A Practical Module System for LF *

Florian Rabe[1] and Carsten Schürmann[2]

[1] Jacobs University
f.rabe@jacobs-university.de
[2] 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.
In this paper we describe a module system for the logical framework LF. It is based on two main primitives: signatures and signature morphisms, which provide a semantically transparent module level and permit to represent logic translations as homomorphisms. Modular LF is a conservative extension over LF, and defines an elaboration of modular into core LF signatures. We have implemented our design in the Twelf system and used it to modularize large parts of the Twelf example library.

1 Introduction

The Twelf system [PS99] 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 [Cra03] and Standard ML [LCH07], for checking cut-elimination proofs for intuitionistic and classical logic [Pfe95], and for specifying and validating logic morphisms, for example, between HOL and Nuprl [SS06]. 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 [HHP93] in particular.

If one subscribes to the judgment-as-types methodology (as we do in the Twelf community), the defining features of a logical framework, such as its equational theory, determine usually the application areas it excels in. The logical framework LF shines, for example, when applied to areas of programming languages and logics, where variable binding and substitution application are prevalent. LF is dependently typed, it supports higher-order abstract syntax, and its inductive definition of canonical forms has led to complex inductive reasoning and logic programming environments that are well-known to and frequently used by the users of the Twelf system.

* 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.

Retrofitting a logical framework with a module system is therefore 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 neither of the logical framework nor of its reasoning and programming environments.

Therefore, we advocate a module system that is conservative over LF, which means that code that is written using the features of the module system will eventually be elaborated into core LF that is implemented in the Twelf system. After elaboration, the set of tools and algorithms that are already part of the Twelf system, such as mode analysis, termination checking, coverage checking, etc. can without modification still be applied to the elaborated Twelf code.

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 a simple, flexible, and scalable abstraction to express interrelations between signatures (see [ST88,CoF04,AHMS99]).

In the current design, signature morphisms are not aware of meta-theoretic properties yet, such as termination, totality, or coverage; these may have been established for type families in one signature but might not be preserved under a signature morphism. However, this is not a restriction because the user may manually recheck the desired property wherever necessary.

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. Furthermore we provide experimental evidence that the module system for LF does not jeopardize runtime performance. Further examples are available from the project homepage, which include a modular and type directed development of the meta theory of Mini-ML and a modular definition of the algebraic hierarchy.

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 report on our experimental findings that provide evidence that the module system implemented in Twelf does not degrade performance. And finally, we assess results and discuss future work in Section 5.

$$\dfrac{\dfrac{}{A\ \texttt{true}}u \quad \vdots \quad B\ \texttt{true}}{A \supset B\ \texttt{true}} \supset \texttt{I}^u \qquad \dfrac{A \supset B\ \texttt{true} \quad A\ \texttt{true}}{B\ \texttt{true}} \supset \texttt{E} \qquad \dfrac{\dfrac{}{A\ \texttt{true}}u \quad \vdots \quad p\ \texttt{true}}{\neg A\ \texttt{true}} \neg\texttt{I}^{p,u} \qquad \dfrac{\neg A\ \texttt{true} \quad A\ \texttt{true}}{B\ \texttt{true}} \neg\texttt{E}$$

Fig. 1. Intuitionistic Logic

2 Preliminaries

2.1 The Logical Framework LF

The Twelf system is an implementation of the logical framework LF [HHP93] 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	::=	`type` | $\{\texttt{x}:A\}\ K$ | A `->` K
Types:	A, B	::=	a | $A\,M$ | $\{\texttt{x}:A\}\ B$ | A `->` B
Objects:	M	::=	c | `x` | $[\texttt{x}:A]\ M$ | $M_1\, M_2$

where we write $\{\cdot\}$ for the Π-type constructor and $[\cdot]$ for a λ-binder. We will omit type labels whenever they are inferable. Core LF permits declarations of type- or object-level constants. Constants are declared in the form of declarations "a : K." or "c : A.", or definitions "a : K = A." or "c : A = M." Constants c may be used infix, below the declaration "`%infix` n m c." n defines the associativity of c, which can either be `left` or `right`, 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 will not discuss further in this paper, but which will remain available in modular Twelf.

2.2 The Kolmogorov Translation

We illustrate the design of our module system by giving a modular definition of the embedding of classical logic into intuitionistic logic, which is often called the Kolmogorov translation. We focus on the fragment containing 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 the excluded middle), 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

3.1 Syntax

In the past, various module systems for proof assistants have been proposed, for example, for Agda [Nor07], Coq [Chr03], Isabelle [KWP99,HW07], and even LF [HP98,LSL06] itself. With the exception of Agda, these differ from ours in that they do not insist that modules be elaborated into the core theory. In this sense, our design is more closely related to the systems, described in [ST88] and [AHMS99].

With this goal in mind, our central idea is to collect various declarations and definitions in larger named entities, called **signatures**. We use R, S, T for named signatures, and define that two signatures are equal iff they have the same name.

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

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

We will simplify the presentation of LF and describe the module system in terms of pure type systems [Bar91], which collapse the three syntactic categories into terms.

$$\text{Terms:} \quad C ::= T"\vec{c} \mid x \mid \texttt{type} \mid \{\texttt{x}{:}C\}\ C \mid [\texttt{x}{:}C]\ C \mid C\,C$$

Here $T"\vec{c}$ refers to the constant $\vec{c}$ of signature T.

Signature morphisms define mappings between signatures. A morphism from a signature S to T maps every constant c declared in S to a term C over T such that C is typed (or kinded) by $\mu(A)$ where $\mu(-)$ is the homomorphic extension of μ to terms. This homomorphic extension preserves the typing relation and the definitional equality of S (see, e.g., [HST94]).

Signature morphisms come in two flavors: **structures**, which copy and instantiate a signature S into T, and **views**, which translate from a signature S to T. In the following we will look at structures first.

In the simplest case, a structure declaration in a signature T consists of a fresh name s and a signature S. If S declares a constant c, then the structure declaration induces a constant $s.c$ in T by copying c. s induces a signature morphism, which maps all constants $S"c$ to $T"s.c$. In general, this leads to the definition of qualified identifiers $\vec{c} ::= s.\ \ldots\ .s.c$ of constants. Similarly, we define qualified structure identifiers $\vec{s} ::= s.\ \ldots\ .s.s$. In our running example, we get around this complex syntax by using `%open` to introduce constants c abbreviating $s.c$.

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.
}.
```

Here we import `o` and `true` from `JUDGMENTS` using a structure `J`. It induces the constants `IMP"J.o` and `IMP"J.true`. Within `IMP`, we can refer to them as `J.o` and `J.true`, and `%open o true` makes them available as `o` and `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`.

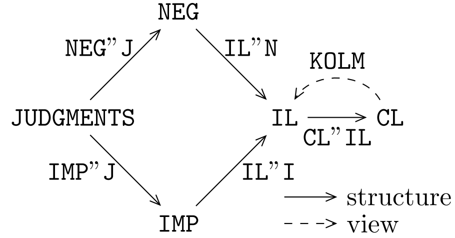
In `%struct N : NEG  = { %struct J := I.J.}`, readers familiar with SML may think of `NEG` as a functor that is passed `I.J` as an argument with the result being assigned to `N`.

Example 5 (Classical Logic).
Finally, by extending intuitionistic logic with the axiom of the excluded middle, we obtain the definition of classical logic.

```
%sig CL = {
   %struct IL : IL %open true ⊃ ¬.
   exm : 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 definition/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. For the sake of convenience, we omit the keywords `%sig`, `%struct`, `%view` from the formal presentation.

A **signature graph** is a multigraph 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. Now 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 with 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. Note that these views are total: There is an assignment for every constant declared in the domain with the only exception of those that are defined.

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

```
%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 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 KOLM : CL -> IL = {
  %struct IL.I := KOLMI.
  %struct IL.N := KOLMN.
  exm :=  [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 the excluded middle.

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, and it is clearly terminating. □

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

Finally, given a set of signature and view declarations, we define signature morphisms μ by $\mu ::= T"\vec{s} \mid v \mid \mu \bullet \mu$. Here $T"\vec{s}$ refers to the structure $\vec{s}$ of the signature T, v refers to a view, and $\mu \bullet \mu'$ 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 also state the semantics of qualified structure identifiers more precisely: The structure CL"IL.I.J is meant to be equal to the morphism IMP"J • IL"I • CL"IL.

Thus, morphisms are lists of named links in the signature graph. While it is straightforward to define equality and normal forms for morphisms (using their semantics as mappings between terms), it is non-trivial to do this in a scalable way. We will return to this observation in Section 3.4.

This concludes the definition of the syntactic categories of our module system for LF, which we summarize in Figure 2. The figure introduces two productions that have not been covered by the above: Structures and views may also be declared as abbreviations of existing morphisms μ. This is not only syntactic sugar — it also makes our language strong enough to express all aspects of the semantics of structures.

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

Fig. 2. The Grammar for Expressions

3.2 Elaboration

The semantics of a structure declaration $s : S := \{\sigma\}$ in T is defined by its elaboration into a set of *induced* constant declarations in the core theory. For example, if S contains $c : A$ and s contains no assignment for c, elaboration induces the declaration $s.c : T\text{"}s(A)$ and similarly, if s contains the assignment $c := B$, then elaboration induces the declaration $s.c : T\text{"}s(A) := B$. The elaboration of an assignment $s := \mu$ where the domain of s is R yields a set of induced assignments containing $s.c := \mu(R\text{"}c)$ for every constant c declared in R.

Formally, we define elaboration by three mutually dependent judgments. The judgment $\mathcal{G} \ggg_T \vec{c} : A := B$ expresses that the declaration $\vec{c} : A := B$ may be induced in the signature T. In the interest of brevity, we write $B = \bot$ if the declaration is of the form $\vec{c} : A$. Similarly to the elaboration of structures in signatures, we elaborate assignments to structures in links. The judgment $\mathcal{G} \ggg_m \vec{c} := B$ expresses that the assignment $\vec{c} := B$ is induced in the link m. If the domain of m has an induced declaration for the constant $\vec{c}$ but m provides no assignment for it, we write $\mathcal{G} \ggg_m \vec{c} := \bot$.

Elaborating 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 $S\text{"}r \bullet T\text{"}s$. Therefore, we use the judgment $\mathcal{G} \ggg_T m : S := D$ to express that m is a link from S to T (alternative reading: a structure expression over T of type S) defined by D where D may be a morphism μ or a list of assignments $\{\sigma\}$.

These three judgments are defined in Fig. 3 and 4. There $\mu(C)$ denotes the result of applying the morphism μ to the expression C, which is defined below. These judgments can be seen as functions if there are no name clashes in $\mathcal{G}$: No signature graph or signature may declare the same identifier twice, and no link may assign an object to the same identifier twice. In this case we write $\mathcal{G}^T(\vec{c}) = (A, B)$ if $\mathcal{G} \ggg_T \vec{c} : A := B$, and $\mathcal{G}^v(\vec{c}) = B$ if $\mathcal{G} \ggg_T \vec{c} := B$, and $\mathcal{G}(m) = (S, T)$ if $\mathcal{G} \ggg_T m : S := _$.

Finally we define the application of a morphism μ to a term C by induction on μ and C. The definition is relative to a fixed signature graph $\mathcal{G}$, which we drop from the notation.

$$
\begin{array}{ll}
\mu \bullet \mu'(S\text{"}\vec{c}) & := \mu'(\mu(S\text{"}\vec{c})) \\
m(S\text{"}\vec{c}) & := \left\{ \begin{array}{ll} m(B) & \text{if } B \neq \bot \\ B' & \text{if } B = \bot,\ m = v \text{ view} \\ T\text{"}s.\vec{c} & \text{if } B = \bot,\ m = T\text{"}s \text{ structure} \end{array} \right\} \begin{array}{l} \text{where } \mathcal{G}^S(\vec{c}) = (_, B), \\ \quad \mathcal{G}^m(\vec{c}) = B' \end{array} \\
\mu(\mathit{type}) & := \mathit{type} \\
\mu(x) & := x \\
\mu([x : A]C) & := [x : \mu(A)]\mu(C) \\
\mu(\{x : A\}C) & := \{x : \mu(A)\}\mu(C) \\
\mu(C\ C') & := \mu(C)\ \mu(C')
\end{array}
$$

In the interest of brevity, and without loss of generality we let $\mu(\bot) = \bot$.

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

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

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

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

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

Fig. 3. Elaboration of Structures

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

$$\frac{\mathcal{G} \ggg_T T"s : S := _ \quad \mathcal{G} \ggg_S S"\vec{r} : R := _}{\mathcal{G} \ggg_T T"s.\vec{r} : R := S"\vec{r} \bullet T"s}$$

Fig. 4. Elaborated Links

3.3 Type System

In this section we present an inference system to define **well-formed** expressions. The judgments are given in Fig. 5. 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 module, symbol, or assignment 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.

In order to express that a declaration or assignment for the identifier n can be added to the signature or link N, we use the auxiliary judgment $noClash(\mathcal{G}, N, n)$, which is true if one of the following holds. (i) $\mathcal{G}$ ends in the declaration of a signature with name N and n is an identifier that has not been declared in N yet; (ii) $\mathcal{G}$ ends in the declaration of a view N and $\mathcal{G}^N(p) = \bot$ for every qualified identifier p for which $n = p$, $n.n' = p$, or $n = p.p'$; (iii) $N = T"s$ and $\mathcal{G}$ ends in the declaration of a signature T whose body ends in the declaration of a struc-

Judgment	Intuition
$\vdash \mathcal{G}$	$\mathcal{G}$ is a well-formed signature graph.
$\mathcal{G} \rhd D$	The declaration D can be added to $\mathcal{G}$.
$\mathcal{G} \vdash_T \mu : S$	μ 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.

Fig. 5. Main Judgments

ture s, and $\mathcal{G}^N(-)$ satisfies the same property as in (ii). Similarly, $noClash(\mathcal{G}, N)$ holds iff N does not occur as the identifier of a signature or view in $\mathcal{G}$.

Furthermore, 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. 6. 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 (i.e., provide assignments for all constants that do not have a definiens) 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. 7.

Firstly, 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 constant or structure declaration.

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. Alternatively, $View_\mu$ adds a view that is defined by an existing morphism.

The rules for structures correspond to those for views: $Str_\emptyset$ and Str_{Ass} construct structures by successively adding well-formed assignments, and Str_μ adds a structure that is defined by an existing morphism.

The rules above the dotted line in Fig. 7 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 well-typedness of an assignment $\vec{c} := B$. The first three premises look up the domain and codomain of the last link 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. Then the final premise type-checks B against the translation of

$$\frac{}{\vdash \cdot}\mathcal{G}_\emptyset \qquad \frac{\vdash \mathcal{G} \quad \mathcal{G} \rhd T := \{\Sigma\}}{\vdash \mathcal{G},\, T := \{\Sigma\}} Sig$$

$$\frac{\vdash \mathcal{G} \quad \mathcal{G} \rhd v : S \to T := \{\sigma\} \quad \mathcal{G}^v(\vec{c}) \neq \bot \text{ whenever } \mathcal{G}^S(\vec{c}) = (A, \bot)}{\vdash \mathcal{G},\, v : S \to T := \{\sigma\}} View$$

$$\frac{noClash(\mathcal{G}, T)}{\mathcal{G} \rhd T := \{\cdot\}} Sig_\emptyset \qquad \frac{\mathcal{G} \rhd T := \{\Sigma\} \quad \mathcal{G},\, T := \{\Sigma\} \rhd D_{Sym}}{\mathcal{G} \rhd T := \{\Sigma, D_{Sym}\}} Sym$$

$$\frac{noClash(\mathcal{G}, v) \quad S \in Sig(\mathcal{G}) \quad T \in Sig(\mathcal{G})}{\mathcal{G} \rhd v : S \to T := \{\cdot\}} View_\emptyset \qquad \frac{noClash(\mathcal{G}, T, s) \quad S \in Sig(\mathcal{G}) \setminus \{T\}}{\mathcal{G} \rhd s : S := \{\cdot\}} Str_\emptyset$$

$$\frac{\mathcal{G} \rhd v : S \to T := \{\sigma\} \quad \mathcal{G},\, v : S \to T := \{\sigma\} \rhd D_{Ass}}{\mathcal{G} \rhd v : S \to T := \{\sigma, D_{Ass}\}} View_{Ass}$$

$$\frac{\mathcal{G},\, T := \{\Sigma\} \rhd s : S := \{\sigma\} \quad \mathcal{G},\, T := \{\Sigma,\, s : S := \{\sigma\}\} \rhd D_{Ass}}{\mathcal{G},\, T := \{\Sigma\} \rhd s : S := \{\sigma, D_{Ass}\}} Str_{Ass}$$

$$\frac{noClash(\mathcal{G}, v) \quad \mathcal{G} \vdash_T \mu : S}{\mathcal{G} \rhd v : S \to T := \mu} View_\mu \qquad \frac{noClash(\mathcal{G}, T, s) \quad \mathcal{G} \vdash_T \mu : S}{\mathcal{G} \rhd s : S := \mu} Str_\mu$$

Fig. 6. Structural Rules

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 very 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 μ 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 μ agrees with $S"\vec{s} \bullet m$ for all constants for which m is already determined.

$$R \xrightarrow{S"\vec{s}} S \xrightarrow{m} T, \quad R \xrightarrow{\mu} T$$

The rules below the dotted line in Fig. 7 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 [Pfe01]). 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 $\mathcal{M}_\bullet$ construct morphisms as sequences of links. Composition is written in diagrammatic order, i.e., from the domain to the codomain.

$$\frac{noClash(\mathcal{G}, T, c) \quad \mathcal{G} \vdash_T B : A}{\mathcal{G} \rhd c : A := B}\,Con \qquad \frac{noClash(\mathcal{G}, m, \vec{c}) \quad \mathcal{G}(m) = (S, T) \quad \mathcal{G}^S(\vec{c}) = (A, \bot) \quad \mathcal{G} \vdash_T B : m(A)}{\mathcal{G} \rhd \vec{c} := B}\,ConAss$$

$$\frac{noClash(\mathcal{G}, m, \vec{s}) \quad \mathcal{G}(m) = (S, T) \quad \mathcal{G}(S\text{"}\vec{s}) = (R, S) \quad \mathcal{G} \vdash_T \mu : R \quad \begin{array}{c}[\mathcal{G}^S(\vec{s}.\vec{c}) = (_, B),\ B \neq \bot] \\ \vdots \\ \mathcal{G} \vdash_T \mu(R\text{"}\vec{c}) \equiv m(B)\end{array}}{\mathcal{G} \rhd \vec{s} := \mu}\,StrAss$$

$$\frac{\mathcal{G}^T(\vec{c}) = (A, _)}{\mathcal{G} \vdash_T T\text{"}\vec{c} : A}\,\mathcal{T}_: \qquad \frac{\mathcal{G}^T(\vec{c}) = (_, B),\ B \neq \bot}{\mathcal{G} \vdash_T T\text{"}\vec{c} \equiv B}\,\mathcal{T}_\equiv$$

$$\frac{\mathcal{G}(m) = (S, T)}{\mathcal{G} \vdash_T m : S}\,\mathcal{M}_m \qquad \frac{\mathcal{G} \vdash_S \mu : R \quad \mathcal{G} \vdash_T \mu' : S}{\mathcal{G} \vdash_T \mu \bullet \mu' : R}\,\mathcal{M}_\bullet$$

Fig. 7. Typing Rules

3.4 Meta Theory

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

Definition 1. *Assume $\mathcal{G} \vdash_T \mu : S$ and $\mathcal{G} \vdash_T \mu' : S$. We define the judgment $\mathcal{G} \vdash \mu \equiv \mu'$ to hold iff for all $\vec{c}$ for which $\mathcal{G}^S(\vec{c})$ is defined we have $\mathcal{G} \vdash_T \mu(S\text{"}\vec{c}) \equiv \mu'(S\text{"}\vec{c})$.*

Theorem 1. *Assume $\mathcal{G} \vdash_T \mu : S$. If $\mathcal{G} \vdash_S C : A$, then $\mathcal{G} \vdash_T \mu(C) : \mu(A)$. 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 [Rab08].

The following result is the cornerstone of adequacy proofs. For example, it immediately entails the adequacy of the LF encoding of intuitionistic logic under structure sharing in Example 4.

Theorem 2. *Assume* $\vdash \mathcal{G}$. *If there is an assignment* $\vec{s} := \mu$ *in a link* m *from* S *to* T *in* $\mathcal{G}$, *then* $\mathcal{G} \vdash S"\vec{s} \bullet m \equiv \mu$.

Proof. This is a special case of the results given in [Rab08].

Finally, we show that modular LF is conservative over core LF. The core of the argument is that elaboration is sound. The only caveat is easily explained: Qualified identifiers in modular LF need to be considered constants in core LF, which means that """ and "." may occur in constant names.

Theorem 3. *Assume a signature graph* $\mathcal{G}$. *Let* Σ *be the core LF signature containing the declarations*

- *for all signatures* T *declared in* $\mathcal{G}$: *whenever* $\mathcal{G} \ggg_T \vec{c} : A := B$, *the declaration* $T"\vec{c} : A := B$ *(where* B *is omitted if* $B = \bot$*)*,
- *for all views* v *with domain* S *declared in* $\mathcal{G}$: *whenever* $\mathcal{G} \ggg_S \vec{c} : A := \bot$, *the declaration* $v"\vec{c} : v(A) := B$ *where* $\mathcal{G}^v(\vec{c}) = B$,

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

Proof. This is a special case of the results given in [Rab08]. The only modification of the argument is that here Σ is always ill-formed if any view in $\mathcal{G}$ is not total because it will contain the illegal symbol $\bot$.

As core LF signatures elaborate to themselves, modular LF is a conservative extension over core LF.

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 can be found on the webpage at `http://www.twelf.org/mod`. The implementation uses the same code base as the traditional Twelf, except that it uses hash tables for constant lookup, which renders the modular Twelf implementation at times faster than the traditional one. Figure 8 provides empirical evidence for this claim. All timings are measured in seconds and rounded. The experiments where conducted on a Dell Poweredge 1950 equipped with two dual-core Xeon 5140 2.33GHz processors and 8GB RAM.

The experiments compare the run times of various mechanisms inside Twelf when loading a core Twelf signature. We conclude that the module system does not come at the price of lost efficiency — on the contrary it enables us to study *separate checking* in analogy to *separate compilation* in future work. We conducted experiments with three larger developments in Twelf: first, the cut-elimination proof for intuitionistic and classical logic [Pfe95], the formalization of the meta-theory of typed assembly language [Cra03] (which consists of about 2500 meta-theorems), and the formalization of the meta-theory of the intermediate language of full Standard ML of New Jersey [LCH07] (which consists of about 1300 meta-theorems).

	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)

Fig. 8. Experimental Data. Modular Twelf (Traditional Twelf) in seconds.

5 Conclusion

We have described a practical module system for the logical framework LF that is deceptively simple because it is designed around signatures and signature morphisms. It is expressive, sound, and conservative over LF because each signature, structure, or view is fully elaborated into core LF, which implies that the module system per se does not pollute any prior and future meta-theoretic analysis of LF encodings. And finally, 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 generic system that the first author developed with Michael Kohlhase. Design and implementation of the LF version benefited from discussions with Frank Pfenning and previous work by Kevin Watkins.

References

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

Bar91. Henk Barendregt. An introduction to generalized type systems. *Journal of Functional Programming*, 1(2):125–154, April 1991.

Chr03. Jacek Chrzaszcz. Implementing modules in the coq system. In David Basin and Burkhart Wolff, editors, *Theorem Proving in Higher Order Logics (TPHOLs 03)*, pages 270–286. Springer Verlag, LNCS 2758, 2003.

CoF04. CoFI (The Common Framework Initiative). *Casl Reference Manual*, volume 2900 (IFIP Series) of *LNCS*. Springer, 2004.

Cra03. Karl Crary. Toward a foundational typed assembly language. In Greg 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, January 2003. ACM Press.

HHP93. Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, January 1993.

HP98. Robert Harper and Frank Pfenning. A module system for a programming language based on the LF logical framework. *Journal of Logic and Computation*, 8(1):5–31, 1998.

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

HW07. Florian Haftmann and Makarius 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.

KWP99. Florian Kammüller, Markus Wenzel, and Lawrence C. Paulson. Locales: A sectioning concept for isabelle. In *Theorem Proving in Higher Order Logics (TPHOLs 99), LNCS 1690*, pages 149–165. Springer, 1999.

LCH07. Daniel K. Lee, Karl Crary, and Robert 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.

LSL06. Daniel Licata, Rob Simmons, and Daniel Lee. A simple module system for Twelf. `http://www.cs.cmu.edu/~drl/pubs.html`, 2006.

Nor07. Ulf 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.

Pfe95. Frank 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.

Pfe01. F. Pfenning. Logical frameworks. In *Handbook of automated reasoning*, pages 1063–1147. Elsevier, 2001.

PS99. Frank Pfenning and Carsten 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.

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

SS06. Carsten Schürmann and Mark Oliver 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.

ST88. D. Sannella and A. Tarlecki. Specifications in an arbitrary institution. *Information and Control*, 76:165–210, 1988.