Representing Model Theory in a Type-Theoretical Logical Framework

Fulya Horozal and Florian Rabe

Jacobs University Bremen, Germany

Abstract

We give a comprehensive formal representation of first-order logic using the recently developed module system for the Twelf implementation of the Edinburgh Logical Framework LF. The module system places strong emphasis on signature morphisms as the main primitive concept, which makes it particularly useful to reason about structural translations, which occur frequently in proof and model theory.
Syntax and proof theory are encoded in the usual way using LF's higher order abstract syntax and judgments-as-types paradigm, but using the module system to treat all connectives and quantifiers independently. The difficulty is to reason about the model theory, for which the mathematical foundation in which the models are expressed must be encoded itself. We choose a variant of Martin-Löf's type theory as this foundation and use it to axiomatize first-order model theoretic semantics. Then we can encode the soundness proof as a signature morphism from the proof theory to the model theory. We extend our results to models given in terms of set theory by encoding Zermelo-Fraenkel set theory in LF and giving a signature morphism from Martin-Löf type theory into it. All encodings are given explicitly and can thus be checked mechanically.
Our results demonstrate the feasibility of comprehensively formalizing large scale representation theorems and thus promise significant future applications.

1 Introduction

Since the Grundlagenkrise of mathematics logic has been an important research topic in mathematics and computer science. A central issue has always been what a logic actually is. Important research results to answer this question are logical frameworks – abstract formalisms that permit the formal definition of specific logics.

Today we observe that there are two groups of logical frameworks: those based on set theoretical foundations of mathematics that characterize logics model theoretically, and those based on type theoretical foundations that characterize logics proof theoretically. The former go back to Tarski's view of consequence ([Tar33,TV56]) with institutions ([GB92,GR02]) and general logics ([Mes89]) being the most important examples. The latter are usually based on the Curry-Howard correspondence ([CF58,How80]), examples being Automath ([dB70]), Isabelle ([Pau94]), and the Edinburgh Logical Framework (LF, [HHP93]).

While some model-theoretical frameworks attempt to integrate proof theory (e.g.,

[Mes89,MGDT05,Dia06]), the opposite integration is less developed. This is unfortunate because many of the results and techniques developed for proof theoretical logics could also benefit model-theoretical reasoning. We are particularly interested in logic encodings in LF, which is related to Martin-Löf type theory (MLTT). These represent syntax and proof theoretical semantics of a logic using higher order abstract syntax and the judgments-as-types paradigm ([ML96]). This has proved very successful for proof-theoretical logic representations ([Pfe00,HST94,AHMP98,NSM01]).

In [Rab09], we introduced a framework that attempts to preserve and exploit the respective advantages of model and proof theoretical representation. The central idea is to also represent the model theory of a logic in a type-theoretical logical framework by axiomatizing models and the interpretation of formulas using MLTT as a meta-language.

In this paper we show how to implement and reason about such logic representations in LF. We pick LF because we have recently equipped the Twelf implementation of LF with a strong module system [RS09]. This module system is rigorously based on theory morphisms, which have proved very successful to reason about model-theoretical logic representations (e.g., [GB92,AHMS99,SW83]). Therefore, it is particularly appropriate for the modular development of syntax, proof theory, and models, and the translations between them.

In Sect. 2, we describe the Twelf system and its module system, and in Sect. 3, we describe how logics are represented in it. Our main result is the full representation of first-order logic (FOL) in Sect. 4: It comprises syntax, proof theory, model theory, and soundness proof of first-order logic, all of which use the module system to treat all connectives and quantifiers independently. In particular, the soundness is verified mechanically by Twelf. In Sect. 5, we go one step further: We show how to encode both Zermelo-Fraenkel set theory (ZFC) and a translation from MLTT to ZFC in LF. Thus, we can represent set-theoretical models as LF theory morphisms and reason about them within the logical framework.

2 The Twelf System

The Twelf system is an implementation of the logical framework LF designed as a meta-language for the representation of deductive systems. It is a dependent type theory with typed terms and kinded type families.

$$\begin{array}{lll} \text{Kinds:} & K & ::= \texttt{type} \mid A \to K \\ \text{Type families:} & A, B & ::= a \mid A\ M \mid \Pi_{x:A} B \mid A \to B \\ \text{Objects:} & M, N & ::= c \mid x \mid \lambda_{x:A} M \mid M\ N \end{array}$$

Twelf features the dependent product type constructor $\Pi_{x:A}B$ and its introductory axiom, the λ-binder $\lambda_{x:A}M$. As usual, application is written as iuxtaposition $M\ N$. $A \to B$ abbreviates $\Pi_{x:A}B$ if x does not occur freely in B. Type families are kinded by kinds, where types are the type families kinded by `type`, and objects are typed by types.

Twelf signatures contain declarations of type- or object-level constants. Con-

stants are declared in the form of declarations $a : K$ or $c : A$, or definitions $a : K = A$ or $c : A = M$. Variables $x : A$ are typed, never kinded.

The Twelf module system permits to use multiple named signatures that can be related via inheritance, i.e., *structures*, and translations, i.e., *views*. Readers familiar with modular theory development languages such as development graphs ([AHMS99]) will recognize structures as definitional theory morphisms and views as postulated theory morphisms. The grammar is given in Fig. 1 and explained below.

Signature graph	$\mathcal{G}$	::=	$\cdot \mid \mathcal{G},\ D_T \mid \mathcal{G},\ D_v$
Signature	D_T	::=	$T := \{\Sigma\}$
View	D_v	::=	$v : T \to T := \{\sigma\}$
Signature body	Σ	::=	$\cdot \mid \Sigma,\ D_c \mid \Sigma,\ D_s \mid \Sigma,\ D_I$
Constant	D_c	::=	$c : C \mid c : C := C$
Structure	D_s	::=	$s : T := \{\sigma\}$
Inclusion	D_I	::=	include T
Assignment list	σ	::=	$\cdot \mid \sigma,\ \boldsymbol{c} := C \mid \sigma,\ \boldsymbol{s} := \mu$
Term	C	::=	kind, type family, or object
Morphism	μ	::=	$\boldsymbol{s} \mid v \mid \mu\mu$
Qualified identifiers	$\boldsymbol{c}$	::=	$s.\ \ldots\ .s.c$
	$\boldsymbol{s}$	::=	$s.\ \ldots\ .s.s$
Identifiers	T, v, c, s		

Fig. 1. The Grammar for Expressions

We will give all larger listings of signatures in Twelf's concrete syntax as given below. Keywords are introduced with `%` and precede all declarations except for constant declarations. The Π-binder is written with braces `{}`, the λ-binder with square brackets `[]`.

```
Signatures:  D_T  ::= %sig T = { (D_c | D_s | D_I)* }.
Views:       D_v  ::= %view v : S -> T = { (c := C | %struct s := μ)* }.
Inclusions:  D_I  ::= %include T.
Constants:   D_c  ::= c : C. | c : C = C.
Structures:  D_s  ::= %struct s : S = { (c := C. | %struct s := μ.)* }.
Expressions: C    ::= type | c | x | C -> C | C C | {x:C} C | [x:C] C
```

An *inheritance* relation from signature S to signature T is represented by a *structure declaration* occurring within T, which creates a possibly translated copy of S. The copied constants are accessible by qualified names formed by prefixing the structure name. The declaration of a structure s induces a signature morphism from the instantiated signature S to its containing signature, which maps every constant c of S to $s.c$.

For example, consider the following signature declarations, which we will reuse later on. `Base` contains a type o for formulas and an o-indexed type family *ded*. This type family exemplifies how logic encodings in LF represent judgments as types and derivations as objects: Objects of type *ded* A represent derivations of the judgment

"A is true".

```
%sig Base = {
  o    : type.
  ded : o -> type.
}.
%sig NEG = {
  %struct base : Base.
  ¬ : base.o -> base.o.
}.
```

The signature `NEG` encodes the negation connective by inheriting from `Base` with a structure called `base` and adding the constant ¬ encoding the unary negation connective. Then the signature `NEGPF` encodes natural deduction style proof rules for it. Firstly, it inherits from `Base` and `NEG` where the instantiation `%struct base := base.` in the structure `neg` works as follows: The left side of the instantiation is a symbol declared in the domain signature – here: `NEG` – and the right side is an expression over the codomain signature – here: `NEGPF`. This instantiation has the effect of a sharing declaration: The two structures `base` and `neg.base` inheriting from `Base` are identified. Secondly, it declares the constants `notI` and `notE` encoding the introduction and the elimination rule of negation.

```
%sig NEGPF = {
  %struct base : Base.
  %struct neg  : NEG = {%struct base := base.}.
  notI : (base.ded A -> {B} base.ded B) -> base.ded (neg.¬ A).
  notE : base.ded A -> base.ded (neg.¬ A) -> {B} base.ded B.
}.
```

Note that we use implicit arguments: Upper case free variables in declarations are assumed be implicitly Π-bound on the outside. This has the effect of free parameters. For example, in `notE`, the variable `A` is free. The verbal reading of the rule is "For any A, if A is true and $\neg A$ is true, then for all B we have B is true ".

Twelf computes the semantics of the modular signatures by elaborating them to the non-modular syntax. For example, `NEGPF` is equivalent to

```
%sig NEGPF2 = {
  base.o       : type.
  base.ded     : base.o -> type.
  neg.base.o   : type                = base.o.
  neg.base.ded : neg.base.o -> type = base.ded.
  neg.¬  : neg.base.o -> neg.base.o.
  notI : (base.ded A -> {B} base.ded B) -> base.ded (neg.¬ A).
  notE : base.ded (neg.¬ A) -> base.ded A -> {B} base.ded B.
}.
```

A special case of inheritance relation is the *include declaration*. The declaration `%include` S occurring in T creates an inclusion from S to T. This is similar to a structure declaration but simpler and only possible in certain cases. If a signature is included in multiple ways, all inclusions are identified. Therefore, a symbol c included from S into T is identified uniquely by the name $S..c$.

A *translation* relation between two signatures is encoded by *view declarations*. A view is similar to a structure except that it occurs on toplevel and gives domain and codomain explicitly. Views must instantiate all constants (except those that have definitions) of the domain signature with expressions of the codomain signature. Views induce signature morphisms in the obvious way.

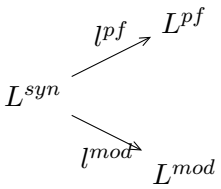
Finally, we call a list of (names of) structures or views a *morphism*. Its semantics

is that of the diagram-order composition of the signature morphisms induced by its components. Structure names may be qualified, e.g., above the structure `neg.base` is a morphism from `Base` to `NEGPF`; it is an abbreviation of the composition of the morphism induced by the structure `base` of `NEG` and the morphism induced by the structure `neg` of `NEGPF`. Morphisms preserve all judgments regarding well-formedness, typing, and equality (see [HST94,RS09] for the preservation results).

3 A Logical Framework Combining Proof and Model Theory

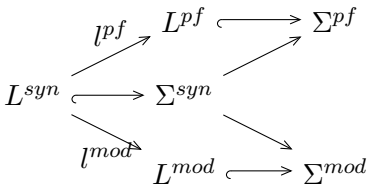
Since the structuring concepts of the Twelf module system are based on signature morphisms, we can obtain very elegant encodings of mathematical concepts that are defined in terms of signature morphisms. One example is the logical framework we gave in [Rab09].

A representation of a logic L in this framework consists of three signatures: L^{syn} for the syntax, L^{pf} for the proof theory, and L^{mod} for the model theory. These are related by signature morphisms that translate from L^{syn} to the others as in the diagram on the right. The morphism l^{pf} is typically just an inclusion; this is in keeping with the practice of logic encodings in LF where syntax and proof theory are closely related. The morphism l^{mod} interprets the syntax in the semantic realm. In particular, l^{mod} maps o to a type encoding the set of truth values of L. Therefore, L^{mod} must contain an encoding of the mathematical foundation in which the models are expressed, which makes l^{mod} typically more complicated than l^{pf}.



For example, for propositional logic with only negation, L^{syn} and L^{pf} could be the signatures `NEG` and `NEGPF` from Sect. 2. l^{pf} would be the morphism induced by the structure `neg` of `NEGPF`. L^{mod} could be a signature formalizing the two-element boolean lattice. Note that this formalization might be carried out in some other logic, e.g., first-order logic, which would have to be included in L^{mod}. Alternatively, L^{mod} could include a full axiomatization of set theory; then l^{mod} could map `base.o` to the set $\{0, 1\}$.

A specific theory Σ of L is represented as an extension of L. This corresponds to the uniform logic encodings in LF given in [HST94]. For example, signatures of propositional logic are sets of propositional variables, and the set $\Sigma = \{p_1, \ldots, p_n\}$ is encoded as the LF-signature L^{syn}, $p_1 : o$, $\ldots$, $p_n : o$. The corresponding extensions Σ^{pf} and Σ^{mod} of L^{pf} and L^{mod} lead to the diagram on the right.



With two further assumptions, it becomes possible to represent all aspects of L in LF: L^{syn} should contain a type o and a type family $ded : o \to type$, which have the same intuition as explained for the signature `Base` in Sect. 2.

Then Σ-sentences are represented as β-η-normal LF-terms of type o over the

signature Σ^{syn}. Σ-proofs of A using assumptions $A_1, \ldots, A_n$ are represented as β-η-normal LF-terms over Σ^{pf} of type $l^{pf}(ded\ A_1 \to \ldots \to ded\ A_n \to ded\ A)$ where $l^{pf}(-)$ denotes morphism application.

Σ-models are represented as LF-models of the signature Σ^{mod}. Such a model I interprets all types A as sets $[A]^I$ and all terms s of type A as elements $[s]^I \in [A]^I$. The details can be found in [Rab09] and [AR09]. Finally, the satisfaction $I \models_\Sigma A$ of the Σ-sentence A in the Σ-model I is represented by the condition $[ded\ A]^I \neq \emptyset$.

Note how the type $ded\ A$ is used to represent truth both proof- and model-theoretically. Proof-theoretically, elements of type $ded\ A$ represent proofs of A. And model-theoretically, and $[ded] : [o] \to Set$ acts as a predicate on the set of truth values singling out the designated truth values: A truth value v is designated iff $[ded](v) \neq \emptyset$.

A particular strength of this framework is that soundness can be represented very naturally: A soundness proof of L is represented as a view from L^{pf} to L^{mod} that makes the diagram above commute. In the language of category theory ([Lan98]), Σ^{pf} and Σ^{mod} arise as pushouts along the inclusion $L^{syn} \to \Sigma^{syn}$. (The category of LF-signatures has pushouts along inclusions.) This implies that a signature morphism from L^{pf} to L^{mod} induces one from Σ^{pf} to Σ^{mod} thus proving soundness for all theories of L.

4 Representing First-Order Logic

As described in Sect. 3, the encoding of FOL consists of three main LF signatures: `FOL` for the syntax, `FOLPF` for the proof theory, and `FOLMOD` for the semantics.

A specific theory Σ of FOL is a list of function and predicate symbols and axioms. Therefore, theories can be encoded naturally in LF by extending `FOL` with declarations of the form $f : i \to \ldots \to i \to i$ for function symbols, $f : i \to \ldots \to i \to o$ for predicate symbols, and $a : ded\ F$ for axioms where the types i and o of `FOL` represent the universe and the sentences of FOL.

We will describe the signature `FOL` in Sect. 4.1, `FOLPF` in Sect. 4.2, and `FOLMOD` in Sect.4.4. To describe the semantics of first-order logic, we need a meta-language in which the models are expressed. In less formal descriptions, this meta-language is usually natural language implicitly based on some set-theoretical foundation of mathematics. Here we use a variant of Martin-Löf type theory [ML74], which we refer to as MLTT, as a simple formal language that is expressive enough to define the semantics of first-order logic. Therefore, we give the encoding of MLTT in Sect. 4.3 before we are able to define `FOLMOD`. Finally, we encode soundness by giving a view from `FOLPF` to `FOLMOD` in Sect. 4.5.

4.1 Syntax

We encode the signature `FOL` modularly, where each logical connective and quantifier is declared in a separate Twelf signature. The modular

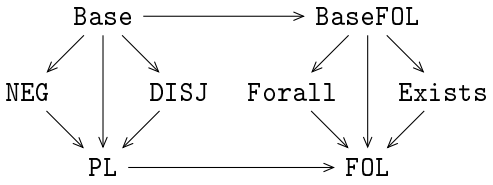


Fig. 2: Modular Encoding of FOL Syntax

representation of FOL is illustrated as a diagram in Fig. 2. Each node in this diagram corresponds to a signature declaration in Twelf, and each edge corresponds to a structure. For example, the edge between Base and NEG represents a structure declared in NEG that imports Base into NEG. For the sake of simplicity, we use only negation ($\neg$) and disjunction ($\vee$) as primitive connectives in this paper. In our Twelf encoding, we declare all connectives as primitive connectives except for equivalence.

The signature Base introduces a type o for propositions and a type family $ded : o \to type$ for the proofs of propositions. $\neg$ and $\vee$ are declared in the signatures NEG and DISJ, respectively. NEG and DISJ copy the content of Base via a structure – named as base locally in both signatures – so that o can be used in the declaration of $\neg$ and $\vee$.

```
%sig Base = {
  o   : type.
  ded : o -> type.
}.
%sig NEG = {
  %struct base : Base  %open o.
  ¬ : o -> o.
}.
%sig DISJ = {
  %struct base : Base  %open o.
  ∨ : o -> o -> o.     %infix ∨.
}.
```

The keyword %open provides syntactic sugar to avoid qualified names. Without %open o one would have to refer to the type o as base.o. The keyword %infix permits to declare a constant as infix. %infix requires arguments for the associativity and precedence, which we omit here simply because they are not so relevant.

Then we import Base, NEG and DISJ into the signature PL via the structures base, neg and disj, respectively, as shown below. This gives us an encoding of the PL syntax. In neg and disj, the assignment %struct base := base. maps the structure base of NEG and base of DISJ, respectively, to the structure base of PL. This assignment allows us to identify the copies of Base that are imported from NEG and DISJ with the copy of Base that is imported by the structure base of PL.

```
%sig PL = {
  %struct base : Base.
  %struct neg  : NEG  = {%struct base := base.}.
  %struct disj : DISJ = {%struct base := base.}.
}.
```

In the signature BaseFOL, we import Base and introduce a type i for the individuals of FOL. The universal quantifier $\forall$ and the existential quantifier $\exists$ are declared in the signatures Forall and Exists, respectively. BaseFOL is imported to both Forall and Exists so that i and o can be used to declare the quantifiers.

```
%sig BaseFOL = {
  %struct base : Base.
  i : type.
}.
%sig Forall = {
  %struct basefol : BaseFOL
              %open base.o i.
  ∀ : (i -> o) -> o.
}.
%sig Exists = {
  %struct basefol : BaseFOL
              %open base.o i.
  ∃ : (i -> o) -> o.
}.
```

Finally, we define FOL based on the signatures BaseFOL, PL, Forall and Exists. Once again, we identify the multiple imports of signatures into FOL using structure assignments. These signatures are Base imported from BaseFOL and PL, and BaseFOL imported from Forall and Exists.

```
%sig FOL = {
  %struct basefol : BaseFOL.
  %struct pl      : PL      = {%struct base    := basefol.base.}.
  %struct univq   : Forall  = {%struct basefol := basefol.}.
  %struct existq  : Exists  = {%struct basefol := basefol.}.
}.
```

4.2 Proof Theory

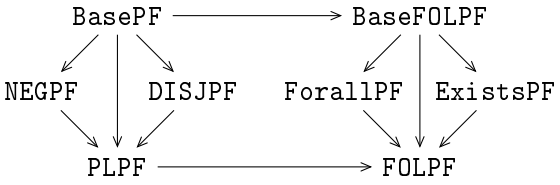


Fig. 3: Modular Encoding of FOL Proof Rules

The encoding of the signature FOLPF follows the modularity in the encoding of FOL: We define a separate signature for the natural deduction style proof rules of each logical connective and quantifier. This is illustrated in Fig. 3.

The signature BasePF is a copy of the signature Base to carry over the constants o and *ded* to the encoding proof theory. Similarly, the signature BaseFOLPF imports the signature BaseFOL to carry over i to the encoding of proof theory, and extends BasePF to first-order. We distinguish these signatures anyway because of the conceptual clarity. (The analogues of BasePF in the encodings of other logics do contain additional declarations, e.g., the structural rules of sequent style calculi.)

```
%sig BasePF = {
  %struct base : Base.
}.
%sig BaseFOLPF = {
  %struct basepf  : BasePF.
  %struct basefol : BaseFOL = {
    %struct base := basepf.base.}.
}.
```

The signatures NEGPF, DISJPF, ForallPF and ExistsPF encode the introduction and elimination rules for $\neg$, $\vee$, $\forall$ and $\exists$, respectively. They extend the respective node in Fig. 2 that declares the corresponding logical symbol by adding the proof rules. We use the well-known encoding of FOL natural deduction rules in LF, see e.g. [HHP93]. Therefore, we only present the rules for negation as an example.

```
%sig NEGPF = {
  %struct basepf : BasePF %open base.ded.
  %struct neg    : NEG = {%struct base := basepf.base.}  %open ¬.
  notI   : (ded A -> {B} ded B) -> ded (¬ A).
  notE   : ded A -> ded (¬ A) -> {B} ded B.
}.
```

The signature PLPF imports all the signatures in which the proof rules for the logical connectives are declared, and adds tertium non datur.

```
%sig PLPF = {
  %struct basepf : BasePF.
  %struct negpf  : NEGPF  = {%struct basepf := basepf.}.
  %struct disjpf : DISJPF = {%struct basepf := basepf.}.
  tnd : ded (A ∨ (¬ A)).
}.
```

The signature FOLPF extends PLPF by importing the signatures for the proof rules of the quantifiers. The multiple copies of signatures are identified to one specific copy using structure assignments.

```
%sig FOLPF = {
  %struct basefolpf : BaseFOLPF.
  %struct plpf      : PLPF      = {
    %struct basepf  := basefolpf.basepf.}
  %struct forallpf  : ForallPF  = {%struct basefolpf := basefolpf.}.
  %struct existspf  : ExistsPF  = {%struct basefolpf := basefolpf.}.
}.
```

Note that our encodings in [HR09] factor out tertium non datur into a separate signature so that we can distinguish intuitionistic and classical reasoning. We skip this here because we focus on classical model theory.

4.3 Martin-Löf Type Theory

The abstract syntax for MLTT is given below.

$$\textbf{Types } A, B, C \ldots \quad ::= a\, s_1 \ldots s_n \mid id(s,t) \mid unit \mid void \mid A + B \mid \Sigma_{x:A} B \mid \Pi_{x:A} B$$

$$\textbf{Terms } s, t, s_1, s_2, \ldots ::= x \mid c \mid refl(s) \mid * \mid !!_A \mid inj_1(s) \mid inj_2(s) \mid case(s, s_1, s_2)$$
$$\mid \quad (s,t) \mid \pi_1(s) \mid \pi_2(s) \mid \lambda_{x:S} t \mid s\ t$$

We use the following type constructors: the application of type-valued constants $a\, s_1 \ldots s_n$, extensional identity type $id(s,t)$ for terms s and t of the same type, unit type $unit$, disjoint union $A + B$, dependent product types $\Sigma_{x:A} B$, and dependent function types $\Pi_{x:A} B$. As usual $\Sigma_{x:A} B$ and $\Pi_{x:A} B$ are written as $A \times B$ and $A \to B$, respectively, if the variable x does not occur freely in B. In addition, MLTT adds an empty type $void$, and a negation type $-A$ defined as an abbreviation for $A \to void$.

The term constructors are constants c, variables x, the element $*$ of the unit type, functions $!!_A$ from $void$ to any type A, the element $refl(s)$ of the type $id(s,s)$, injections $inj_1(s)$ and $inj_2(s)$ of s into a union type, case distinctions $case(s, s_1, s_2)$ (where $s : A_1 + B_2$ and $s_i : A_i \to B$), pairs (s,t), projections $\pi_1(s)$ and $\pi_2(s)$ for a pair s, λ-abstractions $\lambda_{x:A} t$, and function applications $s\ t$.

Thus, our type theory corresponds to intuitionistic first-order logic via the Curry-Howard correspondence. The addition of the empty type to express negation is crucial to reason about model theory. For example, without negation, it would be impossible to express the condition that models must be consistent, i.e., may not interpret all formulas as truths.

Our encoding of MLTT uses the module system to treat all type constructors independently as illustrated in Fig. 4. In this paper, we will omit the somewhat complicated encoding we have for substitution and equality of types.

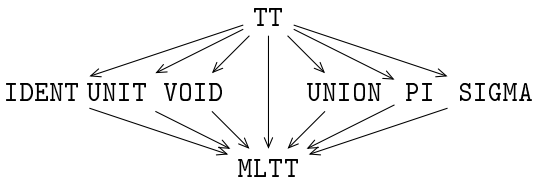


Fig. 4: Modular Encoding of MLTT

In the signature `TT`, we introduce a type `tp` for types and a type family `tm` for terms. The encoding uses intrinsic typing, i.e., the intuition of the LF-type `tm A` is that its LF-terms encode the MLTT-terms of MLTT-type `A`.

```
%sig TT = {
  tp : type.
  tm : tp -> type.
}.
```

As an example, we give the encoding of dependent sum types, which is encoded in the signature `SIGMA`. The Σ-binder is encoded using higher-order abstract syntax: `S' ([x: tm A] B)` encodes the type $\Sigma_{x:A}B$ where `[x : tm A] B` is an LF function representing an MLTT-type `B` with a free variable of MLTT-type `A`. The constants `pair`, `pi1` and `pi2` encode the usual introduction and elimination rules for dependent sum types (see, e.g., [ML74]).

```
%sig SIGMA = {
  %struct TT : TT %open tp tm.
  S'   : (tm A -> tp) -> tp.
  pair : {a : tm A} tm (B a) -> tm (S' [x] (B x)).
  pi1  : tm (S' [x : tm A] (B x)) -> tm A.
  pi2  : {u : tm (S' [x : tm A] (B x))} tm (B (pi1 u)).
}.
```

Similarly, the other type constructors are encoded in the signatures `IDENT`, `UNIT`, `VOID`, `UNION`, and `PI`. The types $id(s,t)$, $unit$, $void$, $A+B$, and $\Pi_{x:A}B$ are encoded as the LF-terms `s ==' t`, `unit`, `void`, `A +' B`, and `P' [x: tm A] B`, respectively. Note that all type constructors are primed in Twelf. This is because we will use the unprimed variants as abbreviations later. Finally, we merge all these signatures into the signature `MLTT` in the same way as for `FOL` and `FOLPF`.

4.4 Model Theory

In this section, we define the model theory of FOL in Twelf. We follow the same modularity (see Fig. 5) as in the encoding of the FOL syntax and proof theory. We use MLTT as the meta-language for the model theory. Therefore, all signatures that encode the model theory start with the declaration `%include MLTT.` in order to include the signature `MLTT`.

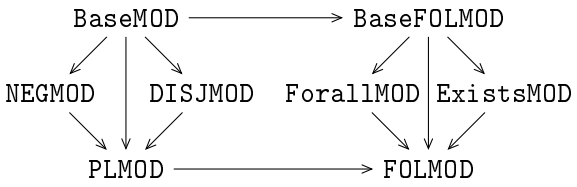


Fig. 5: Modular Encoding of the Model Theory

In `BaseMOD`, we declare the constant `o'` as an MLTT-type of propositions. We encode the truth values 1 and 0 as terms of MLTT-type `o'`, i.e., LF-type `tm o'`. The MLTT-type family `ded'` acts as a judgment on truth values: `desig1` makes `1` a designated truth value, and `desig0` makes `0` non-designated. `0` being non-designated means that a model must interpret `ded' 0` as the empty set because otherwise it could not interpret `void` as the empty set. Given a proposition `A`, `boole` returns the proof of the fact that `A` is equal to either `1` or `0`. This encodes that truth values are the elements of the set $\{1,0\}$. Note that we use unprimed type constructors as abbreviations for prefixing `tm`. For example, `A + B` abbreviates `tm (A +' B)`, `void` abbreviates `tm void'`, and

```
%sig BaseMOD = {
  %include MLTT %open tm tp.
  o'      : tp.
  1       : tm o'.
  0       : tm o'.
  ded'    : tm o' -> tp.
  desig1  : tm (ded' 1).
  desig0  : tm -' (ded' 0).
  boole   : {A} (A ==' 1 + A ==' 0).
}.
```

`-A` abbreviates `tm (-' A)`.

In `BaseFOLMOD`, we extend `BaseMOD` with the MLTT-type `i'` for individuals. The interpretation of this type is the universe of a FOL model. Then we declare `non_empty_universe`, which is an axiom stating the non-emptiness of the universe. This is provable in usual first-order axiomatizations because they permit arbitrary variables and thus representatives of elements of the universe. However, it must be added explicitly in encodings within a framework like LF, which only permits those variables that are in the current – possibly empty – context.

```
%sig BaseFOLMOD = {
  %include MLTT %open tm tp.
  %struct basemod : BaseMOD.
  i' : tp.
  non_empty_universe : tm i'.
}.
```

For all signatures of Fig. 2, there are morphisms into their counterparts in Fig. 5. For `NEGMOD`, `DISJMOD`, `ForallMOD` and `ExistsMOD`, these morphisms are induced by structure declarations. For the signatures `BaseMOD` and `BaseFOLMOD`, these are given by the following two views, which interpret the constants `o`, `ded` and `i` in terms of `o'`, `ded'`, and `i'`. Note that the view `BaseFOLMODView` reuses the translation of `o` and `ded` by using the view `BaseMODView` in the assignment to `base`. This has the effect that the rectangle made up of the morphism compositions `base BaseFOLMODView` and `BaseMODView basemod` commutes.

```
%view BaseMODView : Base -> BaseMOD = {
  o := tm o'.
  ded := [A] tm (ded' A).
}.
%view BaseFOLMODView : BaseFOL -> BaseFOLMOD = {
  %struct base := BaseMODView basemod.
  i := tm i'.
}.
```

For each logical connective and quantifier, the semantics is encoded by declaring two terms: One to encode when a formula containing the connective or quantifier is interpreted as true, i.e., when it is equal to the LF-term `1`, and one to encode when the formula is interpreted as false, i.e., when it is equal to the LF-term `0`.

As examples, we give the signatures `NEGMOD` and `ForallMOD` below. They are based on `BaseMOD` and `BaseFOLMOD`, respectively, and reuse the above views when importing from their counterparts `NEG` and `Forall`. `not1` axiomatizes that $\neg A$ is true if A is false, and `not0` axiomatizes that $\neg A$ is false if A is true. Similarly, `forall1` and `forall0` axiomatize the semantics of $\forall$. In the latter, the dependent product and sum types are used as meta-level universal and existential quantifiers: For example, the type `P ([x] (F x) ==' 1)` is inhabited iff for all `x` the type `(F x) ==' 1` is inhabited, and thus encodes the judgment that the truth value of $F(x)$ is 1 for all values of x.

```
%sig NEGMOD = {
  %include MLTT %open ==.
  %struct basemod : BaseMOD %open 1 0.
  %struct neg     : NEG = {%struct base := BaseMODView basemod} %open ¬.
  not1 : A == 0 -> (¬ A) == 1.
  not0 : A == 1 -> (¬ A) == 0.
```

```
}.
%sig ForallMOD = {
  %include MLTT %open ==' P' S'.
  %struct basefolmod : BaseFOLMOD %open 1 0.
  %struct univq      : Forall = {
    %struct basefol  := BaseFOLMODView basefolmod.} %open ∀.
  forall1 : P ([x] (F x) ==' 1) -> (∀ [x] F x) == 1.
  forall0 : S ([x] (F x) ==' 0) -> (∀ [x] F x) == 0.
}.
```

Finally, these signatures can be merged into PLMOD and FOLMOD in the same way as for syntax and proof theory above, e.g.:

```
%sig FOLMOD = {
  %include MLTT.
  %struct basefolmod : BaseFOLMOD.
  %struct plmod      : PLMOD = {%struct basemod := basefolmod.basemod.}.
  %struct univq      : ForallMOD = {%struct basefolmod := basefolmod.}.
  %struct existq     : ExistsMOD = {%struct basefolmod := basefolmod.}.
}.
```

4.5 Soundness

A view $v : S \to T$ from a signature S to a signature T maps every symbol s in S to an expression t in T, where the typing relation is preserved. The homomorphic extension of v maps all S-expressions (i.e., terms, types and kinds) to T-expressions. This means that a term $s : A$ of S is translated to a term $v(s) : v(A)$ in T.

We encode the soundness of FOL by means of a view from FOLPF to FOLMOD. This encoding follows the same modularity we used in order to encode the proof and model theory, i.e., we give separate views from each signature for proof theory in Fig. 3 to the respective signature for model theory in Fig. 5.

This is illustrated in Fig. 6 for the soundness of PL. The left and right diamonds in the diagram illustrate the encoding of proof and model theory, respectively, using single arrows to denote structures and double arrows to denote the views.

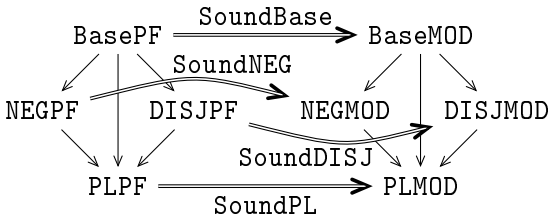
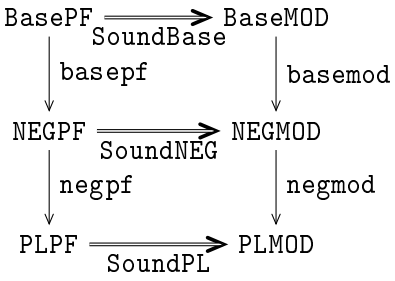


Fig. 6: Modular Encoding of Views for PL

For the sake of simplicity, we present the view SoundPL; the view from FOLPF to FOLMOD is encoded in a similar way. SoundPL encodes the soundness of PL by interpreting all expressions in PLPF in terms of the expressions in PLMOD. PLPF consists of expressions that come from BasePF, NEGPF and DISJPF, which are interpreted in BaseMOD, NEGMOD and DISJMOD via the views SoundBase, SoundNEG and SoundDISJ, respectively. Then SoundPL can be pieced together as follows, where we omit the proof of tertium non datur for brevity.

```
%view SoundPL : PLPF -> PLMOD = {
  %struct basepf  := SoundBase basemod.
  %struct negpf   := SoundNEG negmod.
  %struct disjpf  := SoundDISJ disjmod.
```

```
  tnd              := ...
}.
```



To see how this works, we explain the view `SoundNEG` in detail. The view `SoundPL` uses this view in the assignment `%struct negpf := SoundNEG negmod.`. Its semantics is that `negpf` as mapped by `SoundPL` is intended to be equal to the composition of the view `SoundNEG` and the structure `negmod`. This corresponds to the commutativity of the lower rectangle in the diagram above. In general, the Twelf implementation checks all the preconditions that are necessary to guarantee the commutativity.

`SoundNEG` interprets all the expressions of `NEGPF`, i.e., the structures `basepf`, `neg` and the terms `notI` and `notE`, in terms of the expressions of `NEGMOD`. The view `SoundBase` interprets `BasePF` in `BaseMOD`. This interpretation is carried over to `NEGMOD` via the composed morphism `SoundBase basemod`. Then the structure `basepf` is mapped to `SoundBase basemod`, which ensures the commutativity of the upper rectangle in the diagram on the right. The structure `neg` is mapped to the signature `neg` in `NEGMOD`.

```
%view SoundNEG : NEGPF -> NEGMOD  = {
  %struct basepf := SoundBase basemod.
  %struct neg    := neg.
  notI := [A][p : basemod.base.ded A -> {B} basemod.base.ded B]
             (MLTT..case (basemod.boole A)
                ([q : A == 1] p (basemod.1-ded q) (not A))
                ([q : A == 0] basemod.1-ded (not1 q))).
  notE := ...
}.
```

Finally, the proof rules are mapped to their soundness proofs. These proofs are straightforward in principle but quite complex to formalize. Therefore, we give only one example here and refer to [HR09] for the rest. `notI` is mapped to a proof term. It formalizes the following proof, where `1-ded : A ==' 1 -> ded A` is a lemma proved in the signature `BaseMOD` that establishes one half of an equivalence between a formula having truth value `1` and that formula's truth judgment:

`A`: Let A be a proposition.

`p`: Assume that given A is true, any proposition B is also true.

`MLTT..case (basemod.boole A)`: We do a case analysis on the truth value of A.

(Case 1) `q`: Assume $A = 1$.

`p (basemod.1-ded q) (not A)`: Then $\neg A = 1$ by instantiating B with $\neg A$ in `p`.

(Case 2) `q`: Assume $A = 0$.

`basemod.1-ded (not1 q)`: Then $\neg A = 1$ by `not1`.

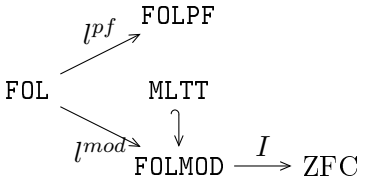
5 Representing Set-Theoretical Models

The representation of models given in Sect. 3 and employed in Sect. 4 uses LF to axiomatize the semantics of models of a logic L. The signature morphism l^{mod} interprets L-syntax in terms of the LF-signature L^{mod}, and the individual L-models

are represented as LF-models of that signature.

This can be unsatisfactory as the L-models are still represented as set-theoretical entities. It would be more appealing if L-models could be represented as LF-signatures themselves. This is indeed possible without changing the principal features of our approach: All we have to do is to refine the signature L^{mod} so much that it becomes (a syntactical representation of) specific L-models.

More precisely, we can axiomatize the particular set theory in which we want to express L-models as a part of L^{mod}. We can do this in such a way that every choice of some free parameters in the signature morphism l^{mod} corresponds to a specific L-model. Thus, we can represent L-models as certain LF-signature morphisms out of L^{syn}.



For our representation of FOL, we arrive at the diagram on the right. The signatures `FOL`, `FOLPF`, and `FOLMOD` are as before. Here we also make the signature `MLTT` that is included into `FOLMOD` explicit. Then we axiomatize Zermelo-Fraenkel set theory in the signature ZFC and give a view I that interprets `FOLMOD` in set theory. By composing l^{mod} and I, we obtain a signature morphism out of `FOL` that represents FOL-models. Note that the morphism of Sect. 4.5 proving soundness can be reused immediately to prove the soundness of models in ZFC.

Naturally, this approach requires a significant investment in order to represent set theory within a logical framework. The representation of ZFC is highly non-trivial and leads to lots of significant design questions. For example, very advanced encodings have been established in Mizar ([TB85]) and Isabelle ([Pau94]) employing sophisticated machine support. Even the proof of concept that we hand-coded in LF for this paper took over a week (which we consider in fact quite an achievement) to implement and still has some gaps. We will only sketch our encoding of ZFC and refer the reader to our full encoding for the details.

We encode ZFC as theory of a variant of first-order logic with universe Set, namely intuitionistic first-order logic with a description operator. The latter permits to define the basic operations such as union and replacement set, whose existence is only axiomatized indirectly in the ZFC axioms. This is different from Isabelle and Mizar where these operations are primitive.

After building up the basic operations on sets and the natural numbers, we introduce an operator $Elem : Set \to type$, which serves the purpose of lifting sets – which are LF terms of type Set – to the type level in order to employ typed reasoning. $Elem\ A$ can be regarded as an abbreviation of $\Sigma_{x:Set}(ded\ x \in A)$. Then, using proof-irrelevance, the LF terms of type $Elem\ A$ are in correspondence with the ZFC elements of the set A. This trick was inspired by the similar treatment in Scunak ([Bro06]).

Then it becomes possible to give a view from MLTT to ZFC, which is a prerequisite to obtain a view from `FOLMOD` to ZFC. For example, o is mapped to the set $\{0, 1\}$ and all connectives are mapped to the corresponding functions. An important technical detail of the views is the treatment of the universe of FOL, which is already

part of `FOL` and `FOLMOD`: the view I has a free parameter mapping the universe to an arbitrary set.

6 Conclusion

We took two developments on logical framework research to a practical test. Firstly, the module system for the logical framework LF and its implementation Twelf were designed to enhance the scalability of logic encodings ([RS09]). Secondly, the proof- and model-theoretical framework given by the second author in [Rab09].

Both were developed very recently, and our case study is the first scalability test for either one. We picked classical first-order logic and represented its syntax, proof theory, model theory, and soundness in LF. The most difficult part is the encoding of model theory and soundness, and we undertook two approaches to it that differ in the meta-language used to represent models.

Firstly, we defined models in Martin-Löf type theory. This is relatively simple to represent in LF, and it makes the reasoning about models manageably easy. On the down side, it does not correspond exactly to the usual paper definition of first-order models. This means that – depending on one's familiarity with LF – one might require adequacy proofs to trust that the representation indeed formalizes the right notion of models.

Secondly, we defined them in Zermelo-Fraenkel set theory. This permits a very direct representation of models. While it has the disadvantage of the higher investment in representing set theory in LF, it offers the additional advantage that even individual models can be represented, namely as signature morphisms from the signature representing the syntax to the signature representing set theory.

We evaluate both developments very favorably. The representation of first-order logic was straightforward and made easy by the module system. All logical symbols are treated separately so that the encodings can be used to piece together different logics. The representation of model theory feels elegant and appealing. We are already working on further encodings of intuitionistic logic, higher-order logic, and description logics. The only doubt we have is about the representation of set theory: Clearly, the lack of automated proving support in Twelf will prevent scalable applications. Therefore, we will investigate the possibilities of borrowing formalizations from other systems in the future. This will be supported by the logic-independent module system we designed in [Rab08], which permits to express cross-framework translations (e.g., LF to Isabelle) in terms of signature morphisms as well.

References

[AHMP98] B. Avron, F. Honsell, M. Miculan, and C. Paravano. Encoding modal logics in logical frameworks. *Studia Logica*, 60(1):161–208, 1998.

[AHMS99] S. Autexier, D. Hutter, H. Mantel, and A. Schairer. Towards an Evolutionary Formal Software-Development Using CASL. In D. Bert, C. Choppy, and P. Mosses, editors, *WADT*, volume 1827 of *Lecture Notes in Computer Science*, pages 73–88. Springer, 1999.

[AR09] S. Awodey and F. Rabe. Kripke Semantics for Martin-Löf Type Theory. 2009. Accepted at Conference on Typed Lambda Calculi and Applications (TLCA), see `http://kwarc.info/frabe/Research/LamKrip.pdf`.

[Bro06] C. Brown. Combining Type Theory and Untyped Set Theory. In U. Furbach and N. Shankar, editors, *International Joint Conference on Automated Reasoning*, pages 205–219. Springer, 2006.

[CF58] H. Curry and R. Feys. *Combinatory Logic*. North-Holland, Amsterdam, 1958.

[dB70] N. de Bruijn. The Mathematical Language AUTOMATH. In M. Laudet, editor, *Proceedings of the Symposium on Automated Demonstration*, volume 25 of *Lecture Notes in Mathematics*, pages 29–61. Springer, 1970.

[Dia06] R. Diaconescu. Proof systems for institutional logic. *Journal of Logic and Computation*, 16(3):339–357, 2006.

[GB92] J. Goguen and R. Burstall. Institutions: Abstract model theory for specification and programming. *Journal of the Association for Computing Machinery*, 39(1):95–146, 1992.

[GR02] J. A. Goguen and G. Rosu. Institution morphisms. *Formal Aspects of Computing*, 13:274–307, 2002.

[HHP93] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, 1993.

[How80] W. Howard. The formulas-as-types notion of construction. In *To H.B. Curry: Essays on Combinatory Logic, Lambda-Calculus and Formalism*, pages 479–490. Academic Press, 1980.

[HR09] Feryal Fulya Horozal and Florian Rabe. Twelf Encoding of the Soundness of FOL, 2009. https://svn.kwarc.info/repos/twelf/soundness.

[HST94] R. Harper, D. Sannella, and A. Tarlecki. Structured presentations and logic representations. *Annals of Pure and Applied Logic*, 67:113–160, 1994.

[Lan98] S. Mac Lane. *Categories for the working mathematician*. Springer, 1998.

[Mes89] J. Meseguer. General logics. In H.-D. Ebbinghaus et al., editors, *Proceedings, Logic Colloquium, 1987*, pages 275–329. North-Holland, 1989.

[MGDT05] T. Mossakowski, J. Goguen, R. Diaconescu, and A. Tarlecki. What is a logic? In J. Béziau, editor, *Logica Universalis*, pages 113–133. Birkhäuser Verlag, 2005.

[ML74] P. Martin-Löf. An Intuitionistic Theory of Types: Predicative Part. In *Proceedings of the '73 Logic Colloquium*. North-Holland, 1974.

[ML96] P. Martin-Löf. On the meanings of the logical constants and the justifications of the logical laws. *Nordic Journal of Philosophical Logic*, 1(1):3–10, 1996.

[NSM01] P. Naumov, M. Stehr, and J. Meseguer. The HOL/NuPRL proof translator - a practical approach to formal interoperability. In *14th International Conference on Theorem Proving in Higher Order Logics*. Springer, 2001.

[Pau94] L. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer, 1994.

[Pfe00] F. Pfenning. Structural cut elimination: I. intuitionistic and classical logic. *Information and Computation*, 157(1-2):84–141, 2000.

[Rab08] F. Rabe. *Representing Logics and Logic Translations*. PhD thesis, Jacobs University Bremen, 2008.

[Rab09] F. Rabe. A Logical Framework Combining Model and Proof Theory. Submitted to Conference on Algebra and Coalgebra in Computer Science (CALCO) see http://kwarc.info/frabe/Research/rabe_combining_09.pdf, 2009.

[RS09] F. Rabe and C. Schürmann. A Practical Module System for LF. Submitted to Conference on Automated Deduction (CADE), see http://kwarc.info/frabe/Research/lf.pdf, 2009.

[SW83] D. Sannella and M. Wirsing. A Kernel Language for Algebraic Specification and Implementation. In M. Karpinski, editor, *Fundamentals of Computation Theoryational*, pages 413–427. Springer, 1983.

[Tar33] A. Tarski. Pojęcie prawdy w językach nauk dedukcyjnych. *Prace Towarzystwa Naukowego Warszawskiego Wydzial III Nauk Matematyczno-Fizycznych*, 34, 1933. English title: The concept of truth in the languages of the deductive sciences.

[TB85] A. Trybulec and H. Blair. Computer Assisted Reasoning with MIZAR. In A. Joshi, editor, *Proceedings of the 9th International Joint Conference on Artificial Intelligence*, pages 26–28, 1985.

[TV56] A. Tarski and R. Vaught. Arithmetical extensions of relational systems. *Compositio Mathematica*, 13:81–102, 1956.