Systematic Translation of Formalizations of Type Theory from Intrinsic to Extrinsic Style

Florian Rabe          Navid Roux
University Erlangen-Nuremberg
Department of Computer Science
University Erlangen-Nuremberg
Erlangen, Germany
florian.rabe@fau.de    navid.roux@fau.de

Type theories can be formalized using the intrinsically (hard) or the extrinsically (soft) typed style. In large libraries of type theoretical features, often both styles are present, which can lead to code duplication and integration issues.

We define an operator that systematically translates a hard-typed into the corresponding soft-typed formulation. Even though this translation is known in principle, a number of subtleties make it more difficult than naively expected. Importantly, our translation preserves modularity, i.e., it maps structured sets of hard-typed features to correspondingly structured soft-typed ones.

We implement our operator in the MMT system and apply it to a library of type-theoretical features.

1 Introduction

Motivation and Related Work  Soft type theory goes back to Curry’s work \cite{curry}, where typing is a meta-language (ML) predicate \( \text{term} \to \text{tp} \to \text{prop} \) between object-language (OL) terms and types. This is also called extrinsic typing. This leads to a deep embedding of typing where the OL judgment \( \text{t : A} \) corresponds to the existence of a typing proof, i.e., an ML term witnessing \( \text{tm : t A} \).

Hard type theory goes back to Church’s work \cite{church}, where typing is a function from terms to types. If that function is sufficiently simple, formalizations in a dependently-typed ML like LF \cite{lf} may be able to capture it directly in the framework’s type system. Concretely, such a representation uses a shallow embedding of typing where an object-language typing judgment \( \text{t : A} \) corresponds to the meta-language (ML) typing judgment \( \text{tm : t A} \). Only well-formed terms can be encoded at, and the OL type of a term can be read off of its ML type. This is also called intrinsic typing. If ML type-checking is decidable, that means OL must be as well.

Soft typing is more expressive and flexible than hard typing. But

- introducing an OL variable \( x \) of type \( a \) requires two ML variables of \( \text{term} \), \( x^* : \text{of A} \),
- correspondingly, substituting a term for a variable requires a proof of its typing judgment,
- type-checking is reduced to ML non-emptiness-checking, which is usually undecidable.

In the LATIN project \cite{latin} going back to ideas developed in the Logosphere project \cite{logosphere}, we built a highly modular library of formalizations of logics and type theories. The goal was to create a library of Little Logics (in the style of Little Theories \cite{lt}), each formalizing one language feature such as product types, which can be combined to form concrete systems. This allows the reuse, translation, and combination of formalizations in the style of \cite{reusability,translation}. 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 [9, 10].

Due to the incomparable advantages of soft and hard encodings, we had to formalize each feature in both styles. This led not only to a duplication of code but also caused significant maintenance problems. In particular, it is difficult to ensure coherent encoding styles (e.g., naming conventions, order of arguments, notations, etc.) in such a way that the two sets of encodings are related systematically.

For every pair of a hard- and a soft-typed encoding of the same language feature, there is a type erasure translation from the former to the latter with an associated type preservation property. Such translations have been investigated in various forms, see e.g., [1] for a systematic study in the form of realizability theories. We will cast the type preservation as a logical relation proof as formalized in an LF-based logical framework in [16]. Notably, given the hard-typed encoding, it is possible to derive the soft-typed one, the erasure translation, and the preservation proof automatically. We call this derivation softening. Systematic softening not only greatly reduces the encoding effort but simplifies maintenance and produces more elegant code.

Contribution and Overview We define an operator $\textsf{Soften}$ in the logical framework LF [6]. Despite being conceptually straightforward, softening is a rather complex process, and an ad-hoc implementation, while possible, would be error-prone and hard to maintain. Therefore, we employ a systematic approach for deriving the softening operator that constructs the logical relation proof along with the softened theory. A particular subtlety was to ensure the generated code to still be human-readable. That required softening to consider pragmatic aspects like notations and choice of implicit arguments.

Our work is carried out under the LATIN2 header, which aims at a complete reimplementation of the LATIN library. While LATIN worked with modular Twelf [15], LATIN2 uses the MMT/LF incarnation of LF [13]. In addition to an implementation of LF and a module system, MMT provides a framework for diagram operators [14], which supports the meta-theory and implementation of operators that systematically derive formalizations from one another. It also makes it easy to annotate declarations, which we will use to guide the softening operator in a few places.

Importantly, these diagram operators are functorial in the category of LF theories and theory morphisms. That enables scaling them up to entire libraries in a way that preserves modularity. That is important to derive human-readable formalizations.

Sect. 2 introduces MMT/LF. Sect. 3 shows the key definition of the softening operator, and Sect. 4 establishes meta-theoretical properties that allow lifting it to libraries. Sect. 5 shortly sketches our implementation in the MMT system.

2 The MMT Framework and Basic Formalizations

MMT [13] 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., [6]) and only recap the notions of theories and morphisms that MMT adds on top.
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 1.** 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 \( \overline{m} \).
constants of $S$  
\[ \overline{m}(c) = m(c) \]
other expressions  
\[ \overline{m}(x) = x \]
\[ \overline{m}(\text{type}) = \text{type} \]
\[ \overline{m}(\Pi_{x:A} B) = \Pi_{x:\overline{m}(A)} \overline{m}(B) \]
\[ \overline{m}(\lambda_{x:A} t) = \lambda_{x:\overline{m}(A)} \overline{m}(t) \]
\[ \overline{m}(f t) = \overline{m}(f) \overline{m}(t) \]
contexts  
\[ \overline{m}(\cdot) = \cdot \]
\[ \overline{m}(\Gamma, x : A) = \overline{m}(\Gamma), x : m(A) \]

Figure 1: Map induced by a Morphism

**Theorem 1.** For a morphism $m : S \rightarrow T$ and a theory $E$ that includes $S$, if $\Gamma \vdash_E t : A$, then $m(\Gamma) \vdash_{E^m} m(t) : m(A)$. In particular for $E = S$, we have $m(\Gamma) \vdash_T m(t) : m(A)$.

In terms of category theory, a morphism $m$ induces a pushout functor $P(m)$ 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 them is mapped to a corresponding diagram of theories $E^m$ including $T$. Moreover, for each $E$, $m$ extends to a morphism $E \rightarrow E^m$ 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. These morphisms form a natural transformation, and we speak of a natural functor.

\[
\text{morph } T_E : \text{HTyped} \rightarrow \text{STyped} = \\
\quad \text{include Proofs} \\
\quad \text{tp} = \text{tp} \\
\quad \text{tm} = \lambda_{a:\text{tp}} \text{term}
\]

\[
\text{theory } HProd = \\
\quad \text{include HTyped} \\
\quad \text{prod} : \text{tp} \rightarrow \text{tp} \rightarrow \text{tp} \\
\quad \text{pair} : \Pi_{a,b} \text{term} \rightarrow \text{term} \rightarrow \text{term} \\
\quad \text{projL} : \Pi_{a,b} \text{term} \rightarrow \text{term} \\
\quad \text{projR} : \Pi_{a,b} \text{term} \rightarrow \text{term}
\]

\[
\text{theory } HProd^{TE} = \\
\quad \text{include STyped} \\
\quad \text{prod} : \text{tp} \rightarrow \text{tp} \rightarrow \text{tp} \\
\quad \text{pair} : \Pi_{a,b} \text{term} \rightarrow \text{term} \rightarrow \text{term} \\
\quad \text{projL} = \text{projL} \\
\quad \text{projR} = \text{projR}
\]

Figure 2: Pushout along the Type Erasure Morphism

**Example 2** (related to Fig. 2). The type erasure translation $T_E : \text{HTyped} \rightarrow \text{STyped}$ maps types $a : \text{tp}$ to types $TE(a) : \text{tp}$, which we formalize by $\text{tp} = \text{tp}$. And it maps typed terms $t : tma$ to untyped...
terms $\text{TE}(t) : \text{term}$, which we formalize by $\text{tm} = \lambda \text{tp} \text{term}$ and thus $\text{TE}(\text{tm}a) = \text{term}$. We also use include $\text{Proofs}$ to include the identity morphism on $\text{Proofs}$, i.e., all constants of $\text{Proofs}$ are mapped to themselves.

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

However, $\text{HProd}^{\text{TE}}$ is not the desired formalization of soft-typed products (e.g., because it lacks constants relating types and terms), and we develop a more suitable functor in the next section.

### 3 The Softening Operator

#### 3.1 Basic Overview

Soften translates theories based on $\text{HTyped}$ to theories based on $\text{STyped}$. The key idea is that whenever we have an expression $t : \text{tm}a$ in $\text{HTyped}$, then in $\text{STyped}$ we need to synthesize two things: an expression $\text{TE}(t) : \text{term}$ and an expression $t^* : \text{of} \text{TE}(t) \text{TE}(a)$ acting as a witness of type preservation. And whenever we have an expression $a : \text{tp}$, we need to synthesize one thing only, namely $\text{TE}(a) : \text{tp}$. (Note that for simple types such as product and function types, we have $\text{TE}(a) = a$. We discuss dependent function types in the next section.) Both intuitions extend homomorphically to all concepts of LF such as function types and contexts.

As an example, consider the constants $\text{pair}, \text{projL}, \text{projR}$ in $\text{HProd}$ in Fig. 3. For each of them we synthesize a type-erased constant of the same name and a starred typing witness in $\text{STyped}$. Note that the type parameters $a$ and $b$ are removed in their corresponding type-erased constant in $\text{SProd}$. We have been unable to find a systematic way to determine when arguments need to be removed and discuss this problem in Sec. 3.3. For the arguments in the starred constants such as $\text{pair}^*$, we synthesize two parameters $x : \text{term}$ and $x^* : \text{dedof}xa$ (whose name can often be omitted).

```plaintext
theory HProd =
  include HTyped
  prod : tp → tp → tp
  pair : $\Pi_{a,b} \text{tm}a → \text{tm}b → \text{tmprod}ab$
  projL : $\Pi_{a,b} \text{tmprod}ab → \text{tm}a$
  projR : $\Pi_{a,b} \text{tmprod}ab → \text{tm}b$

theory SProd =
  include STyped
  prod : tp → tp → tp
  pair : $\Pi_{a,b} \text{term} → \text{term} → \text{term}$
  pair* : $\Pi_{a,b} \Pi_x \text{dedof}xa → \Pi_y \text{dedof}yb → \text{dedof}(\text{pair}xy)(\text{prod}ab)$
  projL : $\text{term} → \text{term}$
  projL* : $\Pi_{a,b} \Pi_x \text{dedof}x(\text{prod}ab) → \text{dedof}(\text{projL}x)a$
  projR : $\text{term} → \text{term}$
  projR* : $\Pi_{a,b} \Pi_y \text{dedof}x(\text{prod}ab) → \text{dedof}(\text{projR}x)b$
```

Figure 3: Hard and Soft Product Types
3.2 Logical Relations

We capture the type preservation proof using logical relations. The meta-theory for using logical relations to represent type preservation was already sketched in [16], but we have to make a substantial generalization to partial logical relations and extend those to natural functors. Besides allowing the representation of partial translations, this has an important practical advantage: the translations in [16] must introduce unit argument types in places where no particular property about a term is proved. While semantically irrelevant, softening must remove these in order to produce the softened theories actually expected by humans, thus potentially violating the correctness of the translation. Partiality allows constructing the softened theories in a way that these artefacts are not introduced in the first place.

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 [16]. Moreover, we advise readers to maintain the following intuitions while perusing the formal treatment below:

- The morphism \( m : S \to T \) is the type erasure translation \( \text{TE} : \text{HTyped} \to \text{STyped} \).

- The logical relation \( r \) is a mapping \( TP \) from \( \text{HTyped-synt} \) to \( \text{STyped-synt} \) that maps
  - types \( A : \text{type} \) to unary predicates \( TP(A) : \text{TE}(A) \to \text{type} \) about \( \text{TE} \)-translated terms of type \( A \)
  - terms \( t : A : \text{type} \) to proofs \( TP(t) : TP(A) \text{TE}(t) \) of the predicate associated with \( A \)

- Even more concretely,
  - \( TP(\text{prop}), TP(\text{ded}), TP(\text{tp}) \) are all undefined because we need not prove anything about terms at those types
  - \( TP(\text{tm}) = \lambda_{x: \text{tp}} \lambda_{x: \text{term}} \text{of} x a \) and thus \( TP(\text{tm}a) = \lambda_{x: \text{term}} \text{of} x a \), i.e., TP maps every \( t : \text{tm}a \) to its typing proof \( TP(t) : \text{of} \text{TE}(t)a \).

Moreover, it may help readers to compare Def. 1 and 2 as well as Thm. 1 and 2.

**Definition 2.** 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 \( \overline{T}(A) \) and \( \vdash_T r(c) : \overline{T}(A) m(c) \). \( r \) is called term-total if it is defined for a typed constant if it is for the type. The partial mapping \( \overline{T} \) of \( S \)-synt to \( T \)-synt is defined in Fig. 4. In the sequel, we write \( r \) for \( \overline{T} \).

The key idea of the map \( \overline{T} \) is to attempt to construct \( \overline{T} \) in the same way as in [16] for total \( r \). Whenever \( \overline{T} \) is applied to an argument for which it is not defined, the expression is simply removed: if the type of a bound variable would be undefined, the whole binding is removed; if an argument of a function application would be undefined, the function is applied to one fewer argument. The next theorem states that these removals fit together in the sense that \( \overline{T} \) still satisfies the main property of logical relations whenever it is defined:

**Theorem 2.** 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

**Proof.** The inductive definition is the same as in [16] 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 \( A : \text{type} \) then \( r \) is defined for \( t \) iff it is defined for \( A \).

- constant \( c : A \): True by assumption.
\[\bar{r}(c) = r(c)\]
\[\bar{r}(x) = \begin{cases} x^* & \text{if } x^* \text{ was declared when traversing into the binder of } x \\ \text{undefined} & \text{otherwise} \end{cases}\]
\[\bar{r}(\text{type}) = \lambda_{a:\text{type}} a \to \text{type}\]
\[\bar{r}(\Pi_{c:A} B) = \lambda_{f:m(\Pi_{c:A} B)} \Pi_{\bar{r}(A)} (f x)\]
\[\bar{r}(\lambda_{c:A} t) = \lambda_{\bar{r}(x:A)} \bar{r}(t)\]
\[\bar{r}(f t) = \begin{cases} \bar{r}(f) m(t) \bar{r}(t) & \text{if } \bar{r}(t) \text{ defined} \\ \bar{r}(f) m(t) & \text{otherwise} \end{cases}\]
\[\bar{r}(\cdot) = \cdot\]
\[\bar{r}(\Gamma, x : A) = \bar{r}(\Gamma), \begin{cases} x : m(A), x^* : \bar{r}(A), x & \text{if } \bar{r}(A) \text{ defined} \\ x : m(A) & \text{otherwise} \end{cases}\]
\[\bar{r}(\cdot) \text{ is undefined whenever an expression on the right-hand side is.}\]

Figure 4: Map induced by a Logical Relation

- 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.
- \(\lambda\)-abstraction \(\lambda_{c:A} t : \Pi_{c:A} B\): \(r\) is always defined for \(x : A\). By induction hypothesis, it is defined for \(t\) if it is defined for \(B\).
- \(\tau\) cannot be a \(\Pi\)-abstraction
- application \(f t : B(t)\) for some \(f : \Pi_{c:A} B(x)\): 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 \(\Pi_{c:A} B(x)\), 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 \(A : \text{kind}\), 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.

We now capture how to synthesize \(S\text{prod}\) from \(H\text{prod}\) via logical relations. First we define the type erasure morphism \(TE : H\text{Typed} \to S\text{Typed}\) and the type preservation property as a logical relation \(TP\) on \(TE\). Then we define and apply the following functor on \(H\text{prod}\), which can be thought of as an analog of pushout along a logical relation:

**Definition 3.** Consider a morphism \(m : S \to T\) and a term-total logical relation \(r\) on \(m\). Then the functor \(LR(m, r)\) from \(S\)-extensions to \(T\)-extensions maps theories \(E\) as follows:

1. We compute the pushout \(E^m := \mathcal{P}(m)(E)\).
2. \(E^m\) has the same shape as \(E\) and 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. For each declaration \(c : A[= t]\) in \(E\) for which \(r_E(A)\) is defined, we add
   (a) the constant declaration \(c^* : r_E(A)m_E(c)[= r_E(t)]\) to \(E^m\)
   (b) the case \(r(c) = c^*\) to \(r_E\).

Concretely we get:
theory \mathcal{LR}(\text{TE}, \text{TP})(\text{HProd}) = \\
\quad \text{include STyped} \\
\quad \text{prod} : \text{tp} \rightarrow \text{tp} \rightarrow \text{tp} \\
\quad \text{pair} : \Pi_{i,b} \text{term} \rightarrow \text{term} \rightarrow \text{term} \\
\quad \text{pair}^* : \Pi_{i,b} \text{dedof}(\text{tx}) \rightarrow \text{Pi}, \text{dedof}(\text{ty}) \\
\quad \quad \quad \rightarrow \text{dedof}(\text{pair}(\text{tx})(\text{ty})) \\
\quad \text{projL} : \Pi_{i,b} \text{term} \rightarrow \text{term} \\
\quad \text{projL}^* : \Pi_{i,b} \text{dedof}(\text{tx}) \rightarrow \text{dedof}(\text{projL}(\text{tx}))a \\
\quad \text{projR} : \Pi_{i,b} \text{term} \rightarrow \text{term} \\
\quad \text{projR}^* : \Pi_{i,b} \text{dedof}(\text{tx}) \rightarrow \text{dedof}(\text{projR}(\text{tx}))b

Here we see that pair, projL, projR all take undesired (and unused) type arguments. In the sequel, we will suitably extend the naive definition given above.

Fig. 5 gives some additional examples of hard-typed features. Here we also include hard-typed equality HEqual to formulate the reduction rules for function types. Fig. 6 shows the corresponding soft-typed variants that we intend to obtain. Note that these examples already foreshadow that Soften can be extended to theories containing includes in a straightforward way. We will define that formally in Sect. 4.

3.3 Removal of Unnecessary Parameters

In Sect. 3.2 we developed a translation from \text{HTyped} to \text{STyped} that maps every constant \( c : A \) to a translated constant \( c : m(A) \) and a witness \( c^* : r(A)c \), where we chose \( m = \text{TE} \) to be our type erasure morphism and \( r = \text{TP} \) our logical relation capturing type preservation. This translation almost produced the desired formalization \text{SProd} except that some translated constants featured undesired type parameters. Pre- of post-composing our translation with one that removes selected type parameters is straightforward and presented in the following. The major problem is identifying these parameters in the first place. For example, in the library in Fig. 5 above we can distinguish the following cases:

- removal required, e.g., \( \text{pair} : \Pi_{i,b} \text{tm}(\text{tx}) \rightarrow \text{tm}(\text{ty}) \rightarrow \text{tm}(\text{prod}(\text{tx})(\text{ty})) \) should go to \( \text{pair} : \text{term} \rightarrow \text{term} \rightarrow \text{term} \)
- removal optional depending on the intended result, e.g., \( \text{eq} : \Pi_{i} \text{tm}(\text{tx}) \rightarrow \text{tm}(\text{tx}) \rightarrow \text{prop} \) can go to \( \text{eq} : \text{term} \rightarrow \text{term} \rightarrow \text{prop} \) or to \( \text{eq} : \Pi_{i} \text{term} \rightarrow \text{term} \rightarrow \text{prop} \); analogously \( \text{lam} : \Pi_{i,b} (\text{tm}(\text{tx}) \rightarrow \text{tm}(\text{ty})) \) can go to \( \text{lam} : (\text{term} \rightarrow \text{term}) \rightarrow \text{term} \) or to \( \text{lam} : \Pi_{i} (\text{term} \rightarrow \text{term}) \rightarrow \text{term} \)
- removal forbidden, e.g., \( \text{dfun} : \Pi_{i,\text{tp}} (\text{tm}(\text{tx}) \rightarrow \text{tp}) \rightarrow \text{tp} \) should go to \( \text{dfun} : \Pi_{i} (\text{term} \rightarrow \text{tp}) \rightarrow \text{tp} \)

Definition 4 (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 \( n \) \( \Pi \)-bindings, and any definition of \( c \) (direct or morphism) must start with the same variable sequence \( \lambda \)-bound. We write \( c^1, \ldots, c^n \) for these variable bindings. Each occurrence of \( c \) in an expression in \( D \) is (after suitably \( \eta \)-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 \text{unused} if for every \( c^i \in P \), the \( i \)-th bound variable of the type or any definition of \( c \) occurs at most as a subexpression of 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 \( i = 2 \),
theory HEqual =  
  include HTyped  
  @keep(eq^1)  
  eq : \Pi a tma \rightarrow tma \rightarrow prop  
  refl : \Pi a,ax dedeqaxx  
  eqsub : \Pi a,x,y dedeqaxy \rightarrow  
    \Pi F:tm a \rightarrow prop dedFx \rightarrow dedFy  

theory HFun =  
  include HEqual  
  fun : tp \rightarrow tp \rightarrow tp  
  @keep(lam^1)  
  lam : \Pi a,b (tm a \rightarrow tm b) \rightarrow tmfunab  
  app : \Pi a,b tmfunab \rightarrow tm a \rightarrow tm b  

theory HBeta =  
  include HFun  
  beta : \Pi a,b \Pi F:tm a \rightarrow tm b \Pi x  
    dedeqb (appab (lambdabF)x) (Fx)  

theory HEta =  
  include HFun  
  eta : \Pi a,b \Pi F:tmfunab  
    dedeq (funab) f (lambdab \lambda x, app fx)  

theory HDepFun =  
  include HEqual  
  @keep(dfun^1)  
  dfun : \Pi a (tm a \rightarrow tp) \rightarrow tp  
  @keep(dlam^1)  
  dlam : \Pi a,\Pi Fx:tm a \rightarrow tp (\Pi xtma tm bx)  
    \rightarrow tmdfunab  
  dapp : \Pi a,b tmdfunab \rightarrow \Pi xtma tm bx  

theory HDepBeta =  
  include HDepFun  
  dbeta : \Pi a,b \Pi F:Pi xtma tm bx \Pi x  
    dedeq (bx) (dappab (dlambdabF)x) (Fx)  

Figure 5: Theories for Function Types with Annotations for Needed Arguments

- the i-argument of any application of c, e.g., ct_1t_2 becomes ct_1 if i = 2.

**Lemma 3** (Removing Unused Positions). Consider a well-typed diagram D and a set P of argument positions unused in D. Then D \ P is also well-typed.

*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 \ P so that all types and definitions stay well-typed, and (ii) the type, definitions, and uses of all constants are changed consistently so that they stay well-typed. The only subtlety is that we need to apply LF’s \eta-equality to expand not fully applied uses of a constant. \qed

Note that, in the presence of include declarations or morphisms, the decision whether an argument position may be removed is not local: we must consider the entire diagram to check for all occurrences. If a theory T includes the theory S and uses a constant c declared in S, then an argument position c^i may be unused in S but used in T. Thus, the functor that removes argument positions may have to be undefined on T.

Implementing the operation D \ P is straightforward. However, much to our surprise and frustration, automatically choosing an appropriate set P turned out to be difficult:
Example 3. The undesired argument positions in $\text{TE}^{\text{HProd}}$ are exactly the named variables in $\text{HProd}$ that do not occur in their scopes in $\text{TE}^{\text{HProd}}$ anymore. This includes the positions $\text{pair}^1$ and $\text{pair}^2$, and removing them yields the desired declaration of $\text{pair}$. However, that does not hold for $\text{HDepFun}$. Here the argument $\text{dfun}^1$ is named in $\text{HDepFun}$ and unused in the declaration $\text{dfun} : \Pi_{a: \text{tp}} (\text{term} \to \text{tp}) \to \text{tp}$ that occurs in $\text{TE}^{\text{HDepFun}}$. However, that is in fact the desired formalization of the soft-typed dependent function type. Removing $\text{dfun}^1$ would yield the undesired $\text{dfun} : (\text{term} \to \text{tp}) \to \text{tp}$. While we do not mention MMT’s implicit arguments in this paper, note also that $\text{dfun}^1$ is an implicit argument in $\text{HDepFun}$ that must become explicit in $\text{SDepFun}$.

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 largest possible $P$ 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., `prod : tp` to `tp` to `tp`). A smarter choice is to remove all named argument positions that become redundant through pushout, e.g., that are named and used in $\text{HProd}$ but unused in $\text{TE}(\text{HProd})$. (Note that the pushout $\mathcal{P}(m)(D)$ has at least the argument positions that $D$ has. It may have more if $m$ 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 $P$. For now, we remove all named variables that never occur in their scope anymore, and we allow users to annotate constants like `@keep(dfun^1)` where the system should deviate from that heuristic (see Fig. 5). 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. $\text{HProd}^-(\text{TE})$ yields the theory $\text{SProd}$ except that it still lacks the `^`-ed constants. The following lemma shows that we can now obtain the morphism $e : \text{HProd} \rightarrow \text{SProd}$ from above as $\mathcal{P}^-(\text{TE})_{\text{HProd}}$:

**Lemma 4** (Removing Arguments Preserves Naturality). Consider a natural functor $O$ and a functor $O'\langle D \rangle := O(D) \setminus P_D$ for some heuristic $P$. Then $O'$ is natural as well.

**Proof.** $O$ being natural yields morphisms $O_E : E \rightarrow E^O$ 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 $\eta$-expand the right-hand sides of all assignments in the morphisms $O_E$ and remove the same argument positions in $P_D$ as well.

It is straightforward to extend Def. 2 to all theories extending $S$ in the same way as pushout extends a morphism. That would yield an include- and definition-preserving natural functor. 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 5.** Consider a morphism $m : S \rightarrow T$ and a term-total logical relation $r$ on $m$. Then the functor $\mathcal{L}R(m,r)$ maps a theory $E$ as follows:

1. We compute $E^m = \mathcal{P}^-(m)(E)$.
2. Due to Lem. 4, $E^m$ has the same shape as $E$, and there is a morphism $m_E : E \rightarrow E^m$. We create an initially empty logical relation $r_E$ on $m_E$.
3. For each $E$-declaration $c : A[\vdash t]$ for which $r_E(A)$ is defined, we add
   a) the constant declaration $c^* : r_E(A)c[\vdash r_E(t)]$ to $E^m$
   b) the case $r(c) = c^*$ to $r_E$.

**Theorem 5.** In the situation of Def. 5, the operator $\mathcal{L}R(m,r)$ is a natural functor. And every $r_E$ is a term-total logical relation on $m_E$.

**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 prove 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 $D$: We appeal to Thm. 2 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.
Now the functor \( LR(TE, TP) \) generates for every hard-typed feature \( F \)
- the corresponding soft-typed feature \( F' \)
- the type-erasure translation \( TE_F : F \to F' \) as a compositional/homomorphic mapping,
- the type preservation proof \( TP_F \) for the type erasure as a logical relation on \( TE_F \).

In particular, we have \( SProd = LR(TE, TP)(HProd) \).

### 3.4 Translating Proof Rules Correctly

We omitted the reduction rules in our introductory example \( HProd \). This was because \( LR(TE, TP) \) is still not the right operator. To see what goes wrong, assume we leave \( TP(ded) \) undefined, and consider the type of the \( \beta \) rule from \( HBeta \):

\[
\begin{align*}
HBeta & : \Pi_{a,b} \Pi_{F;tm \to tm} \Pi_x ded eq b(app a(b(lam a F)x))(Fx) \\
HBeta^{LR(TE, TP)} (generated) & : \Pi_{a,b} \Pi_{F;term \to term} \Pi_x ded eq b(app (lam a F)x)(Fx) \\
SBeta (needed) & : \Pi_{a,b} \Pi_{F;term \to term} \Pi_{x:F;tm \to tm} \Pi_{x':ded of xa} ded eq b(app (lam a F)x)(Fx)
\end{align*}
\]

The rule generated by \( LR(TE, TP)(HProd) \) is well-typed but not sound. In general, the softening operator must insert \(-\)ed assumptions for all variables akin to how Def. 5 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 \( P \cdot LR(m, r) \) for partial \( m \) and \( r \) in the same way as \( LR(m, r) \), again dropping all variable and constant declarations for whose type the translation is partial.

First we refine \( TE \) and \( TP \) (from Fig. 2 and Page 93, respectively) as follows:
- We leave \( TE(ded) \) 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 [12].
- We put \( TP(ded) = \lambda w; prop ded p \) and thus \( TP(ded P) = ded TE(P) \) for all \( P \). This trick has the effect that \( beta^* \) is generated as well and has the needed type (whereas the generation of \( beta \) is suppressed).

Then we finally define \( Soften = P \cdot LR(TE, TP) \). For every proof rule \( c \) 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 lax morphisms in the sense of [12].

### 4 Translating Libraries

In the examples so far we have applied \( Soften \) on theories extending \( HTyped \) one at a time. We now extend it to a translation on whole structured diagrams of theories and morphisms, mapping whole libraries of hard-typed features at once.

Before spelling out the definition, we show an exemplary library of hard-typed features in Fig. 7. Here, we extend the collection of theories shown so far, the most notable extensions being several morphisms and the theory \( HQuot \) formalizing hard-typed quotient types. Here, the morphism \( HProd \to HDepProd \) realizes simple product types as a special case of dependent product types. Analogously, all of the morphisms \( H\{Fun, Beta, Eta, Exten\} \to HDep\{Fun, Beta, Eta, Exten\} \) realize the simply-typed
feature as a special case of the corresponding dependently-typed feature. And the anti-parallel morphism pairs \( \text{HEta} \iff \text{HExt} \) and \( \text{HDepEta} \iff \text{HDepExt} \) capture that \( \eta \) and extensionality are equivalent in the presence of \( \beta \) reduction. Our goal is to soften this library in a way that preserves the modular structure.

**Definition 6 (Soften on Diagrams).** On the category of theories and partial morphisms, we define \( \text{Soften} \) as the partial functor translating diagrams \( D \) over \( \text{HTyped} \) to diagrams \( D' \) over \( \text{STyped} \) as follows:

- every theory/morphism with name \( X \) in \( D \) yields a theory/morphism \( X^{\text{Soften}} \) in \( D' \)
- every include \( \text{HTyped} \) is replaced by include \( \text{STyped} \); and every include \( X \) by include \( X^{\text{Soften}} \)
- every declaration \( c : A \,[=\, t] \) in a theory \( S \) yields those on the left below, and every assignment \( c := t \) in a morphism yields those on the right below (whenever the involved translations are defined)

\[
\begin{align*}
\text{c} & \quad : m_S(A) \,[=\, m_S(t)] & \quad \text{c} & \quad := m_S(t) \\
\text{c}^* & \quad : \text{TP}^S(A) \, c \,[=\, \text{TP}^S(t)] & \quad \text{c}^* & \quad := \text{TP}^S(t)
\end{align*}
\]

where \( m_S : S \to \mathcal{P}^{-}(\text{TE}^S)(S) \) is the morphism from Lem. 4 and \( \text{TE}^S \) and \( \text{TP}^S \) are given below.

We define \( \text{TE} \) as the partial morphism and \( \text{TP} \) as the partial logical relation on \( \text{TE} \) by

\[
\begin{align*}
\text{TE}(\text{prop}) & = \text{prop} \\
\text{TE}(\text{tp}) & = \text{tp} \\
\text{TE}(\text{tm}) & = \lambda_{x : \text{tp}} \text{term} \\
\text{TP}(\text{ded}) & = \lambda_{p : \text{prop}} \text{ded} \ p \\
\text{TP}(\text{tm}) & = \lambda_{x : \text{tp}} \lambda_{x : \text{term}} \text{of} \ x \ a
\end{align*}
\]

Then we additionally build the following components of \( D' \):

- every theory \( X \) yields a partial morphism \( \text{TE}^T \, : \, T \, \to \, T^{\text{Soften}} \) and a partial logical relation \( \text{TP}^T \) over \( \text{TE}^T \)
• every include $\texttt{HTyped}$ in a theory $S$ is replaced by include $\texttt{TE}$ in $TE^S$, and $TP^S$ is made to extend $TP$; and every other include $T$ in theories $S$ yields include $TE^S$, and the definition of $TP^S$ extends $TP^T$.

• every declaration $c : A [= t]$ in a theory $S$ yields $c := c$ in $TE^S$ and $TP^S(c) := c^*$ (whichever are defined).

**Theorem 6** (Structure Preservation). Consider the category of LF theories and partial type- and substitution-preserving expression translations as morphisms. Then $\texttt{Soften}$ is functorial and preserves the structure of includes and definitions. It is natural via the morphisms $TE^S$ and the relation $TP^S$.

**Proof.** This holds by construction. □

This finally yields the intended soft-typed formulation of $\texttt{SFun} := \texttt{HFun}^{\texttt{Soften}}$ and $\texttt{SDepFun} := \texttt{HDepFun}^{\texttt{Soften}}$. As an example, we give the morphism $\texttt{HSFtoDF} : \texttt{HFun} \rightarrow \texttt{HDepFun}$ and its translation below.

\[
\text{morph } \texttt{HSFtoDF} : \texttt{HFun} \rightarrow \texttt{HDepFun} = \\
\text{include } \texttt{HEqual} \\
\text{fun} = \lambda_{a,b} \text{dfun} \lambda_{x} b \\
\text{lam} = \lambda_{a,b,F} \text{dlam} \lambda_{x} b \ F \\
\text{app} = \lambda_{a,b,f,x} \text{dapp} \lambda_{x} b \ f \ x
\]

\[
\text{morph } \texttt{HSFtoDF}^{\texttt{Soften}} : \texttt{HFun}^{\texttt{Soften}} \rightarrow \texttt{HDepFun}^{\texttt{Soften}} = \\
\text{include } \texttt{HEqual}^{\texttt{Soften}} \\
\text{fun} = \lambda_{a,b} \text{dfun} \lambda_{x} b \\
\text{lam} = \lambda_{a,F} \text{dlam} \ F \\
\text{lam}^\ast = \lambda_{a,b,F,F^\ast} \text{dlam}^\ast \lambda_{x} b \ F \ F^\ast \\
\text{app} = \lambda_{f,x} \text{dapp} \ f \ x \\
\text{app}^\ast = \lambda_{a,b,f,f^\ast,x,x^\ast} \text{dapp} \lambda_{x} b \ f \ f^\ast \ x \ x^\ast
\]

If we generalize the meta-theory of [12] to partial morphisms/relations and work in a variant of LF that adds product types, we could pair up $TE^S$ and $TP^S$ into a single expression translation that maps every term to the pair of its type erasure and its type preservation proof.

## 5 Implementation

The formalizations developed in this paper including the act of softening are available online as part of the LATIN2 library.\footnote{https://gl.mathhub.info/MMT/LATIN2/-/tree/devel/source/casestudies/2021-softening} Our implementation adds a component to MMT that applies logical relation-based translations to entire diagrams of theories and morphisms. Then softening arises as one special case of that construction. While all translations are implemented in the underlying programming language of MMT and thus part of the trusted code base, our general and systematic approach minimizes the amount of new code needed for any given instance such as softening. That makes it much easier to review their correctness. In any case, the generated diagrams can easily be double-checked by the original logical framework.

We are still experimenting with how to trigger these translations. It is non-obvious if softening should be triggered by a new kind of declaration in the logical framework, a library-level script that lives outside the logical framework, or a feature of the implementation that transparently builds the softened theory whenever the user refers to it. As a prototype, we have chosen the first of these approaches.

In our implementation it proved advantageous to not have separate syntax for morphisms and logical relations. Indeed, both are maps of names to expressions that extend to compositional translations of expressions to expressions, differing only in the inductive extension. Instead, we found a way to represent
every logical relation as a morphism, thus obviating the need to introduce additional syntax for relations. The trick is to implement a special include-preserving functor that generates a theory $I$ that specifies exactly the typing requirements for the cases in a logical relation $r$, and then to represent $r$ as a morphism out of $I$. We can even use this trick to represent multiple translations at once in a single morphism. For example, in our implementation we jointly represent $TE$ and $TP$ from Def. 6 as a single morphism $TypePres$, which in implementation-near syntax reads as follows.

```
view TypePres : HTyped_comptrans -> STyped =
  prop/TE = prop
  tp/TE = tp
  tm/TE = [x] term
  tm/TP = [A,t] ⊢ t : A
  ded/TP = ded
```

Here, $HTyped_comptrans$ refers to a suitable kind interface theory for combined translations on $HTyped$.

Softening now emerges as the composition of multiple operators in our implementation, which for the sake of conciseness were combined in one big operator in this paper. Assume we wanted to soften a library of hard-typed features given as a diagram $HLibrary$ (e.g., the one from Fig. 7). First, we compute the diagram $HLibrary_comptrans$ of corresponding interface theories. This diagram has the same shape as $HLibrary$ and is a tree rooted in $HTyped_comptrans$. Then, we compute the pushout of the resulting diagram along $TypePres$. The steps so far are effectively equivalent to applying the operator from Def. 3 accounted for with correct translation of proof rules. Finally, it remains to apply the operator that drops unnecessary parameters (according to the heuristic outlined in Sec. 3.3). Below, we show how the last two steps look like in our.

```
diagram SLibrary :=
  DROP_PARAMS STyped (PUSHOUT TypePres HLibrary_comptrans)
```

6 Conclusion

We have given a translation of hard-typed (intrinsic) to soft-typed (extrinsic) formalizations of type theory. Even though the existence of such translation is known, it had previously proved difficult to derive it from meta-theoretic principles in such a way that it can be studied and implemented easily. Our key insight was that the associated type preservation proof can be cast as a logical relation, which allowed us to derive the translation from the requirement that the logical relation proof succeeds.

Our translation preserves modularity, which makes it suitable for translating modular libraries of formalizations of various type theories. That enhances the quality and coverage of the library while reducing the maintenance effort. Our implementation will serve as a key component in scaling up our modular formalizations of type theories in our LATIN2 library.

We expect our methodology of functors on diagrams of LF theories to extend to other important translations such as adding polymorphism or universes.
References

[1] J. Bernardy and M. Lasson. Realizability and parametricity in pure type systems. In M. Hofmann, editor, Foundations of Software Science and Computational Structures, pages 108–122, 2011. doi:10.1007/978-3-642-19805-2_8.

[2] A. Church. A Formulation of the Simple Theory of Types. Journal of Symbolic Logic, 5(1):56–68, 1940. doi:10.2307/2266170.

[3] 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. doi:10.1007/978-3-642-22673-1_24.

[4] H. Curry and R. Feys. Combinatory Logic. North-Holland, Amsterdam, 1958. doi:10.1016/S0049-237X(08)72041-X.

[5] W. Farmer, J. Guttman, and F. Thayer. Little Theories. In D. Kapur, editor, Conference on Automated Deduction, pages 467–581, 1992. doi:10.1007/3-540-55602-8_192.

[6] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. Journal of the Association for Computing Machinery, 40(1):143–184, 1993. doi:10.1145/138027.138060.

[7] R. Harper, D. Sannella, and A. Tarlecki. Structured theory presentations and logic representations. Annals of Pure and Applied Logic, 67:113–160, 1994. doi:10.1016/0168-0072(94)90009-4.

[8] T. Mossakowski, S. Autexier, and D. Hutter. Development graphs - Proof management for structured specifications. J. Log. Algebr. Program, 67(1–2):114–145, 2006. doi:10.1016/j.jlap.2005.09.005.

[9] 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. doi:10.1007/3-540-44755-5_23.

[10] F. Pfenning. Structural cut elimination: I. intuitionistic and classical logic. Information and Computation, 157(1-2):84–141, 2000. doi:10.1006/inco.1999.2832.

[11] F. Pfenning, C. Schürmann, M. Kohlhase, N. Shankar, and S. Owre. The Logosphere Project, 2003. https://kwarc.info/projects/logosphere/.

[12] F. Rabe. Lax Theory Morphisms. ACM Transactions on Computational Logic, 17(1), 2015. doi:10.1145/2818644.

[13] F. Rabe. How to Identify, Translate, and Combine Logics? Journal of Logic and Computation, 27(6):1753–1798, 2017. doi:10.1093/logcom/exu079.

[14] F. Rabe and N. Roux. Structure-Preserving Diagram Operators. In M. Roggenbach, editor, Recent Trends in Algebraic Development Techniques, pages 142–163. Springer, 2020. doi:10.1007/978-3-030-73785-6_8.

[15] 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. doi:10.1145/1577824.1577831.

[16] F. Rabe and K. Sojakova. Logical Relations for a Logical Framework. ACM Transactions on Computational Logic, 14(4):1–34, 2013. doi:10.1145/2536740.2536741.