# A Practical Module System for LF *

Florian Rabe
Jacobs University Bremen
f.rabe@jacobs-university.de

Carsten Schürmann
IT University of Copenhagen
carsten@itu.dk

## Abstract

Module systems for proof assistants provide administrative support for large developments when mechanizing the meta-theory of programming languages and logics. We describe a module system for the logical framework LF that is based on two main primitives: signatures and signature morphisms. Signatures are defined as collections of constant declarations, and signature morphisms as homomorphism in between them. Our design is semantically transparent in the sense that it is always possible to elaborate modules into the module free version of LF. We have implemented our design as part of the Twelf system and rewritten parts of the Twelf example library to take advantage of the module system.

***Categories and Subject Descriptors*** F.3.1 [*Specifying and Verifying and Reasoning about Programs*]: Mechanical verification; D.3.3 [*Language Constructs and Features*]: Modules, packages

***General Terms*** Proof Assistants, Logical Frameworks

***Keywords*** Twelf, modules, structures, views, Kolmogorov translation

## 1. Introduction

The Twelf system [Pfenning and Schürmann 1999] is a popular tool for reasoning about the design and properties of modern programming languages and logics. It has been used, for example, to verify the soundness of typed assembly language [Crary 2003] and Standard ML [Lee et al. 2007], for checking cut-elimination proofs for intuitionistic and classical logic [Pfenning 1995], and for specifying and validating logic morphisms, for example, between HOL and Nuprl [Schürmann and Stehr 2006]. Twelf, however, supports only monolithic proof developments and does not offer any support for modular proof engineering, composing logic morphisms, code reuse, or name space management. In this paper we develop a simple yet powerful module system for pure type systems in general, and therefore for the logical framework LF [Harper et al. 1993] in particular.

If one subscribes to the judgment-as-types methodology (as we do in the Twelf community), the equational theory underlying a logical framework determines the application areas the framework performs well in. Twelf, for example, excels in the areas of programming languages and logics, where variable binding and substitution application are prevalent. It derives its strength from dependent types, higher-order abstract syntax, and the inductive definition of canonical forms justifies its use as a proof assistant.

Retrofitting a logical framework with a module system is a delicate undertaking. On the one hand, the module system should be as powerful as possible, convenient to use, and support brief, precise, and reusable program code. On the other hand, it must not break any of the features of the logical framework or of its reasoning and programming environments. Therefore, our design is conservative over LF. We guarantee that any development in LF with modules can be elaborated into LF without modules (to which we also refer as *core LF*).

The module system that we describe in this paper is deceptively simple. It introduces two new concepts, namely that of a signature and a signature morphism. A signature is simply a collection of constant declarations and constant definitions. Signature morphisms map terms valid in the source signature into terms valid in the target signature by replacing object-level and type-level constants with objects and types, respectively. This leads to the notion of signature graphs, which have proved to be simple, flexible, and scalable abstractions to express interrelations between signatures [Sannella and Tarlecki 1988, CoFI (The Common Framework Initiative) 2004, Autexier et al. 1999].

In our current design, we do not consider questions with regards to if and how signature morphisms preserve meta-theoretic properties such as, termination, totality, or coverage; these may have been established for type families in one signature but may or may not be preserved under signature morphisms. We realize the importance of this research problem and defer it to future work. In practice, however, not knowing the answer to this question is not a severe restriction. After elaboration, the set of tools and algorithms that are already part of the Twelf system, such as mode, termination, coverage analysis, etc. continue to work and can be applied to analyze the elaborated Twelf code.

We have implemented our design as part of the Twelf distribution, see `http://www.twelf.org/mod/` for details. We demonstrate in this paper that the module system allows for compact and elegant formalizations of logic morphisms when defining for example the Kolmogorov translation from classical into intuitionistic propositional logic in a modular manner. Other examples are available from the project homepage, including a modular and type-directed development of the meta theory of Mini-ML a modular definition of the algebraic hierarchy, and a soundness proof for first-order logic.

This paper is organized as follows. We briefly describe the relevant background of the logical framework LF and our running example in Section 2. In Section 3, we give a formal definition of the module system and its semantics, not only for LF but for pure type systems in general. In Section 4, we then provide experimental

* The second author was in part supported by grant CCR-0325808 of the National Science Foundation and NABIT grant 2106-07-0019 of the Danish Strategic Research Council.



*LFMTP '09,* August 2, 2009, Montreal, Canada.

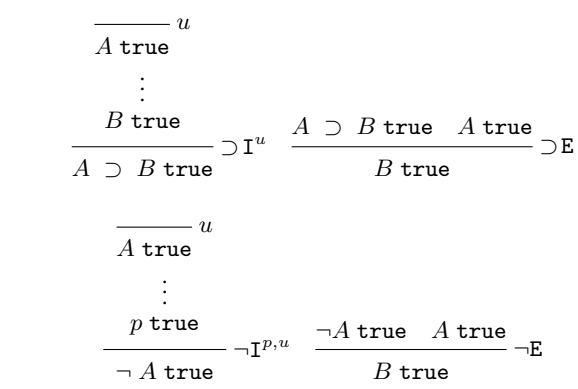


**Figure 1.** Intuitionistic Logic

evidence that the Twelf module system does not degrade runtime performance by comparing the running times for Twelf vs. modular Twelf on large (non-modular) examples. Finally, we assess results and discuss future work in Section 5.

## 2. Preliminaries

### 2.1 The Logical Framework LF

The Twelf system is an implementation of the logical framework LF [Harper et al. 1993] designed as a meta-language for the representation of deductive systems, which we also call *core LF*. Judgments are represented as types, and derivations as objects:

| | | |
|---|---|---|
| Kinds: | $K$ | $::= \texttt{type} \mid \{\texttt{x:}A\}\ K \mid A\ \texttt{->}\ K$ |
| Types: | $A, B$ | $::= a \mid A\ M \mid \{\texttt{x:}A\}\ B \mid A\ \texttt{->}\ B$ |
| Objects: | $M$ | $::= c \mid \texttt{x} \mid \texttt{[x:}A\texttt{]}\ M \mid M_1\ M_2$ |

where we write $\{\cdot\}$ for the $\Pi$-type constructor and $\texttt{[}\cdot\texttt{]}$ for a $\lambda$-binder. We omit type labels whenever they are inferable. Core LF permits declarations of type- or object-level constants. Constant symbols may be declared ("$a$ : $K$." or "$c$ : $A$.") or defined ("$a$ : $K$ = $A$." or "$c$ : $A$ = $M$."). They may be used infix, for example by issuing "`%infix` $n$ $m$ $c$." where $n$ defines if $c$ is `left`- or `right`-associative, and $m$ the binding precedence of $c$. The Twelf system offers a variety of algorithms for checking the meta-theory of signatures, including termination, coverage, and totality, which we do not discuss further in this paper, but which remain available in modular Twelf.

### 2.2 The Kolmogorov Translation

As a running example, we have chosen to use the Kolmogorov translation that embeds classical logic into intuitionistic logic. We illustrate the characteristic features of the module system by defining the relevant logics through implication $\supset$ and negation $\neg$. We say that $A$ is true, if $A$ `true` can be derived using the rules depicted in Figure 1. By adding an axiom $\neg\neg A \supset A$ `true` (the law of double negation elimination), we obtain classical logic. The Kolmogorov translation uses double-negations to map formulas $A$ to $\overline{A}$ satisfying that $A$ `true` is derivable in classical logic iff $\overline{A}$ `true` is derivable in intuitionistic logic. For example, we have $\overline{p \supset q} = \neg\neg(\neg\neg p \supset \neg\neg q)$ for propositional variables $p, q$.

## 3. The Module System

In the past, various module systems for proof assistants have been proposed, for example, for IMPS [Farmer et al. 1993], Agda [Norell 2007], Coq [Chrzaszcz 2003], Isabelle [Kammüller et al. 1999, Haftmann and Wenzel 2007], and even LF [Harper and Pfenning 1998, Licata et al. 2006] itself.

In general, a module is an encapsulated context often with some free parameters. Depending on the module system, the modules may be called *signatures*, *theories*, or *functors*. Different from the module systems above, we use two kinds of signature morphisms to relate our modules. First, the instantiation of a parametrized module $M$ induces a morphism from $M$ into the context in which the instantiation occurs. This morphism is called a *structure* in SML (while the SML modules are called functors) and has a name; all module system designs for LF including ours follow this design choice. IMPS, Agda, and Isabelle, on the other hand, permit parametrized modules but do not use the concept of structures. Second, *views* are explicit morphisms used to translate between modules (sometimes called fitting morphisms). Views were pioneered in IMPS but are not used in any of the other systems. Our use of both structures and views is more related to module systems used in algebraic specification, e.g., in [Sannella and Tarlecki 1988] and [Autexier et al. 1999] where, however, structures are not named.

Furthermore, our system insists on elaborating the modular language into the core language, which is not possible for the Coq module system and not intended for LF/[Harper and Pfenning 1998]. Agda, Isabelle, and LF/[Licata et al. 2006] employ such an elaboration and use it to define the semantics of all modular constructs. In addition to the latter, our modular constructs have an elaboration-independent semantics, against which the elaboration is proved correct. This elaboration-independent semantics is defined in terms of morphisms and permits to reason about the adequacy of encodings without having to refer to the technicalities of the elaboration.

A comprehensive overview over the state of the art in module systems and their relations to ours is given in [Rabe 2008].

### 3.1 Syntax

Our modules are called **signatures**. A signature $R, S, T$ is a named list of constant and structure declarations. Two signatures are equal iff they have the same name, which means that two signatures that contain the same declarations are not necessarily considered equal. This avoids complex equality reasoning about signatures, and is not a loss in expressivity because views, which we introduce in Section 3.1.2, can be used to establish isomorphisms between two such signatures.

EXAMPLE 1 (Judgments). The signature with name `JUDGMENTS` given below defines the judgments from Section 2.2.

```
%sig JUDGMENTS = {
  o    : type.
  true : o -> type.
}.
```

It is easier to explain the module system for LF as an instance of the more general case: pure type systems [Barendregt 1991]. Therefore, we collapse the three syntactic categories of kinds, types, and objects into one single category of **terms** $C$:

$$C ::= \vec{c} \mid x \mid \texttt{type} \mid \{\texttt{x:}C\}\ C \mid \texttt{[x:}C\texttt{]}\ C \mid C\ C$$

The only difference to the core syntax is the case of references to constants, which are now qualified names $\vec{c}$. Qualified names access constants that are part of structures and are explained below.

Signature morphisms define mappings between signatures. A morphism $\mu$ from a signature $S$ to $T$ maps every constant $\vec{c}$ of $S$ with type (kind) $A$ to a term $C$ over $T$ such that $C$ is typed (kinded) by $\mu(A)$. Here $\mu(-)$ is the homomorphic extension of $\mu$ to terms. The mapping $\mu(-)$ preserves the typing relation and the

definitional equality of $S$, e.g., if $B : A$ in $S$, then $\mu(B) : \mu(A)$ in $T$ (see, e.g., [Harper et al. 1994]).

In the following, we explain the two kinds of signature morphisms in detail: **structures** from $S$ to $T$ create an instance of $S$ in $T$, and **views** from $S$ to $T$ define a translation from $S$ to $T$.

### 3.1.1 Structures

In the simplest case, a structure declaration $s : S$ occurring within a signature $T$ has the following semantics: If $S$ declares a constant $c$ then the structure declaration induces a constant $s.c$ in $T$ by copying $c$. Because the structure declarations of $S$ are also copied into $T$, the general way to refer to constants is via *qualified constant identifiers* $\vec{c} ::= s.\ \ldots\ .s.c$. Similarly, we use *qualified structure identifiers* $\vec{s} ::= s.\ \ldots\ .s.s$.

An important advantage of named structures and qualified identifiers is that $T$ may contain multiple distinct instances of $S$. However, qualified identifiers can introduce a lot of clutter. In order to tame this clutter we introduce convenient **aliases** through the `%open` directive that permits the use of the constant name $c$ instead of the qualified name $s.c$. Therefore, contrary to its analogue in SML, `%open` is syntactic sugar that provides a mechanism for introducing aliases that always resolve to the same internal identifier. Dealiasing is merely an implementation detail and therefore not discussed here.

EXAMPLE 2 (Implication). $\supset$ and its introduction and elimination rules from Figure 1 are encoded as follows:

```
%sig IMP = {
  %struct J : JUDGMENTS %open o true.
  ⊃  : o -> o -> o. %infix left 10 ⊃.
  ⊃I : (true A -> true B) -> true (A ⊃ B).
  ⊃E : true (A ⊃ B) -> true A -> true B.
}.
```

A structure $s : S$ occurring in $T$ induces a signature morphism from $S$ to $T$ as follows: Every constant $\vec{c}$ of $S$ is mapped to the constant $s.\vec{c}$ of $T$. For example, `J` declared in `IMP` is a morphism from `JUDGMENTS` to `IMP`. If signatures are seen as records, then this is simply a record projection.

In the example above, we import `o` and `true` from `JUDGMENTS` using a structure `J` as abbreviations for the constants `IMP"J.o` and `IMP"J.true`. Within `IMP`, we refer to these constants as `J.o` and `J.true`.

EXAMPLE 3 (Negation). Similarly, we encode negation and its rules:

```
%sig NEG = {
  %struct J : JUDGMENTS %open o true.
  ¬    : o -> o.
  ¬I   : ({p} true A -> true p) -> true (¬ A).
  ¬E   : true (¬ A) -> true A -> true B.
  n    = [p] (¬ (¬ p)).
  ¬¬I  : true A -> true (n A)
       = [D] (¬I [p:o] [u: true (¬ A)] (¬E u D)).
}.
```

Negation satisfies the double negation introduction rule "If $A$ `true` then $\neg\neg A$ `true`.". The proof is direct, and it defines the derived rule of inference `¬¬I`.

In addition to copying declarations from $S$ to $T$, structures can instantiate constants and structures declared in $S$ with corresponding expressions over $T$. We call such pairs of $S$-symbol and $T$-expression **assignments**.

EXAMPLE 4 (Intuitionistic Logic). To obtain an encoding of intuitionistic logic as in Figure 1, we combine `IMP` and `NEG`. The common structure `J` must be shared:

```
%sig IL = {
  %struct I : IMP %open o true ⊃ ⊃I ⊃E.
  %struct N : NEG  = { %struct J := I.J.}
                     %open ¬ n ¬I ¬E ¬¬I.
}.
```

Here `%struct J := I.J.` is an assignment: `J` refers to the structure declared in `NEG`, which is copied into `IL` resulting in the structure `N.J`; assigning `I.J` to it, yields the desired sharing relation `N.J` = `I.J`. The assignment is well-typed because both `N.J` and `I.J` are instances of the same signature, namely `JUDGMENTS`.

In `%struct N : NEG = { %struct J := I.J.}`, readers familiar with SML may think of `NEG` as a functor, and of `{ %struct J := I.J.}` as its argument.

EXAMPLE 5 (Classical Logic). Finally, by extending intuitionistic logic with the axiom of double negation elimination, we obtain the definition of classical logic.

```
%sig CL = {
  %struct IL : IL %open true ⊃ ¬.
  dne : true (¬ (¬ A) ⊃ A).
}.
```

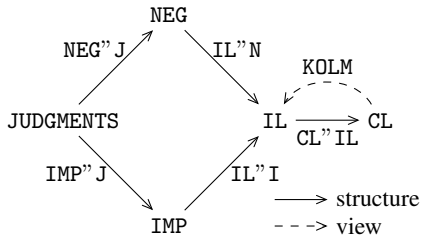
Formally, we define the body of a signature by

$$\Sigma ::= \cdot \mid \Sigma,\ D_c \mid \Sigma,\ D_s.$$

Here $D_c$ stands for a constant declaration $D_c ::= c : C \mid c : C = C$ and $D_s ::= \vec{s} : T = \{\sigma\}$ stands for a structure declaration where

$$\sigma ::= \cdot \mid \sigma,\ \vec{c} := C \mid \sigma,\ \vec{s} := \mu$$

gives a list of assignments, where $\vec{c}$ and $\vec{s}$ are qualified constant and structure names, respectively. For the sake of convenience, we omit the keywords `%sig`, `%struct`, `%view` from the formal presentation.




A **signature graph** is a multi-graph with signatures as nodes and structures or views as edges. The signatures and structures introduced in the running example so far form the signature graph on the right. There, we use the **notation** $S"s$ to refer to the name $s$ declared within the signature $S$. When talking about modular Twelf, we will occasionally use this notation to make references to constants and structures unambiguous.

### 3.1.2 Views

Next, we turn to views and define the view `KOLM` that interprets classical proofs over `CL` as intuitionistic proofs over `IL`. It is composed modularly from four different views into `IL`.

EXAMPLE 6 (Kolmogorov view). We begin by translating the judgments in the view `KOLMJ` from `JUDGMENTS` to `IL`: The assignment `o := o` expresses that formulas are mapped to formulas, and the assignment `true := [x] true (n x)` expresses that the

judgment $A$ `true` is mapped to the judgment $\neg\neg B$ `true` where $B$ is the translation of $A$. As for structures, the left hand side of an assignment is a symbol of the domain, and the right hand side is an expression over the codomain.

Similarly, we define the views `KOLMI` and `KOLMN`, which translate implication and negation, respectively. The proof rules are translated to derived rules of inference, which are easily determined by pen and paper. These views are total: There is an assignment for every constant of the domain with the only exception of those that are defined.

Note that the assignments must abstract over the implicit arguments as well. For example, the assignment to ⊃I must abstract over the implicit arguments `A` and `B` that occur in the (omitted) type of `D`.

```
%view KOLMJ : JUDGMENTS -> IL = {
  o    := o.
  true := [x] true (n x).
}.
```

```
%view KOLMI : IMP -> IL = {
  %struct J := KOLMJ.
  ⊃  := [x][y] ((n x) ⊃ (n y)).
  ⊃I := [A][B][D] ¬¬I (⊃I D).
  ⊃E := [A][B][D][E] ¬I [p][u]
        ¬E D (¬I [q][v]
            ¬E (⊃E v E) u).
}.
```

```
%view KOLMN : NEG -> IL = {
  %struct J := KOLMJ.
  ¬  := [x] ¬ x.
  ¬I := [A][D] ¬I [q][u]
        ¬E (D (¬ A) u) u.
  ¬E := [A][C][D][E] ¬I [p][u]
        ¬E D E .
}.
```

```
%view KOLM : CL -> IL = {
  %struct IL.I := KOLMI.
  %struct IL.N := KOLMN.
  dne := [A] ¬I [p] [u] ¬E u
         (⊃I [u] ¬I [p] [v] ¬E u
           (¬I [q][w] ¬E w v)).
}.
```

In summary, the view `KOLM` is the Kolmogorov translation mapping the embedding from `CL` into `IL`. It illustrates nicely the expressive strength of what we call *deep* assignments: Instead of providing an assignment for the structure `IL` of `CL`, it assigns morphisms to the structures `IL.I` and `IL.N`. Intuitively, the assignment `%struct IL.I := KOLMI.` is justified as follows: `IL.I` is a copy of `IMP` into the domain `CL` of `KOLM`; thus, it is mapped to the view `KOLMI`, which is a translation of `IMP` into the codomain `IL` of `KOLM`. The last assignment in `KOLM` is the translation of the law of double negation elimination.

Thus, `KOLM` implements a meta-theoretic proof that classical proofs can be translated to intuitionistic ones: It covers all cases because it substitutes terms for all constants of `CL`. We do not provide any cases for variables, because inputs are always closed (do not contain free variables), and bound variables map to themselves. The application of `KOLM` is also clearly terminating. □

Formally, we write $D_v ::= v : S \to T = \{\sigma\}$ for a view $v$ from $S$ to $T$ where $\sigma$ is as for structures except that it must provide assignments for *all* constants of $S$ (except for those that have definitions).

Finally, given a set of signature and view declarations, we define signature morphisms by $\mu ::= T"\vec{s} \mid v \mid \mu\ \mu$. Here $T"\vec{s}$ refers to the structure $\vec{s}$ of the signature $T$, $v$ refers to a view, and $\mu\ \mu'$ represents the composition of two morphisms in diagrammatic order. In the running example, `IL"N CL"IL KOLM` is a morphism from `NEG` to `IL`.

Then we can define qualified structure identifiers precisely: The structure `CL"IL.I.J` is defined to be equal to the morphism `IMP"J IL"I CL"IL`.

Readers familiar with SML may interpret a view from $S$ to $T$ in two different ways. The view may be seen as a functor that is parametrized by a structure of type $T$ (a morphism with domain $T$) and produces a structure of type $S$. Alternatively, it may be seen as a structure implementing the signature $S$ defined by explicitly providing values for all symbols of $S$ – the role of $T$ here is to represent the context in which $S$ is implemented. Since LF is a logical framework and not a programming language, the implementation language must itself be represented as a signature, namely $T$. Thus, morphisms are paths in the signature graph.

This concludes the definition of the syntactic categories of our module system for LF, which we summarize in Figure 2.

### 3.2 Another Example

As a further example, we illustrate how signatures, structures, and views introduce module level parametricity. The `List` signature below is parametrized over a `Monoid` signature of elements and provide a relation that computes (parametrically) the fold of that list.

```
%sig Monoid = {
  a    : type.
  unit : a.
  comp : a -> a -> a -> type.
}.
```

```
%sig List = {
  %struct elem : Monoid.
  list     : type.
  nil      : list.
  cons     : elem.a -> list -> list.
  fold     : list -> elem.a -> type.
  foldnil  : fold nil elem.unit.
  foldcons : fold L B -> elem.comp A B C
             -> fold (cons A L) C.
}.
```

The signature `Monoid` declares a monoid as a type together with the usual unit element and the binary composition operation (written as a relation), where we omit the axioms for simplicity. The signature `List` defines lists, which is parametric in type `elem.a`, and the fold relation, which is parametric in `elem.unit` and `elem.comp`.

Furthermore, we define a `Monoid` called `NatMonoid` that is based on natural numbers `Nat`.

```
%sig Nat = {
  nat     : type.
  zero    : nat.
  succ    : nat -> nat.
  add     : nat -> nat -> nat -> type.
  addzero : add N zero N.
  addsucc : add N P Q -> add N (succ P) (succ Q).
}.
```

```
%view NatMonoid : Monoid -> Nat = {
  a    := nat.
  unit := zero.
```

| | | | |
|---|---|---|---|
| Signature graph | $\mathcal{G}$ | ::= | $\cdot \mid \mathcal{G},\ D_T \mid \mathcal{G},\ D_v$ |
| Signature | $D_T$ | ::= | $T = \{\Sigma\}$ |
| View | $D_v$ | ::= | $v : T \to T = \{\sigma\}$ |
| Signature body | $\Sigma$ | ::= | $\cdot \mid \Sigma,\ D_c \mid \Sigma,\ D_s$ |
| Constant | $D_c$ | ::= | $c : C \mid c : C = C$ |
| Structure | $D_s$ | ::= | $s : T = \{\sigma\}$ |
| Assignment list | $\sigma$ | ::= | $\cdot \mid \sigma,\ \vec{c} := C \mid \sigma,\ \vec{s} := \mu$ |
| Term | $C$ | ::= | $\vec{c} \mid \mathit{type} \mid \{x : C\}C \mid [x : C]C \mid C\ C$ |
| Morphism | $\mu$ | ::= | $\vec{s} \mid v \mid \mu\ \mu$ |
| Qualified identifiers | $\vec{c}$ | ::= | $s.\ \ldots\ .s.c$ |
| | $\vec{s}$ | ::= | $s.\ \ldots\ .s.s$ |
| Identifiers | $T, v, c, s, x$ | | |

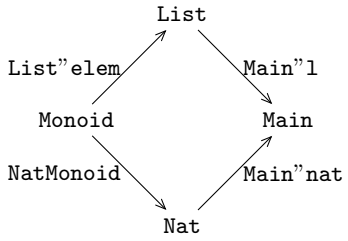
**Figure 2.** The Grammar for Expressions

```
  comp := add.
}.
```

The view `NatMonoid` allows us to view `Nat` as a `Monoid`. Therefore, if we put all the pieces together, instantiate the parametric structure `elem`, we may now compute folds over natural numbers using Twelfs `%solve` directive.

```
%sig Main = {
  %struct nat  : Nat.
  %struct l : List = {
    %struct elem := NatMonoid nat.
  }.
  %solve _ : fold
    (l.cons nat.zero (l.cons nat.zero l.nil)) N.
}.
```

Here the composite `NatMonoid nat` is the result of translating the structure `nat` along `NatMonoid` in order to fit it to the expected type of `elem`. The signature graph looks as below, and Theorem 9 below guarantees that it commutes.




The morphism denoted by `NatMonoid nat` is the result of applying `NatMonoid` to `nat`. It may be confusing that this application takes a structure of type `Nat` as its argument and returns a structure of type `Monoid`, although the view `NatMonoid` is typed as `Monoid -> Nat`. This is an effect of an adjunction between syntax and semantics: a view induces a translation of structures in the opposite direction. Here the translation of structures is seen as a semantical translation because every structure of type $S$ is an implementation or a model of $S$.

### 3.3 Elaboration

In this section, we define the *elaboration semantics* of the module system into core LF. The target language of elaboration is that of Fig. 2 but without structure declarations and without assignments to structures. The elaboration of the remaining views to plain LF is straightforward.

The elaboration semantics provides implicitly an algorithm on how to *lookup* the type of a constant, for example, during type checking. While this is trivial in the non-modular case, in a modular setting constants may be induced by structure declarations or appear in assignments or views.

In the remainder of this section, we define the judgments for elaborating declarations in a signature $T$ and for morphisms $m$ in a signature graph $\mathcal{G}$, respectively: $\mathcal{G} \ggg_T \vec{c} : A = B$ and $\mathcal{G} \ggg_m \vec{c} := B$. The former expresses that the declaration $\vec{c} : A = B$ is present in the signature $T$ after elaboration (or: that the lookup of the constant $\vec{c}$ in the signature $T$ returns the type $A$ and the definition $B$). In the interest of brevity, we reuse the same judgment for defined and undefined constants, by writing $B = \bot$ in the latter case. Very similarly, $\mathcal{G} \ggg_m \vec{c} := B$ expresses that the assignment $\vec{c} := B$ is present in the view or structure $m$ after elaboration (or: that the lookup of the assignment to $\vec{c}$ in $m$ returns $B$). Again we use $\bot$ for brevity: If the domain of $m$ has a constant $\vec{c}$ but $m$ provides no assignment for it, we write $\mathcal{G} \ggg_m \vec{c} := \bot$.

The intuitive definition of these judgments is as follows. Assume a structure declaration $r : R = \{\sigma\}$ occurs in $S$, and assume $R$ contains $c : A$, and $\sigma$ contains no assignment for $c$. We have $\mathcal{G} \ggg_S s.c : S\text{"}s(A) = \bot$. Similarly, if $\sigma$ contains the assignment $c := B$, then we have $\mathcal{G} \ggg_S s.c : S\text{"}s(A) = B$. In both cases $S\text{"}s(A)$ is the result of translating $A$ along the morphism $S\text{"}s$, prefixing all constants with $s$.

Now assume in addition a morphism $\mu$ from $R$ to $T$ and an assignment $s := \mu$ in a view or structure $m$ from $S$ to $T$. We have $\mathcal{G} \ggg_m s.c := \mu(c)$, i.e., $m$ maps the constant $s.c$ of $S$ to the result of applying $\mu$ to the constant $c$ of $R$.

***Views and Structures*** In order to define the above judgments, we need one auxiliary judgment that is defined by the three rules in Fig. 3. $\mathcal{G} \ggg m : S \to T = \{\sigma\}$ expresses that $m$ is a morphism from $S$ to $T$ defined by the list $\sigma$ of assignments. $m$ may be a view (first rule) or a structure (second rule). Finally, a structure $s$ from $S$ to $T$ does not only induce constants $s.\vec{c}$ but also structures $s.\vec{r}$. The meaning of the structure $T\text{"}s.r$ is defined to be the composition $S\text{"}r\ T\text{"}s$. The judgment $\mathcal{G} \ggg m : S \to T = \mu$ expresses that $m$ is a morphism from $S$ to $T$ with definition $\mu$ (third rule).

$$\frac{v : S \to T = \{\sigma\} \text{ in } \mathcal{G}}{\mathcal{G} \ggg v : S \to T = \{\sigma\}} \qquad \frac{T = \{\ldots,\ s : S = \{\sigma\},\ \ldots\} \text{ in } \mathcal{G}}{\mathcal{G} \ggg T\text{"}s : S \to T = \{\sigma\}}$$

$$\frac{\mathcal{G} \ggg T\text{"}s : S \to T = \_ \quad \mathcal{G} \ggg S\text{"}\vec{r} : R \to S = \_}{\mathcal{G} \ggg T\text{"}s.\vec{r} : R \to T = S\text{"}\vec{r}\ T\text{"}s}$$

**Figure 3.** Views and Structures

***Semantics of Structures*** We are now in the position to define the two main judgments. Fig. 4 gives the four rules defining the elaboration of structure occurring in a signature $T$ (or: the lookup in a signature $T$). The first two rules handle constants with and without definition declared in $T$. The other two rules handle constants $s.\vec{c}$ induced by a structure $s$ with domain $S$. If $s$ provides an assignment $B'$ for $\vec{c}$ it is used as the definition of $s.\vec{c}$ (third rule). Otherwise, the existing definition $B$ of $c$ is translated along $T$"$s$. The translation along morphisms is defined formally below. In the interest of brevity, we put $\mu(\bot) = \bot$, i.e., neither $\vec{c}$ has definition in $S$ nor $s$ provides an assignment for it, then $s.\vec{c}$ has no definition either.

$$\frac{T = \{\ldots,\ c : A = B,\ \ldots\} \text{ in } \mathcal{G}}{\mathcal{G} \ggg_T c : A = B} \qquad \frac{T = \{\ldots,\ c : A,\ \ldots\} \text{ in } \mathcal{G}}{\mathcal{G} \ggg_T c : A = \bot}$$

$$\frac{\mathcal{G} \ggg T\text{"}s : S \to T = \_ \quad \mathcal{G} \ggg_S \vec{c} : A = B \quad \mathcal{G} \ggg_{T\text{"}s} \vec{c} := B' \quad B' \neq \bot}{\mathcal{G} \ggg_T s.\vec{c} : T\text{"}s(A) = B'}$$

$$\frac{\mathcal{G} \ggg T\text{"}s : S \to T = \_ \quad \mathcal{G} \ggg_S \vec{c} : A = B \quad \mathcal{G} \ggg_{T\text{"}s} \vec{c} := \bot}{\mathcal{G} \ggg_T s.\vec{c} : T\text{"}s(A) = T\text{"}s(B)}$$

**Figure 4.** Semantics of Structures

***Semantics of Assignments to Structures*** Fig. 5 gives the three rules defining the elaboration of assignments to structures occurring in a morphism $m$. The first rule handles the case where $m$ is defined by a morphism $\mu$, which is simply applied to all constants $\vec{c}$ of $S$. If $m$ is defined a list of assignments, two cases are possible. If $m$ has as assignment for $\vec{c}$, the treatment is clear (second rule). But if $m$ contain an assignment to a structure $\vec{n}$, it induces assignments to all constants $\vec{s}.\vec{c}$ (third rule).

$$\frac{\mathcal{G} \ggg m : S \to T = \mu}{\mathcal{G} \ggg_m \vec{c} := \mu(\vec{c})} \qquad \frac{\mathcal{G} \ggg m : S \to T = \{\ldots,\ \vec{c} := C,\ \ldots\}}{\mathcal{G} \ggg_m \vec{c} := C}$$

$$\frac{\mathcal{G} \ggg m : S \to T = \{\ldots,\ \vec{s} := \mu,\ \ldots\} \quad \mathcal{G} \ggg S\text{"}\vec{s} : R \to S = \_}{\mathcal{G} \ggg_m \vec{s}.\vec{c} := \mu(\vec{c})}$$

**Figure 5.** Semantics of Structure Assignments

***Morphism Application*** Finally, we define the application of a morphism $\mu$ to a term $C$ by induction on $\mu$ and $C$. We define the step cases by

$$\begin{array}{ll} \mu\ \mu'(\vec{c}) & := \mu'(\mu(\vec{c})) \\ \mu(type) & := type \\ \mu(x) & := x \\ \mu([x : A]\ C) & := [x : \mu(A)]\ \mu(C) \\ \mu(\{x : A\}\ C) & := \{x : \mu(A)\}\ \mu(C) \\ \mu(C\ C') & := \mu(C)\ \mu(C') \end{array}$$

The most interesting case is the base case when a view or a structure is applied to a constant: If $\mathcal{G} \ggg m : S \to T = \_$, $\mathcal{G} \ggg_S \vec{c} : \_ = B$, and $\mathcal{G} \ggg_m \vec{c} := B'$, then we define

$$m(\vec{c}) := \begin{cases} m(B) & \text{if } B \neq \bot \\ B' & \text{if } B = \bot,\ m = v \text{ view} \\ \vec{s}.\vec{c} & \text{if } B = \bot,\ m = T\text{"}\vec{s} \text{ structure} \end{cases}$$

| | |
|---|---|
| $\vdash \mathcal{G}$ | $\mathcal{G}$ is a well-formed signature graph. |
| $\mathcal{G} \rhd D$ | The declaration $D$ can be added to $\mathcal{G}$. |
| $\mathcal{G} \rhd_N D$ | The declaration or assignment $D$ can be added to the signature, view, or structure named $N$. |
| $\mathcal{G} \vdash \mu : S \to T$ | $\mu$ is a well-formed morphism from $S$ to $T$ (or: a well-formed structure over $T$ of type $S$). |
| $\mathcal{G} \vdash_T C \equiv B$ | $C$ and $B$ are equal over $\mathcal{G}$ and $T$. |
| $\mathcal{G} \vdash_T C : A$ | $C$ is a well-formed term of type $A$ over $\mathcal{G}$ and $T$. |

**Figure 6.** Main Judgments

If $\vec{c}$ is defined then it is expanded (first case). If $\vec{c}$ is declared then it is mapped according to the assignment provided by the view (second case) or mapped to the constant induced by the structure (third case). Clearly, this is only well-defined if the three judgments defined in this section are functional, i.e., there must be unique values $S$, $B$, and $B'$. This follows from the requirement that there are no name clashes in $\mathcal{G}$, i.e. that constant symbols are not declared (in the case of a signature) or assigned (in the case of a view or structure) twice, as we show in the next section.

### 3.4 Type System

In this section, we present an inference system to define the **well-formedness** of our syntactic categories. The judgments are given in Fig. 6. The judgment $\vdash \mathcal{G}$ states the well-formedness of signature graphs. The judgment $\mathcal{G} \rhd D$ expresses that $\mathcal{G}$ can be extended with the signature or view $D$, and $\mathcal{G} \rhd_N D$ expresses that the signature, view, or structure $N$ can be extended with the declaration or assignment, respectively, $D$. Finally, there are three judgments that define well-formed terms and morphisms relative to a signature graph and a signature declared in that graph.

While the conceptual core of the type system is quite simple, the technical details regarding namespace management can make the notation rather complex. Therefore, we choose a simplified presentation using a judgment $noClash(\mathcal{G}, N, n)$ that expresses that a declaration or assignment for the identifier $n$ can be added to the signature, view, or structure $N$ without creating a name clash. Similarly, $noClash(\mathcal{G}, N)$ expresses that a signature or view named $N$ can be added to $\mathcal{G}$. We refrain from formalizing these judgments; instead, we only state that the consequent use of this judgment guarantees that the judgments $\mathcal{G} \ggg m : S \to T = \_$, $\mathcal{G} \ggg_T \vec{c} : A = B$, and $\mathcal{G} \ggg_m \vec{c} := B$ are functional where the output arguments are indicated in red. Therefore, we can use these judgments to look up names, types, and definitions for the identifiers encountered during type-checking.

***Structure of Signature Graphs*** We define $Sig(\mathcal{G})$ to be the set of signature names declared in $\mathcal{G}$. The structure of signature graphs is defined by the rules in Fig. 7. These rules follow the grammar and iterate a well-formedness judgment over all components of a signature graph. They also check that views are total and that module names do not clash. The well-typedness of constant declarations and assignments (red assumptions) and objects (blue assumptions) is defined by the rules in Fig. 8.

The rule $\mathcal{G}_\emptyset$ constructs an empty signature graph. The rules $Sig$ and $View$ extend a well-formed signature graph with a well-formed signature or view; while signatures can be added directly, views must be *total*, which means that they must provide an assignment for every declared constant.

Whether or not a signature or view is well-formed is defined in the remaining rules. The rules $Sig_\emptyset$ and $Sym$ construct signatures by successively adding well-formed symbols, and the rules $View_\emptyset$ and $View_{Ass}$ construct views by successively adding well-formed assignments. The rules for structures correspond to those for views:

$$\frac{}{\vdash \cdot}\ \mathcal{G}_\emptyset \qquad \frac{\vdash \mathcal{G} \quad \mathcal{G} \rhd T = \{\Sigma\}}{\vdash \mathcal{G},\ T = \{\Sigma\}}\ Sig \qquad \frac{noClash(\mathcal{G}, T)}{\mathcal{G} \rhd T = \{\cdot\}}\ Sig_\emptyset \qquad \frac{\mathcal{G} \rhd T = \{\Sigma\} \quad \mathcal{G},\ T = \{\Sigma\} \rhd_T D}{\mathcal{G} \rhd T = \{\Sigma, D\}}\ Sym$$

$$\frac{\vdash \mathcal{G} \quad \mathcal{G} \rhd v : S \to T = \{\sigma\} \quad \mathcal{G} \ggg_v \vec{c} := B \text{ for some } B \text{ whenever } \mathcal{G} \ggg_S \vec{c} : A = \bot}{\vdash \mathcal{G},\ v : S \to T = \{\sigma\}}\ View$$

$$\frac{noClash(\mathcal{G}, v) \quad S \in Sig(\mathcal{G}) \quad T \in Sig(\mathcal{G})}{\mathcal{G} \rhd v : S \to T = \{\cdot\}}\ View_\emptyset \qquad \frac{\mathcal{G} \rhd v : S \to T = \{\sigma\} \quad \mathcal{G},\ v : S \to T = \{\sigma\} \rhd_v D}{\mathcal{G} \rhd v : S \to T = \{\sigma, D\}}\ View_{Ass}$$

$$\frac{noClash(\mathcal{G}, T, s) \quad S \in Sig(\mathcal{G}) \setminus \{T\}}{\mathcal{G} \rhd_T s : S = \{\cdot\}}\ Str_\emptyset \qquad \frac{\mathcal{G},\ T = \{\Sigma\} \rhd s : S = \{\sigma\} \quad \mathcal{G},\ T = \{\Sigma,\ s : S = \{\sigma\}\} \rhd_{T\text{"}s} D}{\mathcal{G},\ T = \{\Sigma\} \rhd_T s : S = \{\sigma, D\}}\ Str_{Ass}$$

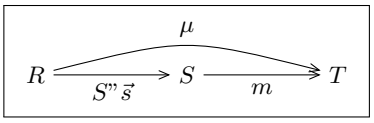
**Figure 7.** Structural Rules

$Str_\emptyset$ and $Str_{Ass}$ construct structures by successively adding well-formed assignments. The rule $Str_\emptyset$ also ensures that a signature may not instantiate itself.

***Well-formed Declarations and Assignments*** The rules above the dotted line in Fig. 8 define when constants and assignments are well-typed. The rule $Con$ says that constant declarations $c : A = B$ are well-typed for a signature $T$ if $B$ has type $A$, and if $c$ is not already declared in $T$. In order to save case distinctions, we use the following convention: We permit the case $B = \bot$ for constants without definitions, and say that the typing judgment $\mathcal{G} \vdash_T \bot : A$ holds if $A$ is a well-formed type or kind. Note that extensions to other type systems only require to modify this convention appropriately.

The rule $ConAss$ defines when an assignment $\vec{c} := B$ is well-typed. The first three premises look up the domain and codomain of the last view or structure $m$ in $\mathcal{G}$, make sure that an assignment for $\vec{c}$ does not clash with existing assignments in $m$, and look up the type of $\vec{c}$. The definition of $\vec{c}$ must be $\bot$, i.e., defined constants cannot be instantiated. The final premise type-checks $B$ against the translation of $A$. If $m(A)$ is not defined, which is possible if $m$ is a view and $A$ contains constants for which $m$ does not provide an assignment yet, we consider the typing judgment not to hold. Thus, the order of assignments in a link must respect the dependency order between the symbols declared in the domain.

The rule $StrAss$ for assignments to structures is similar to $ConAss$. The first three premises correspond to those of $ConAss$. In particular, $R$ corresponds to $A$ as the type of $\vec{s}$, and the fourth premise checks the type of $\mu$ against $R$. To understand the last premise, note that the intended semantics of assignments to structures is that the diagram on the right commutes. This is only possible if $\mu$ agrees with $S\text{"}\vec{s}\ m$ for all constants for which $m$ is already determined.




The rules below the dotted line in Fig. 8 define the typing of objects. $\mathcal{T}_:$ and $\mathcal{T}_\equiv$ replace the core LF rule for the lookup of constants in the signature (called con in [Pfenning 2001]). All other typing and equality rules of core LF are retained. To obtain module systems for other type theories, the typing and equality rules have to be changed accordingly. Finally, the rules $\mathcal{M}_m$ and $\mathcal{M}_{comp}$ construct morphisms as sequences of views and structures. Composition is written in diagrammatic order, i.e., from the domain to the codomain.

### 3.5 Meta-Theory

We turn now to the meta-theoretical results that we have shown about the module system, most notably, conservativity.

First, we define two morphisms to be equal if they agree for all constants of the domain signature.

DEFINITION 7. *Assume $\mathcal{G} \vdash \mu : S \to T$ and $\mathcal{G} \vdash \mu' : S \to T$. We define the judgment $\mathcal{G} \vdash \mu \equiv \mu'$ to hold iff for all $\vec{c}$ for which $\mathcal{G} \ggg_S \vec{c} : \_ = \_$ we have $\mathcal{G} \vdash_T \mu(\vec{c}) \equiv \mu'(\vec{c})$.*

THEOREM 8. *Assume $\mathcal{G} \vdash \mu : S \to T$. If $\mathcal{G} \vdash_S C : A$, then $\mathcal{G} \vdash_T \mu(C) : \mu(A)$. And if $\mathcal{G} \vdash_S C \equiv C'$ and $\mathcal{G} \vdash \mu \equiv \mu'$, then $\mathcal{G} \vdash_T \mu(C) \equiv \mu'(C')$.*

**Proof:** This is a special case of the results given in [Rabe 2008, Ch. 6]. The proof proceeds by induction on $\mu$ and $C$. The key steps are that the rules $ConAss$ and $StrAss$ permit to add assignments to views or structures only if they do not violate the typing or equality relations of the domain signature. □

This result establishes the basic properties of signature morphisms as encoded in our module system: the preservation of typing and equality. The following result states the intended semantics of assignments to structures, namely that the diagram given in Sect. 3.4 commutes. Together with Thm. 8, it shows that the module system can be used to reason about signature morphisms. This is the cornerstone of adequacy proofs because adequacy proofs in terms of signature morphisms are very elegant and concise. For example, we obtain the adequacy of the structure sharing in the LF encoding of intuitionistic logic in Example 4 immediately without having to elaborate any of the involved structures.

THEOREM 9. *Assume $\vdash \mathcal{G}$. If there is an assignment $\vec{s} := \mu$ in a view or structure $m$ from $S$ to $T$ in $\mathcal{G}$, then $\mathcal{G} \vdash S\text{"}\vec{s}\ \ m \equiv \mu$.*

**Proof:** This is a special case of the results given in [Rabe 2008, Ch. 6]. All definitions are targeted at this result, most importantly rule $StrAss$ rejects any assignment that would violate the theorem. □

Finally, we show that modular LF is conservative over core LF. The main argument is that elaboration is sound and that core LF signatures elaborate to themselves. The only caveat is easily explained: Qualified identifiers in modular LF need to be considered constants in core LF, which means that "?" and "." must be allowed in constant names.

THEOREM 10. *Assume a signature graph $\mathcal{G}$. Let $\Sigma$ be the core LF signature containing the declarations*

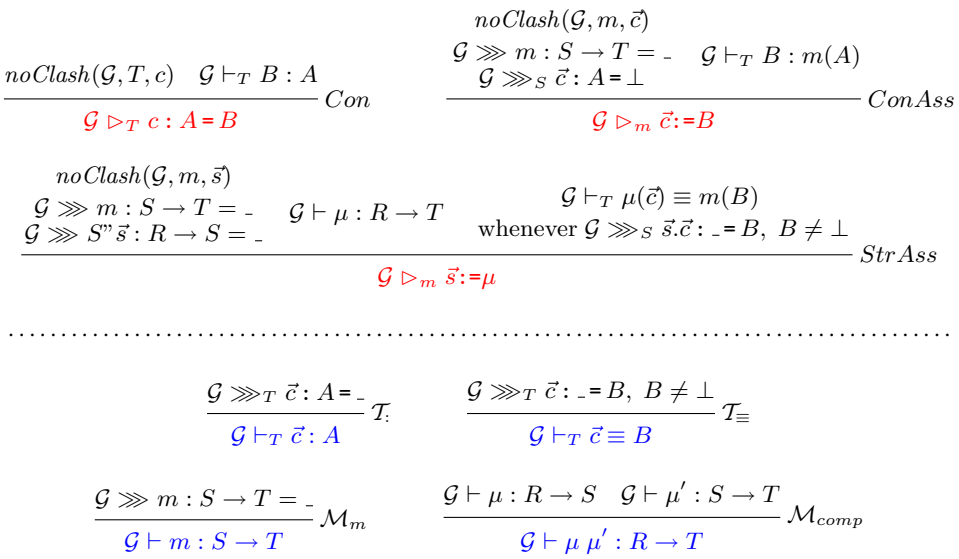


**Figure 8.** Typing Rules

- *for all signatures $T$ declared in $\mathcal{G}$: whenever $\mathcal{G} \ggg_T \vec{c} : A = B$ for $B \neq \bot$, the declaration $T$"$\vec{c} : A = B$; and whenever $\mathcal{G} \ggg_T \vec{c} : A = \bot$, the declaration $T$"$\vec{c} : A$,*
- *for all views $v$ with domain $S$ declared in $\mathcal{G}$: whenever $\mathcal{G} \ggg_S \vec{c} : A = \bot$ and $\mathcal{G} \ggg_v \vec{c} := B$, the declaration $v$"$\vec{c} : v(A) = B$,*
- *for all structures $s$ with domain $S$ declared in a signature $T$ of $\mathcal{G}$: whenever $\mathcal{G} \ggg_S \vec{c} : \_ = B$, $B \neq \bot$, and $\mathcal{G} \ggg_{T\text{"}s} \vec{c} := B'$, $B' \neq \bot$, then a declaration that is well-typed iff $B \equiv B'$,*[1]

*in some order that respects the dependencies between them. Then $\vdash \mathcal{G}$ iff $\Sigma$ is a valid core LF signature.*

**Proof:** This is a special case of the results given in [Rabe 2008, Ch. 6]. The only modification of the argument is that here $\Sigma$ is always ill-formed if any view in $\mathcal{G}$ is not total because it will contain the illegal symbol $\bot$. □

| | Cut-Elim | TALT | SML |
|---|---|---|---|
| Parsing | 0.008 (0.008) | 3.590 (2.733) | 0.583 (0.851) |
| Reconstruction | 0.017 (0.017) | 8.620 (14.00) | 1.688 (2.324) |
| Abstraction | 0.008 (0.007) | 7.154 (6.254) | 0.738 (1.004) |
| Modes | 0.002 (0.002) | 2.130 (3.129) | 0.193 (0.477) |
| Subordination | 0.004 (0.002) | 18.39 (10.87) | 5.851 (4.392) |
| Termination | 0.009 (0.010) | 1.157 (0.698) | 0.273 (0.213) |
| Compilation | 0.001 (0.001) | 0.077 (0.078) | 0.044 (0.045) |
| Solving | 0.000 (0.000) | 0.838 (0.498) | 0.000 (0.000) |
| Coverage | 0.225 (0.270) | 2173 (2176) | 8.003 (7.190) |
| Worlds | 0.002 (0.002) | 2.810 (1.241) | 2.124 (1.922) |
| Total | 0.275 (0.319) | 2218 (2216) | 19.49 (18.42) |

**Figure 9.** Experimental Data. Modular Twelf (Traditional Twelf) in seconds.

## 4. Implementation

The module system for LF discussed in this paper has been implemented as part of the Twelf system. More information about the implementation and some examples can be found on the project webpage at `http://www.twelf.org/mod/`.

Our implementation of modular Twelf builds upon the code base of the original Twelf. Twelf's own highly modular design aided our effort to introduce namespace management and the treatment of qualified identifiers. Its built-in mechanism for notational definitions is heavily used in the implementation of elaboration in particular for dealiasing. Parts of the central data structures had to be changed to maintain qualified identifiers, and we frequently use hash tables instead of arrays. The additional convenience of the module system does not come at the expense of performance, as we can show next. The experimental results are reported in Figure 9. The experiments compare the running times of various mechanisms inside the two implementations of Twelf when loading a Twelf signature (that does not contain any modules). First, the cut-elimination proof for intuitionistic and classical logic [Pfenning 1995], the formalization of the meta-theory of typed assembly language [Crary 2003] (which consists of about 2500 meta-theorems), and the formalization of the meta-theory of the intermediate language of full Standard ML [Lee et al. 2007] (which consists of about 1300 meta-theorems). All timings are measured in seconds and rounded. The experiments were conducted on a Dell Poweredge 1950 equipped with two dual-core Xeon 5140 2.33GHz processors and 8GB RAM.

## 5. Conclusion

We have described a module system for the logical framework LF that is both expressive and elegant. Besides signatures, it introduces signature morphisms, in form of structures and views.

We believe that our examples have shown that named structures and views provide for elegant representations of inheritance relations and translations. In particular, views can themselves be composed modularly. We give another example in [Horozal and Rabe 2009], where we represent a soundness proof for first-order logic by representing proof and model theory as signatures. We define a view from the proof to the model theory. Both proof and model theory as well as the view are developed separately for each connective and quantifier and then plugged together as in the view `KOLM`.

[1] Such a declaration always exists. For example, if $B$ and $B'$ are types, the definition of the identity as a function from $B$ to $B'$ is only well-typed if $B \equiv B'$. This observation is due to Dan Licata.

Views are custom-tailored toward structural translations that translate constant symbols into compound terms. They are, however, too limited to capture the essence of non-structural translations that are defined by cases and appear frequently, especially in the study of the meta theory of deductive systems. We realize the importance of this observation, but leave further investigations to future work.

In summary, the module system described in this paper is conservative over LF because each signature, structure, or view can be fully elaborated into core LF. It is practical, because it is available to users of the Twelf system and we have shown that it does not degrade runtime performance.

***Acknowledgments*** Our module system is a special case of a more generic system that the first author developed with Michael Kohlhase. Design and implementation of the version described here benefited greatly from discussions with Frank Pfenning and prior work by Kevin Watkins.

## References

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

H. Barendregt. An introduction to generalized type systems. *Journal of Functional Programming*, 1(2):125–154, April 1991.

J. Chrzaszcz. Implementing modules in the coq system. In D. Basin and B. Wolff, editors, *Theorem Proving in Higher Order Logics (TPHOLs 03)*, pages 270–286. Springer Verlag, LNCS 2758, 2003.

CoFI (The Common Framework Initiative). *CASL Reference Manual*, volume 2900 (IFIP Series) of *LNCS*. Springer, 2004.

K. Crary. Toward a foundational typed assembly language. In G. Morrisett, editor, *Proceedings of the 30th ACM Symposium on Principles of Programming Languages*, SIGPLAN Notices, Vol. 38, No. 1, pages 198–212, New Orleans, Louisiana, Jan. 2003. ACM Press.

W. Farmer, J. Guttman, and F. Thayer. IMPS: An Interactive Mathematical Proof System. *Journal of Automated Reasoning*, 11(2):213–248, 1993.

F. Haftmann and M. Wenzel. Constructive type classes in isabelle. In T. Altenkirch and C. McBride, editors, *Types for Proofs and Programs, TYPES 2006*, pages 149–165. Springer Verlag LNCS 4502, 2007.

R. Harper and F. Pfenning. A module system for a programming language based on the LF logical framework. *Journal of Logic and Computation*, 8(1):5–31, 1998.

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

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

F. Horozal and F. Rabe. Representing Model Theory in a Type-Theoretical Logical Framework. In *"Logical and Semantic Frameworks, with Applications"*, 2009. To appear.

F. Kammüller, M. Wenzel, and L. C. Paulson. Locales: A sectioning concept for isabelle. In *Theorem Proving in Higher Order Logics (TPHOLs 99), LNCS 1690*, pages 149–165. Springer, 1999.

D. K. Lee, K. Crary, and R. Harper. Towards a mechanized metatheory of standard ML. In *Proceedings of the 34th Annual Symposium on Principles of Programming Languages*, pages 173–184, New York, NY, USA, 2007. ACM Press. ISBN 1-59593-575-4. doi: http://doi.acm.org/10.1145/1190216.1190245.

D. Licata, R. Simmons, and D. Lee. A simple module system for Twelf. `http://www.cs.cmu.edu/~drl/pubs.html`, 2006.

U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.

F. Pfenning. Structural cut elimination. In D. Kozen, editor, *Proceedings of the Tenth Annual Symposium on Logic in Computer Science*, pages 156–166, San Diego, California, June 1995. IEEE Computer Society Press.

F. Pfenning. Logical frameworks. In *Handbook of automated reasoning*, pages 1063–1147. Elsevier, 2001.

F. Pfenning and C. Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, pages 202–206, Trento, Italy, July 1999. Springer-Verlag LNAI 1632.

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

D. Sannella and A. Tarlecki. Specifications in an arbitrary institution. *Information and Control*, 76:165–210, 1988.

C. Schürmann and M. O. Stehr. An executable formalization of the HOL/Nuprl connection in the meta-logical framework Twelf. In *Proceedings of the 13th International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 150–166, Phnom Penh, Cambodia, 2006. Springer Verlag.