# Towards Logical Frameworks in the Heterogeneous Tool Set Hets

Mihai Codescu[1], Fulya Horozal[2], Michael Kohlhase[2], Till Mossakowski[1], Florian Rabe[2], Kristina Sojakova[3]

[1] DFKI GmbH, Bremen, Germany,
[2] Computer Science, Jacobs University, Bremen, Germany,
[3] Carnegie Mellon University, Pittsburgh, USA

**Abstract.** LF is a meta-logical framework that has become a standard tool for representing logics and studying their properties. Its focus is proof theoretic, employing the Curry-Howard isomorphism: propositions are represented as types, and proofs as terms.
Hets is an integration tool for logics, logic translations and provers, with a model theoretic focus, based on the meta-framework of institutions, a formalisation of the notion of logical system.
In this work, we combine these two worlds. The benefit for LF is that logics represented in LF can be (via Hets) easily connected to various interactive and automated theorem provers, model finders, model checkers, and conservativity checkers - thus providing much more efficient proof support than mere proof checking as is done by systems like Twelf. The benefit for Hets is that (via LF) logics become represented formally, and hence trustworthiness of the implementation of logics is increased, and correctness of logic translations can be mechanically verified. Moreover, since logics and logic translations are now represented declaratively, the effort of adding new logics or translations to Hets is greatly reduced.
This work is part of a larger effort of building an atlas of logics and translations used in computer science and mathematics.

## 1 Introduction

There is a large manifold of different logical systems used in computer science, such as propositional, first-order, higher-order, modal, description, temporal logics, and many more. These logical systems are supported by software, like e.g. (semi-)automated theorem provers, model checkers, computer algebra systems, constraint solvers, or concept classifiers, and each of these software systems comes with different foundational assumptions and input languages, which makes them non-interoperable and difficult to compare and evaluate in practice.

There are two main approaches to remedy this situation. The model theoretic approach of *institutions* [GB92,Mes89] provides a formalisation of the notion of logical system. The benefit is that a large body of meta-theory can be developed independent of the specific logical system, including specification languages for structuring large logical theories. Recently, even a good part of model theory has been generalised to this setting [Dia08]. Moreover, the Heterogeneous Tool Set (Hets, [MML07]) provides an institution-independent software interface, such that a heterogeneous proof management involving different tools (as listed above) is practically realised. In Hets, logic translations, formalized as so-called institution comorphisms, become first-class citizens. Heterogeneous specification and proof management is done relative to a graph of logics and translations.

The proof theoretic approach of logical frameworks starts with one "universal" logic that is used as a *logical framework*. This is used for representing logics as theories (in the "universal" logic of the framework). For instance, the Edinburgh Logical Framework LF [HHP93] has been used extensively to represent logics [HST94,PSK$^+$03,AHMP98], many of them included in the Twelf distribution [PS99]. Logic representations in Isabelle [Pau94] are notable for the size of the libraries in the encoded logics, especially for HOL [NPW02]. Logic representations in rewriting logic [MOM94] using the Maude system [CELM96] include the examples of equational logic, Horn logic and linear logic. A notable property of rewriting logic is *reflection* i.e. one can represent rewriting logic within itself. Zermelo-Fraenkel and related set theories were encoded in a number of systems, see, e.g., [PC93] or [TB85]. Other systems employed to encode logics include Coq [BC04], Agda [Nor05], and Nuprl [CAB$^+$86]. Only few logic *translations* have been formalized systematically in this setting. Important translations represented using the logic programming interpretation of LF include cut elimination [Pfe00] and the HOL-Nurpl translation [SS04]. The latter guided the design of the Delphin system [PS08] for logic translations.

Both approaches provide the theoretical and practical infrastructure to define logics. However, there are two major differences. Firstly, Hets is based on model theory – the semantics of implemented logics and the correctness of translations are determined by model theoretic arguments. Proof theory is only used as a tool to discharge proof obligations and is not represented explicitly.

Secondly, the logics of Hets are specified on the meta-level rather than within the system itself. Each logic or logic translation has to be specified by implementing a Haskell interface that is part of the Hets code, and tools for parsing and static analysis have to be provided. Consequently, only Hets developers but not users can add them. Besides the obvious disadvantage of the cost involved when adding logics, this representation does not provide us with a way to reason about the logics or their translations themselves. In particular, each logic's static analysis is part of the trusted code base, and the translations cannot be automatically verified for correctness.

The work reported here is part of the ongoing project LATIN (Logic Atlas and Integrator, [KMR09]). LATIN has two main goals: to *fully integrate proof and model theoretic frameworks* described above preserving their respective advantages, and to create *modular formalizations of commonly used logics* together with *logic morphisms interrelating them*: the **Logic Atlas**. To this end, we develop general definition of a logical framework (the **LATIN metaframework** that covers logical frameworks such as LF, Isabelle, and rewriting logic and implement it in Hets. The LATIN metaframework follows a "logics as theories/translations as morphisms" approach such that a theory graph in a logical framework leads to a graph of institutions and comorphisms via a general construction. This means that new logics can now be added to Hets in a purely declarative way. Moreover, the declarative nature means that logics themselves are no longer only formulated in the semi-formal language of mathematics, but now are fully formal objects, such that one can reason about them (e.g. prove soundness of proof systems or logic translations) within proof systems like Twelf.

This paper is organized as follows. We give introductions to the model and proof theoretic approaches and the LATIN Atlas in Sect. 2. We introduce the LATIN metaframework in Sect. 3 and describe its integration into the Hets system in Sect. 4. We will use an encoding of first-order logic in the logical framework LF as a running example.

## 2 Preliminaries

### 2.1 The Heterogeneous Tool Set

The Heterogeneous Tool Set (Hets, [MML07]) is a set of tools for multi-logic specifications, which combines parsers, static analyzers, and theorem provers. Hets provides a heterogeneous specification language built on top of CASL [ABK$^+$02] and uses the development graph calculus [MAH06] as a proof management component. The graph of logics supported by Hets and their translations is presented in Fig. 1.

Hets formalizes the logics and their translations using the abstract model theory notions of institutions and institution comorphisms (see [GB92]).

**Definition 1.** *An institution is a quadruple $I = (\mathbf{Sig}, \mathbf{Sen}, \mathbf{Mod}, \models)$ where:*

- $\mathbf{Sig}$ *is a category of* signatures*;*
- $\mathbf{Sen} : \mathbf{Sig} \to Set$ *is a functor to the category Set of small sets and functions, giving for each signature $\Sigma$ its set of* sentences $\mathbf{Sen}(\Sigma)$ *and for any signature morphism $\varphi : \Sigma \to \Sigma'$ the* sentence translation *function* $\mathbf{Sen}(\varphi) : \mathbf{Sen}(\Sigma) \to \mathbf{Sen}(\Sigma')$ *(denoted by a slight abuse also $\varphi$);*
- $\mathbf{Mod} : \mathbf{Sig}^{op} \to Cat$ *is a functor to the category of categories and functors Cat* [4] *giving for any signature $\Sigma$ its category of models $\mathbf{Mod}(\Sigma)$ and for any signature morphism $\varphi : \Sigma \to \Sigma'$ the* model reduct *functor* $\mathbf{Mod}(\varphi) : \mathbf{Mod}(\Sigma') \to \mathbf{Mod}(\Sigma)$ *(denoted $\_|_\varphi$);*
- *a satisfaction relation $\models_\Sigma \subseteq |\mathbf{Mod}(\Sigma)| \times \mathbf{Sen}(\Sigma)$ for each signature $\Sigma$*

*such that the following satisfaction condition holds:*

$$M'|_\varphi \models_{\Sigma'} e \Leftrightarrow M' \models_\Sigma \varphi(e)$$

*for each $M' \in |\mathbf{Mod}(\Sigma')|$ and $e \in \mathbf{Sen}(\Sigma)$, expressing that truth is invariant under change of notation and context.*

[4] We disregard here the foundational issues, but notice however that *Cat* is actually a so-called quasi-category.

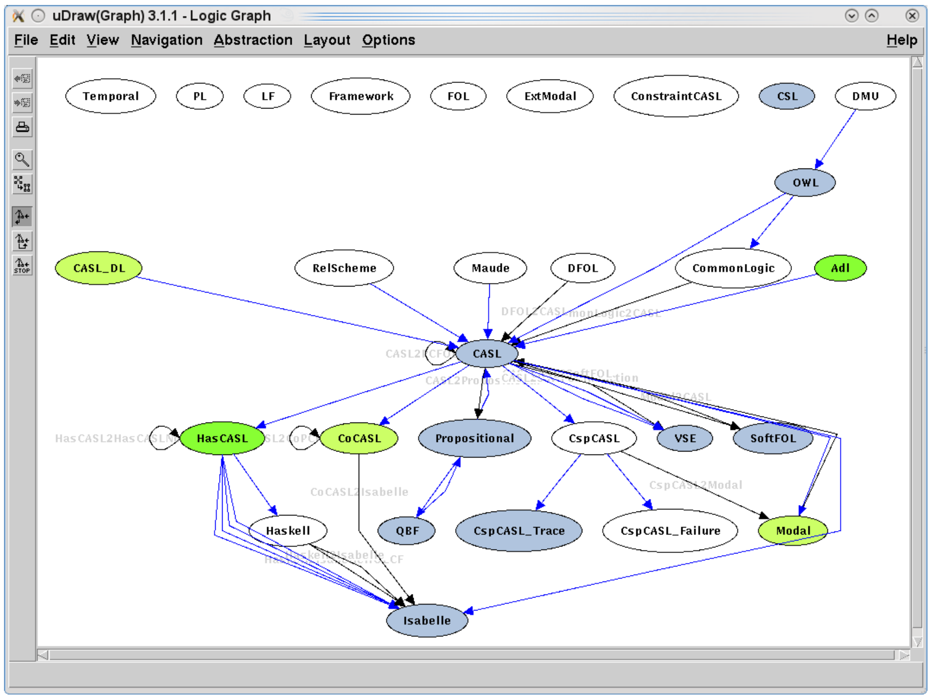


**Fig. 1.** Hets logic graph

For example, the institution of unsorted first-order logic $\mathbb{FOL}$ has signatures consisting of a set of function symbols and a set of predicate symbols, with their arities. Signature morphisms map symbols such that their arities are preserved. Models are first-order structures, and sentences are first-order formulas. Sentence translation means replacement of the translated symbols. Model reduct means reassembling the model's components according to the signature morphism. Satisfaction is the usual satisfaction of a first-order sentence in a first-order structure.

**Definition 2.** *Given two institutions $I_1$, $I_2$ with $I_i = (\mathbf{Sig}_i, \mathbf{Sen}_i, \mathbf{Mod}_i, \models^i)$, an institution comorphism from $I_1$ to $I_2$ consists of a functor $\Phi : \mathbf{Sig}_1 \to \mathbf{Sig}_2$ and natural transformations $\beta : \mathbf{Mod}_2 \circ \Phi \Rightarrow \mathbf{Mod}_1$ and $\alpha : \mathbf{Sen}_1 \Rightarrow \mathbf{Sen}_2 \circ \Phi$, such that the following satisfaction condition holds:*

$$M' \models^2_{\Phi(\Sigma)} \alpha_\Sigma(e) \iff \beta_\Sigma(M') \models^1_\Sigma e,$$

*where $\Sigma$ is an $I_1$ signature, $e$ is a $\Sigma$-sentence in $I_1$ and $M'$ is a $\Phi(\Sigma)$-model in $I_2$.*

The process of extending Hets with a new logic can be summarized as follows. First, we need to provide Haskell datatypes for the constituents of the logic, e.g. signatures, morphisms and sentences. This is done via instantiating various Haskell type classes, namely *Category* (for the signature category of the institution), *Sentences* (for the sentences), *Syntax* (for abstract syntax of basic specifications, and a parser transforming input text into this abstract syntax), *StaticAnalysis* (for the static analysis, turning basic specifications into theories, where a theory is a signature and a set of sentences). All this is assembled in the type class *Logic*, which additionally provides logic-specific tools like provers and model finders. For displaying the output of model finders, also (finite) models are represented in Hets, and these can even be translated against comorphisms. The model theoretic foundation of Hets also is apparent from the fact that *StaticAnalysis* contains methods for checking amalgamability properties that are defined model theoretically (and therefore not available in purely proof theoretic logical frameworks). The type class *Logic* is used to represent logics in Hets internally. Finally, the new logic is made available by adding it to the list of Hets' known logics. Similarly, Hets represents comorphisms as instances of a type class *Comorphism*, which provides an interface for translating constituents of the source logic to the target logic of the comorphism. Notice that the domain of the translation can be restricted to a certain sublogic of the source using its sublogics hierarchy. Moreover, the methods of the class *Comorphism* include translation of theories, signature morphisms or sentences to the target logic.

The input language of Hets is HetCASL. It combines logic-specific syntax of basic specifications (as specified by an instance of *Syntax*) with the logic-independent structuring constructs of CASL (like extension, union, translation of specifications, or hiding parts). Moreover, there are constructs for choosing a particular logic, as well as for translating a specification along an institution comorphism.

## 2.2 Proof Theoretic Logical Frameworks

We use the term *proof theoretic* to refer to logical frameworks whose semantics is or can be given in a formal and thus mechanizable way without reference to a Platonic universe. These frameworks are declarative formal languages with an inference system defining a consequence relation between judgments. They come with a notion of language extensions called signatures or theories, which admits the structure of a category. Logic encodings represent the syntax and proof theory of a logic as a theory of the logical framework, and logical consequence is represented in terms of the consequence relation of the framework.

The most important logical frameworks are LF, Isabelle, and rewriting logic. LF [HHP93] is based on dependent type theory; logics are encoded as LF signatures, proofs as terms using the Curry-Howard correspondences, and consequence between formulas as type inhabitation. The main implementation is Twelf [PS99]. The Isabelle system [Pau94] implements higher-order logic [Chu40]; logics are represented as HOL theories, and consequence between formulas as HOL propositions. The Maude system [CELM96] is related to rewriting logic [MOM94]; logics are represented as rewrite theories, and consequence between formulas as rewrite judgments. Other languages such as Coq [BC04] or Agda [Nor05] can be used as logical frameworks as well, but this is not the primary application encountered in practice.

In the following, we give an overview of **LF**, which we will use as a running example. LF extends simple type theory with dependent function types and is related to Martin-Löf type theory [ML74]. The following grammar is a simplified version of the LF grammar where we write $\cdot$ for the empty list. It includes LF signature morphisms, which were added to LF in [HST94] and added to Twelf in [RS09]:

$$\begin{array}{lll} \text{Signatures} & \Sigma & ::= \cdot \mid \Sigma,\ c : E \mid \Sigma,\ c : E = E \\ \text{Morphisms} & \sigma & ::= \cdot \mid \sigma,\ c := E \\ \text{Expressions} & E & ::= \texttt{type} \mid c \mid x \mid E\ E \mid \lambda_{x:E}\, E \mid \Pi_{x:E}\, E \mid E \to E \end{array}$$

LF **expressions** $E$ are grouped into kinds $K$, kinded type-families $A : K$, and typed terms $t : A$. The kinds are the base kind `type` and the dependent function kinds $\Pi_{x:A}\, K$. The type families are the constants $a$, applications $a\ t$, and the dependent function type $\Pi_{x:A}\, B$; type families of kind `type` are called types. The terms are constants $c$, applications $t\ t'$, and abstractions $\lambda_{x:A}\, t$. We write $A \to B$ instead of $\Pi_{x:A}\, B$ if $x$ does not occur in $B$.

An LF **signature** $\Sigma$ is a list of kinded type family declarations $a : K$ and typed constant declarations $c : A$. Both may carry definitions, i.e., $c : A = t$ and $a : K = A$, respectively. Due to the Curry-Howard representation, propositions are encoded as types as well; hence a constant declaration $c : A$ may be regarded as an axiom $A$, while $c : A = t$ additionally provides a proof $t$ for $A$. Hence, an LF signature corresponds to what usually is called a logical *theory*.

Relative to a signature $\Sigma$, closed expressions are related by the judgments $\vdash_\Sigma E : E'$ and $\vdash_\Sigma E = E'$. Equality of terms, type families, and kinds are defined by $\alpha\beta\eta$-equality. All judgments for typing, kinding, and equality are decidable.

Given two signatures $\Sigma$ and $\Sigma'$, an LF **signature morphism** $\sigma : \Sigma \to \Sigma'$ is a typing- and kinding-preserving map of $\Sigma$-symbols to $\Sigma'$-expressions. Thus, $\sigma$ maps every constant $c : A$ of $\Sigma$ to a term $\sigma(c) : \overline{\sigma}(A)$ and every type family symbol $a : K$ to a type family $\sigma(a) : \overline{\sigma}(K)$. Here, $\overline{\sigma}$ is the homomorphic extension of $\sigma$ to $\Sigma$-expressions, and we will write $\sigma$ instead of $\overline{\sigma}$ from now on.

Signature morphisms preserve typing, i.e., if $\vdash_\Sigma E : E'$, then $\vdash_{\Sigma'} \sigma(E) : \sigma(E')$, and correspondingly for kinding and equality. Due to the Curry-Howard encoding of axioms, this corresponds to theorem preservation of theory morphisms. Composition and identity are defined in the obvious way, and we obtain a category $\mathbb{LF}$.

In [RS09], a **module system** was given for LF and implemented in Twelf. The module system permits to build both signatures and signature morphisms in a structured way. Its expressivity is similar to that of development graphs [AHMS99].

## 2.3 A Logic Atlas in LF

In the LATIN project [KMR09], we aim at the creation of a logic atlas based on LF. The Logic Atlas is a multi-graph of LF signatures and morphisms between them. Currently it contains formalizations of various logics, type theories, foundations of mathematics, algebra, and category theory.

Among the logics formalized in the Atlas are propositional (*PL*), first (*FOL*) and higher-order logic (*HOL*), sorted (*SFOL*) and dependent first-order logic

(*DFOL*), description logics (*DL*), modal (*ML*) and common logic (*CL*) as illustrated in the diagram below. Single arrows ($\rightarrow$) in this diagram denote translations between formalizations and hooked arrows ($\hookrightarrow$) denote imports. Among the foundations are encodings of Zermelo-Fraenkel set theory, Isabelle's higher-order logic, and Mizar's Set theory [IR11].

Note that a logical framework leaves the choice of the foundation deliberately open. In this way, we can use one logical framework (e.g. LF) with several foundations (e.g. ZFC, as well as category theory). Only the representation of a logic includes the choice of a foundation.

Actually the graph is significantly more complex as we use the LF module system to obtain a maximally modular design of logics. For example, first-order, modal, and description logics are formed from orthogonal modules for the individual connectives, quantifiers, and axioms. For example, the $\wedge$ connective is only declared once in the whole Atlas and imported into the various logics and foundations and related to the type theoretic product via the Curry-Howard correspondence.

$$\begin{tikzcd}
 & & PL \arrow[dl, hook] \arrow[dd, hook] \arrow[dr, hook] \arrow[drr, hook] & & \\
 & ML \arrow[dr] & & SFOL \arrow[r] \arrow[dl, bend left] \arrow[dd] & DFOL \arrow[ddddl] \\
DL \arrow[rr] \arrow[ddr] & & FOL \arrow[ur, hook, bend left] \arrow[dd] \arrow[dddr, hook] & & \\
 & & & HOL \arrow[dd] & \\
 & OWL \arrow[uur] & CL & & \\
 & & Isabelle \arrow[r] & ZFC \arrow[r] & Mizar
\end{tikzcd}$$

Moreover, we use individual modules for syntax, proof theory and model theory so that the same syntax can be combined with different interpretations. For example, our formalization of first-order logic (presented in [HR11]) consists of the signatures *Base* and $FOL^{Syn}$ for syntax, $FOL^{Pf}$ for proof theory, and $FOL^{Mod}$ for model theory as illustrated in the diagram on the right. *Base* contains declarations $o$ : `type` and $i$ : `type` for the type of formulas and first-order individuals, and a truth judgment for formulas. $FOL^{Syn}$ contains declarations for all logical connectives and quantifiers (see Fig. 4). $FOL^{truth}$ is an inclusion morphism from *Base* to $FOL^{Syn}$. $FOL^{Pf}$ consists of declarations for judgments and inference rules associated with each logical symbol declared in $FOL^{Syn}$. $FOL^{pf}$ is simply an inclusion morphism from $FOL^{Syn}$ to $FOL^{Pf}$.

$$\begin{tikzcd}
 & & FOL^{Pf} \arrow[dd, "FOL^{sound}"] \\
Base \arrow[r, hook, "FOL^{truth}"] & FOL^{Syn} \arrow[ur, hook, "FOL^{pf}"] \arrow[dr, "FOL^{mod}"'] & \\
 & & FOL^{Mod} \arrow[d, shift right, "M"'] \\
 & & ZFC \arrow[u, hook, shift right]
\end{tikzcd}$$

For the representation of FOL model theory, LF is not a suitable metalanguage because its type theory is minimalistic and the use of higher-order abstract syntax is incompatible with the natural way of adding computational support needed to express models. However, LF can serve as a minimal, neutral framework to formalize the metalanguage itself. We choose ZFC set theory as the appropriate metalanguage because it is the standard foundation of mathematics, and formalize it in LF (in the signature *ZFC*) and use it as the metalanguage to define models.

The *ZFC* encoding includes the type of sets, the membership predicate as a primitive non-logical symbol, and the usual ZFC set operations and axioms defined in a first-order language with description operator. Additionally, *ZFC* contains a type judgment *elem* for the elements of a set as well as a binary operation $\Longrightarrow$ on sets that returns the set of functions. This is important for being able to represent models as signature morphisms (see below): signature morphisms map types to types, and via *elem*, (carrier) sets can be turned into types.

$FOL^{Mod}$ includes *ZFC* as a metalanguage and uses it to axiomatize the properties of FOL-models. More precisely, $FOL^{Mod}$ declares a set *bool* for the boolean values axiomatizing it to get the desired 2-element set $\{0, 1\}$, declares a fixed set *univ* of individuals, along with an axiom stating that the universe is nonempty. For each logical symbol $s^{Syn}$ in $FOL^{Syn}$, $FOL^{Mod}$ declares a symbol $s^{Mod}$ that represents the semantic operation used to interpret $s^{Syn}$ along with axioms specifying its truth values. For instance, for disjunction, which is declared as $or : o \to o \to o$ in $FOL^{Syn}$, $FOL^{Mod}$ declares the symbol $\vee$ as a ZFC-function from $bool^2$ to *bool* and axiomatizes it to be the binary supremum in the boolean 2-element lattice. This corresponds to the case-based definition of the semantics of a formula.

| $FOL^{Syn}$ | $FOL^{Mod}$ | $ZFC$ |
|---|---|---|
| $i : \texttt{type}$ | $univ : set$ | $set : \texttt{type}$ |
| $o : \texttt{type}$ | $bool : set$ | $prop : \texttt{type}$ |
| $or : o \to o \to o$ | $\vee : elem\,(bool \Longrightarrow bool \Longrightarrow bool)$ | $\vee : prop \to prop \to prop$ |
| $forall : (i \to o) \to o$ | $\forall : elem\,((univ \Longrightarrow bool) \Longrightarrow bool)$ | $\forall : (set \to prop) \to prop$ |
| | | $\in : set \to set \to prop$ |
| | | $elem : set \to \texttt{type}$ |
| | | $\Longrightarrow : set \to set \to set$ |

The morphism $FOL^{mod}$ interprets the syntax of FOL in the semantic realm specified by $FOL^{Mod}$: It maps the type $i$ of individuals to the type of elements of *univ*, the type $o$ of formulas to the type of elements of *bool*, and the logical operations to the corresponding operations on booleans.

The individual FOL-models are represented as LF signature morphisms from $FOL^{Mod}$ to *ZFC* that are the identity on *ZFC*. In other words, a model $M$ maps *univ* to a nonempty set expressed by using the set operations of *ZFC*. $M$ interprets the boolean operations in $FOL^{Mod}$ in terms of the usual set operations in *ZFC*. For instance, the universal quantification for the booleans is mapped to the intersection of a family of subsets. Given such a morphism $M$, the composition $FOL^{mod}\,;M$ then yields the interpretation of $FOL^{Syn}$ in *ZFC*.

A particular aspect of our formalization is that soundness of FOL can be represented naturally as an LF signature morphism from $FOL^{Pf}$ to $FOL^{Mod}$ making the diagram above commute. Note that a morphisms in the opposite direction, i.e., from $FOL^{Mod}$ to $FOL^{Pf}$, does not yield completeness.

## 3 The LATIN Metaframework

In this section we describe the theoretical background of our LATIN metaframework (LMF) based on the approach taken in [Rab10]. The LMF is an abstract framework that allows to represent logical frameworks as declarative languages given by categories of theories. The LMF is generic in the sense that it can be instantiated with specific logical frameworks such as LF, Isabelle or rewriting logic, thus allowing Hets to be flexible in the choice of the logical framework in which logics should be represented.

In Sect. 3.1, we show that our abstract representation of logical frameworks complies with the notion of institutions and institution comorphisms. Here we deliberately restrict attention to a special case of [Rab10] that makes the ideas clearest and discuss generalizations in Sect. 3.2.

### 3.1 Main Definition

**Definition 3 (Inclusions).** *A category with inclusions consists of a category together with a broad subcategory that is a partial order. We write $B \hookrightarrow C$ for the inclusion morphism from $B$ to $C$.*

**Definition 4 (Logical Framework).** *A tuple $(\mathbb{C}, Base, \mathbf{Sen}, \vdash)$ is a logical framework if*

- *$\mathbb{C}$ is a category that has inclusions and pushouts along inclusions,*
- *$Base$ is an object of $\mathbb{C}$,*
- *$\mathbf{Sen} : \mathbb{C}\backslash Base \to Set$ is a functor, where $\mathbb{C}\backslash Base$ is the so-called slice category of $\mathbb{C}$ over Base, whose objects are arrows in $\mathbb{C}$ of source Base and morphisms make triangles commute,*
- *for $t \in \mathbb{C}\backslash Base$, $\vdash_t$ is a unary predicate on $\mathbf{Sen}(t)$,*
- *$\vdash$ is preserved under signature morphisms: if $\vdash_t F$ then $\vdash_{t'} \mathbf{Sen}(\sigma)(F)$ for any morphism $\sigma : t \to t'$ in $\mathbb{C}\backslash Base$.*

$\mathbb{C}$ is the category of theories of the logical framework. Our focus is on declarative frameworks where theories are lists of named declarations. Typically these have inclusions and pushouts along them in a natural way.

Logics are encoded as theories $\Sigma$ of the framework, but not all theories can be naturally regarded as logic encodings. Logic encodings must additionally distinguish certain objects over $\Sigma$ that encode logical notions. Therefore, we consider $\mathbb{C}$-morphisms $t : Base \to \Sigma$ where $Base$ makes precise what objects must be distinguished.

We leave the structure of $Base$ abstract, but we require that slices $t : Base \to \Sigma$ provide at least a notion of sentences and truth for the logic encoded by $\Sigma$. Therefore, $\mathbf{Sen}(t)$ gives the set of sentences, and the predicate $\vdash_t F$ expresses the truth of $F$.

*Example 1 (LF).* We define a logical framework $\mathbb{F}^{LF}$ based on the category $\mathbb{C} = \mathbb{LF}$. $\mathbb{LF}$ has inclusions by taking the subset relation between sets of declarations. Given $\sigma : \Sigma \to \Sigma'$ and an inclusion $\Sigma \hookrightarrow \Sigma, c : A$, a pushout is given by

$$(\sigma,\ c := c) \ \ : \ \ (\Sigma,\ c : A) \ \ \to \ \ (\Sigma',\ c : \sigma(A))$$

(except for possibly renaming $c$ if it is not fresh for $\Sigma'$). The pushouts for other inclusions are obtained accordingly.

*Base* is the signature with the declarations $o$ : `type` and $ded : o \to$ `type`. For every slice $t : Base \to \Sigma$, we define $\mathbf{Sen}(t)$ as the set of closed $\beta\eta$-normal LF-terms of type $t(o)$ over the signature $\Sigma$. Moreover, $\vdash_t F$ holds iff the $\Sigma$-type $t(ded)\ F$ is inhabited.

Given $t : Base \to \Sigma$ and $t' : Base \to \Sigma'$ and $\sigma : \Sigma \to \Sigma'$ such that $\sigma \circ t = t'$, we define the sentence translation by $\mathbf{Sen}(\sigma)(F) = \sigma(F)$. Truth is preserved: assume $\vdash_t F$; thus $t(ded)\ F$ is inhabited over $\Sigma$; then $\sigma(t(ded)\ F) = t'(ded)\ \sigma(F)$ is inhabited over $\Sigma'$; thus $\vdash_{t'} \mathbf{Sen}(\sigma)(F)$.

*Example 2 (Isabelle).* A logical framework based on Isabelle is defined similarly. $\mathbb{C}$ is the category of Isabelle theories and theory morphisms (for the latter, see [BJL06]). *Base* consists of the declarations $bool : type$ and `trueprop` $: bool \to$ `prop` where `prop` is the type of Isabelle propositions. Given $t : Base \to \Sigma$, we define $\mathbf{Sen}(t)$ as the set of $\Sigma$-terms of type $t(bool)$, and $\vdash_t F$ holds if $t($`trueprop`$)\ F$ is an Isabelle theorem over $\Sigma$.

*Example 3 (Rewriting logic).* A logical framework based on rewriting logic can be defined along the lines of [MOM94]. $\mathbb{C}$ is the category of rewriting logic theories and theory morphisms. *Base* consists of the following declarations:

```
sorts Prop FormList Sequent .
subsorts Prop < FormList .
op empty : -> FormList .
op tt : -> Prop .
op __⊢__ : FormList FormList -> Sequent .
```

where `Prop` stands for the type of propositions, `tt` for the formula *true*, and $\vdash$ turns two lists of formulas into a sequent. Given $t : Base \to \Sigma$, we define $\mathbf{Sen}(t)$ as the set of $\Sigma$-terms of type $t($`Prop`$)$, and $\vdash_t F$ holds for some term $F$ of type $t($`Prop`$)$ if `empty` $\vdash F \Rightarrow_\Sigma$ `empty` $\vdash$ `tt`. $\vdash_t$ is preserved by rewriting logic theory morphisms because rewriting must be preserved.

We use logical frameworks to define institutions. The basic idea is that slices $t : Base \to L^{Syn}$ define logics ($L^{Syn}$ specifies the syntax of the logic), signatures of that logic are extensions $L^{Syn} \hookrightarrow \Sigma^{Syn}$, and sentences and truth are given by $\mathbf{Sen}$ and $\vdash$. We could represent the logic's models in terms of the models of the logical framework, but that would complicate the mechanizable representation of models. Therefore, we represent models as $\mathbb{C}$ morphisms into a fixed theory that represents the foundation of mathematics. We need one auxiliary definition to state this precisely:

**Definition 5.** *Fix a logical framework, and assume $L^{mod} : L^{Syn} \to L^{Mod}$ in $\mathbb{C}$ as in the diagram below.*

$$\begin{tikzcd}
L^{Mod} \arrow[r, hook] \arrow[rr, hook, bend left] & \Sigma^{Mod} \arrow[r, "\sigma^{mod}"'] & \Sigma'^{Mod} \\
L^{Syn} \arrow[u, "L^{mod}"] \arrow[r, hook] \arrow[rr, hook, bend right] & \Sigma^{Syn} \arrow[u, "\Sigma^{mod}"] \arrow[r, "\sigma^{syn}"] & \Sigma'^{Syn} \arrow[u, "\Sigma'^{mod}"']
\end{tikzcd}$$

*Firstly, for every inclusion $L^{Syn} \hookrightarrow \Sigma^{Syn}$, we define $\Sigma^{Mod}$ and $\Sigma^{mod}$ such that $\Sigma^{Mod}$ is a pushout. Secondly, for every $\sigma^{syn} : \Sigma^{Syn} \to \Sigma'^{Syn}$, we define $\sigma^{mod} : \Sigma^{Mod} \to \Sigma'^{Mod}$ as the unique morphism such that the above diagram commutes.*

Then we are ready for our main definition:

**Definition 6 (Institutions in LMF).** *Let $\mathbb{F} = (\mathbb{C}, Base, \mathbf{Sen}, \vdash)$ be a logical framework. Assume $L = (L^{Syn}, L^{truth}, L^{Mod}, \mathcal{F}, L^{mod})$ as in the following diagram:*

$$\begin{tikzcd}
 & \mathcal{F} \arrow[r, "id_{\mathcal{F}}"] \arrow[d, hook] & \mathcal{F} & \\
 & L^{Mod} \arrow[r, hook] & \Sigma^{Mod} \arrow[u, "m"] \arrow[r, "\sigma^{mod}"'] & \Sigma'^{Mod} \arrow[ul, "m'"'] \\
Base \arrow[r, "L^{truth}"] & L^{Syn} \arrow[u, "L^{mod}"] \arrow[r, hook] & \Sigma^{Syn} \arrow[u, "\Sigma^{mod}"] \arrow[r, "\sigma^{syn}"] & \Sigma'^{Syn} \arrow[u, "\Sigma'^{mod}"]
\end{tikzcd}$$

*Then we define the institution $\mathbb{F}(L) = (\mathbf{Sig}^L, \mathbf{Sen}^L, \mathbf{Mod}^L, \models^L)$ as follows:*

- $\mathbf{Sig}^L$ *is the full subcategory of $\mathbb{C}\backslash L^{Syn}$ whose objects are inclusions. To simplify the notation, we will write $\Sigma^{Syn}$ for an inclusion $L^{Syn} \hookrightarrow \Sigma^{Syn}$ below.*
- $\mathbf{Sen}^L$ *is defined by*

$$\mathbf{Sen}^L(\Sigma^{Syn}) = \mathbf{Sen}((L^{Syn} \hookrightarrow \Sigma^{Syn}) \circ L^{truth}) \qquad \text{and} \qquad \mathbf{Sen}^L(\sigma) = \mathbf{Sen}(\sigma).$$

- $\mathbf{Mod}^L$ *is defined by*

$$\mathbf{Mod}^L(\Sigma^{Syn}) = \{m : \Sigma^{Mod} \to \mathcal{F} \mid m \circ (\mathcal{F} \hookrightarrow \Sigma^{Mod}) = id_{\mathcal{F}}\}$$
$$\mathbf{Mod}^L(\sigma^{syn})(m') = m' \circ \sigma^{mod}.$$

*All model categories are discrete.*

– *We make the following abbreviation: For a model $m \in \mathbf{Mod}^L(\Sigma^{Syn})$, we write $\overline{m}$ for $m \circ \Sigma^{mod} \circ (L^{Syn} \hookrightarrow \Sigma^{Syn}) \circ L^{truth} : Base \to \mathcal{F}$. Then we define satisfaction by*

$$m \models^L_{\Sigma^{Syn}} F \qquad \text{iff} \qquad \vdash_{\overline{m}} \mathbf{Sen}(m \circ \Sigma^{mod})(F).$$

**Theorem 1 (Institutions in LMF).** *In the situation of Def. 6, $\mathbb{F}(L)$ is an institution.*

*Proof.* We need to show the satisfaction condition. So assume $\sigma^{syn} : \Sigma^{Syn} \to \Sigma'^{Syn}$, $F \in \mathbf{Sen}^L(\Sigma^{Syn})$, and $m' \in \mathbf{Mod}^L(\Sigma'^{Syn})$. First observe that $\overline{m'} = m' \circ \Sigma'^{mod} \circ (L^{Syn} \hookrightarrow \Sigma'^{Syn}) \circ L^{truth} = (m' \circ \sigma^{mod}) \circ \Sigma^{mod} \circ (L^{Syn} \hookrightarrow \Sigma^{Syn}) \circ L^{truth} = \overline{m' \circ \sigma^{mod}}$. Then $\mathbf{Mod}^L(\sigma)(m') \models^L_{\Sigma^{Syn}} F$ iff $\vdash_{\overline{m' \circ \sigma^{mod}}} \mathbf{Sen}((m' \circ \sigma^{mod}) \circ \Sigma^{mod})(F)$ iff $\vdash_{\overline{m'}} \mathbf{Sen}(m' \circ \Sigma'^{mod})(\mathbf{Sen}(\sigma^{syn})(F))$ iff $m' \models^L_{\Sigma'^{Syn}} \mathbf{Sen}^L(\sigma^{syn})(F)$.

*Example 4 (FOL).* We can now obtain an institution from the encoding of first-order logic in Sect. 2.3 based on the logical framework $\mathbb{F}^{LF}$. First-order logic is encoded as the tuple $FOL = (FOL^{Syn}, FOL^{truth}, FOL^{Mod}, ZFC, FOL^{mod})$ as in Sect. 2.3.

We obtain an institution comorphism $\mathbb{FOL} \to \mathbb{F}^{LF}(FOL)$ as follows. Signatures of $\mathbb{FOL}$ are mapped to the extension of $FOL^{Syn}$ with declarations $f : i \to \ldots \to i \to i$ for function symbols $f$, $p : i \to \ldots \to i \to o$ for predicate symbols $p$. If we want to map $\mathbb{FOL}$ theories as well, we add declarations $ax : ded\, F$ for every axiom $F$. Signature morphisms are mapped in the obvious way. The sentence translation is an obvious bijection. The model translation maps every $m : \Sigma^{Mod} \to \mathcal{F}$ to the model whose universe is given by $m(univ)$ and which interprets symbols $f$ and $p$ according to $m(f)$ and $m(p)$. The model translation is not surjective as there are only countably many morphisms $m$ in $\mathbb{F}^{LF}(FOL)$. However, since $\mathbb{FOL}$ has a constructive existence proof of canonical models, these models can be represented as $ZFC$ terms and are in the image of the model translation. The satisfaction condition can be proved by an easy induction. $\mathbb{F}^{LF}(FOL)$ is complete thus $\mathbb{FOL}$ and $\mathbb{F}^{LF}(FOL)$ have the same consequence relation.

Logical frameworks can also be used to encode institution comorphisms in an intuitive way:

**Theorem 2 (Institution Comorphisms in LMF).** *Fix a logical framework $\mathbb{F} = (\mathbb{C}, Base, \mathbf{Sen}, \vdash)$. Assume two logics $L = (L^{Syn}, L^{truth}, L^{Mod}, \mathcal{F}, L^{mod})$ and $L' = (L'^{Syn}, L'^{truth}, L'^{Mod}, \mathcal{F}, L'^{mod})$. Then a comorphism $\mathbb{F}(L) \to \mathbb{F}(L')$ is induced by morphisms $(l^{syn}, l^{mod})$ if the following diagram commutes*

$$\begin{tikzcd}
 & \mathcal{F} \arrow[ld, hook] \arrow[rd, hook] & \\
L^{Mod} \arrow[rr, "l^{mod}"] & & L'^{Mod} \\
 & Base \arrow[ld, "L^{truth}"'] \arrow[rd, "L'^{truth}"] & \\
L^{Syn} \arrow[uu, "L^{mod}"] \arrow[rr, "l^{syn}"] & & L'^{Syn} \arrow[uu, "L'^{mod}"']
\end{tikzcd}$$

*Proof.* A signature $L^{Syn} \hookrightarrow \Sigma^{Syn}$ is translated to $L'^{Syn} \hookrightarrow \Sigma'^{Syn}$ by pushout along $l^{syn}$ yielding $\sigma^{syn} : \Sigma^{Syn} \to \Sigma'^{Syn}$. Sentences are translated by applying $\sigma^{syn}$. We obtain $\sigma^{mod} : \Sigma^{Mod} \to \Sigma'^{Mod}$ as the unique morphism through the pushout $\Sigma^{Mod}$. Then models are translated by composition with $\sigma^{mod}$. We omit the details.

It is easy to see that comorphisms that are embeddings can be elegantly represented in this way, as well as many inductively defined encodings. However, the assumptions of this theorem are too strong to permit the encoding of some less trivial comorphisms. For example, non-compositional sentence translations, which come up when translating modal logic to first-order logic, cannot be represented as signature morphisms. Or signature translations that do not preserve the number of non-logical symbols, which come up when translating partial to total function symbols, often cannot be represented as pushouts. More general constructions for the special case of LF are given in [Rab10] and [Soj10].

### 3.2 Generalizations

In Ex. 4, we do not obtain a comorphism in the opposite direction. There are three reasons for that. Firstly, $\mathbb{F}^{LF}(FOL)$ contains a lot more signatures than needed because the definition of $\mathbf{Sig}^L$ permits any extension of $L^{Syn}$, not just the ones corresponding to function and predicate symbols. Secondly, the discrete model categories of $\mathbb{F}^{LF}(FOL)$ cannot represent the model morphisms of $\mathbb{FOL}$. Thirdly, only a (countable) subclass of the models of $\mathbb{FOL}$ can be represented as $\mathbb{LF}$ morphisms. Moreover, Def. 4 and 6 are restricted to institutions, i.e., the syntax and model theory of a logic, and exclude the proof theory. We look at these problems below.

*Signatures* In order to solve the first problem we need to restrict $\mathbb{F}(L)$ to a subcategory of $\mathbf{Sig}^L$. However, it is difficult to single out the needed subcategory in a mechanizable way. Therefore, we restrict attention to those logical frameworks where $\mathbb{C}$ is the category of theories of a declarative language.

In a declarative language, the theories are given by a list of typed symbol declarations. In order to formalize this definition without committing to a type system, we use MMT expressions ([Rab08]) as the types. MMT expressions are formed from variables, constants, applications $@(E, l)$ of an expression $E$ to a list of expressions $l$, bindings $\beta(E, l, E')$ of a binder $E$ with scope $E'$ binding a list of variables typed by the elements of $l$. To that we add jokers $*$, which matches an arbitrary expressions, and $\overline{E}$, which matches a list of expressions each of which matches $E$.

Such MMT expression patterns give us a generic way to pattern-match declarations of the logical framework. If a concrete logic definition contains a set $P$ of patterns, we represent its logical signatures as $\mathbb{C}$-objects $\Sigma^{Syn}$ that extend $L^{Syn}$ only with declarations matching one of the patterns in $P$. For example, the patterns for first-order logic from Ex. 4 would be $@(\to, \overline{i}, i)$ and $@(\to, \overline{i}, o)$ for function and predicate symbols of arbitrary arity, and $@(ded, *)$ for axioms.

Here $*$ stands for an arbitrary expression, which in this case must be a sentence to be well-typed.

*Model Morphisms* Regarding the second problem, if $\mathbb{C}$ is a 2-category, we can define the model morphisms of $\mathbb{F}(L)$ as 2-cells in $\mathbb{C}$. However it is difficult in practice to obtain 2-categories for type theories such as LF or Isabelle. In [Soj10], we give a syntactical account of logical relations that behave like 2-cells in sufficiently many ways to yield model morphisms.

*Undefinable Models* The third problem is the most fundamental one because no formal logical framework can ever encode all models of a Platonic universe. Our encoding of ZFC is strong enough to encode any **definable** model. We call a model $M$ definable if it arises as the solution to a formula $\exists^! M.F(M)$ for some parameter-free formula $F(x)$ of the first-order language of ZFC. This restriction is philosophically serious but in our experience not harmful in practice. Indeed, if infinite LF signatures are allowed, using canonical models constructed in completeness proofs, in many cases *all* models can be represented up to elementary equivalence.

*Proof Theory* Our examples from Sect. 2.3 already encoded the proof theory of first-order logic in a way that treats proof theory and model theory in a balanced way. Our definitions can be easily generalized to this setting.

Logic encodings in a logical framework become 6-tuples $(L^{Syn}, L^{truth}, L^{Mod}, \mathcal{F}, L^{mod}, L^{Pf}, L^{pf})$ for $L^{pf} : L^{Syn} \to L^{Pf}$. $L^{Pf}$ encodes the proof theory of a logic, which typically means to add auxiliary syntax, judgments, and proof rules to $L^{Syn}$. Def. 5 can be extended to obtain $\Sigma^{pf} : \Sigma^{Syn} \to \Sigma^{Pf}$ as a pushout in the same way as $\Sigma^{mod}$. Finally the logical framework must be extended with a component that yields a data structure of proofs (such as entailment systems or proof trees) for every slice out of *Base*.

For example, for the framework $\mathbb{F}^{LF}$, the proof trees for proofs of $F$ using assumptions $F_1, \ldots, F_n$ can be defined as the $\beta\eta$-normal LF terms over $\Sigma^{Pf}$ of type $\Sigma^{pf}\big(L^{truth}(ded)\, F_1 \;\to\; \ldots \;\to\; L^{truth}(ded)\, F_n \;\to\; L^{truth}(ded)\, F\big)$. A similar construction was given in [Rab10].

## 4 Logical Frameworks in Hets

The differences between LF and Hets mentioned in Sect. 2 exhibit complementary strengths, and a major goal of our work is to combine them. We have enhanced Hets with a component that allows the dynamic definition of new logics. The user specifies a logic by giving the representation of its constituents (syntax, model theory) in a logical framework and the combined system recognizes the new logic and integrates it into the Hets logic graph. The implementation follows the Hets principles of high abstraction and separation of concerns: we provide an implementation for the general concept of logical frameworks, which we describe in Sect. 4.1. This is further instantiated for the particular case of LF in Sect. 4.2. Finally, in Sect. 4.3 we present a complete description of the steps necessary to add a new logic in Hets using the framework of LF.

### 4.1 Implementing the LMF in Hets

This section sketches how the concept of logical frameworks is integrated into Hets. The integration is done entirely on the developer's side and a user wishing to add a new logic to Hets only has to select one of the available logical frameworks, which will serve as a meta-logic for the new object logic he or she specifies. We will give here just a brief overview of how the implementation is done and refer the interested reader to [Mos05] for a presentation of the theoretical foundations of Hets and to the Hets developers documentation pages [5] for a more detailed presentation of how the coding is actually done.

The central part of the implementation is a Haskell type class *LogicalFramework*, which is instantiated by the logics which can be used as logical frameworks, i.e. in which object logics can be specified by the user. Such candidates are for example LF, rewriting logic and Isabelle [6]. The class provides a selector for the *Base* signature and a method *writeLogic*, which takes an object logic name as an argument and generates the instances of the classes *Syntax*, *Sentences*, *StaticAnalysis*, and *Logic* for the given object logic.

Each logic implementing *LogicalFramework* must likewise implement the class *Category*, from which we get the category $\mathbb{C}$ mentioned in Def. 4. The sentence functor **Sen** is specified implicitly by the *writeLogic* method: the instantiation of the *StaticAnalysis* class determines exactly which sentences are valid for a particular signature of $L$, thus giving **Sen** on objects. Since the current implementation of logics in Hets does not include satisfaction of sentences in models, the predicate $\vdash_t$ is currently not represented as its main purpose is to define the satisfaction relation for object logics.

At the syntactic level, we must provide a way to write down new logic definitions in HetCASL, the underlying heterogenous algebraic specification language of Hets. Since definitions of new logics have a different status than usual algebraic specifications, we extend the language at the library level.

*Concrete Syntax* We add the following concrete syntax (on the right) to HetCASL in order to define new logics. Here $L$ is the name of the newly defined logic and $\mathbb{F}$ is an identifier pointing to the logical framework used. The identifiers $L^{truth}, L^{mod}, L^{pf}, \mathcal{F}$ are the components of the new logic $L$. They refer to previously declared signature morphisms of $\mathbb{F}$ and the signatures representing $L^{Syn}, L^{Mod}, L^{Pf}$ can be inferred from them. $\mathcal{F}$ is a signature which gives the foundation. The declaration of patterns is optional.

```
newlogic L =
  meta F
  syntax L^truth
  models L^mod
  foundation ℱ
  proofs L^pf
  patterns P
```

[5] See http://www.informatik.uni-bremen.de/agbkb/forschung/formal_methods/CoFI/hets/src-distribution/daily/Hets/docs/Logic.html.

[6] Currently only LF has a full implementation as a logical framework.

After encountering a `newlogic` declaration, Hets invokes a static analyzer, which retrieves the signatures and morphisms constituting the components of the logic $L$. The analyzer verifies the correct shape of the induced diagram and instantiates the *Logic* class for the logic $L$ as specified by the *writeLogic* method of the framework $\mathbb{F}$.

The logic $L$ arising from the above `newlogic L` declaration differs slightly from the one described in Def. 6 in that it uses signatures of $\mathbb{F}$ that extend $L^{Syn}$ rather than $\mathbb{F}$-inclusion morphisms out of $L^{Syn}$. Accordingly, the morphisms of $L$ are those morphisms of $\mathbb{F}$ which are the identity on $L^{Syn}$. This is essentially the same thing, but has the advantage that the data types representing the signatures and morphisms of $\mathbb{F}$ can be directly reused for $L$ and no separate instantiation of the class *Category* is required [7].

## 4.2 LF as a Logical Framework in Hets

In this section we outline how to turn $\mathbb{LF}$ into a logical framework in Hets, i.e. how to instantiate the *LogicalFramework* class for $\mathbb{LF}$. In order to do so we will make use of the instance of the *Logic* class for $\mathbb{LF}$. [8].

The *Base* signature is specified to be the $\mathbb{LF}$ signature containing the symbols $o$ and $ded$, as described in Sect. 3. The instantiations of the classes *Logic*, *Syntax*, etc. provided by the *writeLogic* method mostly inherit their $\mathbb{LF}$ implementations, with one exception being the *StaticAnalysis* class. While both $\mathbb{LF}$ and the $\mathbb{LF}$ object logics use Twelf to verify the well-formedness of input specifications, a specification in an object logic is assumed to have been given relative to the $L^{Syn}$ signature supplied when defining the object logic.

After receiving the input file, Twelf performs parsing, static analysis and reconstruction of types and implicit arguments. If the analysis succeeds, the output is stored as an OMDoc version of the input file, and is subsequently imported into Hets using standard XML technologies. Hets reads the imported OMDoc file and transforms it into corresponding $\mathbb{LF}$ signatures and morphisms in their Hets internal representation.

## 4.3 Adding a New Logic in Hets: FOL

We will now illustrate the steps needed to add first-order logic as a new logic in Hets. The aim of this section is not to show how to encode a particular logic in Twelf, which for the case of first-order logic has been described in [HR11], but rather to show how an existing encoding can be used to add the logic in Hets.

Given a *FOL* encoding as in Section 2.3, all that is needed to be done is to collect the components of the encoding in a `newlogic` definition, as in Fig. 2. The first lines import the morphism $FOL^{truth}$ from *Base* to $FOL^{Syn}$, the morphism $FOL^{mod}$ from $FOL^{Syn}$ to $FOL^{Mod}$, and the morphism $FOL^{pf}$ from $FOL^{Syn}$ to

[7] The theory presented in Section 3 could thus have been formulated equivalently, albeit less elegantly, without referring to slice categories.

[8] An institution for $\mathbb{LF}$ can be defined as for example in [Rab08].

$FOL^{Pf}$ as in Ex. 4, from their respective directories. *STTIFOLEQ* is a fragment of *ZFC* used to represent model theory. It is composed of simple type theory equipped with external intuitionistic first-order logic. Notice that we assume for convenience that the file with the new logic definition is in the folder that contains the directory of logics as sub-folder; the paths need to be adjusted if that is not the case. [9] The directory structure mirrors the modular design of logics in the Logic Atlas. As a result of calling Hets on the above file, a new directory called

```
from logics/first-order/syntax/fol get FOL_truth  %%FOL^truth
from logics/first-order/model_theory/fol get FOL_mod %%FOL^mod
from logics/meta/sttifol get STTIFOLEQ %%𝓕
from logics/first-order/proof_theory/fol get FOL_pf %%FOL^pf

newlogic FOL =
  meta 𝕃𝔽
  syntax FOL_truth
  models FOL_mod
  foundation STTIFOLEQ
  proofs FOL_pf
end
```

**Fig. 2.** Defining *FOL* as a new object logic.

*FOL* is added to the source folder of Hets. The directory contains automatically generated files with the instances needed for the logic *FOL*. Moreover, the Hets variable containing the list of available logics is updated to include *FOL*. After recompiling Hets, the new logic is added to the logic graph of Hets (the node *FOL* in Fig. 1 for the dynamically-added logic) and can be used in the same way as any of the built-in logics.

In particular, we can now use the new object logic to write specifications. For example, the specification in Fig. 3 uses *FOL* as a current logic and declares a constant symbol $c$ and a predicate $p$, together with an axiom that the predicate $p$ holds for the constant $c$. Notice that the syntax for logics specified in a logical framework $\mathbb{F}$ is inherited from the framework (in our case $\mathbb{LF}$), but it has been extended with support for sentences, in the usual CASL syntax i.e. prefixed by the '.' character.

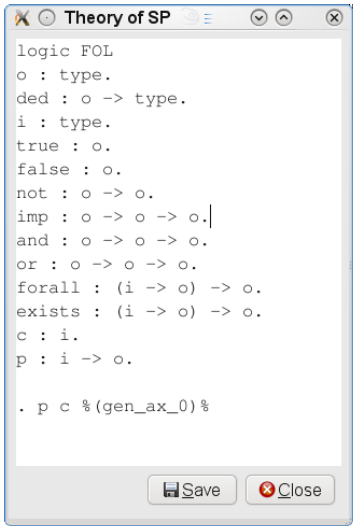
Fig. 4 presents the theory of SP as displayed from within Hets; as mentioned in Section 4.2, the theory is automatically assumed to extend $FOL^{Syn}$. Since in Hets all imports are internally flattened, the theory of SP when displayed will include all the symbols from $FOL^{Syn}$.

[9] The complete specification of *FOL* in LF can be found at `https://svn.omdoc.org/repos/latin/twelf-r1687/`.

```
logic FOL
spec SP =
  c : i.
  p : i -> o.

  . p c
end
```

**Fig. 3.** Specification in the new object logic.



```
logic FOL
o : type.
ded : o -> type.
i : type.
true : o.
false : o.
not : o -> o.
imp : o -> o -> o.
and : o -> o -> o.
or : o -> o -> o.
forall : (i -> o) -> o.
exists : (i -> o) -> o.
c : i.
p : i -> o.

. p c %(gen_ax_0)%
```

**Fig. 4.** Theory of SP

## 5 Conclusion & Future Work

We have described a prototypical integration of the institution-based Heterogeneous Tool Sets (Hets) with logical frameworks in general and LF and the Twelf tool in particular. The structuring language used by Hets has a model theoretic semantics, which has been reflected in the proof theoretic logical framework LF by representing models as theory morphisms into some foundation. While LF is the logical framework of our current choice, both the theory and the implementation are so general that other frameworks like Isabelle can be used as well. We expect important synergy effects from this as Isabelle is already used as one of the main inference engines in Hets.

Proof theory of the represented logics has been treated only superficially in the present work, but in fact, we have represented proof calculi for all the LATIN logics within LF. Representing models in the system as well has enabled us to formally prove soundness of the calculi. It is straightforward to extend the construction of institutions out of logic representations in logical frameworks such that they deliver institutions with proofs. In the long run, we envision that the provers integrated in Hets also return proof terms, which Hets can then fill into the original file and rerun Twelf on it to validate the proof. Thus, Hets becomes the mediator that orchestrates the interaction between external theorem provers and Twelf as a trusted proof checker.

While the theory and implementation described in this paper make it possible to add logics to Hets in a purely declarative way, further work is needed to turn this into a scalable tool. Firstly, the logic translations-as-theory morphisms approach needs to be generalised in order to cover more practically useful examples. Secondly, the new LF generated logics present in Hets need to be connected (via institution comorphisms) to the existing hard-coded logics in order to share the connection of the latter to theorem provers and other tools. Thirdly, it will be desirable to have a declarative interface for specifying the syntax of new logics, such that one is not forced to use the syntax of the logical framework. We are currently examining whether Eclipse and Xtext are helpful here. Finally, also the various tool interfaces of Hets should be made more declarative, such that Hets logics specified in a logical framework can be directly connected to theorem provers and other tools, instead of using a comorphism into a hard-coded logic. Then, in the long run, it will be possible to entirely replace the hard-coded logics with declarative logic specifications in the LATIN metaframework — and only the latter needs to be hard-coded into Hets.

The Logic Atlas currently consists of a around 150 files containing some 700 signatures and views and producing over 10000 lines of Twelf output (including declarations that are generated by the module system). This is the result of roughly one year of development with substantial contributions from six different people, and due to the evolutionary improvement of our methodology, architecture, and expertise, growth has been exponential. Nevertheless, the representation and interconnection of logics is (and will remain) a task that requires a deep understanding of the respective logics, a good eye for the underlying primitives, and sound judgment in the design and layout of atlantes. We consider the current Logic Atlas to be a seed atlas that establishes best practices in these questions and provides a nucleus of logical primitives that can be extended to add particular logics by outside logic and system developers.

We explicitly invite researchers outside the LATIN project to contribute their logics. This should usually be a matter of importing the aspects that are provided by Logic Atlas theories, and LF-encoding the aspects that are not.

*Acknowledgments* This paper mainly addresses the model theoretic side of the logic atlas developed in the LATIN project — funded by the German Research Council (DFG) under grant KO-2428/9-1.

## References

ABK$^+$02. E. Astesiano, M. Bidoit, H. Kirchner, B. Krieg-Brückner, P. Mosses, D. Sannella, and A. Tarlecki. CASL: The Common Algebraic Specification Language. *Theoretical Computer Science*, 286(2):153–196, 2002.

AHMP98. A. 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.

BC04. Y. Bertot and P. Castéran. *Coq'Art: The Calculus of Inductive Constructions.* Springer, 2004.

BJL06. M. Bortin, E. Broch Johnsen, and C. Lüth. Structured formal development in Isabelle. *Nordic Journal of Computing*, 12:1–20, 2006.

CAB$^+$86. R. Constable, S. Allen, H. Bromley, W. Cleaveland, J. Cremer, R. Harper, D. Howe, T. Knoblock, N. Mendler, P. Panangaden, J. Sasaki, and S. Smith. *Implementing Mathematics with the Nuprl Development System*. Prentice-Hall, 1986.

CELM96. M. Clavel, S. Eker, P. Lincoln, and J. Meseguer. Principles of Maude. In J. Meseguer, editor, *Proceedings of the First International Workshop on Rewriting Logic*, volume 4, pages 65–89, 1996.

Chu40. A. Church. A Formulation of the Simple Theory of Types. *Journal of Symbolic Logic*, 5(1):56–68, 1940.

Dia08. R. Diaconescu. *Institution-independent Model Theory*. Birkhäuser, 2008.

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.

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.

HR11. F. Horozal and F. Rabe. Representing Model Theory in a Type-Theoretical Logical Framework. *Theoretical Computer Science*, 2011. To appear, see `http://kwarc.info/frabe/Research/HR_folsound_10.pdf`.

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

IR11. M. Iancu and F. Rabe. Formalizing Foundations of Mathematics. *Mathematical Structures in Computer Science*, 2011. To appear, see `http://kwarc.info/frabe/Research/IR_foundations_10.pdf`.

KMR09. M. Kohlhase, T. Mossakowski, and F. Rabe. The LATIN Project, 2009. See `https://trac.omdoc.org/LATIN/`.

MAH06. T. Mossakowski, S. Autexier, and D. Hutter. Development Graphs - Proof Management for Structured Specifications. *Journal of Logic and Algebraic Programming*, 67(1-2):114–145, 2006.

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

ML74. P. Martin-Löf. An Intuitionistic Theory of Types: Predicative Part. In *Proceedings of the '73 Logic Colloquium*, pages 73–118. North-Holland, 1974.

MML07. T. Mossakowski, C. Maeder, and K. Lüttich. The Heterogeneous Tool Set. In O. Grumberg and M. Huth, editor, *TACAS 2007*, volume 4424 of *Lecture Notes in Computer Science*, pages 519–522, 2007.

MOM94. Narciso Martí-Oliet and José Meseguer. What is a logical system? chapter General logics and logical frameworks, pages 355–391. Oxford University Press, Inc., New York, NY, USA, 1994.

Mos05. T. Mossakowski. Heterogeneous Specification and the Heterogeneous Tool Set, 2005. Habilitation thesis, see `http://www.informatik.uni-bremen.de/~till/`.

Nor05. U. Norell. The Agda WiKi, 2005. `http://wiki.portal.chalmers.se/agda`.

NPW02. T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. Springer, 2002.

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

PC93. L. Paulson and M. Coen. Zermelo-Fraenkel Set Theory, 1993. Isabelle distribution, ZF/ZF.thy.

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

PS99. F. Pfenning and C. Schürmann. System description: Twelf - a meta-logical framework for deductive systems. *Lecture Notes in Computer Science*, 1632:202–206, 1999.

PS08. A. Poswolsky and C. Schürmann. System Description: Delphin - A Functional Programming Language for Deductive Systems. In A. Abel and C. Urban, editors, *International Workshop on Logical Frameworks and Metalanguages: Theory and Practice*, pages 135–141. ENTCS, 2008.

PSK$^+$03. F. Pfenning, C. Schürmann, M. Kohlhase, N. Shankar, and S. Owre. The Logosphere Project, 2003. `http://www.logosphere.org/`.

Rab08. F. Rabe. *Representing Logics and Logic Translations*. PhD thesis, Jacobs University Bremen, 2008. Available at `http://kwarc.info/frabe/Research/phdthesis.pdf`.

Rab10. F. Rabe. A Logical Framework Combining Model and Proof Theory. Submitted to Mathematical Structures in Computer Science, see `http://kwarc.info/frabe/Research/rabe_combining_09.pdf`, 2010.

RS09. F. Rabe and C. Schürmann. A Practical Module System for LF. In J. Cheney and A. Felty, editors, *Proceedings of the Workshop on Logical Frameworks: Meta-Theory and Practice (LFMTP)*, pages 40–48. ACM Press, 2009.

Soj10. K. Sojakova. Mechanically Verifying Logic Translations, 2010. Master's thesis, Jacobs University Bremen.

SS04. C. Schürmann and M. Stehr. An Executable Formalization of the HOL/Nuprl Connection in the Metalogical Framework Twelf. In *11th International Conference on Logic for Programming Artificial Intelligence and Reasoning*, 2004.

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.