A Formalized Set-Theoretical Semantics of Isabelle/HOL

Florian Rabe and Mihnea Iancu

Jacobs University Bremen

Abstract. Higher-order logic (HOL) and Zermelo-Fraenkel set theory (ZFC) are very different foundations of mathematics. The latter is preferred in mathematics, the former has been used successfully in machine-readable formalizations of mathematics and computer science. Isabelle is a logical framework offering a variety of advanced features such as a module system and axiomatic type classes. Its most advanced use is the Isabelle implementation of HOL.
We give a translation from Isabelle/HOL to ZFC set theory that formalizes its set-theoretical semantics. The whole translation is represented in the Edinburgh logical framework LF and includes representations of Isabelle, Isabelle/HOL, and ZFC in LF. The soundness of the semantics is guaranteed by the LF type theory. All representations are implemented in and verified by the Twelf system.

1 Introduction

Both type theory and (axiomatic) set theory were introduced in the beginning of the 20th century as a response to the inconsistencies discovered in naive set theory, the Grundlagenkrise of mathematics. While the majority of mathematicians favor set theory, the advent of computer science and the desire to recreate mathematics in machine-understandable way has fostered the development of type theories. This is because type theories favor algorithmic definitions and decidable notions, and these prove indispensable when formalizing major parts of mathematics in a computer ([6,10]) or when implementing automated reasoning support for mathematics ([13,7,2]). Only few systems use set theoretical foundations such as Mizar ([24]) or Isabelle/ZF ([20]), and the latter is less popular than its sister Isabelle/HOL ([17]).

In this paper, we give a formal translation from the most popular type-theoretical foundation – higher-order logic (HOL, [4]) – to the most popular set-theoretical one – Zermelo-Fraenkel set theory ([26,5]). This translations yields a formal set-theoretical semantics of HOL interpreting types as sets, terms as elements, and theorems as theorems.

Among the versions of HOL, we pick the Isabelle/HOL implementation because Isabelle is arguably the most advanced language for large scale formalizations offering polymorphism, type classes and a strong module system as well as, e.g., structured proofs and notation support. Furthermore, Isabelle and Isabelle/HOL have been used extensively to formalize specification languages and software verification. Therefore, it is interesting to formalize the semantics of Isabelle itself.

A formal translation between two languages requires to represent both of them in a formal framework. We employ the Edinburgh logical framework LF and its Twelf

implementation ([11,21]) for two reasons. Its dependent type theory permits us to formalize ZFC set theory in a way that is very close to a mathematician's intuition. And its module system ([22]) is strong enough to represent Isabelle locales and type classes. Furthermore, the Twelf module system can verify the type-preservation and thus the soundness of the translation.

For a researcher on Isabelle and HOL, our main result is the representation of Isabelle/HOL and its set-theoretical semantics in LF. For a mathematician, this serves as a formal proof that the primitives of Isabelle/HOL can be seen as syntactic sugar to reason about ZFC set theory. And for a computer scientist familiar with type theory but not with the idiosyncrasies of Isabelle and HOL, it provides an entry point that is complementary to the existing documentation.

This paper is organized as follows. In Sect. 2, we will repeat the basics of Isabelle and LF to make the paper self-contained. In Sect. 3, we design a slight extension to the LF module system needed for the representation of Isabelle type classes. In Sect. 4, 5, and 6 we describe the formalizations of Isabelle/HOL, ZFC, and the translation of the former into the latter in LF. The complete Twelf source files are available as [15].

2 Preliminaries

2.1 Isabelle and HOL

Isabelle Isabelle is a logical framework and generic LCF-style interactive theorem prover based on polymorphic higher-order logic ([18,19]). It 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`.

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* declarations provide a second module system. 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 $(\alpha_1, \ldots, \alpha_n)\ t = \tau$.

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

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* | $(type, \ldots, type)$ *name* | *type* $\Rightarrow$ *type* | $prop$
term	::= *var* | *name* | *name term** | $\lambda(var :: type)^*.term$
prop	::= *prop* $\Longrightarrow$ *prop* | $\bigwedge(var :: type)^*.prop$ | *term* $\equiv$ *term*
proof	::= a proof term
name, *var*	::= identifier

Fig. 1. Simplified Isabelle Grammar

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

the new locale loc inherits via n names instances: Instance ins_i inherits from the locale loc_i via the list of parameter instantiations σ_i. The declarations in Σ logically precede the instances, i.e., are available for the parameter instantiations. Thus, the list of declarations of loc consists of the declarations of Σ, a copy of the declarations of each loc_i translated by σ_i, and finally the declarations of 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'. The names ins_i serve as qualifiers to resolve name clashes if two declarations of the same name are inherited from different locales.

Secondly, a `sublocale` declaration $\texttt{sublocale}\ loc' < loc\ \texttt{where}\ \sigma\ \pi$ 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 ([9]) 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 = $C_1 \dots C_n + \Sigma$ 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 $\alpha :: C$. Thus, Σ provides polymorphic operations $c_1, \dots, c_n$ on the parametric type α and axioms about them. An instance of a type class is a tuple $(\tau, c_1\_def, \dots, c_2\_def)$ 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 dropped from the notation; then the type class can be seen as a unary predicate on types τ.

Type class instantiations are of the form `instantiation` $t :: (C_1, \dots, C_n)C$ `begin` Σ π `end` where t is an n-ary type operator. Σ contains the definitions for the operations of C at the type $(\tau_1, \dots, \tau_n)t$ in terms of the operations of the instances $\tau_i :: C_i$. 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 also $(\tau_1, \dots \tau_n)t :: C$. Note that this includes base types for $n = 0$.

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. Finally, we omit the Isar proof language and simply use proof terms from Pure's natural deduction calculus.

Isabelle/HOL HOL ([17]) is the most often used and farthest developed theory of Isabelle. It formalizes classical higher-order logic with shallow polymorphism, nonempty types, and choice operator ([4,8]).

Isabelle/HOL uses the same types and function space as Isabelle. However, it introduces a type $bool$ for HOL-propositions (i.e., booleans since HOL is classical) that is different from the type $prop$ of Isabelle-propositions. The coercion $Trueprop : bool \Rightarrow prop$ is used as the Isabelle truth judgment on HOL propositions. HOL declares primitive constants for implication, equality on all types, definite description operator $the\ x : \tau.P$ for types τ and predicates $P : \tau \Rightarrow bool$, and (in the theory Hibert_Choice) the indefinite description operator eps of the same type.

Furthermore, HOL declares a polymorphic constant $undefined$ of any type and (in the theory Nat) an infinite base type ind. We omit these in the following, but they are part of our encoding in [15]. Based on these primitives and their axioms, simply-typed set theory is developed by purely definitional means.

Isabelle/HOL is not only an Isabelle theory. The Isabelle system also provides special support for the HOL theory. Among those, we will only consider the most important one, namely Gordon/HOL-style type definitions using representing sets. A set A on the type τ is given by its characteristic function, i.e., $A : \tau \Rightarrow bool$. An Isabelle/HOL type definition is of the form $\texttt{typedef}\ (\alpha_1, \ldots, \alpha_n)\ t = A\ P$ where P and A contain the variables $\alpha_1, \ldots, \alpha_n$ and P proves that A is non-empty. If such a definition is in effect, t is an additional type that is axiomatized to be equal to the set A.

2.2 LF

LF ([11]) is a logical framework based on dependent type theory and the judgments-as-types methodology. It is related to Martin-Löf type theory ([16]) and the corner of the λ-cube ([1]) that extends simple type theory with dependent function types. We will work with the Twelf implementation of LF ([21]).

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 $t : A$. $\texttt{type}$ is the kind of types, and $A \to \texttt{type}$ 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 $A \to B$ 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 ([22]) is based on the notions of signatures and signature morphisms ([12]). Given two signatures $\texttt{sig}\ S\ =\ \{\Sigma\}$ and $\texttt{sig}\ T\ =\ \{\Sigma'\}$, a signature morphism from S to T is a type-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 to closed Σ-expressions. Signature morphisms preserve typing and kinding, i.e., if $\vdash_\Sigma E : F$, then $\vdash_{\Sigma'} \mu(E) : \mu(F)$.

The modular declarations are signatures and explicit morphisms called views. Signatures may be nested and may import other signatures via inclusions and structures. We will work with the following slightly simplified grammar:

$$
\begin{array}{lll}
\text{Signatures} & \Sigma & ::= \cdot \mid \Sigma,\ \texttt{sig}\ T\ =\ \{\Sigma\} \mid \Sigma,\ \texttt{view}\ v\ : S\ \to T\ = \{\sigma\} \\
& & \quad \mid\ \Sigma,\ \texttt{include}\ S \mid \Sigma,\ \texttt{struct}\ s\ : S\ = \{\sigma\} \\
& & \quad \mid\ \Sigma,\ c : A[= t] \mid \Sigma,\ a : K[= A] \\
\text{Instantiations} & \sigma & ::= \cdot \mid \sigma,\ c := t \mid \sigma,\ a := A \mid \sigma,\ \texttt{struct}\ s := \mu \\
\text{Kinds} & K & ::= \texttt{type} \mid A \to K \\
\text{Type families} & A & ::= S.a^\mu \mid A\ t \mid \{x : A\}\, A \\
\text{Terms} & t & ::= S.c^\mu \mid x \mid [x : A]\, t \mid t\ t \\
\text{Morphisms} & \mu & ::= (v \mid T.s \mid \{\sigma : S \to T\})^*
\end{array}
$$

We pick a syntactically awkward but conceptually elegant syntax for constants and type family symbols: The constant $S.c^\mu$ is available in signature T and has type $\mu(A)$ iff $c : A$ is a constant declared in signature S and μ is a signature morphism from S to T.

This reduces the difficult question which constants are in scope to defining well-formed morphisms.

Signature morphisms μ are compositions of atomic morphisms: $\mu\ \mu'$ is a morphism from R to T if μ and μ' are morphisms from R to S and S to T, respectively. The atomic morphisms are given below.

$\{\sigma : S \to T\}$ is an explicitly given *signature morphism*. If declared on toplevel, it is called a view. Because judgments are represented as types and proofs as terms, signature morphisms must map axioms and inference rules to proofs and derived rules. Thus, views have the flavor of a theorem expressing the judgment-preserving representation of one signature in another.

An *inclusion* `include` S in T literally includes S into T. The empty list is a signature morphism from S to T iff the pair (S, T) is in the transitive-reflexive closure of the `include` relation. This makes local and included constants available, which we abbreviate as $S.c$ or simply c. Similarly, within a subsignature, the empty list is a morphism from its supersignature.

A *structure* `struct` $s\ : S\ = \{\sigma\}$ declared in T enforces a signature morphism $T.s$ from S into T by copying all declarations of S into T. It has the same effect as declaring a constant $s.c\ :\ T.s(A)$ in T for every constant $c\ :\ A$ of S. In addition, σ may instantiate some symbols of S with expressions over T: If σ contains $c := t$, the constant $s.c$ is defined as t. Technically, $s.c$ is just an abbreviation for the constant $S.c^{T.s}$. The same holds for type family symbols a.

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 $S.r\ T.s$, which we abbreviate as $s.r$, from R to T and thus two copies of the constants of R. Structures may also instantiate whole structures at once: If T declares instead `struct` $r'\ : R\ = \{$`struct` $r := T.r'\}$, then the two instances $s.r$ and r' are shared.

In order to represent Isabelle type classes, we will add one feature to Twelf: morphism variables. We describe them below, but their productions are already given in gray.

3 Morphism Variables in LF

We add a feature to the LF module system that permits to abstract over morphism variables: For signatures S, we permit variables $X\ :\ S$ and use X as a morphism from S into the current signature. Thus, if $c\ :\ A$ is a constant declared in S, the constant $S.c^X$ with type $X(A)$ becomes well-formed in context $X\ :\ S$. Thus, we add the following productions to the grammar (where we omit the binding of morphism variables occurring in kinds for simplicity):

$$\begin{array}{lll} \text{Type families} & A & ::= \{X : S\}\, A \\ \text{Terms} & t & ::= [X : S]\, t \mid t\ \mu \\ \text{Morphisms} & \mu & ::= X \end{array}$$

This can also be seen as using S as a (dependent) record type and morphisms $\mu : S \to T$ as record values over the signature T. Then $S.c^\mu$ is just the projection out of the record type S at the field c applied to the record value μ.

The rules are the same as for the abstraction over term variables $x : A$. For a term $t : A : \texttt{type}$ in a free variable $X : S$, we have the abstraction $[X : S]\, t : \{X : S\}\, A : \texttt{type}$. And for terms $f : \{X : S\}\, A$ and a morphism μ from S into the current signature, we have the application $f\ \mu : A'$ where A' arises by substituting X in A with μ. Furthermore, we have the straightforward $\alpha\beta\eta$-conversions.

The extension is meant to be conservative, i.e., to be elaborated into the core system. If S is of the form $\{c_1 : B_1 \dots c_n : B_n\}$. Then $[X : S]\, t$ is elaborated to $[X.c_1 : X(B_1)] \dots [X.c_n : X(B_n)]\, t'$ where t' is like t but with every occurrence of ${S.c_i}^X$ replaced with the fresh variable $X.c_i$. Accordingly, we elaborate $\{X : S\}\, A.\ f\ \mu$ is elaborated to $f\ {S.c_1}^\mu\ \dots\ {S.c_n}^\mu$.

If S contains structures, we eliminate the structures first, using the elaboration given in [22]. Furthermore, if S contains defined constants, we expand all definitions and then drop the defined constants. Finally, to handle the case of declarations $\texttt{include}\ R$ in S, we make the following restriction: Abstraction over the variable $X : S$ within a signature T is only well-formed if T includes all signatures that are included into S. Furthermore, we define that X maps all included symbols to themselves; and similarly an application $f\ \mu$ is only well-formed if μ maps every included symbol to itself. Then, all included symbols of S can be skipped in the elaboration.

This extended module system is not conservative over LF if S contains type declarations because LF does not permit type variables. But we retain conservativity if we make the following additional restriction: Abstraction over a variable $X : S$ is only well-formed if all type declarations available in S are included from other signatures. In light of the above restriction on inclusion, we can show that the elaboration of abstractions over $X : S$ only introduces term variables. Thus, our extended module system is conservative.

The restriction to signatures with only included type declarations is quite natural. When LF is used as a logical framework, type declarations are usually only used at the meta-level to represent the syntactic categories and judgments of the object logic and to reason about object logic. LF signatures representing theories of the object logic usually only add term declarations.

4 Representing Isabelle/HOL 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. In this section we will only sketch the definition of $\ulcorner E \urcorner$. A full definition is given in the appendix.

For the *inner syntax*, the LF signature $Pure$ is given in Fig. 4. This is a rather typical 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 $\ulcorner t \urcorner : tm\ \ulcorner \tau \urcorner$. Note that contrary to the encoding

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

Fig. 2. LF Signature for Isabelle

of HOL in Isabelle, 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 $\ulcorner\varphi\urcorner : prop$, 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, and overview is given in the following table:

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

Isabelle *theories* and theory imports are encoded directly LF-signature and signature inclusions. The only subtlety is that the LF encodings additionally include our $Pure$ signature. Isabelle *locales* are encoded as subsignatures: A locale

$$\texttt{locale}\ loc = ins_1 : loc_1\ \texttt{where}\ \sigma\ \texttt{for}\ \Sigma + \Sigma'$$

is encoded as the LF signature

$$\texttt{sig}\ loc = \{\ulcorner \Sigma \urcorner\ \texttt{struct}\ ins_1\ : loc_1\ = \{\ulcorner \sigma \urcorner\}.\ulcorner \Sigma' \urcorner\}.$$

Locales inheriting from more than one locale are encoded correspondingly, but some sharing declarations become necessary.

Sublocale declarations

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

are encoded as view from the super- to the sublocale (for some fresh name ν):

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

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).

We have the general result that loc is a sublocale of loc' (loc can be interpreted in the theory T) iff there is an LF signature morphism from loc to loc' (from loc to T). For example, loc is a sublocale of loc_1 from above via the composed morphism $\nu\ loc.ins_1$. Note that there may be several different sublocale relationships between two locales, e.g., for $monoid < ring$ or $semilattice < lattice$. In LF these are distinguished elegantly as different morphisms between the locales.

The representation of Isabelle *type classes* in LF is non-trivial. The basic idea is that an Isabelle type class C is represented as an LF signature C that contains all the declarations of C *and* a field $this : tp$. 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. 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. 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.

Type class instantiations

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

are represented as LF functors taking morphisms from the C_i and returning a morphism from C. LF functors are themselves a derived notion represented as a signature

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

collecting the input of the functor and a signature morphism

$$\texttt{view}\ \nu' : C \to \nu = \{this := t\ \alpha_1(C_1.this)\ \ldots\ \alpha_n(C_n.this).\ulcorner\Sigma\urcorner\ulcorner\pi\urcorner\}.$$

showing how C can be realized in terms of the input. 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 the morphism application $\alpha_i(C_i.this)$ is needed.

Given type class instances $\tau_i :: C_i$ with encodings $\ulcorner\tau_i :: C_i\urcorner : C_i \to S$ (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 $\nu'\ \{\ldots\ \texttt{struct}\ \alpha_i := \ulcorner\tau_i :: C_i\urcorner \ldots\}$. Note that we indeed have

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

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

Finally, an Isabelle constant $c :: \tau$ with *type parameters* $\alpha_i :: C_i$ is represented as an LF-constant $c : \{\alpha_1 : C_1\} \ldots \{\alpha_n : C_n\}\, tm\ \ulcorner\tau\urcorner$. Here in $\ulcorner\tau\urcorner$ every occurrence of the morphism variable α_i is represented as $\alpha_i(C_i.this)$. If c occurs with inferred type arguments $\tau_i :: C_i$ in a composed expression, it is represented in LF as $\ulcorner c\urcorner = c\ \ulcorner\tau_1 :: C_1\urcorner\ \ldots\ \ulcorner\tau_n :: C_n\urcorner$. Definitions, axioms, and theorems with type parameters are represented accordingly.

Theorem 1. *A sequence of Isabelle theories $T_1\ \ldots\ T_n$ is well-formed (in the sense of Isabelle) iff the sequence of LF signatures $Pure\ \ulcorner T_1\urcorner\ \ldots\ \ulcorner T_n\urcorner$ 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 [11].

For the outer syntax, the only difficulty 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 specifications of Isabelle and Twelf. In particular, in our simplified grammar for Isabelle, we have omitted the features that would break this result such as overloading and unqualified locale instantiation, which can only be translated to Twelf by inventing and keeping track of fresh names.

HOL We can apply the above translation directly to obtain ⌜HOL⌝. The fragment arising from translating only the primitive declarations of HOL is given in the upper part of Fig. 4. For readability, we have added the auxiliary functions $\uparrow$, $\doteq'$, and $\longrightarrow$. These have the effect that, e.g., $A \longrightarrow B$ in Isabelle can be encoded as $A \longrightarrow B$ in LF, which abbreviates $\longrightarrow' Pure.@\, A\, Pure.@\, B$. We also use an `open` declaration to use some Pure symbols without qualification. (Note that the defined constants $true$, $false$, and disjunction $|$ are not expanded in the axiom of excluded middle.)

More importantly, we have to extend HOL with LF declarations that represent the HOL type definitions. These are given in the lower part of Fig. 4 where we omit the obvious definitions of $nonempty$. The central declaration here is $typedef$, which takes a set S on the type A and a proof that S is nonempty and returns a new type T. The remaining declarations represent the two new primitive constants Rep and Abs and three axioms that Isabelle/HOL generates for every well-formed type definition. The axioms make Rep and Abs mutually inverse functions between T and the subtype of A containing the elements of S. We refer to [25] for the details.

5 Representing Set Theory in LF

Now we give an LF signature for ZFC set theory ([26,5]). Using the dependent typing of LF, we can represent untyped set theory in a rather natural way and then introduce a typed language on top. Based on this typed language, we can finally formulate the necessary prerequisites to interpret Isabelle/HOL. Our LF signature for ZFC is developed in a strictly definitional way: After the initial primitives all further notions are added using Twelf definitions. We will only sketch the development of set theory here, which covers over 1000 lines of Twelf declarations, and refer to [15] for the Twelf sources.

Our axiomatization of *untyped set theory* is based on classical first-order logic with equality and the single sort set, and the natural deduction proof calculus. We use a type $prop$ for propositions, and a truth judgment $\vdash$: $prop \to$ `type`. To obtain ZFC, we add a binary predicate $\in$: $set \to setprop$ and the axioms of extensionality, (unordered) pairing, union, powerset, and infinity, as well as the axiom schemes of specification and replacement. Our axiomatization stays close to the ZFC axioms as they usually appear in the literature in using only a binary predicate for membership and no function symbols. This is in contrast to Mizar where primitive function symbols are used for singleton, unordered pair, and union ([24,23]), and to Isabelle/ZF where primitive function symbols are used for empty set, powerset, union, infinite set, and replacement ([20]).

Not using primitive function symbols means that there are no terms besides variables. Therefore, we add the (definite) description operator $\delta : \{F : set \to prop\} \vdash$

```
sig HOL = {
   include Pure open tp tm ⊢ ⇒ prop λ @ ⋀ ⟹ ≡.
   bool          : tp.
   trueprop      : tm bool ⇒ prop.
   ↑             : tm bool → tm prop
                 = [x] trueprop @ x.                          prefix 3 ↑.
   the           : tm (A ⇒ bool) ⇒ A.
   eps           : tm (A ⇒ bool) ⇒ A.
   ≐'            : tm A ⇒ A ⇒ bool.
   ≐             : tm A → tmA → tm bool
                 = [x][y] ≐' @ x @ y.                         infix left 50 ≐.
   ⟶'            : tm bool ⇒ bool ⇒ bool.
   ⟶             : tm bool → tm bool → tm bool
                 = [x][y] ⟶' @ x @ y.                         infix left 25 ⟶.
   refl          : ⊢↑ X ≐ X.
   subst         : ⊢↑ S ≐ T ⟶ P @ S ≐ P @ T.
   ext           : ⊢ (⋀[x : tm A] ↑ F @ x ≐ G @ x) ⟹↑ (λ[x] F @ x) ≐ (λ[x] G @ x).
   the_eq_trivial: ⊢↑ the @ (λ[x] x ≐ A) ≐ A.
   someI         : ⊢↑ (P @ X) ⟶↑ (P@(eps@P)).
   impI          : ⊢ (↑ P ⟹↑ Q) ⟹↑ (P ⟶ Q).
   mp            : ⊢↑ (P ⟶ Q) ⟹↑ P ⟹↑ Q.
   iff           : ⊢↑ (P ⟶ Q) ⟶ (Q ⟶ P) ⟶ (P ≐ Q).
   true_or_false : ⊢↑ (P ≐ true) | (P ≐ false).

   set           : tp → tp = [a]a ⇒ bool.
   nonempty      : (tm set A) → bool = ...
   typedef       : {s : tm set A} ⊢↑ nonempty s ⟶ tp.
   Rep           : tm (typedef S P) ⇒ A.
   Abs           : tm A ⇒ (typedef S A)P.
   Rep_thm       : {x : tm typedef S P} ⊢↑ (Rep @ x) ∈ S.
   Rep_inverse   : {x : tm typedef S P} ⊢↑ Abs @ (Rep @ x) ≐ x.
   Abs_inverse   : {y : tm A} ⊢ (↑ y ∈ S) ⟹ (↑ Rep @ (Abs @ y)) ≐ y.
}.
```

Fig. 3. LF signature for HOL with Type Definitions

$\exists^!([x]\,F\ x) \to set$, which takes a formula $F(x)$ with a free variable x and a proof of $\exists^! x.F(x)$ and returns a set (where the binder $\exists^!$ of unique existence is defined as an abbreviation). Because a proof is passed as an argument, the LF type system guarantees that δ encodes the accepted mathematical practice of giving a name for a uniquely determined object. δ is axiomatized using an axiom scheme $F(\delta\ F\ P)$ (from which we can derive proof irrelevance).

Untyped set theory is not able to interpret a typed theory directly because there is only one type set, but we want to interpret every type of Isabelle/HOL as a type over ZFC. Therefore, we develop a *typed set theory* within ZFC inspired by a similar treatment in [3]. First we introduce the type $neset$ of non-empty ZFC sets. Intuitively,

we can think of it as the dependent product $\Sigma_{a:set} ded\ a \neq \varnothing$, i.e., the type of pairs of a set and a proof of non-emptiness. Such pairs are introduced using $elem : \Pi_{a:set} ded\ a \neq \varnothing \to neset$. Then, we use $Elem\ a : \texttt{type}$ to represent the type of elements of the non-empty set a, which we can think of as the product $\Sigma_{e:set} ded\ e \in a$. Again we use $elem : \Pi_{e:set} ded\ e \in a \to Elem\ a$ as the introductory form for such pairs.

Base on this, we can introduce a variety of operations for typed set theory as abbreviations, in particular:

$$\begin{array}{ll} func & : set \to set \to set = \ldots \\ lambda & : (Elem\ A \to\ Elem\ B) \to Elem(func\ A\ B) = \ldots \\ app & : Elem(func\ A\ B) \to Elem\ A \to Elem\ B = \ldots \\ forall & : (Elem\ A \to prop) \to prop = \ldots \\ filter & : (Elem\ A \to prop) \to set = \ldots \\ eq & : Elem\ A \to Elem\ A \to prop = \ldots \\ ifte & : \{F : o\}(\vdash\ F \to Elem\ A) \to (\vdash\ \neg\ F \to Elem\ A) \to Elem\ A = \ldots \end{array}$$

$func\ A\ B$ is the set of functions from A to B, and $lambda[f : Elem\ A]$ and $app\ f\ x$ formalize $\{(x, f(x)) : x \in A\}$ and "the y such that $(x, y) \in f$". $filter[x : Elem\ A]\ F\ x$ formalize $\forall x \in A.F(x)$ and $\{x \in A | F(x)\}$ respectively. $eq\ a\ b$ is equality of elements of the same "type". And $ifte\ F\ X\ Y$ formalizes "if F then X else Y". Note that X and Y take proofs of F and $\neg F$, respectively, as arguments. This permits to use an object X (Y) that only exists if F is true (false). Note that the LF type system guarantees that $ifte\ F\ X\ Y$ exists without the need to assume a default or undefined value, which would alter the ZFC set theory.

A set-theoretical semantics of HOL was given [8] using a universe of non-empty sets, which serve as the interpretation of the monomorphic types, that is closed under subsets, products, powersets, and infinity. HOL models consist of a universe with a choice function for it and interpretations of all declared constants and types. We could axiomatize such models in some other logical formalism and then prove that they can interpret HOL. We did the latter for first-order logic in [14], and it would be easy to do the same for HOL. However, contrary to first-order logic (where the universe is simply a set), both giving a universe for HOL and proving that it satisfies the needed properties is quite cumbersome in practice.

Therefore, we go a slightly different way here and treat set theory itself as the semantics of HOL, i.e., we use the (proper) class $neset$ of non-empty sets as the universe and assume a global choice function $choice : \{a : neset\}\ Elem\ a$. Clearly, every HOL model can be embedded into this semantics. Technically, the use of a global choice function makes our set theory stronger than ZFC, but if a specific universe and a choice function for it are given, it is easy to do without $choice$.

6 Interpreting Isabelle/HOL in Set Theory

Let ZFC be our LF signature for ZFC. Then we can give the semantics of Isabelle and Isabelle/HOL as two LF signature morphisms from $Pure$ and HOL into ZFC. The basic idea of the view is to interpret Isabelle types as non-empty sets and Isabelle

propositions as booleans. Then the interpretation of $\Rightarrow$, λ, @, $\Longrightarrow$, $\equiv$, and $\bigwedge$ is straightforward. Finally, $\vdash\ F$ is interpreted as the ZFC proposition that F is interpreted as 1, and all proof rules of $Pure$ are interpreted as corresponding derived rules.

The listing on the right gives a fragment of the resulting view, and we refer to [15] for the full encoding. We use *bool* to abbreviate the LF-encoding of the set $\{0,1\}$ with $0 = \varnothing$ and $1 = \{0\}$. It would be more elegant to interpret Isabelle propositions as propositions about set theory, i.e., $prop := prop$. However, that is impossible because $prop$ is a regular type in Isabelle and thus must be interpreted as a set. Instead of the booleans, any Heyting algebra could be chosen.

$$
\begin{array}{l}
\texttt{view}\ PureSem : Pure \to ZFC = \{ \\
\quad tp \quad := neset. \\
\quad \Rightarrow \quad := func. \\
\quad tm \quad := elem. \\
\quad \lambda \quad := lambda. \\
\quad @ \quad := app. \\
\quad prop := bool. \\
\quad \bigwedge \quad := \forall. \\
\quad \Longrightarrow \ := \Rightarrow. \\
\quad \equiv \quad := eq. \\
\quad \vdash \quad := [x] \vdash\ eq\ x\ 1. \\
\quad \vdots \\
\}.
\end{array}
$$

Similarly, we obtain a view $HOLSem$ from HOL to ZFC. Again we only give a fragment. *bool* is also mapped to the booleans, which means that $trueprop$ is simply the identity. The description operators the and eps are interpreted using $ifte$ and $choice$. Here the then-branch has to use $elem$ to construct an element of $neset$, i.e., a non-empty set, and the second argument of $elem$ must be a proof that this set is non-empty. In both cases, this proof must use the assumption p that the condition of the $ifte$-split is true. In both cases, we omit some bookkeeping proof steps and elide the projections from $Elem\ A$ to set; in the former case, we use P to abbreviate the proof of $\exists^! x.F(x) \Rightarrow \exists x.F(x)$.

Finally, $typedef\ f\ p$ is interpreted as $filter\ f$ and then using the proof p of the non-emptiness of $filter\ f$ to obtain an element of $neset$. Thus, Rep is mapped to the inclusion function from $filter\ f$ to A, and Abs to one of its inverses. Then their three axioms are mapped to the corresponding proofs.

$$
\begin{array}{l}
\texttt{view}\ HOLSem : HOL \to ZFC = \{ \\
\quad bool \quad := bool. \\
\quad trueprop := [x]\, x. \\
\quad the \quad := [f : Elem\ (func\ A\ bool)]\, ifte\ (\exists^! [x]\, x \in (filter\ f)) \\
\qquad\qquad ([p]\, (choice\ (elem\ (filter\ f)(\Rightarrow_{Elim}\ P\ p)))) \\
\qquad\qquad ([p]\, choice\ A). \\
\quad the \quad := [f : Elem\ (func\ A\ bool)]\, ifte\ (nonempty\ (filter\ f)) \\
\qquad\qquad ([p]\, (choice\ (elem\ (filter\ f)\ p))) \\
\qquad\qquad ([p]\, choice\ A). \\
\quad \doteq' \quad := Eq. \\
\quad \vdots \\
\quad typedef \ := [A]\, [f : Elem\ (func\ A\ bool)]\, [p]\, elem\ (filter\ f)\ p. \\
\quad \vdots \\
\}.
\end{array}
$$

Theorem 2. *For every closed term $t : \tau$ over Isabelle/HOL, we obtain its semantics in ZFC as the term $HOLSem(\ulcorner t \urcorner) : Elem\ HOLSem(\ulcorner \tau \urcorner)$.*
If a locale interpretation of loc in Isabelle/HOL is represented as a view $\nu : loc \to HOL$, then the morphism $\nu\ HOLSem : loc \to ZFC$ is a ZFC-model of loc.

Proof. Both results follow from the type preservation of views.

7 Conclusion

We have represented both Isabelle/HOL type theory and the ZFC set theory in the logical framework LF. Both encodings are new and quite difficult.

The representation of Isabelle/HOL covers a large fragment of Isabelle. In particular, it includes locales and type classes so that the modularity of Isabelle theories is preserved in their LF representation. This is especially valuable as the LF module system is significantly simpler than that of Isabelle and thus more amenable to utilize for system interoperability. The representation of HOL covers all primitives of HOL except for the natural numbers, which can be added easily. In particular, it includes the most distinctive idiosyncrasies of HOL: the two description operators and type definitions.

The encoding of ZFC provides a new formalization of set theory. While not as far developed as existing formalizations, it is interesting because its choice of primitives is very close to textbook mathematics.

Finally, the translation of Isabelle/HOL and ZFC is unique in that it provides a completely formalized and verified translation between two very different foundations of mathematics.

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. Y. Bertot and P. Castéran. *Coq'Art: The Calculus of Inductive Constructions*. Springer, 2004.
3. C. Brown. Combining Type Theory and Untyped Set Theory. In U. Furbach and N. Shankar, editors, *International Joint Conference on Automated Reasoning*, pages 205–219. Springer, 2006.
4. A. Church. A Formulation of the Simple Theory of Types. *Journal of Symbolic Logic*, 5(1):56–68, 1940.
5. A. Fraenkel. The notion of 'definite' and the independence of the axiom of choice. 1922.
6. G. Gonthier. A computer-checked proof of the four colour theorem, 2006. `http://research.microsoft.com/~gonthier/`.
7. M. Gordon. HOL: A Proof Generating System for Higher-Order Logic. In G. Birtwistle and P. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 73–128. Kluwer-Academic Publishers, 1988.
8. M. Gordon and A. Pitts. The HOL Logic. In M. Gordon and T. Melham, editors, *Introduction to HOL, Part III*, pages 191–232. Cambridge University Press, 1993.
9. F. Haftmann and M. Wenzel. Constructive Type Classes in Isabelle. In T. Altenkirch and C. McBride, editors, *TYPES conference*, pages 160–174. Springer, 2006.
10. T. Hales. The flyspeck project, 2003. See `http://code.google.com/p/flyspeck/`.

11. R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, 1993.
12. R. Harper, D. Sannella, and A. Tarlecki. Structured presentations and logic representations. *Annals of Pure and Applied Logic*, 67:113–160, 1994.
13. J. Harrison. HOL Light: A Tutorial Introduction. In *Proceedings of the First International Conference on Formal Methods in Computer-Aided Design*, pages 265–269. Springer, 1996.
14. F. Horozal and F. Rabe. Representing Model Theory in a Type-Theoretical Logical Framework. In *Fourth Workshop on Logical and Semantic Frameworks, with Applications*, volume 256 of *Electronic Notes in Theoretical Computer Science*, pages 49–65, 2009.
15. M. Iancu and F. Rabe. A formal semantics of isabelle/hol, 2010. See `https://svn.kwarc.info/repos/twelf/projects/isabelle`.
16. P. Martin-Löf. An Intuitionistic Theory of Types: Predicative Part. In *Proceedings of the '73 Logic Colloquium*, pages 73–118. North-Holland, 1974.
17. T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. Springer, 2002.
18. L. Paulson. The Foundation of a Generic Theorem Prover. *Journal of Automated Reasoning*, 5(3):363–397, 1989.
19. L. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer, 1994.
20. L. Paulson and M. Coen. Zermelo-Fraenkel Set Theory, 1993. Isabelle distribution, ZF/ZF.thy.
21. F. Pfenning and C. Schürmann. System description: Twelf - a meta-logical framework for deductive systems. *Lecture Notes in Computer Science*, 1632:202–206, 1999.
22. 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.
23. A. Trybulec. Tarski Grothendieck Set Theory. *Journal of Formalized Mathematics*, Axiomatics, 1989.
24. A. Trybulec and H. Blair. Computer Assisted Reasoning with MIZAR. In A. Joshi, editor, *Proceedings of the 9th International Joint Conference on Artificial Intelligence*, pages 26–28, 1985.
25. M. Wenzel. The Isabelle/Isar Reference Manual, 2009. `http://isabelle.in.tum.de/documentation.html`, Dec 3, 2009.
26. E. Zermelo. Untersuchungen über die Grundlagen der Mengenlehre I. *Mathematische Annalen*, 65:261–281, 1908. English title: Investigations in the foundations of set theory I.

A Encoding Isabelle in LF

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

- Theories:
  $\ulcorner \texttt{theory}\ T\ \texttt{imports}\ T_1, \ldots, T_n\ \texttt{begin}\ \Sigma\ \texttt{end} \urcorner =$
  $\texttt{sig}\ T = \{\ \texttt{include}\ Pure.\ \texttt{include}\ T_1. \ldots \texttt{include}\ T_n. \ulcorner \Sigma \urcorner\}.$
  where theory content Σ is translated component-wise.
- Locales:
  $\ulcorner \texttt{locale}\ loc = I_1 \ldots I_n\ \texttt{for}\ \Sigma + \Sigma' \urcorner =$
  $\texttt{sig}\ loc = \{\ \ulcorner \Sigma \urcorner \ulcorner I_1 \urcorner \ldots \ulcorner I_n \urcorner \ulcorner \Sigma' \urcorner\ \}.$
  where the `for` clause Σ and the locale content Σ' are translated component-wise. The named instances I_i are translated according to
  $\ulcorner ins_i : loc_i\ \texttt{where}\ \sigma_i \urcorner = \texttt{struct}\ ins_i : loc_i = \{\ulcorner \sigma_i \urcorner\ \rho_i\}.$
  Here the occurrence of ρ_i is due to a subtlety in the semantics of Isabelle locales: If a locale inherits two equal instances (same locale, equivalent instantiations), they are identified rather than producing two distinct copies of the same declarations (which they do in LF). Therefore, we need an explicit sharing declaration. In LF, that is just a special case of instantiation: ρ_i contains $\texttt{struct}\ ins := ins'.$ for every instance ins imported via I_i that is equal to an instance ins' already imported via $I_1, \ldots, I_{i-1}$. Furthermore, if loc inherits from more than one type class, all their $this$ fields are shared. [1]
- Sublocales:
  $\ulcorner \texttt{sublocale}\ loc' < loc\ \texttt{where}\ \sigma\ P_1\ \ldots\ P_n \urcorner =$
  $\texttt{view}\ \nu : loc \to loc' = \{\ulcorner \sigma \urcorner\ a_1 := \ulcorner P_1 \urcorner.\ \ldots\ a_n := \ulcorner P_n \urcorner.\}.$
  Here ν is a fresh name because LF views are always named. a_i is the qualified name of the axiom declared in or imported into loc inducing the proof obligation discharged by P_i. Note how the discharging of proof obligations is just a special case of an instantiation.
  If loc contains an import $ins_1 : loc_1$ and if a sublocale declaration $loc' < loc_1$ encoded as a view $\nu_1 : loc_1 \to loc$ is present, it is possible to discharge all proof obligations inherited from loc_1 at once with an instantiation $\texttt{struct}\ ins_1 := \nu_1.$. The same is possible if $loc' < loc_1$ holds because loc' also inherits from loc_1. In both cases the involved instantiations have to match which is checked by Twelf.
- Interpretations:
  $\ulcorner \texttt{interpretation}\ loc\ \texttt{where}\ \sigma\ P_1\ \ldots\ P_n \urcorner =$
  $\texttt{view}\ \nu : loc \to T = \{\ulcorner \sigma \urcorner\ a_1 := \ulcorner P_1 \urcorner.\ \ldots\ a_n := \ulcorner P_n \urcorner.\}.$
  Here ν is a fresh name and T is the name of the Isabelle theory containing the interpretation (which is also the LF signature containing the view). a_i is as for sublocales above.
- Type classes:
  $\ulcorner \texttt{class}\ C = C_1\ \ldots\ C_n + \Sigma \urcorner = \texttt{sig}\ C = \{this : tp.\ I_1.\ \ldots\ I_n. \ulcorner \Sigma \urcorner\}.$

[1] Interestingly, LF structures seem both simpler and more expressive than Isabelle locale instances.

where I_i abbreviates $\texttt{struct}\ ins_i\ : C_i\ = \{this := this.\ \rho_i\}$ for some fresh names ins_i.
Σ is translated component-wise with one exception: All references in Σ to the single permitted type variable $\alpha :: C$ are translated to $this$. Note that another $this$ is imported from each superclass C_i, but they are all shared using the instantiation $this := this.$.
ρ_i contains one structure sharing declaration for each type class imported by I_i that has already been imported by $I_1, \ldots, I_{i-1}$. This is exactly the same as for locales above.

- Instantiations: The representation of instantiations requires an auxiliary signature ν holding instances of the n type variables of the classes C_i. Then a view ν interprets C in terms of these instances.
  $\ulcorner\texttt{instantiation}\ t :: (C_1, \ldots, C_n)C\ \texttt{begin}\ D_1 \ldots D_m\ P_1 \ldots P_n$
  $\texttt{end}\urcorner$
  $= \texttt{sig}\ \nu = \{\texttt{struct}\ \alpha_1\ : C_1 \ldots \texttt{struct}\ \alpha_n\ : C_n\}.$
  $\texttt{view}\ \nu' : C \to \nu = \{this := (t\ \alpha_1.this\ \ldots\ \alpha_n.this).\ \sigma\}.$
  ν maps the $this$ field of C to the type that instantiates C – the application of t to its type arguments. σ provides the instantiations for the operations and axioms of C. It is obtained as follows. Each D_i must be a definition $c_i\_def : c_i\ x_1\ \ldots\ x_r \equiv t_i$ for a (possibly imported) constant c_i of C. And each P_i must be the proof a proof obligation induced by a (possibly imported) axiom of C. Then σ contains (i) $q_i := [x_1] \ldots [x_r]\ulcorner t_i\urcorner$. for $i = 1, \ldots, m$ where q_i is the qualified name of c_i ($q_i = c_i$ if c_i is declared in C; otherwise, q_i is obtained by prefixing the structure names chosen when translating the type class C) and (ii) $a_i := \ulcorner P_i\urcorner$. for $i = 1, \ldots, n$ where a_i is the qualified name of the axiom of C inducing the respective proof obligation.
- Typed constants where $\alpha_1 :: C_1, \ldots, \alpha_n :: C_n$ are the type variables in τ:
  $$\ulcorner c :: \tau\urcorner \quad = \quad c : \{\alpha_1 : C_1\} \ldots \{\alpha_n : C_n\}\ tm\ \ulcorner\tau\urcorner.$$
- Constant definitions where $\alpha_1 :: C_1, \ldots, \alpha_n :: C_n$ are the type variables in t:
  $$\ulcorner c\_def : f\ x_1 \ldots\ x_n \equiv t\urcorner \quad =$$
  $$c\_def : \{\alpha_1 : C_1\} \ldots \{\alpha_n : C_n\}\ \vdash\ (f\ \alpha_1\ \ldots\ \alpha_n)\ @x_1 \ldots @x_n \equiv \ulcorner t\urcorner.$$
- Axioms where $\alpha_1 :: C_1, \ldots, \alpha_n :: C_n$ are the type variables in φ:
  $$\ulcorner a :: \varphi\urcorner \quad = \quad a : \{\alpha_1 : C_1\} \ldots \{\alpha_n : C_n\}\ \vdash \ulcorner\varphi\urcorner.$$
- Lemmas where $\alpha_1 :: C_1, \ldots, \alpha_n :: C_n$ are the type variables in φ:
  $$\ulcorner l :: \varphi\ P\urcorner \quad = \quad l : \{\alpha_1 : C_1\} \ldots \{\alpha_n : C_n\}\ \vdash \ulcorner\varphi\urcorner = [\alpha_1 : C_1] \ldots [\alpha_n : C_n]\ulcorner P\urcorner.$$
- Type declarations (where we include the case $n = 0$ of base types):
  $$\ulcorner(\alpha_1, \ldots, \alpha_n)\ t\urcorner \quad = \quad t : tp \to \ldots \to tp \to tp.$$
- Type definitions:
  $$\ulcorner(\alpha_1, \ldots, \alpha_n)\ t = \tau\urcorner \quad = \quad t : tp \to \ldots \to tp \to tp = [\alpha_1 : tp] \ldots [\alpha_n : tp]\ulcorner\tau\urcorner.$$

- Named instantiations:

$$\ulcorner c_1 = t_1 \ \ldots \ c_m = t_m \urcorner \quad = \quad c_1 := \ulcorner t_1 \urcorner. \ \ldots \ c_m := \ulcorner t_m \urcorner.$$

- Constants. An Isabelle constant c that is declared in the current theory or locale and is used with inferred type arguments $\tau_1 :: C_1, \ldots, \tau_n :: C_n$ is represented as an LF application $T.c \ \ulcorner \tau_1 :: C_1 \urcorner \ \ldots, \ \ulcorner \tau_n :: C_n \urcorner$.
  An Isabelle constant c that is declared in an imported theory T is represented in the same way except for using $T.c$ instead of c.
  In a theory or locale, an Isabelle constant c that is declared in the type class C and that is used on the inferred type class instance $\tau :: C$ is translated to the morphism application $\ulcorner \tau :: C \urcorner(c)$.
  Within a type class C, an Isabelle constant c that is imported from another type class is translated to the qualified name arising by prefixing the structure name ins generated in the representation of the class C.
- Type class instances. Given an Isabelle type class instance $\tau :: C$ in some type variables $\alpha_1 :: C_1, \ldots, \alpha_n :: C_n$, we obtain an LF-morphism $\ulcorner \tau :: C \urcorner$ in structure variables $\texttt{struct}\ \alpha : C_1, \ldots, \texttt{struct}\ \alpha_n : C_n$.
  Firstly, let $\tau{=}\alpha_i$ be a type variable. If $C{=}C_i$, we can put $\mu{=}\alpha$. Otherwise, C_i must be a subclass of C. That means there must be a morphism $incl{=}D_1.ins_1 \ \ldots \ D_r.ins_r$ from C to C_i where (i) ins_j is a type class import from D_{j-1} to D_j and (ii) $D_1.ins_1$ instantiates C and $D_r = C_i$. Thus, we put $\mu{=}incl\ \alpha$.
  Secondly, let $\tau{=}(\omega_1, \ldots, \omega_m)t$ be a type operator application (including the case $n = 0$ for base types). If τ has Isabelle type class C, there must be a corresponding type class instantiation $t :: (\Omega_1, \ldots, \Omega_m)D$ (*) and type class instances $\omega_i :: \Omega_i$ (**).
  Due to (*), there are an LF signature $S = \{\texttt{struct}\ \beta_1 \ : D_1 \ldots \texttt{struct}\ \beta_m \ : D_m\}$. and a view v from D to S (see the representation of type class instantiations above). Due to (**), we obtain recursively morphisms μ_i from Ω_i into the current signature. Then $\mu_S{=}\{\beta_1 := \mu_1. \ \ldots \ \beta_m := \mu_m.\}$ is a morphism from S into the current signature.
  Now if $C{=}D$, we obtain the needed morphism $\ulcorner \tau :: C \urcorner{=}v \ \mu_S$ from C to the current signature. Otherwise, D must be a subclass of C, so let $incl : C \to D$ be obtained as for type variables above; then we obtain $\ulcorner \tau :: C \urcorner{=}incl\ v\ \mu_S$.
- Proofs are mapped in a straightforward way replacing all invocations of a proof rule r with $Pure.r$ and all references to definitions, axioms, or theorems are translated as for constants.
- Remaining inner syntax:

$$\begin{array}{lcl}
\ulcorner (\tau_1, \ldots, \tau_n)\ t \urcorner & = & \ulcorner t \urcorner \ulcorner \tau_1 \urcorner \ \ldots \ \ulcorner \tau_n \urcorner \\
\ulcorner \tau \Rightarrow \tau' \urcorner & = & \ulcorner \tau \urcorner \Rightarrow \ulcorner \tau' \urcorner \\
\ulcorner prop \urcorner & = & prop \\
\ulcorner c\ t_1 \ \ldots, t_n \urcorner & = & \ulcorner c \urcorner @ \ulcorner t_1 \urcorner \ \ldots \ @ \ulcorner t_n \urcorner \\
\ulcorner \lambda x_1 :: \tau_1 \ \ldots \ x_n :: \tau_n.t \urcorner & = & \lambda[x_1 : tm \ulcorner \tau_1 \urcorner] \ldots \lambda[x_n : tm \ulcorner \tau_n \urcorner] \ulcorner t \urcorner \\
\ulcorner \varphi \Longrightarrow \psi \urcorner & = & \ulcorner \varphi \urcorner \Longrightarrow \ulcorner \psi \urcorner \\
\ulcorner \bigwedge \lambda x_1 :: \tau_1 \ \ldots \ x_n :: \tau_n.\varphi \urcorner & = & \bigwedge[x_1 : tm \ulcorner \tau_1 \urcorner] \ldots \bigwedge[x_n : tm \ulcorner \tau_n \urcorner] \ulcorner \varphi \urcorner \\
\ulcorner t \equiv t' \urcorner & = & \ulcorner t \urcorner \equiv \ulcorner t' \urcorner
\end{array}$$

Term variables are represented as themselves. Type variables $\alpha :: C$ are translated to $\ulcorner \alpha :: C \urcorner(this)$ where $\ulcorner \alpha :: C \urcorner$ is an LF-morphism from C into the current signature. Note that the single type variable permitted within type classes is treated differently (see type classes above).

Actually, we have to escape all characters occurring in Isabelle identifiers that are illegal in Twelf identifiers, but we omit this technicality.