Lax Theory Morphisms

Florian Rabe, Jacobs University, Bremen, Germany

When relating formal languages, e.g., in logic or type theory, it is often important to establish representation theorems. These interpret one language in terms of another in a way that preserves semantic properties such as provability or typing. Meta-languages for stating representation theorems can be divided into two groups: Firstly, computational languages are very expressive (usually Turing-complete), but verifying the representation theorems is very difficult (often prohibitively so); secondly, declarative languages are restricted to certain classes of representation theorems (often based on theory morphisms), for which correctness is decidable.

Neither is satisfactory, and this paper contributes to the investigation of the trade-off between these two methods. Concretely, we introduce *lax* theory morphisms, which combine some of the advantages of each: they are substantially more expressive than conventional theory morphisms; but they share many of the invariants that make theory morphisms easy to work with.

Specifically, we introduce lax morphisms between theories of a dependently-typed logical framework, but our approach and results carry over to most declarative meta-languages.

We demonstrate the usefulness of lax theory morphisms by stating and verifying a type erasure translation from typed to untyped first-order logic. The translation is stated as a single lax theory morphism, and the invariants of the framework guarantee its correctness. This is the first time such a complex translation can be verified in a declarative framework.

CCS Concepts: •**Theory of computation** → **Logic and verification;** *Proof theory; Type theory;*

Additional Key Words and Phrases: theory morphism, logical relation, translation, logical framework, dependent type theory, representation theorem

ACM Reference Format:
Florian Rabe, 2015. Lax Theory Morphisms. *ACM Trans. Comput. Logic* V, N, Article 0 (2015), 35 pages.
DOI: http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION AND RELATED WORK

Theory Morphisms. In declarative languages, especially logics and type theories, we often encapsulate a group of related declarations and axioms into a **theory**. For example, in first-order logic the theories are lists of declarations of function and predicate symbols and axioms. For example, the first-order theory `Mon` of monoids consists of a binary function symbol `comp` for composition, a nullary function symbol `e` for the unit, and axioms for associativity and neutrality.

For virtually every declarative language, we can define the concept of *theory morphisms*. Even though the exact definition depends on the specific declarative language, the underlying intuition is almost always the same. A **theory morphism** σ from a theory Σ to a theory Σ' is a function that maps all Σ-expressions (contexts, substitutions, terms, types, formulas, proofs, etc.) to corresponding Σ'-expressions and satisfies the following two properties:

— **Homomorphic extension**: σ is determined by its action on the Σ-symbols only and is extended homomorphically to all Σ-expressions.

The author was supported by DFG grant RA-18723-1 OAF.

DOI: http://dx.doi.org/10.1145/0000000.0000000

For example, a morphism from `Mon` to some Σ' is given by mapping the binary `Mon`-function symbol `comp` to a binary Σ'-function $t(x,y)$ and accordingly the nullary `Mon`-function symbol `e` to a Σ'-term u. Then the homomorphic extension property forces, e.g., $\sigma(\texttt{comp}(x,\texttt{e})) = t(\sigma(x),\sigma(\texttt{e})) = t(x,u)$ where σ maps the `Mon`-variable $x : a$ to the Σ'-variable $x : \sigma(a)$.

An important consequence is the **substitution lemma**, which states that theory morphisms commute with substitution, i.e., $\sigma(f[x/t]) = \sigma(f)[x/\sigma(t)]$.

— **Judgment preservation**: σ preserves all judgments about expressions, in particular typing and provability. For example in type theory, if $\Gamma \vdash_\Sigma t : A$, then $\sigma(\Gamma) \vdash_{\Sigma'} \sigma(t) : \sigma(A)$.

The above intuitions also apply to **axioms**: For each Σ-*axiom* F, we require that $\sigma(F)$ is a Σ'-*theorem*. If we subscribe to a Curry-Howard representation of proofs as terms [Curry and Feys 1958; Howard 1980], this becomes a special case of the above. Then we think of proofs as certain expressions, of axioms as proof symbols, and of theorems as proof expressions. Thus, a theory morphism maps an axiom a asserting F to a proof $\sigma(a)$ proving $\sigma(F)$. Now judgment preservation subsumes truth preservation: If p is a Σ-proof of any formula F, then $\sigma(p)$ is a Σ'-proof of $\sigma(F)$.

The above two properties have made theory morphisms a crucial tool for relating and structuring theories in mathematics and computer science.

Firstly, the homomorphic extension property makes it easy to define individual theory morphisms by giving the value of $\sigma(c)$ for all Σ-symbols c. In particular, σ is finite in the typical case where Σ contains only finitely many declarations. Moreover, we can usually (a very general case is proved in [Rabe 2014]) prove inductively that judgment preservation already holds for all expressions if it holds for all symbols c. Therefore, the correctness of theory morphisms is decidable if the judgments of the underlying declarative language are.

Secondly, the judgment preservation property expresses a representation theorem: All theorems proved in Σ give rise to theorems in Σ' by translation along σ. This allows stating every result (e.g., a definition, theorem, or algorithm) in the smallest possible theory Σ and then reusing it in many larger theories Σ'.

Historically, the systematic use of theory morphisms to build mathematical theories goes back to the works by Bourbaki [Bourbaki 1964] although they did not make the method explicit. To the author's knowledge, the first formal definition of theory morphisms was given in [Enderton 1972] (under the name *interpretation* and for first-order logic). The *little theories* methodology [Farmer et al. 1992] first advocated the systematic use of theory morphisms to build mathematical theories. Multiple deduction tools allow using theory morphisms including IMPS [Farmer et al. 1993], OBJ [Goguen et al. 1993], Isabelle [Paulson 1994], and Twelf [Pfenning and Schürmann 1999].

Theory morphisms have proved especially powerful in computer-supported **module systems**, where theories are built modularly from reusable components. This is because the concepts of extension and refinement, which are essential for module systems, can be formalized as theory morphisms. Among programming languages, this includes the SML module system [Milner et al. 1997] (where the theories and morphisms are called *signatures* and *functors*). In algebraic specification, the framework of institutions [Goguen and Burstall 1992] is based on a category of theories and morphisms, and the ASL language [Sannella and Wirsing 1983] provides a generic module system for any institution. In formal deduction, the IMPS proof system [Farmer et al. 1993] was built to exploit theory morphisms in a computer system. An extensive survey can be found in [Rabe and Kohlhase 2013].

Language Morphisms. Many representation theorems can be expressed as theory morphisms. To do so, we use a **logical framework** like LF [Harper et al. 1993] or Isabelle [Paulson 1994] that permits defining declarative languages as theories of the framework. Then representation theorems between languages can be stated as theory morphisms of the framework. This permits relating and structuring declarative languages modularly in the

same way as the theories of a fixed declarative language. This was first suggested in [Harper et al. 1994]. It was applied systematically to build the LATIN library [Codescu et al. 2011] of modular logics and type theories defined in the Twelf module system [Rabe and Schürmann 2009].

If a language translation is represented as a framework-level theory morphism, the properties of theory morphisms guarantee that the translation is homomorphic and judgment-preserving. This, of course, has the drawback that only homomorphic and judgment-preserving translations can be represented. While this restriction may appear very natural, it excludes many interesting representation theorems. For example, it excludes already the type-erasure translation from typed to untyped first-order logic, one of the simplest non-trivial logic translations. We will get back to this in our running example.

Alternatively, we can use **unrestricted meta-languages** to represent language translations as arbitrary (i.e., not necessarily homomorphic or judgment-preserving) mappings of Σ-expressions to Σ'-expressions. Mathematically, this is best developed in abstract logical frameworks like institutions [Goguen and Burstall 1992], which use functors between categories of theories. Computationally, we can use systems such as Hets [Mossakowski et al. 2007], which represent languages and translations in a general purpose programming language. The drawback of this generality is that each translation must be described ad hoc and correctness must be proved separately for each translation.

But judgment preservation is often the central theorem that establishes the correctness of a language translation. Therefore, **mechanically verified language translations** have been limited to

— simple translations that can be represented declaratively in a framework that guarantees correctness, e.g., the translation from modal logic to first-order logic in [Rabe and Sojakova 2013] or from HOL to Nuprl in [Schürmann and Stehr 2004],
— ad-hoc implementations that are verified at great effort, e.g., the translation from HOL to FOL verified in [Blanchette and Popescu 2013][1].

Logical Relations. A second class of representation theorems can be formalized in logical frameworks by using logical relations. While theory morphisms are well-suited for stating language translations, logical relations can be used to state additional invariants enjoyed by a language translation.

Logical relations can be defined for most declarative languages. Given a theory Σ, a logical relation ρ is always a family of relations $\rho(A)$ indexed by the Σ-types A. Intuitively, the key property is that ρ must be closed under all Σ-expressions.

Logical relations have originally been defined semantically going back to [Reynolds 1974]. A **semantic** logical relation ρ operates on a Σ-model M: It maps every Σ-type A to a relation $\rho(A) \subseteq [\![A]\!]^M$ and every Σ-term $t : A$ to a proof that $[\![t]\!]^M \in \rho(A)$.

We will look at logical relations syntactically following [Bernardy et al. 2012; Rabe and Sojakova 2013]. A **syntactic** logical relation operates on a theory morphism $\sigma : \Sigma \to \Sigma'$: It maps every Σ-type A to a Σ'-predicate $\rho(A) : \sigma(A) \to \texttt{type}$ (in some appropriate variant of dependent type theory) and every Σ-term $t : A$ to a Σ'-term $\rho(t) : \rho(A)\,\sigma(t)$.

Syntactic logical relations are structurally similar to theory morphisms: ρ maps all Σ-expressions to Σ'-expressions; $\rho(-)$ is determined by its values on Σ-symbols only and extended inductively to all Σ-expressions; and ρ satisfies a certain judgment preservation property. The main difference is that theory morphisms and logical relations use different inductive extensions and different preservation properties.

[1] In this example, both logics happen to be represented declaratively in the logical framework Isabelle, but the translation is given computationally.

More precisely, we speak of a **strict** logical relation if $\rho(-)$ is extended inductively from its action on symbols. If the inductive extension property is waived, we speak of a **lax** logical relation. The latter were introduced in [Plotkin et al. 2000] after observing that the lax logical relations enjoy much stronger closure properties than the strict ones, e.g., compositions and intersections of strict logical relations are usually lax but not strict.

Historically, strict semantic relations were considered first [Reynolds 1974] and then generalized independently to strict syntactic [Bernardy et al. 2012; Rabe and Sojakova 2013] and lax semantic relations [Plotkin et al. 2000]. Individual lax syntactic relations have been given ad hoc (e.g., in [Schürmann and Sarnat 2008]), but a general treatment has so far been lacking.

We give such a general treatment of lax syntactic relations, systematically combining the ideas of [Plotkin et al. 2000] and [Rabe and Sojakova 2013]. Our results show that the properties strict/lax and semantic/syntactic are orthogonal, resulting in 2×2 styles of logical relations.

Contribution. We strive to combine the generality of unrestricted meta-languages with the effective and decidable representations enjoyed by theory-morphism-based approaches. Our long-term motivation is to build tools for stating and verifying complex representation theorems. Our concrete contribution in this paper is threefold:

Firstly, we introduce **lax theory morphisms**. These are maps of Σ-expressions to Σ'-expressions that satisfy the judgment preservation property of theory morphisms but waive the homomorphic extension property required by conventional theory morphisms. To avoid ambiguity, we will call conventional theory morphisms **strict**.

We study the basic properties of lax morphism and show that they enjoy nice closure properties, in particular the closure under finite products.

Secondly, we give a systematic treatment of **lax syntactic logical relations**. In [Rabe and Sojakova 2013], we already studied *strict* syntactic logical relations on *strict* theory morphisms of the *LF type theory* [Harper et al. 1993]. Our treatment here is more general in three ways: We study *lax* logical relations on *lax* theory morphisms for a *larger type theory*.

The importance of logical relations for our purposes is that they play a crucial role in constructing lax theory morphisms: We use them to obtain lax theory morphisms that pair the translation $\sigma(t)$ of a term t with an invariant $\rho(t)$ about $\sigma(t)$. This allows the definition of a translation and the proof of an invariant about it to be mutually recursive.

Thirdly, we apply our results to represent and **verify a type-erasure translation** from typed to untyped first-order logic. We will see that our results yield a framework in which such translations can be stated declaratively and verified mechanically. This is the first time that such a complex logic translation is represented and verified purely declaratively.

Overview. In Sect. 4, we introduce lax theory morphisms, and in Sect. 5, we introduce lax logical relations between them. The type-erasure translation serves as our running example throughout the paper and is finished in Sect. 6.

We conclude in Sect. 7 with an extensive preview of future work that is enabled by our results. Most interestingly this preview shows that lax morphisms are a promising concept for representing model theory in logical frameworks, including Henkin models (i.e., models that use a non-standard interpretation of function types) and initial models. It also indicates that lax morphisms naturally arise in connection with record and inductive types.

Our results are stated generically for an arbitrary declarative language in order to maximize reusability and to emphasize that the general ideas are language-independent. Therefore, we work with a basic type theory, from which specific type theories can be obtained by adding features. To make this precise, we begin by introducing this basic type theory

in Sect. 2. In Sect. 3, we give the example features that we will use throughout this paper: dependent products types and dependent function types.

The definitions (albeit not all notations) introduced in Sect. 2 and 3 are well-known. Experienced readers may try the following **short-cut** on a first read: read only Fig. 1, Def. 2.6, and Sect. 3.2 and 3.3, as well as the introduction of the running example in Sect. 3.5.

2. A BASIC TYPE THEORY

As the starting point of our investigation, we use a basic type theory. Our development is unusual in that this section focuses on the theories and contexts but does not fix the set of expressions. The choice of expressions such as function types and λ-abstraction is relegated to individual features, of which we introduce several in Sect. 3. Other than that, all material described in this section is well-known.

2.1. Syntax

	Grammar	Typing judgment	Equality judgment
Theories	$\Sigma ::= \cdot \mid \Sigma,\ c : E$	$\vdash \Sigma\ \texttt{Thy}$	
Contexts	$\Gamma ::= \cdot \mid \Gamma,\ x : E$	$\vdash_\Sigma \Gamma\ \texttt{Ctx}$	$\vdash_\Sigma \Gamma = \Gamma'$
Strict morphisms	$\sigma ::= \cdot \mid \sigma,\ c \mapsto E$	$\vdash \sigma : \Sigma_1 \to \Sigma_2$	
Substitutions	$\gamma ::= \cdot \mid \gamma,\ E$	$\vdash_\Sigma \gamma : \Gamma_1 \to \Gamma_2$	$\vdash_\Sigma \gamma = \gamma' : \Gamma_1 \to \Gamma_2$
Expressions	$E ::= c \mid x \mid \texttt{type} \mid \ldots$	$\Gamma \vdash_\Sigma E : E'$	$\Gamma \vdash_\Sigma E = E'$

Fig. 1. Basic Syntax and Judgments

The grammar and judgments of our basic type theory are given in Fig. 1. Theories, contexts, strict morphisms, and substitutions are comma-separated lists, and we will use comma also for concatenating lists. The names introduced in theories and contexts may occur in later elements of the list.

Theories and Contexts. **Theories** Σ declare the globally available **constants** c. Relative to such a theory, the **contexts** Γ declare the locally available **variables**. Both constants and variables are typed, and each constant/variable may occur in the types of later declarations. Variables (but not constants) are subject to binding, substitution, α-renaming, and shadowing in the usual way.

Both theories and contexts are equipped with homomorphic mappings: **strict theory morphisms** σ map between theories and **substitutions** γ between contexts. A morphism $\vdash \sigma : \Sigma_1 \to \Sigma_2$ maps every Σ_1-constant c to a Σ_2-expression E of the appropriate type. Similarly, for a fixed theory Σ, a substitution $\vdash_\Sigma \gamma : \Gamma_1 \to \Gamma_2$ maps every Γ_1-variable to a Γ_2-expression. Both mappings can be extended homomorphically to all contexts and expressions (see Def. 2.2 and 2.3). In particular, σ maps Σ_1-expressions in context Γ to Σ_2-expressions in context $\sigma(\Gamma)$.

The declarations in Γ_1 and the corresponding cases in γ must occur in the same order. Therefore, we omit the variable names in substitutions. Analogously, we could omit the constant names in strict morphisms, but it is more practical to repeat them (especially when using large theories where declarations are often reordered).

Expressions. Relative to a theory Σ and a Σ-context Γ, we use typed **expressions** E. The expressions always include the Σ-constants c and the Γ-variables x and the universe $\texttt{type}$.

Expressions are subject to the typing judgment $E : E'$. We do not fix a universe hierarchy, i.e., E and E' may be terms, types, kinds, etc. We assume that there is at least the universe

`type` of all types only because it makes some definitions more convenient. In particular, given $t : A$ and $A : \texttt{type}$, we say that t is a **term** and A a **type**.

Axioms. We will use the Curry-Howard correspondence [Curry and Feys 1958; Howard 1980] to treat axioms as special cases of constant declarations. For example, a specific type theory can add a nullary constructor `prop` such that if $F : \texttt{prop}$ the declaration $c : F$ represents an axiom asserting F. The distinction between normal constant declarations and axioms will not be relevant in this paper.

Notation 2.1 (*Antecedents*). As usual, we will omit the antecedent of a judgment if it is the empty context.

Moreover, it is often convenient to use a fixed context as the antecedent in the judgments for contexts and substitutions. This permits talking about context and substitution fragments that are well-formed relative to a fixed initial fragment. Therefore, we define the following abbreviations:

$$\begin{array}{rcl} \Gamma \vdash_\Sigma \Gamma' \ \texttt{Ctx} & \text{for} & \cdot \vdash_\Sigma \Gamma, \Gamma' \ \texttt{Ctx} \\ \Gamma \vdash_\Sigma \gamma : \Gamma_1 \to \Gamma_2 & \text{for} & \cdot \vdash_\Sigma id_\Gamma, \gamma : \Gamma, \Gamma_1 \to \Gamma, \Gamma_2 \\ \Gamma \vdash_\Sigma \gamma = \gamma' : \Gamma_1 \to \Gamma_2 & \text{for} & \cdot \vdash_\Sigma id_\Gamma, \gamma = id_\Gamma, \gamma' : \Gamma, \Gamma_1 \to \Gamma, \Gamma_2 \end{array}$$

where the identity substitution id_Γ is defined in Def. 2.2.

In particular, it is sometimes useful to think of $\Gamma \vdash_\Sigma \gamma : \Gamma_1 \to \cdot$ as a record in context Γ of type Γ_1.

We make some basic definitions for substitutions:

Definition 2.2 (*Substitutions*). Given $\Gamma = x_1 : A_1, \dots, x_n : A_n$, we define the identity substitution

$$id_\Gamma := x_1, \dots, x_n$$

Given $\vdash_\Sigma \gamma : \Gamma_1 \to \Gamma_2$, we define the application $-[\gamma]$ to well-typed expressions, contexts, and substitutions by

— for an expression in context Γ_1:

$$\begin{array}{lll} x_i[\gamma] & := t_i & \text{if } \gamma = t_1, \dots, t_n \\ c[\gamma] & := c & \\ \texttt{type}[\gamma] & := \texttt{type} & \end{array}$$

— for a context Δ such that $\Gamma_1 \vdash_\Sigma \Delta \ \texttt{Ctx}$:

$$\begin{array}{ll} \cdot[\gamma] & := \cdot \\ (\Delta, x : A)[\gamma] & := \Delta[\gamma], x : A[\gamma, id_\Delta] \end{array}$$

— for a substitution δ:

$$\begin{array}{ll} \cdot[\gamma] & := \cdot \\ (\delta, t)[\gamma] & := \delta[\gamma], t[\gamma] \end{array}$$

Given $\Gamma \vdash_\Sigma \gamma : \Gamma_1 \to \Gamma_2$, we define the application $-[\gamma]$ as an abbreviation of $-[id_\Gamma, \gamma]$ (i.e., left-most entries in a substitution can be omitted if they are identity maps).

We have the analogue to Def. 2.2 for the homomorphic extension of a strict theory morphism:

Definition 2.3 (*Morphisms*). Given $\vdash \sigma : \Sigma_1 \to \Sigma_2$, we define the **homomorphic extension** $\sigma(-)$ to well-typed expressions, contexts, and substitutions over Σ_1 in the same way as for substitutions with the exception of the following two cases:

$$\begin{array}{lll} \sigma(x) & := x & \\ \sigma(c) & := E & \text{if } c \mapsto E \text{ in } \sigma \end{array}$$

Technically, substitution and morphism application are partial mappings, but we will see below that they are well-defined total mappings if all involved entities are well-typed.

2.2. Inference Rules

$$\begin{array}{lll}
\text{Theories} & \dfrac{}{\vdash \cdot\ \texttt{Thy}} & \dfrac{\vdash \Sigma\ \texttt{Thy} \quad c \text{ not in } \Sigma \quad \cdot \vdash_\Sigma A : \texttt{type}}{\vdash \Sigma,\ c : A\ \texttt{Thy}} \\[2ex]
\text{Contexts} & \dfrac{\vdash \Sigma\ \texttt{Thy}}{\vdash_\Sigma \cdot\ \texttt{Ctx}} & \dfrac{\vdash_\Sigma \Gamma\ \texttt{Ctx} \quad \Gamma \vdash_\Sigma A : \texttt{type}}{\vdash_\Sigma \Gamma,\ x : A\ \texttt{Ctx}} \\[2ex]
\text{Morphisms} & \dfrac{\vdash \Sigma'\ \texttt{Thy}}{\vdash \cdot : \cdot \to \Sigma'} & \dfrac{\vdash \sigma : \Sigma \to \Sigma' \quad \vdash_{\Sigma'} t : \sigma(A)}{\vdash \underline{\sigma, c \mapsto t}\ :\ \underline{\Sigma, c : A}\ \to\ \Sigma'} \\[2ex]
\text{Substitutions} & \dfrac{\vdash_\Sigma \Gamma'\ \texttt{Ctx}}{\vdash_\Sigma \cdot : \cdot \to \Gamma'} & \dfrac{\vdash_\Sigma \gamma : \Gamma \to \Gamma' \quad \Gamma' \vdash_\Sigma t : \gamma(A)}{\vdash_\Sigma \underline{\gamma, t}\ :\ \underline{\Gamma, x : A} \to \Gamma'} \\[2ex]
\text{Lookup} & \dfrac{\vdash_\Sigma \Gamma\ \texttt{Ctx} \quad c : E \text{ in } \Sigma}{\Gamma \vdash_\Sigma c : E} & \dfrac{\vdash_\Sigma \Gamma\ \texttt{Ctx} \quad x : E \text{ in } \Gamma}{\Gamma \vdash_\Sigma x : E}
\end{array}$$

Fig. 2. Basic Typing Rules

$$\begin{array}{lll}
\text{Contexts} & \dfrac{\vdash \Sigma\ \texttt{Thy}}{\vdash_\Sigma \cdot = \cdot} & \dfrac{\vdash_\Sigma \Gamma = \Gamma' \quad \Gamma \vdash_\Sigma A = A'}{\vdash_\Sigma \underline{\Gamma, x : A} = \underline{\Gamma', x : A'}} \\[2ex]
\text{Substitutions} & \dfrac{\vdash_\Sigma \Gamma'\ \texttt{Ctx}}{\vdash_\Sigma \cdot = \cdot : \cdot \to \Gamma'} & \dfrac{\vdash_\Sigma \gamma_1 = \gamma_2 : \Gamma \to \Gamma' \quad \Gamma' \vdash_\Sigma t_1 = t_2}{\vdash_\Sigma \underline{\gamma_1, t_1} = \underline{\gamma_2, t_2} : \underline{\Gamma, x : A} \to \Gamma'} \\[2ex]
\text{Expressions} & \dfrac{\Gamma \vdash_\Sigma E : E'}{\Gamma \vdash_\Sigma E = E} \quad \dfrac{\Gamma \vdash_\Sigma E = E'}{\Gamma \vdash_\Sigma E' = E} & \dfrac{\Gamma \vdash_\Sigma E = E' \quad \Gamma \vdash_\Sigma E' = E''}{\Gamma \vdash_\Sigma E = E''} \\[2ex]
\text{Typing} & \dfrac{\Gamma \vdash_\Sigma t : A \quad \Gamma \vdash_\Sigma t = t' \quad \Gamma \vdash_\Sigma A = A'}{\Gamma \vdash_\Sigma t' : A'} &
\end{array}$$

Fig. 3. Basic Equality Rules

The rules of our type theory are given in Fig. 2 and 3, where we occasionally use underlining to clarify the grouping of expressions. These provide only minimal information about typing and equality of expressions. Additional rules to govern the typing of expressions must be added together with the respective productions.

Fig. 2 gives the typing rules concerning the formation of and the lookup in theories and contexts. Theories and morphisms as well as contexts and substitutions are typed component-wise in a straightforward manner. Theories and contexts are essentially the same at this point in that both are lists of typed identifiers. Similarly, theory morphisms and substitutions are the same in that both map identifiers to expressions of the appropriate type. However, we separate them here so that features can add new declarations in theories without changing the definition of contexts.

Fig. 3 gives the equality rules. Equality of contexts and substitutions is defined component-wise. Equality of expressions is a congruence relation, defined by reflexivity, symmetry, transitivity, and congruence with respect to typing.

2.3. Properties

We state some well-known properties of our basic type theory that we will need later on. Most importantly, for a fixed theory, the contexts and substitutions form a category:

THEOREM 2.4 (CATEGORY OF CONTEXTS AND SUBSTITUTIONS). *The following rules are admissible*

$$\frac{\vdash_\Sigma \Gamma\ \texttt{Ctx}}{\vdash_\Sigma id_\Gamma : \Gamma \to \Gamma} \qquad \frac{\vdash_\Sigma \gamma^* : \Gamma^* \to \Gamma \quad \vdash_\Sigma \gamma : \Gamma \to \Gamma'}{\vdash_\Sigma \gamma^*[\gamma] : \Gamma^* \to \Gamma'}$$

$$\frac{\vdash_\Sigma \gamma : \Gamma \to \Gamma' \quad \Gamma \vdash_\Sigma \Delta\ \texttt{Ctx}}{\Gamma' \vdash_\Sigma \Delta[\gamma]\ \texttt{Ctx}} \qquad \frac{\vdash_\Sigma \gamma : \Gamma \to \Gamma' \quad \Gamma \vdash_\Sigma \delta : \Delta \to \Delta'}{\Gamma' \vdash_\Sigma \delta[\gamma] : \Delta[\gamma] \to \Delta'[\gamma]}$$

$$\frac{\vdash_\Sigma \gamma : \Gamma \to \Gamma' \quad \Gamma \vdash_\Sigma t : A}{\Gamma' \vdash_\Sigma t[\gamma] : A[\gamma]}$$

and the contexts and substitutions (both quotiented by the equality judgment) form a category with composition $\gamma \circ \gamma^*$ given by substitution application $\gamma^*[\gamma]$.

Remark 2.5 (*Quotienting Out Equality*). The phrase *up to the equality judgment* in Thm. 2.4 means that we quotient the expressions, contexts, and substitutions by the respective equality judgment. However, we will routinely and implicitly use individual expressions, contexts, and substitutions as representatives for the respective equivalence class.

Note that taking the quotient is not necessary to prove Thm. 2.4. However, quotienting at least the substitutions (and therefore also the terms) is necessary to establish other desirable properties, e.g., to show that certain substitutions are isomorphisms.

Definition 2.6 (*Category of Contexts and Substitutions*). For a theory Σ, let $\texttt{Cont}(\Sigma)$ be the category of Thm. 2.4. We write $\texttt{Terms}_\Sigma$ for the functor $\texttt{Cont}(\Sigma) \to \texttt{Set}$ that maps
— every context Γ to the set of pairs (t, A) satisfying $\Gamma \vdash_\Sigma t : A$,
— every substitution γ to the function $(t, A) \mapsto (t[\gamma], A[\gamma])$.

Analogously to Thm. 2.2, we obtain the category of theories and strict theory morphisms. We omit it here because it is a special case of our results in Sect. 4.

Remark 2.7. These results depend on which features are added to our basic type theory because the proof of Thm. 2.4 must use induction on derivations. However, they can be proved generically for a large class of features, including the ones we introduce in Sect. 3. Even the symbol `type` is built into our basic type theory only for convenience and does not affect the proofs. Details can be found in [Rabe 2014].

3. SPECIFIC TYPE THEORIES

We can now extend our basic type theory with various specific type features. We will look at 2 such features in particular: dependent product and dependent function types.[2]

3.1. Complex Expressions

In order to treat a large variety of complex expressions generically, we first introduce a very general form of expression constructor taken from [Rabe 2014]: We extend our grammar with

$$E ::= \ldots \mid C(\Gamma; E^*)$$

The expression $C(\Gamma; \vec{E})$ applies the constructor C and binds a list of variables Γ, which are bound in a list of arguments $\vec{E}$. Both Γ and $\vec{E}$ may be empty.

Such expressions subsume many relevant cases:

— Non-binding n-ary operators C take an empty Γ and a list $\vec{E}$ of n arguments.
For example, the application operator $C = @$ forms the expressions $@(\cdot; f, x)$, where $\vec{E}$ has length 2.

— Unary binders C take a Γ and $\vec{E}$ of length 1.
For example, the abstraction binder $C = \lambda$ forms the expressions $\lambda(x : A; t)$.

The advantage of these complex expressions is that we can define substitution generically: We extend Def. 2.2 to complex expressions by

$$C(\Delta; \ldots, E_i, \ldots)[\gamma] = C(\Delta[\gamma]; \ldots, E_i[\gamma, id_\Delta], \ldots)$$

and accordingly for the morphism application of Def. 2.3.

Similarly, we can give the inference rules for α-equality and congruence generically:

$$\frac{\Delta = x_1 : A_1, \ldots, x_n : A_n \quad \Delta' = y_1 : A_1, \ldots, y_n : A_n[y_1, \ldots, y_{n-1}]}{\Gamma \vdash_\Sigma C(\Delta; \ldots, E_i, \ldots) = C(\Delta'; \ldots, E_i[y_1, \ldots, y_n], \ldots)}$$

$$\frac{\Gamma \vdash_\Sigma \Delta = \Delta' \quad \Gamma, \Delta \vdash_\Sigma E_i = E'_i \text{ for all } i}{\Gamma \vdash_\Sigma C(\Delta; \ldots, E_i, \ldots) = C(\Delta'; \ldots, E'_i, \ldots)}$$

In particular, the latter rule expresses that equality is a congruence with respect to complex expression formation. For example, if $C = \lambda$, it subsumes the ξ rule for conversion under a λ.

We can now instantiate this generic construction in Sect. 3.2 and 3.3 to add product and function types concisely.

3.2. Dependent Product Types

We obtain dependent product types by adding the following expression constructors, notations, and rules, where we identify substitutions γ and lists $\vec{E}$ of expressions.

Constructor	Complex expression	Notation
formation	$\texttt{prod}(\Gamma; \cdot)$	$\langle\Gamma\rangle$
introduction	$\texttt{tuple}(\cdot; \gamma)$	$\langle\gamma\rangle$
i-th projection	$\pi_i(\cdot; t)$	$t.i$

[2]The terminology for these types is not always unique: They are sometimes called dependent *sum* and dependent *product* types, respectively.

$$\frac{\Gamma \vdash_\Sigma \Gamma' \ \texttt{Ctx}}{\Gamma \vdash_\Sigma \langle \Gamma' \rangle : \texttt{type}}$$

$$\frac{\Gamma \vdash_\Sigma \gamma : \Gamma' \to \cdot}{\Gamma \vdash_\Sigma \langle \gamma \rangle : \langle \Gamma' \rangle} \qquad \frac{\Gamma \vdash_\Sigma t : \langle x_1 : A_1, \dots, x_n : A_n \rangle}{\Gamma \vdash_\Sigma t.i : A_i[t.1, \dots, t.(i-1)]}$$

$$\frac{\Gamma \vdash_\Sigma t_1, \dots, t_n : \Gamma' \to \cdot}{\Gamma \vdash_\Sigma \langle t_1, \dots, t_n \rangle . i = t_i} \qquad \frac{\Gamma \vdash_\Sigma t : \langle \Gamma' \rangle}{\Gamma \vdash_\Sigma \langle t.1, \dots, t.n \rangle = t}$$

To simplify some notations later on, our formulation uses n-ary products: Given a context Γ, Γ', the first rule makes $\langle \Gamma' \rangle$ a type over Γ. If $\Gamma' = x_1 : A_1, \dots, x_n : A_n$, our dependent product type $\langle \Gamma' \rangle$ is usually written as $\Sigma_{x_1:A_1} \dots \Sigma_{x_{n-1}:A_{n-1}} A_n$. Our notation also highlights the correspondence to record types. As usual, we write $A \times B$ for $\langle x : A, y : B \rangle$ if x does not occur free in B.

The second and third rule correspond to the usual introduction and elimination rules for n-ary products. In particular, tuples of type $\langle \Gamma' \rangle$ are substitution fragments γ such that $id_\Gamma, \gamma : \Gamma, \Gamma' \to \Gamma$.

As the special case $n = 0$, i.e., if Γ is empty, we obtain the unit type $\langle \cdot \rangle$ containing only the unique term $\langle \cdot \rangle$.

The last two rules are the usual conversions for product types. The first one is the β-style rule that computes a projection of a tuple; the second one is the η-style rule that represents any term of product type as the tuple of its projections.

3.3. Dependent Function Types

We obtain dependent function types by adding the following expression constructors, notations, and rules:

Constructor	Complex expression	Notation
formation	$\Pi(\Gamma; A)$	$\Pi_\Gamma A$
introduction	$\lambda(\Gamma; t)$	$\lambda_\Gamma t$
application	$@(\cdot; t, \gamma)$	$t\,\gamma$

$$\frac{\Gamma, \Gamma' \vdash_\Sigma A : \texttt{type}}{\Gamma \vdash_\Sigma \Pi_{\Gamma'} A : \texttt{type}}$$

$$\frac{\Gamma, \Gamma' \vdash_\Sigma t : A}{\Gamma \vdash_\Sigma \lambda_{\Gamma'} t : \Pi_{\Gamma'} A} \qquad \frac{\Gamma \vdash_\Sigma t : \Pi_{\Gamma'} A \quad \Gamma \vdash_\Sigma \gamma : \Gamma' \to \cdot}{\Gamma \vdash_\Sigma t\,\gamma : A[\gamma]}$$

$$\frac{\Gamma, \Gamma' \vdash_\Sigma t : A \quad \Gamma \vdash_\Sigma \gamma : \Gamma' \to \cdot}{\Gamma \vdash_\Sigma (\lambda_{\Gamma'} t)\,\gamma = t[\gamma]} \qquad \frac{\Gamma \vdash_\Sigma t : \Pi_{\Gamma'} A}{\Gamma \vdash_\Sigma \lambda_{\Gamma'} (t\, id_{\Gamma'}) = t}$$

This is also a generalized version that uses n-ary functions. If $\Gamma' = x_1 : A_1, \dots, x_n : A_n$, our dependent function type $\Pi_{\Gamma'} A$ is usually written $\Pi_{x_1:A_1} \dots \Pi_{x_n:A_n} A$.

For example, $(\lambda_\Gamma t)\,\gamma$ is a β-redex, whose argument signature is given by Γ and whose arguments are provided by $\gamma : \Gamma \to \cdot$, and β-reduction yields $t[\gamma]$. Similarly, $\lambda_{\Gamma'}(t\, id_{\Gamma'})$ is the η-expansion of t.

As usual, we write $A \to B$ for $\Pi_{x:A} B$ if x does not occur free in B.

3.4. Type Declarations and Kinds

So far we have not described how to add type declarations to a theory. Indeed, inspecting the inference system, we see that we can so far declare only typed constants $c : A$ but no

types $c : \texttt{type}$. This is intentional, and we obtain different type theories depending on what type declarations we allow.

We will only allow type declarations in theories and never in contexts. This is not necessary in general but very helpful for the purposes of this paper.

The simplest solution is to add the rule

$$\frac{\vdash \Sigma\ \texttt{Thy} \quad c \text{ not in } \Sigma}{\vdash \Sigma,\ c : \texttt{type}\ \texttt{Thy}}$$

which yields simple type theory.

To obtain dependent type theory, we add a constructor `kind` and a rule that allows kinded declarations:

$$\frac{\vdash \Sigma\ \texttt{Thy} \quad c \text{ not in } \Sigma \quad \cdot \vdash_\Sigma E : \texttt{kind}}{\vdash \Sigma,\ c : E\ \texttt{Thy}} \qquad \frac{}{\cdot \vdash_\Sigma \texttt{type} : \texttt{kind}}$$

Now we can fine-tune the dependent type theory by adding rules that form kinds. Specifically, we add the rules that arise from the ones in Sect. 3.3 by replacing `type` with `kind`. Thus, we add λ-abstraction over typed variables in kinded expressions.

Note that in the terminology of pure type systems [Berardi 1990], Sect. 3.3 describes the rule $(\texttt{type}, \texttt{type})$. Therefore, our replacement of `type` with `kind` means that we also add the rule $(\texttt{type}, \texttt{kind})$.

3.5. Running Example: A Type-Erasure Translation

We will exemplify our concepts using a type theory with all of the above features, i.e., kinded declarations and dependent product and function types. The resulting type theory can be seen as LF [Harper et al. 1993] with product types or as Martin-Löf type theory [Martin-Löf 1974] with only 2 universes and without inductive and identity types.

Syntax and Syntax Translation. The following are two theories in this type theory:

Example 3.1 (*First-Order Logic*). The theory `FOL` contains the following declarations:

$$\begin{array}{lcl}
\texttt{i} & : & \texttt{type} \\
\texttt{o} & : & \texttt{type} \\
\texttt{ded} & : & \texttt{o} \to \texttt{type} \\
\texttt{imp} & : & \texttt{o} \to \texttt{o} \to \texttt{o} \\
\texttt{forall} & : & (\texttt{i} \to \texttt{o}) \to \texttt{o}
\end{array}$$

The expressions $t : \texttt{i}$ and $F : \texttt{o}$ represent the terms and formulas of first-order logic. Given a formula $F : \texttt{o}$, the expressions $p : \texttt{ded}\, F$ represent the proofs of F. `imp` is the binary implication connective in curried form, and we will use infix notation for it. `forall` is the universal quantifier using higher-order abstract syntax.

We omit the other connectives, which are not crucial for our example.

Example 3.2 (*Sorted First-Order Logic*). The theory `SFOL` contains the following declarations:

$$\begin{array}{lcl}
\texttt{sort} & : & \texttt{type} \\
\texttt{tm} & : & \texttt{sort} \to \texttt{type} \\
\texttt{o} & : & \texttt{type} \\
\texttt{ded} & : & \texttt{o} \to \texttt{type} \\
\texttt{imp} & : & \texttt{o} \to \texttt{o} \to \texttt{o} \\
\texttt{forall} & : & \Pi_{s:\texttt{sort}}\, (\texttt{tm}\, s \to \texttt{o}) \to \texttt{o}
\end{array}$$

The expressions $s : \texttt{sort}$ represent the sorts, and the expressions $t : \texttt{tm}\, s$ represents the terms of sort s. The universal quantifier takes a sort argument and quantifies over that sort.

The following is a theory morphism between them:

Example 3.3 (*Type-Erasure*). The theory morphism $\texttt{ER} : \texttt{SFOL} \to \texttt{FOL}$ (erase and relativize) contains the following declarations:

$$\begin{array}{lcl}
\texttt{sort} & \mapsto & \texttt{i} \to \texttt{o} \\
\texttt{tm} & \mapsto & \lambda_{s:i\to\texttt{o}}\, i \\
\texttt{o} & \mapsto & \texttt{o} \\
\texttt{ded} & \mapsto & \texttt{ded} \\
\texttt{imp} & \mapsto & \texttt{imp} \\
\texttt{forall} & \mapsto & \lambda_{s:i\to\texttt{o}}\, \lambda_{F:i\to\texttt{o}}\, \texttt{forall}\, \lambda_{x:i}\, (s\, x)\, \texttt{imp}\, (F\, x)
\end{array}$$

It maps every $\texttt{SFOL}$-sort $s : \texttt{sort}$ to a unary $\texttt{FOL}$-predicate $\texttt{ER}(s) : \texttt{i} \to \texttt{o}$. Every $\texttt{SFOL}$-term $t : \texttt{tm}\, s$ is mapped to a $\texttt{FOL}$-term $\texttt{ER}(t) : \texttt{i}$, which is the essence of type erasure.

By applying the homomorphic extension and β-reduction, we see that an $\texttt{SFOL}$-formula $\texttt{forall}\, s\, \lambda_{x:\texttt{tm}\, s}\, F(x)$ is mapped to the $\texttt{FOL}$-formula $\texttt{forall}\, \lambda_{x:\texttt{i}}\, (\texttt{ER}(s)\, x)\, \texttt{imp}\, (\texttt{ER}(F(x)))$, which relativizes the sorted quantification at s to the predicate $\texttt{ER}(s)$.

Proofs and the Failure of a Proof Translation. To verify the logic translation of Ex. 3.3, we want to show that if F is a theorem of $\texttt{SFOL}$, then $\texttt{ER}(F)$ is a theorem of $\texttt{FOL}$. Proof-theoretically, this means that if there is an $\texttt{SFOL}$-expression of type $\texttt{ded}\, F$, then there is also a $\texttt{FOL}$-expression of type $\texttt{ded}\, \texttt{ER}(F)$. Currently, of course, this holds vacuously simply because neither $\texttt{SFOL}$ nor $\texttt{FOL}$ contains any representation of the proof rules.

Therefore, we extend $\texttt{FOL}$ and $\texttt{SFOL}$ as follows. These representations are standard (see, e.g., [Harper et al. 1993]):

Example 3.4 (*First-Order Logic Proof Theory*). The theory $\texttt{FOLPF}$ extends $\texttt{FOL}$ with the following declarations:

$$\begin{array}{lcl}
\texttt{imp}_I & : & \Pi_{F:\texttt{o}}\, \Pi_{G:\texttt{o}}\, (\texttt{ded}\, F \to \texttt{ded}\, G) \to \texttt{ded}\, (F\, \texttt{imp}\, G) \\
\texttt{imp}_E & : & \Pi_{F:\texttt{o}}\, \Pi_{G:\texttt{o}}\, \texttt{ded}\, (F\, \texttt{imp}\, G) \to \texttt{ded}\, F \to \texttt{ded}\, G \\
\texttt{forall}_I & : & \Pi_{F:\texttt{i}\to\texttt{o}}\, (\Pi_{x:i}\, \texttt{ded}\, (F\, x)) \to \texttt{ded}\, (\texttt{forall}\, \lambda_{x:\texttt{i}}\, (F\, x)) \\
\texttt{forall}_E & : & \Pi_{F:\texttt{i}\to\texttt{o}}\, \texttt{ded}\, (\texttt{forall}\, \lambda_{x:\texttt{i}}\, (F\, x)) \to \Pi_{x:\texttt{i}}\, \texttt{ded}\, (F\, x)
\end{array}$$

These represent the natural deduction introduction and elimination rules for implication and universal quantification. $\texttt{imp}_I$ constructs a proof of $F\, \texttt{imp}\, G$ if given a proof of G under the assumption that F holds. $\texttt{imp}_E$ represents modus ponens. $\texttt{forall}_I$ and $\texttt{forall}_E$ work accordingly.

Example 3.5 (*Sorted First-Order Logic Proof Theory*). The theory $\texttt{SFOLPF}$ contains the following declarations:

$$\begin{array}{lcl}
\texttt{imp}_I & : & \Pi_{F:\texttt{o}}\, \Pi_{G:\texttt{o}}\, (\texttt{ded}\, F \to \texttt{ded}\, G) \to \texttt{ded}\, (F\, \texttt{imp}\, G) \\
\texttt{imp}_E & : & \Pi_{F:\texttt{o}}\, \Pi_{G:\texttt{o}}\, \texttt{ded}\, (F\, \texttt{imp}, G) \to \texttt{ded}\, F \to \texttt{ded}\, G \\
\texttt{forall}_I & : & \Pi_{s:\texttt{sort}}\, \Pi_{F:\texttt{tm}\, s\to\texttt{o}}\, (\Pi_{x:\texttt{tm}\, s}\, \texttt{ded}\, (F\, x)) \to \texttt{ded}\, (\texttt{forall}\, s\, \lambda_{x:\texttt{tm}\, s}\, (F\, x)) \\
\texttt{forall}_E & : & \Pi_{s:\texttt{sort}}\, \Pi_{F:\texttt{tm}\, s\to\texttt{o}}\, \texttt{ded}\, (\texttt{forall}\, s\, \lambda_{x:\texttt{tm}\, s}\, (F\, x)) \to \Pi_{x:\texttt{tm}\, s}\, \texttt{ded}\, (F\, x)
\end{array}$$

These declarations are very similar to the ones in $\texttt{FOLPF}$. The only difference is that the rules for $\texttt{forall}$ work with an arbitrary sort.

To verify the translation, it would be sufficient to extend $\texttt{ER}$ to a theory morphism $\texttt{SFOLPF} \to \texttt{FOLPF}$. Then the judgment preservation property would guarantee that $\texttt{ER}$ preserves theoremhood. But we cannot give such a morphism.

The problem is the case for $\texttt{forall}_E$: We would have to map $\texttt{forall}_E$ to a term of type

$$\Pi_{s:\texttt{i}\to\texttt{o}}\, \Pi_{F:\texttt{i}\to\texttt{o}}\, \texttt{ded}\, (\texttt{forall}\, \lambda_{x:\texttt{i}}\, (s\, x)\, \texttt{imp}\, (F\, x)) \to \Pi_{x:\texttt{i}}\, \texttt{ded}\, (F\, x)$$

This means we have to prove $F\,x$ for an arbitrary x, but our assumption about F only states $(s\,x)\,\texttt{imp}\,(F\,x)$ – we are missing the assumption $s\,x$. A similar problem would occur with the existential introduction rule.

Indeed, the type-erasure translation of Ex. 3.3 satisfies the invariant that every SFOL-term $t : \texttt{tm}\,s$ is mapped to a FOL-term $\texttt{ER}(t) : \texttt{i}$ *such that* $\texttt{ER}(s)\,\texttt{ER}(t)$ *holds*. This invariant recovers exactly the type information that ER erases. Without formalizing this invariant, we cannot verify the translation.

At first sight, an alternative solution seems possible:

Example 3.6. Consider a theory morphism $\texttt{ER}' : \texttt{SFOL} \to \texttt{FOL}$ that contains the declarations:

$$\begin{array}{lcl} \texttt{sort} & \mapsto & \texttt{i} \to \texttt{o} \\ \texttt{tm} & \mapsto & \lambda_{s:\texttt{i}\to\texttt{o}}\,\langle \_ : \texttt{i},\, ! : \texttt{ded}\,(s\,\_)\rangle \\ \texttt{o} & \mapsto & \texttt{o} \\ \texttt{ded} & \mapsto & \texttt{ded} \end{array}$$

where we use _ and ! as convenient variable names.

$\texttt{ER}'$ maps every term $t : \texttt{tm}\,s$ to a pair $\langle u, p\rangle$ such that $u : \texttt{i}$ is the intended translation of t and $p : \texttt{ded}\,(\texttt{ER}'(s)\,u)$ is the proof of the invariant for t.

This approach already fails at the case for `forall`: The type of $\texttt{ER}'(\texttt{forall})$ would have to be

$$\Pi_{s:\texttt{i}\to\texttt{o}}\,\Pi_{F:\langle \_:\texttt{i},\,!:\texttt{ded}\,(s\,\_)\rangle\to\texttt{o}}\,\texttt{o}$$

To give a term of that type, we have to build a formula out of F. But we can only apply F to pairs of a $t : \texttt{i}$ and a proof of $s\,t$. In FOL, there is no way for us to come by such a proof. (And even if there were, the resulting translation function would not be the same as the one we set out to formalize.)

Using the results of this paper, we can state the above dilemma very simply: The type-erasure translation is a *lax* theory morphism. Therefore, the above attempts of representing it as a *strict* theory morphism must fail.

Throughout this paper, we will construct the lax theory morphism that represents and verifies the type-erasure translation. This construction will involve multiple complex steps, which we introduce and analyze generally before applying them to our running example. The example is continued in Ex. 5.13, 5.19, and 6.1.

4. LAX MORPHISMS

4.1. Formal Definition

Lax morphisms as Syntax Transformations. To motivate our definition of lax morphisms, we first revisit some properties of strict morphisms. Let us first define an auxiliary concept:

Definition 4.1. A **syntax transformation** Φ from a theory Σ to a theory Σ' is a triple of mappings that map
— Σ-contexts Γ to Σ'-contexts Γ^Φ,
— Σ-substitutions $\gamma : \Gamma \to \Gamma'$ to Σ'-substitutions $\gamma^\Phi : \Gamma^\Phi \to \Gamma'^\Phi$,
— Σ-terms/types E in context Γ to Σ'-terms/types E^{Φ_Γ} in context Γ^Φ,
such that
— for Σ-contexts $\vdash_\Sigma \Gamma\ \texttt{Ctx}$

$$\begin{aligned} .^\Phi &= . \\ (\Gamma,\ x : A)^\Phi &= \Gamma^\Phi,\ x : A^{\Phi_\Gamma} \end{aligned}$$

— for Σ-substitutions $\vdash_\Sigma \gamma : \Gamma' \to \Gamma$

$$\begin{aligned} .^\Phi &= . \\ (\gamma,\ t)^\Phi &= \gamma^\Phi,\ t^{\Phi_\Gamma} \end{aligned}$$

Note that a syntax transformation is already uniquely determined if we fix E^{Φ_Γ} for all Γ and E. Conversely, E^{Φ_Γ} is already uniquely determined if we fix the action of Φ on contexts and substitutions.

Notation 4.2. We can usually omit Γ in E^{Φ_Γ} because it will be clear from E. But we will only do so in a few examples and not in definitions or theorems.

The usual (i.e., strict) theory morphisms Φ are syntax transformations. Moreover, they have the following properties:

(i) Φ satisfies the substitution lemma, i.e., the mapping of expressions commutes with substitution.
(ii) Φ is determined by its action on Σ-constants via homomorphic extension.
(iii) Φ preserves all judgments, in particular the typing of expressions, e.g., $\Gamma \vdash_\Sigma t : A$ implies $\Gamma^\Phi \vdash_{\Sigma'} t^{\Phi_\Gamma} : A^{\Phi_\Gamma}$.

The main theorem about strict morphisms (see, e.g., [Rabe 2014]) states that Property (ii) implies Property (i) and allows inductively proving Property (iii). The central observation behind the present paper is that the converse does not hold: We can give syntax transformations Φ that satisfy Properties (i) and (iii) but not Property (ii). Therefore, we introduce the name *lax* morphisms for syntax transformations that satisfy Property (i) and (iii) but not necessarily (ii), and we use the name *strict* morphisms for the special case where (ii) is satisfied as well.

Example 4.3. Consider a type theory with a unit type $\langle\cdot\rangle$ and let Σ and Σ' be arbitrary theories. We define a syntax transformation U by
— U maps every Σ-type A to the unit type $\langle\cdot\rangle$, i.e., $A^{U_\Gamma} = \langle\cdot\rangle$.
— U maps every Σ-term t to the unique term $\langle\cdot\rangle$ of type $\langle\cdot\rangle$, i.e., $t^{U_\Gamma} = \langle\cdot\rangle$.
This already determines the action of U on contexts and substitutions, e.g., $(x : A, y : B)^U = x : \langle\cdot\rangle, y : \langle\cdot\rangle$.

U clearly satisfies Properties (i) and (iii). But except for degenerate cases, U does not satisfy Property (ii): For example, assume a type theory with function types and consider $\Sigma = a : \texttt{type}, b : \texttt{type}$. Then we have $(a \to b)^{U_\Gamma} = \langle\cdot\rangle \neq \langle\cdot\rangle \to \langle\cdot\rangle = a^{U_\Gamma} \to b^{U_\Gamma}$, i.e., U is not a homomorphic extension.

Thus U is a lax morphism that is not strict. The formal definition of U will be a special case of Def. 4.14.

Equipped with this intuition, we can state our main definition:

Definition 4.4 (*Lax Morphism*). A **lax morphism** from a theory Σ to a theory Σ' is a syntax transformation Φ such that $-^\Phi$ defines a functor $\texttt{Cont}(\Sigma) \to \texttt{Cont}(\Sigma')$.

Theorem 4.5 (Characterization of Lax Morphisms). *Assume a syntax transformation $\Phi : \Sigma \to \Sigma'$. Then the following are equivalent:*

(1) Φ is a lax morphism.
(2) Φ preserves variables, judgments, and substitution, i.e., the following rules are admissible for all terms/types:

$$\frac{\vdash_\Sigma \Gamma, x : A\ \texttt{Ctx}}{(\Gamma, x : A)^\Phi \vdash_{\Sigma'} x^{\Phi_{\Gamma, x:A}} = x}$$

$$\frac{\Gamma \vdash_\Sigma t : A}{\Gamma^\Phi \vdash_{\Sigma'} t^{\Phi_\Gamma} : A^{\Phi_\Gamma}} \qquad \frac{\Gamma \vdash_\Sigma s = t}{\Gamma^\Phi \vdash_{\Sigma'} s^{\Phi_\Gamma} = t^{\Phi_\Gamma}}$$

$$\frac{\vdash_\Sigma \gamma : \Gamma \to \Gamma' \qquad \Gamma \vdash_\Sigma t : A}{\Gamma'^\Phi \vdash_{\Sigma'} t^{\Phi_\Gamma}[\gamma^\Phi] = t[\gamma]^{\Phi_{\Gamma'}}}$$

where we put $\mathtt{type}^{\Phi_\Gamma} = \mathtt{type}$.

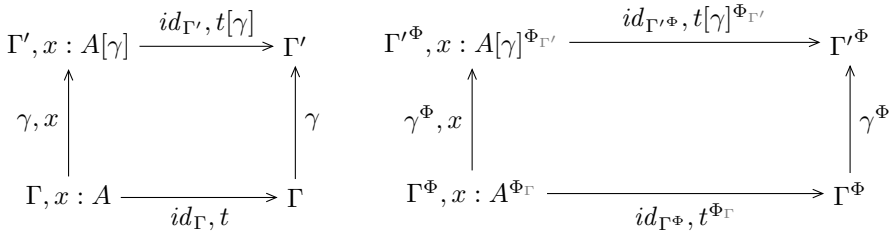
Proof. Assume Φ to be a lax morphism.

Firstly, the preservation of equality must hold for all terms and types anyway because Φ must preserve equality to be a well-defined functor.

Secondly, the preservation of variables follows from the following equality of substitutions out of a context $\Gamma, x : A$:

$$id_{\Gamma^\Phi}, x = id_{(\Gamma, x:A)^\Phi} = (id_{\Gamma, x:A})^\Phi = (id_\Gamma, x)^\Phi = id_{\Gamma^\Phi}, x^{\Phi_{\Gamma, x:A}}$$

Thirdly, to prove the preservation of typing and substitution, we assume $\Gamma \vdash_\Sigma t : A$ and distinguish two cases. In the first case, A is a type, i.e., $t : A : \mathtt{type}$. Then the left diagram below commutes, and thus by the properties of Φ also the right one:



Inspecting the right diagram at x yields $t^{\Phi_\Gamma}[\gamma^\Phi] = t[\gamma]^{\Phi_{\Gamma'}}$, which yields the preservation of substitution and typing.

In the second case, we have $A = \mathtt{type}$, i.e., $t : \mathtt{type}$. Then $\Gamma^\Phi \vdash_{\Sigma'} t^{\Phi_\Gamma} : \mathtt{type}$ holds because $(\Gamma, x : t)^\Phi$ must be a well-formed context. Similarly, the preservation of substitution follows because $(\gamma, x)^\Phi$ from the above diagram must be a well-formed substitution.

Conversely, assume the preservation properties. The functor is already determined uniquely and well-defined due to the preservation of typing and equality. So we only have to prove the functoriality.

The identity law $\vdash_{\Sigma'} id_\Gamma{}^\Phi = id_{\Gamma^\Phi} : \Gamma^\Phi \to \Gamma^\Phi$ follows from the preservation of variables. For the composition law, consider $\vdash_\Sigma \gamma' : \Gamma'' \to \Gamma'$ and $\vdash_\Sigma \gamma : \Gamma' \to \Gamma$ with $\gamma' = t_1, \ldots, t_n$. Then,

$$\vdash_{\Sigma'} (\gamma \circ \gamma')^\Phi = t_1[\gamma]^{\Phi_\Gamma}, \ldots, t_n[\gamma]^{\Phi_\Gamma} : \Gamma''^\Phi \to \Gamma^\Phi$$

and

$$\vdash_{\Sigma'} \gamma^\Phi \circ \gamma'^\Phi = t_1^{\Phi_{\Gamma'}}[\gamma^\Phi], \ldots, t_n^{\Phi_{\Gamma'}}[\gamma^\Phi] : \Gamma''^\Phi \to \Gamma^\Phi$$

and the two right hand sides are equal due to the preservation of substitution. □

Thus, lax morphisms retain some properties of strict morphisms: They map contexts and substitutions homomorphically, they map variables to themselves, they commute with substitution, and they preserve judgments. But the action of a lax morphism on an expression is not defined homomorphically – it can be defined arbitrarily as long as it satisfies the preservation properties.

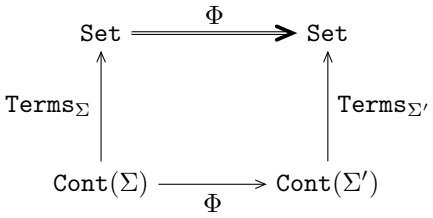
Remark 4.6 (*Laxness for Terms vs. Types*). The requirement that lax morphisms $\Phi : \Sigma \to \Sigma'$ commute with substitution may appear to imply the homomorphic extension property. Indeed, we can reason as follows. Consider a Σ-constant $c : A \to B$. We define a Σ'-term $c' := \lambda_{x:A^\Phi}(c\,x)^{\Phi_{x:A}}$. Now we have $c^\Phi : (A \to B)^\Phi$ and $c' : A^\Phi \to B^\Phi$, and the preservation of substitution yields $(c\,t)^\Phi = c'\,t^\Phi$. Thus, we might conjecture that Φ is strict and maps $c \mapsto c'$ for all Σ-constants c.

The problem with this argument is that it cannot be generalized to all cases because contexts may not contain type variables. For example, we cannot construct $(a \to b)^{\Phi_{a:\mathtt{type},b:\mathtt{type}}}$ and use substitution to deduce information about $(A \to B)^\Phi$. Indeed, it does not hold in general that $(A \to B)^\Phi = A^\Phi \to B^\Phi$, let alone that $c^\Phi = c'$.

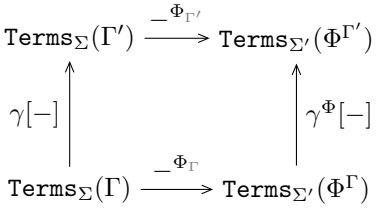
Thus, the preservation of substitution has consequences that resemble the homomorphic extension property, but it does not imply it. This provides a good trade-off between unrestricted and too restricted syntax transformations.

Naturality of Lax Morphisms. For readers familiar with natural transformations, the following result may provide a deeper intuition: We can think of a lax morphism as a natural transformation "on itself". In particular, the naturality law exactly captures the intuition that substitution commutes with lax morphisms. The following makes this precise:

THEOREM 4.7. *Given a lax morphism $\Phi : \Sigma \to \Sigma'$, the Γ-indexed family of maps $(t, A) \mapsto (t^{\Phi_\Gamma}, A^{\Phi_\Gamma})$ is a natural transformation $\mathtt{Terms}_\Sigma \Rightarrow \mathtt{Terms}_{\Sigma'} \circ \Phi : \mathtt{Cont}(\Sigma) \to \mathtt{Set}$.*



PROOF. We have to show that the diagram below commutes for every substitution $\vdash_\Sigma \gamma : \Gamma \to \Gamma'$, which follows immediately from the preservation of substitution.



□

Lax Morphisms as Mappings of All Expressions. So far, we have defined lax morphism as maps of contexts, substitutions, types, and terms. But it is more convenient if we can think of them as maps of all Σ-syntax to Σ'-syntax. Therefore, we want to extend $-^{\Phi_\Gamma}$ to
— context and substitution fragments as introduced in Not. 2.1,
— kinds and type families as introduced in Sect. 3.4.
Therefore, we extend Def. 4.4 as follows:

Definition 4.8. Consider a lax morphism $\Phi : \Sigma \to \Sigma'$.

— For $\Gamma \vdash_\Sigma \Gamma'\ \mathtt{Ctx}$, we have $(\Gamma, \Gamma')^\Phi = \Gamma^\Phi, \Delta$ for some Δ, and we define $\Gamma'^{\Phi_\Gamma} = \Delta$.

— For $\Gamma \vdash_\Sigma \gamma : \Gamma_1 \to \Gamma_2$, we have $(id_\Gamma, \gamma)^\Phi = id_{\Gamma^\Phi}, \delta$ for some δ, and we define $\gamma^{\Phi_\Gamma} = \delta$.
— For kinds, we define

$$\begin{array}{ll} \texttt{kind}^{\Phi_\Gamma} & = \texttt{kind} \\ \texttt{type}^{\Phi_\Gamma} & = \texttt{type} \\ (\Pi_{\Gamma'} K)^{\Phi_\Gamma} & = \Pi_{\Gamma'^{\Phi_\Gamma}} K^{\Phi_{\Gamma,\Gamma'}} \\ (\lambda_{\Gamma'} A)^{\Phi_\Gamma} & = \lambda_{\Gamma'^{\Phi_\Gamma}} A^{\Phi_{\Gamma,\Gamma'}} \\ (A\,\gamma)^{\Phi_\Gamma} & = \lambda_{\Gamma'}(A\,(\gamma, id_{\Gamma'}))^{\Phi_\Gamma} \qquad \text{if } \Gamma \vdash_\Sigma A\,\gamma : \Pi_{\Gamma'} K \end{array}$$

Essentially, all concepts are mapped using the homomorphic extension. There is only one exception: We perform η-expansion in the case for $A\,\gamma$ so that the definition eventually recurses into an atomic type. This is necessary because $(A\,\gamma)^{\Phi_\Gamma}$ is already defined if $A\,\gamma$ is atomic,

Correspondingly, we extend Thm. 4.5 as follows:

THEOREM 4.9. *A lax morphism $\Sigma \to \Sigma'$ satisfies the judgment preservation properties from Thm. 4.5 also if the involved expressions are type families or kinds.*

PROOF. The proof is straightforward. □

4.2. Concrete Lax Morphisms

We will now construct some lax morphisms.

Lax Morphisms Induced by Homomorphic Extension. We recover strict morphisms as a special case of lax morphisms:

THEOREM 4.10. *The homomorphic extension of $\vdash \sigma : \Sigma \to \Sigma'$ is a lax morphism.*

PROOF. Preservation of equality, judgments, and variables follow immediately from the definition of the homomorphic extension.

The preservation of substitution can be seen easily by regarding contexts as special cases of theories and substitutions as special cases of strict morphisms. □

We call a lax morphism **strict** if it arises as the homomorphic extension of some σ. Not all lax morphisms are strict as we will see from the examples below.

Rem. 4.6 essentially says that laxness does not matter for typed subexpressions. One consequence is the following result: If the translation of types is already fixed, then lax morphisms are determined by their action on the remaining typed constants.

Definition 4.11 (*Strict Extension of a Lax Morphism*). Consider a lax morphism $\Phi : \Sigma \to \Sigma'$ and an extension Σ, Γ of Σ such that Γ declares only typed constants (i.e., Γ is essentially a context). Moreover, consider a strict morphism $\vdash id_{\Sigma'}, \gamma : \Sigma', \Gamma^\Phi \to \Sigma'$.

Then we define a lax morphism $\Phi, \gamma : \Sigma, \Gamma \to \Sigma'$ by

$$\Delta^{\Phi,\gamma} = \Delta^{\Phi_\Gamma}[\gamma] \qquad \text{for } \vdash_{\Sigma,\Gamma} \Delta\ \texttt{Ctx}$$

$$\delta^{\Phi,\gamma} = \delta^{\Phi_\Gamma}[\gamma] \qquad \text{for } \vdash_{\Sigma,\Gamma} \delta : \Delta \to \Delta'$$

Remark 4.12 (*Running Example*). Our running example crucially applies Def. 4.11. Unfortunately, it applies Def. 4.11 to the lax morphism constructed only in Ex. 5.19. Therefore, we bump the example to Sect. 6.

THEOREM 4.13. *In the situation of Def. 4.11, Φ, γ is a lax morphism $\Sigma, \Gamma \to \Sigma'$ that agrees with Φ on $\texttt{Cont}(\Sigma)$. Vice versa, every lax morphism $\Sigma, \Gamma \to \Sigma$ that agrees with Φ on $\texttt{Cont}(\Sigma)$ is of the form Φ, γ for some γ.*

In particular, every lax morphism $\Sigma, \Gamma \to \Sigma$ that agrees with $id_\Sigma : \Sigma \to \Sigma$ is strict.

PROOF. Regarding the first claim, it is clear from the definition that Φ and Φ, γ agree on Σ. Moreover, the functoriality follows immediately because we are only composing two functors.

Thus, it is enough to show that Φ, γ is well-defined. Because Γ can be regarded as a Σ-context, we obtain
— Γ^Φ can be regarded as a Σ'-context
— γ can be regarded as a substitution and then $\vdash_{\Sigma'} \gamma : \Gamma^\Phi \to \cdot$
— $\vdash_{\Sigma,\Gamma} \Delta$ `Ctx` is equivalent to $\Gamma \vdash_\Sigma \Delta$ `Ctx`
— $\vdash_{\Sigma,\Gamma} \delta : \Delta \to \Delta'$ is equivalent to $\Gamma \vdash_\Sigma \delta : \Delta \to \Delta'$
For contexts Δ, we obtain first $\Gamma^\Phi \vdash_{\Sigma'} \Delta^{\Phi_\Gamma}$ `Ctx` and then $\vdash_{\Sigma'} \Delta^{\Phi_\Gamma}[\gamma]$ `Ctx` as needed. Accordingly, for substitutions δ, we obtain first $\Gamma^\Phi \vdash_{\Sigma'} \delta^{\Phi_\Gamma} : \Delta^{\Phi_\Gamma} \to \Delta'^{\Phi_\Gamma}$ and then $\vdash_{\Sigma'} \delta^{\Phi_\Gamma}[\gamma] : \Delta^{\Phi_\Gamma}[\gamma] \to \Delta'^{\Phi_\Gamma}[\gamma]$ as needed.

Regarding the second claim, given $\Psi : \Sigma, \Gamma \to \Sigma$, we define γ by $c[\gamma] = c^\Psi$ for every c declared in Γ. Seen as a substitution, γ is well-typed because Ψ preserves typing.

Thus, we only have to show that $\Phi, \gamma = \Psi$. Consider a Σ, Γ-expression E, which we assume to be closed for simplicity. Let Γ' be a copy of Γ regarded as a context, and let E' be a copy of E regarded as a Σ, Γ-expression in context Γ', and let $\vdash_{\Sigma,\Gamma} r : \Gamma' \to \cdot$ be the substitution mapping every x of Γ' to its counterpart of Γ. Then E' does not use any constants declared in Γ and $E = E'[r]$. Because Ψ preserves substitution and agrees with Φ on E', the action of Ψ on E is already determined, and we can compute it to be $E^\Phi[\gamma]$. In the general case where E has free variables, the argument proceeds accordingly.

The third claim follows immediately because id_Σ, γ is strict. □

The situation of Thm. 4.13 is relatively common when dependent type theory is used as a logical framework like in LF [Harper et al. 1993]. Then all type declarations occur in a fixed theory Σ, which represents a logic L, and the L-theories are represented as LF-theories Σ, Γ. That was one of our initial motivations to consider lax morphisms at all.

Lax Morphisms Induced by Product Types. Let us assume a type theory with dependent product types as in Sect. 3.2.[3] Then, given lax morphisms Φ^1, Φ^2, we can define their product "point-wise": If $t : A$, we put

$$A^{\Phi^1 \times \Phi^2} = A^{\Phi^1} \times A^{\Phi^2} \qquad \text{and} \qquad t^{\Phi^1 \times \Phi^2} = (t^{\Phi^1}, t^{\Phi^2}).$$

Note that $\Phi^1 \times \Phi^2$ is not strict even if Φ^1 and Φ^2 are because, e.g.,

$$(A \to B)^{\Phi^1 \times \Phi^2} = (A \to B)^{\Phi^1} \times (A \to B)^{\Phi^2} \neq (A^{\Phi^1} \times A^{\Phi^2}) \to (B^{\Phi^1} \times B^{\Phi^2}) = A^{\Phi^1 \times \Phi^2} \to B^{\Phi^1 \times \Phi^2}.$$

The formal definition is a bit more involved because it has to account for free variables:

Definition 4.14 (*Products*). Consider a list of lax morphisms $\Phi^i : \Sigma \to \Sigma'$ for $i = 1, \ldots, n$. By mutual induction, we define a lax morphism $\Phi : \Sigma \to \Sigma'$ and substitutions $\varphi^i_\Gamma : \Gamma^{\Phi^i} \to \Gamma^\Phi$:

$$A^{\Phi_\Gamma} = \langle \ldots, x_i : A^{\Phi^i_\Gamma}[\varphi^i_\Gamma], \ldots \rangle$$

$$t^{\Phi_\Gamma} = \langle \ldots, t^{\Phi^i_\Gamma}[\varphi^i_\Gamma], \ldots \rangle$$

$$\varphi^i_\cdot = \cdot \qquad \text{and} \qquad \varphi^i_{\Gamma, x:A} = \varphi^i_\Gamma, x.i.$$

We denote this lax morphism by $\Phi^1 \times \ldots \times \Phi^n$.

[3] Actually, simple product types would be sufficient here.

Def. 4.14 is a lot simpler than its formal statement makes it appear. Intuitively, Φ maps every type A to the n-ary product of the A^{Φ^i} and every term t to the n-tuple consisting of the t^{Φ^i}. But if t or A contain free variables, we have to be careful as Γ^Φ and Γ^{Φ^i} are not the same context: The former contains variables of product type. Therefore, the substitutions φ^i_Γ map every variable x declared in Γ^{Φ^i} to the corresponding projection $x.i$, which is valid over Γ^Φ. With this intuition, it is easy to prove that Φ is well-defined:

THEOREM 4.15. *In the situation of Def. 4.14, Φ is a lax morphism.*

PROOF. The preservation of typing is obvious and the preservation of equality and substitution straightforward. The preservation of variables follows by observing that

$$x^{\Phi_\Gamma} = \langle \dots, x^{\Phi^i_\Gamma}[\varphi^i_\Gamma], \dots \rangle$$

and

$$\Gamma^\Phi \vdash_{\Sigma'} x^{\Phi^i_\Gamma}[\varphi^i_\Gamma] = x.i$$

and using the conversion rule for products. □

Note that both our definition of product types in Sect. 3.2 and Def. 4.14 include the case $n = 0$. The empty product of types yields a unit type $\langle\cdot\rangle$ with a unique term $\langle\cdot\rangle$. The empty product of lax morphism maps every type to the unit type and every term to the unit term. This is the simplest example of a lax morphism that is not strict.

Remark 4.16. Lax morphisms are closed under finite products if the underlying type theory is. We sketch the proof of the universal property in Sect. 7.3. We expect the same for any limit in the category of types that the syntax can express.

Lax Morphisms Induced by Function Types. Now assume a type theory with dependent function types as in Sect. 3.3.[4] Function types yield exponentials in the category of types, but we do not expect that function types induce exponentials of lax morphisms. But they do at least induce "constant powers":

Definition 4.17 (*Constant Powers*). Consider a lax morphism $\Phi : \Sigma \to \Sigma'$ and a context $\vdash_{\Sigma'} \Delta$ `Ctx`. By mutual induction, we define a functor $\Phi^\Delta : \mathtt{Cont}(\Sigma) \to \mathtt{Cont}(\Sigma')$ and substitutions $\varphi_\Gamma : \Gamma^\Phi \to \Phi^\Delta(\Gamma), \Delta$:

$$A^{\Phi^\Delta_\Gamma} = \Pi_\Delta A^{\Phi_\Gamma}[\varphi_\Gamma]$$

$$t^{\Phi^\Delta_\Gamma} = \lambda_\Delta t^{\Phi_\Gamma}[\varphi_\Gamma]$$

$$\varphi_\cdot = \cdot \qquad \text{and} \qquad \varphi_{\Gamma,x:A} = \varphi_\Gamma, x\ id_\Delta$$

THEOREM 4.18. *In the situation of Def. 4.17, Φ^Δ is a lax morphism.*

PROOF. The preservation of typing is obvious and the preservation of equality and substitution straightforward. The preservation of variables follows by observing that

$$x^{\Phi^U_\Gamma} = \lambda_\Delta x^{\Phi_\Gamma}[\varphi_\Gamma]$$

and

$$\Gamma^\Phi \vdash_{\Sigma'} x^{\Phi^\Delta_\Gamma}[\varphi_\Gamma] = x\ id_\Delta$$

and using the conversion rules for functions. □

[4] Actually, simple function types would be sufficient here.

4.3. Lax Morphisms and Type Constructors

Due to the homomorphic property, strict morphisms commute with all constructors, in particular the ones from Sect. 3. We will now show that, even though this requirement is waived for lax morphisms, there are some valuable relationships between lax morphisms and constructors.

Product Types. We will show that lax morphisms commute with dependent products up to a canonical isomorphism, i.e., $\langle\Gamma\rangle^\Phi$ and $\langle\Gamma^\Phi\rangle$ are isomorphic types. That makes sense because dependent products are essentially contexts and lax morphisms commute with context formation.

The following makes this precise:

Definition 4.19. In any category, we say that A and B are *canonically isomorphic* and write $A \overset{!}{\cong} B$ iff there is a canonical choice for an isomorphism between A and B.

THEOREM 4.20. *For every lax morphism* $\Phi : \Sigma \to \Sigma'$ *and every* $\Gamma \vdash_\Sigma \Gamma'$ `Ctx`

$$\langle\Gamma'\rangle^{\Phi_\Gamma} \overset{!}{\cong} \langle\Gamma'^{\Phi_\Gamma}\rangle$$

in the category of types in context Γ^Φ.

PROOF. We know that the $\Gamma' \overset{!}{\cong} x : \langle\Gamma'\rangle$ in the category of Σ-contexts that extend Γ. The canonical isomorphism is the substitution that maps each x_i in Γ' to $x.i$. Similarly, we have $\Gamma'^{\Phi_\Gamma} \overset{!}{\cong} x : \langle\Gamma'^{\Phi_\Gamma}\rangle$.

Since lax morphisms are functors, they preserve isomorphisms so that $\Gamma'^{\Phi_\Gamma} \overset{!}{\cong} (x : \langle\Gamma'\rangle)^{\Phi_\Gamma}$. Then the canonical isomorphism is obtained by composition. □

Function Types. Contrary to product types, dependent function types do not commute with lax morphisms, i.e., for a Σ-type $\Pi_\Gamma A$ (here assumed to be closed for simplicity), the Σ'-types $(\Pi_\Gamma A)^\Phi$ and $\Pi_{\Gamma^\Phi} A^{\Phi_\Gamma}$ are not in general isomorphic. However, we can show that $(\Pi_\Gamma A)^\Phi$ is canonically embedded into $\Pi_{\Gamma^\Phi} A^{\Phi_\Gamma}$.

The following theorem makes this precise:

THEOREM 4.21. *For every lax morphism* $\Phi : \Sigma \to \Sigma'$ *and every term* $\Gamma^\Phi \vdash_\Sigma f : (\Pi_{\Gamma'} A)^{\Phi_\Gamma}$, *there is a definable term*

$$\Gamma^\Phi \vdash_{\Sigma'} f^* : \Pi_{\Gamma'^{\Phi_\Gamma}} A^{\Phi_{\Gamma,\Gamma'}}$$

and thus a canonical function

$$(\Pi_{\Gamma'} A)^{\Phi_\Gamma} \to \Pi_{\Gamma'^{\Phi_\Gamma}} A^{\Phi_{\Gamma,\Gamma'}}$$

PROOF. We apply Φ to the judgment

$$\Gamma,\ x : \Pi_{\Gamma'} A,\ \Gamma' \vdash_\Sigma x\ id_{\Gamma'} : A$$

This yields

$$\Gamma^\Phi,\ x : (\Pi_{\Gamma'} A)^{\Phi_\Gamma},\ \Gamma'^{\Phi_\Gamma} \vdash_{\Sigma'} (x\ id_{\Gamma'})^{\Phi_{\Gamma,\Delta}} : A^{\Phi_{\Gamma,\Gamma'}} \qquad \text{with} \qquad \Delta = x : (\Pi_{\Gamma'} A)^{\Phi_\Gamma}, \Gamma'$$

Here, we have already "cleaned up the notation" by removing irrelevant variable declarations from the contexts in X^{Φ_Δ}. This is possible because the preservation of substitution implies $X^{\Phi_{\Delta,x:B}} = X^{\Phi_\Delta}$ if x does not occur free in X.

Now after λ-abstraction over Γ'^{Φ_Γ} and substituting f for x, we put

$$f^* \ := \ \big(\lambda_{\Gamma'^{\Phi_\Gamma}} (x\ id_{\Gamma'})^{\Phi_{\Gamma,\Delta}}\big)[f]$$

□

Notation 4.22. In the situation of Thm. 4.21, f^* depends on the context and on the type of f, which we omit in the notation. Moreover, f^* depends on Φ – if that is not clear from the context, we write $f^{*\Phi}$.

Example 4.23. Thm. 4.21 is more intuitive if we construct $(f^{\Phi_\Gamma})^*$ for some $\Gamma \vdash_\Sigma f : \Pi_{\Gamma'} A$: We have (with Δ defined as in the proof of Thm. 4.21)

$$(f^{\Phi_\Gamma})^* \quad = \quad \big(\lambda_{\Gamma'^{\Phi_\Gamma}} (x\ id_{\Gamma'})^{\Phi_{\Gamma,\Delta}}\big)[f^{\Phi_\Gamma}] \quad = \quad \lambda_{\Gamma'^{\Phi_\Gamma}} (f\ id_{\Gamma'})^{\Phi_{\Gamma,\Gamma'}}$$

at type $\Pi_{\Gamma'^{\Phi_\Gamma}} A^{\Phi_{\Gamma,\Gamma'}}$.

Given f, we know nothing about the type of f^Φ. In particular, f^Φ does not have to be a function. But by applying f to arbitrary arguments $id_{\Gamma'}$, then applying Φ, and then λ-abstracting over the arbitrary arguments, we can obtain a functional behavior for f^Φ after all.

The construction of f^* is enough to map Σ-functions to Σ'-functions. But this is only useful if there is a meaningful relation between $(f^\Phi)^*$ and f^Φ. This relation indeed exists: Via the $*$-operator, lax morphisms commute with function application:

THEOREM 4.24. *Whenever* $\Gamma \vdash_\Sigma f\,\gamma : A$, *then*

$$\Gamma^\Phi \vdash_{\Sigma'} (f\,\gamma)^{\Phi_\Gamma} = (f^{\Phi_\Gamma})^*\,\gamma^{\Phi_\Gamma}$$

PROOF. Let $\Gamma \vdash_\Sigma f : \Pi_{\Gamma'} A'$. Then the result follows by applying Φ to $\Gamma \vdash_\Sigma f\,\gamma = (f\ id_{\Gamma'})[\gamma]$ and then using $\Gamma^\Phi \vdash_\Sigma (f\ id_{\Gamma'})^{\Phi_{\Gamma,\Gamma'}}[\gamma^{\Phi_\Gamma}] = (f^{\Phi_\Gamma})^*\,\gamma^{\Phi_\Gamma}$. □

Example 4.25. For the lax morphisms from Sect. 4.2, given an appropriate $f : (\Pi_{\Gamma'} A)^{\Phi_\Gamma}$ where $x_1, \ldots, x_n$ are the variables in Γ', the function f^* looks as follows.

— If Φ is a strict morphism, then $f^* = f$.
— If $\Phi = \Phi^1 \times \ldots \times \Phi^m$, then

$$f^{*\Phi} = \lambda_{\Gamma'^{\Phi_\Gamma}} \langle \ldots, f.i^{*\Phi_i}\,(x_1.i, \ldots, x_n.i), \ldots \rangle$$

In particular, in the special case where $\Gamma = \cdot$, $\Gamma' = x : A$, and $m = 2$, this yields

$$f^{*\Phi} = \lambda_{x:A^\Phi} \langle f.1^*\,x.1,\ f.2^*\,x.2 \rangle$$

— If Φ is a power of the form Ψ^U, then

$$f^{*\Phi} = \lambda_{\Gamma'^{\Phi_\Gamma}} \lambda_U \big((f\ id_U)^{*\Psi}\,(x_1\ id_U), \ldots, (x_n\ id_U)\big)$$

In particular, in the special case where $\Gamma = \cdot$, $\Gamma' = x : A$, and $U = p : P$, this yields

$$f^{*\Phi} = \lambda_{x:A^\Phi} \lambda_{p:P} \big((f\,p)^*\,(x\,p)\big)$$

Example 4.26 (*Product of Strict Morphisms*). Let us further specialize the case of products in Ex. 4.25 to the situation where Φ^1 and Φ^2 are strict. Let us abbreviate E^{Φ^i} as E^i for any expression E. Because of strictness, we have $(A \to B)^i = A^i \to B^i$. Then for a Σ'-term

$$f \quad : \quad (A \to B)^\Phi \quad = \quad (A^1 \to B^1) \times (A^2 \to B^2)$$

we have

$$f^* \quad = \quad \lambda_{x:A^1 \times A^2} \langle f.1\,x.1,\ f.2\,x.2 \rangle \quad : \quad A^\Phi \to B^\Phi \quad = \quad (A^1 \times A^2) \to (B^1 \times B^2)$$

Thus, the construction $f \mapsto f^*$ specializes to the well-known transformation of a pair of functions into a function on pairs.

5. LOGICAL RELATIONS ON LAX MORPHISMS

In [Rabe and Sojakova 2013], we introduced the notion of a logical relation on strict morphisms. We can now generalize this definition to lax morphisms.

5.1. Formal Definition

Consider a lax morphism $\Phi : \Sigma \to \Sigma'$ and fix an arbitrary Σ-context Γ. Then Φ provides for every Σ-type A:
— a target type A^{Φ_Γ},
— a map that maps every Σ-term t of type A to a Σ'-term t^{Φ_Γ} of type A^{Φ_Γ}.
Similarly, an invariant ρ about Φ should provide for every Σ-type A:
— a unary predicate A^{ρ_Γ} on the type A^{Φ_Γ},
— a proof for every Σ-term t of type A that the term t^{Φ_Γ} satisfies the predicate A^{ρ_Γ}.
Logical relations ρ are the result of formalizing this intuition.

Because we use dependent type theory, we formalize a unary predicate on a type A as a dependent type $x : A \vdash B[x]$. Then $t : A$ satisfies predicate B if the type $B[t]$ is non-empty. Thus, we obtain the following definition:

Definition 5.1. Given a lax morphism $\Phi : \Sigma \to \Sigma'$. A **lax logical relation** ρ on Φ is a functor $\Phi : \mathtt{Cont}(\Sigma) \to \mathtt{Cont}(\Sigma')$, denoted $-^\rho$, that satisfies

$$\begin{aligned} .^\rho &= . \\ (\Gamma,\ x : A)^\rho &= \Gamma^\rho,\ x : A^{\Phi_\Gamma},\ x^! : A^{\rho_\Gamma}[x] \end{aligned}$$

$$\begin{aligned} .^\rho &= . \\ (\gamma,\ t)^\rho &= \gamma^\rho,\ t^{\Phi_\Gamma},\ t^{\rho_\Gamma} \qquad \text{if } \vdash_\Sigma \gamma : \Gamma' \to \Gamma \end{aligned}$$

for arbitrary types A^{ρ_Γ} and terms t^{ρ_Γ}.

Notation 5.2. Γ^ρ contains two variable declarations for every variable x in Γ. We reuse the name x for the first one and assume that we can always form the fresh name of the second one as $x^!$.

Remark 5.3. A^{ρ_Γ} is a unary predicate on A^{Φ_Γ}, i.e., a type in context $\Gamma^\rho, x : A^{\Phi_\Gamma}$ for some fresh variable x. The presentation becomes simpler if we do not fix the name of the variable x. Therefore, we never refer to A^{ρ_Γ} directly. Instead, we always use a term $A^{\rho_\Gamma}[t]$, which substitutes some term t for x.

Like lax morphisms, lax logical relations uniquely determine the types/terms A^{ρ_Γ} and t^{ρ_Γ}, and vice versa every logical relation is uniquely determined by these terms. In the same way as for lax morphisms, we obtain the following equivalent characterization; in particular, Def. 5.1 and Thm. 5.4 correspond to Def. 4.4 and Thm. 4.5:

Theorem 5.4 (Characterization of Logical Relations). *Assume a lax morphism $\Phi : \Sigma \to \Sigma'$ and a family of maps ρ_Γ that map Σ-expressions to Σ'-expressions. Then the following are equivalent:*

(1) ρ (seen as a functor) is a lax logical relation on Φ.
(2) ρ preserves variables, judgments, and substitution in the sense that the following rules are admissible:

$$\frac{\vdash_\Sigma \Gamma, x : A\ \mathtt{Ctx}}{(\Gamma, x : A)^\rho \vdash_{\Sigma'} x^{\rho_{\Gamma, x:A}} = x^!}$$

$$\frac{\Gamma \vdash_\Sigma t : A}{\Gamma^\rho \vdash_{\Sigma'} t^{\rho_\Gamma} : A^{\rho_\Gamma}[t^{\Phi_\Gamma}]} \qquad \frac{\Gamma \vdash_\Sigma A : \mathtt{type}}{\Gamma^\rho,\ x : A^{\Phi_\Gamma} \vdash_{\Sigma'} A^{\rho_\Gamma}[x] : \mathtt{type}}$$

$$\frac{\Gamma \vdash_\Sigma t = t'}{\Gamma^\rho \vdash_{\Sigma'} t^{\rho_\Gamma} = t'^{\rho_\Gamma}} \qquad \frac{\Gamma \vdash_\Sigma A = A'}{\Gamma^\rho,\ x : A^{\Phi_\Gamma} \vdash_{\Sigma'} A^{\rho_\Gamma}[x] = A'^{\rho_\Gamma}[x]}$$

$$\frac{\vdash_\Sigma \gamma : \Gamma \to \Gamma' \qquad \Gamma \vdash_\Sigma t : A}{\Gamma'^{\rho} \vdash_{\Sigma'} t^{\rho_\Gamma}[\gamma^\rho] = t[\gamma]^{\rho_{\Gamma'}}} \qquad \frac{\vdash_\Sigma \gamma : \Gamma \to \Gamma' \qquad \Gamma \vdash_\Sigma A : \mathtt{type}}{\Gamma'^{\rho},\ x : A^{\Phi_\Gamma}[\gamma^\rho] \vdash_{\Sigma'} A^{\rho_\Gamma}[\gamma^\rho, x] = A[\gamma]^{\rho_{\Gamma'}}[x]}$$

Proof. These conditions are essentially the same as in Thm. 4.5 except that terms and types are transformed slightly differently.

The proof proceeds in the same way as for Thm. 4.5. In particular, the well-definedness, the identity law, and the composition law of the functor are equivalent to the preservation of judgments, variables, and substitution, respectively. □

Remark 5.5 (*n-ary Relations*). The definition given here is that of a unary logical relation. Corresponding to [Rabe and Sojakova 2013], our definition extends to n-ary logical relations between n lax morphisms. However, because lax morphisms are closed under products in the presence of product types, we can restrict attention to the unary case without loss of generality: n-ary logical relations can be recovered as unary ones on the product of n morphisms. This was not possible in [Rabe and Sojakova 2013] because strict morphisms are not closed under product formation.

5.2. Strict Logical Relations

Just like lax and strict morphisms, we obtain a distinction between lax and strict relations. Strict relations are the special case in Def. 5.1 where a lax relation is determined already by its action on the constants and extended inductively to all expressions.

For identifiers, we can define strict relations in a way that is very similar to strict morphisms:

Definition 5.6 (*Base Cases*). A strict logical relation on $\Phi : \Sigma \to \Sigma'$ is given as a list $\rho = \ldots, c \mapsto E, \ldots$ containing one Σ'-expression E for every Σ-constant c.

For typed constants $c : A : \mathtt{type}$ in Σ and $c \mapsto E$ in ρ, we require that

$$\vdash_{\Sigma'} E : A^{\rho}[A^{\Phi}]$$

and we put

$$c^{\rho_\Gamma} := E$$

Moreover, for variables x in Γ, we put

$$x^{\rho_\Gamma} := x^!$$

However, contrary to strict morphisms, there is no generic way of defining logical relations inductively for arbitrary type theoretical features. Instead, we have to find the induction cases separately for each feature. Correspondingly, the inductive proof that strict relations are indeed lax relations has to be carried out separately for each feature.

For product and function types, the inductive definitions of strict logical relations are well-known. In the remainder, we formally define them in our setting and generalize them to lax morphisms.

Strict Relations for Kinded Constants. For the kinded constants of Sect. 3.4, we define the condition on ρ and the base case as follows:

Definition 5.7. For $c : \Pi_\Delta \mathtt{type}$ in Σ and $c \mapsto E$ in ρ, we require that

$$\vdash_{\Sigma'} E : \Pi_{\Delta^\rho}(c\, id_{\Delta^\rho}) \to \mathtt{type}$$

and for a type $c\,\delta$ and a variable $x : (c\,\delta)^{\Phi_\Gamma}$, we put

$$(c\,\delta)^{\rho_\Gamma}[x] := E\,(\delta^{\rho_\Gamma}, x)$$

Strict Relations at Function Types. Intuitively, a function satisfies a logical relation if all its applications do, i.e., if it maps arguments that satisfy the relation to results that satisfy the relation. Formally, we obtain the following cases for the inductive definition:

Definition 5.8. In a type theory with dependent function types, the cases for strict relations are

$$(\Pi_\Delta A)^{\rho_\Gamma}[x] = \Pi_{\Delta^{\rho_\Gamma}}\left(A^{\rho_{\Gamma,\Delta}}[(x\ id_\Delta)^{\Phi_{\Gamma,x:\Pi_\Delta A,\Delta}}]\right)$$

$$(\lambda_\Delta t)^{\rho_\Gamma} = \lambda_{\Delta^{\rho_\Gamma}} t^{\rho_{\Gamma,\Delta}}$$

$$(t\,\delta)^{\rho_\Gamma} = t^{\rho_\Gamma}\ \delta^{\rho_\Gamma}$$

Def. 5.8 is structurally very similar to Def. 5.10. The cases for terms are homomorphic, and the case for types is almost homomorphic. We understand it best by specializing to the case where Γ is empty. Then

$$f : (\Pi_\Delta A)^\Phi \vdash_{\Sigma'} (\Pi_\Delta A)^\rho[f] = \Pi_{\Delta^\rho}\left(A^{\rho_\Delta}[(f\ id_\Delta)^{\Phi_{f:\Pi_\Delta A,\Delta}}]\right)$$

If we further specialize to the case where Φ is a strict morphism, we obtain

$$f : (\Pi_\Delta A)^\Phi \vdash_{\Sigma'} (\Pi_\Delta A)^\rho[f] = \Pi_{\Delta^\rho}\left(A^{\rho_\Delta}[f\ id_\Delta]\right)$$

If we specialize even further to the unary case where $\Delta = y : B$, we obtain

$$f : \Pi_{y:B^\Phi} A^{\Phi_{y:B}} \vdash_{\Sigma'} (\Pi_{y:B} A)^\rho[f] = \Pi_{y:B^\Phi,\, y^!:B^\rho}\left(A^{\rho_{y:B}}[f\ y]\right)$$

This is the familiar definition of logical relations at function types: A function f is in the relation at $\Pi_{y:B}A$ if it maps arguments y that are in the relation at B (which is assumed by $y^!$) to results $f\ y$ that are in the relation at A.

THEOREM 5.9. *In a type theory with dependent function types, strict relations are indeed lax relations.*

PROOF. The proof proceeds in the same way as for Thm. 5.11. □

Strict Relations at Product Types. Intuitively, a tuple satisfies a strict relation iff all its projections do. Formally, we obtain the following cases for the inductive definition:

Definition 5.10. In a type theory with dependent product types, the cases for strict relations are

$$\langle\Delta_n\rangle^{\rho_\Gamma}[x] = \langle\Theta_n\rangle$$

$$\langle\ldots, t_i, \ldots\rangle^{\rho_\Gamma} = \langle\ldots, t_i^{\rho_\Gamma}, \ldots\rangle$$

$$(t.i)^{\rho_\Gamma} = t^{\rho_\Gamma}.i$$

where we abbreviate

$$\begin{array}{l}\Delta_i = \ldots, x_i : A_i \\ \Theta_i = \ldots, x_i^! : A_i^{\rho_{\Gamma,\Delta_{i-1}}}[\delta_{i-1}, (x.i)^{\Phi_{\Gamma,x:\langle\Delta_n\rangle}}] \\ \delta_i\ \ = \ldots, (x.i)^{\Phi_{\Gamma,x:\langle\Delta_n\rangle}}, x_i^!\end{array}$$

satisfying

$$\Gamma^\rho \vdash_{\Sigma'} \delta_i : \Delta_i^{\rho_\Gamma} \to x : \langle\Delta_n\rangle^{\Phi_\Gamma}, \Theta_{i-1}$$

for $i = 1, \ldots, n$.

These definitions are more straightforward than they look. For terms, the definition is homomorphic. For types, we understand it best by specializing to the case where Γ is empty and where the product is simply-typed. Then

$$x : A^{\Phi} \vdash_{\Sigma'} A^{\rho}[x] = \ldots \times A_i^{\rho}[(x.i)^{\Phi_{x:A}}] \times \ldots \qquad \text{where} \qquad A = \ldots \times A_i \times \ldots$$

If we specialize further to the case where Φ is a strict morphism, then $(x.i)^{\Phi_{\Gamma,x:\langle\Delta_n\rangle}} = x.i$ and we obtain

$$x : A^{\Phi} \vdash_{\Sigma'} A^{\rho}[x] = \ldots \times A_i^{\rho}[x.i] \times \ldots \qquad \text{where} \qquad A = \ldots \times A_i \times \ldots$$

This is the familiar form of a logical relation at product types: A tuple x is in the relation at $\ldots \times A_i \times \ldots$ if each component $x.i$ is in the relation at A_i.

The rest of the definition is just bureaucracy to keep track of dependent typing in $\Delta = x_1 : A_1, \ldots, x_n : A_n$. In particular, the ghastly-looking substitutions δ_i become intuitively clear if we only look at the variable names. The variable names in Δ_i are $x_1, \ldots, x_i$, and the ones in $\Delta_i^{\rho_\Gamma}$ are $x_1, x_1^!, \ldots, x_i, x_i^!$. Thus, the variables names in Θ_i are $x_1^!, \ldots, x_i^!$. If Φ is strict, δ_i maps every x_i to $x.i$ and every $x_i^!$ to itself. Then Θ_n is simply $\Delta_n^{\rho_\Gamma}$ with all the declarations of any x_i dropped and all the occurrences of any x_i replaced with $x.i$.

THEOREM 5.11. *In a type theory with dependent product types, strict relations are indeed lax relations.*

PROOF. The preservation of variables holds generically.

The preservation of typing and substitution are shown by induction on the syntax. The cases for substitution follow from the induction hypothesis by inspecting the cases in Def. 5.10. The cases for typing follow by inferring the types of the expressions on the right-hand sides of Def. 5.10 and comparing them to the types expected for a lax relation. The latter comparison is tedious but straightforward; it already uses the preservation of substitution. □

Combining Features. Def. 5.10 and 5.8 provide exactly the cases needed to define strict relations inductively for one type theoretical feature each. Similarly, Thm. 5.11 and 5.9 provide the cases to prove the necessary properties inductively for each feature. These definitions and theorems combine so that we obtain the corresponding results for a type theory with both dependent product and function types.

If additional features are introduced, we can handle and combine them accordingly.

Remark 5.12. Note the similarity between the occurrences of $(x.i)^{\Phi_{\Gamma,x:\langle\Delta_n\rangle}}$ in Def. 5.10 and $(x\, id_\Delta)^{\Phi_{\Gamma,x:\Pi_\Delta A,\Delta}}$ in Def. 5.8 – in both cases, Φ is applied to the elimination form of the variable for which the relation is stated.

Intuitively, a term at a complex type satisfies the logical relation if all its elimination forms do. This hints at the possibility of a generic definition for arbitrary type theoretical features, of which Def. 5.10 and 5.8 would be special cases. However, we have so far not been able to give one.

Example 5.13 (*Running Example: Type Preservation as a Strict Relation*). We continue Ex. 3.3 by giving a strict logical relation `TP` (type preservation) on the strict theory morphism `ER`. `TP` contains the following declarations:

$$\begin{array}{lcl}
\texttt{sort} & \mapsto & \lambda_{s:\texttt{i}\to\texttt{o}}\, \langle\cdot\rangle \\
\texttt{tm} & \mapsto & \lambda_{s:\texttt{i}\to\texttt{o}}\, \lambda_{s^!:\langle\cdot\rangle}\, \lambda_{x:\texttt{i}}\, \texttt{ded}\,(s\,x) \\
\texttt{o} & \mapsto & \lambda_{x:\texttt{o}}\, \langle\cdot\rangle \\
\texttt{ded} & \mapsto & \lambda_{F:\texttt{o}}\, \lambda_{F^!:\langle\cdot\rangle}\, \lambda_{p:ded\,F}\, \langle\cdot\rangle \\
\texttt{imp} & \mapsto & \lambda_{F:\texttt{o}}\, \lambda_{F^!:\langle\cdot\rangle}\, \lambda_{G:\texttt{o}}\, \lambda_{G^!:\langle\cdot\rangle}\, \langle\cdot\rangle \\
\texttt{forall} & \mapsto & \lambda_{s:\texttt{i}\to\texttt{o}}\, \lambda_{s^!:\langle\cdot\rangle}\, \lambda_{F:\texttt{i}\to\texttt{o}}\, \lambda_{F^!:\Pi_{x:\texttt{i}}\, \Pi_{x^!:\texttt{ded}\,(s\,x)}\, \langle\cdot\rangle}\, \langle\cdot\rangle
\end{array}$$

$\mathtt{TP}$ maps $\mathtt{sort}$, $\mathtt{o}$, and $\mathtt{ded}\,F$ for any F to the unit type. $\mathtt{tm}$ is the only type-valued constant that is mapped non-trivially. Thus, $\mathtt{TP}$ establishes a non-trivial property only for terms $t : \mathtt{tm}\,s$.

This property is that every term $\vdash_{\mathtt{SFOL}} t : \mathtt{tm}\,s$ is translated to a proof $\vdash_{\mathtt{FOL}} t^{\mathtt{TP}} : \mathtt{ded}\,(s^{\mathtt{ER}}\,t^{\mathtt{ER}})$, i.e., that $\mathtt{TP}$ maps $\mathtt{SFOL}$-terms of sort s to $\mathtt{FOL}$-terms for which $s^{\mathtt{ER}}$ holds. That is exactly the type preservation property we need.

The cases for the term-valued constants (here: $\mathtt{imp}$ and $\mathtt{forall}$) correspond to proofs that $\mathtt{ER}$ is closed under every constant. In our case, this happens to be trivial because we omitted all constructors from $\mathtt{SFOL}$ whose return type is $\mathtt{tm}\,s$. If we add such constructors, we have to add the corresponding non-trivial case to $\mathtt{TP}$.

Nonetheless, we can see the characteristic shape of these cases even if they are trivial. The case for a constant c takes one pair of arguments x_i and $x_i^!$ for every argument x_i that c takes, where each $x_i^!$ is the assumption that x_i is in the relation. It returns a proof that $c\,(x_1, \ldots, x_n)$ is in the relation.

For example, $\mathtt{forall}$ of $\mathtt{SFOL}$ takes two arguments: a sort s and a unary predicate F on $\mathtt{tm}\,s$. Consequently, its case in $\mathtt{TP}$ takes the following 4 arguments
— a translation $s : \mathtt{i} \to \mathtt{o}$ of a sort,
— an assumption $s^!$ that s in the relation (which is trivial in our example),
— a translation $F : \mathtt{i} \to \mathtt{o}$ of a unary predicate,
— an assumption $F^!$ that F is in the relation, which means for all $x : \mathtt{i}$ that are in the relation (as assumed by $x^!$), the result $F\,x$ is in the relation (which is expressed by the type $(\lambda_{x:\mathtt{o}}\,\langle\cdot\rangle)\,(F\,x)$, which β-reduces to $\langle\cdot\rangle$).
and returns a proof that $\mathtt{forall}\,(\lambda_{x:\mathtt{i}}\,(s\,x)\,\mathtt{imp}\,(F\,x))$ is in the relation (which β-reduces to the trivial $\langle\cdot\rangle$ again).

5.3. Concrete Logical Relations

We describe some general constructions that yield lax logical relations. They will not be used later on but provide valuable examples and closure properties.

THEOREM 5.14 (MORPHISMS AS RELATIONS). *Given a lax morphism Φ, we obtain a lax relation $\tilde{\Phi}$ on Φ by putting*

$$A^{\tilde{\Phi}_\Gamma}[x] = A^{\Phi_\Gamma}$$

$$t^{\tilde{\Phi}_\Gamma} = t^{\Phi_\Gamma}$$

PROOF. The properties of $\tilde{\Phi}$ follow immediately from the ones of Φ. □

In these lax relations, the predicate $A^{\tilde{\Phi}_\Gamma}[x]$ never depends on x and is always satisfied if A is non-empty. By using the embedding $\Phi \mapsto \tilde{\Phi}$, we can occasionally treat lax morphisms as special cases of lax relations and use that to state results uniformly.

In the presence of product types, we obtain an intersection of lax relations in essentially the same was as the product of lax morphisms:

Definition 5.15 (*Intersection*). Assume a type theory with product types. Consider lax relations ρ^i on $\Phi : \Sigma \to \Sigma'$ for $i = 1, \ldots, n$. By mutual induction, we define a lax relation ρ on Φ and substitutions $\varphi^i_\Gamma : \Gamma^{\rho^i} \to \Gamma^\rho$:

$$A^{\rho_\Gamma}[x] = \ldots \times A^{\rho^i_\Gamma}[\varphi^i_\Gamma, x] \times \ldots$$

$$t^{\rho_\Gamma} = \langle \ldots, t^{\rho^i_\Gamma}[\varphi^i_\Gamma], \ldots \rangle$$

$$\varphi^i_\cdot = \cdot \qquad \text{and} \qquad \varphi^i_{\Gamma, x:A} = \varphi^i_\Gamma, x, x^*.i.$$

We denote this lax relation by $\rho^1 \wedge \ldots \wedge \rho^n$.

If we think of lax relations ρ on Φ as judgments on Φ, then $\rho^1 \wedge \rho^2$ is the conjunction (intersection) of the judgments ρ_1 and ρ_2: It holds if each ρ^i holds. As the special case $n = 0$, this yields a total lax relation $\top_\Phi$, which maps every type to the unit type, i.e., the relation that holds everywhere.

Just like in Def. 4.14, the substitutions φ^i are just bookkeeping. If Γ is empty, we obtain the very simple

$$A^\rho[x] = \ldots \times A^{\rho^i}[x] \times \ldots$$

We can also define a product, where every factor is a lax relation on a different lax morphism:

Definition 5.16 (*Product*). Assume a type theory with product types. Consider lax relations ρ^i on $\Phi^i : \Sigma \to \Sigma'$ for $i = 1, \ldots, n$. By mutual induction, we define a lax relation ρ on $\Phi^1 \times \ldots \times \Phi^n$ and substitutions $\varphi^i_\Gamma : \Gamma^{\rho^i} \to \Gamma^\rho$:

$$A^{\rho_\Gamma}[x] = \ldots \times A^{\rho^i_\Gamma}[\varphi^i_\Gamma, x.i] \times \ldots$$

$$t^{\rho_\Gamma} = \langle \ldots, t^{\rho^i_\Gamma}[\varphi^i_\Gamma], \ldots \rangle$$

$$\varphi^i_\cdot = \cdot \qquad \text{and} \qquad \varphi^i_{\Gamma,x:A} = \varphi^i_\Gamma, x.i, x^*.i.$$

We denote this lax relation by $\rho^1 \times \ldots \times \rho^n$.

This amounts to taking a product in the category of functors $\mathtt{Cont}(\Sigma) \to \mathtt{Cont}(\Sigma')$. The special case $n = 0$, yields a terminal element. If we think of lax relations ρ on Φ as elements of Φ, then $\rho_1 \times \rho_2$ on $\Phi_1 \times \Phi_2$ corresponds to the cartesian product.

Theorem 5.17. *In the situation of Def. 5.15 and Def. 5.16, the defined objects are indeed lax morphisms.*

Moreover, we have the isomorphism

$$(\rho_1 \times \rho_2) \wedge (\rho_1' \times \rho_2') \cong (\rho_1 \wedge \rho_1') \times (\rho_2 \wedge \rho_2').$$

Proof. The proofs are tedious but straightforward. □

Composition of Binary Logical Relations. We expect the constructions given for lax logical relations in [Plotkin et al. 2000] to carry over to our setting. These include, e.g., the diagonal and dual relation and the union of relations.

The most important construction is the composition of binary lax relations (i.e., lax relations on lax morphisms $\Phi \times \Phi'$). If ρ_i is a lax relation on $\Phi_i \times \Phi_{i+1}$ for $i = 1, 2$, then $\rho_2 \circ \rho_1$ is a lax relation on $\Phi_1 \times \Phi_3$ defined by

$$A^{(\rho_2 \circ \rho_1)_\Gamma}[\langle x, z \rangle] = \langle y : A^{\Phi_{2\Gamma}}, \_ : A^{\rho_{1\Gamma}}[\langle x, y \rangle] \times A^{\rho_{2\Gamma}}[\langle y, z \rangle] \rangle$$

$$t^{\rho_2 \circ \rho_{1\Gamma}} = \langle t^{\Phi_{2\Gamma}}, \langle t^{\rho_{1\Gamma}}, t^{\rho_{2\Gamma}} \rangle \rangle$$

Here, intuitively, y is the witness in between x and z, and underscore is the pair of proofs showing $x \rho_1 y$ and $y \rho_2 z$.

5.4. Lax Morphisms Induced by Logical Relations

Given a lax morphism Φ and an invariant ρ on Φ, we would like to pair them up into a single object. In the presence of dependent products, this is possible and yields another lax morphism:

Definition 5.18. Consider a type theory with dependent products, a lax morphism Φ, and a logical relation ρ on it. By mutual induction we define a lax morphism Ψ and two families of substitutions $\varphi_\Gamma : \Gamma^\Phi \to \Gamma^\Psi$ and $\varphi^!_\Gamma : \Gamma^\rho \to \Gamma^\Psi$:

$$A^{\Psi_\Gamma} = \langle \_ : A^{\Phi_\Gamma}[\varphi_\Gamma],\ ! : A^{\rho_\Gamma}[\varphi^!_\Gamma,\ \_]\rangle$$

$$t^{\Psi_\Gamma} = \langle t^{\Phi_\Gamma}[\varphi_\Gamma], t^{\rho_\Gamma}[\varphi^!_\Gamma]\rangle$$

$$\varphi_{\cdot} = \cdot \qquad \text{and} \qquad \varphi_{\Gamma,x:A} = \varphi_\Gamma,\ x.1$$

$$\varphi^!_{\cdot} = \cdot \qquad \text{and} \qquad \varphi^!_{\Gamma,x:A} = \varphi^!_\Gamma,\ x.1,\ x.2$$

where we use $\_$ and $!$ as convenient variable names.

We denote this lax morphism by $\Phi|\rho$.

We can understand Def. 5.18 by comparing it to Def. 4.14 for the case $n = 2$: Φ and ρ correspond to Φ^1 and Φ^2, and φ and $\varphi^!$ correspond to φ^1 and φ^2. The remaining complexity of Def. 5.18 over Def. 4.14 corresponds to the complexity of a dependent product type relative to a simple product type.

Another way to understand Def. 5.18 is that $A^{(\Phi|\rho)_\Gamma}$ is the subtype of A^{Φ_Γ} containing exactly those x for which the relation ρ holds, i.e., for which the type $A^{\rho_\Gamma}[x]$ is inhabited.

More precisely, if we have a type A and a unary predicate B on A, we can regard the type $\langle \_ : A, ! : B[\_]\rangle$ as the subtype of A containing exactly the $x : A$ for which $B[x]$ holds. This works particularly well if we use a propositions-as-types representation with proof irrelevance, i.e., if the type $B[x]$ represents a proposition inhabited by proofs and if proofs of the same proposition are considered equal. In that case, we could use the suggestive notation $\{x : A \,|\, B[x]\}$ for $\langle x : A, ! : B[x]\rangle$.

Applying this notation in our setting, we obtain $A^{(\Phi|\rho)_\Gamma} = \{x : A^{\Phi_\Gamma} \,|\, A^{\rho_\Gamma}[x]\}$, which inspired the notation $\Phi|\rho$. Thus, the lax morphism $\Phi|\rho$ interprets every type A as the subtype of A^Φ defined by A^ρ. The properties of lax relations ensure that all these subtypes are closed under all operations that are definable in Σ.

Example 5.19 (*Running Example: Type Preservation as a Lax Morphism*). We can now form the lax morphism $\mathtt{ER}|\mathtt{TP}$, which combines $\mathtt{ER}$ from Ex. 3.3 and $\mathtt{TP}$ from Ex. 5.13. It captures both the type erasure translation $\mathtt{ER}$ and the type preservation invariant $\mathtt{TP}$.

$\mathtt{ER}|\mathtt{TP}$ maps every term

$$\vdash_{\mathtt{SFOL}} u : \mathtt{tm}\, s$$

to the pair

$$\vdash_{\mathtt{FOL}} \langle u^{\mathtt{ER}}, u^{\mathtt{TP}}\rangle : \langle \_ : \mathtt{i},\ ! : \mathtt{ded}\,(s^{\mathtt{ER}}\ \_)\rangle$$

where $u^{\mathtt{ER}}$ is the type erasure translation and $u^{\mathtt{TP}}$ the type preservation proof for $u^{\mathtt{ER}}$.

More generally, it maps every function

$$\vdash_{\mathtt{SFOL}} f : \mathtt{tm}\, s \to \mathtt{tm}\, t$$

to the pair

$$\vdash_{\mathtt{FOL}} \langle f^{\mathtt{ER}}, f^{\mathtt{TP}}\rangle : \langle \_ : \mathtt{i} \to \mathtt{i},\ ! : \Pi_{x:\mathtt{i}}\, \mathtt{ded}\,(s^{\mathtt{ER}}\, x) \to \mathtt{ded}\,(t^{\mathtt{ER}}\,(\_\, x))\rangle$$

Here $f^{\mathtt{ER}}$ of type $i \to i$ is the function that results from applying type erasure to the function f. The type of $f^{\mathtt{TP}}$ expresses the type preservation property of $f^{\mathtt{ER}}$: It maps any x that satisfies the predicate $s^{\mathtt{ER}}$ to something that satisfies the predicate $t^{\mathtt{ER}}$. Thus, $f^{\mathtt{TP}}$ is the proof that $f^{\mathtt{ER}}$ preserves typing.

THEOREM 5.20. *In the situation of Def. 5.18, $\Phi|\rho$ is a lax morphism.*

PROOF. The argument proceeds in the same way as for Thm. 4.15. □

6. VERIFYING THE TYPE-ERASURE TRANSLATION

Finally, we can conclude our running example and verify the type erasure translation:

Example 6.1 (*Running Example: Truth Preservation as a Strict Extension of a Lax morphism*). Consider the lax morphism $\mathtt{ER}|\mathtt{TP} : \mathtt{SFOL} \to \mathtt{FOL}$ from Ex. 5.19. Because FOLPF extends FOL, we also have $\mathtt{ER}|\mathtt{TP} : \mathtt{SFOL} \to \mathtt{FOLPF}$. SFOLPF extends SFOL only with typed declarations so that we can apply Def. 4.11 to form a lax morphism $(\mathtt{ER}|\mathtt{TP}), \mathtt{PP} : \mathtt{SFOLPF} \to \mathtt{FOLPF}$.

PP contains the following declarations:

$$
\begin{array}{lcl}
\mathtt{imp}_I & \mapsto & \lambda F : \langle \_ : \mathtt{o}, \, ! : \langle\cdot\rangle\rangle \\
 & & \lambda G : \langle \_ : \mathtt{o}, \, ! : \langle\cdot\rangle\rangle \\
 & & \lambda p : \langle \_ : \mathtt{ded}\, F \to \mathtt{ded}\, G, \, ! : \Pi_{x:\mathtt{o}} \langle\cdot\rangle \to \langle\cdot\rangle\rangle \\
 & & \quad \langle \mathtt{imp}_I \, F.1 \, G.1 \, p.1, \, \langle\cdot\rangle\rangle \\
\mathtt{imp}_E & \mapsto & \lambda F : \langle \_ : \mathtt{o}, \, ! : \langle\cdot\rangle\rangle \\
 & & \lambda G : \langle \_ : \mathtt{o}, \, ! : \langle\cdot\rangle\rangle \\
 & & \lambda p : \langle \_ : \mathtt{ded}\, (F \, \mathtt{imp} \, G), \, ! : \langle\cdot\rangle\rangle \\
 & & \lambda q : \langle \_ : \mathtt{ded}\, F, \, ! : \langle\cdot\rangle\rangle \\
 & & \quad \langle \mathtt{imp}_E \, F.1 \, G.1 \, p.1 \, q.1, \langle\cdot\rangle\rangle \\
\mathtt{forall}_I & \mapsto & \lambda s : \langle \_ : \mathtt{i} \to \mathtt{o}, \, ! : - \rangle \\
 & & \lambda F : \langle \_ : \mathtt{i} \to \mathtt{o}, \, ! : - \rangle \\
 & & \lambda p : \Pi_{x:\langle \_ : \mathtt{i}, \, ! : \mathtt{ded}\, (s.1 \, \_) \rangle} \langle \_ : \mathtt{ded}\, (F.1 \, x.1), \, ! : - \rangle \\
 & & \quad \langle \mathtt{forall}_I \; - \; \lambda_{x:\langle \_ : \mathtt{i}, \, ! : - \rangle} \; \mathtt{imp}_I \; - \; - \lambda_{q:\langle \_ : \mathtt{ded}\, (s.1 \, x.1), \, ! : - \rangle} \, p.1 \, \langle x.1, \, q.1 \rangle \; , - \rangle \\
\mathtt{forall}_E & \mapsto & \lambda s : \langle \_ : \mathtt{i} \to \mathtt{o}, \, ! : - \rangle \\
 & & \lambda F : \langle \_ : \mathtt{i} \to \mathtt{o}, \, ! : - \rangle \\
 & & \lambda p : \langle \_ : \mathtt{ded}\, (\mathtt{forall} \lambda_{x:\mathtt{i}} \, (s.1 \, x.1) \, \mathtt{imp} \, (F.1 \, x.1)), \, ! : - \rangle \\
 & & \lambda x : \langle \_ : \mathtt{i}, \, ! : \mathtt{ded}\, (s.1 \, \_) \rangle \\
 & & \quad \langle \mathtt{imp}_E \; - \; - \, (\mathtt{forall}_E \; - \; p.1 \; x.1) \, x.2, \; - \rangle
\end{array}
$$

The cases for implication are straightforward but helpful to illustrate the general shape. Consider $\mathtt{imp}_E$, which takes 4 arguments in SFOLPF – 2 propositions F and G and two proofs of $F \, \mathtt{imp} \, G$ and F – and returns a proof of G. The case for $\mathtt{imp}_E$ in PP takes the corresponding 4 arguments translated along TP, but each of them is now paired up with a proof of the invariant given by ER. Because ER does not make any claims about propositions and proofs, all 4 invariants are trivial, i.e., just the unit type. Accordingly, the case for $\mathtt{imp}_E$ returns a proof of G, which we obtain easily from $\mathtt{imp}_E$ of FOLPF and the first components of all arguments; it also returns a proof of the invariant, which is again trivial.

The case for $\mathtt{imp}_I$ proceeds in the same way as the one for $\mathtt{imp}_E$. Note how applying the logical relation TP to the function type $\mathtt{ded}\, F \to \mathtt{ded}\, G$ leads to the type $\Pi_{x:\mathtt{o}} \langle\cdot\rangle \to \langle\cdot\rangle$. This is different from $\langle\cdot\rangle$ but also a singleton type.

For simplicity, in the cases for universal quantification, we have written $-$ for some less relevant subexpressions that are trivial or can be inferred from the context.

Recall that $\mathtt{forall}_E$ in SFOLPF has type

$$\Pi_{s:\mathtt{sort}} \, \Pi_{F:\mathtt{tm}\, s \to \mathtt{o}} \, \mathtt{ded} \, (\mathtt{forall} \, s \, \lambda_{x:\mathtt{tm}\, s} \, (F \, x)) \to \Pi_{x:\mathtt{tm}\, s} \, \mathtt{ded} \, (F \, x)$$

It takes 4 arguments: s, F, an unnamed proof, and x. The case for $\texttt{forall}_E$ in PP takes the corresponding 4 arguments translated along TP paired up with the proof given by ER:
— The argument $s : \texttt{sort}$ gives rise to the argument of type $\texttt{i} \to \texttt{o}$. It is paired with a trivial proof which we do not need.
— The argument F gives rise to an argument of type $\texttt{i} \to \texttt{o}$. It is also paired with a trivial proof.
— The argument $\texttt{ded}\,(\texttt{forall}\,s\,\lambda_{x:\texttt{tm}\,s}\,(F\,x))$ gives rise to the assumption p of the TP-translated formula. It is also paired with a trivial proof.
— The argument $x : \texttt{tm}\,s$ gives rise to an argument $x.1$ of type i. This is now – crucially – paired up with the assumption $x.2$ that $x.1$ satisfies the invariant, i.e., $x.2$ has type $\texttt{ded}\,(s.1\,x.1)$.
Relative to these assumptions, the case for $\texttt{forall}_E$ has to provide a proof of $(\texttt{ded}\,(F\,x))^{\texttt{ER}|\texttt{TP}}$, which is equal to $\langle \_ : \texttt{ded}\,(F.1\,x.1),\ ! : \langle\cdot\rangle\rangle$. This proof is now possible and straightforward due to the presence of the assumption $x.2$.

Thus, we have now succeeded at giving the case for $\texttt{forall}_E$, which failed in Sect. 3.5.

The case for $\texttt{forall}_I$ proceeds accordingly.

Thus, the lax morphism $(\texttt{ER}|\texttt{TP}), \texttt{PP}$ represents and verifies the translation. Moreover, $(\texttt{ER}|\texttt{TP}), \texttt{PP}$ can be checked mechanically. Firstly, it is within a class of lax morphisms that can be represented effectively: ER, TP, and PP are finite lists of declarations. Secondly, it is decidable whether these lists of declarations do indeed induce a lax morphism: All we have to do is check the respective typing judgment for each declaration.

Therefore, we can build a checker that mechanically verifies the translation. For ER and TP, such checkers already exist as described in [Rabe and Schürmann 2009] and [Rabe and Sojakova 2013], respectively. Extending them to the full example is conceptually (albeit not practically) straightforward.

Alternatively, it is possible to check the type erasure translation in a sufficiently strong proof assistant. First, one would define inductive types for the syntax and proof systems of SFOL and FOL. Then the involved lax morphisms would be formalized as recursive functions. The definition of these functions would follow our construction of $(\texttt{ER}|\texttt{TP}), \texttt{PP}$, and their termination would be relatively easy to prove.

The main difference between this approach and our proposed checker is that (i) the former yields an ad hoc formalization that has to be repeated for every variant, whereas (ii) the latter explicates the concepts of theories and lax morphisms, which allows for more systematic and reusable formalizations. For example, in our proposed checker we can
— formalize the corresponding result for the variants of SFOL and FOL with existential quantification and conjunction,
— join the two formalizations using a module system to obtain the result for the full logics,
[Rabe and Sojakova 2013] already shows how theories and logical relations can be defined and checked modularly.

Moreover, in any current proof assistant, it would be extremely awkward to formalize the translation in such a way that the result can be easily instantiated with specific SFOL-theories. This would be possible and elegant if we combine our proposed checker with the results of [Horozal 2014].

7. CONCLUSION AND FUTURE WORK

We have introduced a new concept in the study of logics and related languages: lax theory morphisms, which formalize certain representation theorems that express one theory in another. They are more expressive than conventional theory morphisms but retain strong invariants that guarantee the correctness of the formalized representation theorems.

Categorically, lax morphism are functors that preserve the formation of contexts and substitutions (and thus products). Syntactically, they are maps of types and terms that

preserve variables, judgments, and substitution. Types are mapped arbitrarily, but terms are mapped homomorphically.

Our results are stated very generally and can be transfered to most logics and type theories. However, the distinct ways of mapping types and terms might be less natural in languages that add higher universes.

We exemplified our results by formalizing and verifying the representation of typed in untyped first-order logic as a lax morphism. This example also serves as a template for verifying a number of similar translations that code parts of the domain's type system in terms of the codomain's logic. These include the translation from dependent to simple theory that erases dependencies, and the interpretation of type theory in set theory that maps types to sets. Sect. 7.2 sketches the similar (albeit partial) type erasure translation from higher-order logic to untyped first-order logic.

Our work forms the basis for a number of interconnected advanced results, which we have already partially developed but which can only be finalized in subsequent investigations.[5] We sketch the most impactful among them in the remainder.

7.1. Implementation

A major drawback of lax morphisms and lax relations over their strict counterparts is the lack of an effective representation in computer systems. It is plausible (but not proved) that the set of lax morphisms between certain theories is not countable, let alone recursively enumerable.

However, we can use the constructions presented here to form a grammar that generates a rich class of lax theory morphisms. For example, we can extend our language with the following productions and judgments:

	Grammar	Typing judgment
Lax morphisms	$\Phi ::= \{\sigma\} \mid \top \mid \Phi \times \Phi \mid (\Phi \mid P) \mid \Phi^{\Gamma}$	$\vdash \Phi :^{l} \Sigma \to \Sigma'$
Strict morphisms	$\sigma ::= \cdot \mid \Phi \mid \sigma,\ c \mapsto E$	$\vdash \sigma :^{s} \Sigma \to \Sigma'$
Lax relations	$P ::= \{\rho\} \mid \top \mid P \times P \mid P \wedge P$	$\vdash \rho \leq^{l} \Phi : \Sigma \to \Sigma'$
Strict relations	$\rho ::= \cdot \mid \rho,\ c \mapsto E$	$\vdash \rho \leq^{s} \Phi : \Sigma \to \Sigma'$

This grammar can express all lax morphisms constructed in this paper, in particular our running example. It is straightforward to add typing rules for the new constructors, and all judgments will be decidable (if typing is decidable in the underlying type theory).

We intend to pursue this approach within the Mmt system [Rabe and Kohlhase 2013; Rabe 2013b], which already implements strict theory morphisms for arbitrary declarative languages.

7.2. Partial Lax Morphisms

When motivating lax morphisms, we said that we waived one and retained one of the two properties characterizing strict theory morphisms. However, there is a third implicit property of strict theory morphisms, which we did not mention: the property that they are *total* mappings of expressions.

But many representation theorems require partiality. This often comes up in practice when translating from a more to a less expressive language (which allows using the usually stronger automation support of the latter).

We propose defining partial lax morphisms Φ by relaxing the requirements on lax morphism Φ as follows:

— Preservation of typing: If $t : A$ and Φ is defined for t, then it is also defined for A, and typing is preserved. But Φ may be undefined for both t and A or just for t.

[5] In fact, the ideas sketched in 7.3 and 7.4 predate and motivated the present work.

— Preservation of equality: If $E = E'$ and Φ is defined for both, then equality is preserved. But Φ may be undefined for E or E'.
— Preservation of substitution: If Φ is defined for E and for γ, then it is also defined for $E[\gamma]$, and substitution is preserved. But Φ may be undefined for E or γ even if it is defined for $E[\gamma]$.

For example, we can give a partial strict morphism $\texttt{ER}^{@} : \texttt{HOL} \to \texttt{FOL}^{@}$ that represents higher-order logic in first-order logic but is undefined for all λ-abstractions. Here $\texttt{HOL}$ arises from $\texttt{SFOL}$ by adding $\Rightarrow$: $\texttt{sort} \to \texttt{sort} \to \texttt{sort}$ as well as constants $\texttt{lam}$ and $@$ for λ-abstraction and typed function application. $\texttt{FOL}^{@}$ arises from $\texttt{FOL}$ by adding a binary function symbol $@ : \texttt{i} \to \texttt{i} \to \texttt{i}$ for untyped function application.

The key idea of $\texttt{ER}^{@}$ is to map $(s \Rightarrow t)^{\texttt{ER}^{@}} = \lambda_{f:\texttt{i}}\, \texttt{forall}\, \lambda_{x:\texttt{i}}\, (s^{\texttt{ER}'}\, x)\, \texttt{imp}\, (t^{\texttt{ER}'}\, (f\, x))$, i.e., typed functions are mapped to untyped functions that preserve typing. Then typed function application is mapped to untyped function application, and $\texttt{ER}^{@}$ is undefined for $\texttt{lam}$. We conjecture that type preservation can be proved in essentially the same way as in our running example.

7.3. Representing Models as Lax Morphisms

In [Rabe 2013a; Horozal and Rabe 2011], we showed how to use strict morphisms $\Sigma \to \mathcal{F}$ to represent models of Σ, where the theory $\mathcal{F}$ represents the semantic foundation, in which the models are defined, e.g., axiomatic set theory, higher-order logic, or constructive type theory. This can be seen as a formalist variant of the functorial models [Lawvere 1963] in categorical logic.

But this approach did not carry over to an elegant representation of the *category* of models. If we represent Σ-models as morphisms $\Sigma \to \mathcal{F}$, then we should be able to represent Σ-model morphisms as "morphisms between theory morphisms". The appropriate mathematical structure for this purpose is well-known, namely that of 2-categories. In a 2-category, each set of morphisms $\Sigma \to \Sigma'$ has itself the structure of a category. The morphisms between the morphisms are called 2-cells. However, it has proved difficult to equip the category of theories and *strict* morphisms with an elegant notion of 2-cell. But because lax morphisms are already functors, we immediately obtain a 2-category by using natural transformations as 2-cells.

Now the product of lax morphisms from Def. 4.14 turns out to be indeed a product in the sense of category theory (and the families of substitutions of Def. 4.14 are the natural transformations that are the projections out of the product). This yields a general notion of products of models that subsumes the one known from universal algebra. Similarly, the construction $\Phi|\rho$ corresponds to taking the submodel of Φ by restricting all universes according to ρ.

Notably, all these constructions work in the general case where Σ may contain, e.g., higher-order function symbols. Moreover, we can show that these 2-cells do indeed reduce to the usual model morphisms in the special case where Σ is a theory of first-order logic.

Finally, even the distinction between strict and lax morphisms has a well-known analogue in model theory: The strict morphisms are the standard models, and the lax morphisms correspond to Henkin models. In particular, the embedding of Thm. 4.21 corresponds to the subset relation $[\![A \to B]\!] \subseteq [\![B]\!]^{[\![A]\!]}$ required in Henkin models. This observation provided an unexpected validation of our interest in lax morphisms.

7.4. Record Types and Inductive Types

While the features of dependent functions and products only have to modify the structure of expressions, there is a more complex group of features that require additional declarations. Often these have in common that a single declaration introduces multiple identifiers. 2 examples are of particular importance: inductive type declarations (which introduce a new

type and constructors for it) and record type declarations (which introduce a new type and selectors for it).

We can give an elegant formulation of these two features in terms of theories and morphisms. However, our formulation is itself novel, and we will only sketch the ideas here in simplified form.

Record Types. Record types $\{R\}$ over a theory Σ can be identified with theory extensions R such that Σ, R is a theory. The declarations in R are the fields of the record. Record values $R\{r\} : \{R\}$ can be identified with strict theory morphisms $r : \Sigma, R \to \Sigma$ which agree with id_Σ. (Due to Thm. 4.13, in the typical case where R does not contain type declarations, we can equivalently consider the lax theory morphisms.) The elimination rule defines the selector $u.c$ for every constant c in R, and the β-style rule computes $(R\{r\}).c$ as c^r.

For each theory extension R, we can now obtain a characteristic lax morphism $\mathcal{R}_R : \Sigma, R \to \Sigma$, which maps in particular every field c of R to its selection function

$$c^{\mathcal{R}_R} = \lambda_{u:\{R\}} u.c.$$

Inductive Types. Alternatively, a theory I can be regarded as a family of inductive types. For example, using function types, we can use the theory $\mathtt{Nat} = \mathtt{nat} : \mathtt{type}, \mathtt{z} : \mathtt{nat}, \mathtt{succ} : \mathtt{nat} \to \mathtt{nat}$ for the inductive type of natural numbers. In particular, every typed declaration in I declares a constructor.

Now each I-type A yields an inductive type over Σ, which we write $\ulcorner A \urcorner$, and every closed I-term $t : A$ yields a Σ-term $\ulcorner t \urcorner : \ulcorner A \urcorner$. But it is difficult to design the elimination rule, which has to capture induction, because the inductive functions cannot be Turing-complete and decidable at the same time. As an interesting trade-off, we propose using lax morphisms $\Phi : I \to \Sigma$ in the elimination rule: If $u : \ulcorner A \urcorner$, then $u\,\mathtt{match}\,\Phi : A^\Phi$, and the β-style rule computes $\ulcorner t \urcorner \mathtt{match}\,\Phi$ as t^Φ.

This is expressive enough to define primitive recursive functions such as $+ : \ulcorner \mathtt{nat} \urcorner \to \ulcorner \mathtt{nat} \urcorner \to \ulcorner \mathtt{nat} \urcorner$ as

$$\begin{array}{l}\lambda_{m:\ulcorner \mathtt{nat} \urcorner}\, m\,\mathtt{match}\,(\\ \quad \mathtt{nat} \mapsto \ulcorner \mathtt{nat} \urcorner \to \ulcorner \mathtt{nat} \urcorner,\\ \quad \mathtt{z} \mapsto \lambda_{n:\ulcorner \mathtt{nat} \urcorner}\, n,\\ \quad \mathtt{succ} \mapsto \lambda_{f:\ulcorner \mathtt{nat} \urcorner \to \ulcorner \mathtt{nat} \urcorner}\, \lambda_{n:\ulcorner \mathtt{nat} \urcorner} \ulcorner \mathtt{succ}\,(f\,n) \urcorner\\ )\end{array}$$

But the approach is much more general and allows any kind of declaration in I, even those cases for which a conventional fixed point semantics is not applicable.

Similarly to record types, every I yields a characteristic lax morphism $\mathcal{I}_I : I \to \Sigma$, namely the one that maps every I-type A to $\ulcorner A \urcorner$ and every I-term t to $\ulcorner t \urcorner$. In particular, for every typed constant c of I, the term $(c^{\mathcal{I}_I})^*$ is the corresponding constructor function.

Moreover, we conjecture that $\mathcal{I}_I$ is an initial object in the category of lax morphisms $I \to \Sigma$. Moreover, if I is an algebraic theory, $\mathcal{I}_I$ reduces to the well-known construction of the initial model.

REFERENCES

S. Berardi. 1990. *Type dependence and constructive mathematics*. Ph.D. Dissertation. Dipartimento di Matematica, Università di Torino.

J. Bernardy, P. Jansson, and R. Paterson. 2012. Proofs for Free - Parametricity for Dependent Types. *Journal of Functional Programming* 22, 2 (2012), 107–152.

J. Blanchette and A. Popescu. 2013. Mechanizing the Metatheory of Sledgehammer. In *FroCos*. Springer. to appear.

N. Bourbaki. 1964. Univers. In *Séminaire de Géométrie Algébrique du Bois Marie - Théorie des topos et cohomologie étale des schémas*. Springer, 185–217.

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

H. Curry and R. Feys. 1958. *Combinatory Logic*. North-Holland, Amsterdam.

H. B. Enderton. 1972. *A Mathematical Introduction to Logic*. Academic Press.

W. Farmer, J. Guttman, and F. Thayer. 1992. Little Theories. In *Conference on Automated Deduction*, D. Kapur (Ed.). 467–581.

W. Farmer, J. Guttman, and F. Thayer. 1993. IMPS: An Interactive Mathematical Proof System. *Journal of Automated Reasoning* 11, 2 (1993), 213–248.

J. Goguen and R. Burstall. 1992. Institutions: Abstract Model Theory for Specification and Programming. *Journal of the Association for Computing Machinery* 39(1) (1992), 95–146.

J. Goguen, Timothy Winkler, J. Meseguer, K. Futatsugi, and J. Jouannaud. 1993. Introducing OBJ. In *Applications of Algebraic Specification using OBJ*, J. Goguen, D. Coleman, and R. Gallimore (Eds.). Cambridge.

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

R. Harper, D. Sannella, and A. Tarlecki. 1994. Structured Presentations and Logic Representations. *Annals of Pure and Applied Logic* 67 (1994), 113–160.

F. Horozal. 2014. *A Framework for Defining Declarative Languages*. Ph.D. Dissertation. Jacobs University Bremen.

F. Horozal and F. Rabe. 2011. Representing Model Theory in a Type-Theoretical Logical Framework. *Theoretical Computer Science* 412, 37 (2011), 4919–4945.

W. Howard. 1980. The formulas-as-types notion of construction. In *To H.B. Curry: Essays on Combinatory Logic, Lambda-Calculus and Formalism*. Academic Press, 479–490.

F. Lawvere. 1963. *Functional Semantics of Algebraic Theories*. Ph.D. Dissertation. Columbia University.

P. Martin-Löf. 1974. An Intuitionistic Theory of Types: Predicative Part. In *Proceedings of the '73 Logic Colloquium*. North-Holland, 73–118.

R. Milner, M. Tofte, R. Harper, and D. MacQueen. 1997. *The Definition of Standard ML,* Revised edition. MIT Press.

T. Mossakowski, C. Maeder, and K. Lüttich. 2007. The Heterogeneous Tool Set. In *Tools and Algorithms for the Construction and Analysis of Systems 2007 (Lecture Notes in Computer Science)*, O. Grumberg and M. Huth (Ed.), Vol. 4424. 519–522.

L. Paulson. 1994. *Isabelle: A Generic Theorem Prover*. Lecture Notes in Computer Science, Vol. 828. Springer.

F. Pfenning and C. Schürmann. 1999. System Description: Twelf - A Meta-Logical Framework for Deductive Systems. In *Automated Deduction*, H. Ganzinger (Ed.). 202–206.

G. Plotkin, J. Power, D. Sannella, and R. Tennent. 2000. Lax Logical Relations. In *Colloquium on Automata, Languages and Programming (LNCS)*, Vol. 1853. Springer, 85–102.

F. Rabe. 2013a. A Logical Framework Combining Model and Proof Theory. *Mathematical Structures in Computer Science* 23, 5 (2013), 945–1001.

F. Rabe. 2013b. The MMT API: A Generic MKM System. In *Intelligent Computer Mathematics*, J. Carette, D. Aspinall, C. Lange, P. Sojka, and W. Windsteiger (Eds.). Springer, 339–343.

F. Rabe. 2014. How to Identify, Translate, and Combine Logics? *Journal of Logic and Computation* (2014). doi:10.1093/logcom/exu079.

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

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

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

J. Reynolds. 1974. On the relation between direct and continuation semantics. In *Second Colloq. Automata, Languages and Programming (LNCS)*. Springer, 141–156.

D. Sannella and M. Wirsing. 1983. A Kernel Language for Algebraic Specification and Implementation. In *Fundamentals of Computation Theory*, M. Karpinski (Ed.). Springer, 413–427.

C. Schürmann and J. Sarnat. 2008. Structural Logical Relations. In *Logic in Computer Science*, F. Pfenning (Ed.). IEEE Computer Society Press, 69–80.

C. Schürmann and M. Stehr. 2004. An Executable Formalization of the HOL/Nuprl Connection in the Metalogical Framework Twelf. In *11th International Conference on Logic for Programming Artificial Intelligence and Reasoning*, F. Baader and A. Voronkov (Eds.). Springer.