HAL open science

Compiling Logics

Mihai Codescu, Fulya Horozal, Aivaras Jakubauskas, Till Mossakowski, Florian Rabe

To cite this version:

Mihai Codescu, Fulya Horozal, Aivaras Jakubauskas, Till Mossakowski, Florian Rabe. Compiling Logics. 21th InternationalWorkshop on Algebraic Development Techniques (WADT), Jun 2012, Salamanca, Spain. pp.111-126, 10.1007/978-3-642-37635-1_7 . hal-01485981

HAL Id: hal-01485981

https://inria.hal.science/hal-01485981v1

Submitted on 9 Mar 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

Compiling Logics

Mihai Codescu[1], Fulya Horozal[3], Aivaras Jakubauskas[3], Till Mossakowski[2], and Florian Rabe[3]

[1] Friedrich-Alexander University, Erlangen-Nürnberg, Germany
[2] DFKI GmbH Bremen, Germany
[3] Computer Science, Jacobs University Bremen, Germany

Abstract. We present an architecture that permits compiling declarative logic specifications (given in some type theory like LF) into implementations of that logic within the Heterogeneous Tool Set Hets. The central contributions are the use of declaration patterns for singling out a suitable subset of signatures for a particular logic, and the automatic generation of datatypes and functions for parsing and static analysis of declaratively specified logics.

1 Introduction

In [5], we presented an extension of the Heterogeneous Tool Set HETS [14] with a framework for representing logics independently of their foundational assumptions. The key idea [17] is that a graph of theories in a type theoretical logical framework like LF [9] can fully represent a model theoretic logic. Our integration used this construction to make the process of extending HETS with a new logic more declarative, on one side, and fully formal, on the other side.

However, the new logic in HETS inherits the syntax of the underlying logical framework. This is undesirable for multiple reasons. Firstly, a logical framework unifies many concepts that are distinguished in individual logics. Examples are binding and application (unified by higher-order abstract syntax), declarations and axioms (unified by the Curry-Howard correspondence), and different kinds of declarations (unified by LFP – LF with declaration patterns). Therefore, users of a particular logic may find it unintuitive to use the concrete syntax (and the associated error messages) of the logical framework.

Secondly, only a small fragment of the syntax of the logical framework is used in a particular logic. For example, first-order logic only requires two base types *term* for terms and *form* for formulas and not the whole dependent type theory of LF. Therefore, it is unnecessarily complicated if implementers of additional services for a particular logic have to work with the whole abstract syntax of the logical framework. Such services include in particular logic translations from logics defined in LF to logics implemented by theorem provers.

Therefore, we introduce an architecture that permits compiling logics defined in LF into custom definitions in arbitrary programming languages. This is similar to parser generators, which provide implementations of parsers based on a language definition in a context-free grammar. Our work provides implementations of a parser and a type-checker based on a context-sensitive language

definition in the recent extension of LF with *declaration patterns* [11] (LFP). Here declaration patterns give a formal specification of the syntactic shape of the declarations in the theories of a logic.

For realising this approach, we will use the MMT framework, which provides a scalable Module system for Mathematical Theories [19] independently of the logic and foundation. The MMT tool provides an API for parsing, checking, and flattening MMT theories. We will build our implementation on this, and use it in particular for modular theories in LFP.

2 Preliminaries

2.1 Institutions and Hets

The Heterogeneous Tool Set (Hets, [14]) 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 [1] and uses the development graph calculus [13] as a proof management component.

Hets formalizes the logics and their translations using the abstract model theory notions of institutions and institution comorphisms (see [6] and [7]).

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* Σ *its set of* sentences $\mathbf{Sen}(\Sigma)$ *and for each 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* φ*);*
- $\mathbf{Mod} : \mathbf{Sig}^{op} \to$ *Cat is a functor to the category of categories and functors Cat* [4] *giving for each signature* Σ *its category of models* $\mathbf{Mod}(\Sigma)$ *and for each 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* Σ

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.*

For example, the institution of propositional logic *PL* has signatures consisting of a set of propositional symbols, and signature morphisms are just functions between those sets. Models are functions from the signature to the set of truth values $\{true, false\}$ giving the interpretation of each proposition, and sentences

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

are defined inductively, starting with the propositional symbols and applying a finite number of Boolean connectives. Sentence translation means replacement of the translated symbols. The reduct of a Σ'-model $m : \Sigma' \to \{true, false\}$ along a signature morphism $\varphi : \Sigma \to \Sigma'$ is just the composition $\varphi; m : \Sigma \to \{true, false\}$. Finally, satisfaction is given by the standard truth-tables semantics and it is straightforward to see that the satisfaction condition holds.

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 $\beta : \Phi; \mathbf{Mod}_2 \Rightarrow \mathbf{Mod}_1$ and $\alpha : \mathbf{Sen}_1 \Rightarrow \Phi; \mathbf{Sen}_2$, such that the following satisfaction condition holds:*

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

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

Hets has been designed as an extensible tool: new institutions can be plugged in without having to modify the institution-independent parts. Hets implements institutions in Haskell using a multiparameter type class [12] with functional dependencies. Functional dependencies are needed because no operation will involve all types of the multiparameter type class; hence we need a method to derive the missing types.

```
class Logic lid sublogics sign mor sen basic_spec symb_map
        | lid -> sublogics sign mor sen basic_spec symb_map
   where
     logic_name :: lid -> String
     id :: lid -> sign -> mor
     comp :: lid ->  mor -> mor -> mor
     parse_basic_spec :: lid -> String -> basic_spec
     parse_symb_map :: lid -> String -> symb_map
     map_sen :: lid -> mor -> sen -> sen
     basic_analysis :: lid -> sign -> basic_spec
                  -> (sign, [sen])
     stat_symb_map :: lid -> symb_map -> sign -> mor
     minSublogic :: lid -> sublogics -> sign -> Bool
     minSublogic :: lid -> sublogics -> sen -> Bool
```

Fig. 1. The basic components of a logic in Hets.

The `Logic` class of Hets is presented in Fig. 1, in a very simplified form (e.g., error handling is omitted). For each logic, we introduce a new singleton type `lid` that gives the name (`logic_name`), or constitutes the identity of the logic. All other parameters of the type class depend on this type, and all operations

take it as first argument. The types `basic_spec` and `symb_map` serve as a more user-friendly syntax for signatures and morphisms, respectively. The methods of the type class `Logic` give then the definition of the category of signatures and signature morphisms. Parsers for basic specifications and symbol maps must be provided, as well as functions for static analysis, that transforms basic specifications into theories of the logic and symbol maps into signature morphisms, respectively. We also have a function that gives the translation of a sentence along a signature morphism. Finally, Hets includes a sublogic analysis mechanism, which is important because it reduces the number of logic instances. A sublogic can use the operations of the main logic and functions are available that give the minimal sublogic of an item (methods `minSublogic`).

2.2 The LATIN Meta-Framework

Logical Frameworks in LATIN The Logic Atlas and Integrator (LATIN) project [4] develops a foundationally unconstrained framework for the representation of institutions and comorphism [17]. It abstract from individual logical frameworks such as LF [9] or Isabelle [15] by giving a general definition of a logical framework, called the LATIN meta-framework [5].

The central component of a logical framework $\mathbb{F}$ in the sense of LATIN is a category (whose components are called signatures and signature morphisms) with inclusions. LATIN follows a "logics-as-theories and translations-as-morphisms" approach, providing a general construction of an institution from a $\mathbb{F}$-signature and of an institution comorphism from a $\mathbb{F}$-morphism [5].

The basic idea is that, given an $\mathbb{F}$-signature L, the signatures of the institution $\mathbb{F}(L)$ are the extensions $L \hookrightarrow \Sigma$ in $\mathbb{F}$. Similarly morphisms from $L \hookrightarrow \Sigma$ to $L \hookrightarrow \Sigma'$ are the commuting triangles formed by morphism $\sigma : \Sigma \to \Sigma'$. The $\mathbb{F}(L)$-sentences over Σ are obtained as certain Σ-objects in $\mathbb{F}$, and the $\mathbb{F}(L)$-models of Σ are obtained as certain $\mathbb{F}$-morphisms out of Σ. We refer to [5] for the details.

The Logical Framework LFP For the purposes of this paper, we will use a logical framework $\mathbb{F}$ = LFP based on an extension of modular LF [9,20] with declaration patterns. Here we will briefly introduce LFP and direct the reader to [11] for the details on declaration patterns.

Figure 2 gives the fragment of the grammar for LFP-theories that is sufficient for this paper. Here the parts pertaining to declaration patterns are <u>underlined</u>. In particular, LFP-signatures and morphisms are produced from Σ and σ, respectively.

Let us first consider the language without declaration patterns. Modules are the toplevel declarations. Their semantics is defined in terms of the category of LF *signatures* and signature morphisms (see, e.g., [10]); the latter are called *views* in modular LF.

A non-modular signature Σ declares a list of typed constants c. Correspondingly, views from a signature T_1 to a signature T_2 consist of assignments $c := E$, which map T_1-constants to T_2-expressions.

Modules	M	$::= \%\texttt{sig}\ T\ =\ \{\Sigma\} \mid \%\texttt{view}\ v : T_1 \to T_2\ =\ \{\sigma\}$
Theories	Σ	$::= c : E \mid \%\texttt{include}\ T \mid \underline{\%\texttt{pattern}\ p = P}$
Morphisms	σ	$::= c := E \mid \%\texttt{include}\ v \mid \underline{\%\texttt{pattern}\ p\ :=\ P}$
Expressions	E	$::= \texttt{type} \mid c \mid x \mid \{x : E\}\, E \mid [x : E]\, E \mid E\, E$
		$\mid\ \underline{E, E} \mid \underline{[E]^{E}_{x=1}} \mid \underline{E_E} \mid \underline{Nat} \mid \underline{0} \mid \underline{succ(E)}$
$\underline{\text{Patterns}}$	$\underline{P}$	$::= \underline{p} \mid \underline{\{\Sigma\}} \mid \underline{[x : E]\, P} \mid \underline{P\, E}$

Fig. 2. Grammar of LFP

Expressions are formed from the universe of types `type`, constants c, bound variables x, dependent function types (Π-types) $\{x : E\}\, E$, λ-abstraction $[x : E]\, E$, and application $E\, E$. As usual, we write $E_1 \to E_2$ instead of $\{x : E_1\}\, E_2$ whenever x does not occur in E_2. A valid view extends homomorphically to a (type-preserving) map of all T_1-expressions to T_2-expressions.

To this, the module system adds the ability for signatures *include* other signatures. Morphisms out of such signatures must correspondingly include a morphism.

Example 1 (First-order logic in LF). The LF signatures below encode propositional logic (PL) and first-order logic (FOL) in LF. We will use these encodings as our running example in this paper.

$$
\begin{array}{l}
\%\texttt{sig}\ \mathit{Base}\ =\ \{ \\
\quad \mathit{form} : \texttt{type} \\
\quad \mathit{ded}\ \ : \mathit{form} \to \texttt{type} \\
\} \\
\%\texttt{sig}\ \mathit{PL}^{Syn}\ =\ \{ \\
\quad \%\texttt{include}\ \mathit{Base} \\
\quad \mathit{false} : \mathit{form} \\
\quad \mathit{imp}\ \ : \mathit{form} \to \mathit{form} \to \mathit{form} \\
\} \\
\%\texttt{sig}\ \mathit{FOL}^{Syn}\ =\ \{ \\
\quad \%\texttt{include}\ \mathit{PL}^{Syn} \\
\quad \mathit{term} : \texttt{type} \\
\quad \forall\quad : (\mathit{term} \to \mathit{form}) \to \mathit{form} \\
\}
\end{array}
$$

The signature *Base* declares an LF-type *form* of propositional formulas and a *form*-indexed type family *ded*. This type family exemplifies how logic encodings in LF follow the Curry-Howard correspondence to represent judgments as types and proofs as terms: Terms of type $\mathit{ded}\, F$ represent derivations of the judgment "F is true". Furthermore, PL^{Syn} encodes the syntax of PL by declaring propositional connectives (here only falsehood and implication). Finally, we obtain the FOL syntax in the signature FOL^{Syn} by adding to PL^{Syn} a constant *term* for

the type of terms and the universal quantifier $\forall$. The latter is a binder and thus is represented in LF using higher-order abstract syntax.

Then the LF signature PL^{Pf} below encodes the proof theory of PL by providing the inference rules associated with each symbol declared in PL^{Syn}:

$$
\begin{array}{l}
\%\texttt{sig}\ PL^{Pf}\ = \{ \\
\quad \mathit{falseE} : \mathit{ded}\ \mathit{false} \to \{F\}\ \mathit{ded}\ F \\
\quad \mathit{impI}\ \ : (\mathit{ded}\ A \to \mathit{ded}\ B) \to \mathit{ded}\ (A\ \mathit{imp}\ B) \\
\quad \mathit{impE}\ \ : \mathit{ded}\ (A\ \mathit{imp}\ B) \to \mathit{ded}\ A \to \mathit{ded}\ B \\
\}
\end{array}
$$

We obtain the proof theory of first-order logic accordingly.

Declaration Patterns Let us now consider the extension of modular LF with declaration patterns. Declaration patterns formalize what it means to be an arbitrary theory of a logic L: A declaration pattern gives a formal specification of the syntactic shape of a class of L-declarations, and in a legal L-theory, each declaration must match one of the L-patterns.

Declaration patterns P are formed from pattern constants p, signatures $\{\Sigma\}$, λ-abstractions $[x : E]\, P$, and applications $P\, E$ of patterns P to expressions E.

Example 2 (Declaration patterns for PL). Consider the following version of the signature PL^{Syn} with declaration patterns for legal PL-signatures.

$$
\begin{array}{l}
\%\texttt{sig}\ PL^{Syn}\ = \{ \\
\quad \%\texttt{include}\ \mathit{Base} \\
\quad \mathit{false} : \mathit{form} \\
\quad \mathit{imp}\ \ : \mathit{form} \to \mathit{form} \to \mathit{form} \\
\quad \%\texttt{pattern}\ \mathit{props}\ =\ \{ \\
\qquad q : \mathit{form} \\
\quad \} \\
\quad \%\texttt{pattern}\ \mathit{axiom}\ =\ [F : \mathit{form}]\ \{ \\
\qquad m : \mathit{ded}\ F \\
\quad \} \\
\}
\end{array}
$$

The declaration pattern *props* allows for the declaration of propositional variables of the form $q : \mathit{form}$ in PL-signatures. And the declaration pattern *axiom* formalizes the shape of axiom declarations. Each axiom declaration must be of the form $m : \mathit{ded}\ F$ for some proposition F.

More technically, consider an LFP-signature $L = L_S, L_P$ where L_S is a signature of modular LF and L_P is a list of pattern declarations. Then we define a morphism $L \hookrightarrow \Sigma$ to exist if $L_S \hookrightarrow \Sigma$ is an inclusion in modular LF and every declaration in $\Sigma \setminus L_S$ matches one of the patterns in L_P. Consequently, the institution LFP(L) contains exactly the "well-patterned" signatures.

Sequences Even though most logics do not use sequences, it turns out that sequences are usually necessary to write down declaration patterns. For example, in theories of typed first-order logic, function symbol declarations use a sequence of types – the argument types of the function symbol. More precisely, in the case of FOL, the signatures Σ should contain only declarations of the form $f : \mathit{term} \to \ldots \to \mathit{term} \to \mathit{term}$ (n-ary function symbols) and $p : \mathit{term} \to \ldots \to \mathit{term} \to \mathit{form}$ (n-ary predicate symbols). Therefore, our language also uses *expression sequences* and *natural numbers*. These are formed by the underlined productions for expressions:

- E_1, E_2 for the concatenation of two sequences,
- E_n for the n-th element of E,
- $[E(x)]_{x=1}^{n}$ for the sequence $E(1), \ldots, E(n)$ where n has type Nat and $E(x)$ denotes an expression E with a free variable $x : \mathit{Nat}$; we write this sequence as E^n whenever x does not occur free in E,
- Nat for the type of natural numbers,
- 0 and $\mathit{succ}(n)$ for zero and the successor of a given natural number n.

Example 3 (Declaration patterns for FOL). Using sequences, we can give the following version of the signature FOL^{Syn} that declares appropriate declaration patterns:

```
%sig FOL^Syn = {
   %include PL^Syn
   term : type
   ∀    : (term → form) → form
   %pattern ops = [n : Nat] {
      f : term^n → term
   }
   %pattern preds = [n : Nat] {
      q : term^n → form
   }
}
```

The declaration pattern *ops* allows for the declaration of function symbols of the form $f : \mathit{term}^n \to \mathit{term}$, which take a sequence of first-order terms of length n and return a first-order term for any natural number n. Similarly, the declaration pattern *preds* allows for the declaration of predicate symbols of the form $q : \mathit{term}^n \to \mathit{form}$ for any natural number n.

More complex patterns arise if we consider sorted first-order logic (SFOL):

Example 4 (Declaration patterns for Sorted FOL). SFOL^{Syn} is similar to FOL^{Syn} except that we use a type *sort* to encode the set of sorts and an LF type family *tm* indexed by *sort* that provides the type $\mathit{tm}\, S$ of terms of sort S (i.e., $t : \mathit{tm}\, S$ is a declaration of a term t of sort S). Universal quantification $\forall$ is sorted, i.e., it first takes a sort argument S and then binds a variables of type $\mathit{tm}\, S$.

Now the declaration pattern *sorts* allows for the declaration of sorts $s : \mathit{sort}$. The declaration patterns *sortedOps* and *sortedPreds* formalize the

shape of declarations of sorted function and predicate symbols of the form $f : tm\, s_1 \to \ldots \to tm\, s_n \to tm\, t$ and $q : tm\, s_1 \to \ldots tm\, s_n \to form$, respectively, for any sort $s_1 : sort, \ldots, s_n : sort$ and $t : sort$. Note that we can extend all left or right associative infix operators to sequences: The sequence $[tm\, s_i]_{i=1}^n$ normalizes to $tm\, s_1, \ldots, tm\, s_n$ and the type $[tm\, s_i]_{i=1}^n \to tm\, t$ normalizes to $tm\, s_1 \to \ldots \to tm\, s_n \to tm\, t$. Correspondingly, for a function f of that type and a sequence E that normalizes to $E_1, \ldots, E_n$, the expressions $f\, E$ normalizes to $f\, E_1 \ldots E_n$.

```
%sig SFOL^Syn = {
  %include PL^Syn
  sort : type
  tm   : sort → type
  ∀    : {S : sort} (tm S → form) → form
  %pattern sorts = {
     s : sort
  }
  %pattern sortedOps = [n : Nat] [s : sort^n] [t : sort] {
     f : [tm s_i]_{i=1}^n → tm t
  }
  %pattern sortedPreds = [n : Nat] [s : sort^n] {
     q : [tm s_i]_{i=1}^n → form
  }
}
```

We avoid giving the type system for this extension of LF and refer to [11] for the details. Intuitively, natural numbers and sequences occur only in pattern expressions, and fully applied pattern expressions normalize to expressions of the form $\{\Sigma\}$ where Σ is a plain LF signature.

3 Simple Signatures

In principle, the category of LFP(L) is already defined by giving an LFP-signature L such as FOL^{Syn}. However, the resulting LFP(L)-signatures are defined in terms of LFP-signatures, which is often more complex than desirable in practice.

Therefore, we introduce simple LFP-signatures below. These form a subclass of LFP-signatures that covers all the typical cases for the syntax of logics. At the same time, they are so restricted that they can be described in a way that is closer to what would appear in a stand-alone definition of a logic (i.e., in a setting without a logical framework).

First we need an auxiliary definition about LF types:

Definition 3. *A type is called* atomic *if it is of the form* $t\, a_1 \ldots, a_n$ *and* composed *if it is of the form* $\{x : A\}\, B$.

The dependency erasure T^- *of a type* T *is defined by* $(t\, a_1 \ldots, a_n)^- = t$ *and* $(\{x : A\}\, B)^- = A^- \to B^-$.

Intuitively, dependency erasure turns all dependent type families $t : A \to \texttt{type}$ into simple types $t : \texttt{type}$ and all dependent function types into simple function types. For example, the dependency erasure of the $SFOL^{Syn}$ declarations tm and $\forall$ yields $tm : \texttt{type}$ and $\forall : sort \to (tm \to form) \to form$, respectively. In particular, applying the dependency erasure everywhere turns an LF signature into a simply-typed signature.

Then we define simple signatures and their constituents as follows:

Definition 4. *Let N be the set of identifiers declared in an* LFP*-signature L. Let $c \in N$ be the identifier of a type family.*

A declaration $n : T$ in L is called

- *a* connective *for c with arguments $a_1, \ldots, a_n \in N$ if $T^- = a_1 \to \ldots \to a_n \to c$.*
- *an* untyped quantifier *for c quantifying over $b \in N$ with scope $s \in N$ if $T = T^- = (b \to s) \to c$.*
- *a* typed quantifier *for c with argument $a \in N$ quantifying over $b \in N$ with scope $s \in N$ if $T^- = a \to (b \to s) \to c$ and $T = \{x : A\}\,(b\,x \to C) \to S$ (i.e., A, C, and S must be atomic and $A^- = a$, $C^- = c$, and $S^- = s$).*

A pattern declaration is called simple *if it is of the form* %`pattern` $p := [x_1 : E_1] \ldots [x_n : E_n]\,\{\Sigma\}$ *such that (i) each E_i is either atomic or of the form $[E(x)]_{x=1}^n$ for atomic E and (ii) Σ declares only connectives. Then the connectives in Σ are called* p-connectives.

Finally, L is called simple *if it declares only connectives, untyped and typed quantifiers, and simple patterns.*

In particular, our running examples are simple:

Example 5. PL^{Syn}, FOL^{Syn} and $SFOL^{Syn}$ are simple signatures. In PL^{Syn}, *imp* is a connective for *form* with arguments *form*, *form*. In FOL^{Syn}, $\forall$ is an untyped quantifier for *form* quantifying over *term* with scope *form*. In $SFOL^{Syn}$, $\forall$ is a typed quantifier for *form* with argument *sort* quantifying over *tm* with scope *form*. Also in $SFOL^{Syn}$, *sortedOps* is a simple pattern, and f is an *sortedOps*-connective with arguments $tm, \ldots, tm$.

4 Compiling Simple Signatures to Hets

In [5], we introduced an extension of Hets with a component that generates an instance of the type class `Logic` from its representation in a logical framework.

This generation is conceptually straightforward by reducing all necessary operations to the ones of LF. As described in the introduction, this reduction to the logical framework leads to the counter-intuitive identification of many notions. For example, for the syntax of FOL^{Syn}, it generates a logic with *i)* a single Haskell type for expressions (not distinguishing between formulas and terms), *ii)* one Haskell constructor for each logical symbol (not distinguishing

between connectives and quantifiers) and *iii)* a single type of FOL-declarations (not distinguishing function and predicate symbols).

By restricting attention to simple signatures, we can obtain a – much more complex – algorithm that generates Haskell classes that avoid these identifications. In the following, we describe this algorithm using our running examples. Throughout the following, we assume a fixed simple signature L with one distinguished identifier o (designating the type of formulas).

Expressions For every declaration of a type family with name C in L, we generate one inductive data type named C whose constructors are obtained from L as follows:

- for every connective n for C with arguments $a_1, \ldots, a_n$: a constructor n with arguments $a_1, \ldots, a_n$,
- for every p-connective n for C with arguments $a_1, \ldots, a_n$: a constructor p_n with arguments `String` (intuition: the name of the matching declaration), $a_1, \ldots, a_n$,
- for every untyped quantifier n for C quantifying over b with scope s: a constructor n with arguments `String` (intuition: the name of the bound variable) and s,
- for every typed quantifier n for C with argument a quantifying over b with scope s: a constructor n with arguments a, `String` (intuition: the name of the bound variable), and s,
- if there is any quantifier over C: a constructor `C_var` with argument `String` (intuition: references to bound variables).

For example, in the case of FOL^{Syn}, this yields the following Haskell types [5] [6]

```
data Form  =
   False |
   Imp Form Form |
   Forall String Form |
   Preds_q [Term]

data Term =
   Ops_f [Term]
```

As expected, sentences are mapped along a signature morphism homomorphically on the structure of the sentence, by replacing each symbol according to the corresponding declaration map that composes the morphism.

[5] Here and below, we ignore minor syntactical issues, e.g., when compiling into Haskell, an additional step makes all identifiers begin with an upper case letter.

[6] `[T]` is Haskell's notation for the type of lists over `T`.

Declarations For every declaration of a pattern, we generate one record type with the following selectors: *i)* a selector returning `String` (intuition: the name of the matching declaration), *ii)* for each bound variable $x : E$ of the pattern: one selector of the same name returning *i)* E^- if E is atomic or *ii)* $[E'^-]$ if $E = [E'(x)]_{x=1}^n$.

For example, %`pattern` $\mathit{props} = \{p : \mathit{form}\}$ from PL^{Syn} generates the Haskell type

```
data Props_decl = Props_d {name :: String}
```

The declaration patterns %`pattern` $\mathit{ops} = \lambda n : \mathit{Nat}.\ \{f : \mathit{term}^n \to \mathit{term}\}$ and %`pattern` $\mathit{preds} = \lambda n : \mathit{Nat}.\ \{p : \mathit{term}^n \to \mathit{form}\}$ from FOL^{Syn} generate:

```
data Ops_decl = Ops_decl {
   name :: String,
   n    :: Int
}
data Preds_decl = Preds_decl {
   name :: String,
   n    :: Int
}
```

At the specification level, we get a new type of declarations for axioms, using the pattern in the `Base` signature:

```
data Axiom_decl = Axiom_d {
    name :: String,
    formula :: Form
}
```

Signatures Finally, we define a type of declarations as the disjoint union of the types generated for the declaration patterns and a type of signatures as lists of declarations. For example, for FOL^{Syn}, this yields the Haskell types:

```
data Decl = Ops_d Ops_decl | Preds_d Preds_decl
data Sign = Sign {decls :: [Decl]}
```

The assumption that signatures are lists of declarations is typical of logical frameworks like LFP. This is not a substantial restriction because many logics directly have such signatures, and even relations like a subsort relation can be coded as special declarations. Hets makes extensive use of equality tests between signatures. In our setting, we generate code that reports two signatures to be equal if they both have the same declarations, perhaps in a different order.

Similarly, signature morphisms have a source and a target signature and maps between declarations of the same kind. In the case of first-order logic, this amounts to:

```
data Morphism = Morphism {
  ssign, tsign :: Sign,
```

```
  ops_map :: Map Ops_decl Ops_decl,
  preds_map :: Map Preds_decl Preds_decl
}
```

This construction of signature morphisms can be applied for logics whose signatures are collections of sets of symbols of the same kind. In particular, this is the case for the logics in the logic graph of Hets. As a more general criterion, the construction covers signatures that can be represented as tuple sets [8]. There are however institutions that fall outside this framework. For example, the signatures of institutions for UML [3] are based on graphs rather than sets, and therefore are difficult to represent in LF.

External Syntax The above steps of the compilation have generated the internal (abstract) syntax. The generation of data types for external (concrete) syntax and for parsing function is substantially more complex. Therefore, we use a logic-independent representation that can be implemented (in our case, as a part of Hets) once and for all.

Expressions are represented as syntax trees, which are inspired by OpenMath objects [2]:

```
data Tree =
            Var String |
            App String [Tree] |
            Bind String String Tree |
            TBind String String Tree Tree
```

The constructors represent references to bound variables, application of a connective to a list of arguments, untyped quantifier applications, and typed quantifier applications. This data type differs crucially from the above in that it is untyped, i.e., all expressions have the same type, which makes it logic-independent. At the same time this lack of typing does not impede sophisticated parsing methods (which often do not depend on type information), e.g., by using mixfix notations for connectives or complex notations for binders. As a default notation, our implementation assumes that connectives are applied as $(c\, t_1 \ldots t_n)$, and quantifiers as $q\, x : t.\, t'$ where $: t$ is omitted if q is untyped.

The logic-independent representation of declarations is as follows:

```
data ParseDecl = ParseDecl {
   pattern   :: String,
   name      :: String,
   arguments :: [Tree]
  }
```

Here `name` is the name of the declaration, `pattern` is the name of the instantiated declaration pattern, and `arguments` gives the list of arguments to that declaration pattern. The external syntax for signatures is a list of such `ParseDecl`.

In concrete syntax, we use the name of the declaration pattern as a keyword for the declaration. Thus, when defining a logic, we choose the names of the

declaration patterns accordingly, e.g., *ops* and *preds* in FOL^{Syn} are chosen to mimic the Hets syntax for CASL. For example, the SFOL specification of a magma would look like as follows:

```
logic SFOL
spec Magma =
 sorts u
 ops comp 2 u u u
```

Similarly, we might use • instead of *axiom* as the name of the declaration pattern for axioms in order to mimic the Hets syntax of axioms.

Parsing functions for these data types are easy to write in Hets. Thus, the compilation process only generates logic-specific functions that translate from external to internal syntax. Comprehensive type-checking is performed separately by the MMT framework as described in Section 5.

Sublogics For the sublogic analysis, we make use of the modular structure of the logic representation in LFP. Ideally, every feature of the logical language is introduced in a separate module (say L_i^{Syn}) that can be later re-used. As a result, the signature representing the syntax of the logic contains a number of imports, e.g.:

%sig L^{Syn} = {
  %include L_1^{Syn}
  ⋮
  %include L_n^{Syn}
}

We can then generate a data type for sublogics with Boolean flags that indicate whether a feature is present in the sublogic or not:

```
data L_SL = L_SL {
   has_L_1 :: Bool,
   ...,
   has_L_n :: Bool
}
```

Thus each of the possible combinations of signatures included in `L_syn` gives rise to a lattice of sublogics of the logic. Note that with this approach we cannot represent sublogics based on certain restrictions made on the structure of sentences, e.g. the Horn fragment of a logic.

5 Workflow

For parsing and static analysis, we use the MMT tool [18,16], which already provides logic-independent parsing and type checking. In fact, the logic compilation itself is implemented as a part of MMT already.

Thus, we obtain two workflows. Firstly, the logic-level workflow occurs when Hets reads a new logic definition, e.g.,

```
logic LFP
spec SFOL^Syn = ...

newlogic SFOL =
  meta LFP
  syntax SFOL^Syn
  proofs ...
  models ...
  foundation ...
end
```

Here $SFOL^{Syn}$ is defined in Hets using the logical framework LFP, and then $SFOL^{Syn}$ it is used as a part of the definition of the logic `SFOL`. Such logic definitions have the advantage that the logic becomes a fully formal object, verified within the logical framework, and users of the tool can specify their favorite logic without having to understand the implementation details behind Hets. In this case, Hets invokes MMT to compile $SFOL^{Syn}$ into Haskell code, which is then compiled as a part of Hets resulting in a new Hets logic.

Secondly, the specification-level work flow occurs when the logic is used to write a new specification such as in `Magma` in Section 4. In this case, Hets invokes MMT for parsing and static analysis of the specification `Magma`. The result is exported by MMT in terms of the logic-independent `ParseDecl` and `Tree` data types, which are then read by Hets into the logic-specific data types produced by the compilation during the first workflow.

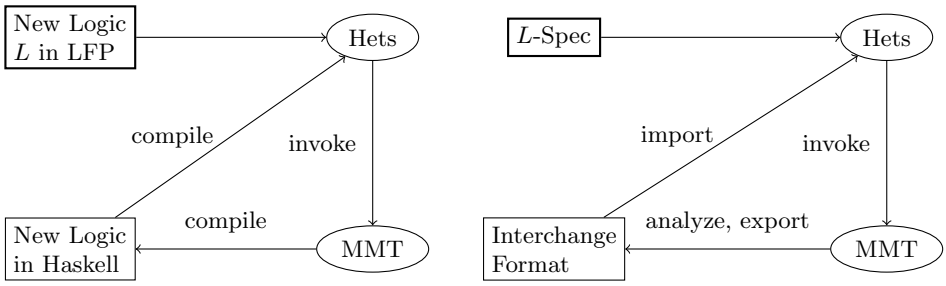

Fig. 3. Logic-Level (left) and Specification-Level (right) Workflows

6 Conclusions and Future Work

We have presented a milestone on the way of making the specification of new logics in Hets fully declarative, while maintaining the good and diverse tool

support that logics in Hets enjoy. Declaration patterns allow the specification of signatures and syntax of logics in a declarative way that simultaneously further automatizes the integration of logics in Hets.

Future work will strengthen the implementation of the approach, in particular improving the customizability of the notations for concrete syntax of expressions (`Tree`) and declarations (`ParseDecl`). Moreover, we want to integrate more features of logics into the compilation workflow. For example, it is relatively easy to extend the workflow to proof terms so that Hets can read and verify user-written proofs. It is also interesting but more difficult to generate data types that represent models, model reduction, and model morphisms. Then, colimits, amalgamability checks, and institution comorphisms can in principle be integrated into both the declarative specification in LFP and the compilation to to Hets, but finding concise declarative representations with good properties will be a challenge. Last but not least, we aim at the declarative integration of proof tools into Hets. At the language level, such an integration can already be achieved now, either via adding the prover's logic declaratively to Hets, or by looking for a logic already integrated to Hets that is close to the prover's logic, and provide a syntax translation to adapt this to the prover's input language(s). At the level of interaction with the prover, one could think of grammars for interpreting the prover's output in terms of a status ontology, and more complex reactive specifications of interfaces of interactive provers.

References

1. 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.
2. S. Buswell, O. Caprotti, D. Carlisle, M. Dewar, M. Gaetano, and M. Kohlhase. The Open Math Standard, Version 2.0. Technical report, The Open Math Society, 2004. See http://www.openmath.org/standard/om20.
3. María Victoria Cengarle, Alexander Knapp, Andrzej Tarlecki, and Martin Wirsing. A Heterogeneous Approach to UML Semantics. In Pierpaolo Degano, Rocco De Nicola, and José Meseguer, editors, *Concurrency, Graphs and Models, Essays Dedicated to Ugo Montanari on the Occasion of His 65th Birthday*, volume 5065 of *Lect. Notes Comp. Sci.*, pages 383–402. Springer, Berlin, 2008.
4. M. Codescu, F. Horozal, M. Kohlhase, T. Mossakowski, and F. Rabe. Project Abstract: Logic Atlas and Integrator (LATIN). In J. Davenport, W. Farmer, F. Rabe, and J. Urban, editors, *Intelligent Computer Mathematics*, pages 289–291. Springer, 2011.
5. M. Codescu, F. Horozal, M. Kohlhase, T. Mossakowski, F. Rabe, and K. Sojakova. Towards Logical Frameworks in the Heterogeneous Tool Set Hets. In T. Mossakowski and H. Kreowski, editors, *Recent Trends in Algebraic Development Techniques 2010*, pages 139–159. Springer, 2012.
6. 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.
7. J. Goguen and G. Rosu. Institution morphisms. *Formal Aspects of Computing*, 13:274–307, 2002.

8. Joseph Goguen and Will Tracz. An implementation-oriented semantics for module composition, 1997.
9. R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, 1993.
10. R. Harper, D. Sannella, and A. Tarlecki. Structured presentations and logic representations. *Annals of Pure and Applied Logic*, 67:113–160, 1994.
11. F. Horozal. Logic translations with declaration patterns. https://svn.kwarc.info/repos/fhorozal/pubs/patterns.pdf, 2012.
12. Simon P. Jones, Mark Jones, and Erik Meijer. Type classes: exploring the design space. In *Proceedings of the ACM Haskell Workshop*, 1997.
13. 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.
14. 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.
15. L. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer, 1994.
16. F. Rabe. The MMT System, 2008. see https://trac.kwarc.info/MMT/.
17. F. Rabe. A Logical Framework Combining Model and Proof Theory. *Mathematical Structures in Computer Science*, 2012. to appear; see http://kwarc.info/frabe/Research/rabe_combining_10.pdf.
18. F. Rabe and M. Kohlhase. A Scalable Module System. see http://arxiv.org/abs/1105.0548, 2011.
19. F. Rabe and M. Kohlhase. A Web-Scalable Module System for Mathematical Theories. Under review, see http://kwarc.info/frabe/Research/mmt.pdf, 2011.
20. 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.

Acknowledgement. This work was funded by the German Federal Ministry of Education and Research under grant 01 IW 10002 (project SHIP) and by the Deutsche Forschungsgemeinschaft (DFG) within grants KO 2428/9-1 and MO 971/2-1 (project LATIN).