Modular Formalization of Formal Systems

Florian Rabe
University Erlangen-Nuremberg, Germany

Navid Roux
University Erlangen-Nuremberg, Germany

Abstract

Theorem provers use a wide variety of foundational systems. While a natural by-product of prover evolution, this variety can make it more difficult for integrating libraries, porting ideas across systems, or for users to start using and to switch systems. Moreover, it makes it very difficult to establish and formalize meta-theorems that compare and relate these foundations to each other.

We contribute to this problem by providing a systematically modular and integrated formalization of the most elementary formal systems including first and higher-order logic, dependent type theory, and set theory. We start with the fundamental concepts of terms, types, and propositions and mergers between them such as propositions-as-types. Then we formalize individual language features such as universal quantification and product types, which can then be combined into the respective formal systems.

We take particular care to state every feature only once and relative to minimal base languages and then to translate them automatically to other base languages, e.g., we generate the formalization of the typed universal quantification from the untyped one. The latter required developing novel mechanisms for formalizing the meta-theorems that guarantee the correctness of these translations. Our work shows how many formal systems, often seen as fundamentally different, can be formalized uniformly in a way that captures their similarities and allows knowledge sharing.

We use the MMT implementation of the logical framework LF, and our formalizations are available online.

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

Keywords and phrases formalization, meta-theory, modularity, translation, logical relation

Digital Object Identifier 10.4230/LIPIcs...

1 Introduction and Related Work

Motivation and Related Work The formalizing of formal systems in meta-logics is a long-standing research thread within the logic and theorem proving communities, arguably going back to the Automath framework [5]. The most successful modern logical frameworks are the $\lambda\Pi$ family [8, 3], the Isabelle system [18], and the λ-Prolog family [14]. Work in Isabelle has focused on building a generic theorem prover [19], LF has been applied mostly to analyzing meta-theory, e.g. [29], with applications of λ-Prolog somewhere in between, e.g. [7, 32].

Research on how to formalize a large and diverse set of formal systems was mostly done using LF [8, 1, 2], and modularity was a strong motivation from the beginning:

- It simplifies and allows scaling up formalization by allowing the reuse, translation, or combination of formalizations [9].
- It helps the meta-theoretical analysis as each modular construction is itself a meta-theorem, e.g. reusing the formalization of a language feature implies that two languages share that feature, and translations between languages can allow moving theorems across formal systems [16, 20].

- It allows building a library of formal systems (akin in spirit to [17] but with fully formal definitions) that can help both experienced users and novices navigate the space of formal systems [21, 4].

Recent work on logic formalizations has focused on exporting the large libraries of proof assistants into logical frameworks, e.g., for reverification or library translations [13, 31].

Our motivation here is the development of a library of formal systems with strong emphasis on integration. We apply to the realm of logics the little theories principle [6] of stating every result in the weakest possible theory and using theory morphisms [15] to import and translate between theories. For example, we (i) formalize the feature of conjunction in a separate theory that only depends on the existence of propositions and proofs, and then (ii) import it into every system that uses conjunction, possibly along a theory morphism such as the one that realizes propositions as booleans in higher-order logic.

The development of such a library has been a long-standing community goal but has so far proved very difficult. The Logosphere project [21] took place around the year 2000 and used Twelf. Arguably, it failed because Twelf lacked strong support for modularity then. The first author originally got involved around 2010 in the context of the LATIN project [4], which was more successful [11, 10] after developing a module system for Twelf [27].

Since then the first author has devoted a lot of effort towards improving the underlying logical framework infrastructure. This led to the development of the MMT system, which systematizes the modular structure [25, 23], allows efficiently implementing and experimenting with new logical frameworks [24], and offers modern IDE support for logic formalization [12].

Contribution Our present formalizations are made under the LATIN2 header as a complete reimplementation in MMT of the LATIN library, reflecting our improved understanding of the problem and exploiting MMT's improved tool support.

This paper serves as an introductory tour of LATIN2 describing the methodology and indicating the current state. It also presents the library itself with a particular focus on the foundational languages and features.

Moreover, some methodological innovations inspired and necessitated by LATIN2 are presented here for the first time. Firstly, MMT's new *realization* declarations have proved enormously helpful in mediating reuse across the library. Secondly, we present two new diagram operators in the sense of [26] that allow soundly implementing meta-theorems that represent complex translations: these generate (i) typed formalizations of a set of features from the corresponding untyped ones and (ii) soft-typed ones from the corresponding hard-typed ones. The lack of these features was a major limitation in LATIN.

Overview Presenting a highly modular hierarchic formalization can be quite difficult in the linear format of a paper. Therefore, Sect. 2 not only introduces the preliminaries such as the MMT language but also immediately uses the bottom theories of LATIN2 as examples. Then Sect. 3 describes formalizations of base languages, which define the available concepts like having types but not the individual features like product types. Sect. 4 and 5 give a representative sample of these features as well as the two diagram operators that generate some of them. Sect. 6 discusses limitations and future work. Proofs are moved to the appendix.

We prioritize keeping the present paper readable: We avoid introducing advanced MMT aspects (such as parsing rules, type inference, or named structures) even though we use them heavily in LATIN2. We present only a few representative examples in key theories and give only a few theorems (such as derived proof rules or morphisms between languages) even though that is where a modular library shines. LATIN2 is developed at `https://gl.`

mathhub.info/MMT/LATIN2/-/tree/devel/source, and we have copied and simplified the theories mentioned in this paper into the self-contained folder `casestudies/itp2021`.

2 The MMT Framework and Basic Formalizations

MMT [23] is a framework for designing and implementing logical frameworks. To simplify, we only use the implementation of LF that comes with MMT's standard library, and restrict the grammar to the main features of MMT/LF: We assume the reader is familiar with LF (see e.g., [8]) and only recap the notions of theories and morphisms that MMT adds on top.

Δ	$::= \cdot$		diagrams
	$\mid$	$\Delta,\ \texttt{theory}\ T := \{\Theta\}$	theory definition
	$\mid$	$\Delta,\ \texttt{morph}\ m\ : S\ \to T\ = \{\vartheta\}$	morphism definition
	$\mid$	$\Delta,\ \texttt{compute}\ D$	diagram computation
Θ	$::= \cdot$		declarations in a theory
	$\mid$	$\Theta,\ c : A[= t]$	typed, optionally defined constants
	$\mid$	$\Theta,\ \texttt{include}\ S \mid \texttt{realize}\ S$	include/realization of a theory
ϑ	$::= \cdot \mid \vartheta,\ c = t \mid \vartheta, \texttt{include}\ m$		declarations in a morphism
Γ	$::= \cdot \mid \Gamma, x : A$		contexts
t, A, f	$::= c \mid x \mid \texttt{type} \mid \texttt{kind} \mid \lambda_{x:A}\, t \mid \Pi_{x:A}\, B \mid f\, t$		LF expressions
D	$::= \texttt{diagram}((T, m)^*) \mid O(D)$		diagram expressions

Theories An MMT/LF theory is ultimately a list of **constant** declarations $c : A[= t]$ where the definiens t is optional. A constant declaration may refer to any previously declared constant. LF provides the primitives of dependently typed λ-calculus, namely universes `type` and `kind`, function types $\Pi_{x:A}\, B$, abstraction $\lambda_{x:A}\, t$ and application $f\, t$. In a constant declaration $c : A$, we must have $A : \texttt{type}$ or $A : \texttt{kind}$, and in a variable binding $x : A$, we must have $A : \texttt{type}$. As usual, MMT/LF allows writing $A \to B$ for $\Pi_{x:A}\, B$ and omitting inferable brackets, arguments, and types. If we need to be precise about **typing**, we write $\Gamma \vdash_T t : A$ for the typing judgment between two expressions that may use all constants from theory T and all variables from context Γ.

A theory T may **include** or **realize** a previously defined theory S. In both cases, all constants of S are available in T as if they were declared in T. Compared to ML, both ML signatures and structures correspond to MMT theories, `include` S corresponds to including signature S into signature T, and `realize` S corresponds to ascribing signature S to structure T. Thus, `include` S means all S-constants are available in T exactly as in S. But `realize` S means T must provide definitions for all not-yet-defined constants of S.

The relation that T includes or realizes S generates a preorder. The transitive property of this relation works as follows:

relation R to S	relation S to T	resulting relation R to T	source of definitions that witness that T realizes R
include	include	include	n/a
include	realize	realize	must be given in T
realize	include	realize	already given in S
realize	realize	realize	arise by composing those in S and T

▶ **Example 1.** We give the theories at the base of LATIN2. The left shows one theory each for the fundamental concepts of terms, types, propositions, and proofs. On the right, we have one theory each for the fundamental typing principles: `UTyped` is the base for untyped languages. `HTyped` formalizes hard typing, also called intrinsic or Church typing, where typing is a function from terms to types, i.e., every term has a unique type that can be inferred from it. That enables the representation of object language terms $t : A$ as LF terms $t : \mathtt{tm}\, A$. `STyped` formalizes soft typing, also called extrinsic or Curry typing, where typing is a relation between terms and types, i.e., a term may have multiple or no types. That corresponds to a representation of an object language term $t : A$ in LF as an untyped term $t : \mathtt{term}$ for which a proof of $\mathtt{ded}\,\mathtt{of}\, t\, A$ exists.

The theory `Proofs` formalizes proofs in standard LF fashion using the judgments-as-types principle: $\mathtt{ded}\, P$ is the type of proofs of the proposition $P : \mathtt{prop}$, i.e., $\mathtt{ded}\ P$ is non-empty iff p is provable. The definition of `incon` shows that in any formal system with propositions and proofs, we can define the judgment of inconsistency: as the type expressing that every proposition is provable. That definition is not impressive in itself but exemplifies the methodology of stating every definition in the smallest possible theory.

```
theory Terms =
  term  : type

theory Types =
  tp    : type

theory Props =
  prop  : type

theory Proofs =
  include Props
  ded   : prop → type
  incon : type = Π_{p:prop} ded p

theory UTyped =
  include Terms
  include Proofs

theory HTyped =
  include Types
  tm : tp → type
  include Proofs

theory STyped =
  include UTyped
  include Types
  of : term → tp → prop
```

Morphisms A morphism $m : S \to T$ represents a compositional translation of all S-syntax to T-syntax. We spell out the definition and key property:

▶ **Definition 2.** *A morphism $m : S \to T$ is a mapping of S-constants to T-expressions such that for all S-constants $c : A$ we have $\vdash_T m(c) : \overline{m}(A)$ where $\overline{m}$ maps S-syntax to T-syntax as defined in Fig. 1. In the sequel, we write m for $\overline{m}$.*

▶ **Theorem 3.** *For a morphism $m : S \to T$ and a theory E that includes S, if $\Gamma \vdash_E t : A$, then $m(\Gamma) \vdash_{E^m} m(t) : m(A)$.*

Altogether, that yields **three kinds of definitions** $c = t$ for an MMT constant $c : A$ of a theory S: directly in its declaration $c : A = t$, in some other theory that realizes S, and in a morphism out of S. Each realization of S in T can be seen as a special unnamed morphism $S \to T$. Realizations allow distinguishing definitions in different theories T realizing S. And morphisms allow naming and thus distinguishing different ways to realize S in T. This flexibility is important for modular formalizations as many language features can interpret each other in different ways.

In terms of category theory, a morphism m induces a **pushout functor** $\mathcal{P}(m)$ from the category of theories including S to the category of theories including T. As a functor, m extends to diagrams, i.e., any diagram of theories E including S and morphisms between constants of S

$$\overline{m}(c) = m(c)$$

other expressions

$$\begin{aligned}\overline{m}(x) &= x\\ \overline{m}(\mathtt{type}) &= \mathtt{type}\\ \overline{m}(\Pi_{x:A}\,B) &= \Pi_{x:\overline{m}(A)}\,\overline{m}(B)\\ \overline{m}(\lambda_{x:A}\,t) &= \lambda_{x:\overline{m}(A)}\,\overline{m}(t)\\ \overline{m}(f\,t) &= \overline{m}(f)\,\overline{m}(t)\end{aligned}$$

contexts

$$\begin{aligned}\overline{m}(\cdot) &= \cdot\\ \overline{m}(\Gamma, x:A) &= \overline{m}(\Gamma), x:\overline{m}(A)\end{aligned}$$

theories that include S

$$\begin{aligned}\overline{m}(E = \{\dots, D_i, \dots\}) &= E^m = \{\dots, \overline{m}(D_i), \dots\}\\ \overline{m}(\mathtt{include}\ S) &= \mathtt{include}\ T\\ \overline{m}(c : A[= t]) &= c : \overline{m}(A)[= \overline{m}(t)]\\ \overline{m}(\mathtt{include}\ E) &= \mathtt{include}\ E^m\\ \overline{m}(\mathtt{realize}\ E) &= \mathtt{realize}\ E^m\end{aligned}$$

constants of a theory including S

$$\overline{m}(c) = c$$

where E^m generates a fresh name for the translated theory

Figure 1 Map induced by a Morphism

them is mapped to a corresponding diagram of theories E^m including T. Moreover, for each E, m extends to a morphism $E \to E^m$ that maps every S-constant according to m and every other constant to itself. Each of these morphisms maps E-contexts/expressions to E^m and that mapping preserves all judgments. This is essential

▶ **Example 4.** The type erasure translation $\mathtt{TE} : \mathtt{HTyped} \to \mathtt{STyped}$ maps types $A : \mathtt{tp}$ to types $\mathtt{TE}(A) : \mathtt{tp}$, which we formalize by $\mathtt{tp} = \mathtt{tp}$. And it maps typed terms $t : \mathtt{tm}\,A$ to untyped terms $\mathtt{TE}(t) : \mathtt{term}$, which we formalize by $\mathtt{tm} = \lambda_{a:\mathtt{tp}}\,\mathtt{term}$ and thus $\mathtt{TE}(\mathtt{tm}\ A) = \mathtt{term}$. We also use `include Proofs` to include the identity morphism of `Proofs`, i.e., all constants of `Proofs` are mapped to themselves.

$$\begin{aligned}&\mathtt{morph\ TE} : \mathtt{HTyped} \to \mathtt{STyped} = \{\\ &\quad \mathtt{tp} = \mathtt{tp}\\ &\quad \mathtt{tm} = \lambda_{a:\mathtt{tp}}\,\mathtt{term}\\ &\quad \mathtt{include\ Proofs}\end{aligned}$$

$$\begin{aligned}&\mathtt{morph\ TE_{HProd}} : \mathtt{HProd} \to \mathtt{HProd^{TE}} = \{\\ &\quad \mathtt{include\ TE}\\ &\quad \mathtt{prod} = \mathtt{prod}\\ &\quad \mathtt{pair} = \mathtt{pair}\\ &\quad \mathtt{projL} = \mathtt{projL}\\ &\quad \mathtt{projR} = \mathtt{projR}\end{aligned}$$

$$\begin{aligned}&\mathtt{theory\ HProd^{TE}} =\\ &\quad \mathtt{include\ STyped}\\ &\quad \mathtt{prod} : \mathtt{tp} \to \mathtt{tp} \to \mathtt{tp}\\ &\quad \mathtt{pair} : \Pi_{a,b}\,\mathtt{term} \to \mathtt{term} \to \mathtt{term}\\ &\quad \mathtt{projL} : \Pi_{a,b}\,\mathtt{term} \to \mathtt{term}\\ &\quad \mathtt{projR} : \Pi_{a,b}\,\mathtt{term} \to \mathtt{term}\end{aligned}$$

$$\begin{aligned}&\mathtt{theory\ HProd} =\\ &\quad \mathtt{include\ HTyped}\\ &\quad \mathtt{prod} : \mathtt{tp} \to \mathtt{tp} \to \mathtt{tp}\\ &\quad \mathtt{pair} : \Pi_{a,b}\,\mathtt{tm}\,a \to \mathtt{tm}\,b \to \mathtt{tm}\,\mathtt{prod}\,a\,b\\ &\quad \mathtt{projL} : \Pi_{a,b}\,\mathtt{tm}\,\mathtt{prod}\,a\,b \to \mathtt{tm}\,a\\ &\quad \mathtt{projR} : \Pi_{a,b}\,\mathtt{tm}\,\mathtt{prod}\,a\,b \to \mathtt{tm}\,b\end{aligned}$$

Applying this morphism, i.e., the pushout functor $\mathcal{P}(\mathtt{TE})$, to the theory `HProd` of hard-typed simple products yields the theory $\mathtt{HProd^{TE}}$, which arises by replacing every occurrence of $\mathtt{tm}\,A$ with `term`. `TE` also extends to the morphism $\mathtt{TE_{HProd}}$, which translates all expressions of HProd to expressions of $\mathtt{HProd}^{\mathtt{TE}}$. This translations preserves LF-typing, e.g., if $\vdash_{\mathtt{HTyped}} t : \mathtt{tm}\,\mathtt{prod}\,A\,B$, then $\vdash_{\mathtt{HTyped}^{\mathtt{TE}}} \mathtt{TE}_{\mathtt{HProd}}(t) : \mathtt{term}$.

However, $\mathtt{HProd}^{\mathtt{TE}}$ is not the desired formalization of soft-typed products, and we will get back to that in Sect. 4.

Diagram Operators In order to overcome the limitations of $\mathcal{P}(m)$ such as the ones seen in Ex. 4, the more general concept of diagram operators was added to MMT in [30]. Like pushout, a diagram operator O from S to T is a functor from S-extensions T-extensions. Contrary to pushout, it remains open how the diagrams are mapped.

Diagram operators O capture a very common pattern in the meta-theory of formal systems: think of S and T as languages, of D as a library relative to S, and of $O(D)$ as a translation of that library from S to T. Note that often S and T may be small theories whereas D might be huge, e.g., we will define a diagram operator Soften from HTyped to STyped that overcomes the issues of $\mathcal{P}(\mathtt{TE})$.

Because D may be big, and libraries often need to be read by humans, it is important that O preserves the structure of D. The functoriality already ensures that morphisms and thus the structure of diagrams are preserved. (This is particularly relevant for those modular structuring mechanisms that are more complex than include/realize.) An **include-preserving** operator additionally maps include/realize declarations to include/realize declarations, e.g., $O(\mathtt{include}\ E) = \mathtt{include}\ E^O$ (where E^O is a fresh name again). A **definition-preserving** operator additionally maps definitions to corresponding definitions, e.g., $O(c : A = t) \ = \ c : O(A) = O(t)$. $\mathcal{P}(m)$ has both of those properties.

We call a diagram operator **natural** if it additionally provides a natural transformation from D to $O(D)$. That is, O provides a morphism $O_E : E \to E^O$ for every theory E in D such that all rectangles formed by two such morphisms, a morphism m in D and the corresponding morphism m^O in $O(D)$ commute. Natural operators produce not only a new library $O(D)$ but also provide the morphisms that allow translating expressions over a theory in D to the corresponding theory in $O(D)$. That is critical to show all D-theorems do in fact give rise to $O(D)$-theorems. Pushout is natural via the morphisms m_E.

Diagram expressions either collect some previously defined theories/morphism in $\mathtt{diagram}((T, m)^*)$ or apply an operator to such a diagram in $O(D)$. The toplevel declaration $\mathtt{compute}\,D$ invokes all operators in D and inserts all newly generated theories/morphisms into the toplevel diagram Δ.

In [26] we developed a framework that makes it easy to construct such diagram operators O. Here O only has to be defined for flat theories and morphisms, i.e., those that do not contain any includes/realizes or any defined constants, and the framework transparently lifts O to a natural include- and definition-preserving diagram operator. A weakness of diagram operators is that each O is currently defined in the underlying programming language of MMT and thus forms a part of the trusted code base. That is not ideal but relatively harmless in practice for two reasons: The framework takes care of almost all the bureaucracy so that users can add new well-behaved diagram operators very easily and inject them into MMT at run time. Moreover, diagram operators are conservative in that they only produce additional theories and morphisms: $O(D)$ is rendered back to the user in human-readable syntax that does not rely on O anymore.

3 Fundamental Concepts

Ex. 1 shows our four fundamental primitive **concepts**: terms, types, propositions, and proofs, from which we build the three base languages of untyped, hard-typed, and soft-typed logic. Additional base languages can be built by mixing in features such as the following.

▶ Remark 5 (Hard vs. Soft Typing). While it is usually advisable to formalize a soft (hard) typed system using the above soft (hard)-typed style, the soft/hard distinction of a formal system is in fact orthogonal to the soft/hard distinction of its formalization. Formalizing a hard-typed system in `STyped`-style is usually less convenient, as every variable or constant must be paired with a typing axioms and type inference is reduced to proof search as opposed to LF type inference. But it leads to smaller representations.

Vice versa, formalizing a soft-typed system using `HTyped` style requires using casting functions $\texttt{tm}\,a \to \texttt{tm}\,b$ whenever a is a subtype of b. That often hampers scalability.

theory `Classical` $=$
  include `Proofs`
  $\texttt{classical} : \Pi_a\,((\texttt{ded}\,a \to \texttt{incon}) \to \texttt{incon}) \to \texttt{ded}\,a$

theory `ProofIrrel` $=$
  include `Proofs`
  identify all $s, t : \texttt{ded}\,P$ for any P

Logics are **intuitionistic** by default. To formalize a classical logic, we include `Classical`. Like with inconsistency, we can capture classicality in a way that does not depend on the details of the formal system at all. Once concrete connectives are present with appropriate proof rules, `classical` allows deriving the usual classical properties such as $\texttt{ded}\,a \vee \neg a$.

Similarly, logics are **proof-relevant** by default. To obtain proof irrelevance, we need to add a typing rule that makes all terms of type $\texttt{ded}\,F$ equal. That is not possible in plain LF. However, within Mmt, we can easily go beyond LF and write a theory `ProofIrrel` that injects such a rule into the type inference algorithm [24]. That is a routine part of our formalization, but we omit it here.

Not every formal systems uses terms, types, *and* propositions. Some employ **concept mergers** of which we have identified four important ones in practical systems. The first of them is the well-known propositions-as-types principle, and we have invented corresponding names for the other three. Each of them is formalized as a separate theory that can be mixed into a base language.

theory `PrAsTy` $=$
  include `Types`
  realize `Props`
  $\texttt{prop} = \texttt{tp}$

theory `PrAsTe` $=$
  include `HTyped`
  $\texttt{bool} : \texttt{tp}$
  realize `Props`
  $\texttt{prop} = \texttt{tm}\,\texttt{bool}$

theory `TyAsPr` $=$
  include `UTyped`
  realize `STyped`
  $\texttt{tp} = \texttt{term} \to \texttt{prop}$
  $\texttt{of} = \lambda_t\,\lambda_a\,a\,t$

theory `TyAsTe` $=$
  include `UTyped`
  realize `Types`
  $\texttt{tp} = \texttt{term}$
  include `STyped`

Propositions-as-Types is characteristic of systems following the Curry-Howard correspondence to represent propositions as special cases of types. The prototypical example is dependent type theory (although practical systems like Coq additionally use universes and thus require a more complex formalization). `PrAsTy` formalizes this by realizing the theory `Props` after including the theory `Types`. Thus, only `Types` is primitive and the concepts of `Props` are emergent notions. We can extend `PrAsTy` to formalize the various Curry-Howard isomorphisms, e.g., to realize conjunction in terms of `HProd`.

Propositions-as-Terms is characteristic of systems using a distinguished type `bool` : `tp` to represent propositions as a special case of terms. The prototypical example is higher-order logic. `PrAsTe` formalizes this by realizing `prop` = `tm bool`. After including `PrAsTe`, we can proceed as if propositions were primitive and include features that depend on propositions such as `Proofs`. Moreover, we use combinations of include and realize to build HOL in three ways: (i) Including all logical features as primitive describes a neutral version of HOL that can serve as the base of a joint library. (ii) After including just equality, we realize the remaining features of first-order logic in the style of Andrews as done in HOL Light. (iii) After including just universal quantification and implication, we can realize the remaining features in the style of Prawitz.

Types-as-Propositions is characteristic of systems that are a priori untyped and in which typing is an emerging feature given by predicates on objects. This is common in many computer algebra systems and also part of Mizar. Technically, we should call it "types-as-predicates", but our chosen name creates a more memorable symmetry of concept mergers. `TyAsPr` formalizes this by including `Terms` and `Props` and then realizing `STyped`. `tp` becomes the type `term` → `prop` of unary predicates and the `of` relation is realized via LF function application.

Types-as-Terms is characteristic of systems that do not distinguish terms and types and use a binary predicate between objects to capture type-like behavior. The prototypical example is set theory. `TyAsTe` formalizes this by first including `Terms` and `Props` and then realizing `Types` via `tp` = `term`. `STyped` depends on `Terms`, `Props`, and `Types`; the former two are already covered by includes of `TyAsTe`, and the dependency on `Types` is identified with the realization in `TyAsTe`. Thus, the include of `STyped` only adds the `of` constant, now with the type `term` → `term` → `prop`. That is the ∈ predicate of set theory.

Both `TyAsTe` and `TyAsPr` include/realize `STyped`, thus yielding two different ways to realize soft typing. In set theory, we often need to combine both of them. This is possible in MMT, too, but it requires packing one or both realizations into a named morphism. We omit that here for simplicity.

4 Type Theoretical Features

Naturally, there are no type-theoretical language features for untyped systems. But for hard and soft-typed systems we can formalize an array of orthogonal features that can be combined and interrelated flexibly to formalize specific type theories.

Because soft typing is more expressive than hard typing, the hard-typed features can also be expressed using soft typing. To avoid a duplication of formalization effort and to ensure a systematic correspondence between features, we define a diagram operator `Soften` from `HProd` to `SProd` that systematically turns every hard-type feature into its soft-typed analog.

4.1 Hard-Typed Features

Ex. 4 already showed the formalization `HProd` of simple product types. As additional examples, we give the formalizations of simple and dependent function types as well as the morphism `HSFtoDF` that represents the former in terms of the latter. Here we also include hard-typed equality `HEqual` to formulate the reduction rules (where some inferable arguments are omitted for brevity). There is a trade-off regarding whether each reduction rule should be factored into a separate theory. Here we only do it for η and extensionality to that we can exemplify that the former realizes the latter. We could also give a morphism `Exten` → `Eta` for the opposite direction.

$$
\begin{array}{l}
\texttt{theory HEqual} \;=\\
\quad \texttt{include HTyped}\\
\quad \texttt{eq} \quad\; : \Pi_a\, \texttt{tm}\, a \to \texttt{tm}\, a \to \texttt{prop}\\
\quad \texttt{refl} \; : \Pi_{a,x}\, \texttt{ded}\, \texttt{eq}\, a\, x\, x\\
\quad \texttt{eqsub} : \Pi_{a,x,y}\, \texttt{ded}\, \texttt{eq}\, a\, x\, y \to\\
\qquad\qquad \Pi_{F:\texttt{tm}\, a \to \texttt{prop}}\, \texttt{ded}\, F\, x \to \texttt{ded}\, F\, y
\end{array}
$$

$$
\begin{array}{l}
\texttt{theory HSimpFun} \;=\\
\quad \texttt{include HEqual}\\
\quad \texttt{fun} \quad : \texttt{tp} \to \texttt{tp} \to \texttt{tp}\\
\quad \texttt{lam} \quad : \Pi_{a,b}\, (\texttt{tm}\, a \to \texttt{tm}\, b) \to \texttt{tm}\, \texttt{fun}\, a\, b\\
\quad \texttt{app} \quad : \Pi_{a,b}\, \texttt{tm}\, \texttt{fun}\, a\, b \to \texttt{tm}\, a \to \texttt{tm}\, b\\
\quad \texttt{beta} : \Pi_{a,b}\, \Pi_{F:\texttt{tm}\, a \to \texttt{tm}\, b}\, \Pi_x\\
\qquad\qquad \texttt{ded}\, \texttt{eq}\, (\texttt{app}\, (\texttt{lam}\, F)\, x)\, (F\, x)
\end{array}
$$

$$
\begin{array}{l}
\texttt{theory HDepFun} \;=\\
\quad \texttt{include HEqual}\\
\quad \texttt{fun} \quad : \Pi_{a:\texttt{tp}}\, (\texttt{tm}\, a \to \texttt{tp}) \to \texttt{tp}\\
\quad \texttt{lam} \quad : \Pi_a\, \Pi_{b:\texttt{tm}\, a \to \texttt{tp}}\, (\Pi_{x:\texttt{tm}\, a}\, \texttt{tm}\, b\, x)\\
\qquad\qquad \to \texttt{tm}\, \texttt{fun}\, a\, b\\
\quad \texttt{app} \quad : \Pi_{a,b}\, \texttt{tm}\, \texttt{fun}\, a\, b \to \Pi_{x:\texttt{tm}\, a}\, \texttt{tm}\, b\, x\\
\quad \texttt{beta} : \Pi_{a,b}\, \Pi_{F:\Pi_{x:\texttt{tm}\, a}\, \texttt{tm}\, b\, x}\, \Pi_x\\
\qquad\qquad \texttt{ded}\, \texttt{eq}\, (\texttt{app}\, (\texttt{lam}\, F)\, x)\, (F\, x)
\end{array}
$$

$$
\begin{array}{l}
\texttt{morph HSFtoDF} \;: \texttt{HSimpFun} \;\to \texttt{HDepFun} \;= \{\\
\quad \texttt{include HEqual}\\
\quad \texttt{fun} \quad = \lambda_{a,b}\, \texttt{fun}\, a\, \lambda_{x:\texttt{tm}\, a}\, b\\
\quad \texttt{lam} \quad = \lambda_{a,b,f}\, \texttt{lam}\, a\, (\lambda_x\, b)\, f\\
\quad \texttt{app} \quad = \lambda_{a,b,f,x}\, \texttt{app}\, a\, (\lambda_x\, b)\, f\, x\\
\quad \texttt{beta} = \lambda_{a,b,F,x}\, \texttt{beta}\, a\, (\lambda_x\, b)\, F\, x
\end{array}
$$

$$
\begin{array}{l}
\texttt{theory Exten} \;=\\
\quad \texttt{include HSimpFun}\\
\quad \texttt{exten} : \Pi_{a,b}\, \Pi_{f,g:\texttt{tm}\, \texttt{fun}\, a\, b}\\
\qquad\qquad (\Pi_x\, \texttt{ded}\, \texttt{eq}\, b\, (\texttt{app}\, f\, x)\, (\texttt{app}\, g\, x))\\
\qquad\qquad \to \texttt{ded}\, \texttt{eq}\, (\texttt{fun}\, a\, b)\, f\, g
\end{array}
$$

$$
\begin{array}{l}
\texttt{theory Eta} \;=\\
\quad \texttt{include Beta}\\
\quad \texttt{eta} \quad : \; \Pi_{a,b}\, \Pi_{f:\texttt{tm}\, \texttt{fun}\, a\, b}\\
\qquad\qquad \texttt{ded}\, \texttt{eq}\, (\texttt{fun}\, a\, b)\, f\, (\texttt{lam}\, \lambda_x\, \texttt{app}\, f\, x)\\
\quad \texttt{realize Exten}\\
\quad \texttt{exten} = \text{omitted}
\end{array}
$$

4.2 Softening Hard-Typed Features

Type Erasure as a Theory Morphism In Ex. 4, we already defined $\texttt{TE} : \texttt{HTyped} \to \texttt{STyped}$ and saw that the operator $\mathcal{P}(\texttt{TE})$ does not soften correctly. We actually need the theory SProd below, and we can easily adapt the morphism $\texttt{TE}_{\texttt{HProd}}$ to yield a morphism e capturing the intended syntax translation. SProd differs from $\texttt{HProd}^{\texttt{TE}}$ in two ways: The term constructors do not take the spurious type arguments as in $\texttt{projL} : \Pi_{a,b}\, \texttt{tm}\, \texttt{prod}\, a\, b \to \texttt{tm}\, a$, and each term constructor c comes with its typing rule c^*. Yet, we are still missing a formalization of the type preservation invariant: whenever $t : \texttt{tm}\, A$ over HProd, there is a proof of $\texttt{ded}\, \texttt{of}\;\, e(t)\, e(A)$ over SProd.

$$
\begin{array}{l}
\texttt{theory SProd} \;=\\
\quad \texttt{include STyped}\\
\quad \texttt{prod} \quad : \texttt{tp} \to \texttt{tp} \to \texttt{tp}\\
\quad \texttt{pair} \quad : \texttt{term} \to \texttt{term} \to \texttt{term}\\
\quad \texttt{pair}^* \;\; : \Pi_{a,b}\, \Pi_x\, \texttt{ded}\, \texttt{of}\, x\, a \to \Pi_y\, \texttt{ded}\, \texttt{of}\, y\, b\\
\qquad\qquad \to \texttt{ded}\, \texttt{of}\, (\texttt{pair}\, x\, y)\, (\texttt{prod}\, a\, b)\\
\quad \texttt{projL} \;\, : \texttt{term} \to \texttt{term}\\
\quad \texttt{projL}^* : \Pi_{a,b}\, \Pi_x\, \texttt{ded}\, \texttt{of}\, x\, (\texttt{prod}\, a\, b) \to \texttt{ded}\, \texttt{of}\, (\texttt{projL}\, x)\, a\\
\quad \texttt{projR} \;\, : \texttt{term} \to \texttt{term}\\
\quad \texttt{projR}^* : \Pi_{a,b}\, \Pi_x\, \texttt{ded}\, \texttt{of}\, x\, (\texttt{prod}\, a\, b) \to \texttt{ded}\, \texttt{of}\, (\texttt{projR}\, x)\, b
\end{array}
$$

$$
\begin{array}{l}
\texttt{morph}\; e \;: \texttt{HProd} \;\to \texttt{SProd} \;= \{\\
\quad \texttt{include TE}\\
\quad \texttt{prod} \quad = \texttt{prod}\\
\quad \texttt{pair} \quad = \lambda_{a,b,x,y}\, \texttt{pair}\, x\, y\\
\quad \texttt{projL} = \lambda_{a,b,x}\, \texttt{projL}\, x\\
\quad \texttt{projR} = \lambda_{a,b,x,y}\, \texttt{projR}\, x
\end{array}
$$

To obtain SProd systematically from HProd, we first define a new diagram operator that removes the spurious type arguments:

▶ **Definition 6** (Unused Positions). *Consider a constant $c : A$ in a theory S in a diagram D. After suitably normalizing, A must start with a (possibly empty) sequence of n Π-bindings, and any definition of c (direct, realized, or morphism) must start with the same variable sequence λ-bound. We write $c^1, \ldots, c^n$ for these variable bindings. Each occurrence of c in an expression in D is (after suitably η-expanding if needed) applied to exactly n terms, and we also write c^i for those argument positions.*

*We call a set P of argument positions of D-constants **unused** if for every $c^i \in P$, the i-th bound variable of the type or any definition of c occurs at most in argument positions that are themselves in P.*

We write $D \setminus P$ for the diagram that arises from P by removing for every $c^i \in P$

- *the i-th variable binding in the type and all definitions of c, e.g., $c : \Pi_{x_1:A_1} \Pi_{x_2:A_2} B$ becomes $c : \Pi_{x_1:A_1} B$ if $i = 2$,*
- *the i-argument of any application of c, e.g., $c\, t_1\, t_2$ becomes $c\, t_1$ if $i = 2$.*

▶ **Lemma 7** (Removing Unused Positions). *Consider a well-typed diagram D and a set P of argument positions unused in D. Then $D \setminus P$ is also well-typed.*

Implementing the operation $D \setminus P$ is straightforward. However, much to our surprise and frustration, automatically choosing an appropriate set P turned out to be difficult:

▶ **Example 8.** The undesired argument positions in $\mathtt{TE}^{\mathtt{HProd}}$ are exactly the named variables in HProd that do not occur in their scopes in $\mathtt{TE}^{\mathtt{HProd}}$ anymore. This includes the positions $\mathtt{pair}^1$ and $\mathtt{pair}^2$, and removing them yields the desired declaration of pair in SProd.

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

After several failed attempts, we have been unable to find a good heuristic for choosing P. For now, we remove all named variables that never occur in their scope anymore, and we allow users to annotate positions like $\mathtt{fun}^1$ where the system should deviate from that heuristic. We anticipate finding better solutions after collecting more data in the future. In the sequel, we write $\mathcal{P}^-(m)(D) := \mathcal{P}(m)(D) \setminus P_D$ where P_D is any fixed heuristic. $\mathtt{HProd}^{\mathcal{P}^-(\mathtt{TE})}$ yields the theory SProd except that it still lacks the *-ed constants. The following lemma shows that we can now obtain the morphism $e : \mathtt{HProd} \to \mathtt{SProd}$ from above as $\mathcal{P}^-(\mathtt{TE})_{\mathtt{HProd}}$:

▶ **Lemma 9** (Removing Arguments Preserves Naturality). *Consider a natural diagram operator O and an operator $O'(D) := O(D) \setminus P_D$ for some heuristic P. Then O' is natural as well.*

Type Preservation as a Logical Relation The remaining steps towards generating SProd are more complicated. The meta-theory for using logical relations to represent type preservation was already sketched in [28], but we have to make a substantial generalization to *partial* logical relations and extend those to diagram operators.

Because logical relations can be very difficult to wrap one's head around, we focus on the special case needed for softening although we have designed and implemented it for the much more general setting of [28]. Moreover, we advise readers to maintain the following intuitions while perusing the formal treatment below:

$$
\begin{array}{lll}
\overline{r}(c) & = r(c) \\
\overline{r}(x) & = \begin{cases} x^* & \text{if } x^* \text{ was declared when traversing into the binder of } x \\ \text{undefined} & \text{otherwise} \end{cases} \\
\overline{r}(\mathtt{type}) & = \lambda_{a:\mathtt{type}}\, a \to \mathtt{type} \\
\overline{r}(\Pi_{x:A}\, B) & = \lambda_{f:m(\Pi_{x:A}\, B)}\, \Pi_{\overline{r}(x:A)}\, \overline{r}(B)\,(f\, x) \\
\overline{r}(\lambda_{x:A}\, t) & = \lambda_{\overline{r}(x:A)}\, \overline{r}(t) \\
\overline{r}(f\, t) & = \begin{cases} \overline{r}(f)\, m(t)\, \overline{r}(t) & \text{if } \overline{r}(t) \text{defined} \\ \overline{r}(f)\, m(t) & \text{otherwise} \end{cases} \\
\overline{r}(\cdot) & = \cdot \\
\overline{r}(\Gamma, x : A) & = \overline{r}(\Gamma), \begin{cases} x : m(A),\ x^* : \overline{r}(A)\, x & \text{if } \overline{r}(A) \text{defined} \\ x : m(A) & \text{otherwise} \end{cases}
\end{array}
$$

$\overline{r}(-)$ is undefined whenever an expression on the right-hand side is.

Figure 2 Map induced by a Logical Relation

- The morphism $m : S \to T$ is the type erasure translation $\mathtt{TE} : \mathtt{HTyped} \to \mathtt{STyped}$.
- The logical relation r is a mapping $\mathtt{TP}$ from $\mathtt{HTyped}$-syntax to $\mathtt{STyped}$-syntax that maps
  - types $A : \mathtt{type}$ to unary predicates $\mathtt{TP}(A) : \mathtt{TE}(A) \to \mathtt{type}$ about $\mathtt{TE}$-translated terms of type A
  - terms $t : A : \mathtt{type}$ to proofs $\mathtt{TP}(t) : \mathtt{TP}(A)\, \mathtt{TE}(t)$ of the predicate associated with A
- Even more concretely,
  - $\mathtt{TP}(\mathtt{tp})$ is undefined because we need not prove anything about $A : \mathtt{tp}$,
  - $\mathtt{TP}(\mathtt{tm}) = \lambda_{a:\mathtt{tp}}\, \lambda_{x:\mathtt{term}}\, \mathtt{of}\, x\, a$ and thus $\mathtt{TP}(\mathtt{tm}\, A) = \lambda_{x:\mathtt{term}}\, \mathtt{of}\, x\, A$, i.e., $\mathtt{TP}$ maps every $t : \mathtt{tm}\, A$ to its typing proof $\mathtt{TP}(t) : \mathtt{of}\ \mathtt{TE}(t)\, A$.

Moreover, it may help readers to compare Def. 2 and 10 as well as Thm. 3 and 11.

▶ **Definition 10.** *A partial* ***logical relation*** *on a morphism $m : S \to T$ is a partial mapping r of S-constants to T-expressions such that for every S-constant $c : A$, if $r(c)$ is defined, then so is $\overline{r}(A)$ and $\vdash_T r(c) : \overline{r}(A)\, m(c)$. r is called* ***term-total*** *if it is defined for a typed constant if it is for the type. The partial mapping $\overline{r}$ of S-syntax to T-syntax is defined in Fig. 2. In the sequel, we write r for $\overline{r}$.*

▶ **Theorem 11.** *For a partial logical relation r on a morphism $m : S \to T$, we have*
- *if $\Gamma \vdash_S t : A$ and r is defined for t, then r is defined for A and $r(\Gamma) \vdash_T r(t) : r(A)\, m(t)$*
- *if r is term-total, it is defined for a typed term if it is for its type*

It is straightforward to extend Def. 10 to all theories extending S in the same way as pushout extends a morphism. That would yield an include- and definition-preserving natural diagram operator. However, we omit that here because that functor would work with $\mathcal{P}(m)$ whereas we want to use $\mathcal{P}^-(m)$. Instead, we make a small adjustment similar how we obtained $\mathcal{P}^-(m)$ from $\mathcal{P}(m)$:

▶ **Definition 12.** *Consider a morphism $m : S \to T$ and a term-total logical relation r on m. Then the diagram operator $\mathcal{LR}(m, r)$ from S to T maps a diagram D as follows:*
1. *We compute $D' := \mathcal{P}^-(m)$.*
2. *Due to Lem. 9, D' has the same shape as D and for every theory E in D, there is a morphism $m_E : E \to E^m$. For each, we create an initially empty logical relation r_E on m_E.*

3. *We iterate over all declarations in all theories E in D and make the following modifications to D': for each declaration $c : A[= t]$ for which $r_E(A)$ is defined, we add*
   a. *the constant declaration $c^* : r_E(A)\, m(c)[= r_E(t)]$ to E^m*
   b. *the case $r(c) = c^*$ to r_E.*

▶ **Theorem 13.** *In the situation of Def. 12, the operator $\mathcal{LR}(m, r)$ is a natural diagram operator that preserves includes and definitions. And every r_E is a term-total logical relation on m_E.*

Now the operator $\mathcal{LR}(\mathtt{TE}, \mathtt{TP})$ generates for every hard-typed feature F

- the corresponding soft-typed feature F'
- the type-erasure translation $\mathtt{TE}_F : F \to F'$ as a compositional/homomorphic mapping,
- the type preservation proof $\mathtt{TP}_F$ for the type erasure as a logical relation on $\mathtt{TE}_F$.

In particular, we have $\mathtt{SProd} = \mathcal{LR}(\mathtt{TE}, \mathtt{TP})(\mathtt{HProd})$.

The Softening Operator We omitted the reduction rules in our running example `HProd`. This was because $\mathcal{LR}(\mathtt{TE}, \mathtt{TP})$ is still not the right operator. To see what goes wrong, assume we leave $\mathtt{TP}(\mathtt{ded})$ undefined, and consider the type of the `beta` rule from `HSimpFun`:

`HSimpFun`	$\Pi_{a,b}\, \Pi_{F:\mathtt{tm}\, a \to \mathtt{tm}\, b}\, \Pi_x\, \mathtt{ded}\, \mathtt{eq}\, b\, (\mathtt{app}\, (\mathtt{lam}\, F)\, x)\, (F\, x)$
$\mathtt{HSimpFun}^{\mathcal{LR}(\mathtt{TE},\mathtt{TP})}$ (generated)	$\Pi_{a,b}\, \Pi_{F:\mathtt{term} \to \mathtt{term}}\, \Pi_x$ $\mathtt{ded}\, \mathtt{eq}\, (\mathtt{app}\, (\mathtt{lam}\, F)\, x)\, (F\, x)$
`SSimpFun` (needed)	$\Pi_{a,b}\, \Pi_{F:\mathtt{term} \to \mathtt{term}}\, \Pi_{F^*:\Pi_a\, \mathtt{ded}\, \mathtt{of}\, x\, a \to \mathtt{ded}\, \mathtt{of}\, (F\, x)\, b}\, \Pi_x\, \Pi_{x^*:\mathtt{ded}\, \mathtt{of}\, x\, a}$ $\mathtt{ded}\, \mathtt{eq}\, (\mathtt{app}\, (\mathtt{lam}\, F)\, x)\, (F\, x)$

The rule generated by $\mathcal{LR}(\mathtt{TE}, \mathtt{TP})(\mathtt{HProd})$ is well-typed but not sound. In general, the softening operator must insert *-ed assumptions for all variables akin to how Def. 12 inserts them for constants. But it must only do so for proof rules and not for, e.g., `fun`, `lam`, and `app`.

We can achieve that by generalizing to partial logical relations on *partial* morphisms. Intuitively, we define $\mathcal{PLR}(m, r)$ for partial m and r in the same way as $\mathcal{LR}(m, r)$, again dropping all variable and constant declarations for whose type the translation is partial.

First we refine `TE` and `TP` as follows:

- We leave $\mathtt{TE}(\mathtt{ded})$ undefined, i.e., our morphisms do not translate proofs. That is to be expected because we know that `TE` cannot be extended to a morphism that also translates proofs [22].
- We put $\mathtt{TP}(\mathtt{ded}) = \lambda_{p:\mathtt{prop}}\, \lambda_{d:\mathtt{ded}\, p}\, \mathtt{ded}\, p$ and thus $\mathtt{TP}(\mathtt{ded}\, P)\, d = \mathtt{ded}\, \mathtt{TE}(P)$ for all P. This trick that has the effect that $\mathtt{beta}^*$ is generated as well and has the needed type (whereas the generation of `beta` can be suppressed).

Then we finally define $\mathtt{Soften} = \mathcal{PLR}(\mathtt{TE}, \mathtt{TP})$. For every proof rules c over `HTyped`, it

- drops the declaration of c,
- generates the declaration of c^*, which now has the needed type.

`Soften` is still include- and definition-preserving but is no longer natural. We conjecture that it is lax-natural and captures proof translations as a lax morphism in the sense of [22].

4.3 Natively Soft-Typed Features

Not all soft-typed features arise by translation from hard-typed features. Features that use subtyping such as union types and set theory–inspired features such as power types are usually present only in systems with, and can be formalized much more elegantly relative to soft typing. Often these only introduce new types but no new term, and instead give typing rules that check existing terms against the new types. We only give subtyping and predicate subtypes as examples.

```
theory SSubtyping =
  include STyped
  sub  : tp → tp → prop
  subI : Π_{a,b} (Π_x ded of x a → ded of x b)
           → ded sub a b
  subE : Π_{a,b,x} ded sub a b → ded of x a
           → ded of x b
```

```
theory SPredSub =
  include STyped
  ps   : tp → (term → prop) → tp
  psI  : Π_{a,P,x} ded of x a → ded P x
           → ded of x (ps a P)
  psEl : Π_{a,P,x} ded of x (ps a P) → ded of x a
  psEr : Π_{a,P,x} ded of x (ps a P) → ded P x
```

Like all features these are orthogonal to the concept mergers from Sect. 3 and thus can be reused equally easily in type theories and set theories. For example, if we combine `SSubtyping` with `TyAsTe`, the dependency on `STyped` is identified with the realization given in `TyAsTe` and `sub` yields the usual ⊆ predicate of set theory.

5 Logical Features

While propositional features depend only on `Props`, most features use terms and can be formalized relative to an untyped, hard-typed, or soft-typed base language. For most untyped features, we can systematically generate the corresponding hard-typed one and from that obtain the corresponding soft-typed one.

5.1 Propositional Logic

```
theory Conj =
  include Proofs
  conj      : prop → prop → prop
  conjIntro : Π_{F,G} ded F → ded G
                → ded conj F G
  ...

morph CHConj : Conj → HProd = {
  include PrAsTy
  conj      = prod
  conjIntro = pair
  ...
```

```
theory SCConj =
  include Proofs
  scconj      : Π_{F:prop} (ded F → prop)
                  → prop
  scconjIntro : Π_{F,G} Π_{p:ded F} ded G p
                  → ded scconj F G
  realize Conj
  conj        = λ_{F,G} scconj F (λ_p G)
  ...
```

Except for modal and other logics with more complex semantics, the formalization of propositional features is routine, e.g., `Conj` for conjunction. All features can be combined the concept mergers for propositions. However, when using propositions-as-types, we usually do not include `Conj` and instead define in terms of `HProd` via a realization or morphism, here called `CHConj`.

Some formal systems use proof-carrying terms, e.g., proofs may occur in terms to track the well-definedness of a term. In that case, we the short-circuiting variants of connectives. For example, `SCConj` generalizes `Conj` such that the second conjunct may depend on the truth of the first.

5.2 Untyped Logic

Untyped features depend on `UTyped` and are mostly used in standard first-order logic. Equality `UEqual` and quantifiers such as `UUniv` are routine examples.

Less well-known is the theory `UNonempty`, which is often present even when *users* of the formal language are not aware of it. It is equivalent to stating that the universe is non-empty. It is needed to formalize standard first-order logic, which may be surprising: textbook calculi usually imply nonempty through the backdoor by using variables that are not in scope. Because LF does not allow that, `nonempty` must be included explicitly if desired.

We give two variants for definite choice. `UDefChoice` produces a value even when the predicate is not uniquely satisfiable, such as `the`$[x]$`false` : `term`. To avoid inconsistency, any *use* of the chosen term is guarded by the condition that a unique value exists. (We omit the theory `UExistUnique` for the unique existential quantifier.) In `UDefChoiceGuarded` already the *construction* of the chosen term is guarded. Thus, only meaningful values can be constructed at the cost of introducing proof-carrying terms.

```
theory UEqual =
  include UTyped
  eq     : term → term → prop
  refl   : Π_x ded eq x x
  eqsub  : Π_{x,y} ded eq x y →
             Π_{P:term→prop} ded P x → ded P y

theory UUniv =
  include UTyped
  univ       : (term → prop) → prop
  univIntro  : Π_P (Π_{x:term} ded P x)
                 → ded univ P
  univElim   : Π_P ded univ P → Π_{x:term}
                 → ded P x
```

```
theory UNonempty =
  include UTyped
  nonempty : Π_P (term → ded p) → ded p

theory UDefChoice =
  include UExistUnique
  the    : (term → prop) → term
  theAx  : Π_P ded exU P → ded P (the P)

theory UDefChoiceGuarded =
  include UExistUnique
  the    : Π_{P:term→prop} ded exU P → term
  theAx  : Π_{P,d} ded P (the P d)
```

Many features are interrelated by morphisms. For example, we can give a morphism `UNonempty` → `UDefChoice` that shows that the unguarded choice operator already forces a non-empty universe.

5.3 Hard-Typed Logic

We use a diagram operator `Poly` from `UTyped` to `HTyped` by adapting the operator from [26] to our setting:

▶ **Definition 14** (Polymorphification). `Poly` *maps a* `UTyped` *diagram* D *to* `HTyped` *as follows:*

1. *We create a copy* D' *of* D *in which all theories and morphisms* X *are renamed to* $X^{\texttt{Poly}}$*.*
2. *We iterate through all declarations in* D *in dependency order and modify* D' *as follows:*
   - *replace every constant* $c : A[= t]$ *with* $c : \Pi_{u:\texttt{tp}} A^u [= \lambda_{u:\texttt{tp}} t^u]$ *for a fresh variable* u

- *replace every definition $c = t$ accordingly*

Here $e \mapsto e^u$ replaces every occurrence of `term` *with* $\texttt{tm}\,u$ *and every constant c that depends on* `term` *with $c\,u$.*

`Poly` is an include- and definition-preserving functor from D to $\texttt{Poly}(D)$. It is not natural but the compositional translation of all `UTyped`-syntax to `HTyped`-syntax can be captured via lax morphisms in the sense of [22]. Our implementation of `Poly` is in fact slightly more general than defined above: We also allow the diagram D to be non-closed, i.e., to contain some references to theories not in D. Those references are simply kept as is. That is a minor tweak of the definition but critical in practice, e.g., when some propositional features are included as well.

We can now aggregate any of our untyped features and morphisms between them into a single diagram, apply `Poly`, and obtain hard-typed variants in one go, e.g.,the declaration $\texttt{compute}\,\texttt{Poly}(\texttt{diagram}(\texttt{UEqual},\texttt{UUniv},\texttt{UNonempty}))$ yields the theories

$$
\begin{array}{l}
\texttt{theory HNonempty} = \\
\quad \texttt{include HTyped} \\
\quad \texttt{nonempty} : \Pi_{u,P}\,(\texttt{tm}\,u \to \texttt{ded}\,P) \to \texttt{ded}\,P \\[1ex]
\texttt{theory HUniv} = \\
\quad \texttt{include HTyped} \\
\quad \texttt{univ} \quad\;\; : \Pi_u\,(\texttt{tm}\,u \to \texttt{prop}) \to \texttt{prop} \\
\quad \texttt{univIntro} : \Pi_{u,P}\,(\Pi_{x:\texttt{tm}\,u}\,\texttt{ded}\,P\,x) \to \texttt{ded}\,\texttt{univ}\,u\,P \\
\quad \texttt{univElim} \;\, : \Pi_{u,P}\,\texttt{ded}\,\texttt{univ}\,u\,P \to \Pi_{x:\texttt{tm}\,u} \to \texttt{ded}\,P\,x
\end{array}
$$

$$
\begin{array}{l}
\texttt{theory UEqual}^{\texttt{Poly}} = \\
\quad \texttt{include HTyped} \\
\quad \texttt{eq} \quad\;\;\; : \Pi_u\,\texttt{tm}\,u \to \texttt{tm}\,u \to \texttt{prop} \\
\quad \texttt{refl} \quad\; : \Pi_{u,x}\,\texttt{ded}\,\texttt{eq}\,u\,x\,x \\
\quad \texttt{eqsub} : \Pi_{u,x,y}\,\texttt{ded}\,\texttt{eq}\,u\;x\,y \\
\qquad\qquad\;\; \Pi_{F:\texttt{tm}\;u\to\texttt{prop}}\,\texttt{ded}\,F\,x \to \texttt{ded}\,F\,y
\end{array}
$$

`Poly` does not always yield the intended result, and some hard-typed theories must still be hand-written. The only example we have encountered so far is that $\texttt{UEqual}^{\texttt{Poly}}$ is not the theory `HEqual`: the `eqsub` rule for equality contains multiple occurrences of `term` that need to be replaced two different type variables. It would be easy to do that, but we do have a good way to choose the desired number of type variables heuristically. Note that equality `eq` also has multiple occurrences of `term`, but here introducing only one type variable is the desired behavior.

5.4 Soft-Typed Logic

Finally, we compose `Poly` and `Soften` to obtain soft-typed variants of all features. We give the result of $\texttt{compute}\,\texttt{Poly}(\texttt{diagram}(\texttt{UUniv},\texttt{UNonempty}))$ below. Here only the non-gray parts are actually generated. The gray parts would be generated if we falsely defined `Soften` as $\mathcal{LR}(\texttt{TE},\texttt{TP})$ with $\texttt{TE}(\texttt{ded}) = \texttt{ded}$ instead of $\mathcal{PLR}(\texttt{TE},\texttt{TP})$ with $\texttt{TE}(\texttt{ded})$ undefined: One might argue that `SNonempty` is useless as it precludes the empty soft type, but that is, e.g., how soft types are handled in Mizar.

$$
\begin{array}{l}
\texttt{theory SNonempty} = \\
\quad \texttt{include STyped} \\
\quad \texttt{nonempty} \;\; : \Pi_{u,P}\,(\Pi_{x:\texttt{term}} \to \texttt{ded}\,P) \to \texttt{ded}\,P \\
\quad \texttt{nonempty}^* : \Pi_{u,P}\,(\Pi_x\,\texttt{ded}\,\texttt{of}\,x\,u \to \texttt{ded}\,P) \to \texttt{ded}\,P
\end{array}
$$

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathtt{theory\ SUniv}\ =} \\
\quad\mathtt{include\ STyped} & \\
\quad\mathtt{univ} & : \Pi_u\,(\mathtt{tm}\,u \to \mathtt{prop}) \to \mathtt{prop} \\
\quad\mathtt{univIntro} & : \Pi_{u,P}\,(\Pi_{x:\mathtt{term}}\,\mathtt{ded}\,P\,x) \to \mathtt{ded}\,\mathtt{univ}\,u\,P \\
\quad\mathtt{univIntro}^* & : \Pi_{u,P}\,\Pi_{d:\Pi_{x:\mathtt{term}}\,\mathtt{ded}\,P\,x}\,\Pi_{d^*:\Pi_{x:\mathtt{term}}\,\Pi_{x^*:\mathtt{ded\,of}\,x\,u}\,\mathtt{ded}\,P\,x}\,\mathtt{ded}\,\mathtt{univ}\,u\,P \\
\quad\mathtt{univElim} & : \Pi_{u,P}\,\mathtt{ded}\,\mathtt{univ}\,u\,P \to \Pi_{x:\mathtt{term}} \to \mathtt{ded}\,P\,x \\
\quad\mathtt{univElim}^* & : \Pi_{u,P}\,\mathtt{ded}\,\mathtt{univ}\,u\,P \to \Pi_{x:\mathtt{term}}\,\Pi_{x^*:\mathtt{of}\,x\,u} \to \mathtt{ded}\,P\,x
\end{array}
$$

The composition of `Soften` and `Poly` also induces a proof translation that maps all proofs `UTyped` to `STyped`. If, as we conjecture, both `Poly` and `Soften` are lax-natural, then that translation also enjoys strong invariants such as commuting with substitutions.

6 Conclusion and Future Work

Overview We have given an overview of the LATIN2 library of highly modular formalizations of formal systems. Following the little theories methodology, we state language as small modules that can be combined flexibly. Contrary to precursor projects by ourselves and others, LATIN2 is based on the MMT/LF system, which has been developed in response to this eact application, thus enabling particularly elegant formalizations.

We presented representative individual formalizations. The entire library is much larger and is a growing long-term project. The ultimate goal of LATIN2 is to build a reference library of all formal systems and their interrelations. Additionally, we presented meta-operations that generate formalizations systematically. That is critical for a large library in order to ensure uniform naming and structuring across variant formalizations of the same features. Importantly, they preserve modular structure so that large diagrams of interrelated theories can be generated easily. While we only applied them to translate between un/soft/hard-typed languages, we kept the definitions so general that they are widely applicable beyond LATIN2.

Future Work We expect that applying `Poly` to soft-typed features yields the corresponding *semi-soft*-typed one. These combine a hard typing with a lattice of soft subtypes for each hard type. This is used in PVS natively and in many systems via types-as-propositions such as with Isabelle/HOL's set types.

Universes are critical for many dependent type theories like Coq. But they can make the formalizations more complex, inclusive hierarchies particularly so. We believe more work is necessary to study the various self-contained formalizations that have been done by the community before attacking highly modular ones.

We call types like inductive and record types *theory-like* because they can be thought of as given by a theory declaring the constructors resp. fields. Logical frameworks are generally weak at those, and we are using MMT to experiment with extensions of LF that allow for elegant formalizations.

At the level of the module system, we are investigating how to represent *cross-cutting features*. These are features that tend to require one base theory and then one theory for every feature, e.g., `SSubtyping`, a theory of subtyping for product types, etc. Other cross-cutting features are equality, undefinedness, and classical reasoning. It is straightforward to write all the theories, but the resulting multi-dimensional diagram tends to get too complicated to navigate practically. Users sometimes need a theory hierarchy that groups, e.g., the subtyping rules for all features, and sometimes one that groups, e.g., all rules relating to product types.

References

1 A. Avron, F. Honsell, I. Mason, and R. Pollack. Using typed lambda calculus to implement formal systems on a machine. *Journal of Automated Reasoning*, 9(3):309–354, 1992.

2 A. Avron, F. Honsell, M. Miculan, and C. Paravano. Encoding modal logics in logical frameworks. *Studia Logica*, 60(1):161–208, 1998.

3 M. Boespflug, Q. Carbonneaux, and O. Hermant. The λΠ-calculus modulo as a universal proof language. In D. Pichardie and T. Weber, editors, *Proceedings of PxTP2012: Proof Exchange for Theorem Proving*, pages 28–43, 2012.

4 M. Codescu, F. Horozal, M. Kohlhase, T. Mossakowski, and F. Rabe. Project Abstract: Logic Atlas and Integrator (LATIN). In J. Davenport, W. Farmer, F. Rabe, and J. Urban, editors, *Intelligent Computer Mathematics*, pages 289–291. Springer, 2011.

5 N. de Bruijn. The Mathematical Language AUTOMATH. In M. Laudet, editor, *Proceedings of the Symposium on Automated Demonstration*, volume 25 of *Lecture Notes in Mathematics*, pages 29–61. Springer, 1970.

6 W. Farmer, J. Guttman, and F. Thayer. Little Theories. In D. Kapur, editor, *Conference on Automated Deduction*, pages 467–581, 1992.

7 F. Guidi, C. Sacerdoti Coen, and E. Tassi. Implementing Type Theory in Higher Order Constraint Logic Programming. *Mathematical Structures in Computer Science*, 29(8):1125–1150, 2019.

8 R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, 1993.

9 R. Harper, D. Sannella, and A. Tarlecki. Structured presentations and logic representations. *Annals of Pure and Applied Logic*, 67:113–160, 1994.

10 F. Horozal and F. Rabe. Formal Logic Definitions for Interchange Languages. In M. Kerber, J. Carette, C. Kaliszyk, F. Rabe, and V. Sorge, editors, *Intelligent Computer Mathematics*, pages 171–186. Springer, 2015.

11 M. Iancu and F. Rabe. Formalizing Foundations of Mathematics. *Mathematical Structures in Computer Science*, 21(4):883–911, 2011.

12 M. Iancu and F. Rabe. (Work-in-Progress) An MMT-Based User-Interface. In C. Kaliszyk and C. Lüth, editors, *Workshop on User Interfaces for Theorem Provers*, 2012.

13 M. Kohlhase and F. Rabe. Experiences from Exporting Major Proof Assistant Libraries. see `https://kwarc.info/people/frabe/Research/KR_oafexp_20.pdf`, 2020.

14 D. Miller and G. Nadathur. Higher-order logic programming. In E. Shapiro, editor, *Proceedings of the Third International Conference on Logic Programming*, pages 448–462. Springer, 1986.

15 T. Mossakowski, S. Autexier, and D. Hutter. Development graphs - Proof management for structured specifications. *J. Log. Algebr. Program*, 67(1–2):114–145, 2006.

16 P. Naumov, M. Stehr, and J. Meseguer. The HOL/NuPRL proof translator - a practical approach to formal interoperability. In R. Boulton and P. Jackson, editors, *14th International Conference on Theorem Proving in Higher Order Logics*. Springer, 2001.

17 B. Woltzenlogel Paleo and G. Reis, editors. *An Encyclopaedia of Proof Systems: Second Edition*. College Publications, 2018.

18 L. Paulson. The Foundation of a Generic Theorem Prover. *Journal of Automated Reasoning*, 5(3):363–397, 1989.

19 L. Paulson. Isabelle: The Next 700 Theorem Provers. In P. Odifreddi, editor, *Logic and Computer Science*, pages 361–386. Academic Press, 1990.

20 F. Pfenning. Structural cut elimination: I. intuitionistic and classical logic. *Information and Computation*, 157(1-2):84–141, 2000.

21 F. Pfenning, C. Schürmann, M. Kohlhase, N. Shankar, and S. Owre. The Logosphere Project, 2003. `http://www.logosphere.org/`.

22 F. Rabe. Lax Theory Morphisms. *ACM Transactions on Computational Logic*, 17(1), 2015.

23 F. Rabe. How to Identify, Translate, and Combine Logics? *Journal of Logic and Computation*, 27(6):1753–1798, 2017.

24 F. Rabe. A Modular Type Reconstruction Algorithm. *ACM Transactions on Computational Logic*, 19(4):1–43, 2018.

25 F. Rabe and M. Kohlhase. A Scalable Module System. *Information and Computation*, 230(1):1–54, 2013.

26 F. Rabe and N. Roux. Structure-Preserving Diagram Operators. In A. Corradini, M. Huisman, A. Knapp, and M. Roggenbach, editors, *Recent Trends in Algebraic Development Techniques*. Springer, 2020. to appear.

27 F. Rabe and C. Schürmann. A Practical Module System for LF. In J. Cheney and A. Felty, editors, *Proceedings of the Workshop on Logical Frameworks: Meta-Theory and Practice (LFMTP)*, pages 40–48. ACM Press, 2009.

28 F. Rabe and K. Sojakova. Logical Relations for a Logical Framework. *ACM Transactions on Computational Logic*, 14(4):1–34, 2013.

29 C. Schürmann and F. Pfenning. Automated theorem proving in a simple meta-logic for LF. In C. Kirchner and H. Kirchner, editors, *Proceedings of the 15th International Conference on Automated Deduction*, pages 286–300. Springer, 1996.

30 Y. Sharoda and F. Rabe. Diagram Operators in MMT. In C. Kaliszyk, E. Brady, A. Kohlhase, and C. Sacerdoti Coen, editors, *Intelligent Computer Mathematics*, pages 211–226. Springer, 2019.

31 F. Thiré. Sharing a library between proof assistants: Reaching out to the hol family. In F. Blanqui and G. Reis, editors, *Logical Frameworks and Meta-Languages: Theory and Practice*, pages 57–71. EPTCS, 2018.

32 Y. Wang, K. Chaudhuri, A. Gacek, and G. Nadathur. Reasoning about higher-order relational specifications. In R. Peña and T. Schrijvers, editors, *Practice of Declarative Programming*, pages 157–168, 2013.

A Proofs

Proof of 7

Proof. Technically, this is proved by induction on the typing derivation of D. But it is easy to see: by construction, (i) the variables bindings in P do not occur in $D \setminus P$ so that all types and definitions stay well-typed, and (ii) the type, definitions, and uses of all constant are changed consistently so that they stay well-typed. The only subtlety is that we need to apply LF's η-equality to expand not fully applied uses of a constant. ◀

Proof of 9

Proof. O being natural yields morphisms $O_E : E \to E^O$ from D-theories to $O(D)$ theories. $O(D')$ has the same shape as $O(D)$, and to show that $O'(D)$ is natural, we reuse essentially the same morphisms from D-theories to $O'(D)$-theories. We only have to η-expand the right-hand sides of all assignments in the morphisms O_E and remove the same argument positions in P_D as well. ◀

Proof of 11

Proof. The inductive definition is the same as in [28] except for the possibility of undefinedness. Thus, whenever the results are defined, the typing properties follow from the theorems there.

First, it is straightforward to see that r is total on contexts and substitutions because the case distinctions explicitly avoid recursing into arguments for which r is undefined.

Second, we show by induction on derivations of $\Gamma \vdash_S t : A$ that if $A : \mathtt{type}$ then r is defined for t iff it is defined for A.

- constant $c : A$: True by assumption.

- variable $x : A$: The case for $\Gamma, x : A$ introduces the variable x^* into the target context if $r(A)$ is defined. The case for x picks up on that and (un)defines r at x accordingly.
- λ-abstraction $\lambda_{x:A}\, t : \Pi_{x:A}\, B$: r is always defined for $x : A$. By induction hypothesis, it is defined for t if it is for B.
- t cannot be a Π-abstraction
- application $f\, t : B(t)$ for some $f : \Pi_{x:A}\, B(x)$: By definition, r is defined for $f\, t$ if it is defined for f. By induction hypothesis the latter holds iff r is defined for $\Pi_{x:A}\, B(x)$, which by definition holds iff it is defined for $B(x)$. It remains to show that r is defined for $B(t)$ iff it is defined for $B(x)$ in the context extended with $x : A$. By induction hypothesis, r is defined for t iff it is defined for x. Therefore, and because the definition of r is compositional, substituting t for x cannot affect whether r is defined for an expression.

Finally, if $\Gamma \vdash_E t : A$ for $A : \texttt{kind}$, we need to show that r is defined for A if it is for t. That is trivial: inspecting the definition shows that r is always defined for kinds anyway. ◀

Proof of 13

Proof. We already know that $\mathcal{P}^-(-)$ has the desired properties. Moreover, adding well-typed declarations to $\mathcal{P}^-(m)$ does not affect the naturality (because adding declaration to the codomain never affects the well-typedness of a morphism). So for the first claim, we only have to proof that our additions are well-typed.

We prove that and the fact that r_E is a logical relation jointly by induction on the derivation of the well-typedness of D: He appeal to Thm. 11 to show that the added constant declarations are well-typed. And the cases $r(c) = c^*$ satisfy the typing requirements of logical relations by construction. ◀