual operators in Sections 5.3–5.7, and a conclusion in Section 5.8. We showcase our implementation and critical design decisions in Section 6. Finally, we conclude and point to future work in Section 7.

1.5 Incorporated Material

Large parts of the present thesis are based on previous material by the author and, sometimes, also their thesis advisor Florian Rabe. In every case, the material has been substantially (co-)authored by the author. With explicit permission from Florian Rabe, some text parts of this thesis are even based verbatim on that material. But more often than not, the author has refined those parts to match the surrounding thesis. The below table compiles that material and lists influenced sections.

Citation	Form	Title	Authors	Sections
[RR21c]	conference paper (published)	*Structure-Preserving Diagram Operators*	Navid Roux, Florian Rabe	1–3, 4.1, 4.2, 5.3, and 5.4, and abstract
[RR21b]	conference paper (published)	*Systematic Translation of Formalizations of Type Theory from Intrinsic to Extrinsic Style*	Florian Rabe, Navid Roux	2.2 and 4.3–4.5
[RR20]	extended abstract (accepted) & talk	*Diagram Operators in a Logical Framework*	Navid Roux, Florian Rabe	4.5
[RR21a]	conference paper (unpublished)	*Modular Formalization of Formal Systems*	Florian Rabe, Navid Roux	4.3 and 4.4
[Rou21a]	seminar paper for Master's studies	*A Beginner's Guide to Logical Relations for a Logical Framework*	Navid Roux	2.2 and 4.4
[Rou21b]	slides and talk for [Rou21a]	(ditto)	Navid Roux	(ditto)
[Rou20]	Master's studies project	*Structure-Preserving Diagram Operators*	Navid Roux	4.2.3 and all mentioned for [RR21c]

2 Preliminaries

2.1 MMT: A Module System over a Logical Framework

The logical framework LF [HHP93] is a dependent type theory designed for defining a wide variety of formal systems including many variants of first- and higher-order logic and set and type theory [Cod+a]. We work with LF as realized in MMT *language* [Rab17a], which induces a language of structured theories for any such formal system defined in LF. MMT uses structured theories and theory morphisms akin to algebraic specification languages like OBJ [Gog+93] and CASL [Ast+02] We contribute our theoretical results to the MMT *system* [Rabb][1], the reference implementation of the MMT language.

2.1.1 Flat MMT/LF

We first discuss a reduced version of MMT/LF and its semantics. The grammar for **flat theories and morphisms** is given below.[2] Later, we extend the grammar with the exemplary structuring features of includes in theories and morphisms. The very goal of this thesis is to provide a framework that allows specifying certain meta programs on the flat grammar and automatically lifts them to the structured grammar.

Σ	$::=$	$\{(c\colon A\,[= t])^*\}$	flat theories
σ	$::=$	$\Sigma \to \Sigma' = \{(c\colon A' = t')^*\}$	flat morphisms
f, t, A, B	$::=$	$\texttt{type} \mid \texttt{kind} \mid c \mid x \mid f\ t \mid$ $\Pi\, x\colon A.\ B \mid \lambda x\colon A.\ t$	terms

The **flat theories** are inspired by LF-style languages and are anonymous lists of typed/kinded constant declarations $c\colon A\,[= t]$ where A is the type given to c and t an optional definiens of type A. In a constant declaration, if t is given, the whole declaration of c acts as an abbreviation. As usual, we write $A \to A$ for $\Pi\, x\colon A.\ A$ when x does not occur freely in A. Constant declarations subsume many variants of type, function, predicate, function, and axiom symbols.

Correspondingly, a **flat morphism** $\sigma\colon \Sigma \to \Sigma'$ specifies a compositional translation between two flat theories Σ and Σ' by stating an anonymous list of assignments $c\colon A' = t'$. Here, t' is a Σ'-expression of type $\overline{\sigma}(A)$ ($\overline{\sigma}$ being the homomorphic extension of σ), and A' is a Σ'-expression that is Σ'-equal to $\overline{\sigma}(A)$. These conditions guarantee that the homomorphic extension induced by every flat morphism σ is a compositional translation from Σ- to Σ'-syntax that preserves all judgements, in particular typing and equality. For example, if t is an arbitrary, possibly complex, Σ-term of type A, then $\overline{\sigma}(t)$ is a Σ'-term of type $\overline{\sigma}(A)$. We denote the identity morphism on a flat theory Σ by id_Σ.

Both our choices to allow definitions in theories and to make explicit type annotations in morphism assignments are non-standard within the LF community. In particular, we could omit types in morphism assignments since they can be inferred anyway. However, these choices enable a unified treatment of constants and assignments when specifying diagram operators. In fact, under mild assumptions they must treat defined constants in the same

[1] Source code at https://github.com/UniFormal/MMT, documentation at https://uniformal.github.io/doc/

[2] In the literature, flat theories and morphisms are somtimes referred to as *signatures* and *signature morphisms*.

way as assignments anyway. Thus, having a unified syntax for constants and assignments reduces clutter when stating operators and meta theorems, and implementing operators[3].

We identify terms up to α-equivalence and will repeatedly assume freshness conditions without loss of generality.

The following examples are all folklore (e.g., see [HHP93]) with nuances inspired by [LATIN2], a larger archive of MMT formalizations.

▶ **Example 1** (Propositional Logic in MMT/LF)**.** Below we give the flat theory PL formalizing a contrived subset of the syntax of propositional logic.

$$\mathtt{PL} = \left\{ \begin{array}{ll} \mathtt{prop} \colon \mathtt{type} & \\ \wedge & \colon \mathtt{prop} \to \mathtt{prop} \\ \vee & \colon \mathtt{prop} \to \mathtt{prop} \to \mathtt{prop} \\ \neg & \colon \mathtt{prop} \to \mathtt{prop} \end{array} \right\}$$

Concrete theories of propositional logic, i.e., where domain-specific atoms are available, can be represented as PL-extensions as $\Sigma = \mathtt{PL}, \{A_1 \colon \mathtt{prop}, \ldots, A_n \colon \mathtt{prop}\}$, where we use the comma to denote concatenation of flat theories.

In practice, the MMT system allows to associate complex notations [Koh+09] with every constant, e.g., including implicit arguments that are to be inferred when using that constant. For brevity we skip introducing notations formally and simply use them informally. For example, we casually write $A_1 \wedge A_2$ to mean $\wedge\ A_1\ A_2$.

▶ **Example 2** (Sorted First-Order Logic in MMT/LF)**.** In Figure 1 we give a flat theory SFOL defining sorted first-order logic (SFOL). The constants prop and tp are the LF-types holding the TFOL-propositions and TFOL-types, respectively. For any TFOL-type $A \colon \mathtt{tp}$, the LF-type $\mathtt{tm}\ A$ holds the TFOL-terms of TFOL-type A. Following the judgments-as-types paradigm, the validity of a proposition $F \colon \mathtt{prop}$ is captured by the non-emptiness of the type $\vdash F$, which holds the proofs of F. We only give few connectives and proof rules as examples. Again, we implicitly use casual notation, e.g., write $\forall\ p$ to mean $\forall\ T\ p$ for $p \colon \mathtt{tm}\ T \to \mathtt{prop}$ from context. Moreover, we use notation like $\forall\, x \colon \mathtt{tm}\ T.\ p\ x$ to mean...

SFOL-theories can now be represented as SFOL-extensions (i.e., flat LF-theories that begin with SFOL), and similarly SFOL-theory morphisms as $\mathrm{id}_{\mathtt{SFOL}}$-extensions (i.e., flat LF-morphisms that begin with the identity morphism of SFOL). More precisely, the image of this representation contains theories that begin with SFOL and then only add declarations of certain shapes, namely $T \colon \mathtt{tp}$ for a type symbol, $f \colon \mathtt{tm}\ T_1 \to \ldots \to \mathtt{tm}\ T_n \to \mathtt{tm}\ T$ for a function symbol, $p \colon \mathtt{tm}\ T_1 \to \ldots \to \mathtt{tm}\ T_n \to \mathtt{propprop}$ for a predicate symbol, and $a \colon \Vdash F$ for an axiom symbol. More generally, we can allow more patterns to achieve a representation of polymorphic and dependently-typed SFOL. We will come back to that in Section 5.2.2.

Note that in the structured grammar of MMT, SFOL could have been stated to include PL, thus avoiding a lot of overhead. We avoided doing so because we have not yet introduced the structured grammar.

▶ **Example 3** (Specifying Algebra Theories as SFOL-Extensions)**.** We specify the algebra theory

[3] Moreover, the MMT system has been successfully using this unified treatment internally for years, too.

$$
\mathtt{SFOL} = \left\{
\begin{array}{ll}
\mathtt{prop} & : \mathtt{type} \\
\Vdash & : \mathtt{prop} \to \mathtt{type} \\
\mathtt{tp} & : \mathtt{type} \\
\mathtt{tm} & : \mathtt{tp} \to \mathtt{type} \\
\multicolumn{2}{l}{\text{/* propositional connectives \& proof rules */}} \\
\neg & : \mathtt{prop} \to \mathtt{prop} \\
\wedge, \vee, \Rightarrow & : \mathtt{prop} \to \mathtt{prop} \to \mathtt{prop} \\
\multicolumn{2}{l}{\text{/* example of a derived connective */}} \\
\Leftrightarrow & : \mathtt{prop} \to \mathtt{prop} \to \mathtt{prop} \\
 & = \lambda F\, G.\ (F \Rightarrow G) \wedge (G \Rightarrow F) \\
\wedge\mathtt{I} & : \Pi\, F\, G \colon \mathtt{prop}.\ \Vdash F \to \Vdash G \to \Vdash F \wedge G \\
\wedge\mathtt{EL} & : \Pi\, F\, G \colon \mathtt{prop}.\ \Vdash F \wedge G \to \Vdash F \\
\wedge\mathtt{ER} & : \Pi\, F\, G \colon \mathtt{prop}.\ \Vdash F \wedge G \to \Vdash G \\
\vee\mathtt{IL} & : \Pi\, F\, G \colon \mathtt{prop}.\ \Vdash F \to \Vdash F \vee G \\
\vee\mathtt{IR} & : \Pi\, F\, G \colon \mathtt{prop}.\ \Vdash G \to \Vdash F \vee G \\
\vee\mathtt{E} & : \Pi\, F\, G \colon \mathtt{prop}.\ \Vdash F \vee G \to \Pi\, H \colon \mathtt{prop}. \\
 & \qquad (\Vdash F \to \Vdash H) \to (\Vdash G \to \Vdash H) \to \Vdash H \\
\multicolumn{2}{l}{\text{/* equality \& proof rules */}} \\
\doteq & : \Pi\, T \colon \mathtt{tp}.\ \mathtt{tm}\ T \to \mathtt{tm}\ T \to \mathtt{tm}\ T \\
\mathtt{refl} & : \Pi\, T \colon \mathtt{tp}.\ \Pi\, t \colon \mathtt{tm}\ T.\ \Vdash t \doteq t \\
\mathtt{symm} & : \Pi\, T \colon \mathtt{tp}.\ \Pi\, t_1\ t_2 \colon \mathtt{tm}\ T.\ \Vdash t_1 \doteq t_2 \to \Vdash t_2 \doteq t_1 \\
\mathtt{trans} & : \Pi\, T \colon \mathtt{tp}.\ \Pi\, t_1\ t_2\ t_3 \colon \mathtt{tm}\ T.\ \Vdash t \doteq t_2 \to \Vdash t_2 \doteq t_3 \to \Vdash t_1 \doteq t_3 \\
\multicolumn{2}{l}{\text{/* quantifiers \& proof rules */}} \\
\forall, \exists & : \Pi\, T \colon \mathtt{tp}.\ (\mathtt{tm}\ T \to \mathtt{prop}) \to \mathtt{prop} \\
\forall\mathtt{I} & : \Pi\, T \colon \mathtt{tp}.\ \Pi\, p \colon \mathtt{tm}\ T \to \mathtt{prop}.\ (\Pi\, x \colon \mathtt{tm}\ T.\ \Vdash p\ x) \to \Vdash \forall p \\
\forall\mathtt{E} & : \Pi\, T \colon \mathtt{tp}.\ \Pi\, p \colon \mathtt{tm}\ T \to \mathtt{prop}.\ \Vdash \forall p \to \Pi\, x \colon \mathtt{tm}\ T.\ \Vdash p\ x \\
\exists\mathtt{I} & : \Pi\, T \colon \mathtt{tp}.\ \Pi\, p \colon \mathtt{tm}\ T \to \mathtt{prop}.\ \Pi\, x \colon \mathtt{tm}\ T.\ \Vdash p\ x \to \exists p \\
\exists\mathtt{E} & : \Pi\, T \colon \mathtt{tp}.\ \Pi\, p \colon \mathtt{tm}\ T \to \mathtt{prop}.\ \Vdash \exists p \to \Pi\, F \colon \mathtt{prop}. \\
 & \qquad (\Pi\, x \colon \mathtt{tm}\ T.\ \Vdash p\ x \to \Vdash F) \to \Vdash F
\end{array}
\right\}
$$

Figure 1 Formalization of Sorted First-Order Logic

of monoids as an SFOL-extension:

$$\mathtt{Monoid} = \mathtt{SFOL}, \left\{ \begin{array}{ll} U & : \mathtt{tp} \\ \circ & : \mathtt{tm}\ U \to \mathtt{tm}\ U \to \mathtt{tm}\ U \\ e & : \mathtt{tm}\ U \\ \mathtt{assoc} & : \Vdash \forall\, x\ y\ z \colon \mathtt{tm}\ U.\ (x \circ y) \circ z \doteq x \circ (y \circ z) \\ \mathtt{neut} & : \Vdash \forall\, x \colon \mathtt{tm}\ U.\ e \circ x \doteq x \end{array} \right\}$$

Judgements Above we already alluded to certain well-typedness conditions. We now introduce them a bit more formally as sufficient for our purposes (e.g., proving well-typedness of diagram operators). Below we list the most important judgements and inference rules.

$\mathrm{sig}\ \Sigma$	(Σ is a valid flat theory)
$\mathrm{mor}\ \sigma \colon\ \Sigma \to \Sigma'$	(σ is a valid flat morphism $\Sigma \to \Sigma'$)
$\vdash_\Sigma c_1 \colon A_1\,[= t_1], \ldots, c_n \colon A_n\,[= t_n]$	(abbr. for: $\Sigma, c_1 \colon A_1\,[= t_1], \ldots, c_n \colon A_n\,[= t_n]$ is a valid flat theory)
$\vdash_\sigma c_1 \colon A_1\ = t_1, \ldots, c_n \colon A_n\ = t_n$	(abbr. for: $\sigma, c_1 \colon A_1\ = t_1, \ldots, c_n \colon A_n\ = t_n$ is a valid flat morphism)
$\vdash_\Sigma \Gamma$	(Γ is a valid context in Σ)
$\Gamma \vdash_\Sigma t \colon A$	(t has type A in Σ, Γ)
$\Gamma \vdash_\Sigma t \equiv t'$	(t and t' are equal in Σ, Γ)

$$\frac{}{\mathrm{sig}\ \cdot} \qquad \frac{\mathrm{sig}\ \Sigma \qquad c \notin \Sigma \qquad (A \colon \mathtt{type}\ \text{or}\ A \colon \mathtt{kind}) \qquad [t \colon A]}{\mathrm{sig}\ \Sigma, c \colon A\,[= t]} \qquad \frac{\mathrm{sig}\ \Sigma'}{\mathrm{mor}\ \cdot \colon\ \cdot \to \Sigma'}$$

$$\frac{\mathrm{mor}\ \sigma \colon\ \Sigma \to \Sigma' \qquad \mathrm{sig}\ \Sigma, c \colon A\,[= t] \qquad \vdash_{\Sigma'} t' \colon A' \qquad \vdash_{\Sigma'} A' \equiv \sigma(A) \qquad [\vdash_{\Sigma'} t' \equiv \sigma(t)]}{\mathrm{mor}\ \sigma, c \colon A'\ = t' \colon\ \Sigma, c \colon A \to \Sigma'}$$

$$\frac{\vdash_\Sigma \Gamma \qquad x \notin \Gamma \qquad A \colon \mathtt{type}}{\vdash_\Sigma \Gamma, x \colon A} \qquad \frac{\mathrm{sig}\ \Sigma \qquad \vdash_\Sigma \Gamma \qquad (c \colon A\,[= t]) \in \Sigma}{\Gamma \vdash_\Sigma c \colon A}$$

We refer to [HHP93, Tables 1 – 3] for details. In particular, in few proofs in the present thesis, we will need to case-analyze on all inference rules and thus will make transparent use of those listed in these tables. Concerning the interplay of judgements with MMT, we refer to [Rab17a] for details.

▶ **Definition 4** (Homomorphic Extension). *Let $\sigma \colon \Sigma \to \Sigma'$ be a signature morphism. The homomorphic extension $\overline{\sigma}$ is given by*

$$\overline{\sigma}(c) = t' \ \textit{if}\ (c \colon A'\ = t') \in \sigma$$

$$\overline{\sigma}(\mathtt{type}) = \mathtt{type} \qquad \overline{\sigma}(\mathtt{kind}) = \mathtt{kind}$$
$$\overline{\sigma}(x) = x \qquad \sigma(f\ t) = \overline{\sigma}(f)\ \overline{\sigma}(t)$$
$$\overline{\sigma}(\Pi\, x \colon A.\ B) = \Pi\, x \colon \overline{\sigma}(A).\ \overline{\sigma}(B) \qquad \overline{\sigma}(\lambda x \colon A.\ t) = \lambda x \colon \overline{\sigma}(A).\ \overline{\sigma}(t)$$

In the following, we simply write σ for $\overline{\sigma}$.

▶ **Theorem 5** (Morphisms Preserve All Judgements). *For every flat morphism $\sigma \colon \Sigma \to \Sigma'$ we have*

$$\vdash_\Sigma t \colon A \implies \vdash_{\Sigma'} \sigma(t) \colon \sigma(A)$$
$$\vdash_\Sigma t \equiv t' \implies \vdash_{\Sigma'} \sigma(t) \equiv \sigma(t')$$

Proof. By induction, see [Rab17a]. ◀

Unless otherwise noted, we assume well-typedness of all (flat) theories and morphisms mentioned throughout the whole thesis.

Semantics We very briefly give a categorical semantics to flat theories and morphisms, and we refer to [Rab17a] for details.

▶ **Definition 6** (Category of Theories). *By $\mathbb{LF}$ we denote the category of flat theories and morphisms where we identify morphisms $\sigma_1, \sigma_2 \colon \Sigma \to \Sigma'$ if for all $c \in \Sigma$ and respective assignments $(c \colon A = t) \in \sigma$ and $(c \colon A' = t') \in \sigma'$ we have $\vdash_{\Sigma'} A = A'$ and $\vdash_{\Sigma'} t \equiv t'$.*

Morphism composition is denoted $\sigma' \circ \sigma$ and given by $(\sigma' \circ \sigma)(c) = \sigma'(\sigma(c))$.

When beneficial for understanding, we in particular call morphisms $m \colon S \to T$ realizations of S (thinking of S as an interface theory).

▶ **Lemma 7** (Criterion for Morphism Equality). *Two morphisms $\sigma_1, \sigma_2 \colon \Sigma \to \Sigma'$ in $\mathbb{LF}$ are equal if for all constants $c \in \Sigma$ we have $\vdash_{\Sigma'} \sigma_1(c) = \sigma_2(c)$.*

Proof. For constants $c \in \Sigma$ and assignments $(c \colon A_1 = t_1) \in \sigma_1$ and $(c \colon A_2 = t_2)$ we need to check $A_1 = A_2$ and $t_1 = t_2$ over Σ'. The latter follows by assumption. And the former follows by a meta theorem of LF that whenever two terms are equal, then their types are equal, too. ◀

2.1.2 Structured Mmt/LF

We show the grammar of structured Mmt/LF in Figure 2. There and in the sequel, we use S, T for theory identifiers, v, w for morphism identifiers, and c and x for constant and variable identifiers, respectively. The underlined parts in the grammar in Figure 2 are the structuring principles that LF inherits from Mmt and that give rise to **structured theories and morphisms**. For simplicity, we restrict attention to the two most important structuring principles: named theories/morphisms and includes. **Include** declarations allow combining theories into theories and morphisms into morphisms. We omit defining well-typedness for structured theories, morphisms, and diagrams and instead refer to [RK13, Sec. 6.3]. When clear from context, we simply say *theory* or *morphism* to refer to either the flat or the structured concept (or both when irrelevant).

The Mmt system provides further structuring features, in particular for translating theories and renaming constants during an include. Some of these features have emerged from practice, and are yet experimental. Their design, syntax, semantics, and their idiomatic usage patterns are all subject to constant change, gauging what works best in practice for developers and users of the Mmt system. In contrast to the case of flat theories, formal specifications of more complex structuring features are often lengthy to write down in a self-contained way (often lengthier than to implement). Thus, in our presentation we limit ourselves to those structuring features presented in Figure 2 even though in our implementation we support the more advanced ones, too.

Semantics Central to our framework of diagram operators will be endofunctors on $\mathbb{LF}$. Often, such operators will be logic-specific, i.e., they exclusively translate flat theories and morphisms relative to some base such as `SFOL` and $\text{id}_{\texttt{SFOL}}$, respectively. Hence we define:

▶ **Definition 8** (Category of Theory Extensions). *Consider a structured theory S. By $\mathbb{LF}^S$ we denote the category of flat S-extensions and flat S-extension morphisms, i.e., objects are*

Diag	$::=$	$(\mathit{Thy} \mid \mathit{Mor} \mid \textbf{install } D)^*$	diagrams
Thy	$::=$	$\underline{\textbf{theory } T} = \{\mathit{Decl}^*\}$	theory definition
Decl	$::=$	$c\colon A\,[= t] \mid \underline{\textbf{include } T}$	declarations in a theory
Mor	$::=$	$\underline{\textbf{mor } v\colon S \to T} = \{\mathit{Ass}^*\}$	morphism definition
Ass	$::=$	$c\colon A\ = t \mid \underline{\textbf{include } v}$	assignments in a morphism
f, t, A, B	$::=$	$\texttt{type} \mid \texttt{kind} \mid c \mid x \mid f\ t \mid$ $\Pi\, x\colon A.\ B \mid \lambda x\colon A.\ t$ $\lambda x\colon A.\ t \mid \Pi\, x\colon A.\ B$	terms
D	$::=$	$\mathrm{Diagram}(T^*, v^*) \mid O(D)$	diagram expressions

Figure 2 MMT/LF Grammar

theories of the form S, Σ for flat Σ and morphisms are of the form id_S, σ for flat σ. We apply the same identification to morphisms as done in Definition 6.

When clear from context, we often refer to an object S, Σ and morphism id_S, σ simply by Σ and σ, respectively.

The semantics of MMT structuring features is given by the *flattening* operation $-^\flat$ that transforms structured theories and morphisms into flat ones as sketched in Figure 3. Using flattening, we can define the faithful functor $\iota_S\colon \mathbb{LF}^S \to \mathbb{LF}$ that translates theories S, Σ to $S^\flat, \Sigma$ and morphisms id_S, σ to $\mathrm{id}_{S^\flat}, \sigma$.

We say a flat theory S is **included** into a flat theory T if $S \subseteq T$, and a structured theory S is **included** into a structured theory T if **include** S occurs in the body of T. Both cases induce a canonical **inclusion morphism** and we write $S \hookrightarrow T$. Thus, inclusion is a transitive relation on theories.

$$\begin{aligned}
S^\flat &= \Sigma^\flat \quad \text{if } S = \{\Sigma\} & v^\flat &= \sigma^\flat \quad \text{if } v\colon S \to T = \{\sigma\}\\
\cdot^\flat &= \varnothing & \cdot^\flat &= \varnothing\\
(c\colon A\,[= t], \Sigma)^\flat &= c\colon A\,[= t], \Sigma^\flat & (c\colon A\ = t, \sigma)^\flat &= c\colon A\ = t, \sigma^\flat\\
(\textbf{include } T, \Sigma)^\flat &= T^\flat, \Sigma^\flat & (\textbf{include } w, \sigma)^\flat &= w^\flat, \sigma^\flat.
\end{aligned}$$

Figure 3 Flattening of structured theories/morphisms

Diagram expressions D were introduced in [RS19]: they are used as the third toplevel declaration **install** D in Figure 2, whose semantics is to declare all theories and morphisms in D at once. For our purposes, it is sufficient to consider only two simple cases for D: $\mathrm{Diagram}(T^*, v^*)$ builds an anonymous diagram by aggregating some previously defined theories and morphisms. And $O(D)$ applies a diagram operator O to the diagram D. Here, O is simply an identifier that MMT binds to a user-provided computation rule implemented in the underlying programming language.

▶ Remark 9. The exact nature of diagram expressions and the install declaration in MMT is still somewhat of an open question. Here, we follow [RS19] and add them to the formal syntax of the language. Alternatively, we could relegate all diagram expressions to the meta-level. For example, a preprocessor could be used to compute $O(D)$ and generate the theory and morphism declarations in it.

One indication against the latter is that we have already identified some operators that take more arguments than just a diagram. For example, pushout takes a morphism m and returns the diagram operator $O = P_m$. Such operators can benefit from a tight integration with the type checker.

Further work with larger case studies is necessary to identify the most convenient syntax and work flow for utilizing diagram operators. However, large case studies are best done with structured diagrams, which is why we have prioritized the present work. In any case, the work presented here is independent of how that question is answered, and the current implementation can be easily adapted to other work flows.

Limitations In order to pin down the **limitations** of our work more precisely, we introduce the following definition: A formal system is called **straight** if

- its theories consist of lists of declarations $c\colon A$,
- the well-formedness of theories is checked declaration-wise, i.e., $\Sigma, c\colon A$ is a well-formed theory if c is a fresh name and A satisfies some well-formedness condition $\mathrm{WF}_\Sigma(A)$ relative to Σ, and
- $\mathrm{WF}_\Sigma(A)$ depends only on those declarations in Σ that can be reached by following the occurs-in relation between declarations.

Thus, in a straight language, $\mathrm{WF}_\Sigma(A)$ can be reduced to $\mathrm{WF}_{\Sigma_0}(A)$ where Σ_0 consists of the set of declarations in Σ whose names transitively occur in A. We call Σ_0 the **dependencies** of c. In a straight language, it is easy and harmless to **identify theories up to reordering of declarations**, and we will do so in the sequel.

Straightness is a very natural condition and is satisfied by LF and thus any formal system defined in it. However, there are practically relevant counterexamples. Most of those use proof obligations as part of defining $\mathrm{WF}_\Sigma(A)$ and discharging them may have to make use of all declarations in Σ. Typical examples are languages with partial functions, subtyping, or soft typing. (Many of these can be defined in LF as well, but only by enforcing straightness at the cost of simplicity.) Moreover, any kind of backward references such as in mutually recursive declarations violates straightness. We expect that our results can be generalized to such languages, but we have not investigated that question yet.

2.2 Logical Relations for a Logical Framework

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, including logical frameworks and MMT [RS13]. The instantiation for MMT allows fully representing, and thus mechanically verifying, proofs by logical relations in MMT.

Importantly, we not only make use of logical relations *for proof*, but also of logical relations *for data* and use them as a **tool to specify complex translations** on MMT/LF terms. In particular operators from universal algebra presented in Section 5 are prone to necessitate such complex translations. We only collect main definitions and theorems below. For an introductory guide to logical relations with a special focus on their instantiation in MMT, we refer to the guide [Rou21a] by the author. It is based on lectures by Amal Ahmed [Ahm13], for which typeset notes can be found at [Sko19]. For readers familiar with logical relations, we refer to [RS13], which introduced them first for MMT.

From now on by, when exclusively refer to *logical relation* in the sense of [RS13] for the MMT language. Consider some morphisms $m_1, \dots, m_n\colon R \to S$ between two theories S and T.

$$\begin{array}{ll}
\overline{r}(c) & = r(c) \\
\overline{r}(x) & = \begin{cases} x^* & \text{if } x^* \text{ was declared when traversing into the binder of } x \\ \text{undefined} & \text{otherwise} \end{cases} \\
\overline{r}(\mathtt{type}) & = \lambda\, a_1 \colon \mathtt{type}.\ \ldots\ \lambda\, a_n \colon \mathtt{type}.\ \ m'_1(a_1) \to \ldots \to m'_n(a_n) \to \mathtt{type} \\
\overline{r}(\Pi\, x \colon A.\ B) & = \Pi\, f_1 \colon m'_1(\Pi\, x \colon A.\ B).\ \ \ldots\ \Pi\, f_n \colon m'_n(\Pi\, x \colon A.\ B). \\
 & \quad \Pi\, \overline{r}(x \colon A).\ \overline{r}(B)\ (f_1\ x_1)\ \ldots\ (f_n\ x_n) \\
\overline{r}(\lambda x \colon A.\ t) & = \lambda \overline{r}(x \colon A).\ \overline{r}(t) \\
\overline{r}(f\ t) & = \begin{cases} \overline{r}(f)\ m'_1(t)\ \ldots\ m'_n(t)\ \overline{r}(t) & \text{if } \overline{r}(t) \text{ defined} \\ \overline{r}(f)\ m'_1(t)\ \ldots\ m'_n(t) & \text{otherwise} \end{cases} \\
\overline{r}(\cdot) & = \cdot \\
\overline{r}(\Gamma, x \colon A) & = \overline{r}(\Gamma), \begin{cases} x_1 \colon m'_1(A), \ldots, x_n \colon m'_n(A), x^* \colon \overline{r}(A)\ x_1\ \ldots\ x_n & \text{if } \overline{r}(A) \text{ defined} \\ x_1 \colon m'_1(A), \ldots, x_n \colon m'_n(A) & \text{otherwise} \end{cases}
\end{array}$$

where $\overline{r}(-)$ is undefined whenever an expression of the right-hand side is, and where the functions m'_i are given by $m'_i(t) = m_i(t)[x \mapsto x^{(i)}]$ (i.e., m_i postcomposed by a substitution).

Figure 4 Map induced by a Logical Relation (generalizing [RR21b, Fig. 4])

A logical relation r is an object on $m_1, \ldots, m_n$, and more concretely, a function from R-syntax to S-syntax specified by assignments of the form $r(c) = t$ for every $c \in R$ subject to certain typing conditions. For every type constant $\vdash_R T \colon \mathtt{type}$, we require $\vdash_S r(T) \colon m_1(T) \to \ldots \to m_n(T) \to \mathtt{type}$, i.e., $r(T)$ must be an n-ary LF relation on the types $m_1(T), \ldots, m_n(T)$. And for every corresponding term constant $\vdash_S t \colon T$, we require $\vdash_S r(t) \colon r(T)\ m_1(t)\ \ldots\ m_n(t)$, i.e., $r(t)$ must be a proof of $m_1(t), \ldots, m_n(t)$ being in the relation $r(T)$ prescribed by r at t's type. The following definitions makes things precise and lifts these intuitions to all terms. In our presentation, we state for the first time the combination of two orthogonal flavors of logical relations: flexarity from [RS13] and partiality from [RR21b].

▶ **Definition 10** (Partial Logical Relations (generalizing [RS13, Def. 3.5] and [RR21b, Def. 2]))**.** *A **partial logical relation** on morphisms $m_1, \ldots, m_n \colon R \to S$ is a partial mapping of R-constants to S-expressions such that for every R-constant $c \colon A$, if $r(c)$ is defined, then so is $\overline{r}(A)$ and $\vdash_S r(c) \colon \overline{r}(A)\ m_1(c)\ \ldots\ m_n(c)$. The partial mapping $\overline{r}$ of R-syntax to S-syntax is defined in Figure 4.*

*We call r **term-total** if it is defined for a typed constant if it is for the type. And we call it **total** if the map is total, i.e., r is defined for all constants in R.*

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

Much of the interest in logical relations is derived from the following meta theorem, often referred to as the *Basic Lemma* (or parametricity, abstraction, fundamental, or independence theorem). Many important theorems such as strong normalization, type safety, and correctness of compiler optimizations for the simply-typed lambda calculus (and many related calculi) can be cast as corollaries of Basic Lemmas.

▶ **Theorem 11** (Basic Lemma (generalizing [RS13, Thm. 3.9] and [RR21b, Thm. 2]))**.** *For a partial logical relation r on morphisms $m_1, \ldots, m_n \colon R \to S$, we have*

- *if $\Gamma \vdash_R t \colon A$ and r is defined for t, then r is defined for A and*

$$r(\Gamma) \vdash_S r(t) \colon \overline{r}(A)\ m'_1(t)\ \ldots\ m'_n(t)$$

- *if r is term-total, it is defined for a typed term if it is for its type*

Proof. See the proof of [RR21b, Thm. 2], accounting for flexarity being a straightforward generalization. ◀

▶ **Notation 12.** In the case of logical relations on a single morphism m_1, we pretend for readability that Definition 10 and Theorem 11 simply used m_1 everywhere instead of m'.

We conclude this section with a number of examples:

▶ **Example 13** (Representing Type Preservation (based on [RR21b])). Consider the below theories representing the base of hard and soft typed formal systems, respectively [RR21b]. (The theory for soft typing also formalizes a unit type whose necessity becomes apparent in a second.)

$$\begin{array}{ll}
\textbf{theory}\ \texttt{HTyped} = \{ & \textbf{theory}\ \texttt{STyped} = \{ \\
\quad \texttt{tp} \colon \texttt{type} & \quad \texttt{tp} \quad \colon \texttt{type} \\
\quad \texttt{tm} \colon \texttt{tp} \to \texttt{type} & \quad \texttt{term} \colon \texttt{type} \\
\} & \quad :: \quad \colon \texttt{term} \to \texttt{tp} \to \texttt{type} \\
 & \quad \texttt{Unit} \colon \texttt{type} \\
 & \quad \texttt{unit} \colon \texttt{Unit} \\
 & \}
\end{array}$$

We represent the type erasure translation from hard to soft typing as the morphism `TypeEras` shown below on the left. Now we would like to state that for every hard-typed term

$$\vdash_{\texttt{HTyped}} t \colon \texttt{tm}\ a$$

the type-erased image $\texttt{TypeEras}(t)$ fulfills the typing

$$\vdash_{\texttt{STyped}} \texttt{TypeEras}(t) \colon \texttt{TypeEras}(t) :: a$$

We can cast this desired meta theorem as the unary logical relation on `TypeEras` shown below on the right.

$$\begin{array}{ll}
\textbf{mor}\ \texttt{TypeEras} \colon \texttt{HTyped} \to \texttt{STyped} = \{ & \begin{array}{ll} \mathit{TP}(\texttt{tp}) & = \texttt{Unit} \\ \mathit{TP}(\texttt{tm}) & = \lambda T \colon \texttt{tp}.\ \lambda T^* \colon \texttt{Unit}. \\ \mathit{eqcont}\lambda x \colon \texttt{term}.\ x :: T = & \end{array} \\
\quad == & \\
\} &
\end{array}$$

Note that we could use a partial logical relation to get rid of `Unit` that only acts as a placeholder. We will do so below in Example 15.

Instantiating the Basic Lemma from Theorem 11 for TP, we get: for every hard-typed term

$$\vdash_{\texttt{HTyped}} t \colon \texttt{tm}\ a$$

we have

$$\vdash_{\texttt{STyped}} \texttt{TypeEras}(t) \colon \texttt{TypeEras}(t) :: \texttt{TypeEras}(a)$$

which is exactly what we wanted (except for the minor generalization to $\texttt{TypeEras}(a)$ – which makes sense anyway when considering hard-typed dependent function types later on).

Even though type preservation at the level of generality of `HTyped` (without any concrete type formers) is not of much use, it serves as an instructive starter, which we will extend in Example 83.

▶ **Example 14** (Extending Type Preservation to Product Types (cont. Example 13; based on [RS13, Sec. 5] and [RR21b, Sec. 4])). Consider the following formalizations of hard- and soft-typed product types:

$$
\begin{array}{l}
\textbf{theory}\ \texttt{HProd} = \{\\
\quad \textbf{include}\ \texttt{HTyped}\\
\quad \texttt{prod}\ : \texttt{tp} \to \texttt{tp} \to \texttt{tp}\\
\quad \texttt{pair}\ : \Pi\, a\ b.\ \texttt{tm}\ a \to \texttt{tm}\ b \to\\
\qquad\qquad \texttt{tm}\ \texttt{prod}\ a\ b\\
\quad \texttt{projL} : \Pi\, a\ b.\ \texttt{tm}\ \texttt{prod}\, a\, b \to \texttt{tm}\ a\\
\quad \texttt{projR} : \Pi\, a\ b.\ \texttt{tm}\ \texttt{prod}\, a\, b \to \texttt{tm}\ b\\
\}
\end{array}
$$

$$
\begin{array}{l}
\textbf{theory}\ \texttt{SProd} = \{\\
\quad \textbf{include}\ \texttt{STyped}\\
\quad \texttt{prod}\ \ : \texttt{tp} \to \texttt{tp} \to \texttt{tp}\\
\quad \texttt{pair}\ \ : \texttt{term} \to \texttt{term} \to \texttt{term}\\
\quad \texttt{pair}^*\ : \Pi\, a\ b.\ \Pi\, x.\ \Vdash x :: a \to \Pi\, y.\ \Vdash y :: b \to\\
\qquad\qquad \Vdash (\texttt{pair}\, x\, y) :: (\texttt{prod}\, a\, b)\\
\quad \texttt{projL}\ : \texttt{term} \to \texttt{term}\\
\quad \texttt{projL}^* : \Pi\, a\ b.\ \Pi\, x.\ \Vdash x :: \texttt{prod}\, a\, b \to\\
\qquad\qquad \Vdash \texttt{projL}\, x :: a\\
\quad \texttt{projR}\ : \texttt{term} \to \texttt{term}\\
\quad \texttt{projR}^* : \Pi\, a\ b.\ \Pi\, x.\ \Vdash x :: \texttt{prod}\, a\, b \to\\
\qquad\qquad \Vdash \texttt{projR}\, x :: a\\
\}
\end{array}
$$

We now extend Example 13 to also capture type erasure and preservation from hard- to soft-typed products:

$$
\begin{array}{l}
\textbf{mor}\ \texttt{TypePresProd} : \texttt{HProd} \to \texttt{SProd} = \{\\
\quad = =\\
\}
\end{array}
$$

$$
\begin{array}{ll}
TP^{\times}(\texttt{tp}) & = \texttt{Unit}\\
TP^{\times}(\texttt{tm}) & = \lambda T : \texttt{tp}.\ \lambda T^* : \texttt{Unit}.\\
 & \quad \lambda x : \texttt{term}.\ x :: T
\end{array}
$$

$$
\begin{array}{ll}
TP^{\times}(\texttt{prod}) & = \lambda a : \texttt{tp}.\ \lambda a^* : \texttt{Unit}.\ \lambda b : \texttt{tp}.\ \lambda b^* : \texttt{Unit}.\ \texttt{unit}\\
TP^{\times}(\texttt{pair}) & = \lambda a : \texttt{tp}.\ \lambda a^* : \texttt{Unit}.\ \lambda b : \texttt{tp}.\ \lambda b^* : \texttt{Unit}.\ \texttt{pair}^*\ a\ b\\
TP^{\times}(\texttt{projL}) & = \lambda a : \texttt{tp}.\ \lambda a^* : \texttt{Unit}.\ \lambda b : \texttt{tp}.\ \lambda b^* : \texttt{Unit}.\ \texttt{projL}^*\ a\ b\\
TP^{\times}(\texttt{projR}) & = \lambda a : \texttt{tp}.\ \lambda a^* : \texttt{Unit}.\ \lambda b : \texttt{tp}.\ \lambda b^* : \texttt{Unit}.\ \texttt{projR}^*\ a\ b
\end{array}
$$

▶ **Example 15** (Conveniently Representing Type Preservation (cont. Examples 13 and 14)). In Example 14 we represented the type preservation property from hard- to soft-typed product types using a total logical relation. The totality led to a lot of awkward `Unit` type arguments, which we now get rid of by modifying *TP*to the following partial logical relation:

$$
\begin{array}{l}
TP(\texttt{tp}) = \bot\\
TP(\texttt{tm}) = \lambda T : \texttt{tp}.\ \lambda x : \texttt{term}.\ x :: T
\end{array}
$$

The corresponding logical relation for product types looks as follows:

$$\begin{aligned}
TP^{\times}(\texttt{tp}) &= \bot \\
TP^{\times}(\texttt{tm}) &= \lambda T\colon \texttt{tp}.\ \lambda x\colon \texttt{term}.\ x :: T
\end{aligned}$$

$$\begin{aligned}
TP^{\times}(\texttt{prod}) &= \bot \\
TP^{\times}(\texttt{pair}) &= \lambda a\colon \texttt{tp}.\ \lambda b\colon \texttt{tp}.\ \texttt{pair}^{*}\ a\ b \\
TP^{\times}(\texttt{projL}) &= \lambda a\colon \texttt{tp}.\ \lambda b\colon \texttt{tp}.\ \texttt{projL}^{*}\ a\ b \\
TP^{\times}(\texttt{projR}) &= \lambda a\colon \texttt{tp}.\ \lambda b\colon \texttt{tp}.\ \texttt{projR}^{*}\ a\ b
\end{aligned}$$

We observe that all unnecessary unit type arguments have gone.

3 A Framework of Diagram Operators

3.1 Motivating Example: The Pushout Operator

3.2 Linear Functors

Above we have identified a number of abstract structural properties of operators that enable us to apply those operators at large scales with predictable outputs. We now identify a large class of such operators that additionally have the property of being easy to state and verify.

3.2.1 Main Definition

▶ **Definition 16** (Linear Functor). *Given two theories S and T, we call a functor from a subcategory of $\mathbb{LF}^S$ to $\mathbb{LF}^T$* ***linear*** *if there is a partial binary function $\Delta_{\_}(-)$ such that*

i) O is defined declaration-wise on S-extensions:

$$O(S) = T \qquad O(S, \Sigma, c\colon A\,[= t]) = O(\Sigma), \Delta_\Sigma(c\colon A\,[= t])$$

where $\Delta_\Sigma(c\colon A\,[= t])$ is a list of constant declarations

ii) O is defined similarly for S-extension morphisms $\sigma\colon S, \Sigma \to S, \Sigma'$:

$$O(\mathrm{id}_S) = \mathrm{id}_T \qquad O(\mathrm{id}_S, \sigma, c\colon A\ = t) = O(\mathrm{id}_S, \sigma), \Delta_{\Sigma'}(c\colon A\ = t)$$

where $\Delta_{\Sigma'}(c\colon A\ = t)$ is a list of morphism assignments

iii) every declaration in $\Delta_\Sigma(c\colon A\,[= t])$ possesses a definiens whenever the input declaration does (A)

iv) the definedness and result of $\Delta_\Sigma(c\colon A\,[= t])$ are determined by $\Delta_{\Sigma_0}(c\colon A\,[= t])$ where Σ_0 are the dependencies of c, and (B)

We call S and T the ***functor's domain and codomain****, respectively, and Δ its* ***linear action****.*

The above definition is constructive in the sense that it provides us a scheme of conveniently defining linear functors. To define one, we simply need to state domain and codomain theory and a translation mapping declarations individually. The same holds true for implementing one (see also Section 6). Thus, linear functors are much easier to give than arbitrary functors on MMT theories.

▶ **Example 17** (Pushout Functor (Running Example)). In Section 3.1 we have given an ad-hoc specification of the pushout functor. We now give an equivalent specification using Definition 16. Let $m\colon S \to T$ be a fixed theory morphism. Then define `Push` to be the linear functor from S to T given by

$$\Delta(c\colon A\,[= t]) = c\colon m^\Sigma(A)\,[= m^\Sigma(t)]$$

where $m^\Sigma(-)$ is the compositional translation from S- to T-syntax defined as follows:

$$m^\Sigma(c) = \begin{cases} c & \text{if } c \in \Sigma \\ m(c) & \text{otherwise} \end{cases}$$

$$m^\Sigma(\texttt{type}) = \texttt{type} \qquad m^\Sigma(\texttt{kind}) = \texttt{kind}$$
$$m^\Sigma(x) = x \qquad m^\Sigma(f\ t) = m^\Sigma(f)\ m^\Sigma(t)$$
$$m^\Sigma(\Pi\, x\colon A.\ B) = \Pi\, x\colon m^\Sigma(A).\ m^\Sigma(B) \qquad m^\Sigma(\lambda x\colon A.\ t) = \lambda x\colon m^\Sigma(A).\ m^\Sigma(t)$$

$$m^\Sigma(\cdot) = \cdot \qquad m^\Sigma(\Gamma, x\colon A) = m^\Sigma(\Gamma), x\colon m^\Sigma(A)$$

Note that the only interesting case of $m^\Sigma(-)$ is the one on constants. All other cases are determined by compositionality, and we give them solely for the sake of completeness. Bearing that in mind, we observe that this specification is already easier to digest than the ad-hoc one previously given. This carries over to proofs of correctness, too, as we will see in Examples 28 and 39.

Some very general linear functors even attain the extreme case in Definition 16 of $S = T = \varnothing$ (e.g., see the operators in Section 4.4).

We have the following basic property that establishes linear functor to fit the abstract structural properties identified earlier in Section 3.1:

▶ **Theorem 18.** *Every linear functor is functorial from S to T and preserves includes.*

Proof. Straightforward. ◀

In contrast, functorial operators that preserve includes are not necessarily linear. For example, by setting $O(\Sigma) = \{c_1\_c_2 \colon \texttt{type} \mid c_1, c_2 \in \Sigma\}$ we can define an operator that produces a single declaration for every ordered pair of declarations in Σ. This operator still preserves includes but fails to fulfill the above third requirement on linear operators.

In Example 17 we observe that the function Δ also satisfies the following stricter property:

▶ **Definition 19** (Strongly Linear Operator). *A linear functor is called* ***strongly linear*** *if $\Delta_\Sigma(c \colon A\,[= t])$ always contains exactly one declaration (when defined), which is also named c.*

Strongly linear functors are even simpler to implement because – having fixed their domain and codomain, they are already determined by a pair (E, e) of expression translation functions by means of $\Delta_\Sigma(c \colon A\,[= t]) = c \colon E_\Sigma(A)\,[= e_\Sigma(t)]$. These can be seen as one translation function for types and one for typed terms. Thus, developers of functors only need to worry about the inductive expression translation functions, and the framework can take over all bureaucracy for names and declarations:

▶ **Example 20** (Pushout Functor (cont. Example 17)). For a morphism $m \colon S \to T$, the pushout functor is the strongly linear functor given by $E_\Sigma = e_\Sigma = m^\Sigma$.

Moreover, it is even simpler to check that two arbitrary functions (E_Σ, e_Σ) induce a strongly linear operator, and we refer to the below verification criteria in Section 3.2.2.

Even though for the pushout functor we even have $E_\Sigma = e_\Sigma$, this is not common enough to deserve its own definition. However, it is often the case that E_Σ and e_Σ are very similar, e.g., they might be the same except that E inserts a Π-binder where e inserts a λ one. (For an example, see the strongly linear functor described in **??**).

▶ Remark 21 (Name Clashes for Pushout). When defining the pushout functor so far (in Examples 17 and 20), we have been skipping over one subtlety of name clashes. Consequently, we have to exclude certain S-extensions from $\texttt{Push}_m$'s domain. However, since we left its domain implicit anyway, nothing changes in our previous formal definitions.

Suppose $c \in \Sigma$ is a constant, then due to assumed validity of Σ the identifier c is necessarily fresh wrt. S. But it may happen that T contains a constant with the same identifier, thus $\texttt{Push}_m(\Sigma)$ would be an invalid T-extension. Following [Rab17a, Rem. 2.28] there are multiple ways of resolving this problem, each way possibly sacrificing one property for another one in a tension triangle. The simplest way (which we choose) is to sacrifice totality and make $\texttt{Push}$ undefined on those S-extensions whose identifiers clash with declarations in

T. We continue our discussion in our section on the pushout functor in Remark 43. Importantly, in practice this problem does not appear anyway since the MMT system employs namespaces and identifiers that are qualified with theory names, see [Rab17a, Not. 2.29] and [MmtURI].

The problem of name clashes is not unique to the pushout functor and appears for almost all functors. Thus we declare:

▶ **Notation 22** (Partiality of Functors)**.** Unless otherwise noted, we take all our functors to be partial by default, i.e., whenever we mention a functor $F\colon A \to B$, this functor will only be defined on a subcategory of A. When clear from context, we usually omit mentioning this subcategory explicitly. Correspondingly, we implicitly extend all mathematical properties of a functor (functoriality, composition with other functors, etc.) to the partial case.

In particular, we adopt the convention that whenever we specify a linear functor O from S to T with linear action Δ, we implicitly make it partial on S-extensions Σ whenever the constants output by the linear action applied on Σ would nameclash with constants of T.

3.2.2 Verification Criteria

Definition 16 condenses quite a few properties in one definition (linearity, well-typedness, functoriality). In practice we almost always want to specify operators by giving a linear action (making linearity hold by construction) and then proving the fulfillment of well-typedness and functoriality. Indeed, the latter two properties are often non-trivial and merit larger proofs and verification criteria, which we give in this section.

We phrase our criteria mimicking the situation that arises when specifying operators: let S and T be two theories and let Δ be a linear action, i.e., a partial binary function fulfilling properties Items iii and iv from Definition 16. Similarly to Items i and ii from the same definition, we induce an operator O that maps S-extensions to *arbitrary*, i.e., possibly ill-typed T-extensions and that maps S-extension morphisms $\sigma\colon \Sigma \to \Sigma'$ also to *arbitrary* T-extension morphisms $O(\sigma)\colon O(\Sigma) \to O(\Sigma')$. Notably, we do not require O to be a functor: otherwise our verification criteria would not be needed. This operator is linear by construction. We call it well-typed if all theories and morphisms it outputs are well-typed. And we call a well-typed operator functorial if it actually is a functor on subcategories of $\mathbb{LF}$ (see **??**). We are now ready to spell out our verification criteria:

▶ **Definition 23.** *For strongly linear actions* (E, e)*, we define:*

- (E, e) *preserve typing if*

$$\begin{aligned} \vdash^S_\Sigma A\colon \texttt{type/kind} &\implies \vdash^T_{O(\Sigma)} E_\Sigma(A)\colon \texttt{type/kind} \\ \vdash^S_\Sigma t\colon A\colon \texttt{type} &\implies \vdash^T_{O(\Sigma)} e_\Sigma(t)\colon E_\Sigma(A) \end{aligned}$$

- E *preserves equality if*

$$\vdash^S_\Sigma A \equiv A'\colon \texttt{type/kind} \implies \vdash^T_{O(\Sigma)} E_\Sigma(A) \equiv E_\Sigma(A')$$

- E *commutes with morphisms if for all* S*-extension morphisms* $\sigma\colon \Sigma \to \Sigma'$

$$\vdash^S_\Sigma A\colon \texttt{type/kind} \implies \vdash^T_{O(\Sigma)} O(\sigma)(E_\Sigma(A)) \equiv E_{\Sigma'}(\sigma(A))$$

- e *commutes with morphisms if for all* S*-extension morphisms* $\sigma\colon \Sigma \to \Sigma'$

$$\vdash^S_\Sigma t\colon A\colon \texttt{type} \implies \vdash^T_{O(\Sigma)} O(\sigma)(e_\Sigma(t)) \equiv e_{\Sigma'}(\sigma(t))$$

- *e preserves constants if*

$$c \in \Sigma \implies \vdash^T_{O(\Sigma)} e_\Sigma(c) \equiv c$$

▶ **Theorem 24** (Well-Typedness Criterion for Linear Functor)**.** *The induced operator O is well-typed if and only if for all S-extensions Σ and S-extension morphisms σ we have:*

$$\vdash^S_\Sigma c\colon A \implies \vdash^T_{O(\Sigma)} \Delta_\Sigma(c\colon A)$$
$$\vdash^S_\sigma c\colon A = t \implies \vdash^T_{O(\sigma)} \Delta_\Sigma(c\colon A = t)$$

Moreover, if O is strongly linear and given by expression translation functions (E, e), it is sufficient for well-typedness of O that (E, e) preserve typing and E preserves equality and commutes with morphisms.

Proof. For the first claim, the forward implication immediately follows by instantiating the well-typedness hypothesis for the theory $O(S, \Sigma, c\colon A)$ and the morphism $O(\mathrm{id}_S, \sigma, c\colon A = t)$. The backward implication proceeds by inductions on the length of Σ and σ, respectively.

We reduce the second claim to the first one. The condition on constant declarations follows immediately from preservation of typing by the expression translation functions. For assignments, we consider the following situation of an S-extension morphism (on the first line) and its image under O (second line):

$$\begin{tikzcd} \Sigma, c\colon A \arrow[r, "{\sigma, c\colon A' = t}"] & \Sigma' \\ O(\Sigma), c\colon E_\Sigma(A) \arrow[r, "{O(\sigma), c\colon E_{\Sigma'}(A') = e_{\Sigma'}(t)}"] & O(\Sigma') \end{tikzcd}$$

We need to confirm that the arrow shown on the second line is indeed a morphism. By definition, we check *i)* $\vdash^T_{O(\Sigma)} e_{\Sigma'}(t)\colon E_{\Sigma'}(A')$ and *ii)* $\vdash^T_{O(\Sigma)} O(\sigma)(E_\Sigma(A)) \equiv E_{\Sigma'}(A')$. The former is immediate by preservation of typing. For the latter we use the remaining hypotheses and compute $O(\sigma)(E_\Sigma(A)) \equiv E_{\Sigma'}(\sigma(A)) \equiv E_{\Sigma'}(A')$ as desired. ◀

▶ **Lemma 25** (Functoriality Criterion for Strongly Linear Operators)**.** *If O is given by expression translation functions (E, e) and is well-typed, then it is functorial if and only if e preserves constants and commutes with morphisms.*

▶ **Corollary 26.** *If O is induced by expression translation functions (E, e), then it is well-typed and functorial if and only if (E, e) preserve typing and both commute with morphisms, E preserves equality, and e preserves constants.*

The condition in Lemma 25 of preserving constants might look surprising because we might be tempted to think that by induction it would make the translation e the identity on all terms (and thus the theory of strongly linear operators a trivial theory). But that needs not be the case for two reasons. First, e is only required to be the identity on constants occurring in S-extensions and not necessarily on S-constants themselves. Second, e does not need to be compositional, thus even if it was the identity on all base cases, it would not need to be the identity on complex terms.

Proof of Lemma 25. $\Leftarrow$: We need to show that O preserves identities and composition of morphisms. The former immediately follows from e preserving constants. For the latter, consider S-extension morphisms $\Sigma \xrightarrow{\sigma} \Sigma' \xrightarrow{\sigma'} \Sigma''$. We show the equality $O(\sigma' \circ \sigma) =$

$O(\sigma') \circ O(\sigma)$ element-wise on constants (as per Lemma 7): Let $c \in O(\Sigma)$ be a constant; by strong linearity c is also a constant in Σ. In $O(\Sigma'')$ we obtain the following equality chain:

$$\begin{aligned}
O(\sigma' \circ \sigma)(c) &= e_{\Sigma''}((\sigma' \circ \sigma)(c)) && \text{by strong linearity of } O \\
&= e_{\Sigma''}(\sigma'(\sigma(c))) && \text{by def. of morphism composition} \\
&= O(\sigma')(e_{\Sigma'}(\sigma(c))) && \text{by } e \text{ commuting with morphisms} \\
&= O(\sigma')(O(\sigma)(e_{\Sigma}(c))) && \text{by } e \text{ commuting with morphisms} \\
&= (O(\sigma') \circ O(\sigma))(e_{\Sigma}(c)) && \text{by def. of morphism composition} \\
&= (O(\sigma') \circ O(\sigma))(c) && \text{by } e \text{ preserving constants}
\end{aligned}$$

$\Rightarrow$: Let O be functorial. Preservation of constants by e immediately follows from O preserving identities. We recover the commutation of e with morphisms by a clever instantiation of the hypothesis that O preserves morphism compositions. Let $\vdash^{S}_{\Sigma} t\colon A$ be a term and $\sigma\colon \Sigma \to \Sigma'$ an S-extension morphisms. We need to show $\vdash^{T}_{O(\Sigma)} O(\sigma)(e_{\Sigma}(t)) \equiv e_{\Sigma'}(\sigma(t))$. To invoke functoriality, first consider the morphism chain

$$\Sigma, c\colon A \xrightarrow{\mathrm{id}_{\Sigma}, c\colon A = t} \Sigma, c\colon A \xrightarrow{\sigma, c\colon \sigma(A) = c'} \Sigma', c'\colon \sigma(A)$$

where we picked a constant identifier c that is fresh among Σ, Σ', A, t, $\sigma(t)$, and $e_{\Sigma}(t)$. Applying O and functoriality we get the commuting diagram below. (There and in the following we omit the type component in assignments for readability.)

$$\begin{tikzcd}
O(\Sigma, c\colon A) \arrow[r, "O(\mathrm{id}_{\Sigma}, c:=t)"] \arrow[rr, bend right, "O((\sigma, c:=c') \circ (\mathrm{id}_{\Sigma}, c:=t))"'] & O(\Sigma, c\colon A) \arrow[r, "O(\sigma, c:=c')"] & O(\Sigma', c'\colon \sigma(A))
\end{tikzcd}$$

We instantiate the commutation for the constant $c \in O(\Sigma, c\colon A)$ (existing by strong linearity). For the the upper two morphisms applied on c we get

$$\begin{aligned}
& (O(\sigma, c := c') \circ O(\mathrm{id}_{\Sigma}, c := t))\,(c) \\
= \;& O(\sigma, c := c')(e_{\Sigma, c\colon A}(t)) && \text{by def.} \\
= \;& O(\sigma, c := c')(e_{\Sigma}(t)) && \text{by freshness of } c \text{ wrt. } t \\
= \;& O(\sigma)(e_{\Sigma}(t)) && \text{by freshness of } c \text{ wrt. } e_{\Sigma}(t)
\end{aligned}$$

And for the lower morphism applied on c we get

$$\begin{aligned}
& O((\sigma, c := c') \circ (\mathrm{id}_{\Sigma}, c := t))(c) \\
= \;& e_{\Sigma', c'\colon \sigma(A)}(((\sigma, c := c') \circ (\mathrm{id}_{\Sigma}, c := t))(c)) && \text{by def.} \\
= \;& e_{\Sigma', c'\colon \sigma(A)}((\sigma, c := c')(t)) && \text{by def.} \\
= \;& e_{\Sigma', c'\colon \sigma(A)}(\sigma(t)) && \text{by freshness of } c \text{ wrt. } t \\
= \;& e_{\Sigma'}(\sigma(t)) && \text{by freshness of } c' \text{ wrt. } \sigma(t)
\end{aligned}$$

The equality of the two final results is precisely what we needed to prove. ◀

Many of the hypotheses for functoriality needed in Lemma 25 can be let go if we require e to be compositional:

▶ **Corollary 27.** *Suppose O is induced by expression translation functions (E, e) where e is compositional, i.e., for all S-extensions Σ and well-typed LF terms it holds*

$$e_\Sigma(\mathtt{type}) = \mathtt{type} \qquad e_\Sigma(\mathtt{kind}) = \mathtt{kind}$$
$$e_\Sigma(c) = c \qquad e_\Sigma(x) = x$$
$$e_\Sigma(f\ t) = e_\Sigma(f)\ e_\Sigma(t)$$
$$e_\Sigma(\Pi\, x\colon A.\ B) = \Pi\, x\colon e_\Sigma(A).\ e_\Sigma(B) \qquad e_\Sigma(\lambda x\colon A.\ t) = \lambda x\colon e_\Sigma(A).\ e_\Sigma(t)$$

Then O is already functorial if it is well-typed.

Proof. We use Lemma 25 and show that commutation of e with morphisms is already implied by compositionality. We induct on terms $\vdash^S_\Sigma t$:

- constants $t = c \in \Sigma$: we have $O(\sigma)(e_\Sigma(c)) = O(\sigma)(c) = e_{\Sigma'}(\sigma(c))$ by e preserving constants
- constants $c \in S$ from the operator's domain: we note that $e_\Sigma(c) = e(c)$ must be a well-typed term already over T. Since σ (or $O(\sigma)$) is the identity on S (or T), we conclude $O(\sigma)(e_\Sigma(c)) = e(c) = e_{\Sigma'}(c) = e_{\Sigma'}(\sigma(c))$.
- complex cases: they all follow immediately by compositionality. For example for function applications $f\ t$ we have

$$\begin{array}{lll} & O(\sigma)(e_\Sigma(f\ t)) & \\ = & O(\sigma)(e_\Sigma(f)\ e_\Sigma(t)) & \text{by compositionality of } e \\ = & O(\sigma)(e_\Sigma(f))\ O(\sigma)(e_\Sigma(t)) & \text{by compositionality of morphisms} \\ = & e_{\Sigma'}(\sigma(f))\ e_{\Sigma'}(\sigma(t)) & \text{by induction hypotheses} \\ = & e_{\Sigma'}(\sigma(f)\ \sigma(t)) & \text{by compositionality of } e \\ = & e_{\Sigma'}(\sigma(f\ t)) & \text{by compositionality of morphisms} \end{array}$$

◀

▶ **Example 28** (Pushout is Well-Typed and Functorial (cont. Example 20))**.** We consider the specification of $\mathtt{Push}_m$ by expression translation functions $E = e = m^\Sigma$ previously given in Example 20 and verify the properties of the putative functor defined thereby.

For well-typedness, we use the criterion for strongly linear operators given in Theorem 24. We apply induction on the following strengthened claims (adding contexts):

$$\begin{array}{rcl} \Gamma \vdash^S_\Sigma A\colon \mathtt{type/kind} & \Longrightarrow & m^\Sigma(\Gamma) \vdash^T_{\mathtt{Push}_m(\Sigma)} m^\Sigma(A)\colon \mathtt{type/kind} \\ \Gamma \vdash^S_\Sigma t\colon A & \Longrightarrow & m^\Sigma(\Gamma) \vdash^T_{\mathtt{Push}_m(\Sigma)} m^\Sigma(t)\colon m^\Sigma(A) \\ \Gamma \vdash^S_\Sigma A \equiv A' & \Longrightarrow & m^\Sigma(\Gamma) \vdash^T_{\mathtt{Push}_m(\Sigma)} m^\Sigma(A) \equiv m^\Sigma(A') \end{array}$$

In fact, the rather lengthy proof (which we omit here) necessitates further claims, e.g., on contexts and reduction. Almost all cases easily follow from induction hypotheses. We only present the critical case (CONST-OBJ) for the judgement listed on the second line above. Given the judgement

$$\frac{\vdash^S_\Sigma \Gamma \qquad c\colon A \in S, \Sigma}{\Gamma \vdash^S_\Sigma c\colon A}$$

we need to show $m^\Sigma(\Gamma) \vdash^T_{\mathtt{Push}_m(\Sigma)} m^\Sigma(c)\colon m^\Sigma(A)$. In case c was declared in S, we have $\vdash^S c\colon A$ and by applying the morphism m we have $\vdash^T m(c)\colon m(A)$ since morphisms preserve all judgements [Rab17a]. By noting $m^\Sigma(c) = m(c)$ and $m^\Sigma(A) = m(A)$ (since both c and

A are S-terms) we have $\vdash^T m^\Sigma(c) \colon m^\Sigma(A)$. The goal now follows by the *weakening theorem* of LF [HHP93] stating that LF judgements are stable under such extensions of signatures and contexts. Otherwise if $c \in \Sigma$, we have $m^\Sigma(c) = c$. By construction of $\texttt{Push}_m$, we have $\vdash^T_{\texttt{Push}_m(\Sigma)} c \colon m^\Sigma(A)$ from which the goal again follows by weakening. For well-typedness, it remains to show that E commutes with morphisms. But this is clear since E is compositional, e.g., see the proof of Corollary 27 for analogous reasoning.

Since e is compositional and we know that $\texttt{Push}_m$ is well-typed, we get functoriality for free by Corollary 27.

3.3 Linear Connectors

While a functor maps theories X to $O(X)$ and morphisms f to $O(f)$, a *connector from O to O'* serves to interrelate their outputs by mapping theories X to morphisms $O(X) \to O'(X)$. We can intuitively think of connectors as natural transformations between functors, although certain technicalities make this imprecise in the formal sense. Thus, connectors allow transporting content generated by one functor application to another. This feature is critical for our overarching goal of achieving feature-complete and highly interrelated standard libraries.

For example, for the pushout example from Section 3.1, assuming a morphism $m \colon S \to T$ we have already cast the functorial action as a linear functor $\texttt{Push}_m$ from S to T. With connectors, we will be able to cast the generation of the family of morphisms m^X for all S-extensions X as a connector $\texttt{PushIn}_m$ from the identity functor $\texttt{Id}$ to $\texttt{Push}_m$.

3.3.1 Main Definition

▶ **Definition 29** (Linear Connector). *Let $O \colon \mathbb{LF}^S \to \mathbb{LF}^T$ be a linear functor and $O' \colon \mathbb{LF}^S \to \mathbb{LF}^{T'}$ a functor. By $\iota_T \colon \mathbb{LF}^T \to \mathbb{LF}$ and $\iota_{T'} \colon \mathbb{LF}^{T'} \to \mathbb{LF}$ we denote the canonical functors defined after Definition 8.*

Let $m \colon T \to T'$ be a theory morphism. We call a natural transformation C from $\iota_T \circ O$ to $\iota_{T'} \circ O'$ ***linear*** *(or: a* ***linear connector****) over m if there is a partial binary function $\delta(-)$ such that*

i) C is defined for S-extensions declaration-wise:

$$C(S) = m \qquad C(S, \Sigma, c \colon A) = C(S, \Sigma), \delta_\Sigma(c \colon A)$$

where by $C(-)$ we denote the morphism of the natural transformation at the theory given as the argument

ii) whenever defined, the declarations in $\delta_\Sigma(c \colon A)$ all possess definienses, and also have the same names in the same order as the declarations in $\Delta_\Sigma(c \colon A)$ where Δ is the linear action of O

iii) the definedness and result of $\delta_\Sigma(c \colon A)$ are determined by $\delta_{\Sigma_0}(c \colon A)$ where Σ_0 are the dependencies of c; moreover they may depend on the identifier c

We call O and O' the ***connector's domain and codomain****, respectively, and will often say that C is a connector "out of O" or "into O'".*

If clear from context, we will conflate $C(-)$ with morphism application and, e.g., write $C_\Sigma(t)$ to mean the translation of Σ-term t under $C(S, \Sigma)$.

▶ **Theorem 30.** *Every linear connector preserves includes.*

Proof. Straightforward. ◀

Fixing m, linear connectors are uniquely determined by δ and vice versa. It is crucial for the linear action of connectors to be handed the input constant's identifier, see Example 32 below.

Connectors need not be defined for defined constants since morphisms, which connectors serve to output, also need not be defined on them. Concretely, the definition can be induced compositionally by apply the morphism on the constant's definition.

▶ **Definition 31.** *A linear connector out of a strongly linear functor is called* ***strongly linear*** *if $\delta_\Sigma(c\colon A)$ always contains exactly one declaration (when defined).*

Analogously to strongly linear functors, having fixed O, O', and m, strongly linear connectors are already determined by a pair $(\mathcal{E}, \varepsilon)$ of expression translation functions by means of $\delta_\Sigma(c\colon A) = c\colon \mathcal{E}(c\colon A) = \varepsilon(c\colon A)$. (However, in contrast to strongly linear functors, here both functions are applied on the *type* of c.)

▶ **Example 32** (Pushout Connector (cont. Example 20))**.** For a fixed morphism $m\colon S \to T$, we define the connector $\mathtt{PushIn}_m$ from the identity functor into the pushout functor $\mathtt{Push}$ as the strongly linear connector over m given by $\mathcal{E} = m^\Sigma$ and $\varepsilon(c\colon A) = c$. Its linear action is thus given by

$$\delta_\Sigma(c\colon A) = c\colon m^\Sigma(A) = c$$

To see what the connector does, consider an arbitrary S-extension $\Sigma = \{c_1, \dots, c_n\}$. Applying the connector, we get the morphism $\mathtt{PushIn}_m(\Sigma) = m, \{c_1 := c_1, \dots, c_n := c_n\}$ (In both cases we have omitted types for readability, and we explicitly included m in the notation for clarity.) We see that $\mathtt{PushIn}_m(\Sigma)$ on Σ a no-op and on S identical to m. By construction this is the same behavior as $m^\Sigma(-)$, and by a simple induction we can show $\vdash^T_{\mathtt{Push}_m(\Sigma)} \mathtt{PushIn}_{m,\Sigma}(t) = m^\Sigma(t)$ for every term $\vdash^S_\Sigma t\colon A$.

Thus, intuitively, we should be able to get rid of m^Σ and only define the connector. However, this would require circular definitions: we would need $\mathtt{PushIn}_m$ to define $\mathtt{Push}_m$, and we would need $\mathtt{Push}_m$ to define $\mathtt{PushIn}_m$ (to serve as the domain). In the implementation, such a circular definition is feasible, merging the responsibilities of m^Σ and $\mathtt{PushIn}_m$. In fact, not even recursion is needed.

Closure Properties of Functors & Connectors It is typical to compose functors with one another and to compose connectors with one another. Thus, we state the following closure properties:

▶ **Theorem 33** (Category of Theory Extensions Categories)**.** *As objects consider the class*[4] *of theory extensions $\{\mathbb{LF}^T \mid T \text{ theory}\}$ and as morphisms one of*

- *functors*
- *include-preserving functors*
- *linear functors*
- *strongly linear functors*

Every choice of morphisms listed above gives rise to a category. Moreover, in the above order they form a strictly ordered chain of subcategories.

[4] We gloss over size issues here and continue to use the terminology of *class* and *category* even for their larger-sized analogues.

▶ **Theorem 34** (Category of Theory Functors). *Consider the following choices of objects and morphisms:*

- *functors and natural transformations*
- *include-preserving functors and include-preserving connectors*
- *linear functors and linear connnectors*
- *strongly linear functors and strongly linear connectors*

Every choice listed above gives rise to a category. Moreover, in the above order they form a strictly ordered chain of subcategories.

Proof. Both theorems are straightforward to prove, albeit one needs to watch out to define things suitably since our functors and connectors must allow for partiality. ◀

3.3.2 Verification Criteria

Definition 29 condenses quite a few properties in one definition (linearity, well-typedness, naturality). As we did for linear functors, we want to offer verification criteria that disentangle these properties into separate conditions. In practice we almost always specify putative connectors by giving a linear action. Thus, linearity will always be fulfilled by construction. But the other two properties are often non-trivial and merit being proven separately.

We phrase our criteria mimicking the situation that arises when specifying connectors: Let O be a linear functor with linear action Δ and O' be just a partial endofunctor on $\mathbb{LF}$. Let δ be a partial binary function fulfilling Items ii and iii from Definition 29. Similarly to Item i from the same definition, we induce an operator C that maps S-extensions Σ to *arbitrary*, i.e., possibly ill-typed morphisms $C_\Sigma \colon O(\Sigma) \to O'(\Sigma)$. This operator is linear by construction. We call it well-typed if all morphisms it outputs are well-typed. And we additionally call it natural if it actually is a natural transformation in the sense of Definition 29.

▶ **Theorem 35** (Well-Typedness Criterion for Linear Connectors). *The induced operator C is well-typed if and only if and only if for all S-extensions Σ we have*

$$\vdash^S_\Sigma c\colon A \implies \vdash^T_{C(\Sigma)} \delta_\Sigma(c\colon A)$$

Moreover, if C is strongly linear and given by $(\mathcal{E}, \varepsilon)$, then it is well-typed if and only if for all S-extensions Σ and constants $c \in \Sigma$ we have:

i) $\vdash^{T'}_{O'(\Sigma)} C_\Sigma(E_\Sigma(A)) = \mathcal{E}(A)$
ii) $\vdash^{T'}_{O'(\Sigma)} \varepsilon(c\colon A)\colon \mathcal{E}(c\colon A)$

Proof. By definition of well-typedness for morphisms. ◀

For reference and for the sake of completeness, we explicitly spell out what it means to be natural:

▶ **Theorem 36** (Naturality). *Suppose the induced operator C is well-typed out of a linear functor given by linear action Δ. Then it is natural if and only if for all S-extension morphisms $\sigma\colon \Sigma \to \Sigma'$ the square below on the right commutes.*

$$\begin{tikzcd}
\Sigma \arrow[d, "\sigma"] & O(\Sigma) \arrow[r, "C_\Sigma"] \arrow[d, "O(\sigma)"] & O'(\Sigma) \arrow[d, "O'(\sigma)"] \\
\hat{\Sigma} & O(\hat{\Sigma}) \arrow[r, "C_{\hat{\Sigma}}"] & O'(\hat{\Sigma})
\end{tikzcd}$$

Concretely that means for all constants $c \in \Sigma$ and $d \in \delta_\Sigma(c)$ we have:

$$\vdash^{T'}_{O(\hat{\Sigma})} C_{\hat{\Sigma}}(O(\sigma)(d)) = O'(\sigma)(C_\Sigma(d))$$

Proof. By definition of naturality as in category theory. ◀

▶ **Corollary 37** (Naturality of Connector out of Identity)**.** *Suppose the induced operator C is well-typed and goes from the identity functor to a functor O'. Then it is natural if and only if for all S-extension morphisms $\sigma\colon \Sigma \to \Sigma'$ the square below on the right commutes.*

$$\begin{tikzcd} \Sigma \arrow[d, "\sigma"] \\ \hat{\Sigma} \end{tikzcd} \qquad \begin{tikzcd} \Sigma \arrow[r, "C_\Sigma"] \arrow[d, "\sigma"] & O'(\Sigma) \arrow[d, "O'(\sigma)"] \\ \hat{\Sigma} \arrow[r, "C_{\hat{\Sigma}}"] & O'(\hat{\Sigma}) \end{tikzcd}$$

Concretely that means that for all constants $c \in \Sigma$ we have:

$$\vdash^{T}_{O'(\hat{\Sigma})} C_{\hat{\Sigma}}(\sigma(c)) = O'(\sigma)(C_\Sigma(c))$$

An often-occurring pattern for linear functors is to copy input declarations to systematically qualified copies. The following theorem establishes that projections of those constants are always well-typed and natural:

▶ **Theorem 38.** *Consider a well-typed linear functor O that as part of its linear action Δ copies every input declaration c to a systematically qualified copy c^d (for some arbitrary but fixed tag d). Concretely, we assume*

$$\Delta(c\colon A\,[= t]) = \dots, c^d\colon A^d\,[= t^d], \dots$$

where $-^d$ is the compositional translation on terms given by

$$\begin{gathered} \texttt{type}^d = \texttt{type} \qquad \texttt{kind}^d = \texttt{kind} \\ c^d = c^d \qquad x^d = x^d \\ (f\ t)^d = f^d\ t^d \\ (\Pi\, x\colon A.\ B)^d = \Pi\, x^d\colon A^d.\ B^d \qquad (\lambda x\colon A.\ t) = \lambda x^d\colon A^d.\ t^d \end{gathered}$$

where the cases for constants and variables are non-recursive (despite notation suggesting that).

Then the strongly linear connector C into O given by

$$\delta_\Sigma(c\colon A) = c\colon A^d\ = c^d$$

is well-typed and natural.

▶ **Example 39** (Connector into Pushout is Well-Typed and Natural (cont. Examples 20, 28, and 32))**.** We verify the properties of the putative pushout connector that we have previously defined in Example 32.

For well-typedness we use Theorem 35: let $c \in \Sigma$ be a constant in an S-extension Σ. Condition i amounts to showing $\vdash^{T}_{\texttt{Push}_m(\Sigma)} \texttt{PushIn}_{m,\Sigma}(A) = m^\Sigma(A)$, which we already noted in Example 32. And Condition ii amounts to showing $\vdash^{T}_{\texttt{Push}_m(\Sigma)} c\colon m^\Sigma(A)$, which holds since $(c\colon m^\Sigma(A)) \in \texttt{Push}_m(\Sigma)$ is a constant in $\texttt{Push}_m(\Sigma)$ by the very construction of $\texttt{Push}$.

For naturality, we use **??**: let $\sigma\colon \Sigma \to \tilde{\Sigma}$ be an S-extension morphism and $c \in \Sigma$. We start with the RHS and calculate

$$\texttt{Push}_m(\sigma)(\texttt{PushIn}_{m,\Sigma}(c)) = \texttt{Push}_m(\sigma)(c) = m^{\Sigma}(\sigma(c)) = \texttt{PushIn}_{m,\Sigma}(\sigma(c))$$

obtaining the desired LHS.

▶ **Corollary 40** (Naturality of Strongly Linear Connector into Identity). *Suppose the induced operator C is well-typed and strongly linear (given by $(\mathcal{E}, \varepsilon)$ and goes from a strongly linear functor O (given by (E, e)) into the identity. Then it is natural if and only if for all S-extension morphisms $\sigma\colon \Sigma \to \Sigma'$ the square below on the right commutes.*

$$\begin{tikzcd} \Sigma \arrow[d, "\sigma"] \\ \hat{\Sigma} \end{tikzcd} \qquad \begin{tikzcd} O(\Sigma) \arrow[r, "C_\Sigma"] \arrow[d, "O(\sigma)"] & \Sigma \arrow[d, "\sigma"] \\ O(\hat{\Sigma}) \arrow[r, "C_{\hat{\Sigma}}"] & \hat{\Sigma} \end{tikzcd}$$

Concretely that means that for all assignments $(c\colon A = t) \in \sigma$ we have:

$$\vdash^S_{\hat{\Sigma}} C_{\hat{\Sigma}}(e_{\hat{\Sigma}}(t)) = \sigma(\varepsilon(A))$$

3.4 Structure-Preserving Lifting

3.5 Related Work

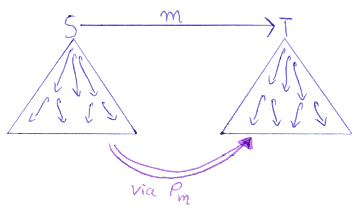

Figure 5 Using pushout to translate whole developments between base languages

4 Logic-Independent Operators

Logic-independent operators work on all MMT theories in principle or on all S-extensions for S representing a very small, i.e., weak theory. These operators are typically very foundational and granted many use cases. A prime example is the pushout functor, which we describe in the following.

4.1 Pushout

$$\begin{tikzcd} S \arrow[r, "m"] \arrow[d, hook] & T \arrow[d, hook] \\ X \arrow[r, "\texttt{PushIn}_m(X)"] & \texttt{Push}_m(X) \end{tikzcd}$$

Let $m\colon S \to T$ be a fixed morphism between structured theories. The linear functor $\texttt{Push}_m$ maps S-extensions X to the pushout $\texttt{Push}_m(X)$ depicted above. And the linear connector $\texttt{PushIn}_m$ creates morphisms $\texttt{PushIn}_m(X)\colon X \to \texttt{Push}_m(X)$ that extend the initial translation m homomorphically.

The pushout operation is typical for *compositionally* transporting developments relative to a base language S to a different base language T. Concretely, to represent a language L (e.g., a type theory or logic) in LF, we usually employ an LF-theory S to represent the syntax and possibly proof calculus of L [HHP93]. L-theories are then represented as LF-theories extending S, and L-morphisms between L-theories as LF-morphisms that agree with id_S. Now whenever for two languages represented as S and T we can phrase a desired translation as a morphism $m\colon S \to T$, then whole developments for the former language (i.e., structured diagrams) can be translated to the latter language by means of $\texttt{Push}_m$, see Figure 5. Moreover, we can relate S-diagrams and resulting T-diagrams using $\texttt{PushIn}_m$. We refer to [Rab17a] for examples of important language translations occurring in practice.

The same pushout functor and connector for the very same setting of MMT theories and morphisms have already been given in [Rab17a, Def. 2.26]. Effectively, our contribution is to showcase how both can be easily phrased and proven correct in our framework of diagram operators.

4.1.1 Definition

▶ **Definition 41** (Pushout Functor). *Let $m\colon S \to T$ be a morphism between two structured theories. The strongly linear functor $\texttt{Push}_m$ from S to T is given by*

$$E = e = m^{\Sigma}$$

where $m^\Sigma(-)$ *is the compositional translation from* S*- to* T*-syntax and given by*

$$m^\Sigma(c) = \begin{cases} c & \textit{if } c \in \Sigma \\ m(c) & \textit{otherwise} \end{cases}$$

$$m^\Sigma(\texttt{type}) = \texttt{type} \qquad m^\Sigma(\texttt{kind}) = \texttt{kind}$$
$$m^\Sigma(x) = x \qquad m^\Sigma(f\ t) = m^\Sigma(f)\ m^\Sigma(t)$$
$$m^\Sigma(\Pi\, x\colon A.\ B) = \Pi\, x\colon m^\Sigma(A).\ m^\Sigma(B) \qquad m^\Sigma(\lambda x\colon A.\ t) = \lambda x\colon m^\Sigma(A).\ m^\Sigma(t)$$

$$m^\Sigma(\cdot) = \cdot \qquad m^\Sigma(\Gamma, x\colon A) = m^\Sigma(\Gamma), x\colon m^\Sigma(A)$$

We only define $\texttt{Push}_m$ *on those* S*-extensions whose constant identifier are fresh wrt. those in* T*.*

▶ **Definition 42** (Pushout Connector). *For a morphism* $m\colon S \to T$*, the strongly linear connector* $\texttt{PushIn}_m\colon \texttt{Id} \to \texttt{Push}_m$ *over* m *is given by*

$$\texttt{PushIn}_{m,\Sigma}(c\colon A) = c$$

▶ Remark 43 (Partiality (cont. Remark 21)). In Remark 21 we already explained the subtlety of name clashes, and Definition 41 now makes explicit our way of avoiding them by sacrificing totality while still retaining natural identifiers. Following [Rab17a, Rem. 2.28], there are multiple ways of resolving this problem, and sacrificing totality is just one way. Other ways include either giving up on coherence properties or on choice of natural identifiers. Here, coherence properties refers to functorial properties of `Push` along compositions of morphisms [Rab17a, Def. 2.4], and natural identifiers refer to keeping input identifiers as well as possible (at best the same, at worst prefixing/suffixing). We will prove some coherence properties in Section 4.1.3.

Generally, in the setting of specification languages computing *some* colimit is usually easy (provided one exists), but actually selecting a *good representative* of all possible colimits (the one that the user works with) is non-trivial. We refer to [CMR16] (superseding [Rab17a] in this regard), where this problem has been extensively studied. (However, we recommend first consulting [Rab17a] as its setting closely matches ours).

4.1.2 Examples

▶ **Example 44** (Pushouts for the Algebraic Hierarchy). Consider the following base theories for a formalization of the algebraic hierarchy:

theory `Set` = {
  include `SFOL`
  U : `tp`
}

On top of those theories, we can formalize several

theory `Magma` = {
  include `Set`
  U : `tp`
  $\circ$: `tm` $U \to$ `tm` $U \to$ `tm` U
}

dozens of important algebra theories, e.g., monoids, groups, modules, fields, and vectorspaces. Now for each algebra theory, we would also like to have formalized the corresponding *finite* and *commutative* version. We can easily achieve this using pushout. First, we formalize the respective commutative and finite base theories shown below.

```
theory FinSet = {
   include Set
   fin: ⊩ ...
}
```

```
theory CommMagma = {
   include Magma
   comm: ⊩ ∀ x y. tm U x ∘ y ≐ y ∘ x
}
```

Let us give the names $i_{\mathtt{fin}} \colon \mathtt{Set} \hookrightarrow \mathtt{FinSet}$ and $i_{\mathtt{comm}} \colon \mathtt{Magma} \hookrightarrow \mathtt{CommMagma}$ to the inclusion morphisms. Then, we aggregate our algebraic hierarchy in two diagrams

$$\begin{aligned} D_{\mathtt{Set}} &= \mathrm{Diagram}(\mathtt{Magma}, \mathtt{Monoid}, \mathtt{Group}, \ldots) \\ D_{\mathtt{Magma}} &= \mathrm{Diagram}(\mathtt{Monoid}, \mathtt{Group}, \ldots) \end{aligned}$$

and obtain the finite and commutative versions in one liners, respectively:

$$\textbf{install } \mathtt{Push}_{i_{\mathtt{fin}}}(D_{\mathtt{Set}})$$
$$\textbf{install } \mathtt{Push}_{i_{\mathtt{comm}}}(D_{\mathtt{Magma}})$$

Pictorially, these statements issue the following theory-wise pushouts:

$$\begin{tikzcd} \mathtt{Set} \arrow[r, hook, "i"] \arrow[d, hook] & \mathtt{FinSet} \\ X \arrow[r] & \mathtt{Push}_i(X) \end{tikzcd} \qquad \begin{tikzcd} \mathtt{Magma} \arrow[r, hook, "j"] \arrow[d, hook] & \mathtt{CommMagma} \arrow[d, hook] \\ X \arrow[r] & \mathtt{Push}_j(X) \end{tikzcd}$$

Thus, the pushout functor allows to conveniently adjoin constants to whole developments. See also Section 4.2.2 for a further example.

We have not introduced notations formally in the present thesis, but in practice pushout can also be used to realize notation changes across whole developments. For example, consider a development of group theory using a multiplicative notation (i.e., using $\circ$, $-^{-1}$, and 1). Using a single pushout similar to above we can change it to an additive notation (i.e., using $+$, $-$, and 0).

Pushouts and colimits play a central role in identifying, translating, and combining logics represented in MMT [Rab17a]. We refer to the cited work for extensive examples.

We also refer to [Koh+] (coauthored by the author) for a user-practical application of MMT pushouts. There, pushouts are used for an educational 3D game to instantiate abstract theorems, e.g., on trigonometry with concrete lengths and angles as measured by the player in the game world.

4.1.3 Meta-Theoretical Properties

In Example 32 we already noted the result below and explained why, despite this result, we needed to define both `PushIn` and m instead of one merged definition.

▶ **Proposition 45.** *For all terms* $\vdash^S_\Sigma t$ *we have*

$$\mathtt{PushIn}_{m,\Sigma}(t) = m^\Sigma(t)$$

Proof. Straightforward by induction. ◀

▶ **Theorem 46.** $\mathtt{Push}_m$ *is well-typed and functorial.*

Proof. Already proven in Example 28. ◀

▶ **Theorem 47.** $\mathtt{PushIn}_m$ *is well-typed and natural.*

Proof. Already proven in Example 39. ◀

Many meta-theoretical properties of our pushout functor can be recovered from its categorical foundations, where it emerges as a special case. Consider a category $\mathcal{C}$ with pushouts, and for objects $c \in \mathcal{C}$ write $c/\mathcal{C}$ for the corresponding coslice category[5]. For every morphism $m\colon c \to c'$ we can define the functor $F\colon c/\mathcal{C} \to c'/\mathcal{C}$ that on objects $u\colon c \to x$ computes *some* pushout along u and m, and on morphisms $(u\colon c \to x) \to (v\colon c \to x')$ computes *some* universal morphism $Fu \to Fv$ out of the pushout Fu. General properties of such functors are well-known and we refer to standard texts on category theory.

After all, we are more interested in properties that depend on our *specific* choice of pushout because that choice matters to human end users. Accordingly, we present several such results next.

Coherence Properties For varying morphisms m, we obtain varying pushout functors $\mathtt{Push}_m$. Coherence properties relate members of this family of functors. Consider Figure 6 for two exemplary properties that we will prove below. At the top, we illustrate that pushouts of theories X along identity morphisms yield X again. And at the bottom, we illustrate how the pushout functor distributes over morphism composition. Our results are strongly guided by and do extend the results of Rabe [Rab17a, Defs. 2.4 and 2.26] who defines the very same pushout functor in the very same setting of flat MMT theories and morphisms (modulo different notation).

We will phrase all coherence properties with two important caveats in mind. First, we state all results only on flat theories and morphisms. In particular whenever we write $\mathtt{Push}_m$ or $\mathtt{PushIn}_m$ we mean the functor or connector and never their lifting to structured theories. (Indeed, we leave supporting coherence properties on structured theories to future work, see Section 7.2.) Second, we recall that we follow Notation 22 by which all of our functors and natural transformations are assumed to be partial, and thus equalities are only required to hold whenever both sides are defined.

▶ **Theorem 48.** `Push_` *is a functor from* $\mathbb{LF}$ *to the category of theory extension categories and strongly linear functors (see Theorem 33) where*

- *on objects we set* $\mathtt{Push}_S = \mathbb{LF}^S$

[5] Objects of $c/\mathcal{C}$ are $\mathcal{C}$-morphisms $u\colon c \to x$, and morphisms $(u\colon c \to x) \to (v\colon c \to x')$ are $\mathcal{C}$-morphisms $x \to x'$ that form a commuting triangle with u and v.

$$\begin{tikzcd}
R \arrow[r, "\mathrm{id}_R"] \arrow[d, hook] & R \arrow[d, hook] \\
X \arrow[r] & \mathtt{Push}_{\mathrm{id}_R}(X) = X
\end{tikzcd}$$

$$\begin{tikzcd}
R \arrow[r, "m"] \arrow[d, hook] & S \arrow[r, "n"] \arrow[d, hook] & T \arrow[d, hook] \\
X \arrow[r] & \mathtt{Push}_m(X) \arrow[r] & \mathtt{Push}_n(\mathtt{Push}_m(X)) = \mathtt{Push}_{n \circ m}(X)
\end{tikzcd}$$

Figure 6 Exemplary Coherence Properties of `Push` on Flat Theories

- *on morphisms $m\colon S \to T$ we take $\mathtt{Push}_m$ from Definition 41*

Moreover, for a morphism $m\colon S \to T$ and two S-extensions X and Y with $X \hookrightarrow Y$ we have:

$$\mathtt{Push}_m(Y) = \mathtt{Push}_{\mathtt{PushIn}_m(X)}(Y)$$

▶ **Theorem 49.** *For two structured morphisms $R \overset{m}{\to} S \overset{n}{\to} T$, the family of connectors* `PushIn` *fulfills*

$$\mathtt{PushIn}_{\mathrm{id}_R}(X) = \mathrm{id}_X$$
$$\mathtt{PushIn}_{n \circ m}(X) = \mathtt{PushIn}_n(\mathtt{Push}_m(X)) \circ \mathtt{PushIn}_m(X)$$

and for two S-extensions X and Y with $X \hookrightarrow Y$ we have

$$\mathtt{PushIn}_m(Y) = \mathtt{PushIn}_{\mathtt{PushIn}_m(X)}(Y)$$

We recommend looking at Figure 6 while perusing Theorem 49.

Theorem 48 subsumes the identities in [Rab17a, Def. 2.4] given there on the left side in the first display math. Our theorem is stronger since it also encodes functoriality on morphisms, namely $\mathtt{Push}_{\mathrm{id}_R}(\sigma) = \sigma$ and $\mathtt{Push}_{n \circ m}(\sigma) = \mathtt{Push}_n(\mathtt{Push}_m(\sigma))$. And Theorem 49 amounts to exactly the identities in [Rab17a, Def. 2.4] given there on the right side. (In fact, we simply copied Rabe's identities into our notation since we have been unable to find a concise categorical way to phrase Theorem 49, which would somehow nicely encode the family of connectors given by `PushIn`.)

Proof of Theorem 48. To see that $\mathtt{Push}_{\mathrm{id}_S}$ is equal to the identity functor on $\mathbb{LF}^S$, note that in Definition 41 the function m^Σ collapses to the identity function on S-syntax.

For functoriality, consider two morphisms $m\colon R \to S$ and $n\colon S \to T$. We prove that $\mathtt{Push}_n \circ \mathtt{Push}_m$ and $\mathtt{Push}_{n \circ m}$ are the same functors by showing $n^{\mathtt{Push}_m(\Sigma)}(m^\Sigma(-)) = (n \circ m)^\Sigma(-)$, i.e., equality of their expression translation functions. Since both functions are compositional, it suffices to show equality on constants. For constants $c \in \Sigma$ we have $n^{\mathtt{Push}_m(\Sigma)}(m^\Sigma(c)) = n^{\mathtt{Push}_m(\Sigma)}(c) = c = (n \circ m)^\Sigma(c)$, noting that for $c \in \Sigma$ we also have $c \in \mathtt{Push}_m(\Sigma)$ by construction. And for constants $c \in R$ we have

$$\begin{array}{lll}
 & n^{\mathtt{Push}_m(\Sigma)}(m^\Sigma(c)) & \\
= & n^{\mathtt{Push}_m(\Sigma)}(m(c)) & \text{since } c \notin \Sigma \\
= & n(m(c)) & \text{since } m(c) \text{ is an } S\text{-term} \\
= & (n \circ m)(c) & \text{by definition} \\
= & n \circ m^\Sigma(c) & \text{since } c \in R
\end{array}$$

◀

Proof of Theorem 49. We use Lemma 7 for all claims. For the first equality, we verify

$$\mathtt{PushIn}_{\mathrm{id}_R}(X)(c) = \left\{\begin{matrix} \mathrm{id}_R(c) & c \in R \\ c & \text{otherwise} \end{matrix}\right\} = c = \mathrm{id}_X(c)$$

Similarly, for the second equality we verify (starting from the RHS)

$$\begin{aligned}
&(\mathtt{PushIn}_n(\mathtt{Push}_m(X)) \circ \mathtt{PushIn}_m(X))(c) \\
=&\begin{cases} \mathtt{PushIn}_n(\mathtt{Push}_m(X))(m(c)) & c \in R \\ \mathtt{PushIn}_n(\mathtt{Push}_m(X)(c)) & \text{otherwise} \end{cases} \\
=&\begin{cases} n(m(c)) & c \in R \\ c & \end{cases} \\
=&\mathtt{PushIn}_{n\circ m}(X)(c)
\end{aligned}$$

where in the third line we used that $m(c)$ is an S-term. Finally, for the third equality let $c \in Y$ be a constant and verify (starting from the RHS)

$$\begin{aligned}
&\mathtt{PushIn}_{\mathtt{PushIn}_m(X)}(Y)(c) \\
=&\begin{cases} \mathtt{PushIn}_m(X)(c) & \text{if } c \in X \\ c & \text{otherwise} \end{cases} \\
=&\begin{cases} \begin{cases} m(c) & \text{if } c \in S \\ c & \text{otherwise} \end{cases} & \text{if } c \in X \\ c & \text{otherwise} \end{cases} \\
=&\mathtt{PushIn}_m(Y)(c)
\end{aligned}$$

◀

4.2 Polymorphic Generalization

$$\begin{tikzcd}
\mathtt{Typed} \arrow[d, hook] & \mathtt{Typed} \arrow[d, hook] \\
T \arrow[r, bend left=60, "(\mathtt{Sel}^{\mathbb{T}}(T))_{\mathbb{T}:\,\mathtt{tp}}"] \arrow[r, bend left=40] \arrow[r, phantom, bend left=25, "\vdots"] \arrow[r, bend left=15] & \mathtt{Poly}(T) \arrow[l, "\mathtt{For}(T)"]
\end{tikzcd}$$

theory $\mathtt{Typed} = \{$
    $\mathtt{tp}\colon \mathtt{type}$
$\}$

The strongly linear functor `Poly` on `Typed` abstracts every constant declaration over a new type parameter $u\colon \mathtt{tp}$. Every constant $c\colon A$ over a signature Σ is mapped to $c\colon \Pi\, u\colon \mathtt{tp}.\ A^{\Sigma,u}$, where $-^{\Sigma,u}$ is the function that replaces every reference to a constant $c' \in \Sigma$ by $c'\ u$. Thus, all concepts represented in a theory T become *polymorphified* in $\mathtt{Poly}_{\Sigma}(T)$. This is similar to how in Kripke semantics the inference rules for all logic connectives but the modalities are pointwise defined for every world, i.e., polymorphic in the world. The connector `For` maps every T-model to the $\mathtt{Poly}(T)$-model with constant polymorphism, i.e., where polymorphic parameters are forgotten, making every polymorphic constant effectively monomorphic. And for every $\mathbb{T}\colon \texttt{tp}$, the connector $\texttt{Sel}^{\mathbb{T}}$ maps every $\texttt{Poly}(T)$-model to the monomorphic T-model by selecting the monomorphism at type $\mathbb{T}\colon \texttt{tp}$.

To the best of our knowledge, the polymorphify operator, while folklore in principle, is formalized here for the first time (except of course the original publication [RR21c] on which this thesis is based).

Let us look at an example before spelling out formal details. We can obtain a large class of examples by special-casing to `SPoly`, the strongly linear functor on `SFOL` that acts like `Poly` except that it is the identity on `SFOL`. Applying the preliminary definition above, we see that `SPoly` maps SFOL-declarations[6] as follows:

- Every type symbol $T\colon \texttt{tp}$ yields a unary type operator $T\colon \Pi\, u\colon \texttt{tp}.\ \texttt{tp}$ (i.e., $T\colon \texttt{tp} \to \texttt{tp}$).
- Every function symbol $f\colon \texttt{tm}\ T_1 \to \ldots \to \texttt{tm}\ T_n \to \texttt{tm}\ T$ yields a polymorphic function symbol.
- Every predicate symbol is mapped in esentially the same way as function symbols.
- Every axiom symbol $ax\colon \Vdash F$ yields an axiom $ax\colon \Vdash \Pi\, u\colon \texttt{tp}.\ F^{\Sigma,u}$ where the operation $-^{\Sigma,u}$ now affects all type, function, and predicate symbols occuring in F.

▶ **Example 50** (`SPoly(Monoid)` is the Theory of Lists)**.** Recall the SFOL-theory `Monoid` from Example 3. Applying `SPoly` yields:

$$
\begin{array}{l}
\textbf{theory}\ \texttt{SPoly}(\texttt{Monoid}) = \{\\
\quad \textbf{include}\ \texttt{SFOL}\\
\quad \begin{array}{ll}
U & \colon \Pi\, u.\ \texttt{tp}\\
\circ & \colon \Pi\, u.\ \texttt{tm}\ U\, u \to \texttt{tm}\ U\, u \to \texttt{tm}\ U\, u\\
e & \colon \Pi\, u.\ \texttt{tm}\ U\, u\\
\texttt{assoc} & \colon \Pi\, u.\ \ \Vdash \forall\, x\ y\ z\colon \texttt{tm}\ U\, u.\ (x \circ^u y) \circ^u z \doteq x \circ^u (y \circ^u z)\\
\texttt{neut} & \colon \Pi\, u.\ \ \Vdash \forall\, x\colon \texttt{tm}\ U\, u.\ e^u \circ^u x \doteq x
\end{array}\\
\}
\end{array}
$$

Above, we denoted application of function symbols (such as $\circ$ and e) to u by superscripts to enhance readability, e.g., we write e^u to mean $e\, u$.

The theory `SPoly(Monoid)` looks close to the theory of polymorphic lists with $U\ u$ for the type of lists over u and $\circ$ and e for concatenation and empty list. Similarly, magmas, commutative monoids, and commutative-idempotent monoids yield theories close to the collection data types for trees, multisets, and sets. In all of these cases, the only thing missing to complete the specification of the collection data types is an operator $\texttt{single}\colon \Pi\, u.\ \texttt{tm}\ u \to \texttt{tm}\ U\, u$ for creating singleton trees, lists, and sets. We can easily adjoin such a constant using the pushout functor from Section 4.1, and we come back to this in Section 4.2.2.

4.2.1 Definition

We now head to formally defining `Poly`. Above, we carelessly said to map every $c\colon A$ to $c\colon \Pi\, u\colon \texttt{tp}.\ c^{\Sigma,u}$ where $c^{\Sigma,u}$ emerges from c by replacing every reference to $c' \in \Sigma$ by $c'\ u$. In reality, we have to ensure that the u in $c'\ u$ really refers to the u as bound by the outermost $\Pi u\colon \texttt{tp}$ that we introduced. To prevent otherwise occurring nameclashes, we have to choose the identifier u such that it is fresh among all free *and* bound variables. Therefore, we define:

[6] See Definition 98 for details on SFOL patterns.

▶ **Definition 51** (All Variables). *Define the set-valued function* $\mathrm{AV}(\cdot)$ *on* MMT *syntax (terms and contexts):*

$$\mathrm{AV}(c) = \mathrm{AV}(\mathtt{type}) = \mathrm{AV}(\mathtt{kind}) = \varnothing$$
$$\mathrm{AV}(x) = \{x\} \qquad \mathrm{AV}(f\ t) = \mathrm{AV}(f) \cup \mathrm{AV}(t)$$
$$\mathrm{AV}(\Pi\, x\colon A.\ B) = \{x\} \cup \mathrm{AV}(A) \cup \mathrm{AV}(B) \qquad \mathrm{AV}(\lambda x\colon A.\ t) = \{x\} \cup \mathrm{AV}(A) \cup \mathrm{AV}(t)$$
$$\mathrm{AV}(\cdot) = \varnothing \qquad \mathrm{AV}(\Gamma, x\colon A) = \mathrm{AV}(\Gamma) \cup \{x\} \cup \mathrm{AV}(A)$$

We call a term t (analouglsy: a context Γ) AV*-fresh wrt. an identifier u if $u \notin \mathrm{AV}(t)$.*

▶ **Definition 52** (Constant Indexing). *Let Σ be a flat theory and u an identifier. We define the function $-^{\Sigma,u}$ on terms and contexts that are* AV*-fresh wrt. u as follows:*

$$c^{\Sigma,u} = \begin{cases} c\,u & \text{if } c \in \Sigma \\ c & \text{else} \end{cases}$$

$$\mathtt{type}^{\Sigma,u} = \mathtt{type} \qquad \mathtt{kind}^{\Sigma,u} = \mathtt{kind}$$
$$x^{\Sigma,u} = x \qquad (f\ t)^{\Sigma,u} = f^{\Sigma,u}\ t^{\Sigma,u}$$
$$(\lambda x\colon A.\ t)^{\Sigma,u} = \lambda x\colon A^{\Sigma,u}.\ t^{\Sigma,u} \qquad (\Pi\, x\colon A.\ B)^{\Sigma,u} = \Pi\, x\colon A^{\Sigma,u}.\ B^{\Sigma,u}$$
$$\cdot^{\Sigma,u} = \cdot \qquad (\Gamma, x\colon A)^{\Sigma,u} = \Gamma^{\Sigma,u}, x\colon A^{\Sigma,u}$$

▶ **Definition 53** (`Poly`). *The strongly linear functor* `Poly` *on* `Typed` *is given by expression translation functions*

$$E_\Sigma(A) = \Pi\, u\colon \mathtt{tp}.\ A^{\Sigma,u} \qquad e_\Sigma(t) = \lambda u\colon \mathtt{tp}.\ t^{\Sigma,u}$$

where in both definitions, respectively, the identifier u is chosen adaptively such that A and t are AV*-fresh wrt. u.*

We might be tempted to think that the adaptive choice of the identifier u introduces some kind of "non-uniformity" and thus breaks key properties of `Poly` such as well-typedness or functoriality, or works unpredicatbly from a user's perspective. Importantly, this is not the case. By α-equivalence built into the MMT/LF calculus, choices of variable identifiers (such as u) are completely transparent to formalism and end users. The only way variable identifiers can be observed is by inspection of syntax trees output by applications of `Poly`; in particular when humans read generated source code. Moreover, note that even the occurrence of u as a constant identifier in Σ is harmless since the MMT/LF syntax distinguishes between references to constants and variables anyway.

We now define the connectors into and out of `Poly` illustrated in Figure 7.

▶ **Definition 54** (Forgetful Connector out of `Poly`). *The strongly linear connector* `For`: `Poly` $\to$ `Id` *is given by*

$$\mathtt{For}_\Sigma(c\colon A) = \lambda\_\colon \mathtt{tp}.\ c$$

▶ **Definition 55** (Selective Connectors into `Poly`). *For a fixed type $\vdash^{\mathsf{Typed}} \mathbb{T}\colon \mathtt{tp}$, the strongly linear connector* $\mathtt{Sel}^{\mathbb{T}}\colon \mathtt{Id} \to \mathtt{Poly}$ *is given by*

$$\mathtt{Sel}^{\mathbb{T}}_\Sigma(c\colon A) = c\ \mathbb{T}$$

$$\begin{tikzcd}
\texttt{Typed} \arrow[d, hook] & \texttt{Typed} \arrow[d, hook] \\
T \arrow[r, bend left=60, "(\texttt{Sel}^{\mathbb{T}}(T))_{\mathbb{T}\colon \texttt{tp}}"] \arrow[r, bend left=40] \arrow[r, bend left=20] \arrow[r, phantom, bend left=40, "\vdots"] & \texttt{Poly}(T) \arrow[l, "\texttt{For}(T)"]
\end{tikzcd}$$

Figure 7 Connectors into and out of `Poly`

$$\begin{tikzcd}
\texttt{Set} \arrow[r, hook] & \texttt{Magma} \arrow[r, hook] & \texttt{Monoid} \arrow[r, hook] & \texttt{CommIdemMonoid} \\
\texttt{SPoly(Set)} \arrow[r, hook] \arrow[d, hook] & \texttt{SPoly(Magma)} \arrow[r, hook] \arrow[d] & \texttt{SPoly(Monoid)} \arrow[r, hook] \arrow[d] & \texttt{SPoly(CommIdemMonoid)} \arrow[d] \\
\texttt{Singleton} \arrow[r, hook] & \texttt{Trees} \arrow[r, hook] & \texttt{Lists} \arrow[r, hook] & \texttt{Sets}
\end{tikzcd}$$

Figure 8 Recovering collection data types: diagrams `Magmas` and results of applying `SPoly` and `Push`

To get a feel for Definitions 54 and 55, consider a `Typed`-extension S and a constant $c\colon A$ declared in S. By means of `Poly` this constant is mapped to $c\colon \Pi\, u\colon \texttt{tp}.\ A^{\Sigma,u}$ in $\texttt{Poly}(S)$. And the morphism $\texttt{For}(S)\colon \texttt{Poly}(S) \to S$ forgets the polymorphism and interprets the family of constructors represented by c in $\texttt{Poly}(S)$ as the constant family $\lambda\_\colon \texttt{tp}.\ c$ in S. This way, `For` maps every S-model to the $\texttt{Poly}(S)$-model with constant polymorphism, i.e., where polymorphic parameters are forgotten, making every polymorphic constant effectively monomorphic.

In the other direction, consider additionally a fixed type $\vdash_S^{\texttt{Typed}} \mathbb{T}\colon \texttt{tp}$. The morphism $\texttt{Sel}^{\mathbb{T}}(S)\colon S \to \texttt{Poly}_{\Sigma}(S)$ maps the constant c in S to the family member $c\ \mathbb{T}$ at index $\mathbb{T}$ of the family of constructors represented by c in $\texttt{Poly}_{\Sigma}(S)$ at index $\mathbb{T}$. This way, $\texttt{Sel}^{\mathbb{T}}$ maps every $\texttt{Poly}(S)$-model to the monomorphic S-model by selecting the monomorphism at type $\mathbb{T}\colon \texttt{tp}$.

Note that formally there is not even a single type $\vdash_{\Sigma}^{\texttt{Typed}} \mathbb{T}\colon \texttt{tp}$ that one can construct. However, as stated below, all results of `Poly` that we later collect in Section 4.2.4 carry over to the case of functors that act like `Poly`, but are the identity on some fixed E for all `Typed`-extensions E.

▶ **Theorem 56.** *Consider a* `Typed`*-extension E. All results collected in Section 4.2.4 seamlessly carry over to the functor $E\texttt{Poly}_{\Sigma}$ that acts like* `Poly`*, but is the identity on E.*

Proof. The only assumption that all proofs in Section 4.2.4 make is the existence of the constant `tp`: `type` in `Typed`. Thus, enlarging the functor's (co)domain to a `Typed`-extension E preserves the validity of all proofs. ◀

4.2.2 Application to Algebraic Hierarchy: Recovering Collection Data Types

$$\begin{tikzcd}
\texttt{SFOL} \arrow[d, hook] & & \texttt{SFOL} \arrow[d, hook] \\
T \arrow[rr, bend left=60, "(\texttt{Sel}^{\mathbb{T}}(T))_{\mathbb{T}:\,\texttt{tp}}"] \arrow[rr, bend left=45] \arrow[rr, phantom, bend left=30, "\vdots"] \arrow[rr, bend left=20] & & \texttt{SPoly}(T) \arrow[ll, "\texttt{For}(T)"]
\end{tikzcd}$$

Recall Example 50, where we applied SPoly to the theory Monoid and obtained a theory close to the theory of polymorphic lists. We now continue this example and recover formalizations of collection data types by applying functors SPoly and the pushout functor Push (from Section 4.1) to the algebraic hierarchy. In principle, the similarity that we exploit between well-known algebra theories within mathematics and well-known data structures within computer science is folklore [Bun93]. However, to the best of our knowledge, our methodology of transforming the former for obtaining the latter is novel, and the closest work we are aware of is an unpublished draft [CAK17, Part IV], where this transformation is carried out manually (i.e., without applying meta-programs as we do).

Consider the diagram Magmas depicted in the first row of Figure 8, which modifies and extends the previous example to a small hierarchy of algebra theories. To be self-contained, we show the complete source in Figure 9. Applying SPoly to Magmas yields the diagram in the second row of Figure 8, whose source is shown in Figure 10. We observe that SPoly(Magma) is a formalization close to the one of trees, SPoly(Monoid) to the one of lists (as discussed in Example 50), and SPoly(CommIdemMonoid) to the one of sets.

In every case, the missing ingredient to SPoly(T) for $T \in \{\texttt{Magma}, \texttt{Monoid}, \texttt{CommIdemMonoid}\}$ to become a complete specification of the collection datatype of trees, lists, and sets, respectively, is an operator for creating singleton trees, lists, and sets. We can uniformly inject it into all theories in the diagram by defining the theory

$$\textbf{theory}\ \texttt{Singleton} = \{\textbf{include}\ \texttt{SPoly}(\texttt{Set}),\ \texttt{singleton}\colon \Pi u\colon \texttt{tp}.\ \texttt{tm}\ u \to \texttt{tm}\ U\,u\}$$

and then applying the pushout functor $\texttt{Push}_i$ along the inclusion morphism $i\colon \texttt{SPoly}(\texttt{Set}) \hookrightarrow \texttt{Singleton}$. The result is shown in the third row of Figure 8.

Boom Hierarchy So far, we have only looked at a small, fixed part of the algebraic hierarchy. We now extend our picture and systematically apply our methodology to a much more fine-grained algebraic hierarchy. Thereby, we will be able to cherry-pick data structures by combining abstract mathematical properties. For a theory S extending Magma (given in Figure 9) consider the following mathematical properties and corresponding MMT formalizations:

Property	Representation as constants extending Magma	
unital	e	$: \texttt{tm}\ U$
	neut	$: \Vdash \forall x.\ (e \circ x \doteq x) \wedge (x \circ e \doteq x)$
commutativity	comm	$: \forall x\ y.\ x \circ y \doteq y \circ x$
associativity	assoc	$: \Vdash \forall x\ y\ z.\ (x \circ y) \circ z \doteq x \circ (y \circ z)$
idempotence	idem	$: \Vdash \forall x.\ x \circ x \doteq x$
invertibility (extending unital)	inv	$: \texttt{tm}\ U \to \texttt{tm}\ U$
	invax	$: \Vdash \forall x.\ (x \circ x^{-1} \doteq e) \wedge (x^{-1} \circ x \doteq e)$

$$
\begin{array}{l}
\textbf{theory}\ \texttt{Set} = \{\\
\quad \textbf{include}\ \texttt{SFOL}\\
\quad U\colon \texttt{tp}\\
\}
\end{array}
$$

$$
\begin{array}{l}
\textbf{theory}\ \texttt{Monoid} = \{\\
\quad \textbf{include}\ \texttt{Set}\\
\quad e \qquad\ \colon \texttt{tm}\ U\\
\quad \texttt{assoc}\colon \Vdash \forall\, x\ y\ z\colon \texttt{tm}\ U.\ (x \circ y) \circ z \doteq x \circ (y \circ z)\\
\quad \texttt{neut}\ \ \colon \Vdash \forall\, x\colon \texttt{tm}\ U.\ e \circ x \doteq x\\
\}
\end{array}
$$

$$
\begin{array}{l}
\textbf{theory}\ \texttt{Magma} = \{\\
\quad \textbf{include}\ \texttt{Set}\\
\quad \circ\colon \texttt{tm}\ U \to \texttt{tm}\ U \to \texttt{tm}\ U\\
\}
\end{array}
$$

$$
\begin{array}{l}
\textbf{theory}\ \texttt{CommIdemMonoid} = \{\\
\quad \textbf{include}\ \texttt{Monoid}\\
\quad \texttt{comm}\colon \Vdash \forall\, x\ y\colon \texttt{tm}\ U.\ x \circ y \doteq y \circ x\\
\quad \texttt{idem}\colon \Vdash \forall\, x.\ x \circ x \doteq x\\
\}
\end{array}
$$

$$\texttt{Magmas} = \mathit{diag}(\texttt{Set}, \texttt{Magma}, \texttt{Monoid}, \texttt{CommIdemMonoid})$$

Figure 9 Source of small algebraic hierarchy for Figure 8

$$
\begin{array}{l}
\textbf{theory}\ \texttt{SPoly}(\texttt{Set}) = \{\\
\quad U\colon \texttt{tp} \to \texttt{tp}\\
\}
\end{array}
$$

$$
\begin{array}{l}
\textbf{theory}\ \texttt{SPoly}(\texttt{Monoid}) = \{\\
\quad \textbf{include}\ \texttt{SPoly}(\texttt{Magma})\\
\quad e \qquad\ \colon \Pi\, u.\ \texttt{tm}\ U\ u\\
\quad \texttt{assoc}\colon \Pi\, u.\ \ \Vdash \forall\, x\ y\ z\colon \texttt{tm}\ U\ u.\\
\qquad\qquad\qquad (x \circ^u y) \circ^u z \doteq x \circ^u (y \circ^u z)\\
\quad \texttt{neut}\ \ \colon \Pi\, u.\ \ \Vdash \forall\, x\colon \texttt{tm}\ U\ u.\ e^u \circ^u x \doteq x\\
\}
\end{array}
$$

$$
\begin{array}{l}
\textbf{theory}\ \texttt{SPoly}(\texttt{Magma}) = \{\\
\quad \textbf{include}\ \texttt{SPoly}(\texttt{Set})\\
\quad \circ\colon \Pi\, u\colon \texttt{tp}.\ \texttt{tm}\ U\ u \to \texttt{tm}\ U\ u \to \texttt{tm}\ U\ u\\
\}
\end{array}
$$

$$
\begin{array}{l}
\textbf{theory}\ \texttt{SPoly}(\texttt{CommIdemMonoid}) = \{\\
\quad \textbf{include}\ \texttt{SPoly}(\texttt{Monoid})\\
\quad \texttt{comm}\colon \Pi\, u.\ \ \Vdash \forall\, x\ y\colon \texttt{tm}\ U\ u.\ x \circ^u y \doteq y \circ^u x\\
\quad \texttt{idem}\colon \Pi\, u.\ \ \Vdash \forall\, x\colon \texttt{tm}\ U\ u.\ x \circ^u x \doteq x\\
\}
\end{array}
$$

Figure 10 Theories of $\texttt{SPoly}(\texttt{Magmas})$, the diagram shown in the second row of **??**

Combinations of those properties spawn a systematic algebraic hierarchy extending `Magma`. And every combination, qua our methodology of applying `SPoly` and `Push`, directly corresponds to a data structure. We show this correspondence in Figure 11. Note that while some of the terminology for the properties on the data structure (e.g., *empty*, *(un)ordered*, or *sequential*) are standard, we also had to make up some new terminology (all other ones in Figure 11). The hierarchy that is spawned with the first four properties is known as the *Boom Hierarchy* [Bun93].

	Datastructure property	
Property ϕ on Binary operation	in presence of ϕ	in absence of ϕ
unital	possibly empty	non-empty
commutativity	unordered	ordered
associativity	sequential	branched
idempotence	single countable	multi countable
invertibility	positive & negative countable	positive countable

Figure 11 Correspondence between abstract properties on a binary operation and properties on data structure

For example, we can obtain the more or lesser known data structures compiled in the table below. We only list few examples of the larger space of data structures induced by our five properties.

Data structure	unital	commutativity	associativity	idempotence	invertibility
lists			x		
non-empty lists	x		x		
trees					
non-empty unordered trees	x	x			
sets		x	x	x	
multisets w/ multiplicities in $\mathbb{N}_{\geq 0}$		x	x	x	
multisets w/ multiplicities in $\mathbb{Z}$		x	x		x
multisets w/ multiplicities in $\{-1, 0, 1\}$		x	x	x	x
non-empty multisets w/ multiplicities in $\{-1, 0, 1\}$	x	x	x	x	x

4.2.3 Application to FOL Formalizations: Recovering SFOL Formalizations

$$\begin{tikzcd}
\texttt{UntypedLogic} \arrow[d, hook] & & \texttt{TypedLogic} \arrow[d, hook] \\
T \arrow[rr, bend left=40, "(\texttt{Sel}^{\mathbb{T}}(T))_{\mathbb{T}:\,\texttt{tp}}"] \arrow[rr, bend left=30] \arrow[rr, bend left=20, phantom, "\vdots"] \arrow[rr, bend left=15] & & \texttt{Typify}(T) \arrow[ll, "\texttt{For}(T)"]
\end{tikzcd}$$

theory UntypedLogic $= \{$

  include PL

  term: type

$\}$

theory TypedLogic $= \{$

  include PL

  tp: type

  tm: tp $\to$ type

$\}$

We suitably modify Poly to a functor Typify that applied to formalizations *of* (not in) first-order logic, such as quantifiers and proof rules, yields corresponding formalizations of *sorted* first-order logic. For example, to represent the universal quantifier of FOL, we extend UntypedLogic shown above with the constant $\forall\colon (\texttt{term} \to \texttt{prop}) \to \texttt{prop}$. Correspondingly, to represent the same concept in SFOL, we extend TypedLogic shown above with the constant $\forall\colon \Pi T : \texttt{tp}.\ (\texttt{tm}\ T \to \texttt{prop}) \to \texttt{prop}$. We observe that the latter emerges from the former by *i*) abstracting over a type $T\colon \texttt{tp}$ and *ii*) rewriting occurrences of term to $\texttt{tm}\ T$. By consistently extending these rules to structured diagrams qua our framework, we can recover large parts of usual formalizations of SFOL from the untyped case, In practice this means we can save those formalizations and thus maintenance effort, and also eliminate redundancy.

▶ **Definition 57** (Typify)**.** *The strongly linear functor* Typify *from* UntypedLogic *to* TypedLogic *is given by expression translation functions*

$$E_\Sigma(A) = \Pi\, u\colon \texttt{tp}.\ A^{\Sigma,u} \qquad e_\Sigma(t) = \lambda u\colon \texttt{tp}.\ t^{\Sigma,u}$$

where in both definitions, respectively, the identifier u is chosen adaptively such that A and t are AV*-fresh wrt. u, and where* $-^{\Sigma,u}$ *is the function taken from Definition 52 modified at the case for constants by*

$$c^{\Sigma,u} = \begin{cases} c\, u & \textit{if } c \in \Sigma \\ \texttt{tm}\ u & \textit{if } c = \texttt{term} \\ c & \textit{else} \end{cases}$$

Within this Section 4.2.3, the notation $-^{\Sigma,u}$ *will always refer to the function above.*

As an example, consider the modular formalization hierarchies of quantifiers and natural deduction rules shown in Figure 12. On the left, we show the hierarchy for FOL, and on the right we show it for SFOL. For ease of readability, we choose short names including greek characters for theories, and use primed names to denote corresponding SFOL concepts.[7] We show excerpts of the formalizations in Figure 13. There, we observe that applying Typify on the universal quantifier and its natural deduction rules indeed yields the corresponding ones for SFOL. This holds true for all of Figure 12

$$\textbf{diagram}\ \texttt{FOLDiag} = \text{Diagram}(\forall, \forall I, \forall E, \forall\texttt{ND}, \texttt{FOLND})$$

[7] In contrast, e.g. LATIN2 in practice uses names like UniversalQuantifierND for FOL and TypedUniversalQuantifierND for SFOL.

$$\begin{tikzcd}[column sep=small]
 & & & \mathtt{PL} \arrow[d, hook] & & & \\
 & & & \mathtt{UntypedLogic} \arrow[dll, hook] \arrow[drr, hook] & & & \\
 & \forall \arrow[dl, hook] \arrow[dr, hook] & & & & \exists \arrow[dl, hook] \arrow[dr, hook] & \\
\forall I \arrow[dr, hook] & & \forall E \arrow[dl, hook] & & \exists I \arrow[dr, hook] & & \exists E \arrow[dl, hook] \\
 & \forall\mathtt{ND} \arrow[drr, hook] & & & & \exists\mathtt{ND} \arrow[dll, hook] & \\
 & & & \mathtt{FOLND} & & &
\end{tikzcd}$$

$$\begin{tikzcd}[column sep=small]
 & & & \mathtt{PL} \arrow[d, hook] & & & \\
 & & & \mathtt{TypedLogic} \arrow[dll, hook] \arrow[drr, hook] & & & \\
 & \forall' \arrow[dl, hook] \arrow[dr, hook] & & & & \exists' \arrow[dl, hook] \arrow[dr, hook] & \\
\forall I' \arrow[dr, hook] & & \forall E' \arrow[dl, hook] & & \exists I' \arrow[dr, hook] & & \exists E' \arrow[dl, hook] \\
 & \forall\mathtt{ND}' \arrow[drr, hook] & & & & \exists\mathtt{ND}' \arrow[dll, hook] & \\
 & & & \mathtt{SFOLND} & & &
\end{tikzcd}$$

Figure 12 Formalization Hierarchy of FOL quantifiers and proof rules (left) and analogously for SFOL (right)

$\mathbf{theory}\ \mathtt{UniversalQuantifier} = \{$
  $\mathbf{include}\ \mathtt{UntypedLogic}$
  $\forall\colon (\mathtt{term} \to \mathtt{prop}) \to \mathtt{prop}$
$\}$

$\mathbf{theory}\ \mathtt{UniversalQuantifier}' = \{$
  $\mathbf{include}\ \mathtt{TypedLogic}$
  $\forall\colon \Pi\, T\colon \mathtt{tp}.\ (\mathtt{tm}\ T \to \mathtt{prop}) \to \mathtt{prop}$
$\}$

$\mathbf{theory}\ \mathtt{UniversalQuantifierND} = \{$
  $\mathbf{include}\ \mathtt{UniversalQuantifier}$
  $\forall\mathtt{I}\colon \Pi\, P\colon \mathtt{term} \to \mathtt{prop}.$
    $(\Pi\, x\colon \mathtt{term}.\ \Vdash P\ x) \to \Vdash \forall P$
  $\forall\mathtt{E}\colon \Pi\, P\colon \mathtt{term} \to \mathtt{prop}.$
    $\Vdash \forall P \to \Pi\, x\colon \mathtt{term}.\ \Vdash P\ x$
$\}$

$\mathbf{theory}\ \mathtt{UniversalQuantifierND}' = \{$
  $\mathbf{include}\ \mathtt{UniversalQuantifier}'$
  $\forall\mathtt{I}\colon \Pi\, T\colon \mathtt{tp}.\ \Pi\, P\colon \mathtt{tm}\ T \to \mathtt{prop}.$
    $(\Pi\, x\colon \mathtt{tm}\ T.\ \Vdash P\ x) \to \Vdash \forall\ T\ P$
  $\forall\mathtt{E}\colon \Pi\, T\colon \mathtt{tp}.\ \Pi\, P\colon \mathtt{tm}\ T \to \mathtt{prop}.$
    $\Vdash \forall\ T\ P \to \Pi\, x\colon \mathtt{tm}\ T.\ \Vdash P\ x$
$\}$

Figure 13 Exemplary Formalizations of the Universal Quantifier for FOL and SFOL

to then replace the *entire* right diagram in Figure 12 by *a single line* invoking our functor:

$$\mathbf{install}\ \mathtt{Typify}_{\Sigma} * (\mathtt{FOLDiag})$$

The previous example was deliberately small to get across the idea of how `Typify` works on diagrams. As a real-world example, we now consider LATIN2 [LATIN], the successor of the LATIN Logic Atlas and Integrator project [Cod+a]. We have verified `Typify` to work as intended on all FOL concepts listed below, which were all taken from LATIN2 (with notation adapted)[8]

universal quantifier

$$
\begin{array}{ll}
\forall & : (\mathtt{term} \to \mathtt{prop}) \to \mathtt{prop} \\
\forall\mathtt{I} & : \Pi\, p\colon \mathtt{term} \to \mathtt{prop}.\ (\Pi\, x\colon \mathtt{term}.\ \Vdash p\ x) \to \Vdash \forall p \\
\forall\mathtt{E} & : \Pi\, p\colon \mathtt{term} \to \mathtt{prop}.\ \Vdash \forall p \to \Pi\, x\colon \mathtt{term}.\ \Vdash p\ x
\end{array}
$$

existential quantifier

$$
\begin{array}{ll}
\exists & : (\mathtt{term} \to \mathtt{prop}) \to \mathtt{prop} \\
\exists I & : \Pi\, p\colon \mathtt{term} \to \mathtt{prop}.\ \Pi\, x\colon \mathtt{term}.\ \Vdash p\ x \to \Vdash \exists p \\
\exists E & : \Pi\, p\colon \mathtt{term} \to \mathtt{prop}.\ \Pi\, F\colon \mathtt{prop}.\ \Vdash \exists p \to \\
 & \quad (\Pi\, x\colon \mathtt{term}.\ \Vdash p\ x \to \Vdash F) \to \Vdash F
\end{array}
$$

unique existential quantifier

$$
\begin{array}{ll}
\exists! & : (\mathtt{term} \to \mathtt{prop}) \to \mathtt{prop} \\
\exists! I & : \Pi\, p\colon \mathtt{term} \to \mathtt{prop}.\ \Pi\, x\colon \mathtt{term}.\ \Vdash p\ x \to (\Pi\, y\colon \mathtt{term}.\ \Vdash p\ y \to \Vdash x \doteq y) \to \exists!\, p \\
\exists! E & : \Pi\, p\colon \mathtt{term} \to \mathtt{prop}.\ \Pi\, F\colon \mathtt{prop}.\ \Vdash \exists p \to \\
 & \quad (\Pi\, x\colon \mathtt{term}.\ \Vdash p\ x \to (\Pi\, y\colon \mathtt{term}.\ \Vdash p\ y \to \Vdash x \doteq y) \to \Vdash F) \to \Vdash F
\end{array}
$$

equality

$$
\begin{array}{ll}
\doteq & : \mathtt{term} \to \mathtt{term} \to \mathtt{prop} \\
\mathtt{refl} & : \Pi\, x\colon \mathtt{term}.\ \Vdash x \doteq x \\
\mathtt{cong} & : \Pi\, x\ y\colon \mathtt{term}.\ \Vdash x \doteq y \to \Pi\, p\colon \mathtt{term} \to \mathtt{prop}.\ \Vdash p\ x \to \Vdash p\ y
\end{array}
$$

description operator

$$
\begin{array}{ll}
\mathtt{the} & : \Pi\, p\colon \mathtt{term} \to \mathtt{prop}.\ \exists!\, p \to \mathtt{term} \\
\mathtt{the\_ax} & : \Pi\, p\colon \mathtt{term} \to \mathtt{prop}.\ \Pi\, pf\colon\ \Vdash \exists!\, p.\ \Vdash p\ (\mathtt{the}\ p\ pf)
\end{array}
$$

choice operator

$$
\begin{array}{ll}
\mathtt{some} & : \Pi\, p\colon \mathtt{term} \to \mathtt{prop}.\ \exists p \to \mathtt{term} \\
\mathtt{some\_ax} & : \Pi\, p\colon \mathtt{term} \to \mathtt{prop}.\ \Pi\, pf\colon\ \Vdash \exists p.\ \Vdash p\ (\mathtt{some}\ p\ pf)
\end{array}
$$

Limitations `Typify` introduces *exactly one* type variable $u\colon \mathtt{tp}$ and rewrites all occurrences of `term` to `tm` u with that very type variable. This is not always the right choice. Consider the constants below, taken from LATIN2's FOL formalization.[9] They serve as shorthands for universally quantified binary predicates and their corresponding proof rules.

$$
\begin{array}{ll}
\forall_2 & : (\mathtt{term} \to \mathtt{term} \to \mathtt{prop}) \to \mathtt{prop} \\
 & \quad = \lambda p.\ \forall\, x.\ \forall\, y.\ p\ x\ y \\
\forall_2 I & : \Pi\, p\colon \mathtt{term} \to \mathtt{prop}.\ (\Pi\, x\ y\colon \mathtt{term}.\ \Vdash p\ x\ y) \to \Vdash \forall_2 p \\
 & \quad = \lambda p\ pf.\ \forall\mathtt{I}_x\ \forall\mathtt{I}_y\ pf\ x\ y \\
\forall_2 E & : \Pi\, p\colon \mathtt{term} \to \mathtt{prop}.\ \Pi\, x\ y\colon \mathtt{term}.\ \Vdash \forall_2 p \to \Vdash p\ x\ y \\
 & \quad = \lambda p\ x\ y\ pf.\ (pf\ \ \forall\mathtt{E}\ \ x)\ \ \forall\mathtt{E}\ \ y
\end{array}
$$

While those declarations look rather innocent, `Typify` fails to translate them to constants representing the corresponding SFOL concepts. The desired formalization of $\forall_2'$ is:

$$\forall_2'\colon \Pi\, u\ v\colon \mathtt{tp}.\ (\mathtt{tm}\ u \to \mathtt{tm}\ v \to \mathtt{prop}) \to \mathtt{prop}$$

[8] The concrete sources are https://gl.mathhub.info/MMT/LATIN2/-/blob/39dc7046f457ff02f695387a8ebd80366789a465/source/logic/fol_like/fol.mmt and https://gl.mathhub.info/MMT/LATIN2/-/blob/39dc7046f457ff02f695387a8ebd80366789a465/source/fundamentals/equality.mmt#L13-15.

[9] https://gl.mathhub.info/MMT/LATIN2/-/blob/39dc7046f457ff02f695387a8ebd80366789a465/source/logic/fol_like/fol_derived.mmt#L10-18

Importantly, it takes two type parameters u and v, while `Typify` would have only introduced one type paramter.

It is non-trivial to detect when different occurrences of `term` are *semantically* independent and should consequently be translated to occurences of $\texttt{tm}\ u$ for different type variables u. For example, in the following theorem the occurrences are *syntactically* independent, yet not semantically so. Thus the SFOL variant must use one type variable only.

$$\begin{array}{ll} c \text{ in FOL}: & \Vdash \forall x\colon \texttt{term}.\ \texttt{false} \Rightarrow \neg\exists x\colon \texttt{term}.\ \texttt{true} \\ c \text{ in SFOL}: & \Pi u\colon \texttt{tp}.\ \Vdash \forall x\colon \texttt{tm}\ u.\ \texttt{false} \Rightarrow \neg\exists x\colon \texttt{tm}\ u.\ \texttt{true} \end{array}$$

Even if there was a clever way for detecting such dependencies between occurrences of `term` (via syntax or typing), this would constitute a non-local operation; and non-local operations are generally incompatiable with the notion of a functor. To see this, suppose we had a theory S with a constant $c\colon A$. Even if we reliably detected dependencies between occurences of `term` within A, any morphism out of S could post-hoc introduce such dependencies in the assignment to c. Hence, the diagram operator would either need to scan a-priori the input diagram for such dependencies (which is incompatible with the notion of a functor), or the functor would need to be left undefined on such morphisms.

Overall, while `Typify` automates large parts of formalizing SFOL concepts, there remain certain concepts that still need to be given by hand for both FOL and SFOL.

Meta-Theoretical Properties We conclude our case study of `Typify` with its meta-theoretical properties, much of which it inherits from `Poly`.

▶ **Lemma 58.** *The function* $-^{\Sigma,u}$ *commutes with substitution:*

$$s^{\Sigma,u}[x \mapsto t^{\Sigma,u}] = (s[x \mapsto t])^{\Sigma,u}$$

Proof. Analogous to Lemma 61 for `Poly`. ◀

▶ **Theorem 59** (Well-Typedness of Constant Indexing)**.** *Let u be an identifier. Then we have for all terms t and A and contexts Γ that are* AV*-fresh wrt. u:*

$$\begin{array}{rcl} \Gamma \vdash^{\texttt{Untyped}}_{\Sigma} A\colon \texttt{kind} & \Longrightarrow & u\colon \texttt{tp}, \Gamma^{\Sigma,u} \vdash^{\texttt{Typed}}_{\texttt{Typify}(\Sigma)} A^{\Sigma,u}\colon \texttt{kind} \\ \Gamma \vdash^{\texttt{Untyped}}_{\Sigma} A\colon \texttt{type} & \Longrightarrow & u\colon \texttt{tp}, \Gamma^{\Sigma,u} \vdash^{\texttt{Typed}}_{\texttt{Typify}(\Sigma)} A^{\Sigma,u}\colon \texttt{type} \\ \Gamma \vdash^{\texttt{Untyped}}_{\Sigma} t\colon A & \Longrightarrow & u\colon \texttt{tp}, \Gamma^{\Sigma,u} \vdash^{\texttt{Typed}}_{\texttt{Typify}(\Sigma)} t^{\Sigma,u}\colon A^{\Sigma,u} \end{array}$$

Proof. Analogous to Theorem 62 with one case added in the induction corresponding to the claim on the second line: for $\Gamma \vdash^{\texttt{Untyped}}_{\Sigma} \texttt{term}\colon \texttt{type}$ we note $\texttt{term}^{\Sigma,u} = \texttt{tm}\ u$ and thus $u\colon \texttt{tp}, \Gamma^{\Sigma,u} \vdash^{\texttt{Typed}}_{\Sigma} \texttt{tm}\ u\colon \texttt{type}$ as desired. ◀

▶ **Theorem 60.** `Typify` *is well-typed and in* $\texttt{LF}+\eta$ *functorial.*

Proof. Using Lemma 58 and Theorem 59, the proof proceeds analogously to the one for `Poly` in Theorem 63 with one critical addition:

In the induction proof for functoriality, there is one subcase where we need to show $O(\sigma)(t^{\Sigma,u}) = \sigma(t)^{\Sigma',u}$ for $t\colon A\colon \texttt{type}$ and $c \in \texttt{Untyped}$. For `Poly` we needed to show this for $c \in \texttt{Typed}$ instead and concluded that there is no such c such that the typing holds. But for `Typify` and $c \in \texttt{Untyped}$, there is such a c. Namely, we need to consider the case $t = \texttt{term}$. But then by definition of $-^{\Sigma,u}$ (with our modification for `Typify`) from **??** we can reason $O(\sigma)(\texttt{term}^{\Sigma,u}) = O(\sigma)(\texttt{tm}\ u) = \texttt{tm}\ u = \sigma(\texttt{tm}\ u)^{\Sigma',u}$ as desired. ◀

4.2.4 Meta-Theoretical Properties

▶ **Lemma 61.** *The function $-^{\Sigma,u}$ commutes with substitution:*

$$s^{\Sigma,u}[x \mapsto t^{\Sigma,u}] = (s[x \mapsto t])^{\Sigma,u}$$

Proof. By induction on s. The most interesting case is λ-abstraction (analogously Π-abstraction) spelled out below.

$$\begin{aligned}(\lambda x\colon A.\ s)^{\Sigma,u}[y \mapsto t^{\Sigma,u}] =& (\lambda x\colon A^{\Sigma,u}.\ s^{\Sigma,u})[y \mapsto t^{\Sigma,u}]\\ =& \lambda x\colon A^{\Sigma,u}[y \mapsto t^{\Sigma,u}].\ s^{\Sigma,u}[y \mapsto t^{\Sigma,u}]\\ =& \lambda x\colon (A[y \mapsto t])^{\Sigma,u}.\ (s[y \mapsto t])^{\Sigma,u}\\ =& (\lambda x\colon A[y \mapsto t].\ s[y \mapsto t])^{\Sigma,u}\\ =& ((\lambda x\colon A.\ s)[y \mapsto t])^{\Sigma,u}\end{aligned}$$

We assume names x and y to be disjoint without loss of generality. And in the third line we exploit induction hypotheses on A and s. ◀

▶ **Theorem 62** (Well-Typedness of Constant Indexing). *Let u be an identifier. Then we have for all terms t and A and contexts Γ that are AV-fresh wrt. u:*

$$\begin{aligned}\Gamma \vdash^{\mathtt{Typed}}_{\Sigma} A\colon \mathtt{kind} &\implies u\colon \mathtt{tp}, \Gamma^{\Sigma,u} \vdash^{\mathtt{Typed}}_{\mathtt{Poly}(\Sigma)} A^{\Sigma,u}\colon \mathtt{kind}\\ \Gamma \vdash^{\mathtt{Typed}}_{\Sigma} A\colon \mathtt{type} &\implies u\colon \mathtt{tp}, \Gamma^{\Sigma,u} \vdash^{\mathtt{Typed}}_{\mathtt{Poly}(\Sigma)} A^{\Sigma,u}\colon \mathtt{type}\\ \Gamma \vdash^{\mathtt{Typed}}_{\Sigma} t\colon A &\implies u\colon \mathtt{tp}, \Gamma^{\Sigma,u} \vdash^{\mathtt{Typed}}_{\mathtt{Poly}(\Sigma)} t^{\Sigma,u}\colon A^{\Sigma,u}\end{aligned}$$

Proof. By mutual induction on derivation trees of the antecedences. The full proof is rather lengthy and dull (and requires straightforwardly strengthening the claims, e.g., to judgements on context validity and equality). Below, we only show the most interesting cases for the induction on typing judgements (corresponding to the last line shown in the theorem statement). We abbreviate $\Gamma' := u\colon \mathtt{tp}, \Gamma^{\Sigma,u}$.

- case (CONST-OBJ) (analogously (CONST-FAM)):

$$\frac{\vdash^{\mathtt{Typed}}_{\Sigma} \Gamma \qquad c\colon A \in \mathtt{Typed}, \Sigma}{\Gamma \vdash^{\mathtt{Typed}}_{\Sigma} c\colon A}$$

  - case c declared in Typed: This implies $\vdash^{\mathtt{Typed}} c\colon A$. And by weakening we have $\Gamma' \vdash^{\mathtt{Typed}}_{\mathtt{Poly}(\Sigma)} c\colon A$ as desired.
  - case c declared in Σ: We need to show $\Gamma' \vdash^{\mathtt{Typed}}_{\mathtt{Poly}(\Sigma)} c\,u\colon A^{\Sigma,u}$. We deduce the desired claim by application of (APP-OBJ):

$$\frac{\Gamma' \vdash^{\mathtt{Typed}}_{\mathtt{Poly}(\Sigma)} c\colon (\Pi\, u\colon \mathtt{tp}.\ A^{\Sigma,u}) \qquad \Gamma' \vdash^{\mathtt{Typed}}_{\mathtt{Poly}(\Sigma)} u\colon \mathtt{tp}}{\Gamma' \vdash^{\mathtt{Typed}}_{\mathtt{Poly}(\Sigma)} c\,u\colon A^{\Sigma,u}}$$

  Here, the first premise above the line follows by construction of $\mathtt{Poly}(\Sigma)$. And the second premise follows by construction of Γ' (and some induction hypothesis on contexts that would emerge from strengthening the claims as noted above).

- case (APP-OBJ) (analogously (APP-FAM)):

$$\frac{\Gamma \vdash^{\mathtt{Typed}}_{\Sigma} f\colon \Pi\, x\colon A.\ B \qquad \Gamma \vdash^{\mathtt{Typed}}_{\Sigma} t\colon A}{\Gamma \vdash^{\mathtt{Typed}}_{\Sigma} f\ t\colon B[x \mapsto A]}$$

  Our goal is $\Gamma' \vdash^{\mathtt{Typed}}_{\mathtt{Poly}(\Sigma)} (f\ t)^{\Sigma,u}\colon (B[x \mapsto A])^{\Sigma,u}$. We first apply (APP-OBJ) on our induction hypotheses:

$$\frac{\Gamma' \vdash^{\mathtt{Typed}}_{\mathtt{Poly}(\Sigma)} f^{\Sigma,u}\colon \Pi\, x\colon A^{\Sigma,u}.\ B^{\Sigma,u} \qquad \Gamma' \vdash^{\mathtt{Typed}}_{\mathtt{Poly}(\Sigma)} t^{\Sigma,u}\colon A^{\Sigma,u}}{\Gamma' \vdash^{\mathtt{Typed}}_{\mathtt{Poly}(\Sigma)} f^{\Sigma,u}\ t^{\Sigma,u}\colon B^{\Sigma,u}[x \mapsto A^{\Sigma,u}]}$$

  The conclusion below the line is syntactically identical to our goal, noting that *i)* $f^{\Sigma,u}\, t^{\Sigma,u} = (f\ t)^{\Sigma,u}$ (commutation with function application; by definition) and *ii)* $B^{\Sigma,u}[x \mapsto A^{\Sigma,u}] = (B[x \mapsto A])^{\Sigma,u}$ (commutation with substitution; by Lemma 61) .

◀

▶ **Theorem 63.** `Poly` *is well-typed and in* `LF` $+\,\eta$ *functorial.*

Proof. For well-typedness we apply Theorem 24, for which we obtain the prerequisites by instantiating Theorem 62 with $\Gamma = \varnothing$ and closing the judgements with Π and λ:

$$\vdash^{\mathtt{Typed}}_{\Sigma} A\colon \mathtt{kind} \implies \vdash^{\mathtt{Typed}}_{\mathtt{Poly}(\Sigma)} \underbrace{\Pi\, u\colon \mathtt{tp}.\ A^{\Sigma,u}}_{}\colon \mathtt{kind}$$

$$\vdash^{\mathtt{Typed}}_{\Sigma} A\colon \mathtt{type} \implies \vdash^{\mathtt{Typed}}_{\mathtt{Poly}(\Sigma)} \overbrace{\Pi\, u\colon \mathtt{tp}.\ A^{\Sigma,u}}^{=E_\Sigma(A)}\colon \mathtt{type}$$

$$\vdash^{\mathtt{Typed}}_{\Sigma} t\colon A \implies \vdash^{\mathtt{Typed}}_{\mathtt{Poly}(\Sigma)} \underbrace{(\lambda u\colon \mathtt{tp}.\ t^{\Sigma,u})}_{=e_\Sigma(t)}\colon \underbrace{\Pi\, u\colon \mathtt{tp}.\ A^{\Sigma,u}}_{=E_\Sigma(A)}$$

For functoriality we apply Lemma 25. First, for $c \in \Sigma$ we have $e_\Sigma(c) = \lambda u\,.\ c^{\Sigma,u} = \lambda u.\ c\,u = c$ by η-reduction as desired. Second, we compute

$$\begin{array}{lll} & O(\sigma)(e_\Sigma(t)) & \\ = & O(\sigma)(\lambda u\colon \mathtt{tp}.\ t^{\Sigma,u}) & \text{by definition of } e \\ = & \lambda u\colon \mathtt{tp}.\ O(\sigma)(t^{\Sigma,u}) & \text{since } O(\sigma) \text{ is the identity on } \mathtt{Typed} \\ = & \lambda u\colon \mathtt{tp}.\ \sigma(t)^{\Sigma',u} & \text{by induction on } t \\ = & e_{\Sigma'}(\sigma(t)) & \text{by definition of } e \end{array}$$

where we show $O(\sigma)(t^{\Sigma,u}) = \sigma(t)^{\Sigma',u}$ for all terms $\vdash^{\mathtt{Typed}}_{\Sigma} t\colon A\colon \mathtt{type}$ that are AV-fresh wrt. u by induction:

- case constant $c \in \Sigma$:

$$\begin{array}{rll} O(\sigma)(c^{\Sigma,u}) & = O(\sigma)(c\,u) & \text{by definition of } -^{\Sigma,u} \\ & = O(\sigma)(c)\ u & \text{by definition of morphisms} \\ & = e_{\Sigma'}(\sigma(c))\ u & \text{by definition of } \mathtt{Typify} \\ & = (\lambda u.\ \sigma(c)^{\Sigma',u})\ u & \text{by definition of } e \\ & = \sigma(c)^{\Sigma',u} & \text{by } \beta\text{-reduction} \end{array}$$

- cases $t = c$ (for $c \in \mathtt{Typed}$), $t = \mathtt{type}$, and $t = \mathtt{kind}$: vacuously true since there is no LF type A to begin with such that $\vdash^{\mathtt{Typed}}_{\Sigma} t\colon A\colon \mathtt{type}$.

- case function application $f\ t$:

$$\begin{aligned} O(\sigma)((f\ t)^{\Sigma,u}) &= O(\sigma)(f^{\Sigma,u})\ O(\sigma)(t^{\Sigma,u}) && \text{by definition of } -^{\Sigma,u} \text{ and morphisms}\\ &= \sigma(f)^{\Sigma',u}\ \sigma(t)^{\Sigma',u} && \text{by induction hypotheses}\\ &= \sigma(f\ t)^{\Sigma',u} && \text{by definition of } -^{\Sigma',u} \text{ and morphisms}\end{aligned}$$

- case function abstraction $\lambda x\colon A.\ t$ (analgously for $\Pi\, x\colon A.\ B$):

$$\begin{aligned} O(\sigma)((\lambda x\colon A.\ t)^{\Sigma,u}) &= \lambda x\colon O(\sigma)(A^{\Sigma,u}).\ O(\sigma)(t^{\Sigma,u}) && \text{by definition of } -^{\Sigma,u} \text{ and morphisms}\\ &= \lambda x\colon \sigma(A)^{\Sigma',u}.\ \sigma(t)^{\Sigma',u} && \text{by induction hypotheses}\\ &= \sigma(\lambda x\colon A.\ t)^{\Sigma',u} && \text{by definition of } -^{\Sigma',u} \text{ and morphisms}\end{aligned}$$

◀

▶ **Lemma 64.** *The connectors and $-^{\Sigma,u}$ are related via*

$$\vdash^{S}_{\Sigma} \mathtt{For}_{\Sigma}(t^{\Sigma,u}) = t$$
$$\vdash^{\mathtt{Typed}}_{\mathtt{Poly}(\Sigma)} \mathtt{Sel}^{\mathbb{T}}_{\Sigma}(t) = t^{\Sigma,\mathbb{T}}$$

for all t-terms that are AV*-fresh wrt. u.*

Proof. Both claims are proven by induction on t, and we only show the case for constants $c \in \Sigma$. All other cases are either trivial ($c \notin \Sigma$ or $c \in \{\mathtt{type}, \mathtt{kind}\}$) or follow immediately by compositionality of involved morphisms and $-^{\Sigma,u}$. For the first claim, we compute $\mathtt{For}_{\Sigma}(c^{\Sigma,u}) = \mathtt{For}_{\Sigma}(c\,u) = (\lambda\_\colon \mathtt{tp.}\ c)\ u = c$ by β-reduction. And for the second claim, we compute $\mathtt{Sel}^{\mathbb{T}}_{\Sigma}(c) = c\ \mathbb{T} = c^{\Sigma,\mathbb{T}}$. ◀

▶ **Theorem 65.** *The connector* $\mathtt{For}\colon \mathtt{Poly} \to \mathtt{Id}$ *is well-typed and natural.*

Proof. For well-typedness we use Theorem 35 and note for all constants $\vdash^{S}_{\Sigma} c\colon A$

$$\vdash^{S}_{\Sigma} \mathtt{For}_{\Sigma}(c\colon A)\colon \mathtt{For}_{\Sigma}(\Pi\, u\colon \mathtt{tp.}\ A^{\Sigma,u})$$
$$\Leftrightarrow \vdash^{S}_{\Sigma} (\lambda\_\colon \mathtt{tp.}\ c)\colon \underbrace{\mathtt{For}_{\Sigma}((\Pi\, u\colon \mathtt{tp.}\ A)^{\Sigma,u})}_{=\Pi\, u\colon \mathtt{tp.}\ A \text{ by Lemma 64}}$$

And for naturality we use Corollary 40 and show

$$\vdash^{S}_{\Sigma} \mathtt{For}_{\hat{\Sigma}}(e_{\hat{\Sigma}} \circ \sigma(c)) = \sigma(\mathtt{For}_{\Sigma}(c))$$

The left-hand side simplifies via

$$\begin{aligned} \mathtt{For}_{\hat{\Sigma}}(e_{\hat{\Sigma}} \circ \sigma(c)) &= \mathtt{For}_{\hat{\Sigma}}(\lambda u\colon \mathtt{tp.}\ \sigma(c)^{\hat{\Sigma},u})\\ &= \lambda u\colon \mathtt{tp.}\ \mathtt{For}_{\hat{\Sigma}}(\sigma(c)^{\hat{\Sigma},u})\\ &= \lambda u\colon \mathtt{tp.}\ \sigma(c)\\ &= \lambda\_\colon \mathtt{tp.}\ \sigma(c) && \text{by } u \notin \mathrm{AV}(\sigma(c))\end{aligned}$$

The right-hand side simplifies to the same via $\sigma(\mathtt{For}_{\Sigma}(c)) = \sigma(\lambda\_\colon \mathtt{tp.}\ c) = \lambda\_\colon \mathtt{tp.}\ \sigma(c)$. ◀

▶ **Theorem 66.** *The connectors* $\mathtt{Sel}^{\mathbb{T}}\colon \mathtt{Id} \to \mathtt{Poly}$ *are well-typed and natural.*

Proof. For well-typedness, we use Theorem 35, we need to show

$$\vdash^{\texttt{Typed}}_{\Sigma} c\colon A \implies \vdash^{\texttt{Typed}}_{\texttt{Poly}(\Sigma)} c\ \mathbb{T}\colon \underbrace{\texttt{Sel}^{\mathbb{T}}_{\Sigma}(A)}_{=A^{\Sigma,\mathbb{T}} \text{ by Lemma 64}}$$

Indeed by construction we have $\vdash^{\texttt{Typed}}_{\texttt{Poly}(\Sigma)} c\colon (\Pi\, u\colon \texttt{tp}.\ A^{\Sigma,u})$ and thus $\vdash^{\texttt{Typed}}_{\texttt{Poly}(\Sigma)} c\ \mathbb{T}\colon A^{\Sigma,\mathbb{T}}$.
For naturality, we use **??** and need to show

$$\vdash^{\texttt{Typed}}_{\texttt{Poly}(\hat{\Sigma})} \texttt{Sel}^{\mathbb{T}}_{\hat{\Sigma}}(\sigma(c)) = \texttt{Poly}_{\hat{\Sigma}}(\sigma)(\texttt{Sel}^{\mathbb{T}}_{\Sigma}(c))$$

for all constants $c \in \Sigma$. Let $(c := t)$ be the assignment to c in σ. For the right-hand side we have

$$\begin{aligned}\texttt{Poly}_{\hat{\Sigma}}(\sigma)(\texttt{Sel}^{\mathbb{T}}_{\Sigma}(c)) &= \texttt{Poly}_{\hat{\Sigma}}(\sigma)(c\ \mathbb{T})\\ &= \texttt{Poly}_{\hat{\Sigma}}(\sigma)(c)\ \texttt{Poly}_{\hat{\Sigma}}(\sigma)(\mathbb{T})\\ &= (\lambda u\colon \texttt{tp}.\ t^{\hat{\Sigma},u})\ \mathbb{T}\\ &= t^{\hat{\Sigma},\mathbb{T}}\end{aligned}$$

And the left-hand side simplifies to the same via $\texttt{Sel}^{\mathbb{T}}_{\hat{\Sigma}}(\sigma(c)) = \texttt{Sel}^{\mathbb{T}}_{\hat{\Sigma}}(t) = t^{\hat{\Sigma},\mathbb{T}}$. ◀

▶ **Theorem 67.** `For` *is the left-inverse of* $\texttt{Sel}^{\mathbb{T}}$ *in the sense that for any theory* T *we have*

$$\texttt{For}(T) \circ \texttt{Sel}^{\mathbb{T}}(T) = \mathrm{id}_T$$

Proof. Immediately follows from Lemma 64 ◀

4.3 Parameter Removal

$$X \xrightarrow{\texttt{CleanIn}_f(X)} \texttt{Clean}_f(X)$$

The linear functor $\texttt{Clean}_f$ removes parameters of constants within theories it is applied to, guided by a *parameter keep heuristic* f. For example, if the constant $c\colon A \to B \to C$ in some theory declares a ternary function and the heuristic, given the whole constant declaration, instructs us to remove the second parameter, then the functor maps this constant to $c\colon A \to C$. The practical signifiance stems from understanding the functor as a refactoring operation [**roux:bsc**], which can be useful for different reasons. For example, the functor could be built into a graphical editor for MMT formalizations, such that users can then apply the refactorings to their formalization. () Moreover, having functors that perform syntax-oriented refactorings eases the specification of complex functors as those can then emerge as the composition of some easily, but slightly off on the syntax side functors O and a refactoring op, see **??**.

Related Work parameter removal is a standard refactoring techniques
supported by enterprise IDEs, e.g., intellij, visual studio.
how about formal systems IDEs?
cite to my bsc thesis

4.3.1 Definition

▶ **Notation 68.** We assume η-conversion in the whole section.

W.l.o.g. after suitable $\alpha\beta\eta$ normalization we make the following assumption: First, for every constant $c\colon A$ we assume A starts with a (possibly empty) sequence of n Π-bindings, and any definition of c (direct or morphism) starts with the same variable sequence λ-bound. Second, we assume that every occurrence of c in a term is applied to exactly n terms. In proofs below that induct on typing derivations, we also effectively assume that the typing rule for constants reads as follows:

$$\frac{\begin{array}{c}\Gamma \vdash_\Sigma t_1\colon A_1 \qquad \Gamma \vdash_\Sigma t_2\colon A_2[x_1 \mapsto t_1] \qquad \cdots \\ \Gamma \vdash_\Sigma t_n\colon A_n[x_i \mapsto t_i]_{i=1,..,n-1} \qquad \Gamma \vdash_\Sigma (c\colon \Pi\, x_1\colon A_1.\ \dots \Pi\, x_n\colon A_n.\ B) \in \Sigma\end{array}}{\Gamma \vdash_\Sigma c\ t_1\ \dots\ t_n\colon B[x_i \mapsto t_i]_{i=1,..,n}}\ (\textsc{const-eta})$$

Sometimes we also write short

$$\frac{\begin{array}{c}\Gamma \vdash_\Sigma t_i\colon A_i[x_k \mapsto t_k]_{k=1,..,i-1} \text{ for } 1 \leq i \leq n \\ \Gamma \vdash_\Sigma (c\colon \Pi\, x_1\colon A_1.\ \dots \Pi\, x_n\colon A_n.\ B) \in \Sigma\end{array}}{\Gamma \vdash_\Sigma c\ t_1\ \dots\ t_n\colon B[x_i \mapsto t_i]_{i=1,..,n}}\ (\textsc{const-eta})$$

▶ **Definition 69** (Parameter Keep Heuristic). *A* ***parameter keep heuristic*** *is a function f that maps constant declarations in context of flat theories Σ to subsets $f^\Sigma(c\colon \Pi\, x_1\colon A_1.\ \dots \Pi\, x_n\colon A_n.\ B\,[= \lambda\, x_1\colon A_1.\dots\lambda\, x_n\colon A_n.\ t]) \subseteq \{1,\dots,n\}$ of parameter indices. If clear from context, we simply write $f^\Sigma(c)$ and leave out type and definiens component.*

We call a heuristic well-typed if for all constants $c\colon \Pi\, x_1\colon A_1.\ \dots \Pi\, x_n\colon A_n.\ B\,[= \lambda\, x_1\colon A_1.\dots\lambda\, x_n\colon A_n.\ t]$

i) if x_i is mentioned in $cl_f^\Sigma(A_j)$ $(i < j)$, then if x_j is kept, so is x_i (in formulae: $x_j \in f^\Sigma(c)$ implies $x_i \in f^\Sigma(c)$)

ii) if $cl_f^\Sigma(B)$ or $cl_f^\Sigma(t)$ mentions x_i, then x_i is kept ($x_i \in f^\Sigma(c)$)

▶ **Definition 70** (Parameter Removal on Terms, Contexts, Substitutions). *Every heuristic f induces a parameter-removing function cl_f on terms, contexts, and substitutions given by*

$$cl_f^\Sigma(c\ t_1\ \dots\ t_n) = c\ cl_f^\Sigma(t_{k_1})\ \dots\ cl_f^\Sigma(t_{k_m}) \textit{ where } \{k_1,\dots,k_m\} = f^\Sigma(c)$$
$$cl_f^\Sigma(\texttt{type}) = \texttt{type} \qquad cl_f^\Sigma(\texttt{kind}) = \texttt{kind} \qquad cl_f^\Sigma(x) = x$$
$$cl_f^\Sigma(f\ t) = cl_f^\Sigma(f)\ f(t)$$
$$cl_f^\Sigma(\Pi\, x\colon A.\ B) = \Pi\, x\colon cl_f^\Sigma(A).\ cl_f^\Sigma(B) \qquad cl_f^\Sigma(\lambda x\colon A.\ t) = \lambda x\colon cl_f^\Sigma(A).\ cl_f^\Sigma(t)$$
$$cl_f^\Sigma(\cdot) = \cdot \qquad cl_f^\Sigma(\Gamma, x\colon A) = cl_f^\Sigma(\Gamma), x\colon cl_f^\Sigma(A)$$
$$cl_f^\Sigma(\cdot) = \cdot \qquad cl_f^\Sigma(\rho, x = t) = cl_f^\Sigma(\rho), x = cl_f^\Sigma(t)$$

▶ **Definition 71** (Parameter Removal). *Every heuristic f induces a linear functor* $\texttt{Clean}_f$ *and a strongly linear connector* $\texttt{CleanIn}_f$ *into* $\texttt{Clean}_f$ *by*

$$\texttt{Clean}_f^\Sigma\left(\begin{array}{c} c\colon \Pi\, x_1\colon A_1.\ \dots \Pi\, x_n\colon A_n.\ B \\ [= \lambda\, x_1\colon A_1.\dots\lambda\, x_n\colon A_n.\ t]\end{array}\right) = \begin{array}{c} c\colon \Pi\, x_{k_1}\colon cl_f^\Sigma(A_{k_1}).\ \dots \Pi\, x_{k_m}\colon cl_f^\Sigma(A_{k_m}).\ cl_f^\Sigma(B) \\ [= \lambda\, x_{k_1}\colon cl_f^\Sigma(A_{k_1}).\dots\lambda\, x_{k_m}\colon cl_f^\Sigma(A_{k_m}).\ cl_f^\Sigma(t)]\end{array}$$

$$\begin{aligned}\texttt{CleanIn}_f^\Sigma(c\colon \Pi\, x_1\colon A_1.\ \dots \Pi\, x_n\colon A_n.\ B) &= c\colon \Pi\, x_{k_1}\colon cl_f^\Sigma(A_{k_1}).\ \dots \Pi\, x_{k_m}\colon cl_f^\Sigma(A_{k_m}).\ cl_f^\Sigma(B) \\ &= \lambda\, x_1\colon cl_f^\Sigma(A_1).\dots\lambda\, x_n\colon cl_f^\Sigma(A_n).\ c\ x_{k_1}\ \dots\ x_{k_m}\end{aligned}$$

where $\{k_1, \dots, k_m\} = f^\Sigma(c)$*, and whenever some removed variable* x_k *is still mentioned in some* A_l $(l > k)$ *or in* B *or in* t*, we leave the whole result undefined.*

The only thing preventing $\mathtt{Clean}_f$ from being strongly linear is the dependence on the constant identifier c in the translation functions for the type and definiens components.

4.3.2 Meta-Theoretical Properties

▶ **Lemma 72.** *We have* $\mathtt{CleanIn}_f^\Sigma(-) = cl_f^\Sigma(-)$ *as functions on* Σ*-syntax (terms, contexts, substitutions).*

Proof. Since both functions are compositional on syntax, it suffices to show equality on constants $c\colon \Pi\, x_1\colon A_1.\ \dots\ \Pi\, x_n\colon A_n.\ B$ which follows by

$$\begin{array}{rll} & \mathtt{CleanIn}_f^\Sigma(c) & \\ = & \lambda\, x_1\colon cl_f^\Sigma(A_1).\ \dots\ \lambda\, x_n\colon cl_f^\Sigma(A_n).\ \ c\ x_{k_1}\ \dots\ x_{k_m} & \text{by def. of } \mathtt{CleanIn}_f \\ = & \lambda\, x_1\colon cl_f^\Sigma(A_1).\ \dots\ \lambda\, x_n\colon cl_f^\Sigma(A_n).\ \ cl_f^\Sigma(c\ x_1\ \dots\ x_n) & \text{by def. of } cl_f \\ = & cl_f^\Sigma(\lambda\, x_1\colon A_1.\ \dots\ \lambda\, x_n\colon A_n.\ \ c\ x_1\ \dots\ x_n) & \text{by def. of } cl_f \\ = & cl_f^\Sigma(c) & \text{by } \eta\text{-conversion} \end{array}$$

◀

▶ **Corollary 73** (cl_f **commutes with substitution**)**.** *For all terms* $\vdash_\Sigma t$ *and substitutions* ρ *we have*

$$cl_f^\Sigma(t\rho) = cl_f^\Sigma(t)\ cl_f^\Sigma(\rho)$$

where the notation on the RHS denotes application of the substituion $cl_f^\Sigma(\rho)$ *to term* $cl_f^\Sigma(t)$*.*

Proof. By Lemma 72 since the claim is known to be true for morphisms. ◀

▶ **Lemma 74.** *Under the conditions in* **??** *we have:*

$$\begin{array}{rcl} \Gamma \vdash_\Sigma A\colon \mathtt{type} & \Longrightarrow & cl_f^\Sigma(\Gamma) \vdash_{\mathtt{Clean}_f(\Sigma)} cl_f^\Sigma(A)\colon \mathtt{type} \\ \Gamma \vdash_\Sigma A\colon \mathtt{kind} & \Longrightarrow & cl_f^\Sigma(\Gamma) \vdash_{\mathtt{Clean}_f(\Sigma)} cl_f^\Sigma(A)\colon \mathtt{kind} \\ \Gamma \vdash_\Sigma t\colon A & \Longrightarrow & cl_f^\Sigma(\Gamma) \vdash_{\mathtt{Clean}_f(\Sigma)} cl_f^\Sigma(t)\colon cl_f^\Sigma(A) \\ \Gamma \vdash_\Sigma t \equiv t' & \Longrightarrow & cl_f^\Sigma(\Gamma) \vdash_{\mathtt{Clean}_f(\Sigma)} cl_f^\Sigma(t) \equiv cl_f^\Sigma(t') \end{array}$$

Proof. By induction on typing derivations (possibly strengthened with further claims about judgements on contexts, substitutions, etc.). We only show the critical case for typing of constants (following Notation 68). There we have

$$\frac{\Gamma \vdash_\Sigma t_i\colon A_i[x_j \mapsto t_j]_{j=1,..,i-1} \text{ for } 1 \le i \le n \qquad \Gamma \vdash_\Sigma (c\colon \Pi\, x_1\colon A_1.\ \dots\ \Pi\, x_n\colon A_n.\ B) \in \Sigma}{\Gamma \vdash_\Sigma c\ t_1\ \dots\ t_n\colon B[x_i \mapsto t_i]_{i=1,..,n}}$$

and, noting Corollary 73, as induction hypotheses may assume

$$cl_f^\Sigma(\Gamma) \vdash_\Sigma cl_f^\Sigma(t_i)\colon cl_f^\Sigma(A_i)[x_j \mapsto cl_f^\Sigma(t_j)]_{j=1,..,i-1}) \quad \text{for every } 1 \le i \le n$$

We need to show $cl_f^\Sigma(\Gamma) \vdash_{\mathtt{Clean}_f(\Sigma)} cl_f^\Sigma(c\ t_1\ \dots\ t_n)\colon cl_f^\Sigma(B[x_i \mapsto t_i]_{i=1,..,n})$, which by definition of cl_f and Corollary 73 is equivalent to

$$cl_f^\Sigma(\Gamma) \vdash_{\mathtt{Clean}_f(\Sigma)} c\ cl_f^\Sigma(t_{k_1})\ \dots\ cl_f^\Sigma(t_{k_m})\colon cl_f^\Sigma(B)[x_i \mapsto cl_f^\Sigma(t_i)]_{i=1,..,n}$$

To prove this, we first set $\{k_1, \ldots, k_m\} = f^\Sigma(c)$ to the set of parameter indices that are kept. Then we apply (CONST-ETA) from Notation 68 within $\texttt{Clean}_f(\Sigma)$ on c:

$$\frac{\begin{array}{c} cl_f^\Sigma(\Gamma) \vdash_{\texttt{Clean}_f(\Sigma)} cl_f^\Sigma(t_{k_i})\colon cl_f^\Sigma(A_{k_i})[x_{k_i} \mapsto cl_f^\Sigma(t_{k_i})]_{i=1,..,k-1} \text{ for } 1 \leq k \leq m \\ cl_f^\Sigma(\Gamma) \vdash_{\texttt{Clean}_f(\Sigma)} (c\colon \Pi\, x_{k_1} : cl_f^\Sigma(A_{k_1}).\ \ldots \Pi\, x_{k_m} : cl_f^\Sigma(A_{k_m}).\ cl_f^\Sigma(B)) \in \Sigma \end{array}}{cl_f^\Sigma(\Gamma) \vdash_{\texttt{Clean}_f(\Sigma)} c\ cl_f^\Sigma(t_{k_1})\ \ldots\ cl_f^\Sigma(t_{k_m})\colon cl_f^\Sigma(B)[x_{k_i} \mapsto cl_f^\Sigma(t_{k_i})]_{i=1,..,m}}$$

Here, the premises are already close to (some of) our induction hypotheses and the consequence close to our proof goal. Concretely, we argue the following syntactic equalities.

$$\begin{aligned} cl_f^\Sigma(A_{k_i})[x_j \mapsto cl_f^\Sigma(t_j)]_{j=1,..,k_i-1} &\overset{?}{=} cl_f^\Sigma(A_{k_i})[x_{k_i} \mapsto cl_f^\Sigma(t_{k_i})]_{i=1,..,k-1} \text{ for } 1 \leq i \leq m \\ cl_f^\Sigma(B)[x_i \mapsto cl_f^\Sigma(t_i)]_{i=1,..,n} &\overset{?}{=} cl_f^\Sigma(B)[x_{k_i} \mapsto cl_f^\Sigma(t_{k_i})]_{i=1,..,m} \end{aligned}$$

(It may help to consider a concrete example: suppose $(c\colon \Pi\, x_1 : A_1.\ \Pi\, x_2 : A_2.\ \Pi\, x_3 : A_3.\ B) \in \Sigma$ declares a ternary function and the the heuristic f determined we should keep the second and third parameters and remove the first one. Then the first line above asks whether $cl_f^\Sigma(A_3)[x_1 \mapsto cl_f^\Sigma(t_1), x_2 \mapsto cl_f^\Sigma(t_2)] \overset{?}{=} cl_f^\Sigma(A_3)[x_2 \mapsto cl_f^\Sigma(t_2)]$. And the second line asks whether $cl_f^\Sigma(B)[x_1 \mapsto cl_f^\Sigma(t_1), x_2 \mapsto cl_f^\Sigma(t_2), x_3 \mapsto cl_f^\Sigma(t_3)] \overset{?}{=} cl_f^\Sigma(B)[x_2 \mapsto cl_f^\Sigma(t_2), x_3 \mapsto cl_f^\Sigma(t_3)]$.)

For the family of claims on the first line, let us pick some fixed i and consider for every variable $x_j \in \{x_1, \ldots, x_{k_i-1}\}$ we argue that it either is kept (thus appears in the substitution on LHS and RHS) or is irrelevant. If x_j is not mentioned in A_i, then neither is it in $cl_f^\Sigma(A_i)$, thus any substitution at x_j is irrelevant. If x_j is mentioned in A_i (and x_{j_i} kept – otherwise we could ignore the claim anyway for that specific i), then x_j must be kept according to Condition i in Definition 69.

For the claim on the second line, we argue similarly: if $x_j \in \{x_1, \ldots, x_n\}$ is not mentioned in B, then neither is it in $cl_f^\Sigma(B)$ and it is irrelevant; if x_j is mentioned in B, then x_j must be kept according to Condition ii in Definition 69, and thus it appears in the substitution on LHS and RHS. ◀

▶ **Theorem 75.** *If f is a well-typed heuristic, then $\texttt{Clean}_f$ is well-typed and functorial.*

Proof. Well-typedness on theories immediately follows from Lemma 74. For morphisms, consider two flat theories Σ and Σ', a constant $(c\colon \Pi\, x_1 : A_1.\ \ldots \Pi\, x_n : A_n.\ B) \in \Sigma$, and an assignment $(c\colon \Pi\, x_1 : A_1'.\ \ldots \Pi\, x_n : A_n'.\ B' = \lambda\, x_1\colon A_1'.\ \ldots \lambda\, x_n\colon A_n'.\ t) \in \sigma$. Our functor $\texttt{Clean}_f$ maps the constant and the assignment to:

$$\begin{array}{ll} \text{constant in } \texttt{Clean}_f(\Sigma)\text{:} & c\colon \Pi\, x_{k_1} : cl_f^\Sigma(A_{k_1}).\ \ldots \Pi\, x_{k_m} : cl_f^\Sigma(A_{k_m}).\ cl_f^\Sigma(B) \\ \text{assignment in } \texttt{Clean}_f(\sigma)\text{:} & c\colon \Pi\, x_{k_1} : cl_f^\Sigma(A_{k_1}').\ \ldots \Pi\, x_{k_m} : cl_f^{\Sigma'}(A_{k_m}').\ cl_f^{\Sigma'}(B') \\ & \quad = \lambda\, x_{k_1}\colon cl_f^{\Sigma'}(A_{k_1}').\ \ldots \lambda\, x_{k_m}\colon cl_f^{\Sigma'}(A_{k_m}').\ cl_f^{\Sigma'}(t) \end{array}$$

For the generated assignment to be well-typed, we need to check two things. First, the assigned term must have the given type. This again follows from Lemma 74. Second, we need to prove

$$\vdash_{\Sigma'} \texttt{Clean}_f(\sigma)(\Pi\, x_{k_1} : cl_f^\Sigma(A_{k_1}).\ \ldots \Pi\, x_{k_m} : cl_f^\Sigma(A_{k_m}).\ cl_f^\Sigma(B)) \equiv \Pi\, x_{k_1} : cl_f^\Sigma(A_{k_1}').\ \ldots \Pi\, x_{k_m} : cl_f^\Sigma(A_{k_m}').\ cl_f^{\Sigma'}(B')$$

Starting from the LHS, we derive the equality as follows:

$$\begin{aligned}
&\mathtt{Clean}_f(\sigma)(\Pi\, x_{k_1} : cl_f^{\Sigma}(A_{k_1}).\ \dots\ \Pi\, x_{k_m} : cl_f^{\Sigma}(A_{k_m}).\ cl_f^{\Sigma}(B))\\
\equiv\ &\Pi\, x_{k_1} : \mathtt{Clean}_f(\sigma)(cl_f^{\Sigma}(A_{k_1})).\ \dots\ \Pi\, x_{k_m} : \mathtt{Clean}_f(\sigma)(cl_f^{\Sigma}(A_{k_m})).\ \mathtt{Clean}_f(\sigma)(cl_f^{\Sigma}(B))\\
\equiv\ &\Pi\, x_{k_1} : cl_f^{\Sigma'}(\sigma(A_{k_1})).\ \dots\ \Pi\, x_{k_m} : cl_f^{\Sigma'}(\sigma(A_{k_m})).\ cl_f^{\Sigma'}(\sigma(B))\\
\equiv\ &\Pi\, x_{k_1} : cl_f^{\Sigma}(A'_{k_1}).\ \dots\ \Pi\, x_{k_m} : cl_f^{\Sigma}(A'_{k_m}).\ cl_f^{\Sigma'}(B')
\end{aligned}$$

Above, the second line follows by definition of morphism application, the third line by rewriting cl_f to $\mathtt{CleanIn}_f$ and noting naturality as per Theorem 76, and the fourth line noting $\vdash_{\Sigma} \sigma(A_i) \equiv A'_i$ and $\vdash_{\Sigma} \sigma(B) \equiv B'$ by the assumption of the original assignment in σ being well-typed.

Functoriality: ◀

▶ **Theorem 76.** *If f is a well-typed heuristic, then* $\mathtt{CleanIn}_f$ *is well-typed and natural.*

Proof. We use **??** and induct on arity of constants. Consider $(c\colon \Pi\, x_1 : A_1.\ \dots\ \Pi\, x_n : A_n.\ B) \in \Sigma$ and $(c = \lambda\, x_1\colon A'_1.\ \dots\ \lambda\, x_n\colon A'_n.\ t) \in \sigma$. We unify the base case (which occurs for $n = 0$) and the induction step in the computation below, proving $\mathtt{Clean}_f(\sigma)(\mathtt{CleanIn}_f^{\Sigma}(c)) = \mathtt{CleanIn}_f^{\Sigma'}(\sigma(c))$. As induction hypotheses we assume $\mathtt{Clean}_f(\sigma)(\mathtt{CleanIn}_f^{\Sigma}(A_i)) = \mathtt{CleanIn}_f^{\Sigma'}(\sigma(A_i))$ for all $1 \le i \le n$.

$$\begin{aligned}
&\mathtt{Clean}_f(\sigma)(\mathtt{CleanIn}_f^{\Sigma}(c)) &&\\
&= \mathtt{Clean}_f(\sigma)(\lambda\, x_1\colon cl_f^{\Sigma}(A_1).\ \dots\ \lambda\, x_n\colon cl_f^{\Sigma}(A_n).\ \ c\ x_{k_1}\ \dots\ x_{k_m}) && \text{by def. of } \mathtt{CleanIn}_f\\
&= \lambda\, x_1\colon \mathtt{Clean}_f(\sigma)(cl_f^{\Sigma}(A_1)).\ \dots\ \lambda\, x_n\colon \mathtt{Clean}_f(\sigma)(cl_f^{\Sigma}(A_n)). && \text{by def. of } \mathtt{CleanIn}_f\\
&\qquad \mathtt{Clean}_f(\sigma)(c\ x_{k_1}\ \dots\ x_{k_m} &&\\
&= \lambda\, x_1\colon \mathtt{Clean}_f(\sigma)(cl_f^{\Sigma}(A_1)).\ \dots\ \lambda\, x_n\colon \mathtt{Clean}_f(\sigma)(cl_f^{\Sigma}(A_n)). && \text{by def. of } \mathtt{Clean}_f\\
&\qquad (\lambda\, x_{k_1}\colon cl_f^{\Sigma'}(A'_1).\ \dots\ \lambda\, x_{k_m}\colon cl_f^{\Sigma'}(A'_n).\ \ cl_f^{\Sigma'}(t))\ x_{k_1}\ \dots\ x_{k_m} &&\\
&= \lambda\, x_1\colon \mathtt{Clean}_f(\sigma)(cl_f^{\Sigma}(A_1)).\ \dots\ \lambda\, x_n\colon \mathtt{Clean}_f(\sigma)(cl_f^{\Sigma}(A_n)).\ \ cl_f^{\Sigma'}(t) && \text{by } \beta\text{-reduction}\\
&= \lambda\, x_1\colon \mathtt{CleanIn}_f^{\Sigma'}(\sigma(A_1)).\ \dots\ \lambda\, x_n\colon \mathtt{CleanIn}_f^{\Sigma'}(\sigma(A_n)).\ \ \mathtt{CleanIn}_f^{\Sigma'}(t) && \text{by induction hypotheses on } A_i\\
& && \text{and Lemma 72}\\
&= \mathtt{CleanIn}_f^{\Sigma'}(\sigma(\lambda\, x_1\colon A_1.\ \dots\ \lambda\, x_n\colon A_n.\ \ t)) && \text{by morphism property}\\
& && \vdash_{\Sigma'} \sigma(A_i) \equiv A'_i\\
&= \mathtt{CleanIn}_f^{\Sigma'}(\sigma(c)) && \text{by def. of } \sigma(c)
\end{aligned}$$

◀

4.4 Representing Logical Relations

$$\begin{tikzcd}
R \arrow[r, shift left=2, "\mathtt{LogrelIn}_n^{(1)}"] \arrow[r, phantom, "\vdots"] \arrow[r, shift right=2, "\mathtt{LogrelIn}_n^{(n)}"'] & \mathtt{Logrel}_n(R) \arrow[r, "r"] & S
\end{tikzcd}$$

Logical relations are an established proof technique for deriving meta-level theorems of formal systems, such as strong normalization, type safety, and correctness of compiler optimizations. In [RS13] Rabe and Sojakova concretize this technique to the setting of formal systems represented by MMT theories and morphisms. Let $m_1, \dots, m_n : R \to S$ be morphisms between two theories S and T. In the sense of these authors, a logical relation r on $m_1, \dots, m_n$ is a mathematical function that specifies an n-ary LF relation $\vdash^S r(T)\colon m_1(T) \to \dots \to m_n(T) \to \mathtt{type}$ for every LF type $\vdash^R T\colon \mathtt{type}$. And for every corresponding term $\vdash^R t\colon T$ it gives a witness (proof) $\vdash^S r(t)\colon r(T)\ m_1(t)\ \dots\ m_n(t)$. Overall, these logical relations capture *mechanically verifiable* meta theorems that relate images of terms under arbitrarily many, but fixed morphisms, where such relations are expressible as LF relations. We refer to Section 2.2 for details.

For an arity $n \geq 1$, the linear functor $\texttt{Logrel}_n$ enables the representation of n-ary logical relations. It maps every theory R to the theory $\texttt{Logrel}_n(R)$ whose realizations (i.e., outgoing morphisms) correspond precisely to all possible n-ary total logical relations on morphisms with domain R. By construction, every morphism $\texttt{Logrel}_n(R) \to S$ will uniquely encode n-many morphisms $m_1, \ldots, m_n \colon R \to S$ and a corresponding logical relation on them. And conversely, every n-ary logical relation on morphisms $m_1, \ldots, m_n \colon R \to S$ will be represented by a unique morphism $\texttt{Logrel}_n(R) \to S$.

Our main contribution is adding support for representing logical relations to the MMT system. While logical relations have been a standard tool for verifying logic translations in MMT or MMT-near settings [Rab17a; Soj10; SS08], this is the first time they can actually be represented in MMT. This means that several important meta theorems – such as strong normalization of the simply-typed lambda calculus – can now be represented and mechanically verified for the first time in MMT. Note that the predecessor of MMT [RS09] (which was based on Twelf) had built-in primitives for representing logical relations [RS13, Sec. 5], but they had never been implemented for the MMT system for lack of development resources. However, the groundwork of our framework allowed a reconsideration. Instead of implementing a primitive – which would be a daunting task, touching many parts of MMT's trusted core – we defer to implementing a linear functor. We continue discussing related work in ??.

4.4.1 Total Logical Relations

Section 2.2, [RR21b] We give a linear functor that enables representing the total logical relations from ??.

▶ **Definition 77** (Total Logical Relation Operators). *For an arity $n \geq 1$ the linear functor $\texttt{Logrel}_n$ is given on all theories and morphisms by*

$$\texttt{Logrel}_{n,\Sigma}(c\colon A\,[=t]) = \begin{cases} c^{(1)}\colon A^{(1)}\,[=t^{(1)}] \\ \vdots \\ c^{(n)}\colon A^{(n)}\,[=t^{(n)}] \\ c^{*}\colon r_{n,\Sigma}(A)\ c^{(1)}\ \ldots\ c^{(n)}\,[=r_{n,\Sigma}(t)] \end{cases}$$

and for every $1 \leq i \leq n$ the strongly linear connector $\texttt{LogrelIn}_n^{(i)}$ into $\texttt{Logrel}_n$ is given by

$$\texttt{LogrelIn}_{n,\Sigma}^{(i)}(c\colon A) = c^{(i)}$$

Above, the function $-^{(i)}_{\Sigma}$ (abbreviated as $-^{(i)}$) replaces every occurrence of a constant $c \in \Sigma$ by $c^{(i)}$, and $r_{n,\Sigma}$ is the n-ary logical relation on $\texttt{LogrelIn}_{n,\Sigma}^{(1)}, \ldots, \texttt{LogrelIn}_{n,\Sigma}^{(n)}$ induced by $r_{n,\Sigma}(c) = c^{*}$ for every $c \in \Sigma$.

▶ **Theorem 78.** $\texttt{Logrel}_n$ *is well-typed.*

Proof of Theorem 78. We use Theorem 24 and show for all constants $\vdash^{S}_{\Sigma} c\colon A\,[=t]$ the following validity judgements on constant declarations:

$$\vdash_{\texttt{Logrel}_n(\Sigma)} c^{(i)}\colon A^{(i)}\,[=t^{(i)}] \qquad \text{for } 1 \leq i \leq n$$

$$\vdash_{\texttt{Logrel}_n(\Sigma),c^{(1)},..,c^{(n)}} c^{*}\colon r_{n,\Sigma}(A)\ c^{(1)}\ \ldots\ c^{(n)}\,[=r_{n,\Sigma}(t)]$$

The family of judgements shown on the first line clearly holds since the function $-^{(i)}$ is just a systematic renaming of identifiers. And the judgement on the second line immediately follows from $r_{n,\Sigma}$ being a logical relation on $\texttt{LogrelIn}^{(1)}_{n,\Sigma}, \ldots, \texttt{LogrelIn}^{(n)}_{n,\Sigma}$. ◀

▶ **Theorem 79.** *The connectors* $\texttt{LogrelIn}^{(i)}_n$ *are well-typed and natural.*

Proof. By Theorem 38 in conjunction with Theorem 78. ◀

The following proof of adequacy is quite instructive to understand in detail how our functor enables the representation of logical relations.

▶ **Theorem 80** (Adequacy). *For $n \geq 1$ and any flat theory R, we have the bijection of sets*

$$\textit{realizations of } \texttt{Logrel}_n(R) \;\cong\; n\textit{-ary total logical relations on } R$$

Proof. We exhibit two functions between the LHS and RHS that are inverses of each other by construction.

For the forward direction, suppose we have a realization $v\colon \texttt{Logrel}_n(S) \to T$ for some theory T. We decode it to the logical relation $r = v \circ r_{n,S}$ on morphisms

$$\begin{aligned} m_1 &= v \circ \texttt{LogrelIn}^{(1)}_n(S) \\ &\vdots \\ m_n &= v \circ \texttt{LogrelIn}^{(n)}_n(S) \end{aligned}$$

Clearly, these morphisms are well-typed as compositions of well-typed morphisms. And r is a logical relation on $m_1, \ldots, m_n$ because $r_{n,R}$ already was a logical relation on $\texttt{LogrelIn}^{(1)}_n(R), \ldots, \texttt{LogrelIn}^{(n)}_n(R)$ and logical relations are closed under composition with morphisms [RS13, Thm. 5.2].

For the backward direction, we encode every logical relation r on morphisms $m_1, \ldots, m_n\colon R \to S$ as the realization $v\colon \texttt{Logrel}_n(S) \to T$ that contains assignments $v(c^{(i)}) = m_i(c)$ and $v(c^*) = r(c)$ for every $c \in R$. Clearly, well-typedness at assignments to constants $c^{(i)}$ is inherited from well-typedness of morphisms m_i. And at assignments to constants c^* well-typedness follows from r being a logical relation. ◀

As a consequence of Theorem 80, **we can think of $\texttt{Logrel}_n(R)$ as the *interface theory* for logical relations on** R. In the bigger picture, we circumvented introducing a new primitive to Mmt's syntax (incl. corresponding formation, introduction, elimination, and typing rules) by specifying a functor that simply creates appropriate interface theories. Not all primitives can be offloaded this way, but when it can be done, it offers a fast way of prototyping new features.

▶ **Notation 81.** In the special case of $\texttt{Logrel}_1$, i.e., the functor representing *unary* logical relations, we pretend for readability that it did not use any suffices, i.e., applied on a constant c we pretend that it output constants with identifiers c and c^* (instead of $c^{(1)}$ and c^*). In practice, this is what users desire anyway.

4.4.2 Examples

▶ **Example 82** (Representing Type Preservation (cont. Example 13; based on [RR21b])). In Example 13 we started with formalizations `HTyped` and `STyped` of base theories for hard- and soft-typed formal systems and phrased the type erasure translation as a morphism `TypeEras`. For convenience, we copy them below once again.

```
theory Logrel₁(HTyped) = {
    tp  : type
    tp* : tp → type
    tm  : tp → type
    tm* : ΠT : tp. ΠT* : tp* T. tm T → type
}
```

```
mor LogrelIn₁⁽¹⁾(HTyped) : HTyped → Logrel₁(HTyped) = {
    = =
}
```

Figure 14 Interface Theory of Unary Logical Relations on HTyped and Projection

```
theory HTyped = {                 theory STyped = {
    tp : type                         tp   : type
    tm : tp → type                    term : type
}                                     ::   : term → tp → type
                                      Unit : type
                                      unit : Unit
                                  }
```

```
mor TypeEras : HTyped → STyped = {
    = =
}
```

Then, we defined a logical relation TP on TypeEras, i.e., a mathematical function from HTyped- to STyped-syntax. Together with its Basic Lemma, we thus expressed type preservation as a meta theorem on TypeEras. Using our functor, we can now internalize the logical relation (albeit not the meta theorem itself). First, we apply $\mathtt{Logrel}_1$ on HTyped to get the theory shown in Figure 14 (for completeness we also show the output of $\mathtt{LogrelIn}_1^{(1)}$). Second, we define the realization of $\mathtt{Logrel}_1$(HTyped) shown below.

```
mor TypePres : Logrel₁(HTyped) → STyped = {
    = =
}
```

Importantly, this realization represents in one morphism what was previously two entities: the assignments to $\mathtt{tp}^{(1)}$ and $\mathtt{tm}^{(1)}$ represent the type erasure morphism TypeEras, and the assignments to $\mathtt{tp}^*$ and $\mathtt{tm}^*$ represent the type preservation logical relation TP.

Linearity and include preservation of our functor imply the same properties for our representation method: given a structured diagram of theories, $\mathtt{Logrel}_n$ **allows to structure**

logical relations (i.e., meta theorems of certain shape) over those theories in the very same structure. We give an example in the following:

▶ **Example 83** (Extending Type Preservation to Product Types (cont. Example 13 and Example 82; based on [RR21b, Sec. 4] and [RR21b, Sec. 4])). In Example 14 we considered hard- and soft-typed product types and expressed the type preservation property of the corresponding type erasure as a logical relation. We now extend Example 82 to internalize said logical relation as well. Importantly, the mechanics of MMT and our functor go hand in hand and allow the inclusion of `TypePres` in `TypePresProd`. This is valid because the (co)domain of `TypePresProd` is an extension of the (co)domain of `TypePres`.

$$\textbf{mor}\ \texttt{TypePresProd}\colon \texttt{Logrel}_1(\texttt{HProd}) \to \texttt{SProd} = \{$$
$$= =$$
$$\}$$

(For readability, we put the unstarred constants first.)

Qua the morphism above and its well-typedness, we have successfully represented *and* mechanically verified a proof of the type preservation meta theorem from hard- to soft-typed product types using a logical relation. Still, we are missing a way of representing the theorem itself, which would emerge from the logical relation's Basic Lemma. This is tricky and we discuss it as part of future work in Section 4.4.4.

▶ Remark 84 (Peculiarities of Representing Type Preservation (cont. Example 83)). Looking at the morphism in Example 83 representing the logical relation, we can observe two peculiarities.

First, all parameters of type `Unit` are never used. The only reason for this boilerplate is that our representation approach so far only supports total logical relations. This forces us to state relations even at LF types for that we do not desire to prove anything. We extend our approach to *partial* logical relations in Section 4.4.3. This will allow a considerably cleaner representation of type preservation (see Example 88).

Second, the morphism is almost a renaming, i.e., a mapping from constants to constants, if we forget about all parameters that we bind and not use afterwards. This suggests that `SProd` shares a lot of structure with $\texttt{Logrel}_1(\texttt{HProd})$. Consider a hard-typed type-theoretical feature formalized in a theory HX over `HTyped`, and the corresponding soft-typed one in SX over `STyped`. Indeed, for many such features it is generally the case that $\texttt{Logrel}_1(HX)$ is structurally very similar to SX. Concretely, SX tends to be very similar to the pushout of $\texttt{Logrel}_1(HX)$ over $\texttt{TypePres}\colon \texttt{HTyped} \to \texttt{STyped}$. In [RR21b] and the very same setting of `HTyped` and `STyped`, Florian Rabe and the author pursue the underlying observation that logical relations play a central role in this structure similarity, and ultimately they define a functor to automatically derive SX from just HX. In Section 4.5 we will retell their results in our framework.

We bring one last example unrelated to the previous ones:

▶ **Example 85** (Representing Meta Theorems of Propositional Logic (based on [RS13, Ex. 4.1] and [Rou21a, Sec. 4.2])). Consider the abridged formalization of intuitionistic propositional logic shown below on the left. And on the right, we show the result of $\texttt{Logrel}_1$.

$$
\begin{array}{l}
\textbf{theory}\ \texttt{PL} = \{\\
\quad \texttt{prop} \colon \texttt{type}\\
\quad \neg \quad \colon \texttt{prop} \to \texttt{prop}\\
\quad \wedge \quad \colon \texttt{prop} \to \texttt{prop} \to \texttt{prop}\\
\quad \Vdash \quad \colon \texttt{prop} \to \texttt{type}\\
\quad \text{/* /* intuitionistic proof rules */ */}\\
\}
\end{array}
$$

$$
\begin{array}{l}
\textbf{theory}\ \texttt{Logrel}_1(\texttt{PL}) = \{\\
\quad \texttt{prop} \ \colon \texttt{type}\\
\quad \texttt{prop}^* \colon \texttt{prop} \to \texttt{type}\\
\quad \neg \quad \colon \texttt{prop} \to \texttt{prop}\\
\quad \neg^* \quad \colon \Pi\, p \colon \texttt{prop}.\ \Pi\, p^* \colon \texttt{prop}^*\ p.\\
\qquad\qquad \texttt{prop}^*\ (\neg p)\\
\quad \wedge \quad \colon \texttt{prop} \to \texttt{prop} \to \texttt{prop}\\
\quad \wedge^* \quad \colon \Pi\, p \colon \texttt{prop}.\ \Pi\, p^* \colon \texttt{prop}^*\ p.\\
\qquad\qquad \Pi\, q \colon \texttt{prop}.\ \Pi\, q^* \colon \texttt{prop}^*\ q.\\
\qquad\qquad \texttt{prop}^*(p \wedge q)\\
\quad \Vdash \quad \colon \texttt{prop} \to \texttt{type}\\
\quad \text{/* /* output for proof rules */ */}\\
\}
\end{array}
$$

We now represent the meta theorem of *tertium non datur* following [RS13, Ex. 4.1]: in `PL` $p \vee \neg p$ is provable for every, possibly complex, proposition p: `prop` if it is for every atom. To do so, we give the realization `TND` of $\texttt{Logrel}_1(\texttt{PL})$ shown below, which encodes a unary logical relation on $\mathrm{id}_{\texttt{PL}}$. For readability, we *i*) reordered the identity assignments to come first and *ii*) omitted concrete proofs terms (and we refer to [RS13, Ex. 4.1]) .

$$
\begin{array}{l}
\textbf{mor}\ \texttt{TND} \colon \texttt{Logrel}_1(\texttt{PL}) \to \texttt{PL} = \{\\
\quad = =\\
\}
\end{array}
$$

The Basic Lemma for the logical relation represented by `TND` yields that the LF type $\Vdash p \vee \neg p$ is inhabited for every, possibly complex, LF term p: `prop`. In other words, `TND` represents that excluded middle is an admissible rule for the theory `PL`, i.e., in the special case without atomic formulae. Indeed, suppose we had a **theory** $\texttt{PL}' = \{\textbf{include}\ \texttt{PL}, A \colon \texttt{prop}\}$ which extended `PL` with one atomic formula. Then, corresponding realizations of $\texttt{Logrel}_1(\texttt{PL}')$ would have expect assignments of types $A \colon \texttt{prop}$ and $A^* \colon\ \Vdash A \vee \neg A$. Thus, from `TND` "it follows that excluded middle is admissible if it is admissible for all atoms." ([RS13, Ex. 4.1])

As another example, we represent the meta theorem of *double negation elimination* following [Rou21a, Sec. 4.2]: in `PL` the formula $p \Leftrightarrow \neg\neg p$ is provable for every, possibly complex, proposition p: `prop` if it is for every atom. The realization `DNE` given below accomplishes this and is analogous to `TND`.

$$
\begin{array}{l}
\textbf{mor}\ \texttt{DNE} \colon \texttt{Logrel}_1(\texttt{PL}) \to \texttt{PL} = \{\\
\quad = =\\
\}
\end{array}
$$

Analogously to before, it follows that double negation elimination is admissible if it is admissible for all atoms.

4.4.3 Partial Logical Relations

Motivated by Remark 84, we now generalize our functor from Definition 77 to allow for representing *partial* logical relations as well.

▶ **Definition 86** (Partial Logical Relation Operators). *For an arity $n \geq 1$ and a set Θ of constant identifiers, the linear functor* $\texttt{Logrel}_{n,\Theta}$ *is given on all theories and morphisms by*

$$\texttt{Logrel}_{n,\Theta,\Sigma}(c\colon A\,[= t]) = \begin{cases} c^{(1)}\colon A^{(1)}\,[= t^{(1)}] \\ \vdots \\ c^{(n)}\colon A^{(n)}\,[= t^{(n)}] \\ \hline \\ c^{*}: r_{n,\Theta,\Sigma}(A)\ c^{(1)}\ \ \ldots\ \ c^{(n)}\,[= r_{n,\Theta,\Sigma}(t)] \\ \quad \textit{only if } c \notin \Theta \textit{ and } r_{n,\Theta,\Sigma}(A) \neq \bot \end{cases}$$

and for every $1 \leq i \leq n$ the strongly linear connector $\texttt{LogrelIn}^{(i)}_{n,\Theta}$ *into* $\texttt{Logrel}_{n}$ *is given by*

$$\texttt{LogrelIn}^{(i)}_{n,\Sigma}(c\colon A) = c^{(i)}$$

Here, $r_{n,\Theta,\Sigma}$ is the n-ary partial logical relation on $\texttt{LogrelIn}^{(1)}_{n,\Theta,\Sigma}, \ldots, \texttt{LogrelIn}^{(n)}_{n,\Theta,\Sigma}$ *induced by*

$$r_{n,\Theta,\Sigma}(c) = \begin{cases} \bot & \textit{if } c \in \Theta \textit{ or } r_{n,\Sigma}(A) = \bot \textit{ for } (c\colon A) \in \Sigma \\ c^{*} & \textit{otherwise} \end{cases}$$

for every $c \in \Sigma$.

We make $\texttt{Logrel}_{n,\Theta}$ *partial on those constant declarations $c\colon A\ = t$ where t is given, $r_{n,\Theta,\Sigma}(A)$ is defined, but $r_{n,\Theta,\Sigma}(t)$ is not.*

We parametrize our functor with a set of identifiers Θ on which the logical relations to be represented will be undefined for sure. Note that the effective set of partialities may grow as the functor traverses linearly through theories:

▶ **Example 87** (Basic Mechanics of $\texttt{Logrel}_{1,\Theta}$). Consider the following contrived theory:

$$\begin{array}{l} \textbf{theory } T = \{ \\ \quad A\colon \texttt{type} \\ \quad x : A \\ \quad y : A \\ \} \end{array}$$

Below we show results of applying $\texttt{Logrel}_{1,\Theta}$ with various parametrizations of partiality:

$$\begin{array}{l} \textbf{theory } \texttt{Logrel}_{1,\{A\}}(T) = \texttt{Logrel}_{1,\{A,x\}}(T) = \texttt{Logrel}_{1,\{A,y\}}(T) = \texttt{Logrel}_{1,\{A,x,y\}}(T) = \{ \\ \quad A\colon \texttt{type} \\ \quad x : A \\ \quad y : A \\ \} \end{array}$$

theory $\texttt{Logrel}_{1,\{x\}}(T) = \{$

A : type
A^* : $A \to$ type
x : A
y : A
y^* : $A^*\ x$

$\}$

theory $\texttt{Logrel}_{1,\{y\}}(T) = \{$

A : type
A^* : $A \to$ type
x : A
y : A
y^* : $A^*\ y$

$\}$

We observe that once A is put onto the partiality list Θ, then none of A, x, or y receive starred constants that witness them being in the relation at their respective type. In those cases, according to Definition 86 emission of A^* is skipped because $A \in \Theta$, and emission of x^* and y^* is skipped because $r_{n,\Theta,\Sigma}(A) = \bot$. In particular, for the fixed theory T the parametrizations $\Theta = \{A\}, \{A, x\}, \{A, y\}, \{A, x, y\}$ all yield effectively equal functors $\texttt{Logrel}_{1,\Theta}$.

Limitations Our way of representing partial logical relations using a parametrized linear functor allowed for an easy specification in Definition 86, which is also reflected in the implementation. However, it is by no means ideal in terms of user experience. For example, users are forced to specify the parametrization a-priori representing their desired logical relations, namely at time of invoking the functor. Moreover, users could apply differently parametrized functors to obtain theories with equal contents (but different theory identifiers, see Example 87) and have no way of identifying or relating those results. Even though connectors could solve this issue, the inherent design simply fails to scale. Instead, a representation measure for logical relations could offer syntax such as $c^* = \bot$ to allow users to specify partiality ad-hoc.

▶ **Example 88** (Conveniently Representing Type Preservation (cont. Example 83 and Example 15)). In Example 83 we represented the type preservation property from hard- to soft-typed product types using a total logical relation. The totality led to a lot of awkward `Unit` type arguments, which we can now get rid of. We set $\Theta = \{\texttt{tp}\}$ and compute $\texttt{Logrel}_{1,\Theta}(\texttt{HTyped})$ and $\texttt{Logrel}_{1,\Theta}(\texttt{HProd})$ as shown in Figure 15. We now formalize type preservation on `HTyped` and `HProd` as shown below, choosing the same morphism identifiers as in Example 83 for convenience.

mor TypePres : $\texttt{Logrel}_{1,\Theta}(\texttt{HTyped}) \to \texttt{STyped} = \{$

$= =$

$\}$

mor TypePresProd : $\texttt{Logrel}_{1,\Theta}(\texttt{HProd}) \to \texttt{SProd} = \{$

$= =$

$\}$

The structural similarity between `SProd` and $\texttt{Logrel}_{1,\Theta}(\texttt{HProd})$ is now even more visible than before. The morphism `TypePresProd` suggests that the only thing distinguishing `SProd` and $\texttt{Push}_{\texttt{TypePres}}(\texttt{Logrel}_{1,\Theta}(\texttt{HProd}))$ is that `pair`, `projL`, and `projR` do not take type arguments in the former, but do in the latter. We come back to this observation in **??**.

$$
\begin{array}{l}
\textbf{theory}\ \texttt{Logrel}_{1,\Theta}(\texttt{HTyped}) = \{\\
\quad \texttt{tp}\ : \texttt{type}\\
\quad \texttt{tm}\ : \texttt{tp} \to \texttt{type}\\
\quad \texttt{tm}^* : \Pi\, T\colon \texttt{tp}.\ \texttt{tm}\ T \to \texttt{type}\\
\}
\end{array}
$$

$$
\begin{array}{l}
\textbf{theory}\ \texttt{Logrel}_{1,\Theta}(\texttt{HProd}) = \{\\
\quad \textbf{include}\ \texttt{Logrel}_{1,\Theta}(\texttt{HTyped})\\
\quad \texttt{prod}\ \ \ : \texttt{tp} \to \texttt{tp} \to \texttt{tp}\\
\quad \texttt{pair}\ \ \ : \Pi\, a\ b\colon \texttt{tp}.\ \texttt{tm}\ a \to \texttt{tm}\ b \to \texttt{tm}\ \texttt{prod}\, a\, b\\
\quad \texttt{pair}^*\ : \Pi\, a\ b\colon \texttt{tp}.\ \Pi\, x\colon \texttt{tm}\ a.\ \Pi\, x^*\colon \texttt{tm}^*\ a\ x.\\
\qquad\qquad \Pi\, y\colon \texttt{tm}\ b.\ \Pi\, y^*\colon \texttt{tm}^*\ b\ y.\\
\qquad\qquad \texttt{tm}^*\ (\texttt{prod}\ a\ b)\ (\texttt{pair}\ a\ b\ x\ y)\\
\quad \texttt{projL}\ : \Pi\, a\ b\colon \texttt{tp}.\ \texttt{tm}\ \texttt{prod}\, a\, b \to \texttt{tm}\ a\\
\quad \texttt{projL}^* : \Pi\, a\ b\colon \texttt{tp}.\ \Pi\, p\colon \texttt{tm}\ \texttt{prod}\, a\, b.\\
\qquad\qquad \Pi\, p^*\colon \texttt{tm}^*\ (\texttt{prod}\, a\, b)\ p.\ \texttt{tm}^*\ a\ (\texttt{projL}\, a\, b\, p)\\
\quad \texttt{projR}\ : \Pi\, a\ b\colon \texttt{tp}.\ \texttt{tm}\ \texttt{prod}\, a\, b \to \texttt{tm}\ b\\
\quad \texttt{projR}^* : \Pi\, a\ b\colon \texttt{tp}.\ \Pi\, p\colon \texttt{tm}\ \texttt{prod}\, a\, b.\\
\qquad\qquad \Pi\, p^*\colon \texttt{tm}^*\ (\texttt{prod}\, a\, b)\ p.\ \texttt{tm}^*\ a\ (\texttt{projR}\, a\, b\, p)\\
\}
\end{array}
$$

$$
\begin{array}{l}
\textbf{mor}\ \texttt{LogrelIn}^{(1)}_{1,\Theta}(\texttt{HProd}) \colon \texttt{HProd} \to \texttt{Logrel}_{1,\Theta}(\texttt{HProd}) = \{\\
\quad = =\\
\}
\end{array}
$$

Figure 15 Interface theories for logical relations on `HProd` that are partial on $\Theta = \{\texttt{tp}\}$

We prove meta-theoretical of the functor and connectors as before. The only critical step hides in the last sentence of Definition 86, which guarantees term-totality of r_{n,Θ,Σ^*} whenever $\texttt{Logrel}_{n,\Theta}$ is defined.

▶ **Theorem 89.** $\texttt{Logrel}_{n,\Theta}$ *is well-typed and functorial.*

Proof. Similar to the total case outlined in the proof of Theorem 78. ◀

▶ **Theorem 90.** *The connectors* $\texttt{LogrelIn}^{(i)}_{n,\Theta}$ *are well-typed and natural.*

Proof. By Theorem 38 in conjunction with Theorem 89. ◀

▶ **Theorem 91** (Adequacy). *For $n \geq 1$, identifier set Θ, and any flat theory R, we have the bijection of sets*

$$
\textit{realizations of}\ \texttt{Logrel}_{n,\Theta}(R) \ \cong\ \begin{array}{l}\textit{n-ary term-total logical relations on } R\\ \textit{that are partial at least on } \Theta \textit{ and otherwise}\\ \textit{maximally total}\end{array}
$$

Proof. Similar to the total case outlined in the proof of Theorem 80. ◀

The last theorem statement reads a bit awkwardly precisely due to the remarks mentioned after Definition 86.

4.4.4 Related and Future Work: TODO

Related Work We outline and compare three different approaches of representing logical relations in MMT or MMT-near systems:

A a new module kind next to theories and morphisms [RS13]

$$\textbf{rel}\ r\colon m_1 \times \ldots \times m_n\colon R \to S = \{\ldots\}$$

B realizations of $\texttt{LogrelAlong}_{m_1,\ldots,m_n\colon R\to S}(X)$ [Rou21a]
C realizations of $\texttt{Logrel}_n(X)$ (this thesis)

The first approach by Rabe and Sojakova adds support for logical relations by extending the module system with a new kind of module. Every **rel** module takes an identifier (here: r), a list of morphisms with common domain and codomain, and, akin to morphisms, a body that for every constant $c \in S$ declares an assignment to c. The second approach, first sketched by the author in [Rou21a], uses a functor that is parametrized by a list of morphisms. It maps every theory $X \hookrightarrow R$ to the theory $\texttt{LogrelAlong}_{m_1,\ldots,m_n}(X)$ whose realizations are precisely the n-ary logical relations on the restricted morphisms $m_1|_X, \ldots, m_n|_X\colon X \to S$. Concretely, for every undefined constant $c \in X$ this theory contains an undefined constant declaration c^* whose type is exactly the type that would have been expected in an assignment to c in an n-ary logical relation on $m_1|_X, \ldots, m_n|_X$. For this reason, $\texttt{LogrelAlong}_{m_1,\ldots,m_n}(X)$ acts as an interface theory and any morphism out of it encodes precisely the assignments necessary for a logical relation on $m_1, \ldots, m_n$. Finally, the third approach generalizes the second one by getting rid of the morphisms as a parameter. It does so by creating even more constants in the interface theory. For every $c \in X$ the interface theory $\texttt{Logrel}_n(X)$ contains undefined constants $c_1, \ldots, c_n, c^*$. Here, the first n constants are just qualified copies of c forcing realizations $r\colon \texttt{Logrel}_n(X) \to S$ to encode n morphisms $m_1, \ldots, m_n\colon X \to S$. And the constant c^* has the same purpose as in the second approach.

The considerable advantage of approaches B and C is that they forgo the need to add anything to MMT's syntax (except syntax for diagram operators – which is useful anyway as the present thesis hopefully makes clear). Moreover, these approaches nicely interplay with the presence of other functors without having to implement any additional business logic. For example, we can directly apply the pushout operator to realizations in approaches B and C. See Section 4.5 for a larger case study on that.

Future Work All approaches listed above, incl. ours, only represent logical relations, i.e., proofs of meta theorems of certain shape. None of them allows the user to actually use the Basic Lemmas resulting from logical relations represented in those ways, something that is desired in practice. Adding support for that is a non-trivial task, as it may lead to all the usual issues that appear when adding reflection capabilities to a formal system.

Our representation approach for partial logical relation forces users to awkwardly and explicitly state the constants on which logical relations should be undefined. We already dicussed this right after Example 87.

4.5 Translating Formalizations of Type Theory from Intrinsic to Extrinsic Style

$$\texttt{Soften} = \texttt{Clean}_f \circ \texttt{Push}_{\texttt{TypePres}} \circ \texttt{Logrel}_{1,\Theta}$$

$$\begin{tikzcd}
\mathtt{HTyped} \arrow[r] \arrow[d, hook] & \mathtt{Logrel}_{1,\Theta}(\mathtt{HTyped}) \arrow[r, "\mathtt{TypePres}"] \arrow[d, hook] & \mathtt{STyped} \arrow[r, "\mathrm{id}_{\mathtt{STyped}}"] \arrow[d, hook] & \mathtt{STyped} \arrow[d, hook] \\
HX \arrow[r] & \mathtt{Logrel}_{1,\Theta}(HX) \arrow[r] & \mathtt{Push}_{\mathtt{TypePres}}(\mathtt{Logrel}_{1,\Theta}(HX)) \arrow[r] & SX
\end{tikzcd}$$

The linear functor `Soften` translates formalizations of type-theoretical features from intrinsic (hard-typed) to extrinsic (soft-typed) style. The underlying translation is based on joint work of Florian Rabe and the author in [RR21b]. We define it as the composition of three previous functors: logical relation $\mathtt{Logrel}_{1,\Theta}$ from Section 4.4.3, pushout `Push` from Section 4.1, and parameter removal $\mathtt{Clean}_f$ from Section 4.3.

4.5.1 Motivation of Case Study

This case study is interesting for the present work for two reasons. First, we are able to phrase the translation from [RR21b] as a functor composed modularly out of three reusable functors from our framework: logical relation $\mathtt{Logrel}_{1,\Theta}$ from Section 4.4.3, pushout `Push` from Section 4.1, and parameter removal $\mathtt{Clean}_f$ from Section 4.3. Thus, this case study shows that we can **modularly build powerful functors from a basic set of reusable functors**. Second, we use **logical relations *for data***, in contrast to them being conventionally used just *for proof*. And our way of representing logical relations as morphisms naturally **blurs the dinstiction between logical relations for proof and for data**, which is an interesting phenomenon on its own. By postcomposing $\mathtt{Logrel}_{1,\Theta}$ with `Push`, we effectively transport data along a logical relation. This can be seen as a generalization of "conventional" pushouts that transport data along morphisms (without any other functors at play). Translating from intrinsic to extrinsic style is precisely a case where such morphisms are too inexpressive of a translation for the desired transport of data, necessating logical relations.[10] We refer to [RR21b] for an explanation why logical relations are a good fit for the translation occurring here.

4.5.2 Heading Towards a Definition

In Examples 82, 83, and 88 we have shown, given formalizations of hard-typed and soft-typed type-theoretical features (e.g., product types), how we can represent the type erasure and preservation property from hard-typed to soft-typed using the functors developed in this section. Our presentation culminated in Example 88 where we gave morphisms $\mathtt{TypePres}\colon \mathtt{Logrel}_{1,\Theta}(\mathtt{HTyped}) \to \mathtt{STyped}$ and $\mathtt{TypePresProd}\colon \mathtt{Logrel}_{1,\Theta}(\mathtt{HProd}) \to \mathtt{SProd}$, both representing logical relations. We critically observed how the latter is pretty dull: except for specialties regarding superfluous parameters, the morphism is a renaming. This motivated seeing `SProd` (and every other soft-typed feature) to be systematically derivable from the corresponding hard-typed feature. In the this section, we retell parts of these results from Florian Rabe and the author [RR21b] in light of the framework developed in the present thesis.

The procedure is roughly as follows, see the diagram below (unlabeled arrows denote the canonical connectors into the respective functors). Let `HTyped` and `STyped` be the base theories of hard- and soft-typed features from Example 82. Initially, compute $\mathtt{Logrel}_{1,\Theta}(\mathtt{HTyped})$

[10] In fact, logical relations can be seen as second-order morphisms (an idea communicated by Florian Rabe). And in general, we can spawn an entire hierarchy of higher-order compositional translations, for all of which we may formulate pushouts.

with $\Theta = \{\mathtt{tp}\}$ and represent type preservation as a logical relation in form of the morphism `TypePres` from Example 88. Now consider any hard-typed feature HX. To translate it to a corresponding soft-typed feature, we first compute $\mathtt{Logrel}_{1,\Theta}(HX)$ then compute the pushout along `TypePres`, and finally take care of removing superfluous parameters using the functor `Clean` from Section 4.3. This way we obtain the intended representation SX of the soft-typed feature.

Except for parameter removal, the procedure should already be clear from Example 88. We now explain the missing piece. Let us abbreviate $\mathtt{LP} = \mathtt{Logrel}_{1,\Theta}\,;\,\mathtt{Push}_{\mathtt{TypePres}}$, a linear functor from `HTyped` to `STyped`. To see what goes wrong when just trying to use `LP` for softening, consider the theory `LP(HProd)` shown below. The source theory `HProd` together with `SProd` were presented in Example 83.

$$
\begin{array}{l}
\textbf{theory}\ \mathtt{LP}(\mathtt{HProd}) = \{\\
\quad \textbf{include}\ \mathtt{STyped}\\
\quad \mathtt{prod}\ \ : \mathtt{tp} \to \mathtt{tp} \to \mathtt{tp}\\
\quad \mathtt{pair}\ \ : \Pi\, a\ b\colon \mathtt{tp}.\ \mathtt{term} \to \mathtt{term} \to \mathtt{term}\\
\quad \mathtt{pair}^*\ : \Pi\, a\ b\colon \mathtt{tp}.\ \Pi\, x\colon \mathtt{term}.\ \Pi\, x^*\colon x :: a.\\
\qquad\qquad\quad \Pi\, y\colon \mathtt{term}.\ \Pi\, y^*\colon y :: b.\\
\qquad\qquad\quad (\mathtt{pair}\, a\, b\, x\, y) :: (\mathtt{prod}\, a\, b)\\
\quad \mathtt{projL}\ : \Pi\, a\ b\colon \mathtt{tp}.\ \mathtt{term} \to \mathtt{term}\\
\quad \mathtt{projL}^*\colon \Pi\, a\ b\colon \mathtt{tp}.\ \Pi\, p\colon \mathtt{term}.\\
\qquad\qquad\quad \Pi\, x^*\colon p :: \mathtt{prod}\, a\, b.\ (\mathtt{projL}\, a\, b\, p) :: a\\
\quad \mathtt{projR}\ : \Pi\, a\ b\colon \mathtt{tp}.\ \mathtt{term} \to \mathtt{term}\\
\quad \mathtt{projR}^*\colon \Pi\, a\ b\colon \mathtt{tp}.\ \Pi\, p\colon \mathtt{term}.\\
\qquad\qquad\quad \Pi\, x^*\colon p :: \mathtt{prod}\, a\, b.\ (\mathtt{projR}\, a\, b\, p) :: b\\
\}
\end{array}
$$

The above theory is almost equal to the desired formalization `SProd` except that some translated constants (`pair`, `projL`, `projR`) feature undesired type parameters. This issue is not unique to product types. Consider Figure 16, where we collect an exemplary diagram `HDiag` of hard-typed features. Figure 17 shows the corresponding soft-typed variants in a diagram `SDiag` that we intend to obtain by applying our functor. Let us compare every constant c in `HDiag` with its equinamed constant in `SDiag`. Concerning type parameters we can distinguish the following cases:

- removal desired: $\mathtt{pair}\colon \Pi\, a\ b.\ \mathtt{tm}\ a \to \mathtt{tm}\ b \to \mathtt{tm}\ \mathtt{prod}\, a\, b$ should go to $\mathtt{pair}\colon \mathtt{term} \to \mathtt{term} \to \mathtt{term}$; analogously for `app`
- removal optional depending on the intended result, e.g., $\mathtt{eq}\colon \Pi\, a.\ \mathtt{tm}\ a \to \mathtt{tm}\ a \to \mathtt{prop}$ can go to $\mathtt{eq}\colon \mathtt{term} \to \mathtt{term} \to \mathtt{prop}$ or to $\mathtt{eq}\colon \Pi\, a.\ \mathtt{term} \to \mathtt{term} \to \mathtt{prop}$; analogously $\mathtt{lam}\colon \Pi\, a\ b.\ (\mathtt{tm}\ a \to \mathtt{tm}\ b) \to \mathtt{tm}\ \mathtt{fun}\, a\, b$ can go to $\mathtt{lam}\colon (\mathtt{term} \to \mathtt{term}) \to \mathtt{term}$ or to $\mathtt{lam}\colon \Pi\, a.\ (\mathtt{term} \to \mathtt{term}) \to \mathtt{term}$
- removal undesired, e.g., $\mathtt{dfun}\colon \Pi\, a\colon \mathtt{tp}.\ (\mathtt{tm}\ a \to \mathtt{tp}) \to \mathtt{tp}$ should go to $\mathtt{dfun}\colon \Pi\, a.\ (\mathtt{term} \to \mathtt{tp}) \to \mathtt{tp}$; consequently the analogous parameter must be kept in `dlam` and `dapp`

Removing selected type parameters by post-composing `LP` with the `Clean` functor from Section 4.3 is straightforward and presented in the following. As the list above may suggest, the major problem is identifying these parameters in the first place, and thus the definition of a suitable parameter keep heuristic.

Much to our surprise and frustration, automatically choosing an appropriate heuristic f turned out to be difficult:

▶ **Example 92.** The undesired argument positions in `LP(HProd)` are exactly the named variables in `HProd` that do not occur in their scopes in `LP(HProd)` anymore. This includes the positions $\texttt{pair}^1$ and $\texttt{pair}^2$, and removing them yields the desired declaration of `pair` in `SProd`.

However, that does not hold for `HDepFun`. Here the argument $\texttt{dfun}^1$ is named in `HDepFun` and unused in the declaration $\texttt{dfun}\colon \Pi\, a\colon \texttt{tp}.\ (\texttt{term} \to \texttt{tp}) \to \texttt{tp}$ that occurs in `LP(HDepFun)`. However, that is in fact the desired formalization of the soft-typed dependent function type. Removing $\texttt{dfun}^1$ would yield the undesired $\texttt{dfun}\colon (\texttt{term} \to \texttt{tp}) \to \texttt{tp}$. While we do not mention MMT's implicit arguments in this paper, note also that $\texttt{dfun}^1$ is an *implicit* argument in `HDepFun` that must become *explicit* in `SDepFun`.

This is trickier than it sounds because some argument positions may only be removable if they are removed at the same time; so a fixpoint iteration might be necessary. Moreover, picking the smallest possible f (i.e., $f^\Sigma(c) \equiv \varnothing$) is entirely wrong as it would remove all argument positions. At the very least, we should only remove *named* argument positions, i.e., those that are bound by a named variable (as opposed to the anonymous variables introduced by parsing, e.g., $\texttt{prod}\colon \texttt{tp} \to \texttt{tp} \to \texttt{tp}$). A smarter choice is to remove all named argument positions that become redundant through the pushout along the type erasure morphism (as part of `TypePres`), i.e., positions for constants c that are named and used in `HProd` but unused in the equinamed declaration in `LP(HProd)`. (Note that in general pushouts of constants have at least the argument positions that they previously had. They may have more if the morphism over which pushouts are computed maps an atomic type to a function type.) That is the right choice almost all the time but not always.

After several failed attempts, we have been unable to find a good heuristic for choosing f. For now, we aim to remove all named variables that after pushout no longer occur in their scope, and we allow users to annotate constants like $\texttt{@keep}(\texttt{dfun}^1)$ where the system should deviate from that heuristic (see Figure 16). Formally, we have to extend our category of theories and morphisms suitably to allow constants declarations to be annotated. We anticipate finding better solutions after collecting more data in the future.

4.5.3 Definition

We omitted one technicality in the last section. Namely, to remove all named variables that after pushout no longer occur in their scope, it is necessary for the translation to look back at what the original constant was. This is easily accomplished when softening is phrased as one self-contained translation (as in [RR21b]), but is a minor technical hurdle when softening is built up modularly as the composition of multiple functors (as in the present thesis). To solve this problem, we make again clever use of annotations. First, we modify $\texttt{Logrel}_{1,\Theta}$ to preserve a snapshot of the original constants in form of annotations in the output constants (see first downwards arrow in Figure 18). Second, we modify $\texttt{Push}_m$ to preserve annotations of input constants (see second downwards arrow in Figure 18). Third, we specify a heuristic for $\texttt{Clean}_f$ that makes use of the annotations preserved this way (see definition below).

▶ **Definition 93** (Softening). *The linear functor* `Soften` *from* `HTyped` *to* `STyped` *is given by*

$$\texttt{Soften} = \texttt{Logrel}_{1,\Theta}\ ;\ \texttt{Push}_{\texttt{TypePres}}\ ;\ \texttt{Clean}_f$$

where ; denotes composition in diagrammatic order and where the heuristic f for $\texttt{Clean}_f$ *is*

```
theory HEqual =
  include HTyped
  @keep(eq^1)
  eq    : Π a. tm a → tm a → prop
  refl  : Π a x.  ⊩ eq a x x
  eqsub : Π a x y.  ⊩ eq a x y →
          Π F : tm a → prop.  ⊩ F x →⊩ F y

theory HFun =
  include HEqual
  fun : tp → tp → tp
  @keep(lam^1)
  lam : Π a b. (tm a → tm b) → tm fun a b
  app : Π a b. tm fun a b → tm a → tm b
```

```
theory HDepFun =
  include HEqual
  @keep(dfun^1)
  dfun : Π a. (tm a → tp) → tp
  @keep(dlam^1)
  dlam : Π a. Π b : tm a → tp. (Π x : tm a. tm b x)
         → tm dfun a b
  dapp : Π a b. tm dfun a b → Π x : tm a. tm b x
```

```
theory HBeta =
  include HFun
  beta : Π a b. Π F : tm a → tm b. Π x.
         ⊩ eq b (app a b (lam a b F) x) (F x)

theory HEta =
  include HFun
  eta : Π a b. Π f : tm fun a b.
        ⊩ eq (fun a b) f (lam a b λx. app f x)
```

```
theory HExten =
  include HFun
  exten : Π a b. Π f g : tm fun a b.
          (Π x.  ⊩ eq b (app a b f x)
          (app a b g x)) →⊩ eq (fun a b) f g

theory HDepBeta =
  include HDepFun
  dbeta : Π a b. Π F : (Π x : tm a. tm b x). Π x.
          ⊩ eq (b x) (dapp a b (dlam a b F) x) (F x)
```

Figure 16 Theories for Function Types with Annotations for Needed Parameters

```
theory SEqual =
  include STyped
  eq      : Π a. term → term → prop
  refl*   : Π a x. ⊩ x :: a →⊩ eq a x x
  eqsub*  : Π a. Π x. Π x*: ⊩ x :: a.
            Π y. Π y*: ⊩ y :: a.
            ⊩ eq a x y →
            Π F: term → prop.
            ⊩ F x →⊩ F y

theory SFun =
  include SEqual
  fun  : tp → tp → tp
  lam  : Π a. (term → term) → term
  lam* : Π a b. Π F: term → term.
         (Π x. ⊩ x :: a →⊩ (F x) :: b)
         →⊩ (lam a F) :: (fun a b)
  app  : term → term → term
  app* : Π a b. Π f. ⊩ f :: (fun a b) →
         Π x. ⊩ ::x a →⊩ (app f x) :: b
```

```
theory SDepFun =
  include SEqual
  dfun  : Π a. (term → tp) → tp
  dlam  : Π a. (term → term) → term
  dlam* : Π a. Π b: term → tp. Π F: term → term.
          (Π x. ⊩ x :: a →⊩ (F x) :: (b x))
          →⊩ (dlam a b F) :: (dfun a b)
  dapp  : term → term → term
  dapp* : Π a b. Π f. ⊩ f :: (dfun a b) →
          Π x. ⊩ x :: a →⊩ (dapp f x) :: (b x)
```

```
theory SBeta =
  include SFun
  beta* : Π a b. Π F: term → term.
          (Π x. ⊩ x :: a →⊩ (F x) :: b)
          → Π x. ⊩ x :: a →
          ⊩ eq b (app (lam a F) x) (F x)
```

```
theory SExten =
  include SFun
  exten* : Π a b. Π f: term. ⊩ f :: (fun a b) →
           Π g: term. ⊩ g :: (fun a b) →
           (Π x. ⊩ x :: a →⊩ eq b (app f x) (app g x))
           →⊩ eq (fun a b) f g
```

```
theory SEta =
  include SFun
  eta* : Π a b. Π f: term. ⊩ f :: (fun a b)
         ⊩ eq (fun a b) f (lam a λx. app f x)
```

```
theory SDepBeta =
  include SDepFun
  dbeta* : Π a b. Π F: term → term.
           (Π x. ⊩ x :: a →⊩ (F x) :: (b x)) →
           Π x. ⊩ x :: a →
           ⊩ eq (b x) (dapp (dlam a F) x) (F x)
```

Figure 17 Intended Result of Softening the Theories from Figure 16

$\texttt{@keep}(\texttt{lam}^1)$

$\texttt{lam}\colon \Pi\, a\; b.\; (\texttt{tm}\; a \to \texttt{tm}\; b) \to \texttt{tm}\; \texttt{fun}\, a\, b$

$\downarrow \texttt{Logrel}_{1,\ominus}$

$\texttt{@original}(\texttt{@keep}(\texttt{lam}^1)\; \texttt{lam}\colon \Pi\, a\; b.\; (\texttt{tm}\; A \to \texttt{tm}\; b) \to \texttt{tm}\; \texttt{fun}\, a\, b)$

$\texttt{lam}\colon \Pi\, a\; b.\; (\texttt{tm}\; a \to \texttt{tm}\; b) \to \texttt{tm}\; \texttt{fun}\, a\, b$

$\texttt{lam}^*\colon \Pi\, a\; b.\; \Pi\, F\colon \texttt{tm}\; a \to \texttt{tm}\; b.\; (\Pi\, x.\; \Vdash \texttt{tm}^*\, a\, x \to \texttt{tm}^*\, b\, (F\, x) \to \texttt{tm}^*\, (\texttt{fun}\, a\, b)\, (\lambda\, .\; a\, F)$

$\downarrow \texttt{Push}_{\texttt{TypePres}}$

$\texttt{@original}(\texttt{@keep}(\texttt{lam}^1)\; \texttt{lam}\colon \Pi\, a\; b.\; (\texttt{tm}\; A \to \texttt{tm}\; b) \to \texttt{tm}\; \texttt{fun}\, a\, b)$

$\texttt{lam}\colon \Pi\, a\; b.\; (\texttt{term} \to \texttt{term}) \to \texttt{term}$

$\texttt{lam}^*\colon \Pi\, a\; b.\; \Pi\, F\colon \texttt{term} \to \texttt{term}.\; (\Pi\, x.\; \Vdash x :: a \to (F\, x) :: b \to (\lambda\, .\; a\, b\, F) :: (\texttt{fun}\, a\, b)$

$\downarrow \texttt{Clean}_f$

$\texttt{lam}\colon \Pi\, a.\; (\texttt{term} \to \texttt{term}) \to \texttt{term}$

$\texttt{lam}^*\colon \Pi\, a\; b.\; \Pi\, F\colon \texttt{term} \to \texttt{term}.\; (\Pi\, x.\; \Vdash x :: a \to (F\, x) :: b \to (\lambda\, .\; a\, F) :: (\texttt{fun}\, a\, b)$

Figure 18 Annotations over the course of functor application

defined by

$$f\begin{pmatrix} \texttt{@original}(\texttt{@keep}(U)\; c\colon \Pi\, x_1\colon A_1.\; \ldots\, \Pi\, x_n\colon A_n.\; B\, [= \lambda\, x_1\colon A_1.\; \ldots\, \lambda\, x_n\colon A_n.\; t]) \\ c\colon \Pi\, x_1\colon A_1.\; \ldots\, \Pi\, x_n\colon A_n.\; B\, [= \lambda\, x_1\colon A_1.\; \ldots\, \lambda\, x_n\colon A_n.\; t] \end{pmatrix}$$
$$= \{1, \ldots, n\} \quad \{x_i \mid x_i \textit{ used in original type or def., but not in current one}\}$$

Correspondingly, the connector $\texttt{SoftenIn}\colon \texttt{Id} \to \texttt{Soften}$ *is given by*

$$\texttt{SoftenIn} = \texttt{LogrelIn}^{(1)}_{1,\ominus}\; ;\; \texttt{PushIn}_{\texttt{TypePres}}\; ;\; \texttt{CleanIn}_f$$

Since `Soften` and `SoftenIn` are defined by composition of well-typed functors and well-typed natural connectors, respectively, they immediately inherit the same properties:

▶ **Theorem 94.** `Soften` *is well-typed and functorial.*

▶ **Theorem 95.** `SoftenIn` *is well-typed and natural.*

Limitation: Translating Proof Rules Applied to Figure 16, Definition 93 yields the intended result for all constants except those representing proof rules (e.g., constants `refl`, `eqsub`, `beta`). Intuitively, this can be easily fixed and we refer to [RR21b, Sec. 3.4] for the idea. But formally, it would require giving a novel definition generalizing even partial logical relations, being outside the scope of the present thesis.

5 Operators for Universal Algebra

5.1 Introduction

INTEGRATE FEEDBACK FOUND IN “2021-10-25-FEEDBACK-THESIS-DRAFT.txt”!

$$\begin{tikzcd}
 & & & \mathrm{Hom}(X) \arrow[dl, bend right, "\mathrm{id}"'] & \\
\mathrm{Prod}(n;X) \arrow[d, Rightarrow, "\mathrm{par}"', "\mathrm{mod}"] & & X \arrow[ll, Rrightarrow, "d_1,\ldots,d_n"'] \arrow[dl, Rrightarrow, "d_1,\ldots,d_n"] \arrow[ur, shift left, "\mathrm{dom}"] \arrow[ur, shift right, "\mathrm{cod}"'] \arrow[rr, shift left, "\mathrm{par}"] \arrow[rr, shift right, "\mathrm{quot}"'] \arrow[dr, shift left, "\mathrm{par}"] \arrow[dr, shift right, "\mathrm{mod}"'] & & \mathrm{Cong}(X) \arrow[ul, "\mathrm{ker}"'] \\
\mathrm{Sub}(\mathrm{Prod}(n;X)) \arrow[r, dash, dashed, "\mathrm{isomorphic}"'] & \mathrm{ModRel}(n;X) & & \mathrm{Sub}(X) \arrow[uu, "\mathrm{img}"] &
\end{tikzcd}$$

The mathematical field of algebra describes mathematical structures – such as monoids, groups, and vectorspaces, etc., – that are ubiquitous throughout all formal sciences including mathematics and all its subfields, computer science, physics, and chemistry. Thus, formalizations of algebra are inevitable for even the most basic formalization endeavours in any of these fields, making it an integral part of every standard library for formal systems.

Algebra equips all these structures with various corresponding constructions, such as homomorphisms, substructures, congruence structures, kernels and images of homomorphisms, submodels, and quotient models, etc. Typically, all these notions are given for each and every structure separately, e.g., they are given for monoids, groups, vectorspaces, etc. Thanks to **Universal Algebra** we know that many of these notions can be systematically derived from a syntactic description of the algebra structure itself. For example, monoid homomorphisms are functions between two monoids that preserve all of the domain monoid's operations, being the binary operation and the neutral element; similarly group homomorphisms additionally preserve the inverse operation; and vector space homomorphisms are functions between two vector spaces that preserve the operations of scalar multiplication and vector addition. Universal algebra classifies mathematical structures into signatures, making explicit a syntax to describe, e.g., the concept of monoids, groups, etc. And then it defines **universal constructions** (e.g., of homomorphisms, etc.) that for every signature induce exactly the ones mentioned above (e.g., of monoid, group and vectorspace homomorphisms).

> These induced constructions and their properties are both universal and constructive. Thus they are ripe for being automated by meta-programming in standard libraries. ([**carette:diagops**])

However, many formalization libraries still handcraft these constructions for lack of suitable meta-programming support. We hope to address this with our framework of diagram operators.

Contribution Our contribution is two-fold. First, we conduct the **case study of phrasing universal constructions as linear operators**. Using a shallow embedding, we represent algebra theories and their signatures as MMT theories and organize them into the large structured diagram `Alg` shown in **??**. Then, we phrase constructions such as homomorphisms, substructures, and congruences as linear functors, and constructions such as kernels and images of homomorphisms as linear connectors. For every theory T, we systematically obtain the following diagram:

$$\begin{tikzcd}
\texttt{Sub}(T) \arrow[dr, "\texttt{Img}"] & \\
& \texttt{Hom}(T) \\
\texttt{Cong}(T) \arrow[ur, "\texttt{Ker}"] &
\end{tikzcd}$$

Applying these diagram operators to `Alg` as a whole, we obtain large, structured, and interrelated diagrams `Hom`(`Alg`), `Sub`(`Alg`), `Cong`(`Alg`), and interrelations `Ker`(`Alg`) and `Img`(`Alg`). We can also nest these functors, e.g., to acquire the theory `Hom`(`Sub`(`Group`)) representing the theory of homomorphisms between subgroups.

Second, **our way of defining these constructions is novel and generalizes** the setting in which universal algebra is typically developed. Typically, these constructions are given for signatures over first-order logic with just function and predicate symbols because that is sufficient to represent typical algebra theories. Using the machinery of **logical relations** (see Section 4.4, [RS13]), we generalize this setting in two directions: First, we extend some constructions to signatures over sorted, polymorphic, and dependently-typed first-order logic, all of which include function, predicate, *and* axiom symbols. Second, we extend some constructions to signature morphisms, which implies that our translations also need to emit proofs, We are not aware of any descriptions of operators formalizing the constructions at this level of generality.

Overview As a preliminary, in Section 5.2 we discuss different variants of representing algebra theories in type theories (shallow vs. deep embeddings) and show how we use MMT theories to realize a shallow embedding. Importantly, we present our main case study in Section 5.2.4: the algebraic hierarchy of LATIN2. Our main contribution is given in Sections 5.3–5.7, where we define operators `Hom`, `Sub`, `Cong`, `Ker`, and `Img` in this order. We recommend to first read `Hom` as a primer on our way of using logical relations to specify complex translations in universal algebra. Afterwards, `Sub` and `Cong` can be read independently. To read about the connectors `Ker`: `Cong` → `Hom` and `Img`: `Sub` → `Hom`, we naturally recommend reading about their domain and codomain functors first. Finally, we conclude in Section 5.8, where we in particular share learnings in how logical relations helped *and* partially lacked in specifying complex translations in universal algebra. Moreover, we list some possible avenues of future work, culminating in the diagram of operators listed at the beginning of this section.

5.2 Representing Algebra Theories and Related Work

Overall, we use a shallow embedding and use certain well-patterned MMT theories extending `SFOL` to represent algebra theories. We discuss all of that at length: In Section 5.2.1 we recap and compare different type-theoretical approaches of representing algebra theories in formalizations. Having settled with a shallow embedding, in Section 5.2.2 we can use MMT theories to realize polymorphic and/or dependently-typed variants of first-order logic that MMT/LF naturally induces on `SFOL`-extensions. Then, in Section 5.2.3 we discuss related work concerning the preceding sections. And finally, in Section 5.2.4 we present our overarching example: the algebraic hierarchy in LATIN2.

Throughout all sections the following algebra theory serves as a concise running example:

▶ **Definition 96** (Unitals). *Mathematically (and with mathematical notation), a* ***unital*** *is a set U equipped with a binary operation $\circ\colon U \to U \to U$ and a designated element $e \in U$, fulfilling the axiom $\forall x \in U.\ e \circ x = x$.*

▶ **Example 97** (Unitals in MMT). We can represent Definition 96 in MMT as the below theory. This is a shallow embedding, as we will explain in Section 5.2.1.

$$
\begin{array}{l}
\textbf{theory}\ \texttt{Unital} = \{\\
\quad \textbf{include}\ \texttt{SFOL}\\
\quad U \quad\ \colon \texttt{tp}\\
\quad \circ \quad\ \colon \texttt{tm}\ U \to \texttt{tm}\ U \to \texttt{tm}\ U\\
\quad e \quad\ \colon \texttt{tm}\ U\\
\quad \texttt{neut}\colon \Vdash \forall\, x\colon \texttt{tm}\ U.\ e \circ x \doteq x\\
\}
\end{array}
$$

5.2.1 Shallow and Deep Embeddings

In general, there are two ways of representing algebra theories in type theories: using a *shallow embedding* or a *deep embedding*. To be clear, we first settle on some terminology: An **algebra theory** is a collection of abstract types, function and predicate symbols, and axiom symbols. Examples include the abstract concepts of monoids, groups, and vectorspaces (but, e.g., not any concrete monoids, groups, or vectorspaces). Sometimes they are called *structures* or *signatures*. An **algebra** is an instantiation of an algebra theory with concrete types, functions and predicates, and proofs of axioms. Model theorists would consider algebras to be models of algebra theories. In the following, let $\mathcal{T}$ denote some unspecified type theory.

In a **shallow embedding** (the approach we are following), algebra theories are represented as $\mathcal{T}$-theories, i.e., as lists of $\mathcal{T}$-constants. For example, below we express the algebra theory of unitals (without the neutrality axiom for simplicity) for the type theories $\mathcal{T} = \mathrm{LF}$ of LF and $\mathcal{T} = \mathrm{LF}, \texttt{SFOL}$ of the one that MMT induces for `SFOL`-extensions (see for `SFOL`).

$$
\begin{aligned}
\mathit{Unitals}_{\mathrm{LF}} &= \{U\colon \texttt{type},\ \circ\colon U \to U \to U,\ e\colon U\}\\
\mathit{Unitals}_{\mathrm{LF},\texttt{SFOL}} &= \{U\colon \texttt{tp},\ \circ\colon \texttt{tm}\ U \to \texttt{tm}\ U \to \texttt{tm}\ U,\ e\colon \texttt{tm}\ U\}
\end{aligned}
$$

Algebras are represented as corresponding instantiations. For example, we can express that natural numbers form a unital under addition using the following instantiations of the preceding $\mathcal{T}$-theories:[11]

$$\begin{aligned} \mathit{Nat}_{\mathrm{LF}}\colon \mathit{Unitals}_{\mathrm{LF}} &= \{U := \mathbb{N},\ \circ := \lambda a\, b\colon \mathbb{N}.\ a+b,\ e := 0\} \\ \mathit{Nat}_{\mathrm{LF},\mathtt{SFOL}}\colon \mathit{Unitals}_{\mathrm{LF},\mathtt{SFOL}} &= \{U := \mathtt{tm\ nat},\ \circ := \lambda a\, b\colon \mathtt{tm\ nat}.\ a+b,\ e := 0\} \end{aligned}$$

Here and above we assumed our type theory to feature a way for stating *$\mathcal{T}$-theories* (i.e., named lists of constants) and *$\mathcal{T}$-instantiations*, using an ad-hoc syntax that is hopefully clear without explanation. Many formal systems fulfill this in practice, e.g., Mmt has Mmt theories and views, Coq has modules, Isabelle has locales; yet other systems call their constructs records, structures, classes, or specifications. Importantly, in a shallow embedding the represented algebra theories and algebras often syntactic, declarative constructs of $\mathcal{T}$.

In contrast, a **deep embedding** employs an additional level of indirection. It defines a general $\mathcal{T}$-theory $\mathtt{Sig}$ whose instantiations represent algebra theories, and for every algebra theory $\mathtt{sig}\colon \mathtt{Sig}$ represented that way it defines a $\mathcal{T}$-theory $\mathtt{Alg}^{\mathtt{sig}}$ whose instantiations represent algebras. Let us give an example following the approach of [DeM21]. We assume $\mathcal{T}$ to be some dependent type theory (e.g., LF) enriched with finite types $\mathbb{1}, \mathbb{2}, \ldots$, pattern matching, and the ability to express parametric theories like $\mathtt{Alg}^{\mathtt{sig}}$. For simplicity, we restrict ourselves to representing those algebra theories that are given by single-sorted signatures with function symbols. We in particular exclude multisortednes and predicate symbols (to which our example easily generalizes) and axioms (which are considerably harder to deep-embed). The deep embedding is as follows:

$$\begin{aligned} \mathtt{Sig} &= \{\mathtt{F}\colon \mathtt{type}, |\cdot|\colon \mathtt{F} \to \mathtt{type}\} \\ \mathtt{Alg}^{\mathtt{sig}\colon \mathtt{Sig}} &= \{\mathtt{U}\colon \mathtt{type}, [\![\cdot]\!]\colon \Pi f\colon \mathtt{F}.\ (|f| \to \mathtt{U}) \to \mathtt{U}\} \end{aligned}$$

Given an instantiation of $\mathtt{Sig}$, we can think of the type assigned to $\mathtt{F}$ as the type of function symbols. For example, to represent an algebra theory with exactly two function symbols, we could assign the finite type 2. And for every $f\colon \mathtt{F}$ representing a function symbol, its arity is encoded by the type $|f|$. For example, if we wanted f to represent a tertiary function symbol, we could assign a function to $|\cdot|$ that returns the finite type 3 upon input f. The algebra theory of unitals (again, without the neutrality axiom) can be given as follows:

$$\begin{aligned} \mathit{Unitals}\colon \mathtt{Sig} = \{\mathtt{F} := \mathbb{2}, |\cdot| := \lambda f\colon \mathbb{2}.\ f\ \mathtt{match}\ & \quad\} \\ \mathtt{case}\ \cdot_1 \Rightarrow \mathbb{2}& \\ \mathtt{case}\ \cdot_2 \Rightarrow \mathbb{0}& \end{aligned}$$

To explain how $\mathtt{Alg}^{\mathtt{sig}}$ works, we state that natural numbers form a unital under addition using the deep embedding above:

$$\begin{aligned} \mathit{Nat}\colon \mathtt{Alg}^{\mathit{Unitals}} = \{&\mathtt{U} := \mathbb{N}, \\ &[\![\cdot]\!] := \lambda f\colon \mathbb{2}.\ f\ \mathtt{match} \qquad\qquad \} \\ &\qquad \mathtt{case}\ \cdot_1 \Rightarrow \lambda i\colon \mathbb{2} \to \mathbb{N}.\ (i\ \cdot_1)\ +\ (i\ \cdot_2) \\ &\qquad \mathtt{case}\ \cdot_2 \Rightarrow \lambda i\colon \mathbb{0} \to \mathbb{N}.\ 0 \end{aligned}$$

[11] For the first line we assume an LF type $\mathbb{N}$ of natural numbers, an addition function $+\colon \mathbb{N} \to \mathbb{N} \to \mathbb{N}$, and zero $0\colon \mathbb{N}$. And for the second line we assume an SFOL type $\mathtt{nat}\colon \mathtt{tp}$, an addition function $+\colon \mathtt{tm\ nat} \to \mathtt{tm\ nat} \to \mathtt{tm\ nat}$, and zero $0\colon \mathtt{tm\ nat}$.

	symbols of algebra theory given…	meaning emerges…	accessible to	
			module system? (instantiations, inheritance, metadata)	object logic? (theorems, quantification, transformations)
shallow embedding	declaratively	syntactically	x	
deep embedding	expressively	semantically		x

	shallow embedding	deep embedding
symbols of algebra theory given	declaratively	expressively (as terms)
meaning emerges	syntactically	semantically
accessible to module system?	x	
accessible to object logic?		x
extensibility	easy	difficult
univ. constructions expressible?	no	somewhat

Figure 19 Comparison of shallow and deep embeddings

By dots ($\cdot_1, \cdot_2, \cdot_3$, etc.) we denote inhabitants of finite types. The type assigned to $\mathtt{U}$ represents the universe of the intended algebra, and the function assigned to $[\![\cdot]\!]$ encodes the interpretations of function symbols. Concretely, for every function symbol $f\colon \mathtt{F}$ the term $[\![f]\!]\colon (|f| \to \mathtt{U}) \to \mathtt{U}$ encodes the interpretation of f. Here, the argument $|f| \to \mathtt{U}$ encodes $|f|$-many arguments for interpretation. For example, for instantiations of *Unitals* (e.g., for *Nat* shown above), we have the expected types $[\![\cdot_1]\!]\colon (\mathbb{2} \to \mathtt{U}) \to \mathtt{U}$ and $[\![\cdot_2]\!]\colon (\mathbb{0} \to \mathtt{U}) \to \mathtt{U}$. These function types are in bijective correspondence to types $U \to U \to U$ and U, thus precisely represent function types for the binary and the nullary operation (i.e., the neutral element) of unital algebras.

Importantly, the collection of names of function symbols and their arity are no longer syntactic, declarative constructs (as was the case with the shallow embedding), but emerge semantically on the object level.

Shallow vs. Deep Embeddings Both approaches have their merits and both are used in practice for different purposes. We summarize the main differences in Figure 19.

With a shallow embedding, the declarative and syntactic nature of how algebra theories are stated makes them accessible to the module system. This is generally a good thing: Consider how type theory advocates building on a type sytem to get in return its services such as type safety, normalization, or termination. In the same spirit, module system developers [RK13] (as is the author) advocate building on a module system to get in return its services such as organization into modules and libraries (inheritance, interrelations like morphisms) and being able to annotate extra-logical metadata (humanly-authored comments, source references, typechecking information) In our case, the primary advantage is that a shallow embedding can make use of inheritance between algebra theories, thus state, e.g., the algebra theory of groups as an extension of monoids. This allows developing the whole algebraic hierarchy using structuring features of the module system. This makes shallow embeddings very attractive for formalizing the algebraic hierarchy (in contrast to universal algebra as a unifying meta theory), as is done in major developments such as for Coq in the *Mathematical Components* [mathcomp21; Gar+09] and previous developments [SW11; Geu+02], for Agda in the *Agda Standard Library* [agd21], for Lean in the *Lean Matehmatics Library* [Com20] and for Nuprl in [Jac95, ch. 6].

$$Hom(Unitals_{\mathtt{LF},\mathtt{SFOL}}) = \left\{ \begin{array}{l} \mathtt{U}^d\colon \mathtt{tp},\ \circ^d\colon \mathtt{tm}\ \mathtt{U}^d \to \mathtt{tm}\ \mathtt{U}^d \to \mathtt{tm}\ \mathtt{U}^d,\ e^d\colon \mathtt{tm}\ \mathtt{U}^d \\ \mathtt{U}^c\colon \mathtt{tp},\ \circ^c\colon \mathtt{tm}\ \mathtt{U}^c \to \mathtt{tm}\ \mathtt{U}^c \to \mathtt{tm}\ \mathtt{U}^c,\ e^c\colon \mathtt{tm}\ \mathtt{U}^c \\ h\ \colon \mathtt{tm}\ \mathtt{U}^d \to \mathtt{tm}\ \mathtt{U}^c \\ \circ^h\colon \Pi\, x\ y\colon \mathtt{tm}\ \mathtt{U}^d.\ \Vdash h\ (x \circ^d y) \doteq (h\ x) \circ^c (h\ y) \end{array} \right\}$$

Figure 20 Theory of homomorphisms for the shallow embedding $Unitals_{\mathtt{LF},\mathtt{SFOL}}$

Continuing our analogy, consider how basing things on a type system usually incurs a loss of flexibility, primarily in reflection capabilities. For example, sometimes tricks that are oneliners in untyped programming languages become very awkward when expressed in typed programming languages. This is similar with module systems. Baring *rare and advanced* reflection capabilities, with a shallow embedding we are unable to quantify over algebra theories or algebras, unable to state meta theorems or transformations on algebra theories. For example, to formalize homomorphisms we would need to specify a theory $Hom(T)$ *for every* theory T representing an algebra theory. Figure 20 shows how $Hom(Unitals_{\mathtt{LF},\mathtt{SFOL}})$ could look like. (Further explanations on this theory can be found in Section 5.3.) For lack of alternatives, manually typing those theories is indeed regularly done in practice, e.g., in the libraries cited above.

In contrast, **with deep embeddings** we can easily quantify over and transform algebra theories and algebras to elegantly formalize universal constructions. For example, based on our deep embedding above, the theory of homomorphisms can be given *once and for all* as the following parametric theory:[12]

$$Hom^{\mathtt{sig}\colon \mathtt{Sig},\mathtt{d}\colon \mathtt{Alg}^{\mathtt{sig}},\mathtt{c}\colon \mathtt{Alg}^{\mathtt{sig}}} = \left\{ \begin{array}{l} \doteq\ \ \colon \mathtt{U}^c \to \mathtt{U}^c \to \mathtt{type} \\ h\ \ \ \colon \mathtt{U}^d \to \mathtt{U}^c \\ \mathtt{hom}\colon \Pi\, f\colon \mathtt{F}.\ \Pi\, a\colon |f| \to \mathtt{U}^d.\ [\![f]\!]^d\ a \doteq [\![f]\!]^c\ (\lambda x.\ h\ (a\,x)) \end{array} \right\}$$

Therefore, deep embeddings are attractive to formalize universal algebra itself (unlike the algebraic hierarchy) and its meta theorems, e.g., in Coq [Cap99; SW11] or Agda [DeM21; GGP18].

We follow a combination of both approaches: we use shallow embeddings to formalize algebra theories and codify universal constructions and meta theorems in the form of diagram operators in the programming language underlying MMT (Scala). This combines all advantages of shallow embeddings with the flexbility to express arbitrarily complicated transformations and meta theorems. The price to pay is that type safety and correctness properties for the latter cannot be mechanically verified anymore. (But they can be proven on paper as we do in this thesis.) Still, as we remark in in our implementation we can opt to typecheck syntax produced by operators. This does not prevent operators from producing senseless output, but at least it guarantees its well-typedness.

[12] Alternatively, we can also give a deep embedding of the theory of homomorphisms itself. Assume a deep embedding of multi-sorted signatures with axioms given by theories $\mathtt{Sig}_+$ (whose instantiations are exactly those signatures) and $\mathtt{Alg}^{\mathtt{sig}}_+$ (whose instantiations are corresponding models). Then we can give a *defined* constant $\mathtt{hom}\colon \Pi\,\mathtt{sig}\colon \mathtt{Sig}.\ \mathtt{Alg}^{\mathtt{sig}} \to \mathtt{Alg}^{\mathtt{sig}} \to \mathtt{Sig}_+ = \ldots$ that constructs the deep embedding variant of $Hom^{\mathtt{sig},d,c}$. In fact, we can generalize to $\mathtt{hom}\colon \Pi\,\mathtt{sig}\colon \mathtt{Sig}_+.\ \mathtt{Alg}^{\mathtt{sig}}_+ \to \mathtt{Alg}^{\mathtt{sig}}_+ \to \mathtt{Sig}_+ = \ldots$ such that the expressivity of the output signatures matches the inputs' ones. This is more in line with some diagram operators that we give that also accept input signatures that are as expressive as their output signatures.

5.2.2 Representing Algebra Theories as Mmt Theories

In Example 97 we foreshadowed how to represent the algebra theory of unitals as an `SFOL`-extension (for theory `SFOL` from). We now distill a handful classes of well-patterned `SFOL`-extensions, which we will use to represent algebra theories in our setting.

▶ **Definition 98** (SFOL, PFOL, and DFOL Theories). *Consider an* `SFOL`*-extension whose declarations follow the patterns*

- *type symbols (aka sort symbols)*

$$T \quad : \Pi\, a_1 \,\dots a_m : \mathtt{tp}.\ \Pi\, x_1 : \mathtt{tm}\ t_1.\ \dots \Pi\, x_n : \mathtt{tm}\ t_n.\quad \mathtt{tp} \qquad [= \lambda\, a_1. \dots \lambda\, a_m.\ \lambda\, x_1. \dots \lambda\, x_n.\ a]$$

- *function symbols*

$$f \quad : \Pi\, a_1 \,\dots a_m : \mathtt{tp}.\ \Pi\, x_1 : \mathtt{tm}\ t_1.\ \dots \Pi\, x_n : \mathtt{tm}\ t_n.\quad \mathtt{tm}\ t \qquad [= \lambda\, a_1. \dots \lambda\, a_m.\ \lambda\, x_1. \dots \lambda\, x_n.\ t]$$

- *predicate symbols*

$$p \quad : \Pi\, a_1 \,\dots a_m : \mathtt{tp}.\ \Pi\, x_1 : \mathtt{tm}\ t_1.\ \dots \Pi\, x_n : \mathtt{tm}\ t_n.\quad \mathtt{prop} \qquad [= \lambda\, a_1. \dots \lambda\, a_m.\ \lambda\, x_1. \dots \lambda\, x_n.\ F]$$

- *axiom symbols*

$$ax \quad : \Pi\, a_1 \,\dots a_m : \mathtt{tp}.\ \Pi\, x_1 : \mathtt{tm}\ t_1.\ \dots \Pi\, x_n : \mathtt{tm}\ t_n.\quad \Vdash F \qquad [= \lambda\, a_1. \dots \lambda\, a_m.\ \lambda\, x_1. \dots \lambda\, x_n.\ pf]$$

where

$$\begin{array}{lll l}
a & ::= & T\ a_1\ \dots\ a_n\ t_1\ \dots\ t_n & \textit{types (sorts)} \\
t & ::= & f\ a_1\ \dots\ a_n\ t_1\ \dots\ t_n \mid x & \textit{terms} \\
F, G & ::= & F \wedge G \mid F \vee G \mid \neg F \mid F \Rightarrow G & \textit{propositions (formulae)} \\
 & ::= & \forall\, x : \mathtt{tm}\ a.\ F \mid \exists\, x : \mathtt{tm}\ a.\ F \mid t \doteq_T t' & \\
 & ::= & p\ a_1\ \dots\ a_n\ t_1\ \dots\ t_n & \\
pf & ::= & ax\ a_1\ \dots\ a_n\ t_1\ \dots\ t_n \mid & \textit{proofs} \\
 & ::= & \textit{proof rules (omitted)} &
\end{array}$$

We call such an `SFOL`*-extension*

- *an* ***SFOL-theory*** *if $m = 0$ in all declarations and $n = 0$ for type symbol declarations*
- *a* ***PFOL-theory*** *if $n = 0$ for sort symbol declarations*
- *a* ***DFOL-theory*** *if $m = 0$ everywhere*
- *a* ***PDFOL-theory*** *if m and n are unrestricted*

In particular, this implies the hierarchy $SFOL \subsetneq \{PFOL, DFOL\} \subsetneq PDFOL$, i.e., all SFOL-theories are PFOL- and DFOL-theories, and all PFOL- and DFOL-theories are also PDFOL-theories.

Arguably, all notions from Definition 98 are folklore. P/D/PDFOL-declarations readily emerge as subsets of well-formed declarations that LF induces for `SFOL`-extensions. Moreover, these variants naturally appear in practice, see Examples 101 and 102 below. We note that DFOL, as the combination of FOL with *just* dependent types, has first been described in [Mak95] and first published in [Raba] (the latter matching our presentation). For SFOL we can heavily simplify our patterns to match standard accounts given in literature:

▶ Remark 99 (SFOL-Theories). For SFOL-theories, the patterns demanded in Definition 98 simplify to the ones below. To see this, note that type symbols can only be of the form $T : \mathtt{tp}$, and thus type expressions can only ever be atomic (i.e., refer to such T).

- type symbols (aka sort symbols)

$$T \qquad : \texttt{tp} \quad [= T]$$

- function symbols

$$f \qquad : \texttt{tm}\ T_1 \to \ldots \to \texttt{tm}\ T_n \to \texttt{tm}\ T \quad [= \lambda\, x_1.\ \ldots\ \lambda\, x_n.\ e]$$

- predicate symbols

$$p \qquad : \texttt{tm}\ T_1 \to \ldots \to \texttt{tm}\ T_n \to \texttt{prop} \quad [= \lambda\, x_1.\ \ldots\ \lambda\, x_n.\ F]$$

- axiom symbols

$$ax \colon \Pi\, x_1 \colon \texttt{tm}\ T_1.\ \ldots\ \Pi\, x_n \colon \texttt{tm}\ T_n.\ \Vdash F \quad [= \lambda\, x_1.\ \ldots\ \lambda\, x_n.\ pf]$$

These symbol patterns induce exactly the notion commonly referred to as SFOL-theories in the literature. This is clear for everything except the term parameters in axiom symbols. For those, observe that the universal quantification expressed by them can equivalently be internalized using SFOL's forall, yielding axiom symbols of the form $\widetilde{ax} \colon \Vdash \forall\, x_1 \colon \texttt{tm}\ T_1.\ \ldots \forall\, x_n \colon \texttt{tm}\ T_n.\ F$.

Concretely, the notion of equivalence we are referring to can be made formal by giving two LF functions translating between the LF types:

$$(\Pi\, x_1 \colon \texttt{tm}\ T_1.\ \ldots\ \Pi\, x_n \colon \texttt{tm}\ T_n.\ \Vdash F) \rightleftharpoons (\Vdash \forall\, x_1 \colon \texttt{tm}\ T_1.\ \ldots \forall\, x_n \colon \texttt{tm}\ T_n.\ F)$$

The LF function from left to right uses forall introduction n times to introduce the consequence and LF function application n times to eliminate the premise. And conversely, the LF function from right to left uses LF function introduction n times to introduce the consequence and forall elimination n times to eliminate the premise.

While equivalent in expressivity, in practice it is much more convenient to formalize quantification in axiom symbols using LF Πs. Namely, this makes the introduction (i.e., proofs) and elimination (i.e., axiom usage) forms much easier to type since they merely use LF function abstraction and application, i.e., primitives of MMT's grammar – in contrast to cascades of forall introduction and eliminations. Nonetheless, for ease of presentation, we will assume axiom symbols in SFOL-theories to always be of the form $ax \colon \Vdash F$.

▶ **Example 100** (Preoder). We give the theory of preorders as an SFOL-theory:

theory `Preorder` $= \{$

  include `SFOL`
  $U \quad : \texttt{tp}$
  $\leq \quad : \texttt{tm}\ U \to \texttt{tm}\ U \to \texttt{prop}$
  $\texttt{refl} \ : \Pi\, x \colon \texttt{tm}\ U.\ \Vdash x \leq x$
  $\texttt{trans} \colon \Vdash \forall\, x\ y\ z \colon \texttt{tm}\ U.\ \Vdash x \leq y \Rightarrow y \leq z \Rightarrow x \leq z$

$\}$

▶ **Example 101** (Lists in PFOL). We give the theory of lists as a PFOL-theory:

theory `Lists` $= \{$

  include `SFOL`
  $\texttt{list} \colon \texttt{tp} \to \texttt{tp}$
  $\texttt{nil} \ : \Pi\, U \colon \texttt{tp}.\ \texttt{tm}\ (\texttt{list}\ U)$
  $\texttt{cons} \colon \Pi\, U \colon \texttt{tp}.\ \texttt{tm}\ U \to \texttt{tm}\ (\texttt{list}\ U) \to \texttt{tm}\ (\texttt{list}\ U)$

$\}$

▶ **Example 102** (Small Categories in DFOL)**.** We give the theory of small categories as the DFOL-theory below. As a reminder, a small category is a category where the size of the collection of objects and all hom-collections are the same. In contrast, in a non-small category, the collection of objects may form a class (e.g., when working in some set theory) and the collection of hom-collections must be a set.

$$
\begin{array}{l}
\textbf{theory}\ \texttt{SmallCat} = \{ \\
\quad \begin{array}{ll}
\texttt{obj} & : \texttt{tp} \\
\texttt{hom} & : \texttt{tm obj} \to \texttt{tm obj} \to \texttt{tp} \\
\texttt{id} & : \Pi\, a\colon \texttt{tm obj}.\ \texttt{tm}\ (\texttt{hom}\ a\ a) \\
\circ & : \Pi\, a\ b\ c\colon \texttt{tm obj}.\ \texttt{tm}\ (\texttt{hom}\ a\ b) \to \texttt{tm}\ (\texttt{hom}\ b\ c) \to \texttt{tm}\ (\texttt{hom}\ a\ c)
\end{array} \\
\quad \texttt{neutL}\colon \Pi\, a\ b\colon \texttt{tm obj}.\ \Pi\, f\colon \texttt{tm hom}\, a\, b.\ \Vdash \texttt{id}\, b \circ f \doteq f \\
\quad \texttt{neutR}\colon \Pi\, a\ b\colon \texttt{tm obj}.\ \Pi\, f\colon \texttt{tm hom}\, a\, b.\ \Vdash f \circ \texttt{id}\, a \doteq f \\
\quad \texttt{assoc}\colon \Pi\, a\ b\ c\ d\colon \texttt{tm obj}.\ \Pi\, f\colon \texttt{tm hom}\, a\, b.\ \Pi\, g\colon \texttt{tm hom}\, b\, c.\ \Pi\, h\colon \texttt{tm hom}\, a\, c. \\
\qquad\qquad \Vdash f \circ (g \circ h) \doteq (f \circ g) \circ h \\
\}
\end{array}
$$

For readability, we suppress the inferrable parameters in the notation for $\circ$.

5.2.3 Related Work

We are interested in large-scale programmatic definitions of universal algebra with shallow embeddings. Such definitions have so far mostly been conducted with deep embeddings, e.g., for Coq [Cap99; SW11] or Agda [DeM21; GGP18]. For shallow embeddings, we are only aware of [CFS20; Sha21], which both focus on programmatically generating many theories derived from the algebraic hierarchy, even including some of the same examples as ours.

Despite having a similar focus as ours, these works primarily expand into breadth and consider single-sorted first-order signatures with function and axiom symbols and present many operators. Complementarily, we focus on depth and state our few constructions – where feasible in the scope of this thesis – for PDFOL-theories and -morphisms, i.e., polymorphic, dependently-typed first-order signature and signature morphisms with type, function, predicate, and axiom symbols (all of which may be defined). This makes the involved syntax translations much more complex. Indeed, universal algebra is typically developed in this setting [BS12]. That is sufficient to represent typical algebra theories, but even vectorspaces require at least two sorts (for scalars and vectors). For the many-sorted case, the basic constructions extend seamlessly, e.g., see [Wec92, ch. 4.1]. We are not aware of existing efforts to generalize the constructions, and our presentation may be the first one to do so.

Lastly, even though these works make use of a rich syntax to build theory graphs (going beyond includes, but not as expressive as MMT's syntax in the implementation), everything is flattened before applied to their programs generating the universal constructions. Thus their output neglects all of the input structure. In contrast, we make structure preservation a main concern of ours.

5.2.4 Overarching Example: Algebraic Hiearchy in LATIN2: TODO

we use this as case study...

▶ **Example 103** (Algebraic Hierarchy in LATIN2)**.** We can extend these two examples to a full-blown algebraic hierarchy containing monoids, groups, rings, fields, vectorspaces, posets, lattices, etc. The LATIN2 project [LATIN2] (the successor of LATIN [Cod+a]) does exactly this and gives a big modular diagram, see Figure 21 for an excerpt. Importantly, the diagram exploits Mmt's structuring features wherever possible, making it a perfect example to apply structure-preserving diagram operators to. Applying the operators that we will describe in the following, we get a diagram larger than possible to typeset. We give excerpts in the appendix

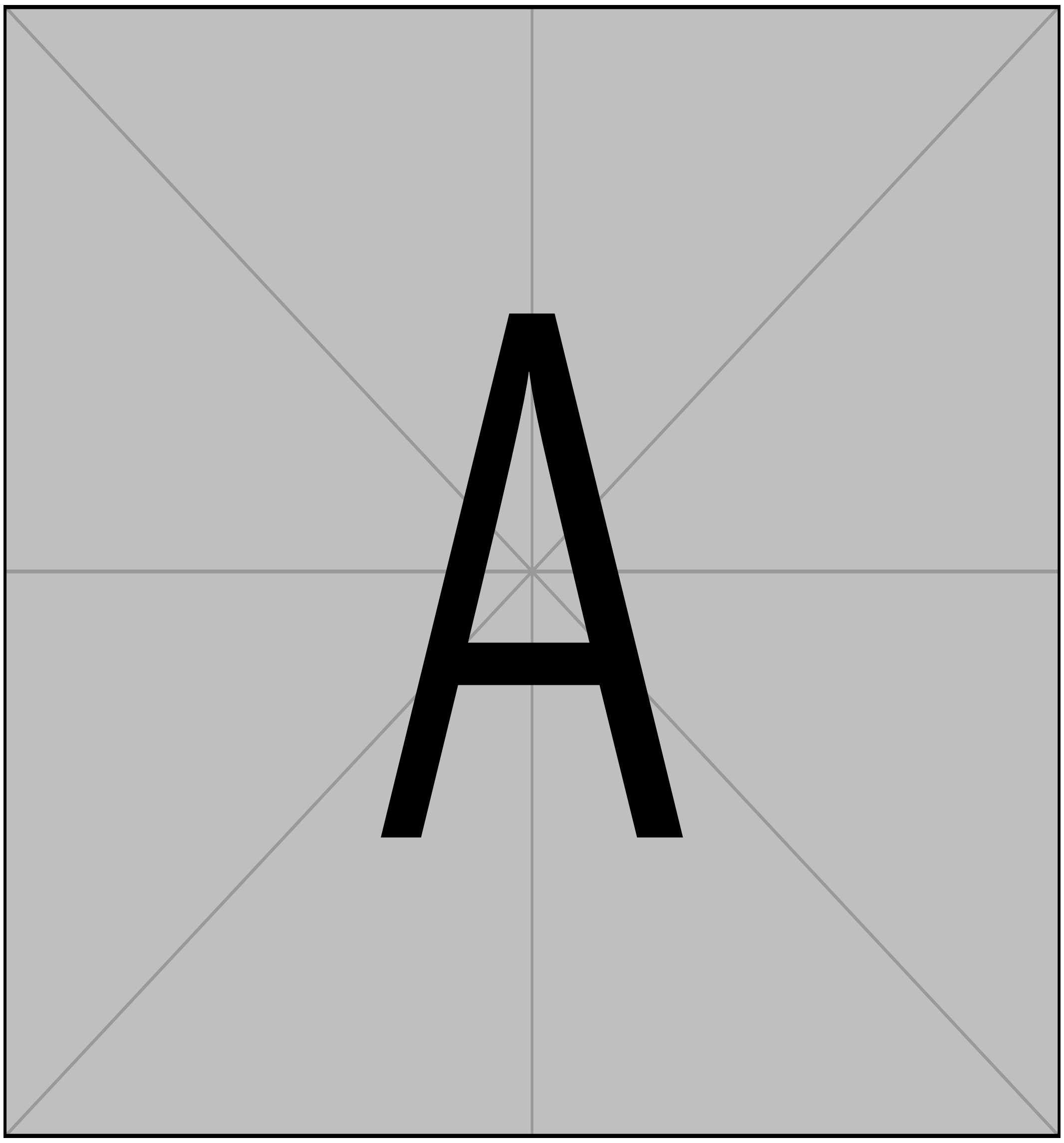

Figure 21 The Algebraic Hierarchy Developed as an Mmt Diagram

5.3 Homomorphisms

$$\begin{tikzcd} T \arrow[r, shift left=2, "\texttt{Hom}^d"] \arrow[r, "\texttt{Hom}^c"] \arrow[r, shift right=2, "h"'] & \texttt{Hom}(T) \end{tikzcd}$$

The linear functor $\texttt{Hom}(-)$ maps every PDFOL-theory T to the theory $\texttt{Hom}(T)$ of T-homomorphisms, whose models are the homomorphisms between two T-models. For example, applied to the theory `Group` it yields the theory `Hom(Group)` representing precisely what mathematicians consider group homomorphisms. The linear connectors d, c into `Hom` are the projections of the very domain and codomain T-models. And the logical relation-inspired translation h proves that T-homomorphisms are homomorphic wrt. (preserve) not only atomic but arbitrarily complex terms (propositions) built out of function (and predicate) symbols from T.[13]

5.3.1 Preliminary Definition: `Hom` on Definitionless `SFOL`

For didactic reasons, we first define `Hom` and the connectors $\texttt{Hom}^d, \texttt{Hom}^c$ for the special case of definitionless SFOL-theories. To do so, we make `Hom` a linear functor from SFOL-theories to PDFOL-theories whose action on declarations is given below. `Hom` maps each type, function, and predicate symbol declaration c to three declarations c^d, c^c, c^h. The constants c^d and c^c serve as qualified copies to build up the domain and codomain structure of the homomorphism (following the idiom outlined in Theorem 38). And the constant c^h accounts for the actual condition imposed on the homomorphism at the very input declaration. For function symbols this amounts to the homomorphism acting homomorphic wrt. corresponding applications, and for predicate symbols this amounts to the homomorphism preserving truth of corresponding applications. Axiom symbols are mapped to their qualified copies only.

- type symbols $T : \texttt{tp}$ are mapped to two copies and a mediating function (the homomorphism on terms of type T^d):

  $$\begin{aligned} &T^d, T^c : \texttt{tp} \\ &\quad T^h : \texttt{tm}\ T^d \to \texttt{tm}\ T^c \end{aligned}$$

- function symbols $f : \texttt{tm}\ T_1 \to \ldots \to \texttt{tm}\ T_n \to \texttt{tm}\ T$ are mapped to two copies and the condition of the homomorphism at types $T_1, \ldots, T_n$ being homomorphic wrt. f:

  $$\begin{aligned} &f^d : \texttt{tm}\ T_1^d \to \ldots \to \texttt{tm}\ T_n^d \to \texttt{tm}\ T^d \\ &f^c : \texttt{tm}\ T_1^c \to \ldots \to \texttt{tm}\ T_n^c \to \texttt{tm}\ T^c \\ &f^h : \Pi\, x_1^d : \texttt{tm}\ T_1^d.\ \ldots\ \Pi\, x_n^d : \texttt{tm}\ T_n^d.\ \Vdash T^h\ (f^d\ x_1^d\ \ldots\ x_n^d) \doteq f^c\ (T_1^h\ x_1^d)\ \ldots\ (T_n^h\ x_n^d) \end{aligned}$$

- predicate symbols $p : \texttt{tm}\ T_1 \to \ldots \to \texttt{tm}\ T_n \to \texttt{prop}$ are mapped to two copies and the condition of the homomorphisms at types $T_1, \ldots, T_n$ preserving truth:

  $$\begin{aligned} &p^d : \texttt{tm}\ T_1^d \to \ldots \to \texttt{tm}\ T_n^d \to \texttt{prop} \\ &p^c : \texttt{tm}\ T_1^c \to \ldots \to \texttt{tm}\ T_n^c \to \texttt{prop} \\ &p^h : \Pi\, x_1^d : \texttt{tm}\ T_1^d.\ \ldots\ \Pi\, x_n^d : \texttt{tm}\ T_n^d.\ \Vdash p^d\ x_1^d\ \ldots\ x_n^d \Rightarrow p^c\ (T_1^h\ x_1^d)\ \ldots\ (T_n^h\ x_n^d) \end{aligned}$$

[13] See Theorem 117.

- axiom symbols $ax\colon \Vdash F$ are mapped to just two copies

$$ax^d : \Vdash F^d$$
$$ax^c : \Vdash F^c$$

The linear connectors $\mathtt{Hom}^d$ and $\mathtt{Hom}^c$ into `Hom` map every constant c to $c := c^d$ and $c := c^c$, respectively.

Our restriction to definitionless SFOL-theories makes it easy to see that `Hom` is well-typed and functorial. Functoriality holds vacuously since `Hom` is partial on all defined constants, including morphism assignments, thus in particular on all morphisms. By Theorem 38, the connectors $\mathtt{Hom}^d, \mathtt{Hom}^c$ are well-typed and natural (the latter again vacuously).

▶ **Example 104** **(`Hom(Unital)`).** For the theory `Unital` from Example 97 we obtain the theory `Hom(Unital)` and incoming morphisms $\mathtt{Hom}^d(\mathtt{Unital})$ and $\mathtt{Hom}^c(\mathtt{Unital})$ shown in Figure 22.[14] We see that the constants $U^d, \circ^d, e^d, \mathtt{neut}^d$ (and analogously the c-superscripted ones) collectively make up a unital domain (codomain) structure. This is witnessed by the morphism $\mathtt{Hom}^d(\mathtt{Unital})$ (and $\mathtt{Hom}^c(\mathtt{Unital})$). Within `Hom(Unital)`, these structures are related via the h-superscripted constants $U^h, \circ^h, e^h$, where U^h represents the homomorphism function translating between the unital structures' universes and the latter two capture that unital homomorphisms preserve the binary operation and the neutral element of the domain structure.

▶ Remark 105 (Axioms using LF's Π vs. `SFOL`'s $\forall$). In the h-superscripted axiom symbols that we generate for function and predicate symbols (representing the respective homomorphism condition), we make use of LF Π-binders to express universal quantification. Consider the polymorphic axiom $\circ^h$ generated by `Hom` in the preceding Example 104:

$$\circ^h : \Pi\, x\; y\colon \mathtt{tm}\; U^d.\ \Vdash U^h\ (x \circ^d y) \doteq (U^h\ x) \circ^c (U^h\ y)$$

These Π-binders are the reason why we needed to declare `Hom` to map SFOL- to PDFOL-theories instead of just being an endofunctor on SFOL-theories. However, we can achieve the latter by internalizing the Π-binders as $\forall$-quantifiers, and we discuss the (dis)advantages in this remark. For example, instead of $\circ^h$ shown above we could have generated the following axiom:

$$\tilde{\circ^h} :\ \Vdash \forall\, x\; y\colon \mathtt{tm}\; U^d.\ U^h\ (x \circ^d y) \doteq (U^h\ x) \circ^c (U^h\ y)$$

For the sake of this remark, imagine we applied this internalization in general and in the above definition of `Hom` we replaced the synthesized axiom symbols f^h and p^h with the following ones:

$$\tilde{f^h} : \Vdash \forall\, x_1^d\colon \mathtt{tm}\; T_1^d, \dots, x_n^d\colon \mathtt{tm}\; T_n^d.\ T^h\ (f^d\ x_1^d\ \dots\ x_n^d) \doteq f^c\ (T_1^h\ x_1^d)\ \dots\ (T_n^h\ x_n^d)$$
$$\tilde{p^h} : \Vdash \forall\, x_1^d\colon \mathtt{tm}\; T_1^d, \dots, x_n^d\colon \mathtt{tm}\; T_n^d.\ p^d\ x_1^d\ \dots\ x_n^d \Rightarrow p^c\ (T_1^h\ x_1^d)\ \dots\ (T_n^h\ x_n^d)$$

Semantically, both approaches using LF Π-binders and SFOL $\forall$-quantifiers are equivalent: the constants f^h and $\tilde{f^h}$ (analogously: p^h and $\tilde{p^h}$) are each definable using the other one, see Remark 99.

On the positive side, via the internalization `Hom` would become an endofunctor on definitionless SFOL-theories, thus yielding an arguably more elegant definition, *at least on*

[14] To serve readability, we α-renamed certain bound variables.

$$
\begin{array}{l}
\textbf{theory}\ \texttt{Hom}(\texttt{Unital}) = \{ \\
\quad \begin{array}{ll}
\multicolumn{2}{l}{\textbf{include}\ \texttt{SFOL}} \\
U^d & : \texttt{tp} \\
U^c & : \texttt{tp} \\
U^h & : \texttt{tm}\ U^d \to \texttt{tm}\ U^c \\
\circ^d & : \texttt{tm}\ U^d \to \texttt{tm}\ U^d \to \texttt{tm}\ U^d \\
\circ^c & : \texttt{tm}\ U^c \to \texttt{tm}\ U^c \to \texttt{tm}\ U^c \\
\circ^h & : \Pi\, x\ y \colon \texttt{tm}\ U^d.\ \Vdash U^h\ (x \circ^d y) \doteq (U^h\ x) \circ^c (U^h\ y) \\
e^d & : \texttt{tm}\ U^d \\
e^c & : \texttt{tm}\ U^c \\
e^h & : \Vdash U^h\ e^d \doteq e^c \\
\multicolumn{2}{l}{\texttt{neut}^d \colon \Vdash \forall\, x^d \colon \texttt{tm}\ U^d.\ e^d \circ^d x^d \doteq x^d} \\
\multicolumn{2}{l}{\texttt{neut}^c \colon \Vdash \forall\, x^c \colon \texttt{tm}\ U^c.\ e^c \circ^c x^c \doteq x^c}
\end{array} \\
\}
\end{array}
$$

$$
\begin{array}{l}
\textbf{mor}\ \texttt{Hom}^d(\texttt{Unital}) \colon \texttt{Unital} \to \texttt{Hom}(\texttt{Unital}) = \{ \\
\quad = = \\
\}
\end{array}
$$

$$
\begin{array}{l}
\textbf{mor}\ \texttt{Hom}^c(\texttt{Unital}) \colon \texttt{Unital} \to \texttt{Hom}(\texttt{Unital}) = \{ \\
\quad = = \\
\}
\end{array}
$$

Figure 22 Theory of unital homomorphisms given by `Hom(Unital)` and corresponding domain and codomain projections

first sight. In particular, this easy change would allow users to apply `Hom` iteratively as in `Hom(Hom(...))` (cf. Example 114).

However, the ∀-quantifiers make things more complicated when trying to extend `Hom`. Even extending to SFOL-theories *with* definitions would become burdensome: For example, for defined function symbols $f := e$, `Hom` maps it to three defined constants $f^d, f^c, \widetilde{f^h}$, where for the definition of f^h `Hom` would now need to synthesize awkward ∀-introduction and ∀-elimination rules corresponding to the ∀-quantifiers in the type of f^h. In contrast, with Π-binders we will merely need LF function abstractions and applications. First, being primitives in the LF syntax, this will make extending `Hom` to a functor accepting all SFOL-theories (including definitions) significantly less burdensome. Second, when generalizing to PDFOL, Π-binders will occur naturally anyway, obviating any inclination to avoid them. Moreover, the output generated by `Hom` will require less abstract syntax, easing human consumption such as reading and using the generated constants.

5.3.2 Building Towards a Generalized Definition

We now aim to generalize Hom to a linear functor accepting all PDFOL-theories. In this section, we collect ideas and an failing ansatz, culminating in a succeeding definition in the next section (Section 5.3.3). The ansatz is failing in the sense that it procudes a functor whose output is semantically equivalent to what is desired, but syntactically cluttered. The general principle remains as before: we copy every input constant $c\colon A\,[= t]$ to two qualified copies $c^d\colon A^d\,[= t^d]$ and $c^c\colon A^c\,[= t^c]$ (where $-^d$ and $-^c$ indicate systematic renaming operations) and moreover output a designated third constant c^h for everything but axioms. As previously, the connectors $\texttt{Hom}^d$ and $\texttt{Hom}^c$ map every constant to $c := c^d$ and $c := c^c$, respectively.

The main hurdle lies in specifying the translation that yields the third constant c^h. Above we saw that specifying this translation was easy on those LF types that occur in SFOL-theories. But specifying the translation already becomes non-obvious for corresponding LF terms, i.e., already in case of SFOL-theories, let alone LF types and terms occuring in PDFOL-theories. For example, in SFOL-theories only atomic types can occur, thus homomorphism structures only need to have a finite number of function symbols translating between domain and codomain structure. In contrast, in DFOL we can have infinitely many complex types, thus we need a function symbol for each one of them. And when we additionally consider PFOL to get PDFOL-theories, carefully handling every such necessary function symbol gets even more complicated.

Our magic trick is to cast the translation as a slightly modified variant of a binary partial logical relation on $\texttt{Hom}^d$ and $\texttt{Hom}^c$ (cf. **??** on logical relations). Concretely, we will specify the third constant as $c^h\colon \overline{h}(A)\ c^d\ c^c\,[= \overline{h}(t)]$ (and only output it when $\overline{h}(A)$ is defined), where $\overline{h}$ is a partial translation function that we have yet to define and that uses the machinery of logical relations Section 2.2 to outsource much of the complexity of defining it.

As a guiding example, recall the theory Lists from Example 101 formalizing generic lists in PDFOL. Among all the constants, we find the type symbol $\texttt{list}\colon \Pi\, A\colon \texttt{tp}.\ \texttt{tp}$ and the function symbol $\texttt{nil}\colon \Pi\, A\colon \texttt{tp}.\ \texttt{tm}\ \texttt{list}\ A$, serving as our example here.[15] Now consider the following fragment of Hom(Lists) that we *expect* a sensible and generalized definition of Hom to yield:

$$
\begin{array}{l}
\textbf{theory}\ \texttt{Hom}(\texttt{Lists}) = \{\\
\quad \texttt{list}^d : \Pi\, A^d : \texttt{tp}.\ \texttt{tp}\\
\quad \texttt{list}^c : \Pi\, A^c : \texttt{tp}.\ \texttt{tp}\\
\quad \texttt{list}^h : \Pi\, A^d : \texttt{tp}.\ \Pi\, A^c : \texttt{tp}.\ \Pi\, A^h : \texttt{tm}\ A^d \to \texttt{tm}\ A^c.\ \texttt{tm}\ \texttt{list}\ A^d \to \texttt{tm}\ \texttt{list}\ A^c\\
\quad \texttt{nil}^d\ : \Pi\, A^d : \texttt{tp}.\ \texttt{tm}\ \texttt{list}^d\ A^d\\
\quad \texttt{nil}^c\ : \Pi\, A^c : \texttt{tp}.\ \texttt{tm}\ \texttt{list}^c\ A^c\\
\quad \texttt{nil}^h\ : \Pi\, A^d : \texttt{tp}.\ \Pi\, A^c : \texttt{tp}.\ \Pi\, A^h : \texttt{tm}\ A^d \to \texttt{tm}\ A^c.\\
\qquad\qquad \Vdash \texttt{list}^h\ A^d\ A^c\ A^h\ (\texttt{nil}^d\ A^d) \doteq (\texttt{nil}^c\ A^c)\\
\quad \ldots\\
\}
\end{array}
$$

Keep in mind that Hom(Lists) is supposed to be a theory whose models are the homomorphisms between Lists-models. And we can think of Lists-models as different implemen-

[15] We opted to explicitly name the first parameter in the type of list (instead of making it anonymous as in $\texttt{tp} \to \texttt{tp}$) to ease visibility of the structure of the translation that we will come up with.

$\Sigma \supseteq$ `SFOL`-expression		mapped to `Hom`(Σ)-expression	
types	$T : \mathtt{tp}$	$\overline{h}(T) : \mathtt{tm}\ T^d \to \mathtt{tm}\ T^c$	homomorphism for type T from dom. to cod.
terms	$t : \mathtt{tm}\ T$	$\overline{h}(t) : \Vdash \overline{h}(T)\ t^d \doteq t^c$	proof of homomorphicity wrt. function symbols
propositions	$F : \mathtt{prop}$	$\overline{h}(F) : \Vdash F^d \Rightarrow F^c$	proof of preservation of monotone formulae
proofs	$pf : \Vdash F$	$\overline{h}(pf)$: undefined	(not mapped at all)

Figure 23 Intuitive overview of the translation to be carried out by $\overline{h}$

tations of lists (cf. linked and array lists in typical programming languages such as C and Java). Thus, intuitively a homomorphism between two such models should translate list terms of one to the other, while preserving certain things, e.g., the empty list of the domain model should be preserved as the empty list of the codomain under this translation. This is precisely what the theory fragment above represents. Moreover, the constants shown there all emerge from the input constants in a systematic way that is similar as before. The only extension as to before is that the *named* Π-bound variable $A\colon \mathtt{tp}$ (which previously could not occur as SFOL input) is made subject to similar transformations as constants, e.g., it gets copied to Π-bound variables $A^d\colon \mathtt{tp}$, $A^c\colon \mathtt{tp}$, and $A^h : \mathtt{tm}\ A^d \to \mathtt{tm}\ A^c$.

Using the guiding example, let us flesh out details of the variant of logical relation that are we going to use. Recall that we intend to map input constants $c\colon A\,[= t]$ to $c^h : \overline{h}(A)\ c^d\ c^c\,[= \overline{h}(t)]$, where $\overline{h}$ resembles a binary partial logical relation on `Hom`^{d} and `Hom`^{c}. Consider the translation from the LF type of `list` to the LF type of `list`^{h}. To realize this transformation as a logical relation, we relate PDFOL types $a^d\colon \mathtt{tp}$ and $a^c\colon \mathtt{tp}$ in the binary relation at LF type $\mathtt{tp}\colon \mathtt{type}$ iff there is a witness of the LF function type $\mathtt{tm}\ a^d \to \mathtt{tm}\ a^c$. This makes the first parameter of `list` expand into the parameters A^d, A^c, A^h and the return type (being `tp`) of `list` into $\mathtt{tm}\ \mathtt{list}\, A^d \to \mathtt{tm}\ \mathtt{list}\, A^c$. In other words, as the relation at PDFOL types we precisely encode LF functions that represent the homomorphisms in our setting. Now consider the translation from the LF type of `nil` to the LF type of `nil`^{h}. To phrase this transformation as a logical relation, we relate terms $t^d : \mathtt{tm}\ a^d$ and $t^c : \mathtt{tm}\ a^c$ in the binary relation at LF type $\mathtt{tm}\ a$ iff there is a witness of $\Vdash a^h\ t^d \doteq t^c$ (where a^h is the recursive witness of a^d and a^c being related, i.e., a function $\mathtt{tm}\ a^d \to \mathtt{tm}\ a^c$ – the homomorphism at type a).

Displeasing Attempt: `Hom` using a Logical Relation The preceding remarks could lead us to defining `Hom` as follows: we map every constant $c\colon A\,[= t]$ to $c^d : A^d\,[= t^d]$, $c^c : A^c\,[= t^c]$, and $c^h : \overline{h}(A)\ c^d\ c^c\,[= \overline{h}(t)]$ (the latter only when $\overline{h}(A)$ is defined), where $\overline{h}$ is the partial logical relation on `Hom`^{d} and `Hom`^{c} with base cases below.

$$\begin{aligned}
\overline{h}(\mathtt{tp}) &= \lambda U^d\ U^c\colon \mathtt{tp}.\ \mathtt{tm}\ U^d \to \mathtt{tm}\ U^c\\
\overline{h}(\mathtt{tm}) &= \lambda T^d\ T^c\colon \mathtt{tp}.\ \lambda T^h\colon \mathtt{tm}\ T^d \to \mathtt{tm}\ T^c.\ \lambda t^d\colon \mathtt{tm}\ T^d.\ \lambda t^c\colon \mathtt{tm}\ T^c.\ \Vdash T^h\ t^d \doteq t^c\\
\overline{h}(\mathtt{prop}) &= \lambda F^d\ F^c.\ \Vdash F^d \Rightarrow F^c\\
\overline{h}(\Vdash) &= \text{undefined}
\end{aligned}$$

See Figure 23 for a descriptive overview. This attempt works insofar that we can extend $\overline{h}$ to a logical relation by giving cases to all `SFOL`-symbols – excluding non-monotone connectives like $\neg$ and $\forall$ (but this is to be expected anyway, we later explain this in).

However, this attempt also translates certain things in a way that is semantically equivalent to what we desire but displeasing in terms of usability and users' expectations. Concretely, the culprit is the handling of terms $t\colon \mathtt{tm}\ T$. As an easy example, consider the function symbol $\circ\colon \mathtt{tm}\ U \to \mathtt{tm}\ U \to \mathtt{tm}\ U$ from the theory `Unital` (see Example 104). The supposed definition above would translate it to constants $\circ^d$, $\circ^c$ (with the obvious LF type), and

$$\begin{aligned}\circ^h : \Pi\, x^d\colon \texttt{tm}\; U^d.\ \Pi\, x^c\colon \texttt{tm}\; U^c.\ \Pi\, x^h\colon\ & \Vdash U^h\ x^d \doteq x^c.\\ \Pi\, y^d\colon \texttt{tm}\; U^d.\ \Pi\, y^c\colon \texttt{tm}\; U^c.\ \Pi\, y^h\colon\ & \Vdash U^h\ y^d \doteq y^c.\\ \Vdash U^h\ (x^d \circ^d y^d) \doteq x^c \circ^c y^c &\end{aligned}$$

Our ansatz yields the expected return type (last line), but unfortunately it also yields superfluous hypotheses $x^c, x^h,\ y^c, y^h$. By comparison, we desire a translation that outputs the following constant instead (cf. Figure 22 from Example 104):

$$\begin{aligned}\circ^h : \Pi\, x^d\colon & \texttt{tm}\; U^d.\\ & \Pi\, y^d\colon \texttt{tm}\; U^d.\\ & \Vdash U^h\ (x \circ^d y) \doteq (U^h\ x) \circ^c (U^h\ y)\end{aligned}$$

In other words, our translation should have *inlined* the superfluous hypotheses as follows:

i) instead of binding x^c (analogously: y^c), all references to x^c should be replaced by $U^h\ x^d$

ii) instead of binding $x^h\colon\ \Vdash U^h\ x^d \doteq x^c$ (analogously: y^h) – whose type, using *i*), has just become the trivial type $\Vdash U^h\ x^d \doteq U^h\ x^d$ anyway – all references to x^h should be replaced by `refl` $(U^h\ x^d)$

Note that the two constants $\circ^h$ above (the one that our failing definition of $\overline{h}$ yields and the intended one) are semantically equivalent: either constant is definable using the other one. Thus, our preliminary definition is only failing in the sense of cosmetic issues. (Also note that *ii*) does not manifest in our example of $\circ^h$ as there are neither references to x^h nor to y^h.)

5.3.3 Final Definition: `Hom` on PDFOL

For our definition of `Hom` on all PDFOL-theories, we take the ansatz from the previous section (Section 5.3.2) as a starting point and implement precisely the two changes outlined in the end of that section. In other words, to define $\overline{h}$ as desired we take the complex cases for $\overline{h}$ that would have been induced if it was a logical relation on $\texttt{Hom}^d, \texttt{Hom}^c$ (with base cases as in the failing definition above) and alter them as little as possible to get the inlining right:

▶ **Definition 106.** *The translation $\overline{h}$ maps contexts $\vdash^{\texttt{SFOL}}_{\Sigma} \Sigma$ and LF terms $\Sigma \vdash^{\texttt{SFOL}}_{\Sigma} t$ to $\overline{h}^{\Sigma}(t)$ as specified in Figure 24. In particular, it has base cases*

$$\begin{aligned}\overline{h}(\texttt{tp}) &= \lambda U^d\, U^c\colon \texttt{tp}.\ \texttt{tm}\; U^d \to \texttt{tm}\; U^c\\ \overline{h}(\texttt{tm}) &= \lambda T^d\, T^c\colon \texttt{tp}.\ \lambda T^h\colon \texttt{tm}\; T^d \to \texttt{tm}\; T^c.\ \lambda t^d\colon \texttt{tm}\; T^d.\ \lambda t^c\colon \texttt{tm}\; T^c.\ \Vdash T^h\ t^d \doteq t^c\\ \overline{h}(\texttt{prop}) &= \lambda F^d\, F^c.\ \Vdash F^d \Rightarrow F^c\\ \overline{h}(\Vdash) &= \textit{undefined}\end{aligned}$$

For brevity, we hide the context in the notation, and in the sequel we also write h for $\overline{h}$.

▶ **Definition 107** (Homomorphism Operators)**.** *The linear functor* `Hom` *from PDFOL to all possible* `SFOL`*-extensions and the linear connectors d, c into* `Hom` *are given by*

$$\begin{aligned}\texttt{Hom}^{\Sigma}(c\colon A\,[=t]) &= c^d\colon A^d\,[=t^d],\ c^c\colon A^c\,[=t^c],\ c^h\colon h^{\Sigma}(A)\ c^d\ c^c\,[=h^{\Sigma}(t)]\\ \texttt{Hom}^{d,\Sigma}(c\colon A) &= c^d\\ \texttt{Hom}^{c,\Sigma}(c\colon A) &= c^c\end{aligned}$$

where h^Σ is the translation from Definition 106 extended by $h^\Sigma(c) = c^h$ for every constant $c \in \Sigma$.

Let us closely look at Figure 24 and compare the definitional cases there with the ones that would taken effect if h was a binary partial logical relation on $\texttt{Hom}^d$ and $\texttt{Hom}^c$. The most important changes are in the cases of h for $\Pi x : A.\ B$ and $\lambda x : A.\ t$. Those have been modified to suppress binding the two superfluous hypotheses x^c and x^h for SFOL terms, as was discussed in the end of the last section (Section 5.3.2). (Variables that do not represent SFOL terms are translated as is usual for binary logical relations, e.g., for SFOL types $T\colon \texttt{tp}$, we still bind $T^d, T^c\colon \texttt{tp}$ and $T^h\colon \texttt{tm}\ T^d \to \texttt{tm}\ T^c$.) The inlining of the now suppressed hypotheses is accounted for at two places. First, for x^c it is done in the subcase of h for $\Pi x\colon \texttt{tm}\ T.\ B$. Here, logical relations would use $\mathfrak{F}\ x^c$, but instead we use $\mathfrak{F}'\ c(x)$, yielding $\mathfrak{F}'(\overline{h}(T)\ x^d)$ by definition of c. The subcase guarantees us that $T\colon \texttt{tp}$ is an SFOL type, thus $\overline{h}(T)\colon \texttt{tm}\ T^d \to \texttt{tm}\ T^c$ is precisely the homomorphism at type T, translating x^d to its codomain counterpart. Second, for x^h the inlining happens in the case of h for variables, where in the subcase for $x\colon \texttt{tm}\ T$ references to x^h are avoided by directly emitting a proof by $\texttt{refl}$. These were the most important changes; all other changes (e.g., in the case for contexts and function application) are a logical consequence.

▶ **Example 108** (Inlining of Hypotheses)**.** To get a feel for how the complex translation h and particularly inlining works in practice, let us consider the following example of a defined constant $f\colon \texttt{tm}\ U \to \texttt{tm}\ U\ = \lambda x.\ x$ which $\texttt{Hom}$ maps to three constants f^d, f^c, f^h:

$$
\begin{aligned}
&f\colon \Pi x\colon \texttt{tm}\ U.\ \texttt{tm}\ U\ = \lambda x\colon \texttt{tm}\ U.\ x\\
\overset{\texttt{Hom}}{\mapsto} &\begin{cases}
f^d\colon \Pi x\colon \texttt{tm}\ U^d.\ \texttt{tm}\ U^d\ = \lambda x\colon \texttt{tm}\ U^d.\ x\\
f^c\colon \Pi x\colon \texttt{tm}\ U^c.\ \texttt{tm}\ U^c\ = \lambda x\colon \texttt{tm}\ U^c.\ x\\
f^h\colon h(\Pi x\colon \texttt{tm}\ U.\ \texttt{tm}\ U)\ f^d\ f^c\ = h(\lambda x\colon \texttt{tm}\ U.\ x)
\end{cases}
\end{aligned}
$$

(For clarity, we are explicit about Π in our notation.) Let us slowly unfold the definition of h to see that the type and definiens of f^h in fact turn out to be the desired LF terms. For the type we have,

$$
\begin{aligned}
&h(\Pi x\colon \texttt{tm}\ U.\ \texttt{tm}\ U)\ f^d\ f^c\\
=&\underbrace{\left(\lambda \mathfrak{F}\mathfrak{F}'.\ \Pi x^d\colon \texttt{tm}\ U^d.\ \overbrace{\overline{h}(\texttt{tm}\ U)}^{\text{relation at } \texttt{tm}\ U}\ (\mathfrak{F}\ x^d)\ \underline{(\mathfrak{F}'\ c(x))}\right)}_{\text{relation at type of } f}\ f^d\ f^c\\
=&\big(\lambda \mathfrak{F}\mathfrak{F}'.\ \Pi x^d\colon \texttt{tm}\ U^d.\ \overline{h}(\texttt{tm})\ U^d\ U^c\ U^h\ (\mathfrak{F}\ x^d)\ (\mathfrak{F}'\ (U^h\ x^d))\big)\ f^d\ f^c\\
=&\big(\lambda \mathfrak{F}\mathfrak{F}'.\ \Pi x^d\colon \texttt{tm}\ U^d.\ \Vdash U^h\ (\mathfrak{F}\ x^d) \doteq \mathfrak{F}'\ (U^h\ x^d)\big)\ f^d\ f^c\\
=&\Pi x^d\colon \texttt{tm}\ U^d.\ \Vdash U^h\ (f^d\ x^d) \doteq f^c\ (U^h\ x^d)\\
=&\Pi x^d\colon \texttt{tm}\ U^d.\ \Vdash U^h\ x^d \doteq U^h\ x^d
\end{aligned}
$$

and for the definiens we have,

$$
\begin{aligned}
&h(\lambda x\colon \texttt{tm}\ U.\ x)\\
=&\lambda x^d\colon \texttt{tm}\ U^d.\ \underline{\texttt{refl}\ U^c\ (U^h\ x^d)}
\end{aligned}
$$

The two places at which the inlining can be observed are underlined.

$$\begin{array}{ll}
\overline{h}(\texttt{tp}) & = \lambda U^d\, U^c \colon \texttt{tp}.\ \texttt{tm}\ U^d \to \texttt{tm}\ U^c \\
\overline{h}(\texttt{tm}) & = \lambda T^d\, T^c \colon \texttt{tp}.\ \lambda T^h \colon \texttt{tm}\ T^d \to \texttt{tm}\ T^c.\ \lambda t^d \colon \texttt{tm}\ T^d.\ \lambda t^c \colon \texttt{tm}\ T^c.\ \Vdash T^h\ t^d \doteq t^c \\
\overline{h}(\texttt{prop}) & = \lambda F^d\, F^c.\ \Vdash F^d \Rightarrow F^c \\
\overline{h}(\Vdash) & = \text{undefined}
\end{array}$$

$$\begin{array}{ll}
\overline{h}(\texttt{type}) & = \lambda \mathfrak{F} \mathfrak{F}'.\ \mathfrak{F} \to \mathfrak{F}' \to \texttt{type} \\
\overline{h}(\Pi\, x \colon A.\ B) = & \begin{cases} \lambda \mathfrak{F}\mathfrak{F}'.\ \Pi\, x^d \colon d(A).\ \overline{h}(B)\ (\mathfrak{F}\ x^d)\ (\mathfrak{F}'\ c(x)) & \text{if } A = \texttt{tm}\ T \\ \lambda \mathfrak{F}\mathfrak{F}'.\ \Pi\, x^d \colon d(A).\ \Pi\, x^c \colon c(A).\ \Pi\, x^h \colon \overline{h}(A)\ x^d\ x^c.\ \overline{h}(B)\ (\mathfrak{F}\ x^d)\ (\mathfrak{F}'\ x^c) & \text{otherwise} \end{cases} \\
\overline{h}(\lambda x \colon A.\ B) & = \begin{cases} \lambda x^d \colon d(A).\ \overline{h}(B) & \text{if } A = \texttt{tm}\ T \\ \lambda x^d \colon d(A).\ \lambda x^c \colon c(A).\ \lambda x^h \colon \overline{h}(A)\ x^d\ x^c.\ \overline{h}(B) & \text{otherwise} \end{cases} \\
\overline{h}(x) & = \begin{cases} \texttt{refl}\ c(T)\ (\overline{h}(T)\ x^d) & \text{if } x \colon A \text{ and } A = \texttt{tm}\ T \\ x^h & \text{otherwise} \end{cases} \\
\overline{h}(f\ t) & = \begin{cases} \overline{h}(f)\ d(t) & \text{if } t \colon A \text{ and } A = \texttt{tm}\ T \\ \overline{h}(f)\ d(t)\ c(t)\ \overline{h}(t) & \text{otherwise} \end{cases} \\
\overline{h}(\cdot) & = \cdot \\
\overline{h}(\Gamma, x \colon A) & = \begin{cases} x^d \colon d(A) & \text{if } A = \texttt{tm}\ T \\ x^d \colon d(A), x^c \colon c(A), x^h \colon \overline{h}(A)\ x^d\ x^c & \text{otherwise} \end{cases}
\end{array}$$

where d and c are defined as

$$d(t) = \begin{cases} x^d & \text{if } t = x \\ \texttt{Hom}^d(c) & \text{if } t = c \\ \text{compositionally recurse into } t & \text{otherwise} \end{cases}$$

$$c(t) = \begin{cases} \overline{h}(T)\ x^d & \text{if } t = x \text{ and } x \colon \texttt{tm}\ T \\ \texttt{Hom}^c(c) & \text{if } t = c \\ \text{compositionally recurse into } t & \text{otherwise} \end{cases}$$

(a) Cases for SFOL's LF types and LF Primitives

$$\begin{array}{lll}
\overline{h}(\wedge) & : & \Pi\, F^d\ F^c\ F^h\ G^d\ G^c\ G^h.\ \Vdash (F^d \wedge G^d) \Rightarrow (F^c \wedge G^c) \\
& & = \lambda F^d\ F^c\ F^h\ G^d\ G^c\ G^h.\ \Rightarrow \texttt{I}_{pf}\ \wedge\texttt{I}\ (F^h\ \Rightarrow\texttt{E}\ (pf\ \wedge\texttt{EL}))\ (G^h\ \Rightarrow\texttt{E}\ (pf\ \wedge\texttt{ER})) \\
\overline{h}(\vee) & : & \Pi\, F^d\ F^c\ F^h\ G^d\ G^c\ G^h.\ \Vdash (F^d \vee G^d) \Rightarrow (F^c \vee G^c) \\
& & = \lambda F^d\ F^c\ F^h\ G^d\ G^c\ G^h.\ \Rightarrow \texttt{I}_{pf}\ pf\ \vee\texttt{E}\ \ (\lambda \tilde{pf} \colon \Vdash F^d.\ \vee\texttt{IL}\ (F^h\ \Rightarrow\texttt{E}\ \tilde{pf})) \\
& & \qquad (\lambda \tilde{pf} \colon \Vdash G^d.\ \vee\texttt{IR}\ (G^h\ \Rightarrow\texttt{E}\ \tilde{pf}')) \\
\overline{h}(\doteq) & : & \Pi\, T^d\ T^c\ T^h.\ \Pi\, x_1\ x_2 \colon \texttt{tm}\ T^d.\ \Vdash x_1 \doteq x_2 \Rightarrow (T^h\ x_1 \doteq T^h\ x_2) \\
& & = \lambda T^d\ T^c\ T^h\ x_1\ x_2.\ \text{apply } T^h \text{ on both sides} \\
\overline{h}(\exists) & : & \Pi\, T^d\ T^c\ T^h\ p^d\ p^c\ p^h.\ \Vdash (\exists p^d) \Rightarrow (\exists p^c) \\
& & = \lambda T^d\ T^c\ T^h\ p^d\ p^c\ p^h.\ \Rightarrow \texttt{I}_{pf}\ pf\ \exists\texttt{E}_{t^d, p_t}\ \ \exists\texttt{I}\ T^c\ p^c\ (T^h\ t^d)\ ((p^h\ t^d)\ \Rightarrow\texttt{E}\ p_t)
\end{array}$$

(b) Cases for SFOL Connectives
Unless otherwise noted, for readability we omit the typings $T^d, T^c \colon \texttt{tp}$, $T^h \colon \texttt{tm}\ T^d \to \texttt{tm}\ T^c$, $p^d \colon \texttt{tm}\ T^d \to \texttt{prop}$, $p^c \colon \texttt{tm}\ T^c \to \texttt{prop}$, $p^h \colon (\Pi\, x^d\ .\ \Vdash p^d\ x^d \Rightarrow p^c\ (T^h\ x^d))$, $F^d, F^c, G^d, G^c \colon \texttt{prop}$, $F^h \colon \Vdash F^d \Rightarrow F^c$, $G^h \colon \Vdash G^d \Rightarrow G^c$.

Figure 24 Translation $\overline{h}$

▶ **Theorem 109.** `Hom` *as given in Definition 107 coincides on definitionless SFOL-theories with the definition given at the beginning of Section 5.3.1. In particular, assuming type symbols* $T_1, \dots, T_n, T : \mathtt{tp}$ *it translates*

- *type symbols* $T : \mathtt{tp}$ *to* T^d, T^c, *and*

  $$T^h : \mathtt{tm}\ T^d \to \mathtt{tm}\ T^c$$

- *function symbols* $f : \mathtt{tm}\ T_1 \to \dots \to \mathtt{tm}\ T_n \to \mathtt{tm}\ T$ *to* f^d, f^c, *and*

  $$f^h : \Pi\, x_1^d : \mathtt{tm}\ T_1^d.\ \dots \Pi\, x_n^d : \mathtt{tm}\ T_n^d.\ \Vdash T^h\ (f^d\ x_1^d\ \dots\ x_n^d) \doteq f^c\ (T_1^h\ x_1^d)\ \dots\ (T_n^h\ x_n^d)$$

- *predicate symbols* $p : \mathtt{tm}\ T_1 \to \dots \to \mathtt{tm}\ T_n \to \mathtt{prop}$ *to* p^d, p^c, *and*

  $$p^h : \Pi\, x_1^d : \mathtt{tm}\ T_1^d.\ \dots \Pi\, x_n^d : \mathtt{tm}\ T_n^d.\ \Vdash p^d\ x_1^d\ \dots\ x_n^d \Rightarrow p^c\ (T_1^h\ x_1^d)\ \dots\ (T_n^h\ x_n^d)$$

Proof. The claim for type symbols follows immediately by definition expansion. For function symbols (analogously for predicate symbols), a straightforward induction suffices, whose base case we have effectively already exercised in Example 108. ◀

Partiality of `Hom` Note that the translation h specified in Figure 24 is partial in two ways. First, it is undefined on axioms with the effect that `Hom` maps axiom symbols to just two qualified copies. Second, it is undefined on certain propositional connectives. Concretely, h is only defined *monotone* propositions:

▶ **Definition 110.** *We call an* `SFOL`*-proposition* ***monotone*** *if it only uses the connectives* $\vee, \wedge, \exists$ *and possibly derived ones (but not* $\neg, \Rightarrow, \forall$*).*

A monotone formulae has the property that if it is true in a model M, it remains true in any supermodel of M. In other words, F stays true when enlarging domains of discourse. It is well-known that homomorphisms only preserve monotone formulae We refer to [Rab17b, Rem. 3.5] for an analysis of the problem in the context of MMT theories and morphisms.

This partiality is a bit awkward but desirable in practice: users trying to apply Hom to a theory or morphism violating the restriction are usually unaware of this issue and thus have inconsistent expectections of the behavior of Hom.

Note that on SFOL propositions `Hom` can only be defined on **monotone formulae** given by the grammar below.

State this theorem:

▶ **Theorem 111.** *Homomorphisms preserve truth of monotone formulae.*

Proof. Follows immediately from above, indeed logrel yields proof for every formulae. ◀

5.3.4 Examples

▶ **Example 112** (Homomorphisms of Categories are Functors). Applying `Hom` on the theory formalizing small categories from Example 102 yields `Hom(SmallCat)`, the theory of functors between small categories:

$$
\begin{array}{l}
\textbf{theory}\ \mathtt{Hom(SmallCat)} = \{ \\
\quad \mathtt{obj}^d, \mathtt{hom}^d, \mathtt{id}^d, \circ^d, \mathtt{neutL}^d, \mathtt{neutR}^d, \mathtt{assoc}^d \\
\quad \mathtt{obj}^c, \mathtt{hom}^c, \mathtt{id}^c, \circ^c, \mathtt{neutL}^c, \mathtt{neutR}^c, \mathtt{assoc}^c \\
\quad \mathtt{obj}^h : \mathtt{tm}\ \mathtt{obj}^d \to \mathtt{tm}\ \mathtt{obj}^c \\
\quad \mathtt{hom}^h : \Pi\, a\ b\colon \mathtt{tm}\ \mathtt{obj}^d.\ \mathtt{tm}\ \mathtt{hom}^d\, a\, b \to \mathtt{tm}\ \mathtt{hom}^c\, (\mathtt{obj}^h\, a)\, (\mathtt{obj}^h\, b) \\
\quad \mathtt{id}^h\ \ : \Pi\, a\colon \mathtt{tm}\ \mathtt{obj}^d.\ \Vdash \mathtt{hom}^h\ a\ a\ (\mathtt{id}^d\ a) \doteq \mathtt{id}^c\ (\mathtt{obj}^h\ a) \\
\quad \circ^h\ \ \ : \Pi\, a\ b\ c\colon \mathtt{tm}\ \mathtt{obj}^d.\ \Pi\, f\colon \mathtt{tm}\ \mathtt{hom}^d\, a\, b.\ \Pi\, g\colon \mathtt{tm}\ (\mathtt{hom}^d\, b\, c). \\
\qquad\qquad \Vdash \quad (\mathtt{hom}^h\ (\mathtt{obj}^h\ a)\ (\mathtt{obj}^h\ c)) f \circ^d g \\
\qquad\qquad\quad \doteq (\mathtt{hom}^h\ (\mathtt{obj}^h\ a)\ (\mathtt{obj}^h\ b)\ f) \circ^c (\mathtt{hom}^h\ (\mathtt{obj}^h\ b)\ (\mathtt{obj}^h\ b)\ c) \\
\}
\end{array}
$$

The d-superscripted constants (respectively, the c-superscriptes ones) collectively represent the domain (codomain) category structure. For readability, we put them both first in the presentation above and omitted their types, which can be read off (modulo systematic renamings) the source theory `SmallCat` from Example 102 anyway. The h-superscripted constants represent the functor as such: $\mathtt{obj}^h$ represents its mapping on objects, $\mathtt{hom}^h$ its mapping on morphisms, $\mathtt{id}^h$ preservation of identity morphisms, and $\circ^h$ represents preservation of morphism composition.

Unwieldy, and to make ready for next example, we bedienen uns hier eines mmt features, welches wir rausgelassen haben aus der arbeit der einfachheit halber, aber ein core feature von mmt implementierung ist:

$$
\begin{array}{l}
\textbf{theory}\ \mathtt{Hom(SmallCat)} = \{ \\
\quad \textbf{struct}\ \mathtt{dom}\colon \mathtt{SmallCat} \\
\quad \textbf{struct}\ \mathtt{cod}\colon \mathtt{SmallCat} \\
\quad \mathtt{obj}^h : \mathtt{tm}\ \mathtt{dom/obj} \to \mathtt{tm}\ \mathtt{cod/obj} \\
\quad \mathtt{hom}^h : \Pi\, a\ b\colon \mathtt{tm}\ \mathtt{dom/obj}.\ \mathtt{tm}\ \mathtt{dom/hom}\, a\, b \to \mathtt{tm}\ \mathtt{cod/hom}\, (\mathtt{obj}^h\, a)\, (\mathtt{obj}^h\, b) \\
\quad \mathtt{id}^h\ \ : \Pi\, a\colon \mathtt{tm}\ \mathtt{dom/obj}.\ \Vdash \mathtt{hom}^h\ a\ a\ (\mathtt{dom/id}\ a) \doteq \mathtt{id}^c\ (\mathtt{obj}^h\ a) \\
\quad \circ^h\ \ \ : \Pi\, a\ b\ c\colon \mathtt{tm}\ \mathtt{dom/obj}.\ \Pi\, f\colon \mathtt{tm}\ \mathtt{dom/hom}\, a\, b.\ \Pi\, g\colon \mathtt{tm}\ (\mathtt{dom/hom}\, b\, c). \\
\qquad\qquad \Vdash \quad (\mathtt{hom}^h\ (\mathtt{obj}^h\ a)\ (\mathtt{obj}^h\ c)) f \mathtt{dom/} \circ g \\
\qquad\qquad\quad \doteq (\mathtt{hom}^h\ (\mathtt{obj}^h\ a)\ (\mathtt{obj}^h\ b)\ f) \mathtt{cod/} \circ (\mathtt{hom}^h\ (\mathtt{obj}^h\ b)\ (\mathtt{obj}^h\ b)\ c) \\
\}
\end{array}
$$

$$
\begin{array}{l}
\textbf{theory}\ \mathtt{Hom(Hom(SmallCat))} = \{ \\
\quad \textbf{struct}\ \mathtt{dom}\colon \mathtt{Hom(SmallCat)} \\
\quad \textbf{struct}\ \mathtt{cod}\colon \mathtt{Hom(SmallCat)} \\
\quad \textbf{struct}\ \mathtt{dom}\colon \mathtt{Hom(SmallCat)} \\
\quad \textbf{struct}\ \mathtt{cod}\colon \mathtt{Hom(SmallCat)} \\
\quad \mathtt{obj}^{h^h} : \Pi\, x\colon \mathtt{tm}\ \mathtt{dom/dom/obj}.\ \mathtt{cod/} \\
\}
\end{array}
$$

▶ Remark 113. We *cannot* extend Example 112 to large categories, where the sizes of the object collection and the one of hom collections differ. It is impossible to formalize such size difference within a PDFOL theory to begin with.

Nonetheless, one minimally invasive way to formalize large categories as an MMT theory would be to modify `SmallCat` such that `obj : type` becomes an LF type (while the return type of `hom` remains an SFOL type). and to replace every occurrence of `tm obj` by `obj`. The declaration of `obj` is what makes `LargeCat` no longer PDFOL-well-patterned.

write more, what happens when objects *and* morphisms are both `tp`. possibly something goes wrong, semantics possibly

▶ **Example 114** (Homomorphisms of Functors are Natural Transformations). ... need identification here (= pushout?)

Finally we can return to our motivating example from ...:

▶ **Example 115** (`Hom(Lists)` formalizes the map operation on lists). We apply `Hom` to the theory `Lists` from Example 101 to obtain `Hom(Lists)` (not shown), in which we identify the constant pairs ($\mathtt{list}^d, \mathtt{list}^c$), ($\mathtt{nil}^d, \mathtt{nil}^c$), and ($\mathtt{cons}^d, \mathtt{cons}^c$) to obtain $\mathtt{Hom(Lists)}'$ (shown below).

theory $\mathtt{Hom(Lists)}' = \{$

$\quad \mathtt{list} \; : \mathtt{tp} \to \mathtt{tp}$

$\quad \mathtt{nil} \; : \Pi\, A\colon \mathtt{tp}.\ \mathtt{tm}\ \mathtt{list}\ A$

$\quad \mathtt{cons} \; : \Pi\, A\colon \mathtt{tp}.\ \mathtt{tm}\ A \to \mathtt{tm}\ \mathtt{list}\ A^p \to \mathtt{tm}\ \mathtt{list}\ A$

$\quad \mathtt{list}^h : \Pi\, A^d\colon \mathtt{tp}.\ \Pi\, A^c\colon \mathtt{tp}.\ \Pi\, A^h\colon \mathtt{tm}\ A^d \to \mathtt{tm}\ A^c.\ \mathtt{tm}\ \mathtt{list}\ A^d \to \mathtt{tm}\ \mathtt{list}\ A^c$

$\quad \mathtt{nil}^h \; : \Pi\, A^d\colon \mathtt{tp}.\ \Pi\, A^c\colon \mathtt{tp}.\ \Pi\, A^h\colon \mathtt{tm}\ A^d \to \mathtt{tm}\ A^c.\ \Vdash \mathtt{list}^h\ A^d\ A^c\ A^h\ (\mathtt{nil}\ A^d) \doteq (\mathtt{nil}\ A^c)$

$\quad \mathtt{cons}^h : \Pi\, A^d\colon \mathtt{tp}.\ \Pi\, A^c\colon \mathtt{tp}.\ \Pi\, A^h\colon \mathtt{tm}\ A^d \to \mathtt{tm}\ A^c.$

$\qquad\qquad \Pi\, x^d\colon \mathtt{tm}\ A^d.\ \Pi\, l^d\colon \mathtt{tm}\ \mathtt{list}\ A^d.$

$\qquad\qquad \Vdash \mathtt{list}\ A^d\ (\mathtt{cons}\ A^d\ x^d\ l^d) \doteq \mathtt{cons}\ A^c\ (A^h\ x^d)\ (\mathtt{list}^h\ l^d)$

$\}$

We can think of $\mathtt{nil}^h$ and $\mathtt{cons}^h$ as the defining (inductive) equations for the function $\mathtt{list}^h$. For any given function $A^h\colon \mathtt{tm}\ A^d \to \mathtt{tm}\ A^c$ (for any types A^d, A^c), these equations uniquely determine the action of the function $\mathtt{list}^h\ A^d\ A^c\ A^h$; namely it maps lists over A^d to A^c elementwise via A^h. This action is the well-known "list map operation".

5.3.5 Meta-Theoretical Properties

We spend the remainder of this section proving meta-theoretical properties of our functor and its connectors.

The following theorem confirms the intuition laid down in Figure 23:

▶ **Proposition 116** (Basic Lemma for h). *We have*

- *if* $\Gamma \vdash_{\mathtt{SFOL}} t\colon A$ *and* h *is defined for* t, *then* $\overline{h}$ *is defined for* A *and* $h(\Gamma) \vdash_{\mathtt{SFOL}} h(t)\colon h(A)\ d(t)\ c(t)$
- *if* h *is term-total, it is defined for a typed term if it is for its type*

Proof. In principle, the proof proceeds similarly to the one of Theorem 11 for actual logical relations. We already made the differences plausible right after **??**. ◀

We can now prove rigorously the well-known theorem

▶ **Theorem 117** (Homomorphisms Preserve Complex Terms & Propositions).
- *Homomorphisms preserve arbitrarily complex terms:*

$$\Sigma \vdash^{\mathtt{SFOL}}_{\Sigma} t\colon \mathtt{tm}\ a \qquad \Longrightarrow \qquad \mathtt{Hom}(\Sigma) \vdash^{\mathtt{SFOL}}_{\Sigma} h(t)\colon h(a)\ d(t) \doteq c(t)$$

For the special case of an atomic term $t = c \in \Sigma$ *we obtain the result* $\mathtt{Hom}(\Sigma) \vdash^{\mathtt{SFOL}}_{\Sigma} c^h : h(a)\ c^d \doteq c^c$.

- *Homomorphisms preserve arbitrarily complex monotone propositions:*

$$\Sigma \vdash^{\mathtt{SFOL}}_{\Sigma} F : \mathtt{prop} \qquad \Longrightarrow \qquad \mathtt{Hom}(\Sigma) \vdash^{\mathtt{SFOL}}_{\Sigma} \Vdash h(F) : d(F) \Rightarrow c(F)$$

Proof. By previous theorem. ◀

▶ **Theorem 118.** $\mathtt{Hom}^d$ *and* $\mathtt{Hom}^c$ *are both well-typed and natural.*

Proof. By Theorem 38 in conjunction with the next theorem. ◀

▶ **Theorem 119.** $\mathtt{Hom}$ *is well-typed.*

Proof. For well-typedness we use the criterion from Theorem 24. Assume $\Sigma \vdash^{\mathtt{SFOL}}_{\Sigma} c : A\,[= t]$ is a well-typed declaration in Σ. Applying $\mathtt{Hom}$ yields three judgements that we need to show:

$$\mathtt{Hom}(\Sigma) \vdash^{\mathtt{SFOL}}_{\Sigma} c^d : A^d\,[= t^d] \qquad \mathtt{Hom}(\Sigma) \vdash^{\mathtt{SFOL}}_{\Sigma} c^c : A^c\,[= t^c]$$
$$\mathtt{Hom}(\Sigma) \vdash^{\mathtt{SFOL}}_{\Sigma} c^h : h(A)\ c^d\ c^c\,[= h(t)]$$

The first two judgements emerge immediately from the assumption given that $-^d$ and $-^c$ are mere systematic renamings. And the third judgement amounts to Proposition 116. ◀

▶ **Conjecture 120.** *Under mild assumptions (e.g., of additional equational theory in* $\mathtt{SFOL}$*),* $\mathtt{Hom}$ *is functorial.*

5.4 Substructures

$$\begin{tikzcd} T \arrow[r, shift left=2, "\texttt{Sub}^p"] \arrow[r, "\texttt{Sub}^m"] \arrow[r, shift right=2, "s"'] & \texttt{Sub}(T) \end{tikzcd}$$

The linear functor $\texttt{Sub}(-)$ maps every PFOL-theory T to the $\texttt{SFOL}$-extension $\texttt{Sub}(T)$ of T-substructures, whose models are submodels of T-models. For example, applied to the theory $\texttt{Group}$ it yields the theory $\texttt{Sub}(\texttt{Group})$ formalizing exactly subgroups. The linear connector p into $\texttt{Sub}$ is the projection of the very $\underline{\text{p}}$arent T-model, and the linear connector m is the realization of a T-$\underline{\text{m}}$odel from a T-submodel via predicate subtypes (which are available when suitably extending PDFOL; we defer describing the latter connector to **??**.) And the logical relation-inspired translation s proves that T-submodels are not only closed under atomic but also arbitrarily complex terms built out of function symbols from T.[16]

5.4.1 Preliminary Definition: Sub on Definitionless SFOL

For didactic reasons, we first define $\texttt{Sub}$ and the connector $\texttt{Sub}^p$ for the special case of definitionless SFOL-theories. As given in the list below, $\texttt{Sub}$ linearly maps each type, function, and axiom symbol declaration $c\colon A$ to two declarations c^p and c^s. The constant c^p serves as a qualified copy to build up the $\underline{\text{p}}$arent model, and the constant c^s accounts for the actual condition imposed on the $\underline{\text{s}}$ubstructure at the very input declaration. For function symbols this amounts to the substructure being closed under corresponding function applications, and for axiom symbols this amounts to the substructure fulfilling relativized variants of the parent's axioms. Finally, predicate symbols are mapped to their qualified copy only.

- type symbols $T\colon \texttt{tp}$ are mapped to a copy and a predicate on terms thereof:

$$\begin{aligned} &T^p : \texttt{tp} \\ &T^s : \texttt{tm}\ T^p \to \texttt{prop} \end{aligned}$$

- function symbols $f\colon \texttt{tm}\ T_1 \to \ldots \to \texttt{tm}\ T_n \to \texttt{tm}\ T$ are mapped to a copy and a closure axiom:

$$\begin{aligned} &f^p : \texttt{tm}\ T_1^p \to \ldots \to \texttt{tm}\ T_n^p \to \texttt{tm}\ T^p \\ &f^s : \Pi\, t_1 : \texttt{tm}\ T_1.\ \Pi\, t_1^s :\ \Vdash T_1^s\ t_1. \\ &\quad\ \vdots \\ &\quad\ \Pi\, t_n : \texttt{tm}\ T_n.\ \Pi\, t_n^s :\ \Vdash T_n^s\ t_n.\ \Vdash T^s\ (f^p\ t_1\ \ldots\ t_n) \end{aligned}$$

- predicate symbols $p\colon \texttt{tm}\ T_1 \to \ldots \to \texttt{tm}\ T_n \to \texttt{prop}$ are mapped to just a copy:

$$p^p : \texttt{tm}\ T_1^p \to \ldots \to \texttt{tm}\ T_n^p \to \texttt{prop}$$

- axiom symbols $ax\colon \Vdash F$ are mapped to a copy and a relativized variant:

$$\begin{aligned} &ax^p : \Vdash F^p \\ &ax^s : \Vdash F^s \end{aligned}$$

  where F^s emerges from F^p by replacing *i*) every universally quantified formula $\forall\, x\colon \texttt{tm}\, T^p.\ \phi(x)$ by $\forall\, x\colon \texttt{tm}\, T^p.\ T^s\ x \Rightarrow \phi(x)$ and *ii*) every existentially quantified formula $\exists\, x\colon \texttt{tm}\, T^p.\ \phi(x)$ by $\exists\, x\colon \texttt{tm}\ T^p.\ T^s\ x \wedge \phi(x)$.

[16] See Theorem 131.

$$
\begin{array}{l}
\textbf{theory}\ \mathtt{Sub}(\mathtt{Unital}) = \{ \\
\quad \textbf{include}\ \mathtt{SFOL} \\
\quad \begin{array}{ll}
U^p & : \mathtt{tp} \\
U^s & : \mathtt{tm}\ U^p \to \mathtt{prop} \\
\circ^p & : \mathtt{tm}\ U^p \to \mathtt{tm}\ U^p \to \mathtt{tm}\ U^p \\
\circ^s & : \Pi\, x^p : \mathtt{tm}\ U^p.\ \Pi\, x^s :\ \Vdash U^s\ x^p. \\
 & \quad \Pi\, y^p : \mathtt{tm}\ U^p.\ \Pi\, y^s :\ \Vdash U^s\ y^p. \\
 & \quad \Vdash U^s\ (x^p \circ y^p) \\
e^p & : \mathtt{tm}\ U^p \\
e^s & : \Vdash U^s\ e^p \\
\mathtt{neut}^p & : \Vdash \forall\, x^p.\ e^p \circ x^p \doteq x^p \\
\mathtt{neut}^s & : \Vdash \forall\, x^p.\ U^s\ x^p \Rightarrow e^p \circ x^p \doteq x^p
\end{array} \\
\} \\
\\
\textbf{mor}\ \mathtt{Sub}^p(\mathtt{Unital}) : \mathtt{Unital} \to \mathtt{Sub}(\mathtt{Unital}) = \{ \\
\quad = = \\
\}
\end{array}
$$

Figure 25 Theory of unital substructures given by $\mathtt{Sub}(\mathtt{Unital})$ and corresponding parent projection

The linear connector $\mathtt{Sub}^p$ into $\mathtt{Sub}$ maps every constant c to $c := c^p$.

Our restriction to definitionless SFOL-theories makes it easy to see that $\mathtt{Sub}$ is well-typed and functorial. Functoriality holds vacuously since $\mathtt{Sub}$ is partial on all defined constants, including morphism assignments, thus in particular on all morphisms. By Theorem 38, the connector $\mathtt{Sub}^p$ is well-typed and natural (the latter again vacuously).

▶ **Example 121** ($\mathtt{Sub}(\mathtt{Unital})$)**.** For the theory $\mathtt{Unital}$ from Example 97 we obtain the theory $\mathtt{Sub}(\mathtt{Unital})$ and the morphism $\mathtt{Sub}^p(\mathtt{Unital}) : \mathtt{Unital} \to \mathtt{Sub}(\mathtt{Unital})$ shown in Figure 25. We see that the constants $U^p, \circ^p, e^p, \mathtt{neut}^p$ collectively make up a unital structure, and this is precisely what is witnessed by the morphism. Within $\mathtt{Sub}(\mathtt{Unital})$, additional structure is imposed on these constants by means of the s-superscripted constants. Importantly, U^s represents a selection of terms of type U^p (a "subset"), and the constants $\circ^s$ and e^s force this selection to be closed under the corresponding operations. Lastly, the generated axiom $\mathtt{neut}^s$ actually does not impose any additional structure since it is trivially provable from $\mathtt{neut}^p$. In fact, for all axioms ax using only $\forall$ and $\doteq$ (i.e., the axioms primarily used in universal algebra) the corresponding axioms ax^s are automatically true in each submodel and thus provable. Here, $\mathtt{Sub}$ could do much better and translate such axioms to theorems by synthesizing an appropriate proof and adding it as the definiens of ax^s. We continue discussing this avenue of future work in Section 5.8.

▶ Remark 122.

▶ Remark 123. say it's weird how we distinguish between defined predicate symbols and just propositional terms, in our encoding defined predicate symbols are no longer subject to the general guarantees of abbreviations.

$\Sigma \supseteq$ SFOL-expression		mapped to Sub(Σ)-expression	
types	$T : \mathtt{tp}$	$\overline{s}(T) : \mathtt{tm}\ T^d \to \mathtt{prop}$	selection of subset of T^p
terms	$t\ : \mathtt{tm}\ T$	$\overline{s}(t)\ \ : \Vdash \overline{s}(T)\ p(t)$	proof of t^p being in subset at T^p selected by $\overline{s}(t)$
propositions	$F : \mathtt{prop}$	$\overline{s}(F) : \mathtt{prop}$	proposition relativized in quantifiers
proofs	$pf : \Vdash F$	$\overline{s}(pf) : \Vdash \overline{s}(F)$	proof of relativized proposition

Figure 26 Intuitive overview of the translation to be carried out by $\overline{s}$

5.4.2 Building Towards a Generalized Definition

We now aim to generalize Sub to a linear functor accepting all PDFOL-theories. In this section, we collect ideas and a failing ansatz, culminating in a succeeding definition for the special case of PFOL-theories in the next section (Section 5.4.3). The general principle remains as before: we copy every input constant $c\colon A\,[= t]$ to one qualified copy $c^p\colon A^p\,[= t^p]$ (where $-^p$ indicates a systematic renaming operation) and moreover output a designated second constant c^s for everything but predicate symbols. Similarly as with generalizing Hom in Section 5.3.1, the main hurdle lies in specifying the translation that is needed to generate the second constant here, and likewise the key idea is to define a translation while having a logical relation on $\mathtt{Sub}^p$ in mind and suitably modifying inductive cases.

Reading off the definition of Sub given above, we come up with the following base cases for a putative logical relation $\overline{s}$ on $\mathtt{Sub}^p$:

$$\begin{aligned}
\overline{s}(\mathtt{tp}) &= \lambda U^p\colon \mathtt{tp}.\ \mathtt{tm}\ U^p \to \mathtt{prop}\\
\overline{s}(\mathtt{tm}) &= \lambda T^p\colon \mathtt{tp}.\ \lambda T^s\colon \mathtt{tm}\ T^p \to \mathtt{prop}.\ \lambda t^p\colon \mathtt{tm}\ T^p.\ \Vdash T^s\ t^p\\
\overline{s}(\mathtt{prop}) &= \lambda F^p\colon \mathtt{prop}.\ \mathtt{prop}\\
\overline{s}(\Vdash) &= \lambda F^p\ F^s\colon \mathtt{prop}.\ \lambda pf\colon\ \Vdash F^p.\ \Vdash F^s
\end{aligned}$$

If we actually extended $\overline{s}$ to a logical relation, it would map SFOL expressions as overviewed in Figure 26.

This sounds right, but is elusive: consider a context with variables $T\colon \mathtt{tp}$ and $p\colon \mathtt{tm}\ T \to \mathtt{prop}$. If $\overline{s}$ was really extended to a logical relation, applying it to the context would yield the variable $p^s : \Pi\, x^p : T^p.\ \Pi\, x^s :\ \Vdash T^s\ x^p.\ \mathtt{prop}$ in the translated context. Here, the introduction of the parameter x^s leads to undesired consequences. This is best seen when trying to define our putative logical relation on the symbols for universal and existential quantification from SFOL. Recall these symbol declarations from SFOL:

$$\forall, \exists\colon \Pi\, T\colon \mathtt{tp}.\ \Pi\, p\colon \mathtt{tm}\ T \to \mathtt{prop}.\ \mathtt{prop}$$

The expected types of respective assignments would be:

$$\begin{aligned}
&\overline{s}(\forall), \overline{s}(\exists)\colon \Pi\, T^p\colon \mathtt{tp}.\ \Pi\, T^s\colon \mathtt{tm}\ T^p \to \mathtt{prop}.\\
&\qquad \Pi\, p^p\colon \mathtt{tm}\ T^p \to \mathtt{prop}.\ \Pi\, p^s\colon\ (\Pi\, x^p\colon \mathtt{tm}\ T^p.\ \Pi\, x^s\colon\ \Vdash T^s\ x^p.\ \mathtt{prop})\,.\\
&\qquad \mathtt{prop}
\end{aligned}$$

We can think of the parameter p^p as the predicate originating from the parent's structure and of p^s as the relativized variant. But here the type of p^s, particularly its parameter x^s, makes it impossible to give any meaningful assignments that are in line with our definition of Sub at the beginning of Section 5.4. Ideally, we would have hoped to make the following assignments:

$$\overline{s}(\forall) = \lambda T^p\, T^s\, p^p\, p^s.\ \forall\, x^p \colon \mathtt{tm}\ T^p.\quad T^s\ x^p \Rightarrow \overbrace{p^s\ x^p}^{\text{impossible!}} \tag{1}$$

$$\overline{s}(\exists) = \lambda T^p\, T^s\, p^p\, p^s.\ \exists\, x^p \colon \mathtt{tm}\ T^p.\quad T^s\ x^p \wedge \underbrace{p^s\ x^p}_{\text{impossible}} \tag{2}$$

But both lines are ill-typed, precisely because we cannot satisfy the required parameter x^s of p^s.

Note that the antecedences in both lines are identical to the sought after type of x^s. If in the consequences

As a fix, in the next section we aim at patching the cases of $\overline{s}$ induced by being a logical relation such that for predicate functions $q \colon \mathtt{tm}\ T \to \mathtt{prop}$ (not necessarily a predicate symbol declaration) applying $\overline{s}$ creates a parameter $x^p \colon \mathtt{tm}\ T^p$, but avoids introducing any parameter $x^s : T^s\ x^p$. In general, we will prescribe this avoidance even for complex predicate functions of the form $q \colon \Pi\, x_1 \colon \mathtt{tm}\ T_1.\ \ldots \Pi\, x_n \colon \mathtt{tm}\ T_n.\ \mathtt{prop}$. Applying $\overline{s}$ to those will simply yield the expected type

$$\overline{s}(q) \colon \Pi\, x_1^p \colon \mathtt{tm}\ T_1^p.\ \ldots \Pi\, x_n^p \colon \mathtt{tm}\ T_n^p.\ \mathtt{prop}$$

This fix also goes hand-in-hand with how the definition of `Sub` at the beginning of Section 5.4 acts on predicate symbol declarations: it only copies them to a parent copy. Since no additional declaration is generated for the substructure, the modified logical relation should also avoid analogous additional parameters/variables when recursing.

▶ Remark 124 (Alternative Fix via Dependent Implication and Conjunction). Having to give up on the original definition of logical relations is frustrating. Consider the desired, but ill-typed assignments in Equations (1) and (2). In both cases, we were unable to supply an argument to the parameter x^s of p^s. Now observe how in Equation (1) the antecedence is identical to the expected type of x^s. If in the consequence we could somehow assume a witness of the antecedence, we could pass it to p^s as the argument for x^s and thereby achieve a well-typed and meaningful assignment. Analogously in Equation (2), if in the right conjunct we could somehow assume a witness of the left conjunct, we could pass it to p^s, too.

Logical features, i.e., variants of implications and conjunctions, that allow such usage are lesser known as *dependent implication* and *dependent conjunction* [HP18]. Their analogue in programming languages is widely known: short circuit evaluation in boolean expressions [Wik22]. For example, in the programming language C if you write the boolean expression `!f() || g()` (encoding an implication from `f()` to `g()`), upon runtime `g()` is only evaluated when `!f()` evaluates to false, i.e., when `f()` evaluates to true. Thus, programmers may write C-style implications (encoded using `!_ || _`) in a way that assumes the truth of the antecedence in the consequence. Analogously, if you write `f() && g()`, upon runtime `g()` is only evaluated when `f()` evalutes to true. Thus, programmers may write C-style conjunctions using `&&` in a way that assumes the truth of left conjuncts in the right conjuncts.

Let us shortly show how the assignments to $\overline{s}(\forall)$ and $\overline{s}(\exists)$ would look like using dependent implication and conjunction. First, we formalize the respective logical features by extending

SFOL as follows [LATIN2][17]:

$$
\begin{aligned}
&\textbf{theory } \mathtt{DepSFOL} = \{\\
&\quad \textbf{include } \mathtt{SFOL}\\
&\quad \mathtt{depImpl}\colon \Pi\, F\colon \mathtt{prop}.\ \Pi\, G\colon\ \Vdash F \to \mathtt{prop}.\ \mathtt{prop}\\
&\quad \mathtt{depConj}\colon \Pi\, F\colon \mathtt{prop}.\ \Pi\, G\colon\ \Vdash F \to \mathtt{prop}.\ \mathtt{prop}\\
&\quad \text{/* /* proof rules omitted */ */}\\
&\}
\end{aligned}
$$

We now conjecture that we can define a linear functor DepSub from PDFOL- to PDFOL + DepSFOL-theories using the exact logical relation suggested at the beginning of Section 5.4.2 and using dependent connectives. (We state our conjecture more formally below as Conjecture 128.) For example the quantifiers could be given the following assignments:

$$
\begin{aligned}
\overline{s}(\forall) &= \lambda T^p\ T^s\ p^p\ p^s.\ \forall\, x^p\colon \mathtt{tm}\ T^p.\quad \mathtt{depImpl}\ (T^s\ x^p)\ (\lambda x^s.\ p^s\ x^p\ x^s)\\
\overline{s}(\exists) &= \lambda T^p\ T^s\ p^p\ p^s.\ \exists\, x^p\colon \mathtt{tm}\ T^p.\quad \mathtt{depConj}\ (T^s\ x^p)\ (\lambda x^s.\ p^s\ x^p\ x^s)
\end{aligned}
$$

We conjecture that the remaining assignments in the logical relation can be taken from our final Definition 126 of Sub below (concretely: Figures 27b and 27c).

The main advantage over the ansatz to modify the logical relation as needed is that we conjecture we could actually get a functor that accepts *all* PDFOL-theories. In contrast, the ansatz we carry out below, culminating in **??**, only works on PFOL-theories and that seems to be limitation that cannot be overcome within that ansatz. And the main disadvantage would be that the theories and morphisms output by Sub would then contain these constructs (which are nothing but artifacts of our way of specifying the translation), and it would be rather awkward for us to push these them onto formalizations that users get to see and need to use.

5.4.3 Final Definition: Sub on PFOL

As for Hom, to finally define $\overline{s}$ as desired, we take the complex cases for $\overline{s}$ that would have been induced if it was a logical relation on $\mathtt{Sub}^p$ and alter them as little as possible to suppress undesired parameters:

▶ **Definition 125.** *The translation $\overline{s}$ maps contexts $\vdash^{\mathtt{SFOL}}_{\Sigma} \Sigma$ and LF terms $\Sigma \vdash^{\mathtt{SFOL}}_{\Sigma} t$ to $\overline{s}^{\Sigma}(t)$ as specified in Figure 27. In particular, it has base cases*

$$
\begin{aligned}
\overline{s}(\mathtt{tp}) &= \lambda U^p\colon \mathtt{tp}.\ \mathtt{tm}\ U^p \to \mathtt{prop}\\
\overline{s}(\mathtt{tm}) &= \lambda T^p\colon \mathtt{tp}.\ \lambda T^s\colon \mathtt{tm}\ T^p \to \mathtt{prop}.\ \lambda t^p\colon \mathtt{tm}\ T^p.\ \Vdash T^s\ t^p\\
\overline{s}(\mathtt{prop}) &= \lambda F^p\colon \mathtt{prop}.\ \mathtt{prop}\\
\overline{s}(\Vdash) &= \lambda F^p\ F^s.\ \lambda pf\colon\ \Vdash F^p.\ \Vdash F^s
\end{aligned}
$$

For brevity, we hide the context in the notation, and in the sequel we also write s for $\overline{s}$.

[17] Inspired by formalizations currently in file source/logic/propositional/dependent_pl.mmt: https://gl.mathhub.info/MMT/LATIN2/-/blob/39dc7046f457ff02f695387a8ebd80366789a465/source/logic/propositional/dependent_pl.mmt

$$
\begin{aligned}
\overline{s}(\texttt{tp}) &= \lambda U^p \colon \texttt{tp}.\ \texttt{tm}\ U^p \to \texttt{prop}\\
\overline{s}(\texttt{tm}) &= \lambda T^p \colon \texttt{tp}.\ \lambda T^s \colon \texttt{tm}\ T^p \to \texttt{prop}.\ \lambda t^p \colon \texttt{tm}\ T^p.\ \Vdash T^s\ t^p\\
\overline{s}(\texttt{prop}) &= \lambda F^p \colon \texttt{prop}.\ \texttt{prop}\\
\overline{s}(\Vdash) &= \lambda F^p\ F^s.\ \lambda pf\colon\ \Vdash F^p.\ \Vdash F^s
\end{aligned}
$$

$$
\begin{aligned}
\overline{s}(\Pi\, x\colon A.\ B) &= \begin{cases} \lambda \mathfrak{F}\colon p(\Pi\, x\colon A.\ B).\ \Pi\, x^p\colon p(A).\ \overline{s}(B)\ (\mathfrak{F}\ x^p) & \text{if } A = \texttt{tm}\ T \text{ and } \mathrm{ret}(B) = \texttt{prop}\\ \lambda \mathfrak{F}\colon p(\Pi\, x\colon A.\ B).\ \Pi\, x^p\colon p(A).\ \Pi\, x^s\colon \overline{s}(A)\ x^p.\ \overline{s}(B)\ (\mathfrak{F}\ x^p) & \text{otherwise}\end{cases}\\
\overline{s}(\lambda x\colon A.\ t) &= \begin{cases} \lambda x^p\colon p(A).\ \overline{s}(t) & \text{if } A = \texttt{tm}\ T,\ t\colon \texttt{prop}\\ \lambda x^p\colon p(A).\ \lambda x^s\colon \overline{s}(A)\ x^p.\ \overline{s}(t) & \text{otherwise}\end{cases}\\
\overline{s}(x) &= x^s\\
\overline{s}(f\ t) &= \begin{cases} \overline{s}(f)\ p(t) & \text{if } f\colon (\Pi\, x\colon \texttt{tm}\ T.\ B) \text{ and } \mathrm{ret}(B) = \texttt{prop}\\ \overline{s}(f)\ p(t)\ \overline{s}(t) & \text{otherwise}\end{cases}
\end{aligned}
$$

where p is defined as $p(t) = \texttt{Sub}^p(t)[x \mapsto x^p]$, i.e., as the result emerging from $\texttt{Sub}^p(t)$ by substituting every variable reference x with x^p

(a) Cases for SFOL's LF types and LF Primitives

$$
\begin{aligned}
\overline{s}(\forall) &: \lambda T^p\ T^s\ p^p\ p^s.\ \forall\, t^p.\ T^s\ t^p \Rightarrow p^s\ t^p\\
\overline{s}(\exists) &: \lambda T^p\ T^s\ p^p\ p^s.\ \exists\, t^p.\ T^s\ t^p \wedge p^s\ t^p\\
\overline{s}(\doteq) &: \lambda T^p\ T^s.\ \lambda x_1^p\ x_2^p.\ x_1^p \doteq x_2^p\\
\overline{s}(\neg) &: \lambda F^p\ F^s.\ \neg F^s\\
\overline{s}(\wedge) &: \lambda F^p\ F^s\ G^p\ G^s.\ F^s \wedge G^s\\
\overline{s}(\vee) &: \lambda F^p\ F^s\ G^p\ G^s.\ F^s \vee G^s\\
\overline{s}(\Rightarrow) &: \lambda F^p\ F^s\ G^p\ G^s.\ F^s \Rightarrow G^s
\end{aligned}
$$

(b) Cases for SFOL Connectives

▶ **Definition 126** (Substructure Operators). *The linear functor* Sub *from PFOL-theories to all possible* SFOL*-extensions and the linear connector p into* Sub *are given by*

$$
\texttt{Sub}^\Sigma(c\colon A\,[= t]) = c^p\colon A^p\,[= t^p],\ \begin{cases} c^s\colon s^\Sigma(A)\ c^p\,[= s^\Sigma(t)] & \textit{if } c \textit{ is a type, fun., or ax. symbol}\\ \cdot & \textit{if } c \textit{ is a predicate symbol}\end{cases}
$$

$$
\texttt{Sub}^{p,\Sigma}(c\colon A) = c := c^p
$$

where s^Σ is the translation from Definition 125 extended by

$$
s^\Sigma(c) = \begin{cases} c^s & \textit{if } c \textit{ is a type, fun., or ax. symbol}\\ c^p & \textit{if } c \textit{ is a predicate symbol}\end{cases}
$$

for every c in Σ.

Let us look at Figure 27 and compare the definitional cases given there with the ones that would have taken effect if s was a unary logical relation on $\texttt{Sub}^p$. The most important change from which all others follow is in the case of s for $\Pi\, x\colon A.\ B$. It has been modified to suppress generation of a parameter $x^s\colon \overline{s}(A)\ x^p$ for function types $\texttt{tm}\ T \to \ldots \to \texttt{prop}$ (incl. the dependent function type cases) where the dots ... indicate an arbitrary number of arbitrary parameter types. Consequently, the cases for function abstraction $(\lambda x\colon A.\ t)$ and application $(f\ t)$ suppress binding/supplying the parameter/an argument for corresponding functions.

$$\begin{array}{ll}
\overline{s}(\forall\mathtt{I}) & : \Pi\, T^p\ T^s\ p^p\ p^s.\ \Pi\, pf^p : (\Pi\, x^p.\ \Vdash p^p\ x^p).\ \Pi\, pf^s : (\Pi\, x^p\ x^s.\ \Vdash p^s\ x^p). \\
 & \quad \Vdash \forall\, t^p.\ T^s\ t^p \Rightarrow p^s\ t^p \\
 & \quad = \lambda T^p\ T^s\ p^p\ p^s\ pf^p\ pf^s.\ \forall\mathtt{I}_{x^p} \Rightarrow \mathtt{I}_{x^s}\ pf^s\ x^p\ x^s \\
\overline{s}(\forall\mathtt{E}) & : \Pi\, T^p\ T^s\ p^p\ p^s.\ \Pi\, pf^p :\ \Vdash \forall p^p.\ \Pi\, pf^s :\ \Vdash \forall\, x^p.\ T^s\ x^p \Rightarrow p^s\ x^p. \\
 & \quad \Pi\, x^p\ x^s.\ \Vdash p^s\ x^p \\
 & \quad = \lambda T^p\ T^s\ p^p\ p^s\ pf^p\ pf^s\ x^p\ x^s.\ (pf^s\ \ \forall\mathtt{E}\ \ x^p)\ \Rightarrow \mathtt{E}\ \ x^s \\
\overline{s}(\exists\mathtt{I}) & : \Pi\, T^p\ T^s\ p^p\ p^s\ x^p\ x^s.\ \Pi\, pf^p :\ \Vdash p^p\ x^p.\ \Pi\, pf^s :\ \Vdash p^s\ x^p.\ \Vdash \exists\, x^p.\ T^s\ x^p \wedge p^s\ x^p \\
 & \quad \lambda T^p\ T^s\ p^p\ p^s\ x^p\ x^s\ pf^p\ pf^s.\ \exists\mathtt{I}\ x^p\ (\wedge\mathtt{I}\ x^s\ pf^s) \\
\overline{s}(\exists\mathtt{E}) & : \Pi\, T^p\ T^s\ p^p\ p^s.\ \Pi\, pf^p :\ \Vdash \exists\, p^p.\ \Pi\, pf^s :\ \Vdash \exists\, x^p.\ T^s\ x^p \wedge p^s\ x^p. \\
 & \quad \Pi\, F^p\ F^s.\ \Pi\, \tilde{pf}^p : (\Pi\, x^p.\ \Vdash p^p\ x^p \rightarrow \Vdash F^p).\ \Pi\, \tilde{pf}^s : (\Pi\, x^p\ x^s.\ \Vdash p^s\ x^p \rightarrow \Vdash F^s).\ \Vdash F^s \\
 & \quad \lambda T^p\ T^s\ p^p\ p^s\ pf^p\ pf^s\ F^p\ F^s\ \tilde{pf}^p\ \tilde{pf}^s.\ pf^s\ \ \exists\mathtt{E}_{x^p,y}\ \ \tilde{pf}^s\ \ x^p\ (y\ \wedge\mathtt{EL})\ (y\ \wedge\mathtt{ER}) \\
\overline{s}(\mathtt{refl}) & : \Pi\, T^p\ T^s\ x^p\ x^s.\ \Vdash x^p \doteq x^p \\
 & \quad = \lambda T^p\ T^s\ x^p\ x^s.\ \mathtt{refl}\ T^p\ x^p \\
\overline{s}(\mathtt{symm}) & : \Pi\, T^p\ T^s\ x^p\ x^s\ y^p\ y^s.\ \Pi\, pf^p\ pf^s :\ \Vdash x^p \doteq y^p.\ \Vdash y^p \doteq x^p \\
 & \quad = \lambda T^p\ T^s\ x^p\ x^s\ y^p\ y^s\ pf^p\ pf^s.\ \mathtt{symm}\ T^p\ pf^p \\
\overline{s}(\mathtt{trans}) & : \Pi\, T^p\ T^s\ x^p\ x^s\ y^p\ y^s\ z^p\ z^s.\ \Pi\, pf^p_1\ pf^s_1 :\ \Vdash x^p \doteq y^p.\ \Pi\, pf^p_2\ pf^s_2 :\ \Vdash y^p \doteq z^p.\ \Vdash x^p \doteq z^p \\
 & \quad = \lambda T^p\ T^s\ x^p\ x^s\ y^p\ y^s\ z^p\ z^s\ pf^p_1\ pf^s_1\ pf^p_2\ pf^s_2.\ \mathtt{trans}\ T^p\ x^p\ y^p\ z^p\ pf^p_1\ pf^p_2 \\
\overline{s}(\wedge\mathtt{I}) & : \Pi\, F^p\ F^s\ G^p\ G^s.\ \Pi\, pf^p_F :\ \Vdash F^p.\ \Pi\, pf^s_F :\ \Vdash F^s.\ \Pi\, pf^p_G :\ \Vdash G^p.\ \Pi\, pf^s_G :\ \Vdash G^s.\ \Vdash F^s \wedge G^s \\
 & \quad = \lambda F^p\ F^s\ G^p\ G^s\ pf^p_F\ pf^s_F\ pf^p_G\ pf^s_G.\ \wedge\mathtt{I}\ pf^s_F\ pf^s_G \\
\overline{s}(\wedge\mathtt{EL}) & : \Pi\, F^p\ F^s\ G^p\ G^s.\ \Pi\, pf^p :\ \Vdash F^p \wedge G^p.\ \Pi\, pf^s :\ \Vdash F^s \wedge G^s.\ \Vdash F^s \\
 & \quad = \lambda F^p\ F^s\ G^p\ G^s\ pf^p\ pf^s.\ pf^s\ \ \wedge\mathtt{EL} \\
\overline{s}(\wedge\mathtt{ER}) & : \Pi\, F^p\ F^s\ G^p\ G^s.\ \Pi\, pf^p :\ \Vdash F^p \wedge G^p.\ \Pi\, pf^s :\ \Vdash F^s \wedge G^s.\ \Vdash G^s \\
 & \quad = \lambda F^p\ F^s\ G^p\ G^s\ pf^p\ pf^s.\ pf^s\ \ \wedge\mathtt{ER} \\
\overline{s}(\vee\mathtt{IL}) & : \Pi\, F^p\ F^s\ G^p\ G^s.\ \Pi\, pf^p :\ \Vdash F^p.\ \Pi\, pf^s :\ \Vdash F^s.\ \Vdash F^s \vee G^s \\
 & \quad = \lambda F^p\ F^s\ G^p\ G^s\ pf^p\ pf^s.\ \vee\mathtt{IL}\ pf^s \\
\overline{s}(\vee\mathtt{IR}) & : \Pi\, F^p\ F^s\ G^p\ G^s.\ \Pi\, pf^p :\ \Vdash G^p.\ \Pi\, pf^s :\ \Vdash G^s.\ \Vdash F^s \vee G^s \\
 & \quad = \lambda F^p\ F^s\ G^p\ G^s\ pf^p\ pf^s.\ \vee\mathtt{IR}\ pf^s \\
\overline{s}(\vee\mathtt{E}) & : \Pi\, F^p\ F^s\ G^p\ G^s.\ \Pi\, pf^p :\ \Vdash F^p \vee G^p.\ \Pi\, pf^s :\ \Vdash F^s \vee G^s.\ \Pi\, H^p\ H^s.\ \Pi\, pf^p_1 :\ \Vdash F^p \rightarrow \Vdash H^p. \\
 & \quad \Pi\, pf^s_1 :\ \Vdash F^s \rightarrow \Vdash H^s.\ \Pi\, pf^p_2 :\ \Vdash G^p \rightarrow \Vdash H^p.\ \Pi\, pf^s_2 :\ \Vdash G^s \rightarrow \Vdash H^s.\ \Vdash H^s \\
 & \quad = \lambda F^p\ F^s\ G^p\ G^s\ pf^p\ pf^s\ H^p\ H^s\ pf^p_1\ pf^s_1\ pf^p_2\ pf^s_2.\ pf^s\ \ \vee\mathtt{E}\ \ pf^s_1\ pf^s_2
\end{array}$$

(c) Cases for `SFOL` Proof Rules

Unless otherwise noted, for readability we omit the typings $T^p\colon \mathtt{tp}$, $T^s\colon \mathtt{tm}\ T^s \rightarrow \mathtt{prop}$, $p^p, p^s\colon \mathtt{tm}\ T^p \rightarrow \mathtt{prop}$, $t^p, x^p, x^p_1, x^p_2\colon \mathtt{tm}\ T^p$, $x^s\colon \Vdash T^s\ x^p$, $F^p, F^s, G^p, G^s, H^p, H^s\colon \mathtt{prop}$.
And ret is defined up to α-renamings as

$$\mathrm{ret}(t) = \begin{cases} \mathrm{ret}(B) & \text{if } t = \Pi\, x\colon A.\ B \\ t & \text{otherwise} \end{cases}$$

Figure 27 Translation $\overline{s}$

▶ **Theorem 127.** $\texttt{Sub}$ *as given in ?? coincides on definitionless SFOL-theories with the definition given at the beginning of Section 5.4.1. In particular, assuming type symbols* $T_1, \dots, T_n, T : \texttt{tp}$ *it translates*

- *type symbols* $T : \texttt{tp}$ *to* T^p *and*

$$T^s : \texttt{tm}\ T^p \to \texttt{prop}$$

- *function symbols* $f : \texttt{tm}\ T_1 \to \dots \to \texttt{tm}\ T_n \to \texttt{tm}\ T$ *to* f^p *and*

$$\begin{aligned} f^s : {} & \Pi\, t_1 : \texttt{tm}\ T_1.\ \Pi\, t_1^s : \ \Vdash T_1^s\ t_1. \\ & \vdots \\ & \Pi\, t_n : \texttt{tm}\ T_n.\ \Pi\, t_n^s : \ \Vdash T_n^s\ t_n.\ \Vdash T^s\ (f^p\ t_1\ \dots\ t_n) \end{aligned}$$

- *predicate symbols* $p : \texttt{tm}\ T_1 \to \dots \to \texttt{tm}\ T_n \to \texttt{prop}$ *to just a copy* p^p
- *axiom symbols* ax: $\Vdash F$ *to* ax^p *and*

$$ax^s : \Vdash F^s$$

where F^s *is the variant of* F^p *in which every quantifier on terms of type* T *is relativized by the predicate* T^s *(see Section 5.4.1)*

Proof. The claims for type and predicate symbols are immediate. The ones for function and axiom symbols follow by unfolding the definition and a simple induction. ◀

Discussion of Limitation to PFOL-Theories To see why $\texttt{Sub}$ as defined in Definition 126 is only applicable to PFOL-theories, consider the DFOL-theory below, which is one of the smallest ones reproducing the issue. For clarity, we are explicit about syntax lest to hide something in notational tricks.

$$\begin{aligned} & \textbf{theory}\ \texttt{Fail} = \{ \\ & \quad T\ : \texttt{tp} \\ & \quad b\ \ : \Pi\, x : \texttt{tm}\ T.\ \texttt{tp} \\ & \quad ax : \Vdash \forall\ T\ (\lambda x : \texttt{tm}\ T.\ \forall\ (b\ x)\ (\lambda y : \texttt{tm}\ b\ x.\ \texttt{true})) \\ & \} \end{aligned}$$

Naively applying $\texttt{Sub}$ would yield the following ill-typed theory, containing a dangling variable reference x^s (underlined):

$$\begin{aligned} & \textbf{theory}\ \texttt{Sub}(\texttt{Fail}) = \{ \\ & \quad T^p\ : \texttt{tp} \\ & \quad T^s\ : \texttt{tm}\ T^p \to \texttt{prop} \\ & \quad b^p\ \ : \Pi\, x^p : \texttt{tm}\ T^p.\ \texttt{tp} \\ & \quad b^s\ \ : \Pi\, x^p : \texttt{tm}\ T^p.\ \Pi\, x^s : \ \Vdash T^s\ x^p.\ \texttt{tm}\ b^p \to \texttt{prop} \\ & \quad ax^p : \Vdash \forall\ T^p\ (\lambda x^p : \texttt{tm}\ T^p.\ \forall\ (b^p\ x^p)\ (\lambda y^p : \texttt{tm}\ b^p\ x^p.\ \texttt{true})) \\ & \quad ax^s : \Vdash \forall\ T^p\ (\lambda x^p : \texttt{tm}\ T^p.\ T^s\ x^p \Rightarrow \forall\ (b^p\ x^p)\ (\lambda y^p : \texttt{tm}\ b^p\ x^p.\ b^s\ x^p\ \underline{x^s}\ y^p \Rightarrow \texttt{true})) \\ & \} \end{aligned}$$

The dangling variable reference stems from applying the case $s(x) = x^s$ from Figure 27a when no such variable x^s has been bound in previous invocations of s in the call stack of s. Observe how, similar to Remark 124, in the line of ax^s with the outer antecedence $T^s\ x^p$ we would have a witness that we could have supplied for the parameter x^s of b^s if we used dependent implications.

In Remark 124 we suggested dependent variants of implications and conjunctions to overcome the *cosmetic issue* of needing to modify the inductive cases of the logical relation-based ansatz. Here, we may be observing a more fundamental problem: possibly, dependent connectives are inevitable when trying to generalize `Sub` to all PDFOL-theories. In any case, we conjecture the following:

▶ **Conjecture 128.** *Assume a theory* `DepSFOL` *formalizing dependent connectives on top of* `SFOL`*. We conjecture that we can define a linear functor* `DepSub` *from PDFOL- to PDFOL +* `DepSFOL`*-theories that on definitionless SFOL-theories reasonably*[18] *agrees with* `Sub` *as defined at the beginning of Section 5.4.1 and that is defined using an (unmodified!) unary logical relation $\overline{s}$ with base cases*

$$\begin{aligned}
\overline{s}(\mathtt{tp}) &= \lambda U^p\colon \mathtt{tp}.\ \mathtt{tm}\ U^p \to \mathtt{prop}\\
\overline{s}(\mathtt{tm}) &= \lambda T^p\colon \mathtt{tp}.\ \lambda T^s\colon \mathtt{tm}\ T^p \to \mathtt{prop}.\ \lambda t^p\colon \mathtt{tm}\ T^p.\ \Vdash T^s\ t^p\\
\overline{s}(\mathtt{prop}) &= \lambda F^p\colon \mathtt{prop}.\ \mathtt{prop}\\
\overline{s}(\Vdash) &= \lambda F^p\ F^s\ \mathit{pf}.\ \Vdash F^s
\end{aligned}$$

and inductive cases as inspired from Figure 27c, e.g., with

$$\begin{aligned}
\overline{s}(\forall) &= \lambda T^p\ T^s\ p^p\ p^s.\ \forall\, x^p\colon \mathtt{tm}\ T^p.\quad \mathtt{depImpl}\ (T^s\ x^p)\ (\lambda x^s.\ p^s\ x^p\ x^s)\\
\overline{s}(\exists) &= \lambda T^p\ T^s\ p^p\ p^s.\ \exists\, x^p\colon \mathtt{tm}\ T^p.\quad \mathtt{depConj}\ (T^s\ x^p)\ (\lambda x^s.\ p^s\ x^p\ x^s)
\end{aligned}$$

5.4.4 Examples

▶ **Example 129** (**`Sub(Lists)` formalizes lists where all elements satisfy a predicate**)**.** Applying `Sub` on the theory `Lists` from Example 101 gives the theory `Sub(Lists)` below.

theory $\mathtt{Sub(Lists)} = \{$

$$\begin{aligned}
&\mathtt{list}^p\colon \mathtt{tp} \to \mathtt{tp}\\
&\mathtt{nil}^p\ \ \colon \Pi\, A^p\colon \mathrm{tp}.\ \mathtt{list}^p\ A^p\\
&\mathtt{cons}^p\colon \Pi\, A^p\colon \mathrm{tp}.\ \mathtt{tm}\ A^p \to \mathtt{tm}\ \mathtt{list}^p\ A^p \to \mathtt{tm}\ \mathtt{list}^p\ A^p\\
&\mathtt{list}^s\colon \Pi\, A^p\colon \mathrm{tp}.\ \Pi\, A^s\colon \mathrm{tm}\ A^p \to \mathrm{prop}.\ \mathtt{tm}\ \mathtt{list}^p\ A^p \to \mathtt{prop}\\
&\mathtt{nil}^s\ \ \colon \Pi\, A^p\colon \mathrm{tp}.\ \Pi\, A^s\colon \mathrm{tm}\ A^p \to \mathrm{prop}.\ \Vdash \mathtt{list}^s\ A^p\ A^s\ \mathtt{nil}^p\\
&\mathtt{cons}^s\colon \Pi\, A^p\colon \mathrm{tp}.\ \Pi\, A^s\colon \mathrm{tm}\ A^p \to \mathrm{prop}.\\
&\qquad \Pi\, x^p\colon \mathtt{tm}\ A^p.\ \Pi\, x^s\colon\ \Vdash A^s\ x^p.\ \Pi\, l^p\colon \mathtt{tm}\ \mathtt{list}^p\ A^p.\ \Pi\, l^s\colon\ \Vdash \mathtt{list}^s\ A^p\, A^s\, l^p.\\
&\qquad \Vdash \mathtt{list}^s\ A^p\ A^s\ (\mathtt{cons}^p\ A^p\ x^p\ l^p)
\end{aligned}$$

$\}$

For any given type $A\colon \mathtt{tp}$ and predicate $p\colon \mathtt{tm}\ A \to \mathtt{prop}$, we can think of the pair of $\mathtt{list}^p\ A$ and $\mathtt{list}^s\ A\ A^p$ (a type and a predicate on terms thereof) as the inductive datatype of lists

[18] e.g., we would allow "dependent connective artifacts" stemming from the way the translation is defined, for instance, we would say a dependent implication where the consequence does not even use the antecedence's witness is reasonably agrees with an ordinary implication

$$
\begin{array}{lll}
r(\mathtt{tp}) & : & \bot \\
r(\mathtt{prop}) & : & \lambda F^p\ F^s.\ \lambda pf\colon\ \Vdash F^p.\ \Vdash F^p \Leftrightarrow F^s \\
r(\mathtt{tm}) & : & \bot \\
r(\Vdash) & : & \bot
\end{array}
$$

$$
\begin{array}{ll}
f(\forall) & :\ \Pi\, T^p\ T^s.\ \Pi\, p^p\ p^s : \mathtt{tm}\ T^p \to \mathtt{prop}.\ \Pi\, p^r : \Pi\, x^p\ x^s.\ \Vdash p^p\ x^p \Leftrightarrow p^s\ x^p. \\
 & \quad \Vdash (\forall\, x^p.\ p^p\ x^p) \Leftrightarrow (\forall\, x^p.\ T^s\ x^p \Rightarrow p^s\ x^p) \\
 & = \lambda T^p\ T^s\ p^p\ p^s.\ \Leftrightarrow \mathtt{I}\ (\lambda pf\colon\ \Vdash \forall\, x^p.\ p^p\ x^p.\ \forall\mathtt{I}_{x^p} \Rightarrow \mathtt{I}_{x^s}\ (p^r\ x^p\ x^s)\ \Leftrightarrow \overset{\rightarrow}{\mathtt{E}}\ (pf\ \forall\mathtt{E}\ x^p)) \\
 & \qquad\qquad\qquad (\lambda pf\colon\ \Vdash \forall\, x^p.\ T^s\ x^p \Rightarrow p^s\ x^p.\ \forall\mathtt{I}_{x^p}\ \ldots) \\
f(\doteq) & :\ \Pi\, T^p\ T^s\ x_1^p\ x_1^s\ x_2^p\ x_2^s.\ \Vdash (x_1^p \doteq x_2^p) \Leftrightarrow (x_1^p \doteq x_2^p) \\
 & = \lambda T^p\ T^s\ x_1^p\ x_1^s\ x_2^p\ x_2^s.\ \Leftrightarrow \mathtt{I}\ (\lambda x.\ x)\ (\lambda x.\ x) \\
f(\neg F) & :\ \Pi\, F^p\ F^s\ F^r.\ \Vdash \neg F^p \Leftrightarrow \neg F^s \\
 & =\lambda F^p\ F^s\ F^r.\ \Leftrightarrow \mathtt{I}\ (\lambda pf\colon\ \Vdash \neg F^p.\ \neg\mathtt{I}_{\tilde{pf}\colon F^s}\ (F^r\ \Leftrightarrow \overset{\leftarrow}{\mathtt{E}}\ \tilde{pf})\ \neg\mathtt{E}\ pf) \\
 & \qquad\qquad\qquad (\lambda pf\colon\ \Vdash \neg F^s.\ \neg\mathtt{I}_{\tilde{pf}\colon F^p}\ (F^r\ \Leftrightarrow \overset{\rightarrow}{\mathtt{E}}\ \tilde{pf})\ \neg\mathtt{E}\ pf) \\
f(\wedge) & :\ \Pi\, F^p\ F^s\ F^r\ G^p\ G^s\ G^r.\ \Vdash (F^p \wedge G^p) \Leftrightarrow (F^s \wedge G^s) \\
 & = \lambda F^p\ F^s\ F^r\ G^p\ G^s\ G^r. \\
 & \qquad \Leftrightarrow \mathtt{I}\ (\lambda pf\colon\ \Vdash F^p \wedge G^p.\ \wedge\mathtt{I}\ (F^r\ \Leftrightarrow \overset{\rightarrow}{\mathtt{E}}\ (pf \wedge\mathtt{EL}))\ (G^r\ \Leftrightarrow \overset{\rightarrow}{\mathtt{E}}\ (pf \wedge\mathtt{ER}))) \\
 & \qquad\qquad (\lambda pf\colon\ \Vdash F^s \wedge G^s.\ \wedge\mathtt{I}\ (F^r\ \Leftrightarrow \overset{\leftarrow}{\mathtt{E}}\ (pf \wedge\mathtt{EL}))\ (G^r\ \Leftrightarrow \overset{\leftarrow}{\mathtt{E}}\ (pf \wedge\mathtt{ER}))) \\
f(\vee) & :\ \Pi\, F^p\ F^s\ F^r\ G^p\ G^s\ G^r.\ \Vdash (F^p \vee G^p) \Leftrightarrow (F^s \vee G^s) \\
 & = \lambda F^p\ F^s\ F^r\ G^p\ G^s\ G^r.\ \mathit{TODO} \\
f(F \Rightarrow G) & :\ \Pi\, F^p\ F^s\ F^r\ G^p\ G^s\ G^r.\ \Vdash (F^p \Rightarrow G^p) \Leftrightarrow (F^s \Rightarrow G^s) \\
 & = \lambda F^p\ F^s\ F^r\ G^p\ G^s\ G^r.\ \mathit{TODO}
\end{array}
$$

Unless otherwise noted, for readability we omit the typings $F^p, F^s, G^p, G^s : \mathtt{prop}$, $F^r : \Vdash F^p \Leftrightarrow F^s$, $G^r : \Vdash G^p \Leftrightarrow G^s$, $T^p : \mathtt{tp}$, $T^s : \mathtt{tm}\ T^p \to \mathtt{prop}$, $x_1^p, x_2^p : \mathtt{tm}\ T^p$, $x_1^s : \Vdash T^s\ x_1^p$, $x_2^s : \Vdash T^s\ x_2^p$.

where all elements satisfy p. In particular, $\texttt{list}^s\ A\ p$ is the predicate p lifted on lists such that (in context of the above theory) $\texttt{list}^s\ A\ p\ l$ is provable iff all elements of l provably satisfy p.

The inductive datatype mentioned above is featured in various placed in the Agda standard library under the name *All* [AgdaAll21].

5.4.5 Meta-Theoretical Properties

The following theorem confirms the intuition laid down in Figure 26:

▶ **Proposition 130** (Basic Lemma for s). *We have*

- *if* $\Gamma \vdash_{\texttt{SFOL}} t : A$ *and* s *is defined for* t*, then* s *is defined for* A *and* $s(\Gamma) \vdash_{\texttt{SFOL}} s(t) : s(A)\ p(t)$
- *if* s *is term-total, it is defined for a typed term if it is for its type*

Proof. In principle, the proof proceeds similarly to the one of Theorem 11 for actual logical relations. We already made the differences plausible right after Definition 126. ◀

▶ **Theorem 131** (Substructures are Closed under Complex Terms). *Let* T *be a theory* T *and* $\Sigma \vdash^{\texttt{SFOL}}_T t : \texttt{tm}\ a$ *an arbitrary, possibly complex term. Then we have*

$$\texttt{Sub}(\Sigma) \vdash^{\texttt{SFOL}}_{\texttt{Sub}(T)} \Vdash s(a)\ p(t)$$

Consider the special case where T *is an SFOL-theory and* $\Sigma = \{x_1 : \texttt{tm}\ a_1, \dots, x_n : \texttt{tm}\ a_n\}$ *exactly contains the free term variables occurring in* t*. This forces all types* $a_1, \dots, a_n, a$ *to be atomic, i.e., references to type symbols declared in* T*. The statement above now reduces to: if all* $x_1, \dots, x_n$ *are contained in the submodel, then so is* t*:*

$$\left\{\begin{matrix} x_1^p : \texttt{tm}\ a_1^p, & x_1^s : \Vdash a_1^s\ x_1^p \\ \vdots & \vdots \\ x_n^p : \texttt{tm}\ a_n^p, & x_n^s : \Vdash a_n^s\ x_n^p \end{matrix}\right\} \vdash^{\texttt{SFOL}}_{\texttt{Sub}(T)} \Vdash a^s\ p(t)$$

where, in this case, $p(t)$ *emerges from* t *by superscripting all references to function symbols and variables with* p*.*

Proof. Direct consequence of Proposition 130. ◀

▶ **Theorem 132.** $\texttt{Sub}^p$ *is well-typed and natural.*

Proof. By Theorem 38 in conjunction with the next theorem. ◀

▶ **Theorem 133.** $\texttt{Sub}$ *is well-typed.*

Proof. Direct consequence of Proposition 130. ◀

▶ **Conjecture 134.** *Under mild assumptions (e.g., of additional equational theory in* SFOL*),* Sub *is functorial.*

$$
\begin{array}{l}
\textbf{theory}\ \texttt{Cong}(\texttt{Unital}) = \{ \\
\quad \textbf{include}\ \texttt{SFOL} \\
\quad \begin{array}{ll}
U^d & : \texttt{tp} \\
U^g & : \texttt{tm}\ U^d \to \texttt{tm}\ U^d \to \texttt{prop} \\
\circ^d & : \texttt{tm}\ U^d \to \texttt{tm}\ U^d \to \texttt{tm}\ U^d \\
\circ^g & : \Pi\, x^d\ x^{d'} : \texttt{tm}\ U^d.\ \Pi\, x^g : \ \Vdash U^g\ x^d\ x^{d'}. \\
 & \quad \Pi\, y^d\ y^{d'} : \texttt{tm}\ U^d.\ \Pi\, y^g : \ \Vdash U^g\ y^d\ y^{d'}. \\
 & \quad \Vdash U^g\ (x^d \circ^d y^d)\ (x^{d'} \circ^d y^{d'}) \\
e^d & : \texttt{tm}\ U^d \\
e^g & : \Vdash U^g\ e^d\ e^d
\end{array} \\
\quad \texttt{neut}^d : \Vdash \forall\, x^d : \texttt{tm}\ U^d.\ e^d \circ x^d \doteq x^d \\
\quad \texttt{neut}^g : \Vdash \forall\, x^d : \texttt{tm}\ U^d.\ U^g\ (e^d \circ x^d)\ x^d \\
\} \\
\\
\textbf{mor}\ \texttt{Cong}^d(\texttt{Unital}) : \texttt{Unital} \to \texttt{Cong}(\texttt{Unital}) = \{ \\
\quad == \\
\}
\end{array}
$$

Figure 28 Theory of unital congruence structures given by $\texttt{Cong}(\texttt{Unital})$ together with domain projection

5.5 Congruences

$$T \xrightarrow{d} \texttt{Cong}(T)$$

The linear functor $\texttt{Cong}(-)$ maps every SFOL-theory T to the $\texttt{SFOL}$-extension $\texttt{Cong}(T)$ of T-congruences, whose models are congruence models of T-models. For example, applied to the theory `Group` it yields the theory $\texttt{Cong}(\texttt{Group})$ of group congruences. (Group congruences are an equivalent characterization of normal subgroups. Even though most group theorists use normal subgroups rather than congruences, we settle with formalizing congruences as that is the more general, more syntactical notion, which is also used by universal algebraists.) The linear connector p into `Cong` is the projection of the very domain T-model.

5.5.1 `Cong` on Definitionless `SFOL`

Before defining `Cong`, let us first look at an example of what we would expect the theory $\texttt{Cong}(\texttt{Unital})$ of unital congruences to look like (using `Unital` from Example 97):

▶ **Example 135** ($\texttt{Cong}(\texttt{Unital})$)**.** For the theory `Unital` from Example 97 we obtain the theory $\texttt{Cong}(\texttt{Unital})$ and the morphism $\texttt{Cong}^d(\texttt{Unital}) : \texttt{Unital} \to \texttt{Cong}(\texttt{Unital})$ shown in Figure 28. We see that the constants $U^d, \circ^d, e^d, \texttt{neut}^d$ collectively make up a unital structure, and this is precisely what is witnessed by the morphism. Within $\texttt{Cong}(\texttt{Unital})$, additional structure is imposed on these constants by means of the g-superscripted constants.

Importantly, U^g represents a binary relation on terms of type U^d. Perhaps suprisingly, we do not require this relation to be an equivalence (i.e., symmetric, reflexive, and transitive).[19] Our decision is guided by the connector `QuotMod` which we envision as part of future work (see also Section 5.8): it maps every theory T to the morphism $\texttt{QuotMod}(T) \colon T \to \texttt{Cong}(T)$ that translates every congruence model to an actual T-model "of equivalence classes" using quotient types. And we envision using the quotient types of the LATIN2 project [Rab21] which for every type $T \colon \texttt{tp}$ and binary relation $R \colon \texttt{tm}\ T \to \texttt{tm}\ T \to \texttt{prop}$ already yield a quotient type T/R. This variant of quotient types that puts no restriction on the relation has the advantage of being dual to predicate subtypes in a certain sense [Rab21, Sec. 2.3] wrt. formation, introduction, and elimination rules, and more. Therefore, we will define `Cong` in alignment with those quotient types (even though we never introduce them in this thesis for conciseness).

The axiom symbols $\circ^g$ and e^g force the relation U^g to be closed under the corresponding operations. In mathematical folklore, the constant $\circ^g$ is often put in words as requiring "$\circ^d$ to be well-defined wrt. the binary relation U^g" (except that in our setting U^g needs not be an equivalence). Interestingly, in our setting well-definedness extends to nullary function symbols too, and for $e \colon \texttt{tm}\ U$, the generated axiom e^g amounts to requiring reflexivity of the relation U^g. Finally, the axiom symbol $\texttt{neut}^g$ is the variant of $\texttt{neut}^d$ that relativizes equality at type U^d by U^g. This axiom will be essential for `QuotMod` when assigning a proof to $\texttt{neut} \in \texttt{Unital}$ in the morphism $\texttt{QuotMod}(\texttt{Unital}) \colon \texttt{Unital} \to \texttt{Cong}(\texttt{Unital})$: it guarantees that the "equivalence class" for $e^d \colon \texttt{tm}\ U^d$ is neutral wrt. all other "equivalence classes".

▶ **Definition 136** (Congruence Operators). *The linear functor* `Cong` *from definitionless SFOL-theories to all possible* `SFOL`*-extensions is given by:*

- *type symbols $T \colon \texttt{tp}$ are mapped to a copy and a binary predicate on terms thereof:*

$$\begin{aligned} &T^d \colon \texttt{tp} \\ &T^g \colon \texttt{tm}\ T^p \to \texttt{tm}\ T^p \to \texttt{prop} \end{aligned}$$

- *function symbols $f \colon \texttt{tm}\ T_1 \to \ldots \to \texttt{tm}\ T_n \to \texttt{tm}\ T$ are mapped to a copy and a closure axiom:*

$$\begin{aligned} &f^d \colon \texttt{tm}\ T_1^d \to \ldots \to \texttt{tm}\ T_n^d \to \texttt{tm}\ T^d \\ &f^g \colon \Pi\, t_1^d \colon \texttt{tm}\ T_1^d.\ \Pi\, t_1^{d'} \colon \texttt{tm}\ T_1^d.\ \Pi\, t_1^g \colon\ \Vdash T_1^g\ t_1^d\ t_1^{d'}. \\ &\qquad \vdots \\ &\qquad \Pi\, t_n^d \colon \texttt{tm}\ T_n^d.\ \Pi\, t_n^{d'} \colon \texttt{tm}\ T_n^d.\ \Pi\, t_n^g \colon\ \Vdash T_n^g\ t_n^d\ t_n^{d'}.\ \Vdash T^g\ (f^d\ t_1^d\ \ldots\ t_n^d)\ (f^d\ t_1^{d'}\ \ldots\ t_n^{d'}) \end{aligned}$$

- *predicate symbols $p \colon \texttt{tm}\ T_1 \to \ldots \to \texttt{tm}\ T_n \to \texttt{prop}$ are mapped to a copy and a preservation axiom:*

$$\begin{aligned} &p^d \colon \texttt{tm}\ T_1^p \to \ldots \to \texttt{tm}\ T_n^p \to \texttt{prop} \\ &p^g \colon \Pi\, t_1^d \colon \texttt{tm}\ T_1^d.\ \Pi\, t_1^{d'} \colon \texttt{tm}\ T_1^d.\ \Pi\, t_1^g \colon\ \Vdash T_1^g\ t_1^d\ t_1^{d'}. \\ &\qquad \vdots \\ &\qquad \Pi\, t_n^d \colon \texttt{tm}\ T_n^d.\ \Pi\, t_n^{d'} \colon \texttt{tm}\ T_n^d.\ \Pi\, t_n^g \colon\ \Vdash T_n^g\ t_n^d\ t_n^{d'}.\ \Vdash p^d\ t_1^d\ \ldots\ t_n^d \Leftrightarrow p^d\ t_1^{d'}\ \ldots\ t_n^{d'} \end{aligned}$$

[19] Although it will be reflexive by means of the axioms that we generate for function symbols, see Proposition 140.

- *axiom symbols ax: $\Vdash F$ are mapped to a copy and a relativized variant:*

  $$ax^d : \Vdash F^d$$
  $$ax^g : \Vdash F^g$$

  where F^g emerges from F^d by replacing every equality $\phi \doteq \psi$ at type T by $T^g\ \phi\ \psi$

And the linear connector $\texttt{Cong}^p$ *into* `Cong` *maps every constant c to $c := c^d$.*

▶ **Example 137** (`Cong` Applied To Graphs)**.** Consider the following formalization of directed graphs as an SFOL-theory:

$$\begin{array}{l}
\textbf{theory}\ \texttt{Graph} = \{\\
\quad \textbf{include}\ \texttt{SFOL}\\
\quad V \quad\ : \texttt{tp}\\
\quad \texttt{edge} : \texttt{tm}\ V \to \texttt{tm}\ V \to \texttt{prop}\\
\}
\end{array}$$

Applying `Cong` yields the theory below, where we manually added some notation (indicated after #) to ease readability.

$$\begin{array}{l}
\textbf{theory}\ \texttt{Cong}(\texttt{Graph}) = \{\\
\quad \textbf{include}\ \texttt{SFOL}\\
\quad V^d \quad\ : \texttt{tp}\\
\quad V^g \quad\ : \texttt{tm}\ V^d \to \texttt{tm}\ V^d \to \texttt{prop}\ \#\ 1 \sim 2\\
\quad \texttt{edge}^d : \texttt{tm}\ V^d \to \texttt{tm}\ V^d \to \texttt{prop}\ \#\ 1 \twoheadrightarrow 2\\
\quad \texttt{edge}^g : \Pi\, u\ u'.\ \Pi\, u^g :\ \Vdash u \sim u'.\\
\qquad\qquad\quad \Pi\, v\ v'.\ \Pi\, v^g :\ \Vdash v \sim v'.\\
\qquad\qquad\qquad \Vdash u \twoheadrightarrow v \Leftrightarrow u' \twoheadrightarrow v'\\
\}
\end{array}$$

Note that actually this is not what is commonly usually understood as a congruce (quotient) graph, e.g., see https://en.wikipedia.org/w/index.php?title=Quotient_graph&oldid=1033184360.

Meta-Theoretical Properties

▶ **Theorem 138.** $\texttt{Cong}^d$ *is well-typed and natural.*

Proof. By Theorem 38 in conjunction with the next theorem. ◀

▶ **Theorem 139.** `Cong` *is well-typed and functorial.*

Proof. Easy to see by Definition 136. ◀

Note that naturality and functoriality hold vacuously since `Cong` is partial on all defined constants anyway, i.e., in particular morphism assignments.

Even though we do not require the binary relations T^g to be equivalence, they become reflexive by means of the axioms that we generate for function symbols:

▶ **Proposition 140.** *For all SFOL-theories S, type symbols $T\colon \texttt{tp}$ in S, and arbitrarily complex terms $\vdash_S t\colon \texttt{tm}\ T$, there is a witness for the type*

$$\vdash_{\texttt{Sub}(S)} \Vdash T^g\ t^d\ t^d$$

where t^d is the systematic renaming of t with d superscripts in all function symbols.

Proof. Since S is an SFOL-theory and t a closed term, it must follow the grammar

$$t \quad ::= \quad f\ t_1\ \dots\ t_n$$

where f ranges over all function symbols of S. We induct on this grammar (and explicitly separate cases for didactic reasons):

- $t = f$ and $f\colon \texttt{tm}\ T$ is a nullary function symbol in S: by construction the theory $\texttt{Sub}(S)$ then contains the axiom symbol $f^g\colon\ \Vdash T^g\ f\ f$ completing the proof.
- $t = f\ t_1\ \dots\ t_n$ and $f\colon \texttt{tm}\ T_1 \to \dots \to \texttt{tm}\ T_n \to \texttt{tm}\ T$ is an $n \geq 1$-ary function symbol in S: by induction hypotheses there are proofs $pf_1, \dots, pf_n$ such that $\vdash_{\texttt{Sub}(S)} pf_i\colon\ \Vdash T_i^g\ t_i\ t_i$. Moreover, $\texttt{Sub}(S)$ contains the axiom symbol f^g, which we use to form the desired witness as
$$\vdash_{\texttt{Sub}(S)} f^g\ t_1\ t_1\ pf_1\ \dots\ t_n\ t_n\ pf_n\colon\ \Vdash T^g\ (f\ t_1\ \dots\ t_n)\ (f\ t_1\ \dots\ t_n)$$

◀

▶ **Proposition 141.** *Let S be an SFOL-theory. For propositions $\vdash_S F\colon \texttt{prop}$ we denote by $\vdash_{\texttt{Sub}(S)} F^g\colon \texttt{prop}$ the systematically renamed (via p-superscripts) and relativized proposition as defined above in the case for axiom symbols.*

If $\vdash_S pf\colon \Vdash F$ is a proof of F without usage of `symm` *and* `trans`*, then there is a proof $\vdash_{\texttt{Sub}(S)} pf^g\colon \Vdash F^g$.*

Proof. We induct on derivations for $\vdash_S pf\colon \Vdash F$:

- case $pf = ax$: set $pf^g = ax^g$ to the axiom symbol $ax^g\colon F^g$ which $\texttt{Sub}(S)$ contains by construction
- case $pf = \texttt{refl}$: follows by Proposition 140
- cases $pf = \texttt{symm}$ or $pf = \texttt{trans}$: excluded by assumption
- cases where pf is the application of any other proof rule: Let us exemplarily consider forall elimination, where $pf = \tilde{pf}\ \forall\texttt{E}\ t$, $F = \Vdash p\ t$, and $\vdash_S \tilde{pf}\colon \forall\, p$. By induction hypothesis there is $\vdash_{\texttt{Sub}(S)} \tilde{pf}^g\colon\ \Vdash \forall\, p^g$ allowing us to set $pf^g = \tilde{pf}^g\ \ \forall\texttt{E}\ t^p$, where t^p is the systematically renamed copy of t.

◀

The logical relation approach below generalizes this idea.

5.5.2 Thoughts on a Generalized Definition

We would like to extend `Cong` to a linear functor on all SFOL-theories (i.e., including those with definitions and morphisms) and PFOL, DFOL or even PDFOL-theories if possible. In Sections 5.3 and 5.4, we used logical relations as a guiding approach to lift ad-hoc definitions of `Hom` and `Sub` to complex, yet systematic specifications. Already there we had to carefully modify inductive cases of logical relations to suit our needs. We were lucky that we got by with few systematic modifications, which made it possible to reuse meta theorems from the theory of logical relations. We suspected the same to be true for `Cong`, but to our surprise

$\Sigma \supseteq$ `SFOL`-expression		mapped to `Cong`(Σ)-expression	
types	$T : \texttt{tp}$	$g(T) : \texttt{tm}\ T^d \to \texttt{tm}\ t^{d'} T \to \texttt{prop}$	binary relation
terms	$t\ : \texttt{tm}\ T$	$g(t)\ : \Vdash T^g\ t^d\ t^{d'}$	reflexivity
propositions	$F : \texttt{prop}$	$g(F) : \Vdash F^d \Leftrightarrow F^{d'}$	reflection & preservation
proofs	$pf : \Vdash F$	$g(pf) : \Vdash F^{rel}$	relativization

Figure 29 Intuitive overview of the translation to be carried out by $\overline{g}$

it seems that `Cong` would necessitate fundamentally more modifications (beyond the scope of this thesis). Therefore, we have opted to solely describe some of the difficulties for future work to pick up on.

Judging from Definition 136, Example 135, and in particular Proposition 140, we would naively try a binary logical relation on $\texttt{Cong}^d$ and $\texttt{Cong}^d$ implementing the translation sketched in Figure 29. There, the superscripts $-^d$ and $-^{d'}$ indicate the two models (model in a lose sense) on which the binary logical relation is operating. If we succeeded in defining a logical relation g that way, we could define the linear functor `Cong` from PDFOL-theories to all possible `SFOL`-extensions and its connector $\texttt{Cong}^d$ as usual by

$$\begin{aligned} \texttt{Cong}^{\Sigma}(c\colon A\,[= t]) &= c^d\colon A^d\,[= t^d],\ c^g\colon g(A)\ c^d\ c^d\,[= r(t)] \\ \texttt{Cong}^{d,\Sigma}(c\colon A) &= c := c^d \end{aligned}$$

Here, we instantiate the relation $g(A)$ at LF type A with one and the same model c^d. Unfortunately, there are two fundamental problems with this approach, for which we have not found a promising and systematic fix.

First, the desired action on propositions (Figure 29, line 3) is problematic because it makes it impossible to define g on the universal and existential quantifier. For example, the expected LF type for the assignment to $\forall$ is

$$\begin{aligned} g(\forall)\colon \Pi\, T^d\ T^{d'}\colon \texttt{tp}.\ \Pi\, T^g\colon \texttt{tm}\ T^d \to T^{d'} \to \texttt{prop}. \\ \Pi\, p^d\colon \texttt{tm}\ T^d \to \texttt{prop}.\ \Pi\, p^{d'}\colon \texttt{tm}\ T^{d'} \to \texttt{prop}. \\ \Vdash (\forall p^d) \Leftrightarrow (\forall p^{d'}) \end{aligned}$$

This LF type is visibly empty due to the disparity of types over which is quantified (T^d and $T^{d'}$), i.e., there is simply no well-typed assignment to $g(\forall)$.

Second, it is impossible to encode lines 3 and 4 of Figure 29 at the same time in a single logical relation. In order to perform the intended relativization of propositions as part of the logical relation (analogous to how we did with `Sub`; see Figure 26 and Figure 26), we would need to map propositions $F\colon \texttt{prop}$ again to propositions $g(F)\colon \texttt{prop}$ such that the assignment to $g(\doteq)$ can account for relativizing equalities. But this clashes with line 3 of Figure 29, which is crucial to achieve the desired action of `Cong` on predicate symbols. Note that `Sub` maps predicate symbol p to just a qualified copy p^p for the parent structure. Thus, in the logical relation-inspired approach employed for `Sub`, there is no need to implement any logic to produce the LF type of any other additional constant, say p^s. Therefore, there is "enough room" to encode `Sub`'s relativization of propositions. In contrast, for `Cong` we need to map predicate symbols to both a qualified copy p^d and a fundamentally new constant p^g, *and* we need to implement relativization of propositions. Possibly, it is ill-guided to try implementing both actions in one and the same logical relation. And possibly it only worked for `Sub` as that was a special case, but it fails generalizing to other settings, e.g., to `Cong`.

$$\textbf{mor } \texttt{Img}(\texttt{Unital})\colon \texttt{Sub}(\texttt{Unital}) \to \texttt{Hom}(\texttt{Unital}) = \{$$
$$: :$$
$$\}$$

Unless otherwise noted, for readability we omit the typings $x, x_1, x_2\colon \texttt{tm}\ U^d,\ y, y_1, y_2\colon \texttt{tm}\ U^c$.

Figure 30 Image of a unital homomorphism given by $\texttt{Img}(\texttt{Unital})$

5.6 Images of Homomorphisms

$$\texttt{Sub}(T) \xrightarrow{\texttt{Img}(T)} \texttt{Hom}(T)$$

The linear connector $\texttt{Img}(-)$ yields for any SFOL-theory T the morphism $\texttt{Img}(T)\colon \texttt{Sub}(T) \to \texttt{Hom}(T)$ that translates every $\texttt{Hom}(T)$-model representing a homomorphism to the $\texttt{Sub}(T)$-model representing the homomorphism's image. This submodel inherits exactly the operations of the homomorphism's codomain structure. For example, applied to the theory `Group` it yields the morphism $\texttt{Img}(\texttt{Group})\colon \texttt{Sub}(\texttt{Group}) \to \texttt{Hom}(\texttt{Group})$ formalizing precisely the notion that the image of every group homomorphism induces a subgroup on the codomain.

The connector `Img` becomes particularly useful in combination with another one (which we left out in this thesis for space reasons): consider the connector $\texttt{SubMod}(T)\colon T \to \texttt{Sub}(T)$ that translates every $\texttt{Sub}(T)$-model representing a submodel to an actual T-model ("sub model"). For example, $\texttt{SubMod}(\texttt{Group})\colon \texttt{Group} \to \texttt{Sub}(\texttt{Group})$ captures that every subgroup is again a group.

$$T \xrightarrow{\texttt{SubMod}(T)} \texttt{Sub}(T) \xrightarrow{\texttt{Img}(T)} \texttt{Hom}(T)$$

Composing these connectors as above gives a connector that, e.g., applied to `Group` yields a morphism $T \to \texttt{Hom}(T)$ that translates every group homomorphism to the group of its image.

5.6.1 `Img` on Definitionless `SFOL`

Before defining `Img` and spelling out the details, let us first look at an example of what we would expect the morphism $\texttt{Img}(\texttt{Unital})\colon \texttt{Sub}(\texttt{Unital}) \to \texttt{Hom}(\texttt{Unital})$ to look like (using `Unital` from Example 97):

▶ **Example 142** ($\texttt{Img}(\texttt{Unital})$)**.** In Figure 30 we show the connecting morphism $\texttt{Img}(\texttt{Unital})$ that we desire. For readability, we annotated expected, possibly α-renamed types.

Before perusing the assignments, recall $\texttt{Sub}(\texttt{Unital})$ and $\texttt{Hom}(\texttt{Unital})$ from Figures 25 and 22, respectively. In particular, recall that $\texttt{Sub}(\texttt{Unital})$ encodes a `Unital` structure for the parent structure via the systematically renamed p-superscripted constants $c^p\colon A^p$ for every constant $c\colon A$ in `Unital`. And $\texttt{Hom}(\texttt{Unital})$ encodes two `Unital` structures for the domain and codomain structures via the d- and c-superscripted constants $c^d\colon A^d$ and $c^c\colon A^c$ for every constant $c\colon A$ in `Unital`.

We want $\texttt{Img}(\texttt{Unital})$ to realize the homomorphism's image as a `Unital` substructure, namely on the homomorphism's codomain t structure. Thus we systematically have assignments $c^p := c^c$ for all c in `Unital`. The remaining assignments serve to actually select a substructure and to fulfill proof obligations for closure properties.

First, in the assignment to U^s we choose the substructure to be composed of those elements from the homomorphism's codomain that possess a preimage.

Next, in the assignment to $\circ^s$ we prove that those elements are closed under applying $\circ^c$. Concretely, given two elements y_1 and y_2 that lie within the image of the homomorphism – as witnessed by y_1^s and y_2^s – we construct a witness of $y_1 \circ^c y_2$ also lying within the image. Intuitively, we take three steps: *i*) we get hold of some corresponding preimages x_1 and x_2, *ii*) we compute $x_1 \circ^d x_2$, and *iii*) prove that $x_1 \circ^d x_2$ is in fact a preimage of $y_1 \circ^c y_2$ (using the homomorphism property) In reality, while the first two steps are easy to formalize (see Figure 30), things get a bit awkward for the third step. Here, we would need to rewrite some equalities using `SFOL`'s congruence axiom `cong`, which would produce quite an unreadable LF term. Thus, for readability we opted to give a proof outline in natural language. And in the assignment to e^s we prove that the homomorphism's image contains the neutral element.

Finally, in the assignment to $\texttt{neut}^s$ we essentially repeat the assignment $\texttt{neut}^p = \texttt{neut}^c$ cluttered with trivial introduction and elimination rules. This seeming repetition goes back to according to which axioms like $\texttt{neut}^s$ are provable in terms of $\texttt{neut}^p$ anyway if they.

We now specify `Img` on definitionless SFOL-theories:

▶ **Definition 143** (Homomorphism Images). *The linear connector* `Img` *from* `Sub` *to* `Hom` *on SFOL-theories is given by:*

- *type symbols* $T\colon \texttt{tp}$ *are mapped to*

$$T^p := T^c$$
$$T^s := \lambda y\colon \texttt{tm}\ T^c.\ \exists x\colon \texttt{tm}\ T^d.\ T^h\ x \doteq y$$

- *function symbols* $f\colon \texttt{tm}\ T_1 \to \ldots \to \texttt{tm}\ T_n \to \texttt{tm}\ T$ *are mapped to*

$$\begin{aligned} f^p &:= f^c \\ f^s &:= \lambda y_1\colon \texttt{tm}\ T_1^c.\ \lambda y_1^s\colon (\Vdash \exists x_1\colon \texttt{tm}\ T_1^d.\ T_1^h\ x_1 \doteq y_1). \\ &\quad\ \vdots \\ &\quad\ \lambda y_n\colon \texttt{tm}\ T_n^c.\ \lambda y_n^s\colon (\Vdash \exists x_n\colon \texttt{tm}\ T_n^d.\ T_n^h\ x_n \doteq y_n). \\ &\quad\quad y_1^s\ \exists\mathrm{E}_{x_1,x_1^h}\ \ldots\ y_n^s\ \exists\mathrm{E}_{x_n,x_n^h} \\ &\quad\quad \exists\texttt{I}\ (f^d\ x_1\ \ldots\ x_n)\ pf \end{aligned}$$

where pf is the LF term representing the `SFOL`*-proof*

$$\left\{ \begin{array}{ll} \Vdash U^h\ (f^d\ x_1\ \ldots\ x_n) \doteq f^c\ (T_i^h\ x_1)\ \ldots\ (T_i^h\ x_n) & \textit{by } f^h \\ \Vdash T_i^h\ x_i \doteq y_i & \textit{by } x_i^h \textit{ for all } i \\ \Vdash T^h\ (f^d\ x_1\ \ldots\ x_n) \doteq f^c\ y_1\ \ldots\ y_n & \textit{by rewriting with prev. eqns.} \end{array} \right\}$$

- *predicate symbols* $p\colon \texttt{tm}\ T_1 \to \ldots \to \texttt{tm}\ T_n \to \texttt{prop}$ *are mapped to*

$$p^p := p^c$$

- *axiom symbols* $ax\colon \Vdash F$ *are mapped to*

$$ax^p := ax^c$$

In particular, we generously assume the variant of `Sub` *that supplies definitions for certain axiom symbols* ax^s *(see) and leave* `Img` *undefined otherwise.*

Again, we invite the reader to expand the natural language proof given in the assignment to f^s to an LF term using `SFOL`'s `cong` axiom.

Note that as a connector on definitionless `SFOL`-theories, `Img` is maximally defined in the sense that there are axiom symbols with $\neg$ and $\exists$, respectively, on which it is *impossible* to define `Img`. We give corresponding examples below. In fact, we conjecture that the axiom symbols ax^s for which `Sub` can (cannot) emit a definition are precisely those for which `Img` can (cannot) emit an assignment. Thus, we never emit an assignment to ax^s, shifting all work onto `Sub` (this also makes sense from a software engeering perspective when implementing said functors).

▶ **Example 144** (`Img` **Not Fully Extensible to** $\neg, \forall$)**.** Consider the theory below which extends `Unital` from Example 97 with an axiom symbol that forces unital structures to be non-trivial, i.e., to consist of more than just the neutral element.

$$\begin{array}{l}\textbf{theory}\ \texttt{NonTrivialUnital} = \{\\ \quad \textbf{include}\ \texttt{Unital}\\ \quad \texttt{nottriv}\colon \neg\forall\, x\colon \texttt{tm}\ T.\ x \doteq e\\ \}\end{array}$$

Correspondingly, `Sub(NonTrivialUnital)` extends `Sub(Unital)` (see Example 121) with axiom symbols

$$\texttt{nottriv}^p\colon \Vdash \neg\forall\, x^p\colon \texttt{tm}\ T^p.\ x^p \doteq e^p$$
$$\texttt{nottriv}^s\colon \Vdash \neg\forall\, x^p.\ T^s\ x^p \Rightarrow x^p \doteq e^p$$

and thus also unital substructures must be non-trivial. And `Hom(NonTrivialUnital)` extends `Hom(Unital)` (see Example 104) with axiom symbols $\texttt{nottriv}^d$ and $\texttt{nottriv}^c$ for the domain and codomain structure.

Now consider a homomorphism, i.e., a `Hom(NonTrivialUnital)`-model, that maps everything to the neutral element of the codomain. Its image is a trivial `Unital`-model, but *not* a `NonTrivialUnital`-model. Thus, there cannot be a total morphism

$$\texttt{Img(NonTrivialUnital)}\colon \texttt{Sub(NonTrivialUnital)} \to \texttt{Hom(NonTrivialUnital)}$$

▶ **Example 145** (`Img` **Not Fully Extensible to** $\exists$)**.** Consider the formalization of unital structures below that instead of a function symbol for the neutral element uses an existential axiom symbol.

$$\begin{array}{l}\textbf{theory}\ \texttt{Unital}^\flat = \{\\ \quad \textbf{include}\ \texttt{SFOL}\\ \quad U \quad\ \colon \texttt{tp}\\ \quad \circ \quad\ \colon \texttt{tm}\ U \to \texttt{tm}\ U \to \texttt{tm}\ U\\ \quad \texttt{neut}^\flat\colon \Vdash \exists\, e\colon \texttt{tm}\ U.\ \forall\, x\colon \texttt{tm}\ U.\ e \circ x \doteq x\\ \}\end{array}$$

Consider a $\texttt{Hom(Unital}^\flat\texttt{)}$-model from some arbitrary domain model to a non-trivial codomain model that maps every element of the domain model to the codomain's non-neutral element.

The image of this homomorphism fails to be a $\texttt{Sub}(\texttt{Unital}^\flat)$-model because the proposition required by $\texttt{neut}^{\flat^s}$ is plain false. Thus, there cannot be a total morphism

$$\texttt{Img}(\texttt{Unital}^\flat)\colon \texttt{Sub}(\texttt{Unital}^\flat) \to \texttt{Hom}(\texttt{Unital}^\flat)$$

As remarked for `Sub` in ..., $\neg$ (and thus propositional logic `PL`) alone is not problematic, but as soon as $\neg$ and $\forall$ meet, there will be unprovable – or from the POV of `Img`: unassignable – axioms symbols ax^s.

▶ **Theorem 146.** `Img` *is well-typed.*

Proof. Easy to see at ??. ◀

▶ **Conjecture 147.** *Under mild assumptions (e.g., of additional equational theory in* `SFOL`*),* `Img` *is natural.*

5.6.2 Thoughts on a Generalized Definition

We defined the connector `Img` only for definitionless SFOL-theories for two reasons. First, the author was limited by time constraints and instead focussed on defining the functors `Hom`, `Sub`, and `Cong` in previous sections. Second, we had difficulties expressing the translation performed by `Img` in terms of logical relations[20] which would have eased generalizing the definition (e.g., as was the case with `Hom`, `Sub`, and `Cong`). Nonetheless, we shortly give two concrete examples that illustrate problems that would need to be overcome when generalizing `Img`.

▶ **Example 148** **(`Img` is Non-Obvious to Extend to PFOL).** Consider the PFOL-theory `Fun` below as well as its images under `Sub` and `Hom`. Note that, up to naming, `Fun` is the beginning of the theory `Lists` of generic lists from Example 101, thus has practical significance.

$$\begin{aligned}
&\textbf{theory}\ \texttt{Fun} = \{\\
&\quad \textbf{include}\ \texttt{SFOL}\\
&\quad \texttt{fun}\colon \texttt{tp} \to \texttt{tp}\\
&\}
\end{aligned}$$

$$\begin{aligned}
&\textbf{theory}\ \texttt{Sub}(\texttt{Fun}) = \{\\
&\quad \textbf{include}\ \texttt{SFOL}\\
&\quad \texttt{fun}^p\colon \texttt{tp} \to \texttt{tp}\\
&\quad \texttt{fun}^s\colon \Pi\, A^p\colon \texttt{tp}.\ \Pi\, A^s\colon \texttt{tm}\ A^p \to \texttt{prop}.\\
&\qquad\qquad \texttt{tm}\ \texttt{fun}^p\ A^p \to \texttt{prop}\\
&\}
\end{aligned}$$

$$\begin{aligned}
&\textbf{theory}\ \texttt{Hom}(\texttt{Fun}) = \{\\
&\quad \textbf{include}\ \texttt{SFOL}\\
&\quad \texttt{fun}^d, \texttt{fun}^c\colon \texttt{tp} \to \texttt{tp}\\
&\quad \texttt{fun}^h\qquad\colon \Pi\, A^d\ A^c\colon \texttt{tp}.\\
&\qquad\qquad\quad \Pi\, A^h\colon \texttt{tm}\ A^d \to \texttt{tm}\ A^c.\\
&\qquad\qquad\quad \texttt{tm}\ \texttt{fun}^d\ A^d \to \texttt{tm}\ \texttt{fun}^c\ A^c\\
&\}
\end{aligned}$$

Now consider how in `Img(Fun)` we have difficulties assigning anything sensible to $\texttt{fun}^s$:

$$\begin{aligned}
&\textbf{mor}\ \texttt{Img}(\texttt{Fun})\colon \texttt{Sub}(\texttt{Fun}) \to \texttt{Hom}(\texttt{Fun}) = \{\\
&\quad = =\\
&\}
\end{aligned}$$

[20] The same thing occurs for `Ker`

In particular note that if we wanted to use $\texttt{fun}^h$ somehow in the assignment, it would be non-obvious on how to instantiate its parameter $A^h : \texttt{tm}\ A^d \to \texttt{tm}\ A^c$. We do not readily have any such function. Overall, the author has not found a way to complete the putative morphism above, apart from trivial, non-sensibile solutions like always returning `true` or `false` in the assignment to $\texttt{fun}^s$.

▶ **Example 149** (Img **Could Be Extended to DFOL).** Consider the PFOL-theory Dep below as well as its images under Sub and Hom. We can see Dep as a minimalized version of the beginning of SmallCat, the theory representing small categories from Example 102. In particular, Hom(SmallCat) represents functors (see Example 112) and consequently Img(SmallCat) – if it was definable – would represent funtors' images. Thus, getting to work Img on Dep bears practical significance.

$$
\begin{array}{l}
\textbf{theory}\ \texttt{Dep} = \{\\
\quad \textbf{include}\ \texttt{SFOL}\\
\quad \texttt{A} : \texttt{tp}\\
\quad \texttt{B} : \texttt{tm A} \to \texttt{tp}\\
\}
\end{array}
$$

$$
\begin{array}{l}
\textbf{theory}\ \texttt{Sub(Dep)} = \{\\
\quad \textbf{include}\ \texttt{SFOL}\\
\quad \texttt{A}^p : \texttt{tp}\\
\quad \texttt{A}^s : \texttt{tm}\ \texttt{A}^p \to \texttt{prop}\\
\quad \texttt{B}^p : \texttt{tm}\ \texttt{A}^p \to \texttt{tp}\\
\quad \texttt{B}^s : \Pi\, x^p : \texttt{A}^p.\ \Pi\, x^s : \ \Vdash \texttt{A}^s\ x^p.\\
\qquad\qquad \texttt{tm}\ \texttt{B}^p\, x^p \to \texttt{prop}\\
\}
\end{array}
$$

$$
\begin{array}{l}
\textbf{theory}\ \texttt{Hom(Dep)} = \{\\
\quad \textbf{include}\ \texttt{SFOL}\\
\quad \texttt{A}^d, \texttt{A}^c : \texttt{tp}\\
\quad \texttt{A}^h \quad : \texttt{tm}\ \texttt{A}^d \to \texttt{tm}\ \texttt{A}^c\\
\quad \texttt{B}^d \quad : \texttt{tm}\ \texttt{A}^d \to \texttt{tp}\\
\quad \texttt{B}^c \quad : \texttt{tm}\ \texttt{A}^c \to \texttt{tp}\\
\quad \texttt{B}^h \quad : \Pi\, x^d : \texttt{tm}\ \texttt{A}^d.\\
\qquad\qquad \texttt{tm}\ \texttt{B}^d\, x^d \to \texttt{tm}\ \texttt{B}^c\, (\texttt{A}^h\, x^d)\\
\}
\end{array}
$$

Now consider how in Img(Dep) we have difficulties assigning anything sensible to $\texttt{B}^s$:

$$
\begin{array}{l}
\textbf{mor}\ \texttt{Img(Dep)} : \texttt{Sub(Dep)} \to \texttt{Hom(Dep)} = \{\\
\quad = =\\
\}
\end{array}
$$

Intuitively, the assignment to $\texttt{B}^s$ should work in the same way as for $\texttt{A}^s$: the original corresponding symbols B and A are both symbols that return an SFOL type. Thus, we would want ??? to be the proposition that checks whether z has a preimage under $\texttt{B}^h$. Unfortunately, we cannot state this as $\exists\ \ldots\ .\ \texttt{B}^h\ \ldots \doteq z$ since the return type of $\texttt{B}^h$ (being $\texttt{tm}\ \texttt{B}^c\ (A^h\, x^d)$ for some x^d) cannot be unified with the type of z (being $\texttt{tm}\ \texttt{B}^c\, y$).

Looking at the assumption x^s, there is actually an x such that we can express y in the form $y \doteq \texttt{A}^h\ x$ for some x. But SFOL's logic is not strong enough to be able to use that fact. Imagine we extended SFOL with a suitable cast operator ... `as` ... `via` ..., we could complete the assignment as:

$$??? = x^s\ \ \exists\mathrm{E}_{x^d, pf}\ \exists\, w.\ \texttt{B}^h\ x^d\ w \doteq (z\ \texttt{as}\ \texttt{B}^c\ (A^h\, x^d)\ \texttt{via}\ pf)$$

We conjecture that we could extend Img to a linear connector on all DFOL-theories this way.

```
mor Ker(Unital): Cong(Unital) → Hom(Unital) = {
   : :
}
```

Figure 31 Kernel of a unital homomorphism given by `Ker(Unital)`

5.7 Kernels of Homomorphisms

$$\texttt{Cong}(T) \xrightarrow{\texttt{Ker}(T)} \texttt{Hom}(T)$$

The linear connector `Ker`$(-)$ yields for any theory T the morphism `Ker`(T): `Cong`$(T) \to$ `Hom`(T) that translates every `Hom`(T)-model to the `Cong`(T)-model representing the homomorphism's kernel (as a congruence on the homomorphism's domain model). For example, applied to the theory `Group` it yields the morphism `Ker(Group)`: `Cong(Group)` $\to$ `Hom(Group)` formalizing precisely the notion that the kernel of every group homomorphism induces a congruence (equivalently: a normal subgroup) on the domain. (Even though most group theorists use normal subgroups rather than congruences, we settle with formalizing congruences as that is the more general, more syntactical notion, which is also used by universal algebraists.)

The connector `Ker` becomes particularly useful in combination with another one (which we left out in this thesis for space reasons): consider the connector `QuotMod`(T): $T \to$ `Cong`(T) that translates every `Cong`(T)-model representing a congruence model to an actual T-model ("quotient model"). For example, `QuotMod(Group)`: `Group` $\to$ `Cong(Group)` captures that every group congruence induces again a group by means of its equivalence classes.

$$T \xrightarrow{\texttt{QuotMod}(T)} \texttt{Cong}(T) \xrightarrow{\texttt{Ker}(T)} \texttt{Hom}(T)$$

Composing these connectors as above gives a connector that, e.g., applied to `Group` yields a morphism `Group` $\to$ `Hom(Group)` that translates every group homomorphism to the group of its kernel.

5.7.1 `Ker` on Definitionless `SFOL`

Before defining `Ker` and spelling out the details, let us first look at an example of what we would expect the morphism `Ker(Unital)`: `Cong(Unital)` $\to$ `Hom(Unital)` to look like (using `Unital` from Example 97):

▶ **Example 150** (`Ker(Unital)`)**.** We already showed `Cong(Unital)` and `Hom(Unital)` in Figures 28 and 22, respectively. In Figure 31 we show the connecting morphism `Ker(Unital)` that we would expect.

Recall that `Cong(Unital)` encodes a `Unital` structure for the domain of the intended `Unital` congruence structure by means of the d-superscripted constants. For every constant $c\colon A$ in `Unital`, the theory `Cong(Unital)` contains a systematically renamed copy $c^d\colon A^d$. Since we want `Ker(Unital)` to realize a `Unital` congruence structure in terms of a `Unital` homomorphism, namely on its domain, we systematically have assignments $c^d := c^d$ for all c in `Unital`. In the remaining assignments it is left to select an actual congruence (on terms of type U^d) and verify its properties (e.g., closure under function symbols).

First, in the assignment to U^g we realize the congruence that relates domain elements iff they are identified under the homomorphism.

Next, in the assignment to $\circ^g$ we prove that $\circ^d$ is well-defined wrt. the relation we assigned to U^g. Concretely, we prove that if the homomorphism identifies the pair of elements $(x_1, x_1^{d'})$ and another pair of elements $(x_2, x_2^{d'})$, then the homomorphism also identifies the elements in the pair $(x_1 \circ^d x_2, x_1^{d'} \circ^d x_2^{d'})$. Intuitively, the proof is a straightforwarded application of the homomorphism property of U^h wrt. $\circ^d$ (twice) and application of the assumptions of relatedness. In reality, things get a bit awkward when constructing the proof as an LF term. As in the previous example, we resort to omitting the full proof in Figure 31 to ease readability. Next, in the assignment to e^g we prove that the homomorphism actually identifies e^d and e^d (which is trivially the case). In general, for nullary (binary, ternary, …) function symbols f, assignments to f^g amount to proofs that the homomorphism identifies $f\ x_1\ \dots\ x_n$ and $f\ x_1^{d'}\ \dots\ x_n^{d'}$ if all arguments $(x_i, x_i^{d'})$ were already related. (NB: By induction this means that arbitrarily complex terms built out of function symbols and free variables are well-defined. For multisorted signatures (unlike `Unital`), the well-definedness holds wrt. the relations of all involved SFOL types.)

Finally, in the assignment to $\texttt{neut}^g$ we use $\texttt{neut}^d$ on the domain to prove neutrality; importantly, not wrt. equality $\doteq$, but wrt. the binary relation we assigned to U^g.

▶ **Definition 151** (Homomorphism Kernels). *The linear connector* `Ker` *from* `Cong` *into* `Hom` *on SFOL-theories is given by:*

- *type symbols* $T\colon \texttt{tp}$ *are mapped to*

$$T^d := T^d$$
$$T^g := \lambda x_1\ x_2 \colon \texttt{tm}\ T^d.\ T^h\ x_1 \doteq T^h\ x_2$$

- *function symbols* $f\colon \texttt{tm}\ T_1 \to \dots \to \texttt{tm}\ T_n \to \texttt{tm}\ T$ *are mapped to*

$$f^d := f^d$$
$$f^g := \lambda\, x_1 x_1' \colon \texttt{tm}\ T_1^d.\ \lambda\, x_1^g \colon\ \Vdash T_1^h\ x_1 \doteq T_1^h\ x_1'.\ \dots\ \lambda\, x_n x_n' \colon \texttt{tm}\ T_n^d.\lambda\, x_n^g \colon\ \Vdash T_n^h\ x_n \doteq T_n^h\ x_n'.$$
proof of
$$\left\{ \begin{array}{lll} T^h\ (f^d\ x_1\ \dots\ x_n) & \doteq f^c\ (T_1^h\ x_1)\ \dots\ (T_n^h\ x_n) & \textit{by homomorphism property } f^h \\ & \doteq f^c\ (T_1^h\ x_1')\ \dots\ (T_n^h\ x_n') & \textit{by assumptions } x_1^g, \dots, x_n^g \\ & \doteq T^h\ (f^d\ x_1'\ \dots\ x_n') & \textit{by homomorphism property } f^h \end{array} \right\}$$

- *predicate symbols* $p\colon \texttt{tm}\ T_1 \to \dots \to \texttt{tm}\ T_n \to \texttt{prop}$ *are left undefined*

- *axiom symbols* $ax\colon \Vdash F$ *are mapped to*

$$ax^d := ax^d$$
$$ax^g := \mathit{KerProof}(\varnothing;\quad F, F^g, F^c, ax^c, ax^c)$$

where KerProof is the partial function sketched in Definition 152 below (in case its output is undefined, we omit the assignment to ax^g*)*

▶ **Definition 152.** *The partial function KerProof is given by:*

Input

an SFOL-context Σ	$\vdash^{\mathtt{SFOL}}_{S} \Sigma$
i.e., of the form $\Sigma = \{x_1 \colon \mathtt{tm}\ U_1, \ldots, x_n \colon \mathtt{tm}\ U_n\}$	
a formula F	$\Sigma \vdash^{\mathtt{SFOL}}_{S} F \colon \mathtt{prop}$
F *as relativized by* Cong:	$\Sigma^d \vdash^{\mathtt{SFOL}}_{\mathtt{Cong}(S)} F^g \colon \mathtt{prop}$
F^c *as codomain-qualified by* Hom	$\Sigma^c \vdash^{\mathtt{SFOL}}_{\mathtt{Hom}(S)} F^c \colon \mathtt{prop}$
a proof of F^c	$\Sigma^c \vdash^{\mathtt{SFOL}}_{\mathtt{Hom}(S)} pf^c \colon \Vdash F^c$
a proof of $F^c\sigma$	$\Sigma^d \vdash^{\mathtt{SFOL}}_{\mathtt{Hom}(S)} pf^z \colon \Vdash F^c\sigma$

where σ *is the substitution given by* $\sigma = \{x^c \mapsto V^h\ x^d \mid (x \colon \mathtt{tm}\ V) \in \Sigma\}$

Output

a proof of $\mathtt{Ker}(F^g)$	$\Sigma^d \vdash^{\mathtt{SFOL}}_{\mathtt{Hom}(S)} pf \colon \Vdash \mathtt{Ker}(F^g)$

Steps

- *universal quantifier, i.e., when*

$$\begin{aligned} F &= \forall\, x \colon \mathtt{tm}\ U.\ G \\ F^g &= \forall\, x^d \colon \mathtt{tm}\ U^d.\ G^g \\ F^c &= \forall\, x^c \colon \mathtt{tm}\ U^c.\ G^c \\ \mathtt{Ker}(F^g) &= \forall\, x^d \colon \mathtt{tm}\ U^d.\ \mathtt{Ker}(G^g) \end{aligned}$$

Output

$$pf \leftarrow \forall\mathtt{I}_{x^d}\ \mathit{KerProof}(\Sigma, x \colon \mathtt{tm}\ U; \quad G, G^g, G^c, pf^c\ \ \forall\mathtt{E}\ \ x^d, pf^z\ \ \forall\mathtt{E}\ \ (U^h\ x^d))$$

- *equality, i.e., when*

$$\begin{aligned} F &= t_1 \doteq_U t_2 \\ F^g &= U^g\ t_1^d\ t_2^d \\ F^c &= t_1^c \doteq t_2^c \\ \mathtt{Ker}(F^g) &= U^h\ t_1^d \doteq U^h\ t_2^d \end{aligned}$$

Define the substitution $\pi = \{x^d \mapsto V^h\ x^d \mid (x \colon \mathtt{tm}\ V) \in \Sigma\}$. *Output the proof of the equality chain*

$$\begin{aligned} U^h\ t_1^d &\doteq \ldots \doteq t_1^d\pi && \textit{by maximally pushing down } U^h \\ &\doteq t_2^d\pi && \textit{by } pf^z \\ &\doteq \ldots \doteq U^h\ t_2^d && \textit{by maximally pulling up } U^h \end{aligned}$$

- *conjunction, i.e., when*

$$\begin{aligned} F &= G \wedge H \\ F^g &= G^g \wedge H^g \\ F^c &= G^c \wedge H^c \\ \mathtt{Ker}(F^g) &= \mathtt{Ker}(G^c) \wedge \mathtt{Ker}(H^c) \end{aligned}$$

Output

$$\begin{aligned} pf \leftarrow \wedge\mathtt{I}\ &(\mathit{KerProof}(\Sigma; \quad G, G^g, G^c, pf^c\ \wedge\mathtt{EL}, pf^z\ \wedge\mathtt{EL})) \\ &(\mathit{KerProof}(\Sigma; \quad G, G^g, G^c, pf^c\ \wedge\mathtt{ER}, pf^z\ \wedge\mathtt{ER})) \end{aligned}$$

- *disjunction, i.e., when*

$$\begin{aligned} F &= G \vee H \\ F^g &= G^g \vee H^g \\ F^c &= G^c \vee H^c \\ \mathtt{Ker}(F^g) &= \mathtt{Ker}(G^c) \vee \mathtt{Ker}(H^c) \end{aligned}$$

Output

$$\begin{aligned}
&pf \leftarrow pf^c\ \vee\mathtt{E}\\
&\qquad (\lambda k\colon\ \Vdash G^c.\ \vee\mathtt{IL}\ \mathit{KerProof}(\Sigma;\quad G, G^g, G^c, k, ???))\\
&\qquad (\lambda k\colon\ \Vdash H^c.\ \vee\mathtt{IR}\ \mathit{KerProof}(\Sigma;\quad H, H^g, H^c, k, ???))
\end{aligned}$$

In particular, KerProof is undefined on formulae containing negations, existential quantifiers, or applications of predicate symbols.

Definition 151 of `Cong` looks straightforward, having discussed the preceding example with the exception of the invocation of Definition 152, which looks rather complicated. We conjecture that a rigorous and digestible definition to be feasible using higher-order logical relations, an idea communicated by Florian Rabe outside the scope of this thesis. Nonetheless, we give a larger example next to better understand the preceding definition:

▶ **Example 153** (*KerProof* on a Complex Axiom)**.** Consider the following made-up theory whose sole purpose is to have a complex axiom symbol with nested function symbols:

$$\begin{array}{l}
\textbf{theory}\ \mathtt{NestedAxiom} = \{\\
\quad \textbf{include}\ \mathtt{SFOL}\\
\quad U : \mathtt{tp}\\
\quad \circ\ : \mathtt{tm}\ U \to \mathtt{tm}\ U \to \mathtt{tm}\ U\\
\quad ax\colon \Vdash \underbrace{\forall\, x\ y\colon \mathtt{tm}\ U.\ (x \circ e) \circ y \doteq y \circ x}_{\text{abbreviate as } F}\\
\}
\end{array}$$

The morphism `Ker(NestedAxiom)` given by applying Definition 151 is shown below. For reference, the theories `Cong(NestedAxiom)` and `Hom(NestedAxiom)` are shown in Figure 32. Even though `NestedAxiom`shares two declarations with `Unital` and thus a lot of syntax that is shown is mere repetition from previous examples, we still do so to remain self-contained in this example.

$$\begin{array}{l}
\textbf{mor}\ \mathtt{Ker(NestedAxiom)}\colon \mathtt{Cong(NestedAxiom)} \to \mathtt{Hom(NestedAxiom)} = \{\\
\quad = =\\
\}
\end{array}$$

Let us unfold Definition 152 to inspect the assignment to ax^g in detail:

$$
\begin{aligned}
&\mathit{KerProof}(\varnothing;\quad F, F^g, F^c, ax^c, ax^c)\\
&= \forall\mathtt{I}_{x^d}\ \mathit{KerProof}\left(\begin{array}{l} x\colon \mathtt{tm}\ U\\ \forall\, y\colon \mathtt{tm}\ U.\ (x \circ e)\circ y \doteq y \circ x\\ \forall\, y\colon \mathtt{tm}\ U.\ U^g\ ((x\circ e)\circ y)(y\circ x)\\ \forall\, y^c\colon \mathtt{tm}\ U^c.\ (x^c \circ^c e^c)\circ^c y^c \doteq y^c \circ x^c\\ ax^c\ \ \forall\mathtt{E}\ \ x^d\\ ax^c\ \ \forall\mathtt{E}\ \ (U^h\ x^d)\end{array}\right)\\
&= \forall\mathtt{I}_{x^d}\ \forall\mathtt{I}_{y^d}\ \mathit{KerProof}\left(\begin{array}{l} x\colon \mathtt{tm}\ U, y\colon \mathtt{tm}\ U\\ (x \circ e)\circ y \doteq y \circ x\\ U^g\ ((x\circ e)\circ y)(y\circ x)\\ (x^c \circ^c e^c)\circ^c y^c \doteq y^c \circ x^c\\ (ax^c\ \ \forall\mathtt{E}\ \ x^d)\ \ \forall\mathtt{E}\ \ y^d\\ (ax^c\ \ \forall\mathtt{E}\ \ (U^h\ x^d))\ \ \forall\mathtt{E}\ \ (U^h\ y^d)\end{array}\right)\\
&= \forall\mathtt{I}_{x^d}\ \forall\mathtt{I}_{y^d}\ \text{proof of}\\
&\left\{\begin{array}{lll} U^h\ ((x^d \circ^d e^d)\circ^d y^d) & \doteq (U^h\ (x^d \circ^d e^d)\circ^c (U^h\ y^d)) & \text{by } \circ^h \text{ (to push down } U^h)\\ & \doteq ((U^h\ x^d)\circ^c (U^h\ e^d))\circ^c (U^h\ y^d) & \text{by } \circ^h \text{ (to push down } U^h)\\ & \doteq ((U^h\ x^d)\circ e^c)\circ^c (U^h\ y^d) & \text{by } e^h \text{ (to push down } U^h\text{; now done)}\\ & \doteq (U^h\ y^d)\circ^c (U^h\ x^d) & \text{by } (ax^c\ \ \forall\mathtt{E}\ \ (U^h\ x^d))\ \ \forall\mathtt{E}\ \ (U^h\ y^d)\\ & \doteq U^h\ (y^d \circ^d x^d) & \text{by } \circ^h \text{ (to pull up } U^h\text{; now done)}\end{array}\right\}
\end{aligned}
$$

Indeed, the last line constitues a valid proof term to be assigned to ax^g (modulo expressing the equality rewriting as an LF term using `SFOL`'s `cong` axiom). Overall, the complexity of *KerProof* lies in the case for equalities, which necessitated defining all the input parameters to begin with.

▶ Remark 154 (Partiality on Predicate Symbols). Neither `Ker` nor *KerProof* can be defined on predicate symbols. The reason is that in general homomorphisms only preserve, but not reflect propositions. To see this for `Ker`, consider the following putative assignment to

$$
\begin{array}{l}
\textbf{theory}\ \texttt{Cong}(\texttt{NestedAxiom}) = \{\\
\quad \textbf{include}\ \texttt{SFOL}\\
\quad U^d : \texttt{tp}\\
\quad U^g : \texttt{tm}\ U^d \to \texttt{tm}\ U^d \to \texttt{prop}\\
\quad \circ^d\ \ : \texttt{tm}\ U^d \to \texttt{tm}\ U^d \to \texttt{tm}\ U^d\\
\quad \circ^g\ \ : \Pi\, x^d\ x^{d'} : \texttt{tm}\ U^d.\ \Pi\, x^g :\ \Vdash U^g\ x^d\ x^{d'}.\\
\qquad\qquad \Pi\, y^d\ y^{d'} : \texttt{tm}\ U^d.\ \Pi\, y^g :\ \Vdash U^g\ y^d\ y^{d'}.\\
\qquad\qquad \Vdash U^g\ (x^d \circ^d y^d)\ (x^{d'} \circ^d y^{d'})\\
\quad e^d\ \ : \texttt{tm}\ U^d\\
\quad e^g\ \ : \Vdash U^g\ e^d\ e^d\\
\quad ax^d : \forall\, x^d\ y^d : \texttt{tm}\ U^d.\ (x^d \circ e^d) \circ^d e^d \doteq y^d \circ^d x^d\\
\quad ax^g : \forall\, x^d\ y^d : \texttt{tm}\ U^d.\ U^g\ ((x^d \circ^d e^d) \circ^d y^d)\ (y^d \circ^d x^d)\\
\}
\end{array}
$$

$$
\begin{array}{l}
\textbf{theory}\ \texttt{Hom}(\texttt{NestedAxiom}) = \{\\
\quad \textbf{include}\ \texttt{SFOL}\\
\quad U^d : \texttt{tp}\\
\quad U^c : \texttt{tp}\\
\quad U^h : \texttt{tm}\ U^d \to \texttt{tm}\ U^c\\
\quad \circ^d\ \ : \texttt{tm}\ U^d \to \texttt{tm}\ U^d \to \texttt{tm}\ U^d\\
\quad \circ^c\ \ : \texttt{tm}\ U^c \to \texttt{tm}\ U^c \to \texttt{tm}\ U^c\\
\quad \circ^h\ \ : \Pi\, x\ y : \texttt{tm}\ U^d.\ \Vdash U^h\ (x \circ^d y) \doteq (U^h\ x) \circ^c (U^h\ y)\\
\quad e^d\ \ : \texttt{tm}\ U^d\\
\quad e^c\ \ : \texttt{tm}\ U^c\\
\quad e^h\ \ : \Vdash U^h\ e^d \doteq e^c\\
\quad ax^d : \forall\, x^d\ y^d : \texttt{tm}\ U^d.\ (x^d \circ^d e^d) \circ y^d \doteq y^d \circ x^d\\
\quad ax^c : \forall\, x^c\ y^c : \texttt{tm}\ U^c.\ (x^c \circ^c e^c) \circ ey^c \doteq y^c \circ x^c\\
\}
\end{array}
$$

Figure 32 TODO

predicate symbols $p\colon \texttt{tm}\ T_1 \to \dots \to \texttt{tm}\ T_n \to \texttt{prop}$.

$$
\begin{aligned}
p^d &:= p^d\\
p^g &:= \Pi\, x_1\ x_1'\colon \texttt{tm}\ T_1^d.\ \Pi\, x_1^g\colon\ \Vdash T_1^h\ x_1 \doteq T_1^h\ x_1'.\ \dots\ \Pi\, x_n\ x_n'\colon \texttt{tm}\ T_n^d.\ \Pi\, x_n^g\colon\ \Vdash T_n^h\ x_n \doteq T_n^h\ x_n'.\\
&\qquad \Vdash p^d\ x_1\ \dots\ x_n \Leftrightarrow p^d\ x_1'\ \dots\ x_n')\\
&\qquad = \lambda x_1\ x_1'\ \dots x_n\ x_n'.\ \text{proof of}\\
&\qquad\quad \begin{array}{lll}
p^d\ x_1\ \dots\ x_n & \Leftrightarrow p^c\ (T_1^h\ x_1)\ \dots\ (T_n^h\ x_n) & \text{by preservation \& reflection}\\
& \Leftrightarrow p^c\ (T_1^h\ x_1')\ \dots\ (T_n^h\ x_n') & \text{by assumptions } x_1^g \dots, x_n^g\\
& \Leftrightarrow p^d\ x_1'\ \dots\ x_n' & \text{by preservation \& reflection}
\end{array}
\end{aligned}
$$

As expected, preservation *and* reflection of the predicate symbol by the homomorphisms are crucial. Thus, we cannot take adop the above assignment for `Ker` since `Hom` only emits axioms requiring preservation. However, we could define a linear functor `ReflectiveHom` that maps every theory T to the theory of T-homomorphisms that preserve and reflect all predicate symbols (and by induction all propositions, too). The definition would be be identical to the one of `Hom` given in Definition 106 except that the relation at propositions F would be $F^d \Leftrightarrow F^c$ (instead of merely $F^d \Rightarrow F^c$). With such a functor, we could define a linear connector `ReflectiveKer`: `Cong` $\to$ `ReflectiveHom` that avoids the partiality of `Ker` on predicate symbols.

▶ **Conjecture 155.** *KerProof works as specified in Definition 152: for any input arguments adhering to the specification, the output is also typed as specified.*

▶ **Conjecture 156.** `Ker` *is well-typed.*

Proof. Assuming Conjecture 155, well-typedness can be read off Definition 151 given the limited domain on which `Ker` is defined. ◀

▶ **Conjecture 157.** *Under mild assumptions (e.g., of additional equational theory in* `SFOL`*),* `Ker` *is natural.*

5.7.2 Thoughts on a Generalized Definition

As a connector `Ker`: `Cong` $\to$ `Hom`, generalizing `Ker` hinges on the domain and codomain functor being suitably generalized. As of now, the bottleneck is `Cong`, which we have only defined for definitionless SFOL-theories. Thus, we defer generalizing `Ker` to future work.

5.8 Conclusion & Future Work

5.8.1 Conclusion

Algebra theories such as monoids, groups, and vectorspaces are ubiquitous throughout formal sciences, meriting their representation in formal systems. Even though universal algebra describes corresponding constructions (e.g., of homomorphisms, substructures, and congruence structures) that for each algebra theory systematically yield the corresponding theory, many current libraries of formal systems still rely on manually specifying all every algebra theory separately. However, due to their universal and constructive nature, these constructions are a perfect fit to be automated. We have cast several such constructions as structure-preserving diagram operators and learnt the following lessons:

We can **easily specify operators for folklore notions** that are given in standard literature (e.g., homomorphisms for first-order signatures with just function symbols). But it gets tricky if we want to extend them to what is needed in practice (predicate and axiom symbols). Yet more **considerable effort and novelty is needed to generalize those notions to the setting that MMT/LF automatically induces** on top of logic formalizations (e.g., polymorphism and dependent typing due to Π-types, and defined symbols and morphisms, i.e., in particular including proof terms). This increase in complexity when putting things onto a firm formal ground is to be expected, as is quite usual in the interactive theorem prover community. Note that we could have saved efforts and opted to specify operators for folklore cases only. But then they would be undefined on most interesting formalizations occurring in practice.

We used **logical relations as a guiding approach to generalize universal constructions** known in literature to our setting of polymorphic dependently-typed first-order logic with definitions (incl. proofs) and morphisms. Ideas inspired by logical relations served us to define complex, yet systematic specifications for constructions such as homomorphisms, substructures, and congruences. Much to our frustration, even though these three constructions in principle each behave like a logical relation, there turn out be minor, but persistent differences. Even worse, these differences seem to vary across construction. For example, for the homomorphism construction these differences are purely syntactical: a pure logical relation ansatz would produce syntax that semantically equals to what is desired. But for the substructure construction the differences are even semantical: logical relations strictly produce something different from what is desired. In any case, this led us to defining translations by first copying all cases as if they were a logical relation and then finetuning cases as desired.

With all those difficulties mentioned above, our framework's value might go unnoticed. But it was our very framework that allowed us to spend almost all our efforts on domain-specific issues (e.g., defining translations on terms), freeing us from any responsbilities of defining translations on theories, morphisms, or theory graphs. Thus, **domain experts can focus on domain problems, leaving all the bureaucracy of structure-preservation to the framework**.

5.8.2 Future Work

Now that specifications have been worked out for several operators of universal algebra, future work could focus on a corresponding **implementation & evaluation** based on the framework's implementation presented in Section 6. Initially, the author had successfully implemented homomorphisms, substructures, and congruences, among others, for definition-less SFOL-theories. But after a refactoring of the framework's implementation, this became dead code. And as we have seen most prominently in Sections 5.3 and 5.4, the ad-hoc specification on definitionless SFOL-theories does not scale anyway. Thus, the author prioritized working out the specification and postponed the implementation to future work.

We believe **higher-order logical relations** – part of ongoing work between Florian Rabe and the author – to be a worthwhile avenue of investigation, e.g., for defining connectors `Img`: `Sub` $\to$ `Hom` and `Ker`: `Cong` $\to$ `Hom`, which we only specified for definitionless SFOL-theories. In our sense (originally coined by [RS13]), a logical relation $r\colon m_1, \dots, m_n$ on morphisms $m_1, \dots, m_n\colon S \to T$ allows to express certain meta-theorems relating terms $\vdash^T m_1(t), \dots, m_n(t)$ for every typed term $\vdash^S t$. Consequently, an n-th order logical relation on $(n-1)$-th order logical relations $r\colon r_1, \dots, r_n$ would allow to express certain meta-theorems relating terms $\vdash^T r_1(t), \dots, r_n(t)$. Nonetheless, we stress that logical relations need not be the panacea for all universal constructions. There might even be different, more suitable abstractions for defining complex functions occurring in universal algebra altogether.

Provable Axioms: Recall that the substructure operator `Sub` maps axiom symbols $ax\colon \Vdash F$ to a qualified copy $ax^p : \Vdash F^p$ for the parent structure and to a relativized variant $ax^s : \Vdash F^s$. Here, even if ax is undefined, the generated constant ax^s may be definable (i.e., provable) in terms of ax^p. For example, all axioms using only $\forall$ and $\doteq$ (i.e., the axioms primarily used in universal algebra) are automatically true in each submodel. Consequently, `Sub` could do much better and translate such axioms to theorems by synthesizing an appropriate proof and adding it as the definiens of ax^s. We believe provable axioms to be a recurring theme for all logic-dependent operators, even beyond universal algebra. Thus, more generally, we could run a theorem prover on every axiom we generate (independent of its shape) and generate a definiens whenever a proof can be found.

This issue can be more subtle as a similar example with `Hom` shows: sometimes the generated axioms only become provable in the context of stronger theories. For example, every magma homomorphism between groups is automatically a group homomorphism. Thus, the preservation axioms e^h for the neutral element and i^h (where i is the unary function symbol for the inverse element) are provable in the theory `Hom`(`Group`), at which place it would therefore be desirable for `Hom` to add definitions. However, this makes it trickier for the operator to be structure-preserving: the theory `Hom`(`Monoid`) must still contain e^h without a definition, and the definition should only be added when `Hom`(`Monoid`) is included into `Hom`(`Group`).

Ideas for Operators Below we list some ideas for linear functors and connectors in the realm of universal algebra. We remain vague on operator domains and codomains and generally assume `SFOL` or suitable extensions thereof. Some ideas have been communicated by Florian Rabe directly and/or were taken from LATIN2[21].

Linear functors:

- `Iso`, `Mono`, `Epi`, `Endo`, ...: variants of `Hom` representing homomorphisms that are isomorphic, monomorphic (injective), epimorphic (surjective), or endomorphic (on a single model)
- `Prod`$_n(T)$: the theory whose models are n-tuples of T-models
- `Monotone`(T): extends T with a preorder on all its type symbols and corresponding compatibility axioms for all function symbols of T

[21] https://gl.mathhub.info/MMT/LATIN2/-/blob/39dc7046f457ff02f695387a8ebd80366789a465/source/math_theories_overview.txt

- $\texttt{Morph}(S, T)$: the bilinear functor maps theories S and T to $\texttt{Morph}(S, T)$ which extends both S and T and adds compatibility axioms between all function and predicate symbols from S and T. Special cases are:
  - $\texttt{Monotone} = \texttt{Morph}(\texttt{Preorder}, -)$, where $\texttt{Preorder}$ formalizes preorders
  - $\texttt{Morph}(\texttt{PartialOrder}, \texttt{Magma}) \cup \texttt{Group}$: partially-ordered group, where $\cup$ is theory union (a concept that pops up in certain fields of mathematics)
  - $\texttt{Morph}(\texttt{Lattice}, \texttt{Magma}) \cup \texttt{Group}$: lattice-ordered groups (ditto)
- $\texttt{GroupAction}(T)$: the theory extending T with a group action
- $\texttt{Action}(S, T)$: the theory extending T (the *actee*) with contains qualified copies c^{act} for every $c \in S$ (the *actor*), and that contains function symbols $\texttt{act}_T \colon \texttt{tm}\ T^{act} \to \texttt{tm}\ T \to \texttt{tm}\ T$ for every shared type symbol T, and that contains axiom symbols that make the collection of all $\texttt{act}_T$ functions a homomorphism from S to the T-endomorphisms. Special case is $\texttt{GroupAction} = \texttt{Action}(\texttt{Group}, -)$.
- $\texttt{Generated}(T)$: extending T with a closure function $\langle - \rangle_T \colon \texttt{tm}\ \mathcal{P}(T) \to \texttt{tm}\ \mathcal{P}(T)$ for every type $T \colon \texttt{tp}$ in T (and where $\mathcal{P}$ is a powerset operator given by suitably extending $\texttt{SFOL}$) and suitable axioms for every function symbol, such that $\langle G \rangle_T$ gives the subset of T generated by the subset G of T.[22]

Linear connectors:

- $\texttt{SubFull} \colon \texttt{Sub} \to \texttt{Id}$: the linear connector that yields morphisms $\texttt{SubFull}(T) \colon \texttt{Sub}(T) \to T$ translating T-models to full $\texttt{Sub}(T)$-models (i.e., T-submodels), i.e., where the subset predicate is constant true
- $\texttt{SubMod} \colon \texttt{Id} \to \texttt{Sub}$: the linear connector that yields morphisms $\texttt{SubMod}(T) \colon T \to \texttt{Sub}(T)$ translating $\texttt{Sub}(T)$-models (i.e., T-submodels) to actual T-models using predicate subtypes (when suitably extending $\texttt{SFOL}$)
- $\texttt{QuotMod} \colon \texttt{Id} \to \texttt{Cong}$: the linear connector that yields morphisms $\texttt{QuotMod}(T) \colon T \to \texttt{Cong}(T)$ translating $\texttt{Cong}(T)$-models (i.e., T-congruences) to actual T-models using quotient types (when suitably extending $\texttt{SFOL}$)

[22] See https://gl.mathhub.info/MMT/LATIN2/-/blob/39dc7046f457ff02f695387a8ebd80366789a465/source/algebra/generated.mmt for examples.

6 Implementation

6.1 Walkthrough: Using and Developing Diagram Operators

6.1.1 Library User's Perspective

6.1.2 Library Developer's Perspective

6.2 Class Hierarchy

6.3 Design Decisions & Limitations

7 Conclusion & Future Work

7.1 Conclusion

We **introduced a class of structure-preserving functorial operators** that act on diagrams of formalizations. This class comes equipped with an easy scheme to specify, verify, and implement operators. Developers only need to specify/implement operators for few syntax cases, focusing on their domain problem, and our framework lifts them to large structured diagrams. Our operators allow building small diagram expressions that evaluate to large diagrams whose structure remains intuitive and predictable to users. Similar to templating techniques in programming languages, they can be used to **automate developments in entire libraries** of formalizations and cut the size of humanly maintained portions. We state our results in the fairly general setting of Mmt where formalizations are composed out of *theories* (lists of constants) and *morphisms* between theories (subusming many compositional translations). We only assume the *Edinburgh Logical Framework (LF)* for concreteness, and in fact our framework, as stated, subsumes most declarative, typed languages including many logics, type theories, and set theories. Even though many presented operators depend on LF, their ideas could equally be worked out in one logical foundation or another.

Our operators can be seen as **different degrees of compositionality-breaking**: pushout-based translations are induced by the homomorphic extension of a morphism and thus entirely compositional; *strongly linear functors* use arbitrary expression translation functions and extend them compositionally to declarations and theories; *linear functors* use arbitrary declaration translation functions and extend them compositionally to theories; finally, the most general class translates theories without any constraint. Thus, our work can be seen as identifying good trade-offs between the rather restrictive pushout-based and the unpredictable arbitrary operators.

We presented several important operators in our framework. First, we presented **logic-independent operators**, which are operators that are applicable to formalizations over very weak logics, thus are rather domain-agnostic and of widespread use. The pushout operator acted as our canonical example here. Moreover, we considered logical relations for a logical framework to state meta theorems on theories and morphisms (as introduced in [RS13]) and gave an operator to internalize proofs of such meta theorems. Thus, our operators especially allow the representation and application of ubiquitous meta theorems, translations such as pushout being a special case thereof. Even though such meta theorems had long been wanted, e.g., in the LATIN atlas [Cod+a] to realize many logic translations, they could not be given previously as the lack of a flexible operator system meant that Mmt's main syntax and core would have needed modification. Thus, our operators also serve early and rapid prototyping of new propsective Mmt features. Finally, as a case study following [RR21b], we composed these operators with a refactoring-inspired operator to obtain the powerful operator that systematically translates formalizations of type theory from intrinsic to extrinsic style.

Second, as **logic-dependent operators** we presented several universal constructions from universal algebra. We specified operators that applied to diagrams D formalizing algebraic hierarchies (e.g., containing formalizations of monoids and groups) yield diagrams $\texttt{Hom}(D)$ of corresponding theories of homomorphisms, $\texttt{Sub}(D)$ of corresponding theories of substructures, and $\texttt{Cong}(D)$ of corrresponding theories of congruence structure. Thus, these operators can be used to significantly reduce the humanly maintained portions in the LATIN atlas needed to represent derived notions of the algebraic hierarchy.

We have **implemented a framework** for structure-preserving diagram operators for the MMT system (cf. source code [MMTb], documentation [MMTa]). Among the operators, we implemented the logic-independent ones and referred to an extensive case study [RR21b] (the implementation in the cited work having been solely developed by the author). We defer implementing the operators for universal algebra to future work.

In fact, it turned out that from the ones we presented most **logic-dependent operators were much more complicated to specify than initially thought**. After all, universal algebra and its constructions ought to be well-established material. However, the level of formality and generality of our setting forced us to invest considerable effort and novelty into generalizng many notions present in literature. Retrospectively, this is not surprising as, e.g., it is folklore that formalizing even decade-old mathematics in interactive theorem provers can require substantial novelty.

Despite those complexities, our theoretical framework proved to be useful insofar that we spent **most time on specifying domain-specific translations** (e.g., on terms) and could **let the framework handle the rest** (e.g., lifting to theories, morphisms, and diagrams). Thus, the present document can be seen as a positive evaluation of our theoretical framework. We leave it to future work to thoroughly evaluate the implemented framework, e.g., by implementing some of the operators from universal algebra that we specified. In some cases, typically concerning small formalizations and overly specific operators, we even believe that fully specifying and implementing an operator might take longer than typing the desired output by hand.

7.2 Limitations & Future Work

Two main directions of future work are *i*) defining and implementing more operators (esp. gaps in the LATIN atlas) and *ii*) improving the (theoretical and practical) framework. Both directions benefit from each other: the former may drive necessary design decisions for the latter, and the latter may make the former even easier in practice. Notably, with the framework laid out in this work, the former can already be done pretty efficiently, thus be a good starting point for an elaborated case study. Below we compiled a few points on the latter direction of future work.

Exploiting More of MMT's Structuring Features For brevity we described and proved correctness of our framework only for the most simple structuring features of MMT, even though our implementation already accounts for many more. Thus, a longer report may focus on bringing this to paper by describing syntax and semantics of MMT's structuring features (which for many features only exist in form of source code at time of writing) and extending our framework to them. In particular, many operators we presented awkwardly copy input declarations to qualified copies, something which could be stated more concisely using a structuring feature that MMT already possesses.

Static Type System for Functors A drawback of our approach is the lack of a static type system at the diagram expression level: all our diagram operators are partial, and the only way to check that $O(D)$ is defined is to successfully evaluate it. This is especially frustrating for more expressive operators that, e.g., take lists or even trees of identifiers as parameters. However, our experiments in this direction have indicated that any type system that could predict definedness would be too complicated to be practical. Moreover, in practice, the immediate evaluation of diagram expressions is needed anyway for two reasons: First, many operators can only be type-checked if their arguments are fully evaluated, thus obviating the main advantage of static type-checking. Second, because diagram expressions introduce theories that are to be used as interfaces later, their evaluation is usually triggered soon after type-checking.

Flexible Generation/Choice of Identifiers In Section 3.4 we defined how to lift an operator on anonymous diagrams in a straightforward way to an operator on named, structured diagrams occurring in practice. To do so, we assumed an injective function that translates those very identifiers occurring in named diagrams. For example, we can choose the function that suffixes all identifiers and, e.g., maps the theory identifier `Monoid` to `MonoidHom`. The assumption of such a function sounds benign on paper, but is challenging in practice since the function needs not only *i*) output unique identifiers for the system, but also identifiers that are *ii*) predictable, *iii*) typable, and *iv*) desired by humans. Arguably, for the homomorphism operator, suffixing with `Hom` is a good choice that fulfills all four properties. But even for the pushout functor along a fixed morphism m, it is unclear what to do in practice. We can easily choose to map theory identifiers X to $\texttt{pushout}.m.X$, but that would only fulfill the first two properties. In particular, since in the MMT system all identifiers are URIs, we would need to encode both URIs of m and X into the desired output URI. While certainly possible, without proper syntax, end users will not be able to enter resulting identifiers. Moreover, theories resulting from pushouts can often be given more sensible identifiers than our canonical choice above, but this absolutely requires a human's domain knowledge.

We believe a satisfactory solution must ultimately offer a syntax for end users to specify the injective naming function that a particular operator invocation should use, possibly overriding any default naming function that the operator comes equipped with. We suggest future work conduct a careful requirements analysis, incl. looking at operator invocations and taxonomies occurring in practice.

Coherence Properties of Operators We can often arrive at semantically equivalent diagrams by different ways of applying and parametrizing operators. In these cases, we are interested in making these *coherence properties* accessible to the user, e.g., by making it possible to treat semantically equivalent diagrams interchangably. Corresponding language features are difficult to design and implement – even if we restrict ourselves to syntactic equivalence. As an example, consider the pushout functor from Section 4.1 for which we proved $\texttt{Push}_n(\texttt{Push}_m(\Sigma)) = \texttt{Push}_{n\circ m}(\Sigma)$ as an equality on flat theories. This is a *flat coherence property*, which fails to extend to a *structured coherence property* on named theories because the left- and right-hand sides will yield differently named theories.[23] Particularly logic-independent operators such as pushout enjoy many flat coherence properties, incl. commutation properties with other functors.

Treating semantically equal theories differently due to existence of names is a general phenomenon of non-flat module systems. Possibly, the only viable solution might be to automatically derive isomorphisms from left- to right-hand sides of flat coherence properties, and to allow the user treating those isomorphisms transparently in syntax (i.e., by omitting them). This is tricky to get right, especially in combination with all other structuring syntax features of MMT.

[23] Technically, this is wrong, e.g., the shown identity with pushout can be made an identity with lifted variants. But we exclude cases of artificially crafted naming functions here that would never occur in practice.

References

[agd21] Contributors of agda/agda-stdlib on GitHub. *Agda standard library – Algebra Structures*. https://github.com/agda/agda-stdlib/blob/dd20869e959eb2eac3b8214ddf124b src/Algebra/Structures.agda. 2021. URL: https://github.com/agda/agda-stdlib/blob/dd20869e959eb2eac3b8214ddf124b40cabc03e7/src/Algebra/Structures.agda.

[AgdaAll21] Agda developers. *The Agda standard library: All. Lists where all elements satisfy a given property*. https://github.com/agda/agda-stdlib/blob/bc9c1b6a117fcb86b321113c0958ffc3b3526b4e/src/Data/List/Relation/Unary/All.agda. 2021. URL: %5Curl%7Bhttps://github.com/agda/agda-stdlib/blob/bc9c1b6a117fcb86b321113c0958ffc3b3526b4e/src/Data/List/Relation/Unary/All.agda%7D.

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

[Ast+02] E. Astesiano et al. "CASL – the Common Algebraic Specification Language". In: *Theoretical Computer Science* 286 (2002), pp. 153–196. URL: http://www.cofi.info.

[BC04] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development — Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. Springer Verlag, 2004.

[BLS18] Baldur Blöndal, Andres Löh, and Ryan Scott. "Deriving via: Or, How to Turn Hand-Written Instances into an Anti-Pattern". In: *Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell*. Haskell 2018. St. Louis, MO, USA: Association for Computing Machinery, 2018, pp. 55–67. ISBN: 9781450358354. DOI: 10.1145/3242744.3242746.

[BS12] Stanley Burris and H. P. Sankappanavar. *A Course in Universal Algebra. The Millennium Edition*. Vol. 78. Graduate Texts in Mathematics. Springer-Verlag New York, 2012. ISBN: 9780988055209.

[Bun93] Alexander Bunkenburg. "The Boom Hierarchy". In: *Functional Programming*. 1993. DOI: 10.1007/978-1-4471-3236-3_1.

[CAK17] Jacques Carette, Musa Al-hassy, and Wolfram Kahl. "Theories and Datastructures. "Two Sides of the Same Coin", or "Library Design by Adjunction"". Accessed 2022-02-18. Aug. 28, 2017. URL: %5Curl%7Bhttps://github.com/JacquesCarette/TheoriesAndDataStructures/blob/692cdcf7433c1533dfcbb1943d TheoriesAndDataStructures.pdf%7D.

[Cap99] Venanzio Capretta. "Universal Algebra in Type Theory". In: *Theorem Proving in Higher Order Logics, 12th International Conference, TPHOLs '99, volume 1690 of LNCS*. Springer-Verlag, 1999, pp. 131–148.

[Car+21] Jacques Carette et al. "Big Math and the One-Brain Barrier – The Tetrapod Model of Mathematical Knowledge". In: *Mathematical Intelligencer* 43.1 (2021), pp. 78–87. DOI: 10.1007/s00283-020-10006-0.

[CB16] David Christiansen and Edwin Brady. "Elaborator Reflection: Extending Idris in Idris". In: *SIGPLAN Not.* 51.9 (Sept. 2016), pp. 284–297. ISSN: 0362-1340. DOI: 10.1145/3022670.2951932.

[CFS20] Jacques Carette, William M. Farmer, and Yasmine Sharoda. “Leveraging the Information Contained in Theory Presentations”. In: *Intelligent Computer Mathematics: 13th International Conference, CICM 2020, Bertinoro, Italy, July 26–31, 2020, Proceedings*. Bertinoro, Italy: Springer-Verlag, 2020, pp. 55–70. ISBN: 978-3-030-53517-9. DOI: 10.1007/978-3-030-53518-6_4.

[Che20] Liang-Ting Chen. “Monadic typed tactic programming by reflection”. In: *Workshop on Type-driven Development (TyDe) 2019, at the 13th MathUI Workshop 2021, Mathematical User Interaction, at the International Conference on Functional Programming*. 2020. URL: https://tydeworkshop.org/2019-abstracts/paper20.pdf (visited on 02/22/2022).

[CMR16] M. Codescu, T. Mossakowski, and Florian Rabe. “Selecting Colimits for Parameterisation and Networks of Specifications”. In: *Workshop on Algebraic Development Techniques*. Ed. by M. Roggenbach and P. James. 2016.

[Cod+a] Mihai Codescu et al. “Project Abstract: Logic Atlas and Integrator (LATIN)”. In: pp. 289–291. DOI: 10.1007/978-3-642-22673-1_24.

[Cod+b] Mihai Codescu et al. “Towards Logical Frameworks in the Heterogeneous Tool Set Hets”. In.

[Com20] The mathlib Community. “The Lean Mathematical Library”. In: *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*. CPP 2020. New Orleans, LA, USA: Association for Computing Machinery, 2020, pp. 367–381. ISBN: 9781450370974. DOI: 10.1145/3372885.3373824.

[Del00] David Delahaye. “A Tactic Language for the System Coq”. In: *Logic for Programming and Automated Reasoning*. Ed. by Michel Parigot and Andrei Voronkov. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 85–95. ISBN: 978-3-540-44404-6. DOI: 10.1007/3-540-44404-1_7.

[DeM21] William DeMeo. “The Agda Universal Algebra Library and Birkhoff’s Theorem in Dependent Type Theory”. In: *CoRR* abs/2101.10166 (2021). source code: https://gitlab.com/ualib/ualib.gitlab.io. URL: https://arxiv.org/abs/2101.10166.

[Ebn+17] Gabriel Ebner et al. “A Metaprogramming Framework for Formal Verification”. In: *Proc. ACM Program. Lang.* 1.ICFP (Aug. 2017). DOI: 10.1145/3110278.

[FGT93] William M. Farmer, Joshua D. Guttman, and F. Javier Thayer. “IMPS: An Interactive Mathematical Proof System”. In: *Journal of Automated Reasoning* 11.2 (Oct. 1993), pp. 213–248.

[Gar+09] François Garillot et al. “Packaging Mathematical Structures”. In: *Theorem Proving in Higher Order Logics*. Ed. by Stefan Berghofer et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 327–342. ISBN: 978-3-642-03359-9.

[Geu+02] Herman Geuvers et al. “A Constructive Algebraic Hierarchy in Coq”. In: *Journal of Symbolic Computation* 34.4 (2002), pp. 271–286. ISSN: 0747-7171. DOI: https://doi.org/10.1006/jsco.2002.0552.

[GGP18] Emmanuel Gunther, Alejandro Gadea, and Miguel Pagano. “Formalization of Universal Algebra in Agda”. In: *Electronic Notes in Theoretical Computer Science* 338 (2018). The 12th Workshop on Logical and Semantic Frameworks, with Applications (LSFA 2017), pp. 147–166. ISSN: 1571-0661. DOI: https://doi.org/10.1016/j.entcs.2018.10.010.

[GM10] Georges Gonthier and Assia Mahboubi. "An introduction to small scale reflection in Coq". In: *J. Formaliz. Reason.* 3 (2010), pp. 95–152. DOI: 10.6092/issn.1972-5787/1979.

[Gog+93] J. Goguen et al. "Introducing OBJ". In: *Applications of Algebraic Specification using OBJ*. Ed. by J. Goguen, D. Coleman, and R. Gallimore. Cambridge, 1993.

[Gon+] G. Gonthier et al. "A Machine-Checked Proof of the Odd Order Theorem". In: pp. 163–179.

[Gon08] Georges Gonthier. "Formal proof – The Four-Color Theorem". In: *Notices of the AMS* 55.11 (2008), pp. 1382–1393. URL: http://www.ams.org/notices/200811/tx081101382p.pdf.

[Hal+17] Thomas Hales et al. "A formal proof of the Kepler conjecture". In: *Forum of Mathematics, Pi* 5 (2017). DOI: 10.1017/fmp.2017.1.

[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.

[HP18] Matthieu Herrmann and Alain Prouté. *On the dependent conjunction and implication.* 2018.

[HR20] Magne Haveraaen and Markus Roggenbach. "Specifying with syntactic theory functors". In: *Journal of Logical and Algebraic Methods in Programming* 113 (Apr. 2020). DOI: 10.1016/j.jlamp.2020.100543.

[Jac95] Paul B. Jackson. "Enhancing the NUPRL proof development system and applying it to computational abstract algebra". In: 1995. URL: https://homepages.inf.ed.ac.uk/pbj/papers/thesis.pdf.

[Kai+18] Jan-Oliver Kaiser et al. "Mtac2: Typed Tactics for Backward Reasoning in Coq". In: *Proc. ACM Program. Lang.* 2.ICFP (July 2018). DOI: 10.1145/3236773.

[Koh+] Michael Kohlhase et al. "FrameIT: Detangling Knowledge Management from Game Design in Serious Games". In: pp. 173–189. DOI: 10.1007/978-3-030-53518-6_11.

[Koh+09] M. Kohlhase et al. *Notations for Active Mathematical Documents.* Tech. rep. 2009-1. Jacobs University Bremen, 2009.

[Koh14] Michael Kohlhase. "Mathematical Knowledge Management: Transcending the One-Brain-Barrier with Theory Graphs". In: *EMS Newsletter* (June 2014), pp. 22–27. URL: https://kwarc.info/people/mkohlhase/papers/ems13.pdf.

[KS15] Pepijn Kokke and Wouter Swierstra. "Auto in Agda. Programming Proof Search Using Reflection". In: *Mathematics of Program Construction.* Ed. by Ralf Hinze and Janis Voigtländer. Cham: Springer International Publishing, 2015, pp. 276–301. ISBN: 978-3-319-19797-5. DOI: 10.1007/978-3-319-19797-5_14.

[LATIN] *LATIN2 – Logic Atlas Version 2.* URL: https://gl.mathhub.info/MMT/LATIN2 (visited on 06/02/2017).

[LATIN2] *LATIN2: Logic Atlas and Integrator.* URL: http://latin.omdoc.org (visited on 01/15/2020).

[LS19] Yannis Lilis and Anthony Savidis. "A Survey of Metaprogramming Languages". In: *ACM Comput. Surv.* 52.6 (Oct. 2019). ISSN: 0360-0300. DOI: 10.1145/3354584.

[Mak95] Michael Makkai. “First order logic with dependent sorts with Applications to Category Theory. Preliminary version”. Nov. 6, 1995. URL: https://www.math.mcgill.ca/makkai/folds/foldsinpdf/FOLDS.pdf (visited on 11/30/2021).

[mathcomp21] Contributors of math-comp/math-comp on GitHub. *Mathematical Components – Algebra Part of Algebraic Hierarchy.* https://github.com/math-comp/math-comp/blob/07830d201e29d208a5a4bf13e630468f03bfadd5/mathcomp/algebra/ssralg.v. 2021. URL: https://github.com/math-comp/math-comp/blob/07830d201e29d208a5a4bf13e630468f03bfadd5/mathcomp/algebra/ssralg.v.

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

[MMTb] *UniFormal/MMT – The MMT Language and System.* URL: https://github.com/UniFormal/MMT (visited on 10/24/2017).

[MmtURI] *MMT - URIs.* URL: https://uniformal.github.io/doc/language/uris.html (visited on 09/02/2021).

[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.

[Naw+19] M. Saqib Nawaz et al. *A Survey on Theorem Provers in Formal Methods.* 2019.

[Nor09] Ulf Norell. “Dependently Typed Programming in Agda”. In: *Proceedings of the 4th International Workshop on Types in Language Design and Implementation.* TLDI ’09. Savannah, GA, USA: Association for Computing Machinery, 2009, pp. 1–2. ISBN: 9781605584201. DOI: 10.1145/1481861.1481862.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic.* LNCS 2283. Springer, 2002.

[PD10] B. Pientka and J. Dunfield. “Beluga: A Framework for Programming and Reasoning with Deductive Systems (System description)”. In: *Automated Reasoning.* Ed. by J. Giesl and R. Hähnle. Springer, 2010, pp. 15–21.

[Péd19] Pierre-Marie Pédrot. “Ltac2: Tactical Warfare”. 2019. URL: https://www.xn--pdrot-bsa.fr/articles/coqpl2019.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.

[Raba] Florian Rabe. “First-Order Logic with Dependent Types”. In: pp. 377–391.

[Rabb] Florian Rabe. “The MMT API: A Generic MKM System”. In: pp. 339–343.

[Rab17a] Florian Rabe. “How to Identify, Translate, and Combine Logics?” In: *Journal of Logic and Computation* 27.6 (2017), pp. 1753–1798.

[Rab17b] Florian Rabe. “Morphism Axioms”. In: *Theoretical Computer Science* 691 (2017), pp. 55–80.

[Rab21] F. Rabe. “A Language with Type-Dependent Equality”. In: *Intelligent Computer Mathematics.* Ed. by F. Kamareddine and C. Sacerdoti Coen. Springer, 2021, pp. 211–227. DOI: 10.1007/978-3-030-81097-9_18.

[Rin+19] Talia Ringer et al. “QED at Large: A Survey of Engineering of Formally Verified Software”. In: *Foundations and Trends® in Programming Languages* 5.2-3 (2019), pp. 102–281. ISSN: 2325-1131. DOI: 10.1561/2500000045.

[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.

[Rou20] Navid Roux. *Structure-Preserving Diagram Operators*. Master Project Report. July 17, 2020. URL: https://gl.kwarc.info/supervision/projectarchive/-/blob/master/2020/Roux_Navid.pdf.

[Rou21a] Navid Roux. “A Beginner’s Guide to Logical Relations for a Logical Framework”. seminar paper. written as a student of the kwarc seminar. Mar. 22, 2021. URL: https://gl.kwarc.info/supervision/seminar/-/blob/master/WS2021/logrels/guide.pdf.

[Rou21b] Navid Roux. “A Beginner’s Guide to Logical Relations for a Logical Framework (slides)”. seminar presentation. presented as a student of the kwarc seminar. Jan. 27, 2021. URL: https://gl.kwarc.info/supervision/seminar/-/blob/master/WS2021/logrels/slides.pdf.

[RR20] Navid Roux and Florian Rabe. “Diagram Operators in a Logical Framework”. Extended Abstract. 2020. URL: https://lfmtp.org/workshops/2020/inc/papers/LFMTP_2020_paper_9.pdf.

[RR21a] Florian Rabe and Navid Roux. “Modular Formalization of Formal Systems”. under review. 2021. URL: https://kwarc.info/people/frabe/Research/RR_modlog_21.pdf.

[RR21b] Florian Rabe and Navid Roux. “Systematic Translation of Formalizations of Type Theory from Intrinsic to Extrinsic Style”. In: *Proceedings of the Workshop on Logical Frameworks: Meta-Theory and Practice (LFMTP)*. Ed. by E. Pimentel and E. Tassi. Open Publishing Association, 2021. URL: https://kwarc.info/people/frabe/Research/RR_soften_21.pdf.

[RR21c] Navid Roux and Florian Rabe. “Structure-Preserving Diagram Operators”. In: *Recent Trends in Algebraic Development Techniques*. Ed. by Markus Roggenbach. Vol. 12669. Lecture Notes in Computer Science. Springer International Publishing, 2021, pp. 142–163. ISBN: 978-3-030-73785-6. DOI: 10.1007/978-3-030-73785-6_8.

[RS09] Florian Rabe and C. Schürmann. “A Practical Module System for LF”. In: *Proceedings of the Workshop on Logical Frameworks: Meta-Theory and Practice (LFMTP)*. Ed. by J. Cheney and A. Felty. Vol. LFMTP’09. ACM International Conference Proceeding Series. ACM Press, 2009, pp. 40–48.

[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.

[RS19] Florian Rabe and Yasmine Sharoda. “Diagram Combinators in MMT”. In: *Intelligent Computer Mathematics*. Ed. by Cezary Kaliszyk et al. Cham: Springer International Publishing, 2019, pp. 211–226. ISBN: 978-3-030-23250-4. URL: https://kwarc.info/people/frabe/Research/RS_diagops_19.pdf.

[RV01] Alexandre Riazanov and Andrei Voronkov. “Vampire 1.1 (System Description)”. In: *Proceedings of the First International Joint Conference on Au-*

tomated Reasoning. IJCAR '01. Berlin, Heidelberg: Springer-Verlag, 2001, pp. 376–380. ISBN: 3540422544.

[SB18] Alexander Steen and Christoph Benzmüller. "The Higher-Order Prover Leo-III". In: *Automated Reasoning*. Ed. by Didier Galmiche, Stephan Schulz, and Roberto Sebastiani. Cham: Springer International Publishing, 2018, pp. 108–116. ISBN: 978-3-319-94205-6. DOI: 10.1007/978-3-319-94205-6_8.

[SCV19] Stephan Schulz, Simon Cruanes, and Petar Vukmirović. "Faster, Higher, Stronger: E 2.3". In: *Proc. of the 27th CADE, Natal, Brasil*. Ed. by Pascal Fontaine. LNAI 11716. Springer, 2019, pp. 495–507.

[Sha21] Yasmine Sharoda. "Leveraging Information Contained in Theory Presentations". PhD thesis. 2021. URL: http://hdl.handle.net/11375/26272.

[SJ95] Y. Srinivas and R. Jüllig. "Specware: Formal Support for Composing Software". In: *Mathematics of Program Construction*. Ed. by B. Möller. Springer, 1995.

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

[Soj10] Kristina Sojakova. "Mechanically Verifying Logic Translations". MA thesis. Jacobs University Bremen, 2010. URL: https://gl.kwarc.info/supervision/MSc-archive/blob/master/2010/sojakova_kristina.pdf.

[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.

[Soz+20] Matthieu Sozeau et al. "The MetaCoq Project". In: *Journal of Automated Reasoning* (Feb. 2020). DOI: 10.1007/s10817-019-09540-0.

[SP02] Tim Sheard and Simon Peyton Jones. "Template meta-programming for Haskell". In: *Proceedings of the 2002 Haskell Workshop, Pittsburgh*. Oct. 2002, pp. 1–16. URL: https://www.microsoft.com/en-us/research/publication/template-meta-programming-for-haskell/.

[SS08] C. Schürmann and J. Sarnat. "Structural Logical Relations". In: *2008 23rd Annual IEEE Symposium on Logic in Computer Science*. 2008, pp. 69–80. DOI: 10.1109/LICS.2008.44. (Visited on 01/26/2020).

[SW11] Bas Spitters and Eelis van der Weegen. "Type classes for mathematics in type theory†". In: *Mathematical Structures in Computer Science* 21 (2011), pp. 795–825. URL: https://arxiv.org/pdf/1102.1323.pdf.

[Wec92] Wolfgang Wechler. *Universal Algebra for Computer Scientists*. EATCS Monographs on Theoretical Computer Science. Berlin, Heidelberg: Springer, Berlin, Heidelberg, 1992. ISBN: 9783540542803. DOI: 10.1007/978-3-642-76771-5. (Visited on 10/22/2021).

[Wik22] Wikipedia contributors. *Short-circuit evaluation — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Short-circuit_evaluation&oldid=1067208237. [Online; accessed 3-February-2022]. 2022.

[WS] Paul van der Walt and Wouter Swierstra. "Engineering Proof by Reflection in Agda". In: *IFL – 24th International Symposium on Implementation and Application of Functional Languages*, pp. 157–173. URL: https://hal.inria.fr/hal-00987610/PDF/ReflectionProofs.pdf.