Towards an Atlas of Formal Logics

Fulya Horozal, Michael Kohlhase, Florian Rabe, Kristina Sojakova

Computer Science, Jacobs University, Bremen
{f.horozal,m.kohlhase,f.rabe,k.sojakova}@jacobs-university.de

LF has been designed as a meta-logical framework to represent logics, and has become a standard tool for studying properties of logics. Building on the newly introduced module system for LF, we present the nucleus of an integrated and structured development of the syntax, semantics, and proof theory of logics, and of the relations between those logics. The methodology is chosen so that it will scale to an atlas for the zoo of logics currently used in reasoning systems, and the modular nature of this development aids the practical integration of systems because shared features of the logics are reused directly. Finally we show how these encodings in LF are imported into the Hets system, which provides automated proof support on the modular level and integrates various automated theorem provers for the represented object logics.

1 Introduction

Logics allow making mathematical knowledge accessible to computational systems like (semi-)automated theorem provers, model checkers, computer algebra systems, constraint solvers, or concept classifiers. Unfortunately, these systems have differing foundational assumptions and input languages, which makes them non-interoperable and difficult to compare and evaluate in practice. To remedy this situation, we need a foundationally unconstrained framework for knowledge representation that allows to represent the meta-theoretic foundations of the mathematical knowledge in the same format and to interlink the foundations at the meta-logical level.

Parts of the necessary framework have been developed under the heading of "logical frameworks". For instance LF has been used extensively to represent logics [HHP93, 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]. Various logics were represented in Coq [BC04] and Agda [Nor05]. 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 [BC04], [Nor05], and [CAB+86].

But simply representing logics in a logical framework is not enough. For establishing interoperability between systems that use different logics, we need logic translations, which are much more sporadic and usually still carried out in an ad hoc manner, i.e. not using a framework that explicitly supports translations. Important logic translations represented using the logic programming interpretation of Twelf include cut elimination [Pfe00] and the HOL-Nurpl translation [SS04]. More systematic approaches have been conducted in the model theoretical community (see, e.g., [GB92, Mes89, MML07]). However, here logical frameworks are understood as a theoretical foundation rooted in classical mathematics, not as an implemented formal system. Recently several systems were developed that subsume LF and provide special support for translations. Delphin [PS08] uses a functional programming language on top of LF to translate between LF signatures using continuations for fresh parameters. Beluga [PD10] is similar but explicitly represents contexts.

In this paper, we report on work in the ongoing project LATIN (Logic Atlas and Integrator) which aims to (i) use a "logics as theories/translations as morphisms" approach to achieve to interoperability of both system behavior and represented knowledge (the Logic Integrator), and to (ii) obtain a comprehensive and interconnected network of formalizations of logics of computational logic systems (the Logic Atlas). In LATIN, we largely abstract from the surface syntax of logical systems and focus on the role of modular developments of logic representations to structure the Logic Atlas and re-use shared components of logics. In [RS09], we gave a module system for LF based on signature morphisms that we (designed for and) use as the base of our Atlas. This system is really without much competition, as use of modularity to represent inheritance between logics is not well-studied in logical frameworks. An exception are the encodings of modal logics in Isabelle in [BMV98], but this approach is not elevated to a general framework for modularity.

One of the main goals of LATIN is to integrate both proof and model theoretical frameworks. While this work focuses on the former with LF and Twelf as our preferred framework, we briefly discuss the model theoretical aspects, for which we employ the Hets system, which is also part of the LATIN project.

This paper is organized as follows. We give an introduction to LF and to its module system in Sect. 2. In Sect. 3, we present the current developments in the Logic Atlas with examples and discuss knowledge management aspects of its underlying infrastructure. In Sect. 4, we describe the Hets system and its integration with Twelf.

2 LF and its Module System

LF [HHP93] is a dependent type theory related to Martin-Löf type theory [ML74]. It is the corner of the λ-cube [Bar92] (see Sect. 3.2) that extends simple type theory with dependent function types. We will work with the Twelf implementation of LF [PS99] using the Twelf module system [RS09]. We will briefly revisit the relevant notions and syntax. We refer to [HHP93], [PS99], and [HST94] for details on LF, Twelf, and LF signature morphisms and to [RS09] for the Twelf module system.

LF and Twelf LF expressions E are grouped into kinds K, kinded type-families $A : K$, and typed terms $t : A$. We will always use Twelf notation for expressions. Then the kinds are the base kind `type` and the dependent function kinds $\{x : A\}\,K$. The type families are the constants a, applications $a\,t$, and the dependent function type $\{x : A\}\,B$; type families of kind `type` are called types. The terms are constants c, applications $t\,t'$, and abstractions $[x : A]\,t$.

We write $A \to B$ instead of $\{x : A\}\,B$ if x does not occur in B, and we will omit the types of bound variables if they can be inferred. Upper case free variables are implicitly bound on the outside of the expression (implicit arguments).

Finally, an LF **signature** Σ 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.

Syntax of the Module System The **module system** for LF and Twelf is based on the notion of signature morphisms: Given two signatures Σ and Σ', an LF **signature morphism** $\sigma : \Sigma \to \Sigma'$ is a type- and kind-preserving map of Σ-symbols to Σ'-expressions. Thus, σ maps every constant $c : A$ of Σ 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 σ to Σ-expressions, and we will write σ 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 equality.

The module level declarations consist of named signatures R, S, T and three kinds of signature morphism declarations: views v, structures r, s, and inclusions. The following grammar yields the modular signatures where terms, type families, and kinds are merged into expressions E and constants c may be typed or kinded. The module level declarations are explained below.

Sign. Graph	G	::=	$\cdot \mid G,\ \%\texttt{sig}\ T\ =\ \{\Sigma\} \mid \%\texttt{view}\ v\ :S\ \to T\ =\{\sigma\}$
Signatures	Σ	::=	$\cdot \mid \Sigma,\ \%\texttt{include}\ S \mid \Sigma,\ \%\texttt{struct}\ s\ :S\ =\{\sigma\} \mid \Sigma,\ c:E \mid \Sigma,\ c:E=E$
Morphisms	σ	::=	$\cdot \mid \sigma,\ \%\texttt{include}\ \mu \mid \sigma,\ \%\texttt{struct}\ s:=\mu \mid \sigma,\ c:=E$
Expressions	E	::=	$\texttt{type} \mid c \mid x \mid E\ E \mid [x:E]E \mid \{x:E\}E \mid E \to E$
Composites	μ	::=	$\cdot \mid T.s\ \mu \mid v\ \mu$

A **view** v from signature S to signature T is a signature morphism that represents a translation or a refinement. We can also think of S as a specification, of T as a semantic realm, e.g., higher-order logic represented in LF, and of v as an implementation or model of S in terms of T.

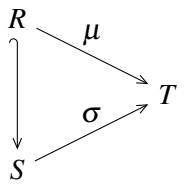
Structures and inclusions declare signature morphisms into the containing signature, they represent an import or inheritance relationships. A **structure** $\%\texttt{struct}\ s\ :S\ =\{\sigma\}$ declared in signature T imports and translates all declarations of S into T. s is used to form qualified names $s.c$ for the imported constants c of S. Contrary to views, σ is usually not total: If σ is defined, i.e., if S declares $c:E$ and σ contains $c:=E'$, the induced constant is $s.c:T.s(E)=E'$. Here $T.s:S\to T$ is the signature morphisms mapping every c of S to $s.c$. Thus, structures represent the instantiation of parametric signatures.

Inclusions $\%\texttt{include}\ S$ declared in T are similar to structures but the induced signature morphism from S to T is always an inclusion. Consequently, include declarations are unnamed and may not carry instantiations. The included symbols c of S are available as $S.c$ in T.

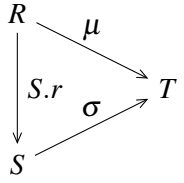
Finally, we can form composed morphisms μ as lists of structures and views. For example, if v is a view from R to S and T declares a structure s of type S, then $v\ T.s$ is a morphism from R to T. If R includes R', then $v\ T.s$ is also a morphism from R' to T.

Naturally, multiple inclusions of the same signature are redundant. But because structures are named, a signature may have multiple structures of the same signature, which are all distinct. For example, if a signature S contains a structure $\%\texttt{struct}\ r\ :R=\{\rho\}$, and T contains structures $\%\texttt{struct}\ r'\ :R=\{\rho'\}$ and $\%\texttt{struct}\ s\ :S\ =\{\sigma\}$, then T has the two structures r' and $s.r$ of S. Structure sharing is possible because morphisms may instantiate whole structures at once: If σ contains $\%\texttt{struct}\ r:=T.r'$, the two structures r' and $s.r$ are shared.

Structure sharing is a special case of **modular morphisms**. In general, a morphism σ occurring in a view or structure from S to T, may be given by mirroring the modular structure of S. Firstly, assume that S includes a signature R. Then σ in a view or structure from S to T must map all constants of S, i.e., also those included from R. But often a morphism μ from R to T is already available. σ may reuse μ with a declaration $\%\texttt{include}\ \mu$. The intended semantics is that the triangle on the right always commutes. In general, S may include R in multiple ways, e.g., by including R_1 and R_2 both of which include R (the diamond situation). If σ includes morphisms $\mu_i:R_i\to T$ for $i=1,2$, then we have to require that μ_1 and μ_2 agree on R to avoid inconsistencies.



Correspondingly, if S instantiates R via a structure $\%\texttt{struct}\ r\ :R\ =\{\rho\}$, then σ may contain a structure instantiation $\%\texttt{struct}\ r:=\mu$. As above this leads to the commutative diagram on the right. Again we also have to require that there are no conflicting declarations in ρ, σ, and μ. For example, if μ maps c to c_1 and ρ instantiates c with c_2, then σ must map c_2 to (an expression equal to) c_1.



Semantics and Implementation The semantics of modular LF is given in two ways. The official semantics appeals directly to the modular structure. For every well-formed signature graph G we obtain a graph $\overline{G}$ whose nodes are the signatures declared in G and whose edges are the views, structures, and inclusions declared in G. The well-formed morphisms from S to T are given by the paths from S to T in $\overline{G}$, possibly inserting inclusion edges.

Secondly, the module system is conservative in the sense that every modular signature can be elaborated (flattened) into a non-modular signature. Each node T of $\overline{G}$ can be labelled with a non-modular Twelf signature $\overline{T}$, which contains exactly one declaration for each constant declared in or imported into T. Similarly, each edge from S to T is labelled with non-modular signature morphism from $\overline{S}$ to $\overline{T}$.

The main theorem is that all signatures and signature morphisms in $\overline{G}$ are well-formed with respect to the semantics of non-modular LF iff G is well-formed as a modular LF signature graph. In particular, all signature morphisms in $\overline{G}$ are well-formed so that all LF judgments are preserved when translating expressions along the paths of $\overline{G}$.

The module system has been implemented as an extension of Twelf and is available as the `twelf-mod` branch of Twelf. The implementation can be succinctly described as computing $\overline{G}$ from G. The supported syntax is a bit more general than described here: It permits nested signatures and treats the toplevel as a signature itself. This has the effect that all legacy content remains well-formed.

3 Seeding the Logic Atlas

In this section, we give an insight into our Logic Atlas. Sect. 3.1 gives a birds-eye view of the Atlas. Then we give some example formalizations that are representative in style for the whole graph in Sect. 3.2 and Sect. 3.3. These illustrate modular representation of type-theories and logics, respectively. Our goal of a logic atlas heavily depends on the scalability of the employed infrastructure. Therefore, we discuss some knowledge management aspects that are particularly relevant to Twelf in Sect. 3.4.

3.1 Overview

One of the biggest challenges in creating and maintaining our Logic Atlas is how to organize the content. We hold that the best conceptual model is that of a graph – more precisely a multi-graph – of mathematical theories and relations between them. Here we use the term "theory" for any mathematical context that can be captured as a formal language and which includes for example arithmetic, lambda calculus, or group theory. All of these are formalized as LF signatures. The edges of the graph are translations between theories, which are represented as LF signature morphisms. They are typically split into inheritance translations, which hold by definition – inclusions and structures – and representation theorems, which have to be proved – views.

Signature morphisms are a good choice as the representational primitive for such translations because they are both simple and expressive, and enjoy a number of desirable properties such as being closed under composition and having a decidable notion of validity. However, they are not the last word yet as they can express only total and only compositional (also called structural or recursive) translations. This excludes non-compositional translations such as cut-elimination and the bracket abstraction translation from natural deduction to Hilbert calculi. We envision that eventually an extension of the module system to the Twelf meta-theory (which is able to express these translations as logic programs) will reconcile these two notions. But for now signature morphisms already get us a long way.

The next question then is how to map a multi-graph to the directory-based infrastructure of a storage

system. Since the structure of the multi-graph reflects that of mathematics, any grouping of theories into directories inevitably requires choices that would better be left open. However, the alternative of putting all files in the same directory scales very badly. We envision that eventually all modules (rather than files) will be stored individually in a database, and the distribution of modules over files and directories will be flexible and customizable for each working copy of the main repository. We made the first step into that direction using the virtual files of the TNTBase system [ZKR10], which we plan to extend to the LF format in the future.

For now we use an SVN repository [KMR09] with a directory structure that roughly reflects the division of research fields:

- `type_theories` contains languages that use typing judgments on expressions,
- `logics` contains languages that have a notion of formulas and truth,
- `set_theories` contains languages that are based on the $\in$ relation between sets,
- `math` contains languages used in specific mathematical domain, usually based on a certain logic; these are somewhat peripheral to our logic graph interest and serve mainly as case studies,
- `category_theory` contains the language of categories seen as an alternative foundation of mathematics,
- `translations` contains the individual edges of the graph.

The type theories are based on a modular development of orthogonal features of λ-calculi including the dimensions of the λ-cube, product types, union types, etc. Specific type theories are obtained by including the needed modules. The formalizations of the sorted first- and higher-order logics are parametric in the underlying type theory; for example, traditional HOL is obtained by combining simple function (and possibly product) types with Andrews-style logical symbols (i.e., with equality as the only primitive symbol).

First-order, modal, and description logics are formed similarly from orthogonal modules for individual connectives, quantifiers, and axioms. For example, the classical $\wedge$ connective is only declared once in the whole Atlas, and the axiom of excluded middle and its consequences reside in a separate signature. We also use individual modules for syntax, proof theory, model theory, and soundness proof so that the same syntax can be combined with different interpretations.

To represent model theory, we use a formalization a Zermelo-Fraenkel set theory. Exploiting the expressive strength of a dependently-typed meta-language, we can represent set theory using only $\in$ and a (definite) description operator as the primitive operations, which is closer to textbook accounts of mathematics than the formalizations in Isabelle/ZF and Mizar. Furthermore, we recover a typed representation of untyped set theory, which permits LF-level typed reasoning, by declaring appropriate abbreviations. This development is currently the largest in the Atlas at around 4000 lines of Twelf output at default verbosity. It is strong enough to develop, e.g., dependent type theory within set theory. A formalization of Mizar and Tarski-Grothendieck set theory is currently under way.

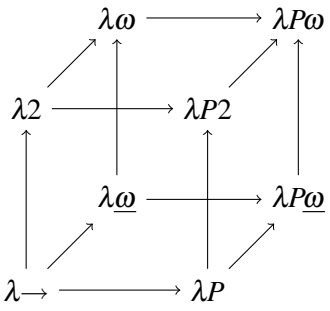
The highly modular structure of these theories already yields a large number of inheritance edges. Additionally, the representation theorems that we have so far formalized as LF views include:

- the translation from unsorted to sorted first-order logic (which is almost but not quite an inclusion),
- the translations by relativization of quantifiers from sorted first-order, modal, and description logics to unsorted first-order logic, in most cases including the translation of the model theory,
- the translation from propositional and sorted first-order logic to Andrews-style higher-order logic,
- the negative translation from classical to intuitionistic logic,
- the translation from type theory to set theory that interprets types as sets and terms as elements, including a translation of Isabelle/HOL to ZF set theory,
- the Curry-Howard correspondence between logic, type theory, and category theory.

All translations include translations of the proof theory, which guarantees their proof theoretical soundness due to the type-preservation of signature morphisms.

3.2 Type Theories

The λ-cube The λ-cube [Bar92] is a framework consisting of eight different typed λ-calculi, each of which is defined uniformly by a different combination of polymorphism, type operators and dependent types. The framework represents systems from simply typed λ-calculus [Chu36] to the calculus of constructions [CH88] as corners of the cube shown on the right.



The systems $\lambda\!\rightarrow$, $\lambda 2$ and λP are simply typed λ-calculus, polymorphic (or second-order) λ-calculus as in [Gir71] and dependently typed λ-calculus corresponding to LF, respectively. $\lambda\underline{\omega}$ is a calculus with type operators studied in [dL92], and $\lambda P\underline{\omega}$ extends this with dependent types/kinds. The system $\lambda P2$ corresponds to LF with universal types studied in [LM88]. The systems $\lambda\omega$ and $\lambda P\omega$ correspond to the system $F\omega$ in [Gir71] of Girard and to calculus of constructions, respectively.

The framework is based on a set of general typing rules for $\lambda\Pi$-abstraction and application that are parametric in the levels of the variable and its scope. Levels s range over the set $\{*,\Box\}$ where $*$ is the universe of types and $\Box$ the one of kinds. The systems are differentiated by the Π-introduction rule

$$\frac{\Gamma \vdash A : s_1 \quad \Gamma, x : A \vdash B : s_2}{\Gamma \vdash \Pi x : A.B : s_2}\,(s_1, s_2)$$

which is present only for different pairs (s_1,s_2) of levels in each system: All systems have the rule $(*,*)$ for typed variables occurring in typed expressions, and the combinations of the remaining three rules yield the eight corners of the cube.

Modular Encoding of the λ-cube in Twelf The encoding of the λ-cube in Twelf benefits from the module system significantly. There are three main parts of the encoding: (i) the encoding of the levels and judgments, (ii) the encoding of the individual features of the cube, and (iii) the combination of these features to form the respective system.

(i) In $\lambda\!\rightarrow$ and $\lambda 2$, expressions are classified as terms and types, and in the remaining systems kinds are used as well. We refer to them as *two-levelled* and *three-levelled* type theories, respectively. Expressions of level n act as classifiers for expressions of level $n-1$. We represent such a classification relation with the signature *Level* with declarations cl : `type` for classifiers and $exp : cl \rightarrow$ `type` for the classifier-indexed type family of classified expressions.

In a two-levelled type theory, classifiers are types and expressions are terms. Therefore, the base signature for two-levelled type theories simply imports *Level* once, and uses cl for types and $exp\,A$ for terms of type A. Similarly, the base signature for three-levelled type theories imports *Level* twice; it is given on the right. Firstly, on the kind-level, LF-terms of LF-type $kind.cl$ represents kinds, and LF-terms of LF-type $kinds.exp\,K$ represent type families of kind K. The kind tp' represents

```
%sig KindsTypesTerms = {
  %struct kinds : Level.
  tp' : kinds.cl.
  %struct types : Level = {cl := kinds.exp tp'.}.
}.
```

the universe of types. Secondly, on the type-level, LF-terms of LF-type *types.cl*, which is instantiated with *kinds.exp tp′*, represent types, and LF-terms of LF-type *types.exp A* represent terms of type *A*.

(ii) We encode the (s_1, s_2)-rule in the signature *DepFun* on the right. We represent the parametricity of the (s_1, s_2)-rule by using two imports of the signature *Level* into *DepFun*: The LF-terms *domain.cl* and *scope.cl* represent s_1 and s_2, respectively. Then we can use LF's higher-order abstract syntax to represent the Π-formation rule as a constant Π that binds a free variable of type or kind (depending on *domain*) *A* in a type or kind (depending on *scope*). *DepFun* also contains the declarations for the corresponding λ-abstraction and application.

```
%sig DepFun = {
  %struct domain : Level.
  %struct scope : Level.
  Π : (domain.exp A → scope.cl) → scope.cl.
  ...
}.
```

For example, the signature *DepTypes* includes *KindsTypesTerms*, i.e., is a three-levelled type theory. We add the $(*,*)$-rule to it by importing *DepFun* and instantiating both *domain* and *scope* with the structure *types* included from *KindsTypesTerms*. Similarly, we can obtain the $(\Box,*)$-rule (polymorphism) using an import from *DepFun* which instantiates *domain* with *KindsTypesTerms.kinds* and *scope* with *KindsTypesTerms.terms*.

```
%sig DepTypes = {
  %include KindsTypesTerms.
  %struct deptypes : DepFun = {
    %struct domain := KindsTypesTerms.types.
    %struct scope  := KindsTypesTerms.types.
  }.
}.
```

(iii) We obtain each corner of the cube by combining the respective instances of *DepFun*. In particular, all four possible instances together yield the calculus of constructions.

In fact, our development is slightly more complicated. Some of the simpler systems cannot utilize the expressivity of their respective (s_1, s_2)-rules. For example, $\lambda\!\!\rightarrow$ has only the $(*,*)$-rule, and its types $\Pi x : A.B$ always degenerate into $A \to B$. To represent this, we use a signature *SimpFun*, which is like *DepFun* but uses $\to$ instead of Π, and give a view *SimpToDep* from *SimpFun* to *DepFun*.

Similarly, both function types and function kinds of $\lambda\underline{\omega}$ are non-dependent. While $\lambda\underline{\omega}$ as part of the cube uses two instances of *DepFun*, we get the same well-formed expressions with a signature $\lambda\underline{\omega}'$ that uses two instances of *SimpFun*. We can give a view from $\lambda\underline{\omega}'$ to $\lambda\underline{\omega}$ by reusing the view *SimpToDep* twice.

3.3 Logics

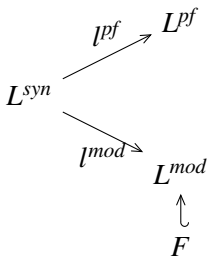
We can use the λ-cube encoding in LF to obtain modular representations of *higher-order logics* based on different type theories. We use a signature *HOL* which includes *SimpFun* and adds a type $o : tp$ for propositions and a truth judgment $ded : tm\ o \to \texttt{type}$. We can then develop HOL in the usual way, and we have done that both for Andrews' and Isabelle/HOL's choice of primitive operations.

```
%sig λPHOL = {
  %include λP.
  %struct hol : HOL = {
    %include SimpToDep.
  }.
}.
```

Now, using the module system, we can reuse *HOL* by replacing the underlying type theory. For example, the signature on the right develops HOL on top of λP. It includes λP and then instantiates *HOL*. As *HOL* includes *SimpFun*, we use the view *SimpToDep* to instantiate the underlying type theory of *HOL* with that of λP.

In general, one of the main goals in developing our Logic Atlas is to represent both proof and *model theory* of logics. To represent model theory in LF we use the approach developed in [Rab08] and [HR10].

Based on that approach, our encoding of a logic L in LF consists of three signatures: L^{syn} for the syntax, L^{pf} for the proof theory, and L^{mod} for the model theory as shown in the diagram on the right.



L^{syn} contains declarations for all logical symbols, e.g., a type o of formulas along with connectives and quantifiers. It also contains a truth judgment $ded : o \to \texttt{type}$. L^{pf} includes declarations for the judgments and inference rules of L. It is typically an extension of L^{syn}, i.e., l^{pf} is an inclusion morphism. Previous logic encodings in LF focused on giving L^{syn} and L^{pf}.

L^{mod} contains declarations that describe models. L^{mod} typically includes a formalization F of the foundation of mathematics in which the models are defined, e.g., set theory. Then a morphism l^{mod} interprets L^{syn} in L^{mod}. Here the inductive definition of the extension of l^{mod} to all L^{syn}-expressions corresponds to the inductive definition of the interpretation function induced by a model. In the simplest case l^{mod} translates o to a type of truth values and ded to a predicate that holds for the designated truth values.

For example, for first-order logic, FOL^{mod} includes Zermelo-Fraenkel set theory. ZFC introduces in particular a type *set* of sets, a type family $Elem : set \to \texttt{type}$ for elements of a set, and a set *bool* for the booleans. Then FOL^{mod} adds a set for the universe and constant declarations for the interpretation of the non-logical symbols; in the example on the right, the latter the case of a single unary predicate symbol p is given.

$$
\begin{array}{l}
\%\texttt{sig}\ FOL^{mod} = \{ \\
\quad \%\texttt{include}\ ZFC. \\
\quad univ \ : \ set. \\
\quad p \ : \ Elem\ (univ \Rightarrow bool). \\
\}.
\end{array}
$$

Then individual models are represented in LF as morphisms from FOL^{mod} to ZFC that are the identity on ZFC. In the example above, such a morphism must map *univ* to a set and p to a function from that set to *bool*. Given such a morphism M, the composition $l^{mod}\ M$ yields the interpretation of FOL^{syn} in M. This yields an adequate representation of models as LF signature morphisms. For example, we can now establish the soundness of the semantics as a view from FOL^{pf} to FOL^{mod}; here the inductive structure of a morphism corresponds to the inductive structure of the soundness proof.

3.4 Knowledge Management Aspects

Our Logic Atlas serves not only as a formalization of logics but also as a case study of how large scale modular developments can be conducted. Since Twelf was never designed to scale to this scenario, our work has raised a number of technical points about Twelf.

The central tool we employ for knowledge management is the OMDoc XML markup language and our scalable infrastructure for it [Koh06]. Therefore, we have implemented a conversion from Twelf to OMDoc within Twelf. In OMDoc, all non-trivial Twelf idiosyncrasies – e.g., operator fixities, implicit arguments, type reconstruction – are transformed into standardized syntax. At the same these idiosyncrasies are preserved as notations so that human-readable Twelf can be generated from OMDoc. Moreover, the modular structure is fully preserved. This makes it possible to implement dedicated knowledge management services without fearing to violate the Twelf semantics.

As a first application we have implemented a web server [KRZ10] that serves the logic graph as cross-linked interactive files. It can be accessed via [KMR09]. It is currently in prototypical stage but under constant development. It serves the LF modules in a way that lets users switch on off subexpressions, inferred types, implicit arguments, or redundant brackets. In particular, it permits to jump to the declaration of a symbol to look up type and definition. The latter is extremely valuable because almost all people working on the logic graph need to import modules they did not author themselves and were previously often puzzled when they had to determine the origin of a symbol.

A second problem was the emacs mode for Twelf. While running the Twelf server as an emacs buffer is an elegant design, it is rather slow and we rarely used the thus-provided interactivity. Therefore, we implemented a Twelf mode for the jEdit text editor, which redirects Twelf's output to a file. That ensued in a speed-up of up to 20 for files that do not contain Twelf meta-theory as in our case. Thus, almost all files type-check in less than a second or – if they require set theory – a few seconds. It can also produce the OMDoc file, and we intend to develop this mode into an interactive Twelf editor by using the OMDoc file to provide features such as cross-references directly in the editor.

Another aspect was the difficulty of using configuration (`.cfg`) files, which are used to provide the names of Twelf (`.elf`) files that should be type-checked. Due to the high degree of modularity, automated validation of files, e.g., when committing to the repository, is indispensable for scalable change management – especially with multiple people editing interconnected files. But it is not possible to validate a Twelf file because dependencies are only recorded in the configuration files so that we had to use a separate configuration file for almost every Twelf file.

As a more scalable, albeit temporary, solution, we implemented the pragma `%read "FILE".` to load Twelf files from within Twelf files. Thus, every Twelf file explicitly imports its dependencies, and configuration files are redundnat. The downside is that the distribution of resources over the file system is not separated from the logical content. In the long run, we envision a package infrastructure for Twelf in which top level names are pairs of a package name and a Twelf identifier. References to external resources can then be made by qualified names and imports between packages, which are resolved by an extra-linguistic catalog that maps package names to file names. In fact, this would even permit to retrieve individual modules or aggregated collections of modules from a database. Using the architecture described in [KRZ10], the server-side architecture for this is essentially in place, but we have deferred the implementation of a client within Twelf because it would require a major revision of the Twelf code.

4 Integrating Twelf and Hets

4.1 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. It uses the abstract model theory notions of institutions and institution comorphisms (see [GB92]) to represent logics and logic translations, respectively. Currently supported logics include modal logic, propositional logic, the CASL family of first- and higher-order logics [ABK$^+$02, SM09], DFOL [Rab06], and OWL [AH03]. Some of the Hets logics are connected to external provers – for instance SoftFOL [LM07], a version of first-order logic, is supported by SPASS [WBH$^+$02] and Vampire [RV02] whereas the higher-order logic Isabelle is connected to the semi-automated Isabelle prover [Pau94]. The proof systems in Hets have been utilized in several large case studies, e.g., [WMS07] uses Hets for the heterogeneous verification of composition tables for qualitative constraint calculi.

Hets provides a strong logic-independent support for structured specifications. Once a logic has been implemented, the user can use Hets to specify modular theories and reason about them on the modular level. The theoretical base of this is the concept of development graphs [MAH06]. A development graph has as nodes the flattened theories of the modular specification and as edges definitional or theorem links. These links intuitively correspond to the structures/inclusions and views of Twelf, respectively. The graph is heterogeneous, i.e., the nodes can be from different logics and the edges can include logic translations. Hets maintains the development graph, and users interact with the graph via a graphical interface.

The main application of Hets is to postulate theorem links, which are then automatically proved using the development graph calculus implemented in Hets. A postulated theorem link corresponds to an incomplete view in the Twelf module system where the missing cases are to be inferred by Hets. The calculus uses theory morphisms and logic translations to move proof obligations between theories and logics, respectively, and eventually discharges them using dedicated external (semi-)automated theorem provers. Translations are chosen automatically or by the user. Advanced reasoning features include information hiding to support reasoning about data encapsulation and proving conservativity of theory extensions.

Both Hets and LF are logical frameworks in the sense that they provide the theoretical and practical infrastructure to define logics. Moreover, the Hets logic graph is very similar in spirit to the LATIN atlas. 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 theoretical 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.

4.2 Combining LF and Hets

The above-mentioned differences between the LF and Hets exhibit complementary strengths, and a major goal of our work is to combine them. We envision a system in which LF is used as a component of Hets to define logics dynamically. The user can use LF to define logics by specifying their syntax, proof theory, and model theory as outlined in Sect. 3.3, and the combined system recognizes the new logics and integrates them into the Hets logic graph.

As a first step towards the integration of the two frameworks we have implemented LF as a logic in Hets. The implementation first specifies the LF signatures and signature morphisms; since axioms in LF are represented as ordinary Twelf declarations, LF sentences are not used at this point and are omitted in the logic implementation. The signature category then serves as a basis for the development graphs generated from Twelf specifications.

Hets reads Twelf files directly and transparently forwards them to Twelf, which does parsing and static analysis in order to benefit from type reconstruction and implicit arguments. If the file is correct, Twelf outputs the OMDoc version of the file that is then imported into Hets using standard XML technologies. The information about implicit arguments, fixities etc., are stored as well so that Hets can reproduce the original Twelf input syntax.

The subsequent analysis flattens all signatures and converts all morphisms to development graph links: inclusions and structures become definitional links and views become theorem links in the Hets development graph. Fig. 1 shows a screenshot of the Hets interface displaying the development graph corresponding to the λ-cube encodings described in Sect. 3.2. The oval nodes correspond to signatures defined in the current file. The square nodes represent signatures imported from other files, either as the domain of a structure or inclusion or the domain or codomain of a view. Green arrows between nodes correspond to theorem links, in our case views, and black arrows to definitional links, in our case structures and inclusions.

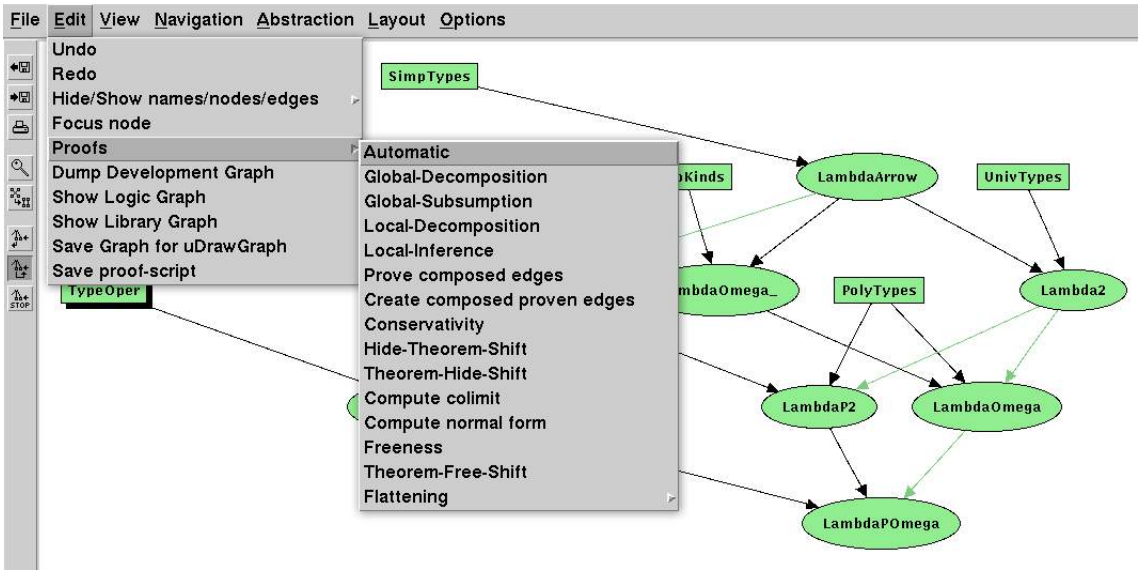


Figure 1: Hets Screenshot Displaying the λ-Cube

Currently LF views are always total, so that a successful Twelf type check proves that the instantiations in all views are correct. Hence, there are no undischarged proof obligations and the theorem links are already constructed as proved. In the future we intend to permit incomplete views, i.e., views in which certain instantiations – typically the costly instantiations where proofs must be given – are missing.

Then Hets can be used to find the missing proofs by identifying the open proof goals and forwarding them to external theorem provers. All we need for that is to tell Hets which signatures of the Twelf file form the triple $(L^{syn}, L^{pf}, L^{mod})$ from Sect. 3.3 so that Hets can determine which external provers are applicable or which logic translations to use in order to find an applicable prover.

In the long run, we envision that these provers 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.

5 Conclusion

The LATIN project can be seen as a continuation of the LogoSphere Project [Log06], which also had interoperability of logic-based computational systems as its aim, and has developed some of the logical framework foundations of the work reported here. But the work reported here concentrates more on modularity, scalability, and re-use of granular logic components to achieve more coverage of logical systems. Alternatively, LATIN can be viewed as a foundational part of the OMDoc project [OMD] that aims to bring interoperability to systems by offering a web-scalable logic-transparent content markup infrastructure to mathematics. We maintain that the strength our work is that it is both combining the virtues of both approaches. Indeed we have used the OMDoc format to simplify the Hets integration by getting around surface syntax issues. Note that this uses the XML-based OMDoc format how it should be used: invisible to the user, but as a facilitator of system integration.

The main contribution of this paper is the systematic attempt to obtain a comprehensive and inter-

connected network of logic formalizations, which we call the Logic Atlas. 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 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 importing the aspects that are provided by Logic Atlas theories, and LF-encoding the aspects that are not.

Acknowledgments This paper mainly addresses the proof-theoretic side of the logic atlas developed in the LATIN project — funded by the German Research Council (DFG) under grant KO-2428/9-1. The work reported here profited significantly from our colleagues Till Mossakowski and Mihai Codescu from the DFKI side of the project. Substantial contributions to the LF encodings were made by our students Stefania Dumbrava, Alin Iacob, and Mihnea Iancu.

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.

[AH03] G. Antoniou and F. Van Harmelen. Web Ontology Language: OWL. In *Handbook on Ontologies in Information Systems*, pages 67–92. Springer, 2003.

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

[Bar92] H. Barendregt. Lambda calculi with types. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2. Oxford University Press, 1992.

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

[BMV98] D. Basin, S. Matthews, and L. Viganò. A Modular Presentation of Modal Logics in a Logical Framework. In J. Ginzburg and Z. Khasidashvili and C. Vogel and J. Levy and E. Vallduvi, editor, *Proceedings of the 1st Tbilisi Symposium on Language, Logic and Computation: Selected Papers*, pages 293–307, 1998.

[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.

[CH88] T. Coquand and G. Huet. The Calculus of Constructions. *Information and Computation*, 76(2/3):95–120, 1988.

[Chu36] A. Church. An Unsolvable Problem of Elementary Number Theory. *American Journal of Mathematics*, 58, 1936.

[dL92] Gerard R. Renardel de Lavalette. Strictness analysis via abstract interpretation for recursively defined types. *Inf. Comput.*, 99(2):154–177, 1992.

[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.

[Gir71] J. Girard. Une extension de l'interpretation de Gödel à l'analyse, et son application à l'élimination des coupures dans l'analyse et la théorie des types. In J. Fenstad, editor, *2nd Scandinavian Logic Symposium*, pages 63–92. North-Holland, 1971.

[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.

[HR10] F. Horozal and F. Rabe. Representing Model Theory in a Type-Theoretical Logical Framework. Under review, see `http://kwarc.info/frabe/Research/EArabe_folsound_10.pdf`, 2010.

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

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

[Koh06] M. Kohlhase. *OMDoc: An Open Markup Format for Mathematical Documents (Version 1.2)*. Number 4180 in Lecture Notes in Artificial Intelligence. Springer, 2006.

[KRZ10] M. Kohlhase, F. Rabe, and V. Zholudev. Towards MKM in the Large: Modular Representation and Scalable Software Architecture. submitted, 2010.

[LM88] Giuseppe Longo and Eugenio Moggi. Constructive natural deduction and its 'modest' interpretation. Technical Report CMU-CS-88-131, Computer Science Department, Carnegie Mellon University, Pittsburgh, Pennsylvania, 1988.

[LM07] K. Lüttich and T. Mossakowski. Reasoning Support for CASL with Automated Theorem Proving Systems. In J. Fiadeiro, editor, *Workshop on Algebraic Development Techniques 2006*, volume 4409 of *LNCS*, pages 74–91. Springer, 2007.

[Log06] Logosphere: a formal digital library. web page at `http://www.logosphere.org/`, seen November2006a 2006.

[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.

[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.

[OMD] OMDoc. web page at `http://omdoc.org`.

[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.

[PD10] B. Pientka and J. Dunfield. A Framework for Programming and Reasoning with Deductive Systems (System description). In *International Joint Conference on Automated Reasoning*, 2010. To appear.

[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/`.

[Rab06] F. Rabe. First-Order Logic with Dependent Types. In N. Shankar and U. Furbach, editors, *Proceedings of the 3rd International Joint Conference on Automated Reasoning*, volume 4130 of *LNCS*, pages 377–391. Springer, 2006.

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

[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.

[RV02] A. Riazanov and A. Voronkov. The design and implementation of Vampire. *AI Communications*, 15:91–110, 2002.

[SM09] L. Schröder and T. Mossakowski. HasCASL: Integrated Higher-Order Specification and Program Development. *Theoretical Computer Science*, 410(12-13):1217–1260, 2009.

[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.

[WBH+02] C. Weidenbach, U. Brahm, T. Hillenbrand, E. Keen, C. Theobalt, and D. Topic. SPASS Version 2.0. In A. Voronkov, editor, *Conference on Automated Deduction*, pages 275–279. Springer, 2002.

[WMS07] S. Wölfl, T. Mossakowski, and L. Schröder. Qualitative Constraint Calculi: Heterogeneous Verification of Composition Tables. In D. Wilson and G. Sutcliffe, editors, *20th International FLAIRS Conference (FLAIRS-20)*, pages 665–670. AAAI Press, 2007.

[ZKR10] V. Zholudev, M. Kohlhase, and F. Rabe. A [insert XML Format] Database for [insert cool application]. In *Proceedings of XMLPrague*, 2010. To appear.