Representing Isabelle in LF

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

LF has been designed and successfully used as a meta-logical framework to represent and reason about object logics. Here we give a representation of the Isabelle logical framework in LF using the recently introduced module system for LF. The major novelty of our approach is that we can naturally represent the advanced Isabelle features of type classes and locales.

Our representation of type classes relies on a feature so far lacking in the LF module system: morphism variables and abstraction over them. While conservative over the present system in terms of expressivity, this feature is needed for a representation of type classes that preserves the modular structure. Therefore, we also design the necessary extension of the LF module system.

1 Introduction

Both Isabelle and LF were developed at roughly the same time to provide formal proof theoretic frameworks in which object logics can be defined and studied. Both use the Curry-Howard correspondence to represent the proofs of the object logic as terms of the meta-logic.

Isabelle ([13, 14]) is based on intuitionistic higher-order logic ([2]) with shallow polymorphism and was designed as a generic LCF-style interactive theorem prover. LF ([6]) is the corner of the λ-cube ([1]) that extends simple type theory with dependent function types and is inspired by the judgments-as-types methodology ([10]). We will work with the Twelf implementation of LF ([16]).

It is straightforward to represent Isabelle's underlying logic as an object logic of LF (see, e.g., [6]). However, Isabelle provides a number of advanced features that go beyond the base logic and that cannot be easily represented in other systems. These include in particular a module system ([9, 5]) and a structured proof language ([11]).

Recently, we gave a module system for LF in [18]. We wanted to choose primitive notions that are so simple that they admit a completely formal semantics. While such formal semantics are commonplace for type theories – in the form of inference systems – they quickly get very complex for module systems on top of type theories. At the same time these primitives should be expressive enough to admit natural representations of modular design patterns. Here by "natural", we mean that we are willing to accept lossy (in the sense of being non-invertible) encodings of modular specifications as long as their modular structure of sharing and reuse is preserved.

In this paper we give such a representation of the Isabelle module system in the LF module system. The main idea of the encoding is that all modules of Isabelle (theories, locales, type classes) are represented as LF signatures, and that all relations between Isabelle modules (imports, sublocales, interpretations, subclasses, instantiations) are represented as LF signature morphisms.

Thus, our contribution is two-fold. Firstly, we validate the design of the LF module system by showing that it provides just the right primitives needed to represent the Isabelle module system. Actually, before arriving at that conclusion we identify one feature that we have to add to the LF module system: abstraction over morphisms. And secondly, the encoding of Isabelle in LF is useful to integrate Isabelle and LF specifications. Moreover, for researchers familiar with LF but not with Isabelle, this paper can complement the Isabelle documentation with an LF-based perspective on the foundations of Isabelle.

In Sect. 2, we will repeat the basics of Isabelle and LF to make the paper self-contained. In Sect. 3, we extend the LF module system with abstraction over morphisms. Then we give our representation in Sect. 4.

2 Preliminaries

2.1 Isabelle

Isabelle is a grown and widely used system, which has led to a rich ontology of Isabelle declarations. We will only consider the core and module system declarations in this paper. And even among those, we will restrict attention to a proper subset of Isabelle's power.

For the purposes of this paper, we make some minor adjustments for simplicity and consider Isabelle's language to be generated by the grammar in Fig. 1. Here | and * denote alternative and repetition, and we use special fonts for *nonterminals* and `keywords`.

theory	::=	`theory` *name* `imports` *name** `begin` *thycont* `end`
thycont	::=	(*locale* | *sublocale* | *interpretation* |
	|	*class* | *instantiation* | *thysymbol*)*
locale	::=	`locale` *name* = (*name* : *instance*)* for *locsymbol** + *locsymbol**
sublocale	::=	`sublocale` *name* $<$ *instance proof**
interpretation	::=	`interpretation` *instance proof**
instance	::=	*name* `where` *namedinst**
class	::=	`class` *name* = *name** + *locsymbol**
instantiation	::=	`instantiation` *type* :: (*name**)*name* `begin` *locsymbol** *proof** `end`
thysymbol	::=	`consts` *con* | `defs` *def* | `axioms` *ax* | `lemma` *lem*
	|	`typedecl` *typedecl* | `types` *types*
locysymbol	::=	`fixes` *con* | `defines` *def* | `assumes` *ax* | `lemma` *lem*
con	::=	*name* :: *type*
def	::=	*name* : *name var** $\equiv$ *term*
ax	::=	*name* : *Prop*
lem	::=	*name* : *Prop proof*
typedecl	::=	(*var**) *name*
types	::=	(*var**) *name* = *type*
namedinst	::=	`name` = *term*
type	::=	*var* :: *name* | *name* | (*type*, ..., *type*) *name* | *type* $\Rightarrow$ *type* | *prop*
term	::=	*var* | *name* | *name term** | λ(*var* :: *type*)*.*term*
Prop	::=	*Prop* $\Longrightarrow$ *Prop* | $\bigwedge$(*var* :: *type*)*.*Prop* | *term* $\equiv$ *term*
proof	::=	a proof term
name, *var*	::=	identifier

Figure 1: Simplified Isabelle Grammar

A *theory* is a named group of declarations. Theories may use `imports` to import other theories, which yields a simple module system. Within theories, *locale* and type *class* declarations provide further sources of modularity. Theories, locales, and type classes may be related using a number of declarations as described below.

The *core declarations* occurring in theories (*thysymbol*) and locales (*locsymbol*) are quite similar. *consts* and *fixes* declare typed constants $c :: \tau$. *defs* and *defines* declare definitions for a constant f taking n arguments as $f\_def : f\, x_1\, \ldots x_n \equiv t$ where t is a term in the variables x_i. *axioms* and *assumes* declare named axioms a asserting a proposition φ as $a : \varphi$. *lemma* declares a named lemma l asserting φ with proof P as $l : \varphi\, P$.

Furthermore, in theories, *typedecl* declares n-ary type operators t as $(\alpha_1, \ldots, \alpha_n)\, t$, and similarly *types* declares an abbreviation t for a type τ in the variables α_i as $(\alpha_1, \ldots, \alpha_n)\, t = \tau$. Locales do not contain type declarations. However, they may declare new types indirectly by declaring constants whose types have free type variables, e.g., $\circ : \alpha \Rightarrow \alpha \Rightarrow \alpha$ in a locale for groups. References to these types are made indirectly using type inference, e.g., if there is another constant $e : \beta$, then an axiom $x \circ e = x$ enforces that α and β refer to the same type.

The constant declarations within a locale serve as parameters that can be instantiated. The intuition is that a locale *instance* loc `where` σ takes the locale with name loc and translates it into a new context (which can be a theory or another locale). Here σ is a list of parameter instantiations (*namedinst*) of the form $c = t$ instantiating the parameter c of loc with the term t in that new context.

Locale instances are used in two places. Firstly, locale declarations may contain a list of instances used to inherit from other locales. In a locale declaration

$$\texttt{locale}\ loc = ins_1 : loc_1\ \texttt{where}\ \sigma_1\ \ \ldots\ \ ins_n : loc_n\ \texttt{where}\ \sigma_n\ \texttt{for}\ \Sigma + \Sigma'$$

the new locale loc inherits via n named instances: Instance ins_i inherits from the locale loc_i via the list of parameter instantiations σ_i. Σ and Σ' declare the core declarations of the locale.

The set of constant declarations of the locale is defined as follows: (i) The declarations in Σ logically precede the instances, i.e., are available in σ_i and Σ'. (ii) A copy of the declarations of each loc_i translated by σ_i is available in each σ_j for $j > i$ and in Σ'; the names ins_i serve as qualifiers to resolve name clashes if two declarations of the same name are present. (iii) The declarations in Σ' are only available in Σ'.

The σ_i do not have to instantiate all parameters of loc_i – parameters that are not instantiated become parameters of loc. Thus, the parameters of loc consist of the not-instantiated parameters of the loc_i and the constants declared in Σ and Σ'.

Secondly, a declaration `sublocale` $loc' < loc$ `where` $\sigma\, \pi$ postulates a translation from loc to loc', which maps the parameters of loc according to σ. The axioms and definitions of loc induce proof obligations over loc' that must be discharged by giving a list π of proofs. If all proof obligations are discharged, all theorems about loc can be translated to yield theorems about loc', and Isabelle does that automatically. A locale `interpretation` is very similar to a `sublocale`. The difference is that all loc expressions are translated into the current theory rather than into a second locale.

The concepts of locales and *type classes* have recently been aligned ([5]) and in particular type classes are also locales. But the syntax still reflects their different use cases. A type class is a locale inheriting only from other type classes and only without parameter instantiations. Thus, the locale syntax can be simplified to `class` $C = C_1 \ldots C_n + \Sigma$ where C inherits from the C_i. All declarations in Σ may refer to at most one type variable, which can be assumed to be of the form $\alpha :: C$. The intuition is that Σ provides operations $c_1, \ldots, c_n$ that are polymorphic in the parametric type α and axioms about them.

An instance of a type class is a tuple $(\tau, c_1\_def, \ldots, c_n\_def)$ where τ is a type and c_i_def is a definition for c_i at the type τ. Because every c_i can only have one definition per type, the definitions can be inferred from the context and be dropped from the notation; then a type class can be seen as a unary predicate on types τ. Type class instantiations are of the form

$$\texttt{instantiation}\, t :: (C_1, \dots, C_n)C \,\texttt{begin}\, \Sigma\, \pi \,\texttt{end}$$

where t is an n-ary type operator, i.e., a type with n free type variables α_i. Σ contains the definitions for the operations of C at the type $(\alpha_1, \dots, \alpha_n)t$ in terms of the operations of the instances $\alpha_i :: C_i$. This creates proof obligations for the axioms of C, and we assume that all the needed proofs are provided as a list π. The semantics is that if $\tau_i :: C_i$ are type class instances, then so is $(\tau_1, \dots \tau_n)t :: C$. Note that this includes base types for $n = 0$.

Example 1. The following sketches type class for ordering ands semilattices with universe α, ordering $\leq$, and infimum $\sqcap$ (where we omit inferable types and write $\cdot$ for empty lists):

$$\begin{array}{lll}
\texttt{class} & \mathit{order} & = \;\; \cdot + \leq :: \alpha \Rightarrow \alpha \Rightarrow \mathit{prop} \\
\texttt{class} & \mathit{semlat} & = \;\; \mathit{order} + \sqcap :: \alpha \Rightarrow \alpha \Rightarrow \alpha \\
\texttt{locale} & \mathit{lat} & = \;\; \mathit{inf} : \mathit{semlat}\; \texttt{where}\; \cdot \\
 & & \mathit{sup} : \mathit{semlat}\; \texttt{where}\; \leq = \lambda x \lambda y.\, y\, \mathit{inf}.\leq\, x\; \texttt{for}\; \cdot + \cdot
\end{array}$$

Here the omitted axioms in *semlat* would enforce that the type variables α in the types of $\leq$ and $\sqcap$ refer to the same type. Then a locale for lattices is obtained by using two named instances of a semilattice where the second one flips the ordering. The parameters of *lat* are $\mathit{inf}.\leq$ (the ordering), $\mathit{inf}.\sqcap$ (the infimum), and $\sup.\sqcap$ (the supremum), but not $\sup.\leq$ which is instantiated.

Finally the *inner syntax* for *term*s, *type*s, *prop*ositions, and *proof* terms – also called the Pure language – is given by an intuitionistic higher-order logic with shallow polymorphism. Types are formed from type variables $\alpha :: C$ for type classes C, base types, type operator applications, function types, and the base type *prop* of propositions. Type class instances of the form $\tau :: C$ are formed from type variables $\alpha :: C$ and type operator applications $(\tau_1, \dots, \tau_n)t$ for a corresponding instantiation $t :: (C_1, \dots, C_n)C$ and type class instances $\tau_i :: C_i$. We will assume every type to be a type class instance by using the special type class *Type* of all types.

Terms are formed from variables, typed constants, application, and lambda abstraction. Constants may be polymorphic in the sense that their types may contain free type variables. When a polymorphic constant is used, Isabelle automatically infers the type class instances for which the constant is used. Propositions are formed from implication, universal quantification over any type, and equality on any type.

We always assume that all types are fully reconstructed. Similarly, we cover neither the Isar proof language nor tactic invocations and simply assume proof terms from Pure's natural deduction calculus.

2.2 LF

The non-modular declarations in an LF signature are *kinded type family* symbols $a : K$ and *typed constants* $c : A$. Both may carry definitions, e.g., $c : A = t$ introduces c as an abbreviations for t. The objects of Twelf are *kinds* K, *kinded type families* $A : K$, and *typed terms* $t : A$. `type` is the kind of types, and $A \rightarrow \texttt{type}$ is the kind of type families indexed by terms of type A. We use Twelf notation for binding and application: The type $\Pi_{x:A}B(x)$ of dependent functions taking $x : A$ to an element of $B(x)$ is written $\{x : A\}\, B\, x$, and the function term $\lambda_{x:A}t(x)$ taking $x : A$ to $t(x)$ is written $[x : A]\, t\, x$. We write $A \rightarrow B$ instead of $\{x : A\}B$ if x does not occur in B, and we will also omit the types of bound variables if they can be inferred.

The Twelf *module system* ([19]) is based on the notions of signatures and signature morphisms ([8]). Given two signatures $\mathtt{sig}\, S = \{\Sigma\}$ and $\mathtt{sig}\, T = \{\Sigma'\}$, a signature morphism from S to T is a type/kind-preserving map μ of Σ-symbols to Σ'-expressions. Thus, μ maps every constant $c : A$ of Σ to a term $\mu(c) : \overline{\mu}(A)$ and every type family symbol $a : K$ to a type family $\mu(a) : \mu(K)$. Here, $\mu(-)$ doubles as the homomorphic extension of μ, which maps closed Σ-expressions to closed Σ' expressions. Signature morphisms preserve typing and kinding, i.e., if $\vdash_\Sigma E : F$, then $\vdash_{\Sigma'} \mu(E) : \mu(F)$.

Signature declarations are straightforward: $\mathtt{sig}\, T = \{\Sigma\}$. Signatures may be nested and may include other signatures. Basic *morphisms* are given explicitly as $\{\sigma : S \to T\}$, and composed morphisms are formed from basic morphisms, identity, composition, and two kinds of named morphisms: views and structures. [1]

We will use the following grammar where the structure identifiers $T.s$ and the symbol identifiers $S.c^\mu$ and $S.a^\mu$ are described below:

Signature graphs	G	$::=$	$\cdot \mid G,\ \mathtt{sig}\, T = \{\Sigma\} \mid G,\ \mathtt{view}\, v : S \to T = \mu$
Signatures	Σ	$::=$	$\cdot \mid \Sigma,\ \mathtt{sig}\, T = \{\Sigma\} \mid \Sigma,\ \mathtt{include}\, S$
			$\mid \Sigma,\ \mathtt{struct}\, s : S = \{\sigma\} \mid \Sigma,\ c : A[=t] \mid \Sigma,\ a : K[=A]$
Morphisms	σ	$::=$	$\cdot \mid \sigma,\ \mathtt{struct}\, s := \mu \quad \mid \sigma,\ c := t \quad \mid \sigma,\ a := A$
Compositions	μ	$::=$	$T.s \mid \{\sigma : S \to T\} \mid v \mid id \mid incl \mid \mu\ \mu$
Contexts	Γ	$::=$	$\cdot \mid \Gamma,\ x : A$
Kinds	K	$::=$	$\mathtt{type} \mid A \to K$
Type families	A	$::=$	$S.a^\mu \mid A\ t \mid \{x : A\}A$
Terms	t	$::=$	$S.c^\mu \mid x \mid [x : A]t \mid t\ t$

Modular LF uses the following judgments for well-formed syntax:

$\rhd G$	well-formed signature graphs
$G \rhd \mu : S \to T$	morphism between signatures S and T declared in G
$G \rhd_T \Gamma\ \mathtt{Ctx}$	contexts for signature T
$G;\Gamma \rhd_T E : E'$	typing in signature T

The judgment for signature graphs mainly formalizes uniqueness of identifiers and type-preservation of morphisms based on the typing judgment for expressions. The judgments for contexts and typing are essentially the same as for non-modular LF except that the identifiers available in signature T and their types are determined by the module system. Therefore, we only describe the judgments for identifiers and morphisms and refer to [18] for details.

Morphisms First of all, to simplify the language, we will employ the following condition on all morphisms from S to T: T must include all signatures that S includes, and if S includes R, the application of μ to symbols of R is the identity. In particular, views and structures may only be declared if this condition holds.

Firstly, the semantics of a *structure* declaration $\mathtt{struct}\, s : S = \{\sigma\}$ in T is that it induces (i) for every constant $c : A$ of S a constant $s.c : T.s(A)$ in T, and (ii) a morphism $T.s$ from S to T that maps every symbol c of S to $s.c$. Here σ is a partial morphism from S to T, and if σ contains $c := t$, the constant $s.c$ is defined as t. In particular, t must have type $T.s(A)$ over T. The same holds for type family symbols a. Thus, structures instantiate parametric signatures.

[1] Anonymous morphisms are actually not present in [19]. They are easy to add conceptually, but are a bit harder to add to Twelf as they violate the phase distinction between modular and non-modular syntax kept by the other declarations. We will need them later on.

Because structures are named, a signature may have multiple structures of the same signature, which are all distinct. For example, if S already contains a structure r instantiating a third signature R, then `struct` r' $: R$ in T leads to the two morphisms r' and the composition $S.r\ T.s$ from R to T and two copies of the constants of R. Structures may instantiate whole structures at once: If T declares instead `struct` r' $: R = \{$`struct` $r := T.r'\}$, then the two copies of R are shared. More generally, σ may contain instantiations `struct` $r := \mu$ for a morphism μ from R to T, which is equivalent to instantiating every symbol c of R with $\mu(c)$. Another way to say this is that the diagram on the right commutes.

$$\begin{tikzcd} R \arrow[dr, "\mu"] \arrow[dd, "S.r"'] & \\ & T \\ S \arrow[ur, "T.s"'] & \end{tikzcd}$$

Secondly, the semantics of *anonymous morphisms* $\{\sigma : S \to T\}$ is straightforward. They are well-formed if σ is total and map all constants according to σ. Thirdly, *views* v are just names given to existing morphisms.

Fourthly, *inclusion*, *identity* and *composition* are defined by

$$\frac{\texttt{sig}\, T \ = \ \{\Sigma\} \text{ in } G \quad \texttt{include}\, S \text{ in } \Sigma}{G \rhd incl : S \to T}\, incl$$

$$\frac{\texttt{sig}\, T \ = \ \{\Sigma\} \text{ in } G}{G \rhd id : T \to T}\, id \qquad \frac{G \rhd \mu : R \to S \quad G \rhd \mu' : S \to T}{G \rhd \mu\ \mu' : R \to T}\, comp$$

Identifiers Defining which symbol identifiers are available in a signature is intuitively easy, but a formal definition can be cumbersome because all included symbols and those induced by structures have to be computed along with their translated types and definitions. Using morphisms and the novel notation $S.c^{\mu}$ for symbol identifiers, we can give a very elegant definition:

$$\frac{\texttt{sig}\, S \ = \ \{\Sigma\} \text{ in } G \quad c : E \text{ in } \Sigma \quad G \rhd \mu : S \to T}{G \rhd_T S.c^{\mu} : \mu(E)}\, tp$$

and similarly for defined symbols and type family constants. The prize to pay is an awkward notation, but we can recover the usual notations as follows:

- id yields local symbols, and we write c instead of $T.c^{id}$.
- $incl$ yields included symbols, and we write $S.c$ instead $S.c^{incl}$.
- If T contains a structure from S, we have $G \rhd T.s : S \to T$, and we write $s.c$ instead of $S.c^{T.s}$. Accordingly, we introduce constants $s.r.c$ for composed morphisms $S.r\ T.s$ from R to T, and so on.
- Al other identifiers $T.c^{\mu}$, e.g., those where μ contains views or anonymous morphisms, are reduced to one of the other cases by applying the morphism.

Functors While views are well-established in logical frameworks based on model theory (see, e.g., [4, 20]), they are an unusual feature in proof theoretical frameworks. (In fact, the LF module system has been criticized for using views instead of functors or even – in light of [7] – for using either one rather than only structures.) Therefore, we quickly describe how functors are a derived notion in the presence of views and anonymous morphisms.

Assume a functor F from S to T. Its input is a structure s instantiating S, and its output is a list τ of instantiations for the symbols of T. Here τ may refer to the symbols induced by s. We can write this in LF as

$$\mathtt{sig}\, F_0 \;=\; \{\mathtt{struct}\, s \,:\, S\} \quad \mathtt{view}\, F \,:\, T \,\to F_0 \,=\{\tau\}$$

Now given a theory D, we can understand instances of a signature S over D as morphisms from S to D. This is justified because a morphism from S to D realizes every declaration of S in terms of D. More generally we can think of morphisms from S to D as implementations or models of S in terms of D. The application of F should map instances of S to instances of T. Thus, given a morphism μ from S to D, we can write the application $F(\mu)$ as the composed morphism

$$F\,\{\mathtt{struct}\, s := \mu : F_0 \to D\}$$

which is indeed a morphism from T to D.

3 Morphism Variables in LF

We add a feature to the LF module system that permits morphism variables and abstraction over them. For morphism variables, the grammar is extended as follows:

$$\text{Contexts} \quad \Gamma \quad ::= \quad \Gamma,\, X : S \qquad \text{Morphisms} \quad \mu \quad ::= \quad \mu\, X$$

Due to the presence of morphism variables, the judgment for well-formed morphisms must be amended to depend on the context. Then we can give the typing rules as:

$$\frac{G \rhd_T \Gamma\,\mathtt{Ctx} \quad \mathtt{sig}\, S \;=\; \{\Sigma\} \text{ in } G}{G \rhd_T \Gamma,\, X : S\,\mathtt{Ctx}}\,\mathit{contmor} \qquad \frac{G \rhd_T \Gamma,\, X : S\,\mathtt{Ctx}}{G;\, \Gamma, X : S \rhd X : S \to T}\,\mathit{morvar}$$

where we retain the restriction on signature inclusions: All signatures included into S must also be included into T.

Note that we can understand the signature S as a (dependent) record type, a morphism $\mu : S \to T$ as a record value of type S visible in the signature T, and an identifier $S.c^{\mu}$ as the projection out of the record type S at the field c applied to μ. Then X is simply a variable of record type, and abstraction over morphism variables is straightforward:

$$\text{Type families} \quad A \quad ::= \quad \{X : S\}A \qquad \text{Terms} \quad t \quad ::= \quad [X : S]\,t \mid t\;\mu$$

and (omitting the obvious Π-rule and the rules for β and η-conversion)

$$\frac{G;\, \Gamma,\, X : S \rhd_T t : A}{G;\, \Gamma \rhd_T [X : S]\,t : \{X : S\}A}\,\mathit{morlam} \qquad \frac{G;\, \Gamma \rhd_T f : \{X : S\}A \quad G;\, \Gamma \rhd \mu : S \to T}{G; \Gamma \rhd_T f\;\mu : A[X/\mu]}\,\mathit{morapp}$$

Here t and A may contain occurrences of the morphism variable X. In particular X may occur as a morphism argument to some expression, e.g., $g\,X$, or in an identifier $S.c^X$, which we write as $X.c$ in accordance with our notation for structures.

A crucial feature of the LF module system is that it is conservative: Modular signatures can be elaborated into non-modular ones (essentially by replacing every structure declaration with the induced constant declarations). We want to elaborate morphism variables similarly.

To elaborate $X : S$, we can assume that all structures in S have already been elaborated and that all defined symbols have been removed by expanding definitions, i.e., (up to reordering) S is of the form $\mathtt{sig}\, S = \{\mathtt{include}\, R_1, \dots, \mathtt{include}\, R_m,\ c_1 : B_1, \dots, c_n : B_n\}$. Then $[X : S]\, t$ in a signature T is elaborated to $[x_1 : B'_1] \dots [x_n : B'_n]\, t'$ where for expressions E over S, we obtain E' by replacing every occurrence of X with the morphism $\{c_1 := x_1, \dots, c_n := x_n : S \to T\}$. (In particular, after using morphism application, the identifiers $S.c_i^X$ simply become x_i.) $\{X : S\}A$ is elaborated accordingly. Finally, $t\ \mu$ is elaborated to $t\ \mu(S.c_1)\ \dots\ \mu(S.c_n)$.

This extended module system is not conservative over LF: S may contain type declarations, but LF does not permit abstraction over type variables. But we obtain conservativity if we make the following additional restriction: Contexts $\Gamma, X : S$ are only well-formed if all type family symbols $R.a^\mu$ available in S are included from other signatures, i.e., $\mu = incl\ \dots\ incl$. Conversely, neither S nor any signature that S instantiates may contain type family declarations.

This restriction seems ad hoc but is in fact quite natural. Assume we have LF signatures $\ulcorner L \urcorner$ and $\ulcorner T \urcorner$ that represent an object logic L and a theory T of L. Then, typically, $\ulcorner L \urcorner$ contains type declarations for the syntactic categories and judgments of L and constant declarations for the logical symbols and inference rules; $\ulcorner T \urcorner$ includes $\ulcorner L \urcorner$ and adds constant declarations for the non-logical symbols (sorts, functions, predicates, etc.) and axioms. Thus, our extension lets us abstract over morphisms out of theories but not over morphisms out of object logics. And the former are exactly the morphisms that we are interested in because morphisms out of $\ulcorner T \urcorner$ can be used to represent models or implementations of T. In particular, below, T will be an axiomatic type class and morphisms out of $\ulcorner T \urcorner$ will be type class instances.

4 Representing Isabelle in LF

The representation of Isabelle in LF proceeds in two steps. In a first step, we declare an LF signature *Pure* for the inner syntax of Isabelle. This syntax declares symbols for all primitives that can occur (explicitly or implicitly) in *Pure* expressions. In a second step, every Isabelle expression E is represented as an LF expression $\ulcorner E \urcorner$. Finally we have to justify the adequacy of the encoding.

For the *inner syntax*, the LF signature *Pure* is given in Fig. 4. This is a straightforward intrinsically typed encoding of higher-order logic in LF. Pure types τ are encoded as LF-terms $\ulcorner \tau \urcorner : tp$ and Pure terms $t :: \tau$ as LF-terms $\ulcorner t \urcorner : tm\, \ulcorner \tau \urcorner$. Using higher-order abstract syntax, the LF function space $A \to B$ with λ-abstraction $[x : A]\, t$ and application $f\ t$ is distinguished from the encoding $tm\ (\ulcorner \sigma \urcorner \Rightarrow \ulcorner \tau \urcorner)$ of the Isabelle function space with application $\ulcorner f \urcorner @ \ulcorner t \urcorner$ and λ-abstraction $\lambda([x : tm\, \ulcorner \tau \urcorner]\, \ulcorner t \urcorner)$. Pure propositions φ are encoded as LF-terms $\ulcorner \varphi \urcorner : tm\ prop$, and derivations of φ as LF-terms of type $\vdash\ \ulcorner \varphi \urcorner$. Where possible, we use the same symbol names in LF as in Isabelle, and we can also mimic most of the Isabelle operator fixities and precedences.

The signature *Pure* only encodes how composed Pure expressions are formed from the atomic ones. The atomic expressions – variables and constants etc. – are added when encoding the outer syntax as LF declarations. For the *non-modular declarations*, this is straightforward, an overview is given in the following table:

$$
\begin{array}{llll}
\multicolumn{4}{l}{\mathtt{sig}\ \mathit{Pure} = \{} \\
\quad tp & : & \mathtt{type}. & \\
\quad \Rightarrow & : & tp \to tp \to tp. & \mathtt{infix}\ \text{right}\ 0 \Rightarrow. \\
\quad tm & : & tp \to \mathtt{type}. & \mathtt{prefix}\ 0\ tm. \\
\quad \lambda & : & (tm\,A \to tm\,B) \to tm\,(A \Rightarrow B). & \\
\quad @ & : & tm\,(A \Rightarrow B) \to tm\,A \to tm\,B. & \mathtt{infix}\ \text{left}\ 1000\ @. \\
 & & & \\
\quad prop & : & tp. & \\
\quad \bigwedge & : & (tm\,A \to tm\,prop) \to tm\,prop. & \\
\quad \Longrightarrow & : & tm\,prop \to tm\,prop \to tm\,prop. & \mathtt{infix}\ \text{right}\ 1 \Longrightarrow. \\
\quad \equiv & : & tm\,A \to tm\,A \to tm\,prop. & \mathtt{infix}\ \text{none}\ 2 \equiv. \\
 & & & \\
\quad \vdash & : & tm\,prop \to \mathtt{type}. & \mathtt{prefix}\ 0 \vdash. \\
\quad \bigwedge\mathrm{I} & : & (x : tm\,A \vdash (B\,x)) \ \to \vdash \bigwedge([x]B\,x). & \\
\quad \bigwedge\mathrm{E} & : & \vdash \bigwedge([x]B\,x) \ \to \ \{x : tm\,A\} \vdash (B\,x). & \\
\quad \Longrightarrow\mathrm{I} & : & (\vdash A \ \to \vdash B) \ \to \vdash A \Longrightarrow B. & \\
\quad \Longrightarrow\mathrm{E} & : & \vdash A \Longrightarrow B \ \to \vdash A \ \to \vdash B. & \\
\quad \mathrm{refl} & : & \vdash X \equiv X. & \\
\quad \mathrm{subs} & : & \{F : tm\,A \ \to \ tm\,B\} \vdash X \equiv Y \ \to \vdash F\,X \equiv F\,Y. & \\
\quad \mathrm{exten} & : & \{x : tm\,A\} \vdash (F\,x) \equiv (G\,x) \ \to \vdash \lambda F \equiv \lambda G. & \\
\quad \mathrm{beta} & : & \vdash (\lambda[x : tm\,A]F\,x)\ @\ X \equiv F\,X. & \\
\quad \mathrm{eta} & : & \vdash \lambda\ ([x : tm\,A]F\ @\ x) \equiv F. & \\
\quad \mathtt{sig}\ \mathit{Type} & = & \{this : tp.\}. & \\
\multicolumn{4}{l}{\}.}
\end{array}
$$

Figure 2: LF Signature for Isabelle

Expression	Isabelle	LF
base type, type operator	$(\alpha_1,\ldots,\alpha_n)\ t$	$t : tp \to \ldots \to tp \to tp$
type variable	α	$\alpha : tp$
constant	$c :: \tau$	$c : tm\,\ulcorner\tau\urcorner$
variable	$x :: \tau$	$x : tm\,\ulcorner\tau\urcorner$
assumption/axiom/definition	$a : \varphi$	$a : \vdash \ulcorner\varphi\urcorner$
theorem	$a : \varphi\ P$	$a : \vdash \ulcorner\varphi\urcorner = \ulcorner P\urcorner$

The main novelty of our encoding is to also cover the *modular declarations*. The basic idea is to represent all high-level scoping concepts as signatures and all relations between them as signature morphisms as in the following table:

Isabelle	LF
theory, locale, type class	signature
theory import	morphism (inclusion)
locale import, type class import	morphism (structure)
sublocale, interpretation, type class instantiation	morphism (view)
instance of type class C	morphism with domain C

In the following, we give the important cases of the mapping $\ulcorner - \urcorner$ from Isabelle to LF by induction on the Isabelle syntax. We occasionally use color to distinguish the meta-level symbols (such as $=$) from Isabelle and Twelf syntax such as $=$.

Theories Isabelle theories and theory imports are encoded directly as LF-signatures and signature inclusions. The only subtlety is that the LF encodings additionally include our *Pure* signature.

$$\ulcorner \texttt{theory}\ T\ \texttt{imports}\ T_1,\ldots,T_n\ \texttt{begin}\ \Sigma\ \texttt{end} \urcorner =$$
$$\texttt{sig}\ T = \{\ \texttt{include}\ \mathit{Pure}.\ \texttt{include}\ T_1.\ \ldots \texttt{include}\ T_n.\ \ulcorner\Sigma\urcorner\}.$$

where the body Σ of the theory is translated component-wise as described by the respective cases below.

Type Classes The basic idea of the representation of Isabelle type classes in LF is as follows: An Isabelle type class C is represented as an LF signature C that contains all the declarations of C *and* a field $\mathit{this} : tp$. All occurrences in C of the single permitted type variable $\alpha :: C$ are translated to *this* such that *this* represents the type that is an instance of C.

This means that α is not considered as a type variable but as a type declaration that is present in the type class. This change of perspective is essential to obtain an elegant encoding of type classes.

In particular, the subsignature *Type* of *Pure* represents the type class of all types. Morphisms with domain *Type* are simply terms of type tp, i.e., types.

The central invariant of the representation is this: An Isabelle type class instance $\tau :: C$ is represented as an LF morphism $\ulcorner\tau :: C\urcorner$ from C into the current LF signature that maps the field *this* to $\ulcorner\tau\urcorner$ and all operations of C to the encoding of their definitions at τ. Thus, in particular, $\ulcorner\tau :: C\urcorner(C.\mathit{this}) = \ulcorner\tau\urcorner$.

Example 2 (Continued). The first type class from Ex. 1 is represented in LF as follows:

$$\texttt{sig}\ \mathit{order}\ =\ \{\mathit{this} : tp.\ \leq : tm(\mathit{this} \Rightarrow \mathit{this} \Rightarrow \mathit{prop})\}$$

In general, we represent type classes as follows:

$$\ulcorner\texttt{class}\ C = C_1\ \ldots\ C_n + \Sigma\urcorner = \texttt{sig}\ C = \{\mathit{this} : tp.\ I_1.\ \ldots\ I_n.\ \ulcorner\Sigma\urcorner\}.$$

where I_i abbreviates $\texttt{struct}\ \mathit{ins}_i\ : C_i\ = \{\mathit{this} := \mathit{this}\ \rho_i\}$ for some fresh names ins_i. Since one *this* is imported from each superclass C_i, they must be shared using the instantiations $\mathit{this} := \mathit{this}$. ρ_i contains one structure sharing declaration for each type class imported by I_i that has already been imported by $I_1,\ldots,I_{i-1}$.

Example 3 (Continued). The second type class from Ex. 1 is represented in LF as follows:

$$\texttt{sig}\ \mathit{semlat}\ =\ \{\mathit{this} : tp.\ \texttt{struct}\ o\ : \mathit{order}\ = \{\mathit{this} := \mathit{this}\}.\ \sqcap : tm(\mathit{this} \Rightarrow \mathit{this} \Rightarrow \mathit{this})\}$$

A type class *instantiation*

$$\texttt{instantiation}\ t :: (C_1,\ldots,C_n)C\ \texttt{begin}\ \Sigma\ \pi\ \texttt{end}$$

is represented as an LF functor taking instances of the C_i and returning an instance of C. We represent such a functor as a signature

$$\texttt{sig}\ \nu = \{\texttt{struct}\ \alpha_1\ : C_1 \ldots \texttt{struct}\ \alpha_n\ : C_n\}.$$

collecting the input and a view

$$\texttt{view}\ \nu' : C \to \nu = \{\mathit{this} := t\ \alpha_1.\mathit{this}\ \ldots\ \alpha_n.\mathit{this}\ \ulcorner\Sigma\urcorner\ulcorner\pi\urcorner\}.$$

describing the output. ν' must map the field tp of C to the type that is an instance of C. This type is obtained by applying t to the argument types that are instances of the C_i. In Isabelle, this is $t\ \alpha_1\ \ldots\ \alpha_n$; in LF, each α_i is a structure of C_i, thus we use the induced constants $\alpha_i.this$.

Here $\ulcorner\Sigma\urcorner$ gives instantiations that map every constant of C to its definition in terms of the α_i. Similarly, $\ulcorner\pi\urcorner$ maps every axiom of C to its proof. Note how – in accordance with the Curry-Howard representation of proofs as terms – the discharging of proof obligations is just a special case of instantiating a constant.

Now assume type class instances $\tau_i :: C_i$ encoded as morphisms $\ulcorner\tau_i :: C_i\urcorner : C_i \to S$ (where S is the current signature). The encoding $\ulcorner(\tau_1,\ldots,\tau_n)t :: C\urcorner : C \to S$ is obtained as the composition

$$\nu'\ \{\mathtt{struct}\ \alpha_1 := \ulcorner\tau_1 :: C_1\urcorner \ldots \mathtt{struct}\ \alpha_n := \ulcorner\tau_n :: C_n\urcorner : \nu \to S\}.$$

Clearly this is a morphism from C to S; we need to show that indeed

$$\ulcorner(\tau_1,\ldots,\tau_n)t :: C\urcorner(C.this) = \ulcorner t\ \tau_1\ \ldots\ \tau_n\urcorner.$$

This holds because

$$\begin{aligned}
&\ulcorner(\tau_1,\ldots,\tau_n)t :: C\urcorner(C.this) = \{\ldots\mathtt{struct}\ \alpha_i := \ulcorner\tau_i :: C_i\urcorner\ldots\}(\nu'(C.this))\\
&\{\ldots\mathtt{struct}\ \alpha_i := \ulcorner\tau_i :: C_i\urcorner\ldots\}(t\ \alpha_1.this\ \ldots\ \alpha_n.this) =\\
&t\ \ulcorner\tau_1 :: C_1\urcorner(C_1.this)\ \ldots\ \ulcorner\tau_n :: C_n\urcorner(C_n.this) = t\ \ulcorner\tau_1\urcorner\ \ldots\ \ulcorner\tau_n\urcorner = \ulcorner t\ \tau_1\ \ldots\ \tau_n\urcorner
\end{aligned}$$

We have the general result that the Isabelle subclass relation $C \subseteq D$ holds iff there is an LF morphism $i : D \to C$. Then if the type class instance $\tau :: C$ (occurring in some theory or locale S) is represented as a morphism $\ulcorner\tau :: C\urcorner : C \to S$, the type class instance $\tau :: D$ is represented as $\ulcorner\tau :: D\urcorner = i\ \ulcorner\tau :: C\urcorner$. Isabelle has the limitation that there can be at most one way how C is a subclass of D, which has the advantage that i is unique and can be dropped from the notation. In LF, we have to make it explicit.

Example 4 (Continued). The trivial subclass relation $order \subseteq Type$ is represented by the morphism $i = \{this := this : Type \to order\}$. The subclass relation $semlat \subseteq order$ is represented by the morphism $semlat.o$. Finally, the morphism $i\ semlat.o\ =\ \{this := this : Type \to semlat\}$ represents $semlat \subseteq Type$.

Locales Similarly to type classes, Isabelle *locales* are encoded as subsignatures: For example,

$$\mathtt{locale}\ loc = ins_1 : loc_1\ \mathtt{where}\ \sigma_1\ \mathtt{for}\ \Sigma + \Sigma'$$

is encoded as the LF signature

$$\mathtt{sig}\ loc = \{\Theta\ \ulcorner\Sigma\urcorner\ \mathtt{struct}\ ins_1\ : loc_1\ = \{\ulcorner\sigma_1\urcorner\}.\ \ulcorner\Sigma'\urcorner\}.$$

Here Θ contains type declarations $\alpha : tp$ for the free type variables of the locale. Those are the free type variables that occur in the declarations of Σ and Σ'. These correspond to the single declaration $this : tp$ in type classes. This encoding of type variables may be surprising because free type variables correspond to universal types whereas the declarations in Θ correspond to existential types. We hold that our LF-encoding precisely captures the intended meaning of locales, whereas the definition of locales within Isabelle prefers universal type variables in order to be compatible with the underlying type theory.

If a locale inherits from more than one locale, the encoding is defined correspondingly using one structure $\mathtt{struct}\ ins_i\ : loc_i\ = \{\vartheta_i\ \ulcorner\sigma_i\urcorner\}$ for each locale instance $ins_i : loc_i\ \mathtt{where}\ \sigma_i$. Here ϑ_i contains the instantiations for free type variables of loc_i that are induced by σ_i and inferred by Isabelle. Furthermore, some additional sharing declarations become necessary due to a subtlety in the semantics of Isabelle locales: If a locale inherits two equal instances (same locale, same instantiations), they are implicitly identified. But in LF different structures are always distinguished unless shared explicitly. Therefore, we have to add to $\ulcorner\sigma_i\urcorner$ one sharing declaration $\texttt{struct}\ ins := ins'$ for each instance ins present in loc_i that is equal to one already imported by one of $ins_1, \ldots, ins_{i-1}$.

Example 5 (Continued). The locale from Ex. 1 is represented in LF as follows:

$$\begin{array}{l}\texttt{sig}\ lat = \{\\ \quad \texttt{struct}\ inf\ : semlat.\\ \quad \texttt{struct}\ sup\ : semlat\ = \{this := inf.this.\ \leq := \lambda[x]\lambda[y]\ \mathrm{inf}.\leq\ @\,y@\,x\}.\\ \}\end{array}$$

Note how the instantiation for $\leq$ induces an instantiation for the type *this*. In other words, the ϑ mentioned above is $this := inf.this$.

Sublocale declarations are encoded as views from the super- to the sublocale. Thus, the declaration

$$\texttt{sublocale}\ loc' < loc\ \texttt{where}\ \sigma\ \pi$$

is encoded as (for some fresh name ν):

$$\texttt{view}\ \nu : loc \to loc' = \{\vartheta\ \ulcorner\sigma\urcorner\ulcorner\pi\urcorner\}.$$

Here ϑ contains the instantiations of the free type variables (see Θ above), which are inferred by Isabelle based on the instantiations in σ.

Locale *interpretations* are interpreted in the same way except that the codomain is the current LF signature (which encodes the Isabelle theory containing the locale interpretation) instead of the sublocale.

As for type classes, we have the general result that loc is a sublocale of loc' iff there is an LF signature morphism from loc to loc'. Accordingly, loc can be interpreted in the theory T iff there is a morphism from loc to T. For example, loc is a sublocale of loc_1 from above via the composed morphism $\nu\ loc.ins_1$. Contrary to type classes, there may be several different sublocale relationships between two locales. In LF these are distinguished elegantly as different morphisms between the locales.

Example 6 (Continued). *lat* is a sublocale of *semlat* in two different ways represented by the LF morphisms $lat.inf$ and $lat.sup$. These are trivial sublocale relations induced by inheritance.

Constant Declarations Finally we have to represent those aspects of the non-modular declarations that are affected by type classes. We will only consider the case of constants. Definitions, axioms, and theorems are represented accordingly. The central idea is that free type variables constrained by type classes are represented using λ abstraction for morphism variables.

An Isabelle constant $c :: \tau$ with free *type variables* $\alpha_i :: C_i$ is represented as the LF-constant taking morphism arguments:

$$c : \{\alpha_1 : C_1\} \ldots \{\alpha_n : C_n\}\, tm\, \ulcorner\tau\urcorner.$$

Here in $\ulcorner\tau\urcorner$ every occurrence of the morphism variable α_i is represented as $\alpha_i.this$.

Whenever c is used with inferred type arguments $\tau_i :: C_i$ in a composed expression, it is represented by application of c to morphisms:

$$\ulcorner c\urcorner = c\,\ulcorner\tau_1 :: C_1\urcorner \ldots \ulcorner\tau_n :: C_n\urcorner.$$

Actually, we cannot use the same identifier c in LF as in Isabelle: Instead, we must keep track how c came into scope. For example, if c was imported from some theory S, we must use $S.c$ in LF; if the current scope is a locale and c was imported from some other locale via an instance *ins*, we must use *ins*.c in LF; if c was moved into the current theory from a locale *loc* via an `interpretation` declaration which was encoded using the fresh name ν, we must use $loc.c^{\nu}$ in LF, and so on.

Types The representation of types was already indicated above, but we summarize it here for clarity. Type operator declarations $(\alpha_1,\ldots,\alpha_n)t$ are encoded as constants $t : tp \to \ldots \to tp \to tp$. And types occurring in expressions are encoded as

$$
\begin{array}{lcl}
\ulcorner \alpha :: C \urcorner & = & \alpha.this \\
\ulcorner t \urcorner & = & t \\
\ulcorner (\tau_1,\ldots,\tau_n)t \urcorner & = & t \ulcorner \tau_1 \urcorner \ulcorner \tau_n \urcorner \\
\ulcorner \tau_1 \Rightarrow \tau_n \urcorner & = & \ulcorner \tau_1 \urcorner \Rightarrow \ulcorner \tau_2 \urcorner \\
\ulcorner prop \urcorner & = & prop.
\end{array}
$$

Adequacy Before we state the adequacy, we need to clarify in what sense our representation is adequate. In Isabelle, locales and type classes are not primitive notions. Instead, they are internally elaborated into the underlying type theory. For example, all declarations in a locale or a type class are relativized and lifted to the top level. Thus, they are available elsewhere and not only within the locale.

While there are certainly situations when this is useful, here we care about the modular structure and the underlying type theory, but not about the elaboration of the former into the latter. Therefore, we do not want a representation in LF that adequately preserves the elaboration. In fact, if we wanted to preserve the elaboration, we could simply use Isabelle to eliminate all modular structure and represent the non-modular result using well-known representations of higher-order logic in LF.

Therefore, we have to forbid all Isabelle theories where names are used outside their scope. Let us call an Isabelle theory *simple* if all declared names are only used in their respective declaration scope – theory, locale, or type class – unless they were explicitly moved into a new scope using `imports`, `sublocale`, `interpretation`, or `instantiation` declarations, or using inheritance between type classes and locales.

Then we can summarize our representation with the following theorem:

Theorem 7. *If a simple sequence of Isabelle theories $T_1 \ \ldots \ T_n$ is well-formed (in the sense of Isabelle), then the LF signature graph Pure $\ulcorner T_1 \urcorner \ \ldots \ \ulcorner T_n \urcorner$ is well-formed (in the sense of LF extended with morphism variables).*

Proof. To show the adequacy for the encoding of the inner syntax is straightforward. A similar proof was given in [6].

The major lemmas for the outer syntax were already indicated in the text:

- For an Isabelle type class instance $\tau :: C$ used in theory or locale S and context Γ, we have $G;\Gamma \rhd \ulcorner \tau :: C \urcorner : C \to S$ and $\ulcorner \tau :: C \urcorner (C.this) = \ulcorner \tau \urcorner$.
- There is an Isabelle sublocale relation $loc' < loc$ via instantiations σ whenever the incomplete LF morphism $\{\ulcorner \sigma \urcorner \ \ldots : loc \to loc'\}$ can be completed (by instantiating the axioms of loc with proof terms over loc').

The main difficulty in the proofs is to show that at any point in the translated LF signatures exactly the right atomic expressions are in scope. This has to be verified by a difficult and tedious comparison of the Isabelle documentation with the semantics of the LF module system. In particular, in our simplified grammar for Isabelle, we have omitted the features that would break this result. These include in particular the features whose translation requires inventing and keeping track of fresh names, such as overloading and unqualified locale instantiation. □

5 Conclusion

We have presented a representation of Isabelle's module system in the LF module system. Previous logic encodings in LF have only covered non-modular languages (e.g., [6, 8, 15]), and ours is the first encoding of a modular logic. We also believe ours to be the first encoding of type classes or locale-like features in any logical framework.

The details of the translation are quite difficult, and a full formalization requires intricate knowledge of both systems. However, guided by the use of signatures and signature morphisms as the main primitives in the LF module system, we could give a relatively intuitive account of Isabelle's structuring mechanisms.

Our translation preserves modular structure; in particular the translation is compositional and the size of the output is linear in the size of the input. We are confident that our approach scales to other systems such as the type classes of Haskell or the functors of SML, and thus lets us study the modular properties of programming languages in logical frameworks. Moreover, we hold that the trade-off made in the LF module system between expressivity and simplicity makes it a promising starting point to investigate the movement of modular developments between systems.

In order to formulate the representation, we had to add abstraction over morphisms to the LF module system. This effectively gives LF a restricted version of dependent record types. This is similar to the use of contexts as dependent records as, e.g., in [17]. Contrary to, e.g., [3] and [12], the LF records may only occur in contravariant positions, which makes them a relatively simple conservative addition.

An integration of this feature into the Twelf implementation of LF remains future work. Similarly, the use of anonymous morphisms has not been implemented in Twelf yet. In both cases, the implementation is conceptually straightforward. However, since it would permit the use of morphisms in terms, types, and kinds, it would requires a closer integration of modular and core syntax in Twelf, which has so far been avoided deliberately. We will undertake the Twelf side of the implementation soon.

In any case, Twelf will hardly be a bottleneck. Any implementation of a translation from Isabelle to LF would have to be implemented from within Isabelle as it requires Isabelle's reconstruction of types and instantiations (let alone proof terms). However, Isabelle currently eliminates most aspects of modularity when checking a theory. For example, it is already difficult to export the local constants of a theory because the methods provided by Isabelle can only return all local, imported, or internally generated constants at once. The most promising albeit still very difficult approach seems to be to use a standalone parser for the Isabelle outer syntax and then fill in the gaps by calling the methods provided by Isabelle. Thus, even though this paper solves the logical questions how to translate from Isabelle to LF, the corresponding software engineering questions are non-trivial and remain open.

References

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

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

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

[4] J. Goguen and R. Burstall. Institutions: Abstract model theory for specification and programming. *Journal of the Association for Computing Machinery*, 39(1):95–146, 1992.

[5] F. Haftmann and M. Wenzel. Constructive Type Classes in Isabelle. In T. Altenkirch and C. McBride, editors, *TYPES conference*, pages 160–174. Springer, 2006.

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

[7] R. Harper and B. Pierce. Design Issues in Advanced Module Systems. In B. Pierce, editor, *Advanced Topics in Types and Programming Languages*. MIT Press, 2005.

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

[9] F. Kammüller, M. Wenzel, and L. Paulson. Locales – a Sectioning Concept for Isabelle. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Thery, editors, *Theorem Proving in Higher Order Logics*, pages 149–166. Springer, 1999.

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

[11] T. Nipkow. Structured Proofs in Isar/HOL. In H. Geuvers and F. Wiedijk, editors, *TYPES conference*, pages 259–278. Springer, 2002.

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

[13] L. Paulson. The Foundation of a Generic Theorem Prover. *Journal of Automated Reasoning*, 5(3):363–397, 1989.

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

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

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

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

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

[19] F. Rabe and C. Schürmann. A Practical Module System for LF. In *Proceedings of the Workshop on Logical Frameworks Meta-Theory and Practice (LFMTP)*, 2009.

[20] D. Sannella and M. Wirsing. A Kernel Language for Algebraic Specification and Implementation. In M. Karpinski, editor, *Fundamentals of Computation Theory*, pages 413–427. Springer, 1983.