A

Logical Relations for a Logical Framework

FLORIAN RABE, Jacobs University Bremen, Bremen, Germany
and KRISTINA SOJAKOVA, Carnegie Mellon University, Pittsburgh, PA, USA

Logical relations are a central concept used to study various higher-order type theories and occur frequently in the proofs of a wide variety of meta-theorems. Besides extending the logical relation principle to more general languages, an important research question has been how to represent and thus verify logical relation arguments in logical frameworks.

We formulate a theory of logical relations for dependent type theory (DTT) with $\beta\eta$-equality which guarantees that any valid logical relation satisfies the Basic Lemma. Our definition is syntactic and reflective in the sense that a relation at a type is represented as a DTT type family but also permits expressing semantic definitions.

We use the Edinburgh Logical Framework (LF) incarnation of DTT and implement our notion of logical relations in the type-checker Twelf. This enables us to formalize and mechanically decide the validity of logical relation arguments. Furthermore, our implementation includes a module system so that logical relations can be built modularly.

We validate our approach by formalizing and verifying several syntactic and semantic meta-theorems in Twelf. Moreover, we show how object languages encoded in DTT can inherit a notion of logical relation from the logical framework.

Categories and Subject Descriptors: F.4.1 [**Mathematical Logic and Formal Languages**]: Mathematical Logic — *Lambda Calculus and Related Systems, Mechanical Theorem Proving*

General Terms: Theory

Additional Key Words and Phrases: Logical Relation, Parametricity, Dependent Type Theory, Logical Framework, LF, Twelf, Module System

ACM Reference Format:
Rabe, F., Sojakova, K. 2012. Logical relations in a logical framework. ACM Trans. Comput. Logic V, N, Article A (January YYYY), 32 pages.
DOI = 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

1. INTRODUCTION AND RELATED WORK

Logical relations provide a general principle for induction on the derivations of a formal system. Because they subsume many important constructions, they have become a key concept in the study of higher-order type theories.

A typical logical relations argument for a given type theory proceeds by *(i)* defining a type-indexed family of relations on terms, constructed inductively from the corresponding definitions at smaller types, and *(ii)* showing that all well-typed terms respect the relation at the respective type . The latter theorem is referred to by various names including *abstraction theorem*, *parametricity*, and *Basic Lemma*, and we will go with the latter for consistency.

Research on logical relations can be roughly divided into three directions.

The first author conducted parts of his research during a stay at IT University Copenhagen.

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

(1) Generalizations. Many type theories admit the definition of logical relations, most importantly simple type theory and System F, although formulating the definition and proving the Basic Lemma may be difficult in individual cases. Advanced examples include extensions of F_ω with representation types or higher-kinded polymorphisms in [Vytiniotis and Weirich 2010] and [Atkey 2012], respectively; a system with explicit strictness in [Johann and Voigtländer 2006]; a language with casting in [Neis et al. 2011]; and the calculus of inductive constructions with an impredicative sort in [Keller and Lasson 2012]. A very general result is given in [Bernardy et al. 2010] for a large class of pure type systems. In fact, authors may even use the Basic Lemma as a guiding intuition when defining new languages such as originally done in [Reynolds 1983].

Specific definitions of the concept "logical relation" vary. Logical relations are usually stated in a meta-language, i.e., a language talking about the type theory of interest. Then, depending on the the meta-language, the definitions can be classified as *semantic* (using, e.g., a set or domain-theoretical universe) or *syntactic* (using a second formal system). We can also distinguish *reflective* definitions as the special case of syntactic ones where the type theory is used as its own meta-language. Semantic definitions usually follow [Reynolds 1983], and early examples of syntactic definitions are given in [Mairson 1991] and [Plotkin and Abadi 1993]. The possibility of reflective definitions has been studied in [Bernardy et al. 2010].

A specialty of semantic definitions is that the type theory is *interpreted* in the meta-language (e.g., by assigning a set to every type), and the logical relation provides relations between different interpretations (e.g., in [Reynolds 1974]). In principle, a similar approach is possible for syntactic definitions. However, it is common to choose the meta-language in such a way that it subsumes the type theory so that the interpretation is the identity function.

Somewhat of an intermediate class arises if the syntactic interpretation is used in a semantic logical relation. In this case, all types are interpreted as their set of terms so that the logical relation is a family of relations between terms after all. Such relations are used in [Coquand and Gallier 1990] and [Harper and Pfenning 2005].

(2) Applications. Applications of logical relations include a wide variety of *meta-level theorems*. Examples include proving normalization of type theories [Statman 1985], [Girard et al. 1989], [Coquand and Gallier 1990]; showing completeness of equivalence algorithms [Harper and Pfenning 2005]; characterizing lambda definability [Plotkin 1973]; proving correctness of program transformations [Johann 2002], and relating different denotational semantics [Reynolds 1974].

Another group of applications uses a fixed logical relation and then instantiates the Basic Lemma to obtain *object-level theorems*. This is valuable even in the seemingly-trivial case in which the fixed logical relation is uniquely determined because there are no user-declared operators or variables. In particular, in polymorphic languages this is called theorems-for-free or parametricity: [Wadler 1989] shows how to use Reynold's abstraction theorem [Reynolds 1983] to derive important properties of parametrically polymorphic functions solely from their type.

(3) Representations. As logical relations are usually stated in a meta-language, their representation in a proof assistant or logical framework usually requires a representation of that meta-language. As the meta-language is often more complex than the object language, this impedes the development of tool support for logical relation arguments. This includes the formalization and verification of the Basic Lemma itself as well as the application of the Basic Lemma to specific logical relations as part of some other formal proof.

[Atkey 2009] formalizes system F in Coq [Bertot and Castéran 2004] and uses Coq itself as the meta-language to define logical relations for System F and prove the Basic

Lemma. [Schürmann and Sarnat 2008] formalize the logical relation-based normalization proof for simple type theory: they formalize both simple type theory and an appropriate meta-language in Twelf [Pfenning and Schürmann 1999] and then establish the Basic Lemma for the specific logical relation needed.

Basic tool support in the theorems-for-free style was developed for Agda [Norell 2005] as part of [Bernardy et al. 2010] and experimentally for Coq as part of [Keller and Lasson 2012]. [Böhme 2007] gives an online free theorem generator for sublanguages of Haskell.

Contribution. The central idea of our logical relations can be summarized as follows. Consider DTT-signature morphisms $\mu_1, \ldots, \mu_n : S \to T$ for DTT-signatures S and T. A logical relation $r : \mu_1 \times \ldots \times \mu_n$ on these morphisms inductively defines a T-predicate $r(A) : \mu_1(A) \to \ldots \to \mu_n(A) \to \texttt{type}$ for all S-types A. Our central result is to show the *Basic Lemma*: If r is well-typed at all base cases (i.e., for the base types and constants declared in S), then it is well-typed for all expressions. Thus, the judgment for the validity of logical relations is simple and decidable.

This yields a definition of logical relations for DTT that attempts to *unify* the above-mentioned semantic and syntactic approaches: Intuitively, S represents the DTT-signature of interest, and T represents the meta-language in which logical relations are stated. Then the μ_i represent interpretations of S in the meta-language T.

In particular, semantic definitions arise when T explicitly represents the semantic meta-language (e.g., as in [Iancu and Rabe 2011]); each μ_i can be understood as giving one denotational semantics of S. Syntactic definitions arise when T represents a formal system; the special case when $T = S$ corresponds to reflective definitions. If S is included into T, the morphisms μ_i are often chosen to be the inclusion morphism $S \hookrightarrow T$.

The most important application of our work is based on the use of DTT as a *logical framework* [Harper et al. 1993]. Our logical relations intend to permit the representation of logical relations for any object language represented in DTT, including, e.g., the pure type systems from [Bernardy et al. 2010]. In those cases, S represents the object language in DTT – e.g., simple type theory – and T the meta-language. Again this representation aims to uniformly subsume semantic and syntactic logical relations of S, including the situation when, in the semantic case, the μ_i are different models of S.

We have *implemented* our logical relations as a new primitive in the Twelf implementation [Pfenning and Schürmann 1999] of DTT. Thus, the formalization of logical relation arguments in Twelf requires users to define the relation only at the base cases (i.e., for the identifiers declared in S), and Twelf decides the validity of the logical relation and thus guarantees the Basic Lemma. Moreover, our implementation is integrated with the Twelf module system so that logical relations can be constructed and verified modularly.

Finally we present a number of *applications* of our results to formalize and verify a variety of syntactic and semantic meta-theorems about formal languages. Examples include the type preservation of language translations, the observational equivalence of two model theories, or the termination of β-reduction. The implementation and the Twelf sources of all examples are available online at [Rabe and Sojakova 2012].

Related Work. [Takeuti 2001] sketches a definition of logical relation for the systems of the λ-cube, which would include DTT. Independently from our work, [Bernardy et al. 2010] gives a definitions of logical relations for pure type systems, which includes DTT; this definition is reflective, and a syntactic variant was later given in [Bernardy and Lasson 2011]. This approach was continued in [Keller and Lasson 2012] to obtain a reflective definition for a variant of the calculus of constructions.

Our definition and the one in [Bernardy et al. 2010] share the key idea that a logical relation induces a single translation function that maps *(i)* types to predicates (which define the relation at a given type) and *(ii)* terms to proofs of these predicates (which proves the

Basic Lemma). This is crucial because the nature of DTT requires *(i)* and *(ii)* to be defined in a mutual induction.

But then both approaches move in different directions. [Bernardy et al. 2010] generalizes the key idea to arbitrary pure type systems whereas we treat only the minimal dependently-typed system. Our approach, on the other hand, generalizes the key idea to logical relations between arbitrary interpretations $\mu_i : S \to T$. Moreover, we use η in addition to β-conversion; while conceptually straightforward, this significantly complicates the proof of the Basic Lemma, which we give in full detail.

Both results share an important special case. We can first specialize the results in [Bernardy et al. 2010] to DTT, i.e., the pure type system with axiom $* : \Box$ and rules $(*,*,*), (*,\Box,\Box)$. Technically, this pure type system does not meet the reflexivity assumption of [Bernardy et al. 2010], but it is straightforward to embed it into one that does. Second, we can specialize our results to the case where S and T are the empty signature and all μ_i are empty morphisms. Then both results yield the same statement of the Basic Lemma for DTT. The embedding mentioned above has the advantage that it automatically extends the Basic Lemma to contexts with type variables, which do not exist in pure DTT. To handle type declarations in our setting, types would be declared in the signature S.

Further related work includes [Harper and Pfenning 2005] and [Coquand and Gallier 1990], which employ logical relation arguments to establish meta-theorems about a dependently-typed language. [Harper and Pfenning 2005] prove the completeness of an equality algorithm for LF by using type erasure and constructing a Kripke logical relation for the resulting simple type theory. While this approach works very well for the intended purpose, it does not yield a general definition of logical relation for DTT and thus does not extend to other applications.

[Coquand and Gallier 1990] use logical relations for the Calculus of Constructions to show strong normalization. Like us, they operate in a dependently-typed setting with no type erasure. However, they restrict themselves to β-reduction and consider only the case of unary logical relations. Furthermore, they interpret types semantically as sets of terms and express predicates on types as their subsets, whereas we express predicates as DTT type families. While the latter makes relations more difficult to express, it has the advantage of formalizing all implicit assumptions and permitting mechanic verification of logical relations.

Consequently, the treatment of type families differs in the respect that in [Coquand and Gallier 1990], a type family $c : A \to \texttt{type}$ determines a family of predicates indexed by terms of type A. In our definition, the indexing only considers those terms of type A which satisfy the relation at A. Apart from being more uniform, this distinction is needed for the syntactic version since many such predicates would otherwise be difficult to express parametrically as DTT type families.

Both results use a semantic logical relation on the syntactic interpretation of types as sets of terms. These are non-standard (Henkin) interpretations, i.e., they do not interpret function types compositionally as exponentials. Because morphisms can only represent standard interpretations, this precludes representing those logical relations in our style. However, we expect that our treatment of logical relations carries over naturally to an appropriate definition of non-standard morphisms.

Finally, [Schürmann and Sarnat 2008] use DTT as a logical framework to formalize an individual logical relation argument. Their formalization of both the object logic and an appropriate assertion logic in Twelf corresponds to our use of the DTT-signatures S and T. In our terms, the given logical relation corresponds to a unary logical relation on the inclusion morphism $S \hookrightarrow T$.

	Grammar		Typing	Equality
Signatures	S	$::= \cdot \mid S,\ c : A \mid S,\ a : K$	$\vdash S\ \mathtt{sig}$	
Morphisms	μ	$::= \cdot \mid \mu,\ c := M \mid \mu,\ a := C$	$\vdash \mu : S \to S$	
Contexts	Γ	$::= \cdot \mid \Gamma,\ x : A$	$\vdash_S \Gamma\ \mathtt{ctx}$	
Terms	M	$::= c \mid x \mid \lambda x : A.\, M \mid M\, M$	$\Gamma \vdash_S M : A$	$\Gamma \vdash_S M = M' : A$
Type families	C, A	$::= a \mid \lambda x : A.\, C \mid C\, M \mid \Pi x : A.\, A$	$\Gamma \vdash_S C : K$	$\Gamma \vdash_S C = C' : K$
Kinds	K	$::= \mathtt{type} \mid \Pi x : A.\, K$	$\Gamma \vdash_S K : \mathtt{kind}$	$\Gamma \vdash_S K = K' : \mathtt{kind}$

Fig. 1. DTT Syntax and Judgments

2. PRELIMINARIES

Intuitively, DTT arises from simple type theory by adding dependent function types $\Pi x : A_1.\, A_2$ and kinded (i.e., type-valued) symbols $a : \Pi x_1 : A_1. \ldots \Pi x_n : A_n.\, \mathtt{type}$ for types A_i. It is distinguished from more expressive languages such as the calculus of constructions by the absence of type variables and polymorphism.

We use the well-known LF incarnation of DTT as given in [Harper et al. 1993; Pfenning 2001; Harper and Pfenning 2005]. Fig. 1 gives an overview of the grammar and judgments.

As usual, we omit the subscript S in $\vdash_S$ if it is clear from the context, and we write $\Pi x : A.\, U$ as $A \to U$ if x does not occur free in U. Moreover, we adopt the convention of writing M, N for terms; C for type families of any kind K; A, B for the special type families of kind $\mathtt{type}$; and K for kinds. Arbitrary expressions (including the symbol $\mathtt{kind}$) will be denoted by U, V, W. Finally, we write the simultaneous capture-avoiding substitution of M_i for x_i in U as $U[M_1/x_1, \ldots, M_n/x_n]$. Finally, we omit the types of bound variables if they can be inferred easily.

To be self-contained, we repeat the judgmental rules for DTT in Fig. 2. They closely follow the ones given in [Harper and Pfenning 2005], except that we allow λ-abstraction at the level of type families, which corresponds to the treatment of DTT as a corner of the λ-cube. In particular, we use functional extensionality as a primitive rule, which is equivalent to η-conversion; for clarity, we include important admissible rules in [brackets]. For simplicity, the rules for contexts, kinds, type families, and terms presuppose the ambient signature S to be valid.

We will freely use the exchange, weakening, and substitution properties of LF, as well as the fact that all judgements imply the validity of the contexts involved. Furthermore, we will use the facts that if $\Gamma \vdash M : A$ then $\Gamma \vdash A : \mathtt{type}$, if $\Gamma \vdash C : K$ then $\Gamma \vdash K : \mathtt{kind}$, and if $\Gamma \vdash U = V : W$ then $\Gamma \vdash U : W$ and $\Gamma \vdash V : W$.

Our logical relations depend crucially on and will be structurally very similar to LF signature morphisms (see, e.g., [Harper et al. 1994]). Since the notion or morphisms is somewhat less common, we state the definition and its main property in detail:

Definition 2.1. A valid morphism $\vdash T : \mu \to S$ consists of a T-term $\vdash_T \mu_c : \mu(A)$ for every constant $c : A$ declared in S and a T-type family $\vdash_T \mu_a : \mu(K)$ for every type family $a : K$ declared in S. Here $\mu(-)$ is the homomorphic extension of μ, which maps S-expressions and contexts to T-expressions and contexts. $\mu(-)$ is defined inductively as follows:

Valid signatures and contexts

$$\frac{}{\vdash \cdot \ \mathtt{sig}} \qquad \frac{\vdash S \ \mathtt{sig} \qquad \cdot \vdash_S A : \mathtt{type}}{\vdash S, c : A \ \mathtt{sig}} \qquad \frac{\vdash S \ \mathtt{sig} \qquad \cdot \vdash_S K : \mathtt{kind}}{\vdash S, a : K \ \mathtt{sig}} \qquad \frac{}{\vdash \cdot \ \mathtt{ctx}} \qquad \frac{\Gamma \vdash A : \mathtt{type}}{\vdash \Gamma, x : A \ \mathtt{ctx}}$$

Valid kinds

$$\frac{\vdash \Gamma \ \mathtt{ctx}}{\Gamma \vdash \mathtt{type} : \mathtt{kind}} \qquad \frac{\Gamma, x : A \vdash K : \mathtt{kind}}{\Gamma \vdash \Pi x : A.K : \mathtt{kind}} \qquad \frac{\Gamma \vdash A = A' : \mathtt{type} \qquad \Gamma, x : A \vdash K = K' : \mathtt{kind}}{\Gamma \vdash \Pi x : A.K = \Pi x : A'.K' : \mathtt{kind}}$$

Valid type families

$$\frac{\vdash \Gamma \ \mathtt{ctx} \qquad a : K \in S}{\Gamma \vdash a : K} \qquad \frac{\Gamma \vdash C : \Pi x : A.K \qquad \Gamma \vdash M : A}{\Gamma \vdash C \ M : K[M/x]} \qquad \frac{\Gamma \vdash C = C' : \Pi x : A.K \qquad \Gamma \vdash M = M' : A}{\Gamma \vdash C \ M = C' \ M' : K[M/x]}$$

$$\frac{\Gamma, x : A \vdash C : K}{\Gamma \vdash \lambda x : A.C : \Pi x : A.K} \qquad \left[\frac{\Gamma \vdash A = A' : \mathtt{type} \qquad \Gamma, x : A \vdash C = C' : K}{\Gamma \vdash \lambda x : A.C = \lambda x : A'.C' : \Pi x : A.K}\right] \qquad \frac{\Gamma, x : A \vdash B : \mathtt{type}}{\Gamma \vdash \Pi x : A.B : \mathtt{type}}$$

$$\frac{\Gamma \vdash A = A' : \mathtt{type} \qquad \Gamma, x : A \vdash B = B' : \mathtt{type}}{\Gamma \vdash \Pi x : A.B = \Pi x : A'.B' : \mathtt{type}} \qquad \frac{\Gamma, x : A \vdash C \ x = C' \ x : K \qquad x \notin \mathit{Free}(C),\ x \notin \mathit{Free}(C')}{\Gamma \vdash C = C' : \Pi x : A.K}$$

$$\left[\frac{\Gamma \vdash C : \Pi x : A.K \qquad x \notin \mathit{Free}(C)}{\Gamma \vdash \lambda x : A.C \ x = C : \Pi x : A.K}\right] \qquad \frac{\Gamma, x : A \vdash C : K \qquad \Gamma \vdash M : A}{\Gamma \vdash (\lambda x : A.C) \ M = C[M/x] : K[M/x]}$$

Valid terms

$$\frac{\vdash \Gamma \ \mathtt{ctx} \qquad c : A \in S}{\Gamma \vdash c : A} \qquad \frac{\vdash \Gamma \ \mathtt{ctx} \qquad x : A \in \Gamma}{\Gamma \vdash x : A} \qquad \frac{\Gamma \vdash M : \Pi x : A.B \qquad \Gamma \vdash N : A}{\Gamma \vdash M \ N : B[N/x]}$$

$$\frac{\Gamma \vdash M = M' : \Pi x : A.B \qquad \Gamma \vdash N = N' : A}{\Gamma \vdash M \ N = M' \ N' : B[N/x]} \qquad \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x : A.M : \Pi x : A.B}$$

$$\left[\frac{\Gamma \vdash A = A' : \mathtt{type} \qquad \Gamma, x : A \vdash M = M' : B}{\Gamma \vdash \lambda x : A.M = \lambda x : A'.M' : \Pi x : A.B}\right] \qquad \frac{\Gamma, x : A \vdash M \ x = M' \ x : B \qquad x \notin \mathit{Free}(M),\ x \notin \mathit{Free}(M')}{\Gamma \vdash M = M' : \Pi x : A.B}$$

$$\left[\frac{\Gamma \vdash M : \Pi x : A.B \qquad x \notin \mathit{Free}(M)}{\Gamma \vdash \lambda x : A.M \ x = M : \Pi x : A.B}\right] \qquad \frac{\Gamma, x : A \vdash M : B \qquad \Gamma \vdash N : A}{\Gamma \vdash (\lambda x : A.M) \ N = M[N/x] : B[N/x]}$$

Equality

$$\frac{\Gamma \vdash U : V}{\Gamma \vdash U = U : V} \qquad \frac{\Gamma \vdash U = U' : V}{\Gamma \vdash U' = U : V} \qquad \frac{\Gamma \vdash U = U' : V \qquad \Gamma \vdash U' = U'' : V}{\Gamma \vdash U = U'' : V}$$

$$\frac{\Gamma \vdash U : V \qquad \Gamma \vdash V = V' : W}{\Gamma \vdash U : V'} \qquad \frac{\Gamma \vdash U = U' : V \qquad \Gamma \vdash V = V' : W}{\Gamma \vdash U = U' : V'}$$

Fig. 2. Typing Rules of DTT

$$\begin{array}{ll} \mu(\cdot) & = \cdot \\ \mu(\Gamma, x : A) & = \mu(\Gamma), x : \mu(A) \\ \mu(\lambda x : A.C) & = \lambda x : \mu(A).\mu(C) \\ \mu(C \ M) & = \mu(C) \ \mu(M) \\ \mu(\Pi x : A.B) & = \Pi x : \mu(A).\mu(B) \\ \mu(a) & = \mu_a \end{array} \qquad \begin{array}{ll} \mu(\mathtt{kind}) & = \mathtt{kind} \\ \mu(\mathtt{type}) & = \mathtt{type} \\ \mu(\Pi x : A.K) & = \Pi x : \mu(A).\mu(K) \\ \mu(\lambda x : A.M) & = \lambda x : \mu(A).\mu(M) \\ \mu(M \ N) & = \mu(M) \ \mu(N) \\ \mu(x) & = x \\ \mu(c) & = \mu_c \end{array}$$

Intuitively, a morphism $S \to T$ formalizes a compositional type-preserving translations from S to T; thus, it demonstrates that all notions of S can be interpreted, represented, or realized in T. An example will be given below in Ex. 3.1. The key property justifying this intuition is the following:

THEOREM 2.2 (MORPHISM INVARIANTS). *If $\vdash \mu : S \to T$, then we have the following invariants:*

$$\begin{array}{llll} \text{if} & \vdash_S \Gamma\ \mathtt{ctx} & \text{then} & \vdash_T \mu(\Gamma)\ \mathtt{ctx} \\ \text{if} & \Gamma \vdash_S U : V & \text{then} & \mu(\Gamma) \vdash_T \mu(U) : \mu(V) \\ \text{if} & \Gamma \vdash_S U = V : W & \text{then} & \mu(\Gamma) \vdash_T \mu(U) = \mu(V) : \mu(W) \end{array}$$

Conversely, all translations that are compositional in the sense of the homomorphic extension and that satisfy Thm. 2.2 are morphisms.

3. LOGICAL RELATIONS

We will define n-ary logical relations $r : \mu_1 \times \ldots \times \mu_n$ as relations on LF-signature morphisms $\mu_1, \ldots, \mu_n : S \to T$. Throughout this section, we will fix S and T as well as $\mu_1, \ldots, \mu_n$, and we will call the signatures S and T the domain and codomain of the relation, respectively.

3.1. General Ideas

As seen in Def. 2.1 and Thm. 2.2, morphisms are compositional mappings of S-expressions to T-expressions that preserve typing, kinding, and equality judgments. We will define logical relations in a very similar way, albeit using a different type preservation property. A logical relation $r : \mu_1 \times \ldots \times \mu_n$ associates to every closed S-type A a T-type family $r(A) : \mu_1(A) \to \ldots \to \mu_n(A) \to \mathtt{type}$, which represents an n-ary relation on the different translations of A. Every closed S-term $M : A$ is then mapped to a T-term $r(M) : r(A)\ \mu_1(M)\ \ldots\ \mu_n(M)$, which proves that the different translations of M are related.

In general, an S-expression with m free variables will be mapped to a T-expression with $m(n+1)$ free variables: each variable x will give rise to n copies of x (one for each morphism μ_i) plus an extra variable asserting that all of these copies are related. At a function type, terms will be considered related if and only if they map related arguments to related values. This intuition is known from simple function types and remains true for dependent functions: the only difference is that the related argument and, crucially, the proof that they are related are used to construct the relations at later argument types.

As the general definition can be difficult to read, we will first consider a simple motivating example of a unary logical relation, i.e., $n = 1$ and then give the formal definition in Sect. 3.2:

Example 3.1 (*Type Preservation of the Church-Curry Translation*). As a simple example let us consider the Church and Curry encodings of simple type theory. In the Church encoding, also referred to as intrinsically typed, terms are indexed by types. Each term thus has a unique type and terms which are not well-typed are never constructed.

$$\begin{array}{l} \%\mathtt{sig}\ \mathit{Church} = \{ \\ \quad \mathit{tp} \quad : \quad \mathtt{type} \\ \quad \Rightarrow \quad : \quad \mathit{tp} \to \mathit{tp} \to \mathit{tp} \qquad\qquad\qquad\qquad\qquad\qquad \%\mathtt{infix}\ \Rightarrow \\ \quad \mathit{tm} \quad : \quad \mathit{tp} \to \mathtt{type} \\ \quad \mathit{lam} \quad : \quad \Pi A.\Pi B.\ (\mathit{tm}\ A \to \mathit{tm}\ B) \to \mathit{tm}\ (A \Rightarrow B) \\ \quad \mathit{app} \quad : \quad \Pi A.\Pi B.\ \mathit{tm}\ (A \Rightarrow B) \to \mathit{tm}\ A \to \mathit{tm}\ B \\ \} \end{array}$$

Here we mimic the concrete syntax of our Twelf implementation [Pfenning and Schürmann 1999; Rabe and Schürmann 2009] and use the keywords %`sig` to introduce signatures and %`infix` to introduce infix notations for certain binary symbols.

In the Curry encoding, also known as extrinsically typed, terms are defined untyped and the typing relation is given separately. This allows the assignment of multiple types to a single term, if needed.

$$
\begin{array}{lcll}
\%\texttt{sig}\ \mathit{Curry} = \{ & & & \\
\quad \mathit{tp} & : & \texttt{type} & \\
\quad \Rightarrow & : & \mathit{tp} \to \mathit{tp} \to \mathit{tp} & \%\texttt{infix}\ \Rightarrow \\
\quad \mathit{tm} & : & \texttt{type} & \\
\quad \mathit{lam} & : & (\mathit{tm} \to \mathit{tm}) \to \mathit{tm} & \\
\quad \mathit{app} & : & \mathit{tm} \to \mathit{tm} \to \mathit{tm} & \\
\quad \# & : & \mathit{tm} \to \mathit{tp} \to \texttt{type} & \%\texttt{infix}\ \# \\
\quad \#_{\mathit{lam}} & : & \Pi A.\Pi B.\Pi f.\ (\Pi x.\ x \# A \to (f\ x) \# B) \to (\mathit{lam}\ f) \# (A \Rightarrow B) & \\
\quad \#_{\mathit{app}} & : & \Pi A.\Pi B.\Pi f.\Pi x.\ f \# (A \Rightarrow B) \to x \# A \to (\mathit{app}\ f\ x) \# B & \\
\} & & &
\end{array}
$$

Here the DTT-type family # represents the typing relation of simple type theory using the Curry-Howard style representation of judgments-as-types.

It is easy give a translation from Church to Curry by simply erasing the type annotations from terms. We can formalize this concisely as a signature morphism (introduced by the keyword %`view` in Twelf):

$$
\begin{array}{lcl}
\%\texttt{view}\ \mathit{TypeEras} : \mathit{Church} \to \mathit{Curry} = \{ & & \\
\quad \mathit{tp} & := & \mathit{tp} \\
\quad \Rightarrow & := & \Rightarrow \\
\quad \mathit{tm} & := & \lambda A.\ \mathit{tm} \\
\quad \mathit{lam} & := & \lambda A.\lambda B.\lambda f.\ \mathit{lam}\ f \\
\quad \mathit{app} & := & \lambda A.\lambda B.\lambda f.\lambda a.\ \mathit{app}\ f\ a \\
\} & &
\end{array}
$$

The crucial mapping here is that of *tm*, which becomes a constant type family assigning to each type the set of untyped terms.

Thm. 2.2 guarantees that $\mathit{TypeEras}(-)$ preserves the DTT-typing relation, i.e., every *Church*-term $M : \mathit{tm}\,A$ is mapped to a *Curry*-term $\mathit{TypeEras}(M) : \mathit{tm}$. We now aim to show that *TypeEras* also preserves the typing relation of the encoded object language, i.e., that we also have that the type $\mathit{TypeEras}(M) \ \#\ \mathit{TypeEras}(A)$ in *Curry* is inhabited. We formalize this as a unary logical relation *TypePres* (introduced by the keyword %`rel`) on the morphism *TypeEras*:

$$
\begin{array}{lcl}
\%\texttt{rel}\ \mathit{TypePres} : \mathit{TypeEras} = \{ & & \\
\quad \mathit{tp} & := & \lambda A : \mathit{tp}.\ \mathit{unit} \\
\quad \mathit{tm} & := & \lambda A : \mathit{tp}.\ \lambda\_.\ \lambda x : \mathit{tm}.\ x \# A \\
\quad \Rightarrow & := & \lambda A : \mathit{tp}.\ \lambda\_.\ \lambda B : \mathit{tp}.\ \lambda\_.\ * \\
\quad \mathit{app} & := & \lambda A : \mathit{tp}.\ \lambda\_.\ \lambda B : \mathit{tp}.\ \lambda\_. \\
 & & \quad \lambda f : \mathit{tm}.\ \lambda f^* : f \# (A \Rightarrow B).\ \lambda x : \mathit{tm}.\ \lambda x^* : x \# A.\ \#_{\mathit{app}}\ A\ B\ f\ x\ f^*\ x^* \\
\quad \mathit{lam} & := & \lambda A : \mathit{tp}.\ \lambda\_.\ \lambda B : \mathit{tp}.\ \lambda\_. \\
 & & \quad \lambda f : \mathit{tm} \to \mathit{tm}.\ \lambda f^* : (\Pi x : \mathit{tm}.\ x \# A \to (f\ x) \# B).\ \#_{\mathit{lam}}\ A\ B\ f\ f^* \\
\} & &
\end{array}
$$

Firstly, *TypePres* assigns a *Curry*-predicate $\mathit{TypePres}(tp) : tp \to \mathtt{type}$ to the *Church*-type tp. As there is nothing to prove about terms of type tp, we choose any trivially true predicate; the easiest solution is to assume a unit type $* : \mathit{unit}$. In the later declarations, we use anonymous variables _ for all the variables of type *unit*.

Secondly, *TypePres* assigns a family

$$\mathit{TypePres}(tm) : \Pi A : \mathit{TypeEras}(tp).\, \Pi A^* : \mathit{TypePres}(tp)\ A.\, \big(\mathit{TypeEras}(tm)\ A \to \mathtt{type}\big)$$

of *Curry*-predicates to the *Church*-type family tm. Just like $tm : tp \to \mathtt{type}$ is indexed by terms of type tp, $\mathit{TypePres}(tm)$ is indexed by terms A of type $\mathit{TypeEras}(tp)$ and proofs A^* that A is in the relation $\mathit{TypePres}(tp)$.

Given a *Church*-term $M : tm\ A$, the inductive translation will be defined such that $\mathit{TypePres}(tm\, A) = \mathit{TypePres}(tm)\ \mathit{TypeEras}(A)\ \mathit{TypePres}(A) = \lambda x : tm.\ x \# \mathit{TypeEras}(A)$. Thus, we have the *Curry*-judgment

$$\mathit{TypePres}(tm\, A)\ \mathit{TypeEras}(M)\ =\ \mathit{TypeEras}(M) \# \mathit{TypeEras}(A)$$

as intended.

This concludes the mapping of type family symbols. In the next step, we map each term symbol to a proof that the relation is satisfied. Of course, this is trivial for the constructors of tp. Since *app* takes four arguments, the case for $\mathit{TypePres}(app)$ takes eight: one for each argument and one for the assumption that they are in the relation, e.g., x^* is the assumption that x is in the relation. For *lam*, we have to show that whenever $f : tm \to tm$ satisfies the relation, then so does $lam\ f : tm\ (A \Rightarrow B)$. Since f has a function type, this yields an assumption f^* stating that f preserves the relation.

The Basic Lemma that we prove below in Thm. 3.9 guarantees that for any term $M : tm\ A$, the term $\mathit{TypeEras}(M)$ satisfies the predicate $\mathit{TypePres}(tm\ A)$, i.e., that the type $\mathit{TypeEras}(M) \# \mathit{TypeEras}(A)$ is inhabited.

3.2. Formal Definition

As mentioned above, $r(-)$ maps expressions with m free variables to expressions with $m(n+1)$ free variables. To handle this conveniently, we introduce a few auxiliary notations.

First we note that if $\Gamma = x_1 : A_1, \ldots, x_m : A_m$, the context $r(\Gamma)$ will be of the form

$$\begin{array}{l} x_1^1 : \mu_1'(A_1),\ \ldots,\ x_1^n : \mu_n'(A_1),\ x_1^* : r(A_1)\ x_1^1\ \ldots\ x_1^n, \\ \vdots \\ x_m^1 : \mu_1'(A_m),\ \ldots,\ x_m^n : \mu_n'(A_m),\ x_m^* : r(A_m)\ x_m^1\ \ldots\ x_m^n \end{array}$$

The need for all these variables is twofold. Firstly, we create a copy of $\mu_i'(\Gamma)$ for each morphism μ_i. Here μ_i' is like μ_i except that the variables of Γ are mapped to the respective copy, i.e., μ_i' maps x_j to x_j^i. Secondly, we add one variable x_j^* for every x_j in Γ, which serves as an assumption that the tuple $(x_j^1, \ldots, x_j^n)$ is in the relation.

Formally, we define μ_i' as follows:

Definition 3.2. If U is an S-expression with free variables among $x_1, \ldots, x_m$, we define $\mu_i'(-)$ by

$$\mu_i'(U) = \mu_i(U)[x_1^i/x_1,\ \ldots,\ x_m^i/x_m]$$

If $\Gamma = x_1 : A_1, \ldots, x_m : A_m$, we put

$$\mu_i'(\Gamma) = x_1^i : \mu_i'(A_1), \ldots, x_m^i : \mu_i'(A_m)$$

Note that, because μ_i' arises as the composition of a morphism and a substitution, it preserves validity, typing, kinding, and equality judgements.

We also use the following notations to bind sequences of variables and to apply functions to sequences of arguments:

NOTATION 3.3. *We use the following notations:*

— *For a context fragment* $\Gamma = x_1 : A_1, \ldots, x_m : A_m$*, we write* $\lambda\Gamma.U$ *for the expression* $\lambda x_1 : A_1 . \ldots . \lambda x_m : A_m . U$*. We write* $\Pi\Gamma.U$ *accordingly.*
— *For a list of terms* $\overline{M} = M_1, \ldots, M_m$ *and a list of variables* $\bar{x} = x_1, \ldots, x_m$*, we write* $U\ \overline{M}$ *for the application* $U\ M_1\ \ldots\ M_m$ *and* $[\overline{M}/\bar{x}]$ *for the substitution* $[M_1/x_1, \ldots, M_m/x_m]$*.*
— *In order to introduce fresh variables conveniently, we assume that for every variable name* x*, we have an unlimited supply of fresh variable names, which are denoted by adding superscripts to* x*.*

Moreover, we use the following notations to produce sequences of bound variables and sequences of arguments:

NOTATION 3.4. *For any type* A *and term* M *in* S *(not necessarily closed) we write*

— $f : \mu(A)$ *for the context fragment* $f^1 : \mu'_1(A),\ \ldots,\ f^n : \mu'_n(A)$
— $x : r^\mu(A)$ *for the context fragment* $x^1 : \mu'_1(A),\ \ldots,\ x^n : \mu'_n(A),\ x^* : r(A)\ x^1\ \ldots\ x^n$
— $\mu(M)$ *for the list of terms* $\mu'_1(M),\ \ldots,\ \mu'_n(M)$
— $r^\mu(M)$ *for the list of terms* $\mu'_1(M),\ \ldots,\ \mu'_n(M),\ r(M)$

Then we can finally define logical relation application $r(-)$ as follows:

Definition 3.5 (*Logical Relation*). Let $\mu_1, \ldots, \mu_n : S \to T$ be DTT-morphisms. Given a T-term r_c for every S-constant c and a T-type family r_a for every S-type family symbol a, we define by induction on the syntax (i) a mapping $r(-)$ from S-terms, type families, and contexts to T-terms, type families, and contexts, as well as (ii) for S-type families C, a mapping $r^C(-)$ from S-kinds to T-kinds:

$$
\begin{array}{ll}
r(\cdot) & = \cdot \\
r(\Gamma, x : A) & = r(\Gamma), x : r^\mu(A) \\
\\
r^A(\mathtt{type}) & = \mu'_1(A) \to \ldots \to \mu'_n(A) \to \mathtt{type} \\
r^C(\Pi x : A.K) & = \Pi x : r^\mu(A).r^{C\ x}(K) \\
\\
r(a) & = r_a \\
r(C\ M) & = r(C)\ r^\mu(M) \\
r(\lambda x : A.C) & = \lambda x : r^\mu(A).r(C) \\
r(\Pi x : A.B) & = \lambda f : \mu(\Pi x : A.B).\Pi x : r^\mu(A).r(B)\ (f^1\ x^1)\ \ldots\ (f^n\ x^n) \\
\\
r(c) & = r_c \\
r(x) & = x^* \\
r(M\ N) & = r(M)\ r^\mu(N) \\
r(\lambda x : A.M) & = \lambda x : r^\mu(A).\ r(M)
\end{array}
$$

r is called a *logical relation*, written $r : \mu_1 \times \ldots \times \mu_n : S \to T$, if for each constant $c : A$ and type family symbol $a : K$ in S we have

$$\vdash_T r_c : r(A)\ \mu(c) \quad \text{and} \quad \vdash_T r_a : r^a(K)$$

Remark 3.6. Note that, as intended, if the expressions M, C, K have free variables among $x_1, \ldots, x_m$, then $r(M)$, $r(C)$, $r^C(K)$ have free variables among $x_1^1, \ldots, x_1^n, x_1^*, \ldots, x_m^1, \ldots, x_m^n, x_m^*$.

Remark 3.7. In a language that extends DTT with abstraction over type variables, it is possible to unify the functions $r(-)$ and $r^C(-)$ into a single function $r(-)$ by putting (in

the unary case) $r(\texttt{type}) = \lambda A : \texttt{type}.\, \mu_1(A) \to \texttt{type}$. In the special case where $\mu_1 = id_S$ is the identity and can be dropped, this corresponds to the definition given in [Bernardy et al. 2010].

Example 3.8. Consider a logical relation $r : \mu_1 \times \mu_2 : S \to T$, and abbreviate $U^i := \mu'_i(U)$ for any expression U. Then we have

$$
\begin{array}{l}
r(\Pi x : A.\Pi y : B.C) = \\
\lambda f^1 : \Pi x^1 : A^1.\Pi y^1 : B^1.C^1. \\
\lambda f^2 : \Pi x^2 : A^2.\Pi y^2 : B^2.C^2. \\
\Pi x^1 : A^1.\Pi x^2 : A^2.\Pi x^* : r(A)\, x^1\, x^2. \\
\Pi y^1 : B^1.\Pi y^2 : B^2.\Pi y^* : r(B)\, y^1\, y^2. \\
r(C)\ (f^1\ x^1\ y^1)\ (f^2\ x^2\ y^2)
\end{array}
$$

Thus, uncurrying works as expected: Two functions f^1 and f^2 are related at $r(\Pi x : A.\Pi y : B.C)$ iff they map related argument tuples to related outputs.

As for morphisms, the key property of logical relations is that the characteristic invariant holds for all expressions iff it holds for all symbols:

THEOREM 3.9 (BASIC LEMMA). *Assume a logical relation $r : \mu_1 \times \ldots \times \mu_n : S \to T$. Then:*

$$
\begin{array}{lll}
\text{if} & \Gamma \vdash_S C : K \text{ then} & r(\Gamma) \vdash_T r(C) : r^C(K) \\
\text{if} & \Gamma \vdash_S M : A \text{ then} & r(\Gamma) \vdash_T r(M) : r(A)\ \mu(M)
\end{array}
$$

3.3. Proof of the Basic Lemma

The proof of the Basic Lemma is rather lengthy and while straightforward in nature, it requires some care to handle a few technical issues. Since typing/kinding and definitional equality are mutually recursive in DTT, the proof must be carried out by a mutual induction on derivations. Consequently, we need a more general induction hypothesis (Thm. 3.11) that also states the preservation of equality.

To simplify the induction, we modify some of the rules for typing and equality by adding kinding hypotheses; these are obviously redundant but needed to justify the application of the induction hypothesis. The modified rules are shown in Fig. 3, with the extra premises enclosed in {braces}.

First we observe that:

— We have $\mu'_i(U[N/x]) = \mu'_i(U)[\mu'_i(N)/x^i]$.

— If $\Gamma \vdash_S U : V$ and $\vdash_T r(\Gamma)\ \texttt{ctx}$[1] then $r(\Gamma) \vdash_T \mu'_i(U) : \mu'_i(V)$. Similarly, if $\Gamma \vdash_S U = V : W$ and $\vdash_T r(\Gamma)\ \texttt{ctx}$ then $r(\Gamma) \vdash_T \mu'_i(U) = \mu'_i(V) : \mu'_i(W)$.

Then we establish the following lemma:

LEMMA 3.10 (SUBSTITUTION). *For any S-expressions N, M, C, K we have*

$$
\begin{array}{l}
r(M[N/x]) = r(M)[r^\mu(N)/x^1, \ldots, x^n, x^*] \\
r(C[N/x]) = r(C)[r^\mu(N)/x^1, \ldots, x^n, x^*] \\
r^{C[N/x]}(K[N/x]) = r^C(K)[r^\mu(N)/x^1, \ldots, x^n, x^*]
\end{array}
$$

PROOF. By induction on the structure of M, C, and K respectively. □

[1] This condition will be guaranteed by the Logical Relation Invariants Thm. 3.11, once proved.

$$\frac{\Gamma \vdash M : \Pi x : A.B \qquad \{\Gamma, x : A \vdash B : \mathtt{type}\} \qquad \Gamma \vdash N : A}{\Gamma \vdash M\ N : B[N/x]}$$

$$\frac{\Gamma, x : A \vdash C : K \qquad \{\Gamma, x : A \vdash K : \mathtt{kind}\}}{\Gamma \vdash \lambda x : A.C : \Pi x : A.K} \qquad \frac{\Gamma, x : A \vdash M : B \qquad \{\Gamma, x : A \vdash B : \mathtt{type}\}}{\Gamma \vdash \lambda x : A.M : \Pi x : A.B}$$

$$\frac{\Gamma, x : A \vdash M\ x = M'\ x : B \qquad \{\Gamma, x : A \vdash B : \mathtt{type}\} \qquad x \notin \mathit{Free}(M),\ x \notin \mathit{Free}(M')}{\Gamma \vdash M = M' : \Pi x : A.B}$$

$$\frac{\Gamma, x : A \vdash C : K \qquad \{\Gamma, x : A \vdash K : \mathtt{kind}\} \qquad \Gamma \vdash M : A}{\Gamma \vdash (\lambda x : A.C)\ M = C[M/x] : K[M/x]}$$

$$\frac{\Gamma, x : A \vdash M : B \qquad \{\Gamma, x : A \vdash B : \mathtt{type}\} \qquad \Gamma \vdash N : A}{\Gamma \vdash (\lambda x : A.M)\ N = M[N/x] : B[N/x]}$$

$$\frac{\Gamma \vdash C = C' : K \qquad \{\Gamma \vdash K : \mathtt{kind}\}}{\Gamma \vdash C' = C : K} \qquad \frac{\Gamma \vdash M = M' : A \qquad \{\Gamma \vdash A : \mathtt{type}\}}{\Gamma \vdash M' = M : A}$$

$$\frac{\Gamma \vdash C = C' : K \qquad \Gamma \vdash C' = C'' : K \qquad \{\Gamma \vdash K : \mathtt{kind}\}}{\Gamma \vdash C = C'' : K}$$

$$\frac{\Gamma \vdash M = M' : A \qquad \Gamma \vdash M' = M'' : A \qquad \{\Gamma \vdash A : \mathtt{type}\}}{\Gamma \vdash M = M'' : A}$$

Fig. 3. Modified Typing Rules of DTT

Now the remainder of this section proves the main theorem:

THEOREM 3.11 (LOGICAL RELATION INVARIANTS). *For each logical relation* $r : \mu_1 \times \ldots \times \mu_n : S \to T$*, we have the following invariants:*

if	$\vdash_S \Gamma\ \mathtt{ctx}$	then	$\vdash_T r(\Gamma)\ \mathtt{ctx}$	(1)
if	$\Gamma \vdash_S K : \mathtt{kind},\ \Gamma \vdash_S C : K$	then	$r(\Gamma) \vdash_T r^C(K) : \mathtt{kind}$	(2)
if	$\Gamma \vdash_S C : K$	then	$r(\Gamma) \vdash_T r(C) : r^C(K)$	(3)
if	$\Gamma \vdash_S M : A$	then	$r(\Gamma) \vdash_T r(M) : r(A)\ \mu(M)$	(4)
if	$\Gamma \vdash_S K = K' : \mathtt{kind},\ \Gamma \vdash_S C : K$	then	$r(\Gamma) \vdash_T r^C(K) = r^C(K') : \mathtt{kind}$	(5)
if	$\Gamma \vdash_S K : \mathtt{kind},\ \Gamma \vdash_S C = C' : K$	then	$r(\Gamma) \vdash_T r^C(K) = r^{C'}(K) : \mathtt{kind}$	(6)
if	$\Gamma \vdash_S C = C' : K$	then	$r(\Gamma) \vdash_T r(C) = r(C') : r^C(K)$	(7)
if	$\Gamma \vdash_S M = M' : A$	then	$r(\Gamma) \vdash_T r(M) = r(M') : r(A)\ \mu(M)$	(8)

PROOF. The proof proceeds by induction on the (first) S-judgement. The invariant being verified is marked as I(N).

— For contexts:
  — I(1): Case $\vdash_S \cdot\ \mathtt{ctx}$. Trivial.

  — I(1): Case $\vdash_S \Gamma, x : A\ \mathtt{ctx}$ from the premise $\Gamma \vdash_S A : \mathtt{type}$. By IH we have

$$r(\Gamma) \vdash_T r(A) : \mu_1'(A) \to \ldots \to \mu_n'(A) \to \mathtt{type} \quad \text{hence}$$
$$r(\Gamma), x : \mu(A) \vdash_T r(A)\ x^1\ \ldots\ x^n : \mathtt{type}$$

  Thus the context $r(\Gamma), x : r^\mu(A)$ is well-formed as desired.

— For kinds:
  — I(2): Case $\Gamma \vdash_S \texttt{type} : \texttt{kind}$ from the premise $\vdash_S \Gamma\ \texttt{ctx}$, with any $\Gamma \vdash_S A : \texttt{type}$. By IH we have $\vdash_T r(\Gamma)\ \texttt{ctx}$. By validity of $r(\Gamma)$ we get $r(\Gamma) \vdash_T \mu_i'(A) : \texttt{type}$. Thus

$$r(\Gamma) \vdash_T \mu_1'(A) \to \ldots \to \mu_n'(A) \to \texttt{type} : \texttt{kind}$$

  as desired.

  — I(6): Case $\Gamma \vdash_S \texttt{type} : \texttt{kind}$ from the premise $\vdash_S \Gamma\ \texttt{ctx}$, and any $\Gamma \vdash_S A = A' : \texttt{type}$. Very similar to the previous case, with equality in place of the kinding judgement.

  — I(2): Case $\Gamma \vdash_S \Pi x : A.K : \texttt{kind}$ from the premise $\Gamma, x : A \vdash_S K : \texttt{kind}$, with any $\Gamma \vdash_S C : \Pi x : A.K$. By IH and the fact that $\Gamma, x : A \vdash_S C\ x : K$ we get

$$r(\Gamma), x : r^\mu(A) \vdash_T r^{C\ x}(K) : \texttt{kind} \quad \text{hence}$$
$$r(\Gamma) \vdash_T \Pi x : r^\mu(A).r^{C\ x}(K) : \texttt{kind}$$

  as desired.

  — I(6): Case $\Gamma \vdash_S \Pi x : A.K : \texttt{kind}$ from the premise $\Gamma, x : A \vdash_S K : \texttt{kind}$, with any $\Gamma \vdash_S C = C' : \Pi x : A.K$. Very similar to the previous case, with equality in place of the kinding judgement.

  — I(5): Case $\Gamma \vdash_S \Pi x : A.K = \Pi x : A'.K' : \texttt{kind}$ from the premises $\Gamma \vdash_S A = A' : \texttt{type}$ and $\Gamma, x : A \vdash_S K = K' : \texttt{kind}$, with any $\Gamma \vdash_S C : \Pi x : A.K$. By IH and the fact that $\Gamma, x : A \vdash_S C\ x : K$ we get

$$r(\Gamma) \vdash_T r(A) = r(A') : \mu_1'(A) \to \ldots \to \mu_n'(A) \to \texttt{type}$$
$$r(\Gamma), x : r^\mu(A) \vdash_T r^{C\ x}(K) = r^{C\ x}(K') : \texttt{kind}$$

  Now $r(\Gamma) \vdash_T \mu_i'(A) = \mu_i'(A') : \texttt{type}$. Furthermore, from the first IH we get

$$r(\Gamma), x : \mu(A) \vdash_T r(A)\ x^1\ \ldots\ x^n = r(A')\ x^1\ \ldots\ x^n : \texttt{type}$$

  Combined with the second IH this gives us

$$r(\Gamma) \vdash_T \Pi x : r^\mu(A).r^{C\ x}(K) = \Pi x : r^\mu(A').r^{C\ x}(K)' : \texttt{kind}$$

  as desired.

— For type families:
  — I(3): Case $\Gamma \vdash_S a : K$ for $a : K \in S$, from the premise $\vdash_S \Gamma\ \texttt{ctx}$. By IH we have $\vdash_T r(\Gamma)\ \texttt{ctx}$. By definition of logical relation, $\cdot \vdash_T r(a) : r^a(K)$. Since $r(\Gamma)$ is well-formed, we get $r(\Gamma) \vdash_T r(a) : r^a(K)$ as desired.

  — I(3): Case $\Gamma \vdash_S C\ M : K[M/x]$ from the premises $\Gamma \vdash_S C : \Pi x : A.K$ and $\Gamma \vdash_S M : A$. By IH we get

$$r(\Gamma) \vdash_T r(C) : \Pi x : r^\mu(A).r^{(C\ x)}(K)$$
$$r(\Gamma) \vdash_T r(M) : r(A)\ \mu(M)$$

  Thus, from the fact that $r(\Gamma) \vdash_T \mu_i'(M) : \mu_i'(A)$ and the second IH we get

$$r(\Gamma) \vdash_T r(C)\ r^\mu(M) : r^{(C\ x)}(K)[r^\mu(M)/x^1, \ldots, x^n, x^*]$$

  By the Substitution Lem. 3.10 we have

$$r^{(C\ x)}(K)[r^\mu(M)/x^1, \ldots, x^n, x^*] = r^{(C\ M)}(K[M/x])$$

hence

$$r(\Gamma) \vdash_T r(C)\ r^{\mu}(M) : r^{(C\ M)}(K[M/x])$$

as desired.

— I(7): Case $\Gamma \vdash_S C\ M = C'\ M' : K[M/x]$ from the premises $\Gamma \vdash_S C = C' : \Pi x : A.K$ and $\Gamma \vdash_S M = M' : A$. Very similar to the previous case, with equality in place of the kinding judgement.

— I(3): Case $\Gamma \vdash_S \lambda x : A.C : \Pi x : A.K$ from the premises $\Gamma, x : A \vdash_S C : K$ and $\Gamma, x : A \vdash_S K : \mathtt{kind}$. By IH and the fact that $\Gamma, x : A \vdash_S (\lambda x : A.C)\ x = C : K$ we get

$$r(\Gamma), x : r^{\mu}(A) \vdash_T r(C) : r^{C}(K)$$
$$r(\Gamma), x : r^{\mu}(A) \vdash_T r^{(\lambda x:A.C)\ x}(K) = r^{C}(K) : \mathtt{kind}$$

We thus have

$$r(\Gamma), x : r^{\mu}(A) \vdash_T r(C) : r^{(\lambda x:A.C)\ x}(K) \qquad \text{hence}$$
$$r(\Gamma) \vdash_T \lambda x : r^{\mu}(A).r(C) : \Pi x : r^{\mu}(A).r^{(\lambda x:A.C)\ x}(K)$$

as desired.

— I(3): Case $\Gamma \vdash_S \Pi x : A.B : \mathtt{type}$ from the premise $\Gamma, x : A \vdash_S B : \mathtt{type}$. By IH we get

$$r(\Gamma), x : r^{\mu}(A) \vdash_T r(B) : \mu'_1(B) \to \ldots \to \mu'_n(B) \to \mathtt{type}$$

Since $r(\Gamma) \vdash_T \mu'_i(\Pi x : A.B) : \mathtt{type}$, we have

$$r(\Gamma), f : \mu(\Pi x : A.B), x : r^{\mu}(A) \vdash_T r(B)\ (f^1\ x^1)\ \ldots\ (f^n\ x^n) : \mathtt{type}$$

Thus

$$r(\Gamma) \vdash_T \lambda f : \mu(\Pi x : A.B).\Pi x : r^{\mu}(A).r(B)\ (f^1\ x^1)\ \ldots\ (f^n\ x^n) :$$
$$\mu'_1(\Pi x : A.B) \to \ldots \to \mu'_n(\Pi x : A.B) \to \mathtt{type}$$

as desired.

— I(7): Case $\Gamma \vdash_S \Pi x : A.B = \Pi x : A'.B' : \mathtt{type}$ from the premises $\Gamma \vdash_S A = A' : \mathtt{type}$ and $\Gamma, x : A \vdash_S B = B' : \mathtt{type}$. By IH we get

$$r(\Gamma) \vdash_T r(A) = r(A') : \mu'_1(A) \to \ldots \to \mu'_n(A) \to \mathtt{type}$$
$$r(\Gamma), x : r^{\mu}(A) \vdash_T r(B) = r(B') : \mu'_1(B) \to \ldots \to \mu'_n(B) \to \mathtt{type}$$

Now $r(\Gamma) \vdash_T \mu'_i(A) = \mu'_i(A') : \mathtt{type}$; $r(\Gamma) \vdash_T \mu'_i(\Pi x : A.B) = \mu'_i(\Pi x : A'.B') : \mathtt{type}$. Thus

$$r(\Gamma), x : \mu(A) \vdash_T r(A)\ x^1\ \ldots\ x^n = r(A')\ x^1\ \ldots\ x^n : \mathtt{type}$$
$$r(\Gamma), f : \mu(\Pi x : A.B), x : r^{\mu}(A) \vdash_T r(B)\ (f^1\ x^1)\ \ldots\ (f^n\ x^n) =$$
$$r(B')\ (f^1\ x^1)\ \ldots\ (f^n\ x^n) : \mathtt{type}$$

This gives us

$$r(\Gamma) \vdash_T \lambda f : \mu(\Pi x : A.B).\Pi x : r^{\mu}(A).r(B)\ (f^1\ x^1)\ \ldots\ (f^n\ x^n) =$$
$$\lambda f : \mu(\Pi x : A'.B').\Pi x : r^{\mu}(A').r(B')\ (f^1\ x^1)\ \ldots\ (f^n\ x^n) :$$
$$\mu'_1(\Pi x : A.B) \to \ldots \to \mu'_n(\Pi x : A.B) \to \mathtt{type}$$

as desired.

— I(7): Case $\Gamma \vdash_S C = C' : \Pi x : A.K$ from the premise $\Gamma, x : A \vdash_S C\ x = C'\ x : K$. By IH we have

$$r(\Gamma), x : r^\mu(A) \vdash_T r(C)\ x^1\ \ldots\ x^n\ x^* = r(C')\ x^1\ \ldots\ x^n\ x^* : r^{C\ x}(K) \quad \text{hence}$$
$$r(\Gamma) \vdash_T r(C) = r(C') : \Pi x : r^\mu(A).r^{C\ x}(K)$$

as desired.

— I(7): Case $\Gamma \vdash_S (\lambda x : A.C)\ M = C[M/x] : K[M/x]$ from the premises $\Gamma, x : A \vdash_S C : K$, and $\Gamma, x : A \vdash_S K : \texttt{kind}$, and $\Gamma \vdash_S M : A$. By IH and the fact that $\Gamma, x : A \vdash_S (\lambda x : A.C)\ x = C : K$ we get

$$r(\Gamma), x : r^\mu(A) \vdash_T r(C) : r^C(K)$$
$$r(\Gamma), x : r^\mu(A) \vdash_T r^{(\lambda x:A.C)\ x}(K) = r^C(K) : \texttt{kind}$$
$$r(\Gamma) \vdash_T r(M) : r(A)\ \mu(M)$$

Using the fact that $r(\Gamma) \vdash_T \mu'_i(M) : \mu'_i(A)$ thus yields

$$r(\Gamma) \vdash_T (\lambda x : r^\mu(A).r(C))\ r^\mu(M) = r(C)[r^\mu(M)/x^1, \ldots, x^n, x^*] : r^C(K)[r^\mu(M)/x^1, \ldots, x^n, x^*]$$
$$r(\Gamma) \vdash_T r^{(\lambda x:A.C)\ x}(K)[r^\mu(M)/x^1, \ldots, x^n, x^*] = r^C(K)[r^\mu(M)/x^1, \ldots, x^n, x^*] : \texttt{kind}$$

Thus

$$r(\Gamma) \vdash_T (\lambda x : r^\mu(A).r(C))\ r^\mu(M) = r(C)[r^\mu(M)/x^1, \ldots, x^n, x^*] : r^{(\lambda x:A.C)\ x}(K)[r^\mu(M)/x^1, \ldots, x^n, x^*]$$

By the Substitution Lem. 3.10 we have

$$r(C)[r^\mu(M)/x^1, \ldots, x^n, x^*] = r(C[M/x])$$
$$r^{(\lambda x:A.C)\ x}(K)[r^\mu(M)/x^1, \ldots, x^n, x^*] = r^{(\lambda x:A.C)\ M}(K[M/x])$$

thus

$$r(\Gamma) \vdash_T (\lambda x : r^\mu(A).r(C))\ r^\mu(M) = r(C[M/x]) : r^{(\lambda x:A.C)\ M}(K[M/x])$$

as desired.

— For terms:

— I(4): Case $\Gamma \vdash_S c : A$ for $c : A \in S$, from the premise $\vdash_S \Gamma\ \texttt{ctx}$. Analogous to the base case for type families.

— I(4): Case $\Gamma \vdash_S x : A$ for $x : A \in \Gamma$, from the premise $\vdash_S \Gamma\ \texttt{ctx}$. By IH we have $\vdash_T r(\Gamma)\ \texttt{ctx}$. By construction we have $x^* : r(A)\ x^1\ \ldots\ x^n \in r(\Gamma)$. The validity of $r(\Gamma)$ thus gives us $r(\Gamma) \vdash_T x^* : r(A)\ x^1\ \ldots x^n$, i.e., $r(\Gamma) \vdash_T r(x) : r(A)\ \mu(x)$ as desired.

— I(4): Case $\Gamma \vdash_S M\ N : B[N/x]$ from the premises $\Gamma \vdash_S M : \Pi x : A.B$ and $\Gamma, x : A \vdash_S B : \mathtt{type}$ and $\Gamma \vdash_S N : A$. By IH we get

$$r(\Gamma) \vdash_T r(M) : \big(\lambda f : \mu(\Pi x : A.B).\Pi x : r^\mu(A).r(B)\ (f^1\ x^1)\ \ldots\ (f^n\ x^n)\big)\ \mu(M)$$
$$r(\Gamma), x : r^\mu(A) \vdash_T r(B) : \mu'_1(B) \to \ldots \to \mu'_n(B) \to \mathtt{type}$$
$$r(\Gamma) \vdash_T r(N) : r(A)\ \mu(N)$$

Using the fact that $r(\Gamma) \vdash_T \mu'_i(M) : \mu'_i(\Pi x : A.B)$, we thus have

$$r(\Gamma) \vdash_T \big(\lambda f : \mu(\Pi x : A.B).\Pi x : r^\mu(A).r(B)\ (f^1\ x^1)\ \ldots\ (f^n\ x^n)\big)\ \mu(M) = \Pi x : r^\mu(A).r(B)\ (\mu'_1(M)\ x^1)\ \ldots\ (\mu'_n(M)\ x^n) : \mathtt{type}$$

Hence

$$r(\Gamma) \vdash_T r(M) : \Pi x : r^\mu(A).r(B)\ (\mu'_1(M)\ x^1)\ \ldots\ (\mu'_n(M)\ x^n)$$

Using the fact that $r(\Gamma) \vdash_T \mu'_i(N) : \mu'_i(A)$ and the last IH we get

$$r(\Gamma) \vdash_T r(M)\ r^\mu(N) : r(B)[r^\mu(N)/x^1, \ldots, x^n, x^*]\ (\mu'_1(M)\ \mu'_1(N))\ \ldots\ (\mu'_n(M)\ \mu'_n(N))$$

Now by the Substitution Lem. 3.10, we have

$$r(B)[r^\mu(N)/x^1, \ldots, x^n, x^*] = r(B[N/x])$$

hence

$$r(\Gamma) \vdash_T r(M)\ r^\mu(N) : r(B[N/x])\ \mu(M\ N)$$

as desired.

— I(8): Case $\Gamma \vdash_S M\ N = M'\ N' : B[N/x]$ from the premises $\Gamma \vdash_S M = M' : \Pi x : A.B$ and $\Gamma \vdash_S N = N' : A$. Very similar to the previous case, with equality in place of the typing judgement.

— I(4): Case $\Gamma \vdash_S \lambda x : A.M : \Pi x : A.B$ from the premises $\Gamma, x : A \vdash_S M : B$ and $\Gamma, x : A \vdash_S B : \mathtt{type}$. By IH we get

$$r(\Gamma), x : r^\mu(A) \vdash_T r(M) : r(B)\ \mu(M)$$
$$r(\Gamma), x : r^\mu(A) \vdash_T r(B) : \mu'_1(B) \to \ldots \to \mu'_n(B) \to \mathtt{type}$$

Thus we have

$$r(\Gamma) \vdash_T \lambda x : r^\mu(A).r(M) : \Pi x : r^\mu(A).r(B)\ \mu(M)$$

Furthermore, using the fact that $r(\Gamma) \vdash_T \mu'_i(\lambda x : A.M) : \mu'_i(\Pi x : A.B)$ we get

$$r(\Gamma) \vdash_T \big(\lambda f : \mu(\Pi x : A.B).\Pi x : r^\mu(A).r(B)\ (f^1\ x^1)\ \ldots\ (f^n\ x^n)\big)\ \mu(\lambda x : A.M) = \Pi x : r^\mu(A).r(B)\ \mu(M) : \mathtt{type}$$

Thus

$$r(\Gamma) \vdash_T \lambda x : r^\mu(A).r(M) : \big(\lambda f : \mu(\Pi x : A.B).\Pi x : r^\mu(A).r(B)\ (f^1\ x^1)\ \ldots\ (f^n\ x^n)\big)\ \mu(\lambda x : A.M)$$

as desired.

— I(8): Case $\Gamma \vdash_S M = M' : \Pi x : A.B$ from the premises $\Gamma, x : A \vdash_S M\ x = M'\ x : B$ and $\Gamma, x : A \vdash_S B : \mathtt{type}$. By IH we get

$$r(\Gamma), x : r^\mu(A) \vdash_T r(M)\ x^1\ \ldots\ x^n\ x^* = r(M')\ x^1\ \ldots\ x^n\ x^* : r(B)\ (\mu'_1(M)\ x^1)\ \ldots\ (\mu'_n(M)\ x^n)$$
$$r(\Gamma), x : r^\mu(A) \vdash_T r(B) : \mu'_1(B) \to \ldots \to \mu'_n(B) \to \mathtt{type}$$

Thus we have

$$r(\Gamma) \vdash_T r(M) = r(M') : \Pi x : r^\mu(A).r(B)\ (\mu'_1(M)\ x^1)\ \ldots\ (\mu'_n(M)\ x^n)$$

Furthermore, using the fact that $r(\Gamma) \vdash_T \mu'_i(M) : \mu'_i(\Pi x : A.B)$, we have

$$r(\Gamma) \vdash_T \left(\lambda f : \mu(\Pi x : A.B).\Pi x : r^\mu(A).r(B)\ (f^1\ x^1)\ \ldots\ (f^n\ x^n)\right)\ \mu(M) = \Pi x : r^\mu(A).r(B)\ (\mu'_1(M)\ x^1)\ \ldots\ (\mu'_n(M)\ x^n) : \mathtt{type}$$

Hence

$$r(\Gamma) \vdash_T r(M) = r(M') : \left(\lambda f : \mu(\Pi x : A.B).\Pi x : r^\mu(A).r(B)\ (f^1\ x^1)\ \ldots\ (f^n\ x^n)\right)\ \mu(M)$$

as desired.

— I(8): Case $\Gamma \vdash_S (\lambda x : A.M)\ N = M[N/x] : B[N/x]$ from the premises $\Gamma, x : A \vdash_S M : B$, and $\Gamma, x : A \vdash_S B : \mathtt{type}$, and $\Gamma \vdash_S N : A$. By IH we get

$$r(\Gamma), x : r^\mu(A) \vdash_T r(M) : r(B)\ \mu(M)$$
$$r(\Gamma), x : r^\mu(A) \vdash_T r(B) : \mu'_1(B) \to \ldots \to \mu'_n(B) \to \mathtt{type}$$
$$r(\Gamma) \vdash_T r(N) : r(A)\ \mu(N)$$

Using the fact that $r(\Gamma) \vdash_T \mu'_i(N) : \mu'_i(A)$ thus yields

$$r(\Gamma) \vdash_T (\lambda x : r^\mu(A).r(M))\ r^\mu(N) = r(M)[r^\mu(N)/x^1, \ldots, x^n, x^*] : r(B)[r^\mu(N)/x^1, \ldots, x^n, x^*]\ \mu'_1(M)[\mu'_1(N)/x^1]\ \ldots\ \mu'_n(M)[\mu'_n(N)/x^n]$$

Since $r(\Gamma) \vdash_T \mu'_i((\lambda x : A.M)\ N) = \mu'_i(M)[\mu'_i(N)/x^i] : \mu'_i(B)[\mu'_i(N)/x^i]$, we get

$$r(\Gamma) \vdash_T r(B)[r^\mu(N)/x^1, \ldots, x^n, x^*]\ \mu'_1(M)[\mu'_1(N)/x^1]\ \ldots\ \mu'_n(M)[\mu'_n(N)/x^n] = r(B)[r^\mu(N)/x^1, \ldots, x^n, x^*]\ \mu((\lambda x : A.M)\ N) : \mathtt{type}$$

Hence

$$r(\Gamma) \vdash_T (\lambda x : r^\mu(A).r(M))\ r^\mu(N) = r(M)[r^\mu(N)/x^1, \ldots, x^n, x^*] : r(B)[r^\mu(N)/x^1, \ldots, x^n, x^*]\ \mu((\lambda x : A.M)\ N)$$

By the Substitution Lem. 3.10 we have

$$r(M)[r^\mu(N)/x^1, \ldots, x^n, x^*] = r(M[N/x])$$
$$r(B)[r^\mu(N)/x^1, \ldots, x^n, x^*] = r(B[N/x])$$

thus

$$r(\Gamma) \vdash_T (\lambda x : r^\mu(A).r(M))\ r^\mu(N) = r(M[N/x]) : r(B[N/x])\ \mu((\lambda x : A.M)\ N)$$

as desired.

— For equality judgements: Straightforward using the modified rules.

This finishes the proof. □

4. APPLICATIONS

In this section we show how a number of known as well as new applications of logical relations can be formalized in DTT. All concrete examples have been formalized and mechanically verified by our implementation (see Sect. 5) and are available online at [Rabe and Sojakova 2012].

4.1. Meta-Theorems

Ex. 3.1 already gave an example of representing an inductively proved meta-theorem about formal systems as a logical relation. As another example, we show that in propositional logic the law of excluded middle can be proved if it is assumed for all atomic formulas:

Example 4.1. The following signature gives (a fragment of) a well-known (see, e.g., [Harper et al. 1993]) encoding of intuitionistic propositional logic with a natural deduction calculus:

$$
\begin{array}{lcll}
\%\texttt{sig}\ PL = \{ & & & \\
\quad o & : & \texttt{type} & \\
\quad pf & : & o \to \texttt{type} & \\
\quad \vee & : & o \to o \to o & \%\texttt{infix}\ \vee \\
\quad \neg & : & o \to o & \\
\quad \vee_{IL} & : & \Pi A.\Pi B.\ pf\, A \to pf\,(A \vee B) & \\
\quad \vee_{IR} & : & \Pi A.\Pi B.\ pf\, B \to pf\,(A \vee B) & \\
\quad \vee_{E} & : & \Pi A.\Pi B.\Pi C.\ pf\,(A \vee B) \to (pf\, A \to pf\, C) \to (pf\, B \to pf\, C) \to pf\, C & \\
\quad \neg_{I} & : & \Pi A.\ (pf\, A \to \lightning) \to pf\,(\neg A) & \\
\quad \neg_{E} & : & \Pi A.\ pf\, A \to pf\,(\neg A) \to \lightning & \\
\} & & &
\end{array}
$$

We use the symbol $\lightning$ to abbreviate the type $\Pi A : o.\ pf\, A$, which encodes inconsistency. We now give a unary logical relation *Tnd* (for tertium non datur) on the identity morphism $id_{PL} : PL \to PL$ to prove that the type $pf\,(A \vee \neg A)$ is inhabited for every formula A.

$$
\begin{array}{lcl}
\%\texttt{rel}\ Tnd : id_{PL} = \{ & & \\
\quad o & := & \lambda A : o.\ pf\,(A \vee \neg A) \\
\quad \vee & := & \lambda A : o.\ \lambda A^* : pf\,(A \vee \neg A).\ \lambda B : o.\ \lambda B^* : pf\,(B \vee \neg B).\ \vee_E\ A^* \\
 & & \big(\lambda P : pf\, A.\ \vee_{IL}\ (\vee_{IL}\ P)\big) \\
 & & \big(\lambda P : pf\,(\neg A).\ \vee_E\ B^* \\
 & & \quad \big(\lambda Q : pf\, B.\ \vee_{IL}\ (\vee_{IR}\ Q)\big) \\
 & & \quad \big(\lambda Q : pf\,(\neg B).\ \vee_{IR}\ (\neg_I\ (\lambda R : pf\,(A \vee B).\ \lambda C.\ \vee_E\ R \\
 & & \qquad (\lambda P' : pf\, A.\ \neg_E\ P'\ P\ C) \\
 & & \qquad (\lambda Q' : pf\, B.\ \neg_E\ Q'\ Q\ C))))\big) \\
\quad \neg & := & \lambda A : o.\ \lambda A^* : pf\,(A \vee \neg A).\ \vee_E\ A^* \\
 & & \big(\lambda P : pf\, A.\ \vee_{IR}\ (\neg_I\ (\lambda P' : pf\,(\neg A).\ \lambda C.\ \neg_E\ P\ P'\ C))\big) \\
 & & \big(\lambda P : pf\,(\neg A).\ \vee_{IL}\ P\big) \\
\quad \vdots & & \\
\} & &
\end{array}
$$

All cases relating to the type family *pf* are trivial because we use a trivially satisfied predicate.

Applying the Basic Lemma to this logical relation yields that excluded middle is an admissible rule – in the special case without atomic formulas. To handle the presence of atomic formulas, we add constants $p : o$ to PL. Consequently, we have to add to Tnd the corresponding cases $p := P$ for proofs $P : pf\,(p \vee \neg p)$. Thus, it follows that excluded middle is admissible if it is admissible for all atoms.

4.2. Congruence Relations

In a single-typed language such as that of universal algebra, congruence relations are binary relations on the type that are closed under all operations on that type. More generally, we can consider congruence relations for arbitrary formal languages, represented by a signature S. In that case, they are type-indexed families of binary relations that are closed under all operations. This closure property is exactly the Basic Lemma of a binary logical relation on id_S:

Definition 4.2 (*Congruence*). A *logical congruence* on a signature S is a binary logical relation on the identity morphism id_S.

Example 4.3. First we extend PL from Ex. 4.1 sufficiently to make the rules for the equivalence connective $\Leftrightarrow$ provable (e.g., by adding conjunction and implication and then using the usual definition of $\Leftrightarrow$):

$$\begin{array}{lcll} \Leftrightarrow & : & o \to o \to o & \%\texttt{infix}\ \Leftrightarrow \\ \Leftrightarrow_I & : & \Pi A.\Pi B.\ (pf\,A \to pf\,B) \to (pf\,B \to pf\,A) \to pf(A \Leftrightarrow B) & \\ \Leftrightarrow_{EL} & : & \Pi A.\Pi B.\ pf\,(A \Leftrightarrow B) \to pf\,A \to pf\,B & \\ \Leftrightarrow_{ER} & : & \Pi A.\Pi B.\ pf\,(A \Leftrightarrow B) \to pf\,B \to pf\,A & \end{array}$$

Then we further extend this to an encoding FOL of first-order logic; we do not need the details of FOL here and will only assume that it adds the declarations

$$\begin{array}{lcll} i & : & \texttt{type} & \\ \doteq & : & i \to i \to o & \%\texttt{infix}\ \doteq \\ \forall & : & (i \to o) \to o & \\ \exists & : & (i \to o) \to o & \end{array}$$

where i is the type of terms.

Then we can give a logical congruence on FOL by

$$\begin{array}{l} \%\texttt{rel}\ Cong : id_{FOL} \times id_{FOL}\ = \{ \\ \quad i \quad := \quad \lambda x : i.\ \lambda y : i.\ pf(x \doteq y) \\ \quad o \quad := \quad \lambda A : o.\ \lambda B : o.\ pf(A \Leftrightarrow B) \\ \quad pf \quad := \quad \lambda A : o.\ \lambda B : o.\ \lambda\_ : pf\,(A \Leftrightarrow B).\ \lambda P : pf\,A.\ \lambda Q : pf\,B.\ unit \\ \quad \neg \quad := \quad \lambda A : o.\ \lambda B : o.\ \lambda r : pf\,(A \Leftrightarrow B).\ \Leftrightarrow_I \\ \qquad\qquad\qquad (\lambda p : pf(\neg A).\ \neg_I \lambda q : pf\,A'.\ \neg_E\ (\Leftrightarrow_{ER}\ r\ q)\ p) \\ \qquad\qquad\qquad (\lambda p : pf(\neg A').\ \neg_I \lambda q : pf\,A.\ \neg_E\ (\Leftrightarrow_{EL}\ r\ q)\ p) \\ \quad \vdots \\ \} \end{array}$$

Again, this uses a trivial judgment for pf, which amounts to a proof irrelevance condition: all proofs of equivalent formulas are related by $Cong$.

Filling in all the cases of this logical relation amounts to proving that the connectives and quantifiers of FOL preserve equality of terms and equivalence of formulas. We only give the case for $\neg$ as an example.

A special case of logical congruences arises when we represent the extensional equality of foundational languages such as set theory:

Example 4.4 (*Extensional Equality*). The signature *ZFC* sketches the encoding of first-order ZFC set theory used in [Iancu and Rabe 2011]. It renames the symbols o, i, pf of *FOL* to *prop*, *set*, *true* and adds the usual $\in$-predicate and a definite description operator δ:

$$
\begin{array}{l}
\%\texttt{sig}\ \mathit{ZFC} = \{ \\
\quad \mathit{set} : \texttt{type} \\
\quad \mathit{prop} : \texttt{type} \\
\quad \mathit{true} : \mathit{prop} \to \texttt{type} \\
\quad \doteq : \mathit{set} \to \mathit{set} \to \mathit{prop} \qquad \%\texttt{infix}\ \doteq \\
\quad \vdots \\
\quad \in : \mathit{set} \to \mathit{set} \to \mathit{prop} \qquad \%\texttt{infix}\ \in \\
\quad \delta : \Pi F : \mathit{set} \to \mathit{prop}.\, \mathit{pf}\,(\exists^! \lambda x : \mathit{set}.\, F\, x) \to \mathit{set} \\
\}
\end{array}
$$

The analogue of the logical relation *Cong* formalizes extensional equality as a congruence on *ZFC*.

$\delta\, F\, P$ returns the unique set that satisfies the unary predicate F (where $\exists^!$ is the easily-definable quantifier of unique existence). Here P acts as a guard that ensures that the term $\delta\, F\, P$ is only well-formed if F is indeed uniquely satisfiable. Such *guarded operators* are useful in a variety of situations, e.g., to avoid division by 0.

The disadvantage of guarded operators is that they permit proofs to occur in other expressions such as sets and propositions. The above congruence constitutes a formalized and verified proof that guarded operators do not affect the adequacy of the encoding. To see that, consider a guarded set, i.e., a term $\vdash_{ZFC} s : \mathit{true}\, G \to \mathit{set}$, and two sets $s\, P$ and $s\, P'$ that differ only in using two different guards P and P' proving G. As all proofs of the same proposition are related, applying the Basic Lemma to s yields $\mathit{true}\,(s\, P) \doteq (s\, P')$. Thus, using different guards never alters the semantics

Note that Def. 4.2 does not require that congruence relations are equivalence relations. It is straightforward to refine Def. 4.2 to require equivalence. In that case, one must avoid the following following deceptively natural inductive argument: If a logical relation has property P at all atomic types, then it has property P at all types. This is true if P is symmetry but does not hold if P is reflexivity or transitivity.

4.3. Logical Relations for Object Languages

One of the most desirable applications is to represent existing logical relation arguments. More specifically, if we use DTT as a logical framework in which object languages are represented, then we would like to represent logical relations of the object language as logical relations of the framework.

Encoding logical relation arguments in Twelf, however, has shown to be very difficult due to the parametric nature of DTT's function space. [Schürmann and Sarnat 2008] show how in certain cases, one can circumvent this problem by introducing an additional layer of abstraction, which they term the *assertion logic*. Intuitively, the assertion logic represents explicitly as a DTT signature the implicitly assumed meta-language in which the logical relation is stated. Thus, a given logical relation for the object language can be encoded in the flexible assertion logic, rather than by using the fixed DTT constructors directly.

Our approach yields a general framework for encodings along these lines. We represent the object language l as a DTT-signature L and the meta language m as a DTT-signature M. Moreover, we assume a morphism $\mu : L \to M$ that interprets the object language in the

meta language. Then we represent (n-ary) l-logical relations as DTT-logical relations on (n copies of) the morphism μ.

For simplicity, we will constrain attention to the special case where $n = 1$ and μ is an inclusion morphism $i : L \hookrightarrow M$. In this case, M corresponds to the assertion logic used in [Schürmann and Sarnat 2008]. The prototypical example arises when l is simple type theory and L is its representation in either Church or Curry-style as described in Ex. 3.1:

Example 4.5 (*Logical Relations for Simple Type Theory*). Let l be simple type theory represented as $L = \mathit{Church}$. Let $M = \mathit{Assertion}$ be an extension of *Church* with a first-order assertion logic such that *Assertion* contains in particular the declarations

$$\begin{array}{lcll} o & : & \texttt{type} & \\ \mathit{pf} & : & o \to \texttt{type} & \\ \supset & : & o \to o \to o & \%\texttt{infix}\ \supset \\ \forall & : & \Pi A : \mathit{tp}.\, (\mathit{tm}\, A \to o) \to o & \\ \beta & : & \Pi A, B : \mathit{tp}.\, \Pi F : \mathit{tm}\, A \to \mathit{tm}\, B . \Pi X : \mathit{tm}\, A. & \\ & & \quad \mathit{pf}(\mathit{app}\,(\mathit{lam}\, F)\, X \doteq (F\, X)) & \end{array}$$

An n-ary logical relation for simple type theory is represented as an n-ary logical relation $r : i \times \ldots \times i : \mathit{Church} \to \mathit{Assertion}$ of DTT, where i is the inclusion morphism $\mathit{Church} \hookrightarrow \mathit{Assertion}$. In the unary case, r is given by

$$\begin{array}{l} \%\texttt{rel}\ \mathit{Basic} : i : \mathit{Church} \to \mathit{Assertion}\ = \{ \\ \quad \begin{array}{lcl} \mathit{tp} & := & \lambda A : \mathit{tp}.\, (\mathit{tm}\, A \to o) \\ \mathit{tm} & := & \lambda A : \mathit{tp}.\, \lambda p : \mathit{tm}\, A \to o.\, \lambda x : \mathit{tm}\, A.\, \mathit{proof}\,(p\, x) \\ \Rightarrow & := & \lambda A : \mathit{tp}.\, \lambda p : \mathit{tm}\, A \to o.\, \lambda B : \mathit{tp}.\, \lambda q : \mathit{tm}\, B \to o. \\ & & \quad \lambda f : \mathit{tm}(A \Rightarrow B).\, \forall (\lambda x : \mathit{tm}\, A.\, (p\, x)\ \supset\ (q\,(f\, x))) \end{array} \\ \quad \vdots \\ \} \end{array}$$

The proof terms for *lam* and *app* are straightforward and prove the type preservation property of the Basic Lemma for simple type theory.

We represent concrete signatures of simple type theory as DTT-signatures S that extend *Church* with declarations of the form $a : \mathit{tp}$ or $c : \mathit{tm}\, A$. Then we represent concrete unary logical relations of simple type theory as logical relations r that extend *Basic*. r maps S-constants $a : \mathit{tp}$ to unary predicates $r(a) : \mathit{tm}\, a \to o$ and S-constants $c : \mathit{tm}\, A$ to proofs $r(c) : \mathit{pf}(r(a)\, c)$. Now the Basic Lemma for our framework language induces the Basic Lemma for the object language in the sense that for all S-types $A : \mathit{tp}$ and S-terms $M : \mathit{tm}\, A$, the type $r(A)\, M$ is inhabited.

Here we used an assertion logic just strong enough to state the relation *Basic*. Alternatively, we can use any stronger language M. Moreover, as we will show in Thm. 5.2, if the fact that M is at least as strong as *Assertion* is witnessed by a morphism $\mathit{sem} : \mathit{Assertion} \to M$, we immediately obtain a logical relation with codomain M by composing *Basic* and *sem*. Of particular interest is the case where M represents set theory, e.g., $M = \mathit{ZFC}$ from Ex. 4.4. In that case, morphisms $\mathit{Church} \to \mathit{ZFC}$ correspond to set assignments and logical relations on them to relation assignments in the sense of [Reynolds 1974].

In Ex. 4.5, we had to give the logical relation *Basic* explicitly. Thus, while we inherited the Basic Lemma from DTT, we had to provide the expressions in *Basic* manually. Therefore, our approach of inheriting DTT-logical relations for a specific object language l requires the initial investment of giving the relation *Basic*. However, this investment is acceptable for two reasons.

Firstly, *Basic* only has to be given once and can be reused whenever a concrete logical relation is formalized. This is particularly easy by using a module system as we present in Sect. 5.

Secondly, giving *Basic* explicitly also gives users the flexibility to use different assertion logics and different definitions of logical relations. Indeed, often any inductively defined type-indexed family of relations that holds for all terms is called a logical relation even if it does not define functions to be related if they map related arguments to related outputs. A typical example is the following:

Example 4.6 (*Termination of* β*-reduction*). We use the same signatures as in Ex. 4.5 except that we state β-conversion in terms of a new binary predicate symbol $\rightsquigarrow$ instead of using the equality predicate $\doteq$ built into the logic *Assertion*. We think of $\rightsquigarrow$ as a directed reduction relation. Moreover, we use a unary predicate *val* on terms that distinguishes the values. Instead, of using the logical relation *Basic*, we give a new logical relation *Termination* that differs from *Basic* in two key respects.

Firstly, *Termination* maps *Church*-types $a : tp$ not simply to unary predicates on $tm\,A$, but to unary predicates on $tm\,A$ that are closed under expansion. Such predicates can be easily encoded using the dependent sum type:

$$\begin{aligned} \mathit{Termination}(tp) = \Pi A.\ \Sigma P : tm \to o.\Pi M.\Pi N.& \\ \mathit{proof}\,(M \rightsquigarrow N) \to \mathit{proof}\,(P\ N) \to \mathit{proof}\,(P\ M)& \end{aligned}$$

In our variant of DTT, we do not have dependent sum types; therefore, we introduce a new type family that encodes this particular instance of dependent sums. For simplicity, we will gloss over this in the sequel and directly use a Σ-type with constructor $(-,-)$ and projections $\pi_1(-)$ and $\pi_2(-)$. The full encoding in plain DTT is available online.

Analogously to the well-known pen-and-paper proof, the relation at a base type requires that terms reduce to a value and the relation at a function type requires that terms evaluate to a function which preserves the relation:

$$\begin{aligned} \mathit{Termination}(i) &= \big(\lambda M : tm\ i.\ \exists(\lambda V.\ M \rightsquigarrow V \wedge \mathit{val}\,V), \dots\big) \\ \mathit{Termination}(\Rightarrow) &= \lambda A.\lambda P_A.\lambda B.\lambda P_B.\ \big(\lambda F : tm\ (A \Rightarrow B).\ \exists(\lambda G.\ F \rightsquigarrow G \wedge \mathit{val}\,G \wedge \\ &\qquad \forall(\lambda N.\ (\pi_1(P_A)\ N) \supset (\pi_2(P_B)\ (\mathit{app}\,G\ N)))), \dots\big) \end{aligned}$$

Here we have omitted the proofs of closure under expansion for clarity. The case for *tm* now states that each term has to satisfy the relation at its type:

$$\mathit{Termination}(tm) = \lambda A.\lambda P_A.\lambda M : tm\ A.\ \mathit{proof}\,(\pi_1(P_A)\ M)$$

Finally, the cases for *lam* and *app* prove that these constructors preserve the relations. The full encoding can be found online.

An analogous formalization is possible for Curry-version. It is also available online.

For a more complex object language, we consider the notion of logical relations for reflective pure type systems from [Bernardy et al. 2010]:

Example 4.7 (*Reflective Pure Type Systems*). To represent pure type systems (PTS) in DTT, we use the following base signature:

$$
\begin{array}{lcll}
\%\texttt{sig}\ \mathit{PTS} & = & \{ & \\
\quad \mathit{sort} & : & \texttt{type} & \\
\quad \mathit{tm} & : & \texttt{type} & \\
\quad \prime & : & \mathit{sort} \to \mathit{tm} & \\
\quad \forall & : & \mathit{tm} \to (\mathit{tm} \to \mathit{tm}) \to \mathit{tm} & \\
\quad \vdots & & & \\
\quad \mathit{rule} & : & \mathit{sort} \to \mathit{sort} \to \mathit{sort} \to \texttt{type} & \\
\quad \# & : & \mathit{tm} \to \mathit{tm} \to \texttt{type} & \%\texttt{infix}\ \# \\
\quad \#\forall & : & \Pi S_1, S_2, S_3 : \mathit{sort}.\, \mathit{rule}\, S_1\, S_2\, S_3 \to \Pi A : \mathit{tm}.\, \Pi B : \mathit{tm} \to \mathit{tm}. & \\
 & & \quad A \,\#\prime S_1 \to (\Pi x : \mathit{tm}.\, x \,\# A \to (B\, x) \,\#\prime S_2) \to \forall\, A\, B \,\#\prime S_3 & \\
\quad \vdots & & & \\
\} & & &
\end{array}
$$

Here we have omitted the declarations and typing rules for abstraction *lam* and application *app*. Now a specific PTS arises by adding sorts $s : \mathit{sort}$, axioms of type $\prime s_1 \#\prime s_2$, and rules of type $\mathit{rule}\, s_1\, s_2\, s_3$ to *PTS*.

A reflective PTS arises by extending *PTS* with meta-axioms that express the closure properties of the sorts, axioms, and rules. These include

$$
\begin{array}{lcl}
\%\texttt{sig}\ \mathit{Reflective} & = & \{ \\
\quad \vdots & & \\
\quad \sim & : & \mathit{sort} \to \mathit{sort} \\
\quad r & : & \Pi S_1, S_2, S_3 : \mathit{sort}.\, \mathit{rule}\, S_1\, S_2\, S_3 \to \mathit{rule}\, (\sim S_1)\, (\sim S_2)\, (\sim S_3) \\
\quad \vdots & & \\
\} & &
\end{array}
$$

Then logical relations of PTS can be represented as DTT-logical relations on the inclusion morphism $i : \mathit{PTS} \hookrightarrow \mathit{Reflective}$. Again we only consider the unary case; n-ary relations are treated accordingly. We present the two key cases and refer to [Rabe and Sojakova 2012] for the details:

$$
\begin{array}{lcl}
\%\texttt{rel}\ \mathit{Parametricity} : i : \mathit{PTS} \to \mathit{Reflective}\ = \{ & & \\
\quad \mathit{tm} & := & \lambda\_.\, \mathit{tm} \\
\quad \vdots & & \\
\quad \# & := & \lambda A : \mathit{tm}.\, \lambda A^* : \mathit{tm}.\, \lambda B : \mathit{tm}.\, \lambda B^* : \mathit{tm}.\, \lambda\_.\, A^* \,\#\, \mathit{app}\, B^*\, A \\
\quad \vdots & & \\
\} & &
\end{array}
$$

The case for *tm* does not make a judgment about terms $M : \mathit{tm}$; instead, it simply mandates that *Parametricity* translates M to some other expression. Then the cases for the constructors of *tm* formalize the inductive translation of expressions (Def. 4 in [Bernardy et al. 2010]).

The case for # states the type preservation property: the translation A^* of A must be typed by the application of the translation B^* of B to A (Thm. 1 in [Bernardy et al. 2010]). Then the cases for the constructors of # formalize the proof of this property (sketched in Fig. 6 of [Bernardy et al. 2010]).

Due to the generality of pure type systems, Ex. 4.7 yields a mechanically verified proof of the Basic Lemma for a wide variety of type theories.

4.4. Model Theoretical Properties

In [Rabe 2012], we showed that we can use DTT morphisms to represent models. The main idea is to use a signature D that represents the semantic universe and to use morphisms $\mu : L \to D$ to represent models of L in D. For simplicity, we will restrict attention to the special case where $L = FOL$ and $D = ZFC$ are as in Ex. 4.3 and 4.4.

In that case, we encode set-theoretical models of first-order logic as morphisms $\mu : S \to ZFC$ where S extends FOL with declarations of function symbols $f : i \to \ldots \to i \to i$ and predicate symbols $p : i \to \ldots \to i \to o$. In particular, $\mu(i)$ is the universe of the model; $\mu(o)$ is the set $\{0, 1\}$ of booleans, and $\mu(pf)\, x$ is the type $true\,(x \doteq 1)$. Morphism application $\mu(-)$ encodes the interpretation function that maps expressions to their denotation in ZFC; and μ satisfies the formula $\vdash_{FOL} F : o$ if there is a term $\vdash_{ZFC} p : \mu(pf\, F)$, i.e., if $\mu(F)$ is extensionally equal to 1. All details can be found in [Horozal and Rabe 2011].

We can now use logical relations to express concisely and uniformly various model theoretical closure and preservation properties.

Example 4.8 (*Submodels*). A submodel of μ is given by a subset of the universe that is closed under all operations.

Such subsets correspond to the unary logical relations $r : \mu$. In particular, $r(i) : \mu(i) \to \texttt{type}$ is the predicate identifying the subset of the universe, and $r(o)$ and $r(pf)$ are trivial judgments. Then for every function symbol f, the term $r(f)$ proves that $r(i)$ is closed under $\mu(f)$.

Example 4.9 (*Quotient Models*). A quotient model of μ is given by an equivalence relation on the universe such that each $\mu(p)$ yields equal truth values for related argument tuples and each $\mu(f)$ respects the equivalence.

Such relations correspond to the binary logical relations $r : \mu \times \mu$, where $r(i) : \mu(i) \to \mu(i) \to \texttt{type}$ is an equivalence relation, $r(o)\, x\, y$ is given by extensional equality $true\,(x \doteq y)$, and $r(pf)$ is a trivial judgment. Then for every function symbol f, the term $r(f)$ proves the congruence property of $r(i)$ with respect to $\mu(f)$. And for every predicate symbol p, the term $r(p)$ proves the respective condition on predicate symbols.

Note that r is not necessarily an equivalence relation at higher types. However, that is acceptable when working with first-order logic.

Example 4.10 (*Algebra Homomorphisms*). Algebraic logic arises from FOL by dropping all connectives and quantifiers except for equality; in that case, we call the models algebras. An *algebra homomorphism* $h : \mu \to \mu'$ is given by a function from the μ-universe to the μ'-universe that commutes with all function symbols and preserves all predicate symbols.

Such functions correspond to the binary logical relations $r : \mu \times \mu'$, where $r(i) : \mu(i) \to \mu(i) \to \texttt{type}$ is a function, $r(o)\, x\, y$ is given by $true\,(x \leq y)$ (where $\leq$ is the usual ordering on the booleans); and $r(pf)$ is a trivial judgment. Then for every function symbol f, the term $r(f)$ proves the commutation property for $r(i)$ and $\mu(f)$. And for every predicate symbol p, the term $r(p)$ proves the preservation of predicates.

The above encodings of submodels and quotient models easily generalize to other logics than FOL. In particular, submodels and quotient models of typed languages are given by type-indexed families of relations, corresponding exactly to logical relations.

Encoding of homomorphisms, however, do not generalize to higher-order logics – not due to a limitation of our approach but due to the inherent difficulty of relating models of higher-order logics. In our terminology, this difficulty can be traced to the following observation: A logical relation that is functional at all atomic types is not necessarily functional at all types. A viable alternative is to drop the functionality requirement altogether and define homomorphisms as typed-indexed families of relations (rather than type-indexed families

of functions) as done in [Reynolds 1974]. Such definitions of homomorphisms correspond exactly to logical relations.

4.5. Observational Equivalence

A well-known application of logical relations is to establish the observational equivalence of two interpretations. By using morphisms to represent interpretations syntactically, we can use our logical relations to represents observational equivalence proofs.

Definition 4.11 (*Observational Equivalence*). Consider a signature S with a predicate $\vdash_S j : A \to \mathtt{type}$. We say that a morphism $\mu : S \to T$ *realizes* a term $\vdash_S M : A$ at j if there is a term $\vdash_T p : \mu(j\, M)$.

We call two morphisms $\mu_1 : S \to T$ and $\mu_2 : S \to T$ *observationally equivalent* at j if they realize the same terms at j.

It is straightforward to generalize Def. 4.11 to the case of n-ary judgments.

Example 4.12. A *FOL*-model μ satisfies the formula $\vdash_{FOL} F : o$ if it realizes F at *pf*. Thus, two models are observationally equivalent at *pf* iff they satisfy the same formulas (which is usually called elementary equivalence in the context of first-order logic).

Example 4.13. We can represent programs as morphisms in the same way as we have represented models as morphisms above. Let S be a software specification and T an implementation language; then implementations of S in T can be represented as morphisms $\mu : S \to T$. Often T provides definitions for all identifiers declared in S so that μ becomes an inclusion.

Now consider a type A that represents observable behavior of the software and a judgment j that describes which behavior occurs. For example, in the binary case, j could be the input-output relation. Then Def. 4.11 captures the intuition of observational equivalence of implementations.

To state the connection between logical relations and observational equivalence, we use one auxiliary definition:

Definition 4.14. Assume a signature S. An *equivalence type* of two types $\Gamma \vdash_S A_1 : \mathtt{type}$ and $\Gamma \vdash_S A_2 : \mathtt{type}$ is a type $\Gamma \vdash_S E : \mathtt{type}$ such that there are terms $\Gamma \vdash_S e : (A_1 \to A_2) \to (A_2 \to A_1) \to E$ and $\Gamma \vdash_S e_1 : E \to A_1 \to A_2$ and $\Gamma \vdash_S e_2 : E \to A_2 \to A_1$.

Clearly, this definition is redundant in an extension of DTT with product types where equivalence types always exist, namely $(A_1 \to A_2) \times (A_2 \to A_1)$. But the assumption of product types is stronger than necessary and we often find equivalence types in special cases. For example, if we add an equivalence connective $\Leftrightarrow$ with appropriate proof rules to Ex. 4.4, then *true* $(F_1 \Leftrightarrow F_2)$ is an equivalence type of *true* F_1 and *true* F_2.

Theorem 4.15. *In the situation of Def. 4.11, assume there exists a logical relation* $r : \mu_1 \times \mu_2 : S \to T$ *and a term* t *such that*

$$x^1 : \mu_1(A),\, x^2 : \mu_2(A),\, x^* : r(A)\; x^1\; x^2 \vdash_T t : E$$

where E *is an equivalence type of* $\mu_1(j)\; x^1$ *and* $\mu_2(j)\; x^2$ *under the context above. Then* μ_1 *and* μ_2 *are observationally equivalent at* j.

Proof. Since the equivalence type E is inhabited, so are the types $\mu_1(j)\; x^1 \to \mu_2(j)\; x^2$ and $\mu_2(j)\; x^2 \to \mu_1(j)\; x^1$. Now consider a term $\cdot \vdash_S M : A$. By the Basic Lemma 3.9 we have $\cdot \vdash_T r(M) : r(A)\; \mu_1(M)\; \mu_2(M)$. Thus the types

$$(\mu_1(j)\; x^1 \to \mu_2(j)\; x^2)[\mu_1(M)/x^1, \mu_2(M)/x^2, r(M)/x^*]$$
$$(\mu_2(j)\; x^2 \to \mu_1(j)\; x^1)[\mu_1(M)/x^1, \mu_2(M)/x^2, r(M)/x^*]$$

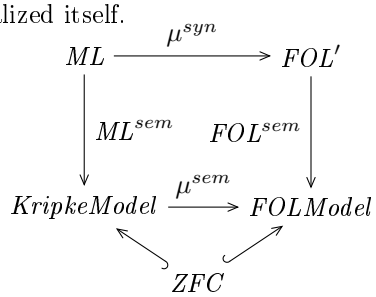
are both inhabited, i.e., $\mu_1(j)\ \mu_1(M) \to \mu_2(j)\ \mu_2(M)$ and $\mu_2(j)\ \mu_2(M) \to \mu_1(j)\ \mu_1(M)$ are both inhabited. In other words, $\mu_1(j)\ \mu_1(M)$ is inhabited iff $\mu_2(j)\ \mu_2(M)$ is, i.e., μ_1 realizes M under j iff μ_2 does. Since M was arbitrary, this shows μ_1 and μ_2 are observationally equivalent at j. □

In our experience, it is usually easy to give the term $t : E$ of Thm. 4.15. And giving the logical relation r corresponds case-by-case to the informal proof of observational equivalence. An example application is given in Sect. 4.16.

4.6. Verifying Logic Translations

Logic translations from a logic L to a logic L' are commonly used to justify borrowing an automated theorem prover for a logic L' to reason about a logic L. There are generally two ways to verify the soundness of borrowing. Proof theoretical borrowing (e.g., as in [Meng and Paulson 2008]) uses the found L'-proofs to construct explicit L-proofs and verifies them. Model theoretical borrowing as in [Cerioli and Meseguer 1997] uses a semantic argument to establish the soundness of the translation once and for all. The former has the disadvantage that the backwards proof construction is often very difficult. The latter has the disadvantage that the model theoretical soundness argument is not formalized itself.

Building on our representation of logics as signatures, models as morphisms, and logic translations as morphisms [Rabe 2012], we can apply observational equivalence to formalize and verify the soundness of model theoretical borrowing.



Example 4.16 (*Model Theoretical Borrowing*). Consider the well-known translation of modal logic to first-order logic, which makes worlds explicit. We will sketch its formalization and verification in DTT focusing on where in the proof a logical relation is used crucially. All details are reported in [Sojakova 2010].

The formalization in LF yields the diagram of LF signatures above. Here FOL' extends FOL from Ex. 4.3 with a binary predicate symbol $p : i \to i \to o$; and ML extends PL with the declaration $\Box : o \to o$. $FOLModel$ and $KripkeModel$ extend ZFC from Ex. 4.4 with declarations that represent models of the respective logic. In particular, the former declares a set $univ$ to interpret i and a binary relation on $univ$ to interpret p; the latter declares a set $world$ representing Kripke worlds and the binary accessibility relation acc on it.

The vertical morphisms formalize the interpretation functions of the respective logics. In particular, FOL^{sem} maps o to the ZFC-type representing the set $\{0,1\}$ of booleans; and ML^{sem} maps o to the ZFC-type representing the set-theoretical functions from $world$ to $\{0,1\}$. The horizontal morphisms formalize the respective translations of syntax and models. In particular, μ^{syn} maps o to $i \to o$; and μ^{sem} maps $world$ to $univ$ and acc to p.

[Rabe 2012] shows that commutativity of the upper rectangle guarantees soundness of the translation. But this requirement is very strong and does not apply in many examples. Now we can improve upon this result by observing that: A model theoretical logic translation is sound iff the morphisms $FOL^{sem} \circ \mu^{syn}$ and $\mu^{sem} \circ ML^{sem}$ are observationally equivalent at pf.

Indeed, we have been able to apply Thm. 4.15 and given a logical relation to formalize and mechanically verify the soundness argument. The equivalence type E simply states the equality of the truth values $(FOL^{sem} \circ \mu^{syn})(F)$ and $(\mu^{sem} \circ ML^{sem})(F)$ for any formula F, and the cases in the logical relation correspond exactly to the cases of the induction in the usual informal soundness proof.

Module context	M	$::=$	$\cdot \mid M,\ \%\texttt{sig}\ S\ =\ \{\Sigma\} \mid M,\ \%\texttt{view}\ m\ :S\ \to S\ =\{\mu\}$
			$\mid M,\ \%\texttt{rel}\ r\ :m\times\ldots\times m:S\to S\ =\{\rho\}$
Signatures	Σ	$::=$	$\cdot \mid \Sigma,\ \%\texttt{include}\ S \mid \Sigma,\ i:U$
Morphisms	μ	$::=$	$\cdot \mid \mu,\ \%\texttt{include}\ m \mid \mu,\ i:=U$
Relations	ρ	$::=$	$\cdot \mid \rho,\ \%\texttt{include}\ r \mid \rho,\ i:=U$
Expressions	U	$::=$	$\texttt{type} \mid i \mid x \mid \Pi x:U.U \mid \lambda x:U.U \mid U\ U$

Fig. 4. Grammar of the Module System

5. IMPLEMENTATION AND MODULE SYSTEM

Twelf [Pfenning and Schürmann 1999] is an implementation of the LF incarnation of DTT that comes with a simple and scalable module system [Rabe and Schürmann 2009]. The latter permits structured modules, i.e., large signatures and morphisms are composed from smaller ones through instantiation and inheritance. We have extended both the module system and the implementation to provide a module system for logical relations.

For the full account of the Twelf module system, we refer the reader to [Rabe and Schürmann 2009]. Here we only consider a fragment that conveys the general idea; in particular, we omit the instantiation of parametric signatures, morphisms, and logical relations. The resulting grammar is given in Fig. 4, where we unify constants c and type family symbols a into identifiers i, and terms M, type families C, and kinds K into expressions U.

Here a module context is a list of named modules, and the $\%\texttt{include}$ declarations import all declarations of another module into the current one. A structured morphism or logical relation follows the structure of its domain signature: if a signature S includes S', then a morphism $m : S \to T$ includes a morphism $m' : S' \to T$ and a logical relation $r : m_1 \times \ldots \times m_n : S \to T$ includes a logical relation $r' : m'_1 \times \ldots \times m'_n : S' \to T$.

Example 5.1. We add product types to our running example of the Church-Curry translation and reformulate it in a modular way in Fig. 5, 6, 7. Then we leverage that structure to give a modular logical relation in Fig. 8.

As our examples are already very close to Twelf's concrete input syntax, they can be mechanically verified by Twelf directly. Modular Twelf and the sources of all examples used in this paper are available at [Rabe and Sojakova 2012].

We omit the formal semantics of the module system and only state two important theorems for composition and summation that permit building the semantics of structured modules from the semantics of their components:

THEOREM 5.2 (COMPOSITION). *We have the following closure properties under composition:*

(1) Given morphisms $\mu : R \to S$ and $\nu : S \to T$, there exists a morphism $\mu\,\nu : R \to T$.
(2) Given relation $r : \mu_1 \times \ldots \times \mu_n : R \to S$ and morphism $\nu : S \to T$, there exists a logical relation $r\,\nu : (\mu_1\ \nu) \times \ldots \times (\mu_n\ \nu) : R \to T$.
(3) Given morphism $\mu : R \to S$ and relation $s : \nu_1 \times \ldots \times \nu_n : S \to T$, there exists a logical relation $\mu\,s : (\mu\ \nu_1) \times \ldots \times (\mu\ \nu_n) : R \to T$.

PROOF. We put:

— $(\mu\,\nu)(U) = \nu(\mu(U))$ and $(\mu\,\nu)(\Gamma) = \nu(\mu(\Gamma))$
— $(r\,\nu)(M) = \nu(r(M))$, $(r\,\nu)(C) = \nu(r(C))$, $(r\,\nu)^C(K) = \nu(r^C(K))$, and $(r\,\nu)(\Gamma) = \nu(r(\Gamma))$
— $(\mu\,s)(M) = s(\mu(M))$, $(\mu\,s)(C) = s(\mu(C))$, $(\mu\,s)^C(K) = s^C(\mu(K))$, and $(\mu\,s)(\Gamma) = s(\mu(\Gamma))$

$$
\begin{array}{l}
\%\texttt{sig}\ \mathit{Church} = \{ \\
\quad \mathit{tp} \quad : \quad \texttt{type} \\
\quad \mathit{tm} \quad : \quad \mathit{tp} \to \texttt{type} \\
\} \\
\\
\%\texttt{sig}\ \mathit{Church}_{\to} = \{ \\
\quad \%\texttt{include}\ \mathit{Church} \\
\quad \Rightarrow \quad : \quad \mathit{tp} \to \mathit{tp} \to \mathit{tp} \\
\quad \mathit{lam} \quad : \quad \Pi A.\Pi B.\ (\mathit{tm}\ A \to \mathit{tm}\ B) \to \mathit{tm}\ (A \Rightarrow B) \\
\quad \mathit{app} \quad : \quad \Pi A.\Pi B.\ \mathit{tm}\ (A \Rightarrow B) \to \mathit{tm}\ A \to \mathit{tm}\ B \\
\} \\
\\
\%\texttt{sig}\ \mathit{Church}_{\times} = \{ \\
\quad \%\texttt{include}\ \mathit{Church} \\
\quad \times \quad : \quad \mathit{tp} \to \mathit{tp} \to \mathit{tp} \\
\quad \mathit{pair} \quad : \quad \Pi A.\Pi B.\ \mathit{tm}\ A \to \mathit{tm}\ B \to \mathit{tm}\ (A \times B) \\
\quad \pi_1 \quad : \quad \Pi A.\Pi B.\ \mathit{tm}\ (A \times B) \to \mathit{tm}\ A \\
\quad \pi_2 \quad : \quad \Pi A.\Pi B.\ \mathit{tm}\ (A \times B) \to \mathit{tm}\ B \\
\} \\
\\
\%\texttt{sig}\ \mathit{Church}_{\to\times} = \{ \\
\quad \%\texttt{include}\ \mathit{Church}_{\to} \\
\quad \%\texttt{include}\ \mathit{Church}_{\times} \\
\}
\end{array}
$$

Fig. 5. Modular Church Encoding of STT

The inductive and preservation properties follow directly from the corresponding properties for the morphisms and relations involved. □

Note that, as is usual for logical relations (see, e.g., [Plotkin et al. 2000]), binary logical relations $r_1 : \mu_1 \times \mu_2$ and $r_2 : \mu_2 \times \mu_3$ do not compose, i.e., if c is the relation arising by index-wise composition of the binary relations r_1 and r_2, we do not in general have $c : \mu_1 \times \mu_3$.

Definition 5.3. We say that two signatures S and S' are compatible if for every $i : E$ in S and $i : E'$ in S', we have $E = E'$. We say that two morphisms μ and μ' are compatible if for every $i := E$ in μ and $i := E'$ in μ', we have $E = E'$. We define compatibility of two logical relations accordingly.

Definition 5.4 (*Sum*). For two compatible signatures, morphisms, or relations L and L', we write $L + L'$ for the signature, morphism, or relation, respectively, arising from the concatenation L, L' by removing all duplicates of previous declarations.

THEOREM 5.5 (SUM). *We have the following closure properties of summation:*

(1) If $\vdash S_1$ `sig` *and* $\vdash S_2$ `sig` *and* S_1 *and* S_2 *are compatible, then* $\vdash S_1 + S_2$ `sig`.
(2) If $\vdash \mu_i : S_i \to T$ *for* $i = 1, 2$ *and if* S_1 *and* S_2 *as well as* μ_1 *and* μ_2 *are compatible, then* $\vdash \mu_1 + \mu_2 : S_1 + S_2 \to T$.
(3) If $\vdash \mu_i : S_i \to T$ *and* $r_i : \mu_i : S_i \to T$ *for* $i = 1, 2$, *and if* S_1 *and* S_2 *as well as* μ_1 *and* μ_2 *as well as* r_1 *and* r_2 *are compatible, then* $r_1 + r_2 : \mu_1 + \mu_2 : S_1 + S_2 \to T$.

PROOF. The proofs are straightforward. □

$$
\begin{array}{l}
\%\texttt{sig}\ \mathit{Curry} = \{ \\
\quad \begin{array}{lcl} \mathit{tp} & : & \texttt{type} \\ \mathit{tm} & : & \texttt{type} \\ \# & : & \mathit{tm} \to \mathit{tp} \to \texttt{type} \end{array} \\
\}
\end{array}
$$

$$
\begin{array}{l}
\%\texttt{sig}\ \mathit{Curry}_{\to} = \{ \\
\quad \%\texttt{include}\ \mathit{Curry} \\
\quad \begin{array}{lcl} \Rightarrow & : & \mathit{tp} \to \mathit{tp} \to \mathit{tp} \\ \mathit{lam} & : & (\mathit{tm} \to \mathit{tm}) \to \mathit{tm} \\ \mathit{app} & : & \mathit{tm} \to \mathit{tm} \to \mathit{tm} \\ \#_{\mathit{lam}} & : & \Pi A.\Pi B.\Pi f.\ (\Pi x.\ x \# A \to (f\ x) \# B) \to (\mathit{lam}\ f) \# (A \Rightarrow B) \\ \#_{\mathit{app}} & : & \Pi A.\Pi B.\Pi f.\Pi x.\ f \# (A \Rightarrow B) \to x \# A \to (\mathit{app}\ f\ x) \# B \end{array} \\
\}
\end{array}
$$

$$
\begin{array}{l}
\%\texttt{sig}\ \mathit{Curry}_{\times} = \{ \\
\quad \%\texttt{include}\ \mathit{Curry} \\
\quad \begin{array}{lcl} \times & : & \mathit{tp} \to \mathit{tp} \to \mathit{tp} \\ \mathit{pair} & : & \mathit{tm} \to \mathit{tm} \to \mathit{tm} \\ \pi_1 & : & \mathit{tm} \to \mathit{tm} \\ \pi_2 & : & \mathit{tm} \to \mathit{tm} \\ \#_{\mathit{pair}} & : & \Pi A.\Pi B.\Pi m.\Pi n.\ m \# A \to n \# B \to (\mathit{pair}\ m\ n) \# (A \times B) \\ \#_{\pi_1} & : & \Pi A.\Pi B.\Pi m.\ m \# (A \times B) \to (\pi_1\ m) \# A \\ \#_{\pi_2} & : & \Pi A.\Pi B.\Pi m.\ m \# (A \times B) \to (\pi_2\ m) \# B \end{array} \\
\}
\end{array}
$$

$$
\begin{array}{l}
\%\texttt{sig}\ \mathit{Curry}_{\to\times} = \{ \\
\quad \%\texttt{include}\ \mathit{Curry}_{\to} \\
\quad \%\texttt{include}\ \mathit{Curry}_{\times} \\
\}
\end{array}
$$

Fig. 6. Modular Curry Encoding of STT

$$
\begin{array}{l}
\%\texttt{view}\ \mathit{TypeEras} : \mathit{Church} \to \mathit{Curry} = \{ \\
\quad \begin{array}{lcl} \mathit{tp} & := & \mathit{tp} \\ \mathit{tm} & := & \lambda A.\ \mathit{tm} \end{array} \\
\}
\end{array}
$$

$$
\begin{array}{l}
\%\texttt{view}\ \mathit{TypeEras}_{\to} : \mathit{Church}_{\to} \to \mathit{Curry} = \{ \\
\quad \%\texttt{include}\ \mathit{TypeEras} \\
\quad \begin{array}{lcl} \Rightarrow & := & \lambda A.\lambda B.\ A \Rightarrow B \\ \mathit{lam} & := & \lambda A.\lambda B.\lambda f.\ \mathit{lam}\ f \\ \mathit{app} & := & \lambda A.\lambda B.\lambda f.\lambda a.\ \mathit{app}\ f\ a \end{array} \\
\}
\end{array}
$$

$$
\begin{array}{l}
\%\texttt{view}\ \mathit{TypeEras}_{\times} : \mathit{Church}_{\times} \to \mathit{Curry} = \{ \\
\quad \%\texttt{include}\ \mathit{TypeEras} \\
\quad \begin{array}{lcl} \times & := & \lambda A.\lambda B.\ A \times B \\ \mathit{pair} & := & \lambda A.\lambda B.\lambda m.\lambda n.\ \mathit{pair}\ m\ n \\ \pi_1 & := & \lambda A.\lambda B.\lambda m.\ \pi_1\ m \\ \pi_2 & := & \lambda A.\lambda B.\lambda m.\ \pi_2\ m \end{array} \\
\}
\end{array}
$$

$$
\begin{array}{l}
\%\texttt{view}\ \mathit{TypeEras}_{\to\times} : \mathit{Church}_{\to\times} \to \mathit{Curry} = \{ \\
\quad \%\texttt{include}\ \mathit{TypeEras}_{\to} \\
\quad \%\texttt{include}\ \mathit{TypeEras}_{\times} \\
\}
\end{array}
$$

Fig. 7. Modular Type Erasure Translation

$\%\texttt{rel}\ \mathit{TypePres} : \mathit{TypeEras} = \{$
$\quad \mathit{tp} := \lambda A : \mathit{tp}.\ \mathit{unit}$
$\quad \mathit{tm} := \lambda A : \mathit{tp}.\ \lambda\_.\ x \# A$
$\}$

$\%\texttt{rel}\ \mathit{TypePres}_{\to} : \mathit{TypeEras}_{\to} = \{$
$\quad \%\texttt{include}\ \mathit{TypePres}$
$\quad \Rightarrow := \lambda A : \mathit{tp}.\ \lambda\_.\ \lambda B : \mathit{tp}.\ \lambda\_.\ *$
$\quad \mathit{app} := \lambda A : \mathit{tp}.\ \lambda\_.\ \lambda B : \mathit{tp}.\ \lambda\_.$
$\qquad \lambda f : \mathit{tm}.\ \lambda f^* : f \# (A \Rightarrow B).\ \lambda x : \mathit{tm}.\ \lambda x^* : x \# A.\ \#_{app}\ A\ B\ f\ x\ f^*\ x^*$
$\quad \mathit{lam} := \lambda A : \mathit{tp}.\ \lambda\_.\ \lambda B : \mathit{tp}.\ \lambda\_.$
$\qquad \lambda f : \mathit{tm} \to \mathit{tm}.\ \lambda f^* : (\Pi x : \mathit{tm}.\ x \# A \to (f\ x) \# B)\ \#_{lam}\ A\ B\ f\ f^*$
$\}$

$\%\texttt{rel}\ \mathit{TypePres}_{\times} : \mathit{TypeEras}_{\times} = \{$
$\quad \%\texttt{include}\ \mathit{TypePres}$
$\quad \times := \lambda A : \mathit{tp}.\ \lambda\_.\ \lambda B : \mathit{tp}.\ \lambda\_.\ *$
$\quad \mathit{pair} := \lambda A : \mathit{tp}.\ \lambda\_.\ \lambda B : \mathit{tp}.\ \lambda\_.$
$\qquad \lambda m : \mathit{tm}.\ \lambda m^* : m \# A.\ \lambda n : \mathit{tm}.\ \lambda n^* : n \# B.\ \#_{pair}\ A\ B\ m\ n\ m^*\ n^*$
$\quad \pi_1 := \lambda A : \mathit{tp}.\ \lambda\_.\ \lambda B : \mathit{tp}.\ \lambda\_.$
$\qquad \lambda m : \mathit{tm}.\ \lambda m^* : m \# (A \times B).\ \#_{\pi_1}\ A\ B\ m\ m^*$
$\quad \pi_2 := \lambda A : \mathit{tp}.\ \lambda\_.\ \lambda B : \mathit{tp}.\ \lambda\_.$
$\qquad \lambda m : \mathit{tm}.\ \lambda m^* : m \# (A \times B).\ \#_{\pi_2}\ A\ B\ m\ m^*$
$\}$

$\%\texttt{rel}\ \mathit{TypePres}_{\to\times} : \mathit{TypeEras}_{\to\times} = \{$
$\quad \%\texttt{include}\ \mathit{TypePres}_{\to}$
$\quad \%\texttt{include}\ \mathit{TypePres}_{\times}$
$\}$

Fig. 8. Modular Type Preservation Proof

For simplicity, we have stated Thm. 5.5 only for binary unions of unary logical relations. The according result holds for m-ary unions of n-ary logical relations.

In particular, Thm. 5.5 permits composing logical relations from separately verified components, as we did when forming $\mathit{TypePres}_{\to\times}$, without having to reverify the composed relation.

6. CONCLUSION

We have presented a definition of logical relations for dependent type theory (DTT) with $\beta\eta$-equality. The definition is reflective in the sense that a relation at a given type is itself represented in DTT as a type family. Our central theorem states that all valid logical relations satisfy the Basic Lemma.

The validity of a logical relation is decidable and we have extended the Twelf system with a new primitive declaration for logical relations. We have used this implementation to represent and mechanically verify a number of meta-theorems in DTT. Since the implementation is modular, our approach is well-suited for the integration with the LATIN logic atlas [Codescu et al. 2011], which already uses Twelf to formalize the syntax and semantics of a wide variety of formal systems and translation theorems in a systematically modular way.

By choosing the appropriate DTT-signatures and morphisms, our definition attempts to unify syntactic and semantic logical relations and permit representing logical relations of object logics as DTT logical relations. Because it is possible to represent DTT in itself, an interesting further application would be to represent our proof of the Basic Lemma in DTT along the lines of our Ex. 4.7. This would yield a mechanical verification of our Basic Lemma for DTT in DTT itself.

While our focus here has been on DTT, we believe our approach can be transferred to other logical frameworks. In particular, the notions of signatures and morphisms can be defined routinely for most declarative languages; the other requirement is a framework's ability to express judgements about terms of arbitrary types. Besides DTT and type theories subsuming it, the most interesting example of such a logical framework is higher-order logic, where one would use logical predicates instead of type families.

Another way to transfer our results is to use the well-known correspondence between signatures and record (or dependent sum) types. In extensions of DTT with record types that can contain type declarations, our signatures S and T can be represented as record types $\overline{S}$ and $\overline{T}$, and our morphisms $\mu : S \to T$ as functions $\overline{\mu} : \overline{T} \to \overline{S}$. Consequently, logical relations $r : \mu : S \to T$ can be represented as terms $\overline{r} : \overline{T} \to \overline{S}^{\#}$ where $\overline{S}^{\#}$ arises from $\overline{S}$ by replacing the type of every field according to Def. 3.5.

REFERENCES

Atkey, R. 2009. A deep embedding of parametric polymorphism in Coq. In *Workshop on Mechanizing Metatheory*. 2

Atkey, R. 2012. Relational Parametricity for Higher Kinds. In *Proceedings of CSL*. to appear. 2

Bernardy, J., Jansson, P., and Paterson, R. 2010. Parametricity and dependent types. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, P. Hudak and S. Weirich, Eds. ACM, 345–356. 2, 3, 4, 11, 22, 23

Bernardy, J. and Lasson, M. 2011. Realizability and parametricity in pure type systems. In *Foundations of Software Science and Computational Structures FOSSACS*, M. Hofmann, Ed. Lecture Notes in Computer Science Series, vol. 6604. Springer, 108–122. 3

Bertot, Y. and Castéran, P. 2004. *Coq'Art: The Calculus of Inductive Constructions*. Springer. 2

Böhme, S. 2007. Free theorems for sublanguages of Haskell. M.S. thesis. 3

Cerioli, M. and Meseguer, J. 1997. May I Borrow Your Logic? (Transporting Logical Structures along Maps). *Theoretical Computer Science 173*, 311–347. 26

Codescu, M., Horozal, F., Kohlhase, M., Mossakowski, T., and Rabe, F. 2011. Project Abstract: Logic Atlas and Integrator (LATIN). In *Intelligent Computer Mathematics*, J. Davenport, W. Farmer, F. Rabe, and J. Urban, Eds. Lecture Notes in Computer Science Series, vol. 6824. Springer, 287–289. 30

Coquand, T. and Gallier, J. 1990. A Proof of Strong Normalization For the Theory of Constructions Using a Kripke-Like Interpretation. In *Workshop on Logical Frameworks–Preliminary Proceedings*. 2, 4

Girard, J., Taylor, P., and Lafont, Y. 1989. *Proofs and Types*. Cambridge University Press. 2

Harper, R., Honsell, F., and Plotkin, G. 1993. A framework for defining logics. *Journal of the Association for Computing Machinery 40,* 1, 143–184. 3, 5, 18

Harper, R. and Pfenning, F. 2005. On Equivalence and Canonical Forms in the LF Type Theory. *ACM Transactions On Computational Logic 6,* 1, 61–101. 2, 4, 5

Harper, R., Sannella, D., and Tarlecki, A. 1994. Structured presentations and logic representations. *Annals of Pure and Applied Logic 67*, 113–160. 5

Horozal, F. and Rabe, F. 2011. Representing Model Theory in a Type-Theoretical Logical Framework. *Theoretical Computer Science 412,* 37, 4919–4945. 24

Iancu, M. and Rabe, F. 2011. Formalizing Foundations of Mathematics. *Mathematical Structures in Computer Science 21,* 4, 883–911. 3, 20

Johann, P. 2002. A generalization of short-cut fusion and its correctness proof. *Higher-Order and Symbolic Computation 15,* 4, 273–300. 2

Johann, P. and Voigtländer, J. 2006. The impact of seq on free theorems-based program transformations. *Fundamenta Informaticae 69*, 63–102. 2

Keller, C. and Lasson, M. 2012. Parametricity in an Impredicative Sort. In *Proceedings of CSL*. to appear. 2, 3

Mairson, H. 1991. Outline of a proof theory of parametricity. In *Proceedings of Functional Programming Languages and Computer Architecture*, J. Hughes, Ed. Springer, 313–327. 2

Meng, J. and Paulson, L. 2008. Translating Higher-Order Clauses to First-Order Clauses. *Journal of Automated Reasoning 40,* 1, 35–60. 26

Neis, G., Dreyer, D., and Rossberg, A. 2011. Non-parametric parametricity. *Journal of Functional Programming 21,* 4-5, 497–562. 2

Norell, U. 2005. The Agda WiKi. http://wiki.portal.chalmers.se/agda. 3

Pfenning, F. 2001. Logical Frameworks. In *Handbook of Automated Reasoning*. 1063–1147. 5

Pfenning, F. and Schürmann, C. 1999. System description: Twelf - a meta-logical framework for deductive systems. *Lecture Notes in Computer Science 1632*, 202–206. 3, 8, 27

Plotkin, G. 1973. Lambda-Definability and Logical Relations. Memorandum SAI–RM–4, University of Edinburgh. 2

Plotkin, G. and Abadi, M. 1993. A logic for parametric polymorphism. In *Typed Lambda Calculi and Applications*, M. Bezem and J. Groote, Eds. Springer, 361–375. 2

Plotkin, G., Power, J., Sannella, D., and Tennent, R. 2000. Lax Logical Relations. In *Colloquium on Automata, Languages and Programming*. LNCS Series, vol. 1853. Springer, 85–102. 28

Rabe, F. 2012. A Logical Framework Combining Model and Proof Theory. *Mathematical Structures in Computer Science*. to appear; see http://kwarc.info/frabe/Research/rabe_combining_10.pdf. 24, 26

Rabe, F. and Schürmann, C. 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 International Conference Proceeding Series Series, vol. LFMTP'09. ACM Press, 40–48. 8, 27

Rabe, F. and Sojakova, K. 2012. Twelf Sources and Examples. Available at https://svn.kwarc.info/repos/twelf/projects/logrels/index.html. 3, 18, 23, 27

Reynolds, J. 1974. On the Relation between Direct and Continuation Semantics. In *Second Colloquium Automata, Languages and Programming*. LNCS. Springer, 141–156. 2, 21, 25

Reynolds, J. 1983. Types, Abstraction, and Parametric Polymorphism. In *Information Processing*. North-Holland, Amsterdam, 513–523. 2

Schürmann, C. and Sarnat, J. 2008. Structural Logical Relations. In *Logic in Computer Science*, F. Pfenning, Ed. IEEE Computer Society Press, 69–80. 3, 4, 20, 21

Sojakova, K. 2010. Mechanically Verifying Logic Translations. Master's thesis, Jacobs University Bremen. 26

Statman, R. 1985. Logical Relations and the Typed Lambda Calculus. *Information and Control 65*, 85–97. 2

Takeuti, I. 2001. The theory of parametricity in lambda cube. 3

Vytiniotis, D. and Weirich, S. 2010. Parametricity, type equality, and higher-order polymorphism. *Journal of Functional Programming 20,* 2, 175–210. 2

Wadler, P. 1989. Theorems for free! In *Proc. ACM Conf. Functional Programming and Computer Architecture*. 347–359. 2