Math. Struct. in Comp. Science (2011), *vol.* 21, *pp.* 883–911. © Cambridge University Press 2011
doi:10.1017/S0960129511000144

Formalising foundations of mathematics[†]

MIHNEA IANCU and FLORIAN RABE

Computer Science, Jacobs University Bremen, Bremen, Germany
Email: {m.iancu;f.rabe}@jacobs-university.de

Received 11 July 2010; revised 15 December 2010

Over recent decades there has been a trend towards formalised mathematics, and a number of sophisticated systems have been developed both to support the formalisation process and to verify the results mechanically. However, each tool is based on a specific foundation of mathematics, and formalisations in different systems are not necessarily compatible. Therefore, the integration of these foundations has received growing interest. We contribute to this goal by using LF as a foundational framework in which the mathematical foundations themselves can be formalised and therefore also the relations between them. We represent three of the most important foundations – Isabelle/HOL, Mizar and ZFC set theory – as well as relations between them. The relations are formalised in such a way that the framework permits the extraction of translation functions, which are guaranteed to be well defined and sound. Our work provides the starting point for a systematic study of formalised foundations in order to compare, relate and integrate them.

1. Introduction

The twentieth century saw significant advances in the field of mathematical foundations, stimulated by the discovery of paradoxes in naive set theory, for example, Russell's paradox of unlimited set comprehension. Several seminal works have redeveloped and advanced large parts of mathematics based on one coherent choice of foundation, most notably the *Principia* (Whitehead and Russell 1913) and the work by Bourbaki (Bourbaki 1964). Today various flavors of axiomatic set theory and type theory provide a number of well-understood foundations.

Given a development of mathematics in one fixed foundation, it is possible to give a fully formal language in which every mathematical expression valid in that foundation can be written down. Then mathematics can, in principle, be reduced to the manipulation of these expressions, an approach called *formalism* and most prominently expressed in Hilbert's program. This approach has recently gained increasing momentum due to the advent of computer technology. With machine support, the formidable effort of formalising mathematics becomes feasible, and trust in the soundness of an argument can be reduced to trust in the implementation of the foundation.

However, compared with 'traditional' mathematics, this approach has the drawback that it relies heavily on the choice of a specific foundation. On the other hand, traditional mathematics frequently, and often crucially, abstracts from and moves freely between

[†] This work was supported by grant KO 2428/9-1 (LATIN) by the Deutsche Forschungsgemeinschaft.

foundations to the extent that many mathematical papers do not mention exactly which foundation is used. This level of abstraction is very difficult to capture if every statement is rigorously reduced to a fixed foundation. Moreover, in formalised mathematics, different systems implementing different (or even the same or similar) foundations are often incompatible, and no reuse across systems is possible.

But the high cost of formalising mathematics makes it desirable to join forces and integrate foundational systems. Currently, due to the lack of integration, significant overlap and redundancies exist between libraries of formalised mathematics, which slows down the progress of large projects such as the formal proofs of the Kepler conjecture (Hales 2003).

Our contribution can be summarised as follows:

(1) We introduce a new methodology for the formal integration of foundations. Using a logical framework, we formalise not only mathematical theories but also the foundations themselves, which allows the formal statement and proof of the relations between foundations.

(2) Then we demonstrate our approach by formalising three of the most widely used and important foundations, as well as translations between them.

Our work provides the starting point of a formal library of foundational systems, which complements the existing foundation-specific libraries and provides the basis for the systematic and formally verified integration of systems for formalised mathematics.

1.1. *Outline of the paper*

We begin by describing our approach and reviewing related work in Section 2. Then, in Section 3, we give an overview of the logical framework we use in the remainder of the paper. We give a new formalisation of traditional mathematics based on ZFC set theory in Section 4. Then we formalise two foundations with particularly large formalised libraries: Isabelle/HOL (Nipkow *et al.* 2002) in Section 5 and Mizar (Trybulec and Blair 1985) in Section 6. We also give a translation from Isabelle/HOL into ZFC and sketch a partial translation from Mizar (which is stronger than ZFC) into ZFC. We discuss our work and present our conclusions in Section 7.

Our formalisations span several thousand lines of declarations, and the descriptions we give here are necessarily simplified – the full sources are available at `https://latin.omdoc.org/wiki/FormalizingFoundations`.

2. Problem statement and related work

Automath (de Bruijn 1970) and the formalisation of Landau's analysis (Landau 1930; van Benthem Jutting 1977) were the first major successes of formalised mathematics. Since then, a number of computer systems have been put forward and have been adopted to varying degrees to formalise mathematics. These include:

— LCF (Gordon *et al.* 1979);

— HOL (Gordon 1988);

https://doi.org/10.1017/S0960129511000144 Published online by Cambridge University Press

— HOL Light (Harrison 1996);
— Isabelle/HOL (Nipkow *et al.* 2002);
— IMPS (Farmer *et al.* 1993);
— Nuprl (Constable *et al.* 1986);
— Coq (Coquand and Huet 1988);
— Mizar (Trybulec and Blair 1985);
— Isabelle/ZF (Paulson and Coen 1993).

The body of peer-reviewed formalised mathematics is also growing (Hales *et al.* 2008; Matuszewski 1990; Klein *et al.* 2004) – see Wiedijk (2006) for a comparison of some formalisations of foundations in Automath, including ZFC and Isabelle/HOL.

The problem of interoperability and integration between these systems has received growing attention recently, and a number of connections between them have been established. Obua and Skalberg (2006) and McLaughlin (2006) translate between Isabelle/HOL and HOL Light; Keller and Werner (2010) from HOL Light to Coq; and Krauss and Schropp (2010) from Isabelle/HOL to Isabelle/ZF. The OpenTheory format (Hurd 2009) was designed as an interchange format for different implementations of higher order logic.

We call these translations *dynamically verified* because they have in common that they translate theorems in such a way that the target system reproves every translated theorem. One can think of the source system's proof as an oracle for the target system's proof search. This approach requires no reasoning about or trust in the translation, so users of the target system can reuse translated theorems without making the source system or the translation part of their trusted code base. Therefore, such translations can be implemented and put to use relatively quickly. It is no surprise that such translations are advanced by researchers working with the respective target system.

Still, dynamically verified translations can be unsatisfactory. The proofs of the source theorems may not be available because they only exist transiently when the source system processes a proof script. The source system might not be able to export the proofs, or they may be too large to translate. In that case, it is desirable to translate only the theorems and appeal to a general result that guarantees the soundness of the theorem translation.

However, the statement of soundness naturally lives outside either of the two foundations involved. Therefore, stating, let alone proving, the soundness of a translation requires a third formal system in which the source and target systems and the translation are represented. We call this third system a *foundational framework*, and if the soundness of a translation is proved in a foundational framework, we say it is a *statically verified* translation.

Statically verified translations are theoretically more appealing because the soundness is proved once and for all. Of course, this requires the additional assumptions that the foundational framework is consistent and that the representations in the framework are adequate. If this is a concern, the soundness proof should be *constructive*, that is, produce for every proof in the source system a translated proof in the target system. Then users of the target system have the option of rechecking the translated proof.

The most comprehensive example of a statically verified translation – from HOL to Nuprl (Naumov *et al.* 2001) – was given in Schürmann and Stehr (2004). HOL and Nuprl proof terms are represented as terms in the framework Twelf (Harper *et al.* 1993; Pfenning

https://doi.org/10.1017/S0960129511000144 Published online by Cambridge University Press

and Schürmann 1999) using the judgments-as-types methodology. The translation and a constructive soundness proof are formalised as type-preserving logic programs in Twelf. The soundness is verified by the Twelf type checker, and the well-definedness, that is, the totality and termination of the logic programs involved, is proved using Twelf.

In the current paper, we demonstrate a general methodology for statically verified translations. We formalise foundations as signatures in the logical framework Twelf, and we use the LF module system's (Rabe and Schürmann 2009) translations-as-morphisms methodology to formalise translations between them as signature morphisms. This yields translations that are well defined and sound by design, and which are verified by the Twelf type checker. Moreover, they are constructive, and the extraction of translation programs is straightforward.

Our work can be seen as a continuation of the Logosphere project (Pfenning *et al.* 2003), of which the above HOL-Nuprl translation was a part. Both Logosphere and our work use LF, and the main difference is that we use the new LF module system to reuse encodings and to encode translations. Logosphere had to use monolithic encodings and used programs to encode translations. The latter were either Twelf logic programs or Delphin (Poswolsky and Schürmann 2008) functional programs, and their well definedness and termination was statically verified by Twelf and Delphin, respectively. Using the module system, translations can be stated in a more concise and declarative way, and the well definedness of translations is guaranteed by the LF type theory.

There are a number of alternative frameworks in which foundations can be formalised: other variants of dependent type theory such as Agda (Norell 2005); type theories such as Coq based on the calculus of inductive constructions; and the Isabelle framework (Paulson 1994) based on polymorphic higher-order logic. All of these provide roughly comparably expressive module systems. We chose LF because the judgments-as-types and relations-as-morphisms methodologies are especially appropriate for formalising foundations and their relations.

We discuss related work pertaining to the individual foundations separately below.

3. The Edinburgh Logical Framework

The Edinburgh Logical Framework (Harper *et al.* 1993) (LF) is a formal meta-language used for the formalisation of deductive systems. It is related to Martin-Löf type theory and the corner of the lambda cube that extends simple type theory with dependent function types and kinds. We will work with the Twelf (Pfenning and Schürmann 1999) implementation of LF and its module system (Rabe and Schürmann 2009).

The central notion of the LF type theory is that of a *signature*, which is a list Σ of *kinded type family* symbols $a : K$ or *typed constant* symbols $c : A$. It is convenient to permit them to carry optional definitions, for example, $c : A = t$ to define c as t. (For our purposes, it is sufficient to assume that these abbreviations are transparent to the underlying type theory, which avoids some technical complications. Of course, they are implemented more intelligently.)

LF *contexts* are lists Γ of typed variables $x : A$, that is, there is no polymorphism. Relative to a signature Σ and a context Γ, the expressions of the LF type theory are *kinds*

https://doi.org/10.1017/S0960129511000144 Published online by Cambridge University Press

K, *kinded type families* $A : K$, and *typed terms* $t : A$. `type` is a special kind, and type families of kind `type` are called *types*.

We will use the concrete syntax of Twelf to represent expressions:

- The dependent function type $\Pi_{x:A}B(x)$ is written $\{x : A\}\, B\ x$, and correspondingly for dependent function kinds $\{x : A\}\, K\ x$. As usual, we write $A \to B$ when x does not occur free in B.
- The corresponding λ-abstraction $\lambda_{x:A}t(x)$ is written $[x : A]\, t\ x$, and correspondingly for type families $[x : A]\,(B\ x)$.
- As usual, application is written as juxtaposition.

Given two signatures $\mathtt{sig}\ S = \{\Sigma\}$ and $\mathtt{sig}\ T = \{\Sigma'\}$, a *signature morphism* σ from S to T is a list of assignments $c := t$ and $a := A$. They are called *views* in Twelf and declared as $\mathtt{view}\ v : S \to T = \{\sigma\}$. Such a view is well formed if

- σ contains exactly one assignment for every symbol c or a that is declared in Σ without a definition.
- Each assignment $c := t$ assigns to the Σ-symbol $c : A$ a Σ'-term t of type $\overline{\sigma}(A)$.
- Each assignment $a := K$ assigns to the Σ-symbol $a : K$ a Σ'-type family K of type $\overline{\sigma}(K)$.

Here $\overline{\sigma}$ is the homomorphic extension of σ that maps all closed expressions over Σ to closed expressions over Σ', and we will write it simply as σ in the rest of the paper. The central result concerning signature morphisms (Harper *et al.* 1994) is that they preserve typing and $\alpha\beta\eta$-equality, so judgments $\vdash_\Sigma t : A$ imply judgments $\vdash_{\Sigma'} \sigma(t) : \sigma(A)$, and similarly for kinding judgments and equality.

Finally, the Twelf module system permits inclusions between signatures and views. If a signature T contains the declaration $\mathtt{include}\ S$, then all symbols declared in (or included in) S are available in T through qualified names, for example, c of S is available as $S.c$. Our inclusions will never introduce name clashes, and we will write c instead of $S.c$ for simplicity. Correspondingly, if S is included in T, and we have a view v from S to T', a view from T to T' may include v through the declaration $\mathtt{include}\ v$.

This yields the following grammar for Twelf, where the gray colour denotes optional parts:

Toplevel	G	::=	$\cdot \mid G,\ \mathtt{sig}\ T = \{\Sigma\} \mid G,\ \mathtt{view}\ v : S \to T = \{\sigma\}$
Signatures	Σ	::=	$\cdot \mid \Sigma,\ \mathtt{include}\ S \mid \Sigma,\ c : A = t \mid \Sigma,\ a : K = t$
Morphisms	σ	::=	$\cdot \mid \Sigma,\ \mathtt{include}\ v \mid \sigma,\ c := t \mid \sigma,\ a := A$
Kinds	K	::=	$\mathtt{type} \mid \{x : A\}\, K$
Type families	A	::=	$a \mid A\ t \mid [x : A]\, A \mid \{x : A\}\, A$
Terms	t	::=	$c \mid t\ t \mid [x : A]\, t \mid x$

We will sometimes omit the type of a bound variable if it can be inferred from the context. Moreover, we will frequently use implicit arguments. For example, if c is declared as $c : \{x : A\}\, B$ and the value of s in $c\ s$ can be inferred from the context, then c may alternatively be declared as $c : B$ (with a free variable in B that is implicitly bound) and used as c (where the argument to c inferred). We will also use fixity and precedence declarations in the style of Twelf to make applications more readable.

Example 1 (Representation of FOL in LF). The following is a fragment of an LF signature for first-order logic, which we will use later to formalise set theory:

$$
\begin{array}{llll}
\multicolumn{4}{l}{\texttt{sig}\ FOL\ =\ \{} \\
\quad & set & : \ \texttt{type} & \\
& prop & : \ \texttt{type} & \\
& ded & : \ prop \rightarrow \texttt{type} & \texttt{prefix}\ 0 \\
& & & \\
& \Rightarrow & : \ prop \rightarrow prop \rightarrow prop & \texttt{infix}\ 3 \\
& \wedge & : \ prop \rightarrow prop \rightarrow prop & \texttt{infix}\ 2 \\
& \forall & : \ (set \rightarrow prop) \rightarrow prop & \\
& \doteq & : \ set \rightarrow set \rightarrow prop & \texttt{infix}\ 4 \\
& \Leftrightarrow & : \ prop \rightarrow prop \rightarrow prop & \texttt{infix}\ 1 \\
& & = [a]\,[b]\,(a\ \Rightarrow\ b)\ \wedge\ (b\ \Rightarrow\ a) & \\
& \wedge_I & : \ ded\ A \rightarrow ded\ B \rightarrow ded\ A\ \wedge\ B & \\
& \wedge_{E_l} & : \ ded\ A\ \wedge\ B \rightarrow ded\ A & \\
& \wedge_{E_r} & : \ ded\ A\ \wedge\ B \rightarrow ded\ B & \\
& \Rightarrow_I & : \ (ded\ A \rightarrow ded\ B) \rightarrow ded\ A\ \Rightarrow\ B & \\
& \Rightarrow_E & : \ ded\ A\ \Rightarrow\ B \rightarrow ded\ A \rightarrow ded\ B & \\
& \forall_I & : \ (\{x : i\}\ ded\ (F\ x)) \rightarrow ded\ (\forall\ [x]\ \ F\ x) & \\
& \forall_E & : \ ded\ (\forall\ [x]\ \ F\ x) \rightarrow \{c : i\}\ ded\ (F\ c) & \\
\multicolumn{4}{l}{\}}
\end{array}
$$

This introduces the two types *set* and *prop* for sets and propositions, respectively, and a type family *ded* indexed by propositions. Terms p of type $ded\ F$ represents proofs of F, and the inhabitation of $ded\ F$ represents the provability of F. Higher-order abstract syntax is used to represent binders, for example, $\forall([x : set]\,F\ x)$ represents the formula $\forall x : set.F(x)$. Equivalence is introduced as a defined connective.

Note that the argument of *ded* does not require brackets as *ded* has the weakest precedence. Moreover, by convention, the Twelf binders [] and {} always bind as far to the right as is consistent with the placement of brackets.

As examples for inference rules, we give natural deduction introduction and elimination rules for conjunction and implication. Here A and B of type *prop* are implicit arguments whose types and values are inferred. For example, the theorem of commutativity of conjunction can now be stated as

$$
\begin{array}{lll}
comm\_conj & : & ded\,(A\ \wedge\ B)\ \Leftrightarrow\ (B\ \wedge\ A) \\
& = & \wedge_I\ (\Rightarrow_I\ \ [p]\,\wedge_I\ (\wedge_{E_r}\ p)\ (\wedge_{E_l}\ p)) \\
& & \quad (\Rightarrow_I\ \ [p]\,\wedge_I\ (\wedge_{E_r}\ p)\ (\wedge_{E_l}\ p))
\end{array}
$$

The LF type system guarantees that the proof is correct.

4. Zermelo-Fraenkel set theory

In this section, we present out formalisation of Zermelo-Fraenkel set theory. We give an overview of our variant of ZFC in Section 4.1 and describe its encoding in Section 4.2 and 4.3. Finally, we discuss related formalisations in Section 4.4.

https://doi.org/10.1017/S0960129511000144 Published online by Cambridge University Press

4.1. *Preliminaries*

Zermelo-Fraenkel set theory (Zermelo 1908; Fraenkel 1922) (with or without choice) is the most common implicitly or explicitly assumed foundation of mathematics. It represents all mathematical objects as sets related by the binary $\in$ predicate. Propositions are stated using an untyped first-order logic. The logic is classical, but we will take care to reason intuitionistically whenever possible.

There are a number of equivalent choices for the axioms of ZFC. Our axioms are

- Extensionality: $\forall x \forall y (\forall z (z \in x \Leftrightarrow z \in y) \Rightarrow x = y)$.
- Set existence: $\exists x\ \mathit{true}$ (this could be derived from the axiom of infinity, but we add it explicitly here to reduce dependence on infinity).
- Unordered pairing: $\forall x \forall y \exists a (\forall z (z = x \vee z = y) \Rightarrow z \in a)$.
- Union: $\forall X \exists a \forall z (\exists x (x \in X \wedge z \in X) \Rightarrow z \in a)$.
- Power set: $\forall x \exists a \forall z ((\forall t (t \in z \Rightarrow t \in x)) \Rightarrow z \in a)$.
- Specification: $\forall X \exists a (\forall z ((z \in X \wedge \varphi(z)) \Leftrightarrow z \in a))$ for a unary predicate φ (possibly containing free variables).
- Replacement: $\forall a (\forall x (x \in a) \Rightarrow \exists^! y (\varphi\ x\ y)) \Rightarrow \exists b (\forall y (\exists x (x \in a \wedge \varphi(x, y)) \Leftrightarrow y \in b))$ for a binary predicate φ (possibly containing free variables) where $\exists^!$ abbreviates the easily definably quantifier of unique existence.
- Regularity: $\forall x (\exists t (t \in x)) \Rightarrow (\exists y (y \in x \wedge \neg (\exists z (z \in x \wedge z \in y))))$.
- Choice and infinity, which we omit here.

It is important to note that there are no first-order terms other than for the variables. Specific sets (that is, first-order constant symbols) and operations on sets (that is, first-order function symbols) are introduced only as derived notions: a new symbol may be introduced to abbreviate a uniquely determined set. For example, the empty set $\varnothing$ abbreviates the unique set x satisfying $\forall y . \neg y \in x$. Adding such abbreviations is conservative over first-order logic but cannot be formalised within the language of first order.

4.2. *Untyped set theory*

Our Twelf formalisation of ZFC uses three main signatures: ZFC_FOL encodes first-order logic; ZFC encodes the first-order theory of ZFC; and, finally, *Operations* introduces the basic operations and their properties, most notably, products and functions. The actual encodings (Iancu and Rabe 2010) comprise several hundred lines of Twelf declarations and are divided between a number of smaller signatures to enhance maintainability and reuse. Therefore, our presentation here is only a summary. Moreover, to enhance readability, we will use more Unicode characters in identifiers here than in the actual encodings.

4.2.1. *First-order logic.* ZFC_FOL is an extension of the signature FOL given in Example 1. Besides the usual components of FOL encodings in LF (see, for example, Harper *et al.* (1993)), we use two special features.

First, we add the (definite) *description operator*

$$\delta \ :\ \{F : set \to prop\}\, ded\ \exists^!\ ([x]\, F\ x) \to set,$$

https://doi.org/10.1017/S0960129511000144 Published online by Cambridge University Press

which encodes the mathematical practice of giving a name to a uniquely determined object. Here $\exists^!$ is the quantifier of unique existence, which is easily definable. Thus δ takes a formula $F(x)$ with a free variable x and a proof of $\exists^! x.F(x)$, and returns a new set. The LF type system guarantees that δ can only be applied after showing unique existence. δ is axiomatised using the axiom scheme $ax_\delta : ded\ F\ (\delta\ F\ P)$, and from this we can derive irrelevance, that is, $\delta\ F\ P$ returns the same object no matter which proof P is used.

Then we add *sequential connectives* for conjunction and implication. In a sequential implication $F \Rightarrow' G$, we only consider G if F is true, and similarly for conjunction. This is very natural in mathematical practice – for example, mathematicians do not hesitate to write $x \neq 0 \Rightarrow' x/x = 1$ when / is only defined for non-zero dividers. All other connectives remain as usual.

Sequential implication and conjunction are formalised in LF as follows:

$$\begin{array}{lll}
\wedge' & : & \{F : prop\}\,(ded\ F \to prop) \to prop \\
\Rightarrow' & : & \{F : prop\}\,(ded\ F \to prop) \to prop \\
\wedge'_I & : & \{p : ded\ F\}\ ded\ G\ p\ \to\ F \wedge' [p]\, G\ p \\
\wedge'_{E_l} & : & ded\ F \wedge' [p]\, G\ p\ \to\ ded\ F \\
\wedge'_{E_r} & : & \{q : ded\ F \wedge' [p]\, G\ p\}\ ded\ G\ (\wedge' E_l\ q) \\
\Rightarrow'_I & : & (\{p : ded\ F\}\ ded\ G\ p)\ \to\ ded\ F \Rightarrow' [p]\, G\ p \\
\Rightarrow'_E & : & ded\ F \Rightarrow' [p]\, G\ p\ \to\ \{p : ded\ F\}\ ded\ G\ p
\end{array}$$

$\wedge'$ and $\Rightarrow'$ are applied to two arguments: first a formula F and then a formula G stated in a context in which F is true. This is written as, for example, $F \wedge' [p]\, G\ p$ where p is an assumed proof of F that may occur in G. We will use $F \wedge' G$ and $F \Rightarrow' G$ as abbreviations when p does not occur in G, which yield the non-sequential cases. The introduction and elimination rules are generalised accordingly. Note that these sequential connectives do not rely on classicality.

In plain first-order logic, such sequential connectives would be useless as a proof cannot occur in a formula. But in the presence of the description operator, the proofs frequently occur in terms, and thus in formulas.

4.2.2. *Set theory.* The elementhood predicate is encoded as $\in : set \to set \to prop$ together with a corresponding infix declaration. The formalisation of the axioms is straightforward, for example, the axiom of extensionality is encoded as:

$$ax\_exten : ded\ \forall\ [x]\ \ \forall\ [y]\ (\forall\ [z]\ \ z\ \in\ x\ \Leftrightarrow\ z\ \in\ y) \Rightarrow\ x\ \doteq\ y$$

It is now easy to establish the adequacy of our encoding in the sense that every well-formed closed LF-term $s : set$ over ZFC encodes a unique set satisfying a certain predicate F. This is obvious because s must be of the form $\delta\ F\ P$. The inverse does not hold as there are models of set theory with more sets than can be denoted by closed terms.

4.2.3. *Basic operations.* We can now derive the basic notions of set theory and their properties: Using the description operator and the respective axioms, we can introduce defined Twelf symbols

https://doi.org/10.1017/S0960129511000144 Published online by Cambridge University Press

$$\begin{aligned}
&\mathit{empty} &&: \mathit{set} = \ldots\\
&\mathit{uopair} &&: \mathit{set} \to \mathit{set} = \ldots\\
&\mathit{bigunion} &&: \mathit{set} \to \mathit{set} = \ldots\\
&\mathit{powerset} &&: \mathit{set} \to \mathit{set} = \ldots\\
&\mathit{image} &&: (\mathit{set} \to \mathit{set}) \to \mathit{set} \to \mathit{set} = \ldots\\
&\mathit{filter} &&: \mathit{set} \to (\mathit{set} \to \mathit{prop}) \to \mathit{set} = \ldots
\end{aligned}$$

such that *empty* encodes $\emptyset$, *uopair* x y encodes $\{x, y\}$, *bigunion* X encodes $\bigcup X$, *powerset* X encodes $\mathscr{P}X$, *image* f A encodes $\{f(x) : x \in A\}$, and *filter* A F encodes $\{x \in A \mid F(x)\}$.

For example, to define *uopair* we proceed as follows:

$$\begin{aligned}
\mathit{is\_uopair} &: \mathit{set} \to \mathit{set} \to \mathit{set} \to \mathit{prop}\\
&= [x]\,[y]\,[a]\,(\forall\,[z]\;(z \doteq x \vee z \doteq y) \Leftrightarrow z \in a)\\
\mathit{p\_uopair} &: \mathit{ded}\ \exists^!\,(\mathit{is\_uopair}\ A\ B)\\
&= \mathit{spec\_unique}\,(\mathit{shrink}\,(\forall_E\,(\forall_E\ \mathit{ax\_pairing}\ A)\ B))\\
\mathit{uopair} &: \mathit{set} \to \mathit{set} \to \mathit{set} = [x]\,[y]\;\delta\,(\mathit{is\_uopair}\ x\ y)\ \mathit{p\_uopair}
\end{aligned}$$

Here *is_uopair* x y a formalises the defining property $a \doteq \{x, y\}$ of the new function symbol, and *p_uopair* shows unique existence. The above uses two lemmas:

$$\begin{aligned}
\mathit{shrink} &: \mathit{ded}\,(\exists\,[X]\ \forall\,([z]\ (\varphi\ z) \Rightarrow z \in X))\\
&\to \mathit{ded}\,(\exists\,[x]\ \forall\,([z]\ (\varphi\ z) \Leftrightarrow z \in x)) = \cdots\\
\mathit{spec\_unique} &: \mathit{ded}\,(\exists\,[x]\ \forall\,([z]\ (\varphi\ z) \Leftrightarrow z \in x))\\
&\to \mathit{ded}\ \exists^!\,[x]\ \forall\,([z]\ (\varphi\ z) \Leftrightarrow z \in x) = \cdots
\end{aligned}$$

shrink states that if there is a set X that contains all the elements for which the predicate $\varphi : \mathit{set} \to \mathit{prop}$ holds, then the set described by φ exists. *spec_unique* states that if a predicate $\varphi : \mathit{set} \to \mathit{prop}$ describes a set, then that set exists uniquely. These lemmas can be proved easily using extensionality and specification.

4.2.4. *Advanced operations.* We can now define the advanced operations on sets in the usual way. For example, the definition of binary union $x \cup y = \bigcup\{x, y\}$ can be directly formalised as

$$\mathit{union} : \mathit{set} \to \mathit{set} \to \mathit{set} = [x]\,[y]\ \mathit{bigunion}\,(\mathit{uopair}\ x\ y)$$

We omit the definitions of singleton sets, ordered pairs, cartesian products, relations, partial functions and functions. Our definitions are standard except for the ordered pair. We define $(x, y) = \{\{x\}, \{\{y\}, \emptyset\}\}$, which is similar to Wiener's definition (Wiener 1967), but different from the more common $(x, y) = \{\{x, y\}, \{x\}\}$ due to Kuratowski. Our definition is a bit simpler to work with than Kuratowski pairs because it avoids the special case $(x, x) = \{\{x\}\}$:

$$\begin{aligned}
\mathit{pair} &: \mathit{set} \to \mathit{set} \to \mathit{set}\\
&= [a]\,[b]\ \mathit{uopair}\,(\mathit{singleton}\ a)\,(\mathit{uopair}\,(\mathit{singleton}\ b)\ \mathit{empty})
\end{aligned}$$

The difference between these pairs and Kuratowski pairs is not significant since we immediately prove the characteristic properties of pairing and then never appeal to the definition again:

https://doi.org/10.1017/S0960129511000144 Published online by Cambridge University Press

$$\begin{array}{ll}
conv_{pi1} & : ded\ pi1\ (pair\ X\ Y)\ \doteq\ X\ = \cdots \\
conv_{pi2} & : ded\ pi2\ (pair\ X\ Y)\ \doteq\ X\ = \cdots \\
conv_{pair} & : ded\ ispair\ X \to ded\ pair\ (pi1\ X)\ (pi2\ X)\ \doteq\ X\ = \cdots
\end{array}$$

The proofs are technical but straightforward.

Finally, we can define function construction $X \ni x \mapsto f(x)$ and application $f(x)$ as

$$\begin{array}{lll}
\lambda & : & set \to (set \to set) \to set = [a]\,[f]\ image\ ([x]\ pair\ x\ (f\ x))\ a \\
@ & : & set \to set \to set \\
 & = & [f]\,[a]\ bigunion\ (image\ pi2\ (filter\ f\ ([x]\,(pi1\ x) \doteq a)))
\end{array}$$

where $\lambda\ A\ f$ encodes $\{(x, f(x)) : x \in A\}$, and $@\ f\ x$ yields 'the b such that $(a, b) \in f$'. Application is defined for all sets: for example, it returns $\emptyset$ if f is not defined for x.

In the same way as for pairs, we can immediately prove the characteristic properties, which are known as $\beta\eta$-conversion and extensionality in computer science, and these are the only properties we will use later on:

$$\begin{array}{ll}
conv_{apply} & : ded\ X\ \in\ A \to ded\ @\ (\lambda\ A\ F)\ X\ \doteq\ F\ X\ = \cdots \\
conv_{lambda} & : ded\ F\ \in\ (\Rightarrow A\,B) \to ded\ \lambda\ A\ ([x]\ \ @\ F\ x)\ \doteq\ F\ = \cdots \\
func_{ext} & : ded\ F\ \in\ (\Rightarrow A\,B) \to ded\ G\ \in\ (\Rightarrow A\,B)\ \to \\
 & \quad (\{a\}\,ded\ a \in A \to ded\ @F\ a \doteq @G\ a) \to ded\ F \doteq G = \cdots
\end{array}$$

Again we omit the straightforward proofs.

4.3. *Typed set theory*

4.3.1. *Classes as types.* A major drawback of formalisations of set theory is the complexity of reasoning about elementhood and set equality. It is well known how to overcome these problems using typed languages, but in mathematical accounts of set theory, types are not primitive but derived notions, and we proceed accordingly. The central idea is to use the predicate subtype $Elem\ A = \{x : set \mid ded\ x \in A\ \ \text{inhabited}\}$ to represent the set A. In fact, we can use the same approach to recover classes as a derived notion: $Class\ F = \{x : set \mid ded\ F\ x\ \ \text{inhabited}\}$ for any unary predicate $F : set \to prop$.

However, LF does not support predicate subtypes (for the good reason that it would make the typing relation undecidable). Therefore, we think of elements x of the class $\{x \mid F(x)\}$ as pairs (x, P) where $P\ : ded\ F\ x$ is a proof that x is indeed in that class. We encode this in LF as follows:

$$\begin{array}{lll}
Class & : & (set \to prop) \to \mathtt{type} \\
celem & : & \{a : set\}\ ded\ F\ a \to Class\ F \\
cwhich & : & Class\ F \to set \\
cwhy & : & \{a : Class\ F\}\ ded\ (F\ (cwhich\ a))
\end{array}$$

$$\begin{array}{lll}
Elem & : & set \to type = [a]\ \ Class\ [x]\ \ x\ \in\ a \\
elem & : & \{a : set\}\ ded\ a\ \in\ A \to elem\ A = [a]\,[p]\ \ celem\ a\ p \\
which & : & elem\ A \to set = [a]\ \ cwhich\ a \\
why & : & \{a : elem\ A\}\ \ ded\ (which\ a)\ \in\ A = [a]\ \ cwhy\ a
\end{array}$$

$Class\ F$ encodes $\{x|F(x)\}$, $celem\ x\ P$ produces an element of a class, and $cwhich\ x$ and $cwhy\ x$ return the set and its proof. The remaining declarations specialise these notions to the classes $\{x|x \in A\}$.

To axiomatise these, we use the additional axiom

$$eqwhich \quad : \quad ded\ cwhich\ (elem\ X\ P)\ \doteq\ X$$

as well as the following axiom for proof irrelevance

$$proofirrel : \{f : ded\ G \to Class\ A\}\, ded\ cwhich\ (f\ P)\ \doteq\ cwhich\ (f\ Q)$$

which formalises the fact that two sets are equal if they only differ in a proof.

4.3.2. *Typed operations.* Using the types $Elem\ A$, we can now lift all the basic untyped operations introduced above to the typed level. In particular, we define typed quantifiers $\forall^*$, $\exists^*$, typed equality $\doteq^*$, typed function spaces $\Rightarrow^*$ and booleans $bool$ as follows:

(1) We define *typed quantifiers* such as $\forall^* : (elem\ A \to prop) \to prop$. In higher-order logic (Church 1940), such typed quantification can be defined easily using abstraction over the booleans. This is not possible in ZFC because the type $prop$ is not a set itself, that is, we have $prop : \mathtt{type}$ and not $prop : set$. If we committed to classical logic, we could use the set $bool : set$ from below.
A natural solution would be relativisation as in $\forall^*\ F := \forall[x]\, x \in A \Rightarrow'\ F\ x$ for $F : elem\ A \to prop$. However, an attempt to define typed quantification like this meets a subtle difficulty since F in $\forall^*\ F$ only needs to be defined for elements of A, whereas in $\forall[x]\, x \in A \Rightarrow'\ F\ x$, it must be defined for all sets even though $F\ x$ is intended to be ignored if $x \notin A$. Therefore, we use sequential connectives:

$$\begin{aligned} \forall^*\ &: (Elem\ A \to prop) \to prop\ =\ [F]\ \big(\forall[x]\, x \in A \Rightarrow'\ [p]\, (F\ (elem\ x\ p))\big) \\ \exists^*\ &: (Elem\ A \to prop) \to prop\ =\ [F]\ \big(\exists[x]\, x \in A \wedge'\ [p]\, (F\ (elem\ x\ p))\big) \end{aligned}$$

It is easy to derive the expected introduction and elimination rules for $\forall^*$ and $\exists^*$.

(2) It is easy to define *typed equality*:

$$\doteq^*\ : Elem\ A \to Elem\ A \to prop\ =\ [a]\ [b]\ (which\ a) \doteq (which\ b)$$

It is easy to see that all rules for $\doteq$ can be lifted to $\doteq^*$.

(3) We can define *function types* in the expected way:

$$\begin{aligned} \Rightarrow^*\ &:\ set \to set \to set = [x]\ [y]\ \ Elem\ (x \Rightarrow y) \\ \lambda^*\ &:\ (Elem\ A \to Elem\ B) \to Elem\ (A \Rightarrow^*\ B)\ = \ldots \\ @^*\ &:\ Elem\ (A \Rightarrow^*\ B) \to Elem\ A \to Elem\ B \\ &= [F]\ [x]\ \ elem\ (@\ (which\ F)\ (which\ x))\ (funcE\ (why\ F)\ (why\ x)) \\ beta\ &:\ ded\ \ (@^*\ (\lambda^*\ [x]\, F\ x)\ A)\ \doteq^*\ F\ A\ = \ldots \\ eta\ &:\ ded\ \ (\lambda^*\ [x]\, (@^*\ F\ x))\ \doteq^*\ F\ = \ldots \end{aligned}$$

We omit the quite involved definitions and only mention the fact that the typed quantifiers, and thus the sequential connectives, are required in the definitions.

(4) We introduce the set $\{\varnothing, \{\varnothing\}\}$ of booleans and derive some important operations for them. In particular, these are the constants 0 and 1, a supremum operation on families

of booleans, a variant of if-then-else where the then-branch (else-branch) may depend on the truth (falsity) of the condition, and a reflection function mapping propositions to booleans.

$$\begin{array}{ll}
bool & : set = uopair\ empty\ (singleton\ empty) \\
0 & : Elem\ bool = \ldots \\
1 & : Elem\ bool = \ldots \\
sup & : (Elem\ A \to Elem\ bool) \to Elem\ bool \\
ifte & : \{F : prop\}\,(ded\ F \to Elem\ A) \to (ded\ \neg\ F \to Elem\ A) \\
 & \ \to Elem\ A = \ldots \\
reflect & : Elem\ prop \to Elem\ bool
\end{array}$$

The definition of the supremum operation is only possible after proving that

$$\{\emptyset, \{\emptyset\}\} = \mathscr{P}\{\emptyset\},$$

which requires the use of excluded middle – in fact, it is equivalent to it. Similarly, $reflect$ and $ifte$ can only be defined in the presence of excluded middle. All other definitions in our formalisation of ZFC are also valid intuitionistically.

4.4. *Related work*

Several formalisations of set theory have been proposed and substantially developed. Most notable are the encodings of Tarski–Grothendieck set theory in Mizar (Trybulec and Blair 1985; Trybulec 1989) and of ZF in Isabelle (Paulson 1994; Paulson and Coen 1993). The most striking difference compared with our formalisation is that they employ sophisticated machine support with structured proof languages. Since there is no comparable machine support for Twelf, our encoding uses hand-written proof terms.

We chose LF because it permits a more elegant formalisation: the only primitive symbol we use is ∈, and we then use a description operator to introduce names for derived concepts. This differs from standard accounts of formalised mathematics and is in contrast to Mizar where primitive function symbols are used for singleton, unordered pair and union, and to Isabelle/ZF where primitive function symbols are used for empty set, power set, union, infinite set and replacement. But it corresponds more closely to mathematical practice, where the implicit use of a description operator is prevalent.

Our encoding depends crucially on dependent types. Description operators are also used in typed formalisations of mathematics such as HOL (Church 1940). They differ from ours by not taking a proof of unique existence as an argument. Consequently, they must assume the non-emptiness of all types and a global choice function. Other language features that are only possible in a dependently typed framework are sequential connectives and our $ifte$ construct. Connectives similar to our sequential ones are also used in PVS (Owre *et al.* 1992) and in de Nivelle (2010), albeit without proof terms occurring explicitly in formulas.

Moreover, using dependent types, we can recover typed reasoning as a derived notion. Our approach here is similar to that in Scunak (Brown 2006), and, in fact, our formalisation of classes and typed reasoning is inspired by the one used in Scunak. Scunak uses a variant

https://doi.org/10.1017/S0960129511000144 Published online by Cambridge University Press

$$
\begin{array}{lll}
con & ::= & c :: \tau \\
ax & ::= & a : \varphi \\
lem & ::= & l : \varphi \; proof \\
typedecl & ::= & (\alpha_1, \ldots, \alpha_n)t \\
types & ::= & (\alpha_1, \ldots, \alpha_n)t = \tau \\
\tau & ::= & \alpha \mid (\tau, \ldots, \tau)\; t \mid \tau \Rightarrow \tau \mid prop \\
term & ::= & x \mid c \mid term\; term \mid \lambda x :: \tau.term \\
\varphi & ::= & \varphi \Longrightarrow \varphi \mid \bigwedge x :: \tau.\varphi \mid term \equiv term \\
proof & ::= & \ldots
\end{array}
$$

Fig. 1. Isabelle grammar

of dependent type theory specifically developed for this purpose – the symbols *set*, *prop* and *Class*, and the axioms *eqwhich* and *proofirrel* are primitives of the type theory. This renders the formalisation much simpler, but at the price of using a less elegant framework.

A compromise between our encoding and Scunak's would be an extension of the LF framework. For example, the dependent sum type $\Sigma_{x:set}(ded\ F\ x)$ could be used instead of our *Class F*. Moreover, a variant of proof irrelevance is introduced in Lovas and Pfenning (2009) for LF, and this might make our encoding more elegant.

5. Isabelle and higher-order logic

5.1. *Preliminaries*

5.1.1. *Isabelle.* Isabelle is a logical framework and generic LCF-style interactive theorem prover based on polymorphic higher-order logic (Paulson 1994). We will only consider the core language of Isabelle here – the *Pure* logic and basic declarations – and omit the module system and the structured proof language. We gave a comprehensive formalisation of *Pure* and the Isabelle module system in Rabe (2010).

The grammar for Isabelle is given in Figure 1, which is a simplified version of the one given in Wenzel (2009).

An Isabelle theory is a list of: declarations of typed constants $c :: \tau$; axioms $a : \varphi$; lemmas $a : \varphi\ P$ where P proves φ; and n-ary type operators $(\alpha_1, \ldots, \alpha_n)t$, which may carry a definition in terms of the α_i. Definitions for constants can be introduced as special cases of axioms, and we consider base types as nullary type operators.

Types τ are formed from type variables α, type operator applications $(\tau_1, \ldots, \tau_n)t$, function types and the base type *prop* of propositions. Terms are formed from variables, constants, application and lambda abstraction. Propositions are formed from implication $\Longrightarrow$, universal quantification $\bigwedge$ at any type and equality on any type. Wenzel (2009) does not give a grammar for proofs but lists the inference rules; they are $\bigwedge$ introduction and elimination, $\Longrightarrow$ introduction and elimination, reflexivity and substitution for equality, β and η conversion and functional extensionality.

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 arguments.

5.1.2. *HOL.* The most advanced logic formalised in Isabelle is HOL (Nipkow *et al.* 2002). Isabelle/HOL is a classical higher-order logic with shallow polymorphism, non-empty types and choice operator (Church 1940; Gordon and Pitts 1993).

Isabelle/HOL uses the same types and function space as Isabelle, but it introduces a type *bool* for HOL-propositions (that is, 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, and the definite and indefinite description operators $some\ x : \tau.P$ and $the\ x : \tau.P$ for a predicate $P : \tau \Rightarrow bool$. Furthermore, HOL declares a polymorphic constant $undefined$ of any type and an infinite base type *ind*, which we omit in the following. Based on these primitives and their axioms, simply typed set theory is developed by purely definitional means.

Going beyond the Isabelle framework, Isabelle/HOL also supports Gordon/HOL-style type definitions using representing sets. A set A on the type τ is given by its characteristic function, that is, $A : \tau \Rightarrow bool$. An Isabelle/HOL type definition is of the form $(\alpha_1, \ldots, \alpha_n)\, t = A\ P$ 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 axiomatised to be isomorphic to the set A.

5.2. *Formalising Isabelle/HOL*

5.2.1. *Isabelle.* Our formalisation of Isabelle follows the one we gave in Rabe (2010). We declare an LF signature $Pure$ for the inner syntax of Isabelle, which declares symbols for all primitives that can occur in expressions. $Pure$ is given in Figure 2.

This yields a straightforward structural encoding function $\ulcorner - \urcorner$, which acts as described in Figure 3. Similar encodings are well known for LF, see, for example, Harper *et al.* (1993). The only subtlety arises for the case of polymorphic constant applications $c\ t_1\ \ldots\ t_n$ where the type of c contains type variables $\alpha_1, \ldots, \alpha_m$. Here we need to infer the types $\tau_1, \ldots, \tau_m$ at which c is applied, and put

$$\ulcorner c\ t_1\ \ldots\ t_n \urcorner = (c\ \ulcorner \tau_1 \urcorner\ \ldots\ \ulcorner \tau_m \urcorner)\ @\ \ulcorner t_1 \urcorner\ \ldots\ @\ \ulcorner t_n \urcorner.$$

Polymorphic axioms and lemmas occurring in proofs are treated accordingly. Finally, an Isabelle theory $S = \Sigma$ is represented as shown below, where $\ulcorner \Sigma \urcorner$ is defined declaration-wise according to Figure 4:

```
sig S = {
   include Pure
   ⌜Σ⌝
}
```

5.2.2. *Adequacy.* It is easy to show the adequacy of this encoding. For an Isabelle theory Σ, Isabelle types τ over Σ with type variables from $\alpha_1, \ldots, \alpha_m$ are in bijection with LF-terms $\ulcorner \tau \urcorner : tp$ in context $\alpha_1 : tp, \ldots, \alpha_m : tp$, and accordingly Isabelle terms $t :: \tau$ with LF-terms $\ulcorner t \urcorner : tm\, \ulcorner \tau \urcorner$, and Isabelle proofs P of φ with LF-terms $\ulcorner P \urcorner : \vdash\ \ulcorner \varphi \urcorner$.

$$
\begin{array}{llll}
\multicolumn{4}{l}{\texttt{sig}\ Pure\ =\ \{} \\
\quad tp & : & \texttt{type} & \\
\quad \Rightarrow & : & tp \to tp \to tp & \texttt{infix}\ 0 \\
\quad tm & : & tp \to \texttt{type} & \texttt{prefix}\ 0 \\
\quad \lambda & : & (tm\ A \to tm\ B) \to tm\ (A \Rightarrow B) & \\
\quad @ & : & tm\ (A \Rightarrow B) \to tm\ A \to tm\ B & \texttt{infix}\ 1000 \\
 & & & \\
\quad prop & : & tp & \\
\quad \bigwedge & : & (tm\ A \to tm\ prop) \to tm\ prop & \\
\quad \Longrightarrow & : & tm\ prop \to tm\ prop \to tm\ prop & \texttt{infix}\ 1 \\
\quad \equiv & : & tm\ A \to tm\ A \to tm\ prop & \texttt{infix}\ 2 \\
 & & & \\
\quad \vdash & : & tm\ prop \to \texttt{type} & \texttt{prefix}\ 0 \\
\quad \bigwedge I & : & (x : tm\ A \vdash (B\ x))\ \ \to \vdash \bigwedge([x]\, B\ x) & \\
\quad \bigwedge E & : & \vdash \bigwedge([x]\, B\ x)\ \ \to\ \ \{x : tm\ A\} \vdash (B\ x) & \\
\quad \Longrightarrow I & : & (\vdash A\ \ \to \vdash B)\ \ \to \vdash A \Longrightarrow B & \\
\quad \Longrightarrow E & : & \vdash A \Longrightarrow B\ \ \to \vdash A\ \ \to \vdash B & \\
\quad refl & : & \vdash X \equiv X & \\
\quad subs & : & \{F : tm\ A\ \ \to\ \ tm\ B\} \vdash X \equiv Y\ \ \to \vdash F\ X \equiv F\ Y & \\
\quad exten & : & (\{x : tm\ A\} \vdash (F\ x) \equiv (G\ x))\ \ \to \vdash \lambda F \equiv \lambda G & \\
\quad beta & : & \vdash (\lambda[x : tm\ A]\ F\ x)\ @\ X \equiv F\ X & \\
\quad eta & : & \vdash \lambda\ ([x : tm\ A]\ F\ @\ x) \equiv F & \\
\multicolumn{4}{l}{\}}
\end{array}
$$

Fig. 2. LF signature for Isabelle

Expression	Isabelle	LF
type	τ	$\ulcorner t \urcorner : tp$
term	$t :: \tau$	$\ulcorner t \urcorner : tm\ \ulcorner \tau \urcorner$
proof	P proving φ	$\ulcorner P \urcorner : \vdash \ulcorner \varphi \urcorner$
	containing type variables $\alpha_1, \ldots, \alpha_m$	in context $\alpha_1 : tp, \ldots, \alpha_m : tp$

Fig. 3. Encoding of expressions

Declaration	Isabelle	LF
type operator	$(\alpha_1, \ldots, \alpha_n)\ t$	$t : tp \to \ldots \to tp \to tp$
type definition	$(\alpha_1, \ldots, \alpha_n)\ t = \tau$	$t : tp \to \ldots \to tp \to tp$ $= [\alpha_1] \ldots [\alpha_n]\, \tau$
constant	$c :: \tau, \quad \alpha_1, \ldots, \alpha_m$ in τ	$c : tp \to \ldots \to tp \to tm\ \ulcorner \tau \urcorner$
axiom	$a : \varphi, \quad \alpha_1, \ldots, \alpha_m$ in τ	$a : tp \to \ldots \to tp \to \vdash \ulcorner \varphi \urcorner$
lemma	$l : \varphi\ P, \quad \alpha_1, \ldots, \alpha_m$ in φ, P	$l : tp \to \ldots \to tp \to \vdash \ulcorner \varphi \urcorner$ $= [\alpha_1] \ldots [\alpha_m]\ \ulcorner P \urcorner$

Fig. 4. Encoding of declarations

$$\begin{array}{lll}
\multicolumn{3}{l}{\texttt{sig}\ HOL\ =\ \{} \\
\multicolumn{3}{l}{\quad\texttt{include}\ Pure} \\
\quad bool & : & tp \\
\quad trueprop & : & tm\ bool \Rightarrow prop \\
\quad eps & : & tm\ (A \Rightarrow bool) \Rightarrow A \\
\quad \doteq & : & tm\ A \Rightarrow A \Rightarrow bool \\
\quad set & : & tp \to tp = [a]\, a \Rightarrow bool \\
\quad nonempty & : & (tm\ set\ A) \to \texttt{type} = \ldots \\
\quad typedef & : & \{s : tm\ set\ A\}\ nonempty\ s \to tp \\
\quad Rep & : & tm\ (typedef\ S\ P) \Rightarrow A \\
\quad Abs & : & tm\ A \Rightarrow (typedef\ (S\ :\ tm\ set\ A)\ P) \\
\}
\end{array}$$

Fig. 5. LF signature for HOL

5.2.3. *HOL.* Since HOL is an Isabelle theory, its LF-encoding follows immediately from the definition above. The fragment arising from translating some of the primitive declarations of HOL is given in the upper part of the signature *HOL* in Figure 5. For example, *eps* is the choice operator. The lower part gives some of the additional declarations needed to encode HOL-style type definitions. The central declaration is *typedef*, which takes a set S on the type A and a proof that S is non-empty and returns a new type, say T. *Rep* and *Abs* translate between A and T – see Wenzel (2009) for details.

5.3. *Interpreting Isabelle/HOL in ZFC*

We formalise the relation between Isabelle/HOL and ZFC by giving the two views $PureZFC$ and $HOLZFC$ from $Pure$ and HOL, respectively, to ZFC^+, as shown below:

$$\begin{tikzcd}
\text{Pure} \arrow[d, hook] \arrow[dr, "\text{PureZFC}"] & \\
\text{HOL} \arrow[r, "\text{HOLZFC}"] & \text{ZFC}^+
\end{tikzcd}$$

These formalise the standard set-theoretical semantics of higher-order logic.

ZFC^+ arises from ZFC by adding a global choice function

$$choice : \{A : Class\ nonempty\}\,(Elem\ (chwich\ A)),$$

which produces an element of a non-empty set A. This is stronger than the axiom of choice (which merely states the existence of such an element), but is required to interpret the choice operators of HOL.

5.3.1. *Isabelle* The general structure of the translation is given in Figure 6 and the view in Figure 7. Types are mapped to non-empty sets, terms are mapped to elements, in particular, propositions are mapped to booleans, and proofs of φ are mapped to proofs of $PureZFC(\varphi) \doteq^* 1$. These invariants are encoded (and guaranteed) by the assignments

Isabelle/HOL	ZFC
$\tau : tp$	$PureZFC(\tau) : Class\ nonempty.$
$t : tm\ \tau$	$PureZFC(t) : Elem\ (cwhich\ PureZFC(\tau)).$
$\varphi : tm\ prop$	$PureZFC(\varphi) : Elem\ (cwhich\ boolne).$
$P : \vdash\ \varphi$	$PureZFC(P) : ded\ PureZFC(\varphi) \doteq^* (bbne\ 1).$

Fig. 6. Isabelle/HOL Declarations in ZFC

$$
\begin{array}{llll}
\texttt{view}\ PureZFC & : Pure\ \to ZFC & = \{ & \\
\quad tp & := & Class\ nonempty & \\
\quad tm & := & elem & \\
\quad prop & := & boolne & \\
\quad \vdash & := & [x]\, ded\ x\ \doteq^* 1 & \\
\quad \Rightarrow & := & \Rightarrow^* & \\
\quad \lambda & := & \lambda^* & \\
\quad @ & := & @^* & \\
\quad \bigwedge & := & \forall^* & \\
\quad \Longrightarrow & := & \Rightarrow & \\
\quad \equiv & := & \doteq^* & \\
\vdots & & & \\
\} & & &
\end{array}
$$

Fig. 7. Interpreting Pure in ZFC

to *tp*, *tm*, *prop* and $\vdash$ in $PureZFC$. It is tempting to map Isabelle propositions to ZFC propositions rather than to booleans. However, in Isabelle, *prop* is a normal type and thus must be interpreted as a set. An alternative would be to map *prop* to a set representing intuitionistic truth values rather than classical ones, but we omit this for simplicity. (Because of our use of a standard model, we cannot expect completeness anyway.)

The case for terms t is a bit tricky. This is because τ is interpreted as an element of $Class\ nonempty$, so we first have to apply *cwhich* to obtain a set. Then we apply $Elem$ to this set to obtain the type of its elements. Similarly, *prop* cannot be mapped directly to $Elem\ bool$. Instead, we have to introduce $boolne : Class\ nonempty$, which couples *bool* to the proof that it is a non-empty set. Therefore, we also have to define the auxiliary functions $bbne : Elem\ bool \to Elem\ (cwhich\ boolne)$ and $bneb : Elem\ (cwhich\ boolne) \to Elem\ bool$ to convert back and forth. These technicalities indicate a drawback of our, otherwise perfectly natural, representation of classes. Different representations that separate the mapping of types to sets from the proofs of non-emptiness may prove more scalable, but would require a more sophisticated framework.

The remaining cases are straightforward. For example, $\Rightarrow$ must be mapped to a ZFC expression that takes two arguments of type $Class\ nonempty$ and returns another, that is, it must respect the invariants above.

$$
\begin{array}{l}
\texttt{view}\ HOLZFC : HOL \to ZFC = \{ \\
\quad \texttt{include}\ PureZFC \\
\quad \begin{array}{lll}
bool & := & bool \\
trueprop & := & [x]\,x \\
\doteq & := & \lambda^*([x](\lambda^*([y]bbne(reflect(x \doteq^* y))))) \\
eps & := & [f : Elem\ (A \Rightarrow^* bool)]\, ifte\ (nonempty\ (filter\ f)) \\
 & & \qquad ([p]\ (choice\ (elem\ (filter\ f)\ p))) \\
 & & \qquad ([p]\ choice\ A) \\
\vdots & & \\
typedef & := & [s : Elem\ (A \Rightarrow bool)]\,[p]\, celem\ (filter\ ([x]\, s\,@\,x \doteq^* 1))\ p \\
\vdots & &
\end{array} \\
\}
\end{array}
$$

Fig. 8. Interpreting HOL in ZFC

5.3.2. *HOL*. We obtain a view from HOL to ZFC similarly – a fragment of it is shown in Figure 8. HOL booleans are mapped to ZFC booleans so that *trueprop* is mapped to the identity. The choice operator *eps* is interpreted using $ifte$ and *choice*. Note that in the given Twelf terms, we elide some bookkeeping proof steps. The then-branch uses $elem\ (filter\ f)\ P$ to construct an element of $Class\ nonempty$, to which *choice* is then applied. In both cases, P must use the assumption p that the condition of the $ifte$-split is true.

$typedef\ s\ p$ is interpreted using $filter$ according to s. Thus, type definitions using sets on A are interpreted as subsets of A in the expected way. The proof p is used to obtain an element of $Class\ nonempty$.

5.4. *Related work*

Our formalisation of Isabelle is a special case of the one we gave in Rabe (2010), where we also covered the Isabelle module system. Together with the formalisation of HOL given here, we now cover interpretations of Isabelle locales in terms of Isabelle/HOL. This is interesting because if Isabelle locales are seen as logical theories and HOL as a foundation of mathematics, then interpretations can be seen as models.

Formalisations of HOL in logical frameworks have been given in Pfenning *et al.* (2003) using LF, and, of course, in Isabelle itself (Nipkow *et al.* 2002). Ours appears to be the first formalisation of Isabelle and HOL and the meta-relation between them. Moreover, we do not know of any other formalisations of HOL-style type definitions in a formal framework – even in the Isabelle/HOL formalisation, the type definitions are not expressed exclusively in terms of the Pure meta-language.

Our semantics of Isabelle/HOL does not quite follow the one given in Gordon and Pitts (1993), where individual models provide a set $\mathscr{U}$ of sets, and every type is interpreted as an element of $\mathscr{U}$. Models must provide further structure to interpret HOL type constructors, in particular, a choice function on $\mathscr{U}$. Our semantics can be seen as a single

https://doi.org/10.1017/S0960129511000144 Published online by Cambridge University Press

model where the set theoretical universe is used instead of $\mathcal{U}$. Consequently, our model is not a set itself, and thus not a model in the sense of Gordon and Pitts (1993), but every individual model in that sense is subsumed by ours.

Independently of our work, a similar semantics of Isabelle/HOL is given in Krauss and Schropp (2010). They translate Isabelle/HOL to Isabelle/ZF where the interpretation of Pure is simply the identity. Their semantics is given as a target-trusting implementation rather than being formalised in a framework. They also use the full set-theoretical universe and a global choice function. An important difference is the treatment of non-emptiness since they assume that interpretations for all type constructors are given that respect non-emptiness, so they can interpret all types as sets (which will always be non-empty) and only have to relativise universal quantifiers over types to quantifiers over non-empty sets. Our translation is more complicated in that respect because it uses *Class nonempty* to guarantee the non-emptiness.

6. Mizar and Tarski–Grothendieck set theory

6.1. *Preliminaries*

6.1.1. *Mizar.* At its core, Mizar (Trybulec and Blair 1985) is an implementation of classical first-order logic. However, it is designed to be used with a single theory, *viz.* set theory following the Tarski–Grothendieck axiomatisation (Tarski 1938; Bourbaki 1964) (TG). Consequently, Mizar is strongly influenced by its representation of TG. Like Isabelle, it includes a semi-automated theorem prover and a structured proof language.

Mizar/TG is notable for being the only major system for the formalisation of mathematics that is based on set theory. Types are only introduced as a means of efficiency and clarity, but not as a foundational commitment. Moreover, the Mizar Mathematical Library is one of the largest libraries of formalised mathematics containing over 50,000 theorems and 9,500 definitions.

Mizar's logic is an extension of classical first-order logic with second-order axiom schemes. The proof system is Jaskowski-style natural deduction (Jaśkowski 1934). Contrary to the LCF style implementations of HOL and to our ZFC, which try to use a small set of primitives, Mizar features a rich ontology of primitive mathematical objects, types and proof principles.

In particular, the type *set* of terms (that is, sets in Mizar/TG) can be refined using a complex type system, see, for example, Wiedijk (2007). The basic types are called *modes*, and while they are semantically predicate subsorts (that is, classes in Mizar/TG), they are technically primitive in Mizar. Modes can be further refined by *attributes*, which are predicates on a type. These two refinement relations generate a subtype relation between type expressions, called type *expansion*. Both modes and attributes may take arguments, which makes Mizar dependently typed. Mizar enforces the non-emptiness of all types, and all mode definitions and attribute applications induce the respective proof obligations.

The notion of typed first-order functions between types, called *functors*, is primitive. Function definitions may be implicit, in which case they induce proof obligations for well-definedness.

https://doi.org/10.1017/S0960129511000144 Published online by Cambridge University Press

$\mathit{Article}$	::=	$\mathit{Article\text{-}Name}^* \ \mathit{Text\text{-}Proper}$
$\mathit{Text\text{-}Proper}$	::=	$(\mathit{Block} \mid \mathit{Theorem})^*$
Block	::=	$\texttt{definition let } (x \texttt{ be } \vartheta)^* \ \mathit{Definition} \texttt{ end}$
$\mathit{Definition}$	::=	$\mathit{Mode} \mid \mathit{Functor} \mid \mathit{Attribute}$
Mode	::=	$\texttt{mode } M \texttt{ of } x_1,\ldots,x_n \texttt{ is } \vartheta$
	$\mid$	$\texttt{mode } M \texttt{ of } x_1,\ldots,x_n \to \vartheta \texttt{ means } \alpha$
		$\texttt{existence } \mathit{proof}$
$\mathit{Attribute}$	::=	$\texttt{attr } x \texttt{ is } (x_1,\ldots,x_n)V \texttt{ means } \alpha$
$\mathit{Functor}$	::=	$\texttt{func } f(x_1,\ldots,x_n) \texttt{ equals } t$
	$\mid$	$\texttt{func } f(x_1,\ldots,x_n) \to \vartheta \texttt{ means } \alpha$
		$\texttt{existence } \mathit{proof} \texttt{ uniqueness } \mathit{proof}$
$\mathit{Theorem}$	::=	$\texttt{theorem } T : \alpha \ \mathit{proof}$
t	::=	$x \mid f(t_1,\ldots,t_n)$
α	::=	$t \texttt{ is } \vartheta \mid t \texttt{ in } t \mid \alpha \& \alpha \mid \texttt{not } \alpha \mid t = t$
	$\mid$	$\texttt{for } x \texttt{ be } \vartheta \texttt{ holds } \alpha$
ϑ	::=	$\mathit{Adjective}^* \ \mathit{Radix}$
$\mathit{Adjective}$	::=	$(t_1,\ldots,t_n)V \mid \texttt{non } (t_1,\ldots,t_n)V$
Radix	::=	$M \texttt{ of } t_1,\ldots,t_n$
proof	::=	$\ldots$

Fig. 9. Mizar Grammar

This expressivity makes theorems and proofs written in Mizar relatively easy to read, but makes it hard to represent Mizar itself in a logical framework. We will use the grammar given in Figure 9, which is a substantially simplified variant of the one given in Mizar (2009). Here … and * denote possibly empty repetition.

A Mizar article starts with one import clause for every kind of declaration to import from other articles. Instead, we only permit cumulative imports of whole articles. This is followed by a list of definitions and theorems. We only permit mode, functor and attribute definitions. Predicate definitions and schemes could be added easily.

All three kinds of definitions introduce a new symbol, which takes a list of typed term arguments x_i. The type ϑ_i of x_i must be given by a `let` declaration or defaults to the type *set*. Mode definitions define M `of` $x_1, \ldots, x_n$ either explicitly as the type $\vartheta(x_1, \ldots, x_n)$ or implicitly as the type of sets *it* of type ϑ satisfying $\alpha(it, x_1, \ldots, x_n)$. In the latter case, non-emptiness must be proved. Similarly, functor definitions define $f(x_1, \ldots, x_n)$ either explicitly as $t(x_1, \ldots, x_n)$ or implicitly as the object *it* of type ϑ satisfying $\alpha(it, x_1, \ldots, x_n)$. In the latter case, well-definedness, that is, existence and uniqueness, must be proved. Finally, attribute definitions define $(x_1, \ldots, x_n)V$ as the unary predicate on the type of x given by $\alpha(x, x_1, \ldots, x_n)$.

Terms t and formulas α are formed in the usual way, and we omit the productions for *proof* terms. `in` and `is` are used for elementhood in a set or a type, respectively. Finally, types are formed by providing a list of possibly negated adjectives on a mode. In Mizar, these types must be proved to be non-empty before they can be used, which we will omit here.

$$
\begin{array}{llll}
\multicolumn{4}{l}{\texttt{sig}\ \mathit{Mizar}\ =\ \{} \\
\quad \mathit{tp} & : & \texttt{type} & \\
\quad \mathit{prop} & : & \texttt{type} & \\
\quad \mathit{proof} & : & \mathit{prop} \to \texttt{type} & \texttt{prefix}\ 0 \\
\quad \mathit{be} & : & \mathit{tp} \to \texttt{type} & \\
\quad \mathit{set} & : & \mathit{tp} & \\
\quad \mathit{is} & : & \mathit{be}\ T \to \mathit{tp} \to \mathit{prop} & \texttt{infix}\ 30 \\
\quad \mathit{in} & : & \mathit{be}\ T \to \mathit{be}\ T' \to \mathit{prop} & \texttt{infix}\ 30 \\
\quad \mathit{not} & : & \mathit{prop} \to \mathit{prop} & \texttt{prefix}\ 20 \\
\quad \mathit{and} & : & \mathit{prop} \to \mathit{prop} \to \mathit{prop} & \texttt{infix}\ 10 \\
\quad \mathit{eq} & : & \mathit{be}\ T \to \mathit{be}\ T' \to \mathit{prop} & \texttt{infix}\ 10 \\
\quad \vdots & & & \\
\quad \mathit{for} & : & (\mathit{be}\ T \to \mathit{prop}) \to \mathit{prop} & \\
\quad \vdots & & & \\
\quad \mathit{func} & : & \{f : \mathit{be}\ T \to \mathit{prop}\}(\mathit{proof}\ \mathit{ex}\ [x]\ \ f\ x) \to & \\
 & & \mathit{proof}\ \mathit{for}\ [x]\ \mathit{for}\ [y]\ (f\ x\ \mathit{and}\ f\ y)\ \mathit{implies}\ x\ \mathit{eq}\ y \to \mathit{be}\ T & \\
\quad \mathit{func}_{\mathit{prop}} & : & \{F\}\,\{\mathit{Ex}\}\,\{\mathit{Unq}\}\ \mathit{proof}\ F\ (\mathit{func}\ F\ \mathit{Ex}\ \mathit{Unq}) & \\
\quad \mathit{attr} & : & \mathit{tp} \to \texttt{type} = [t]\ \ (\mathit{be}\ t \to \mathit{prop}) & \\
\quad \mathit{adjective} & : & \{t : \mathit{tp}\}\ \ \mathit{attr}\ t \to \mathit{tp} & \\
\quad \mathit{adjI} & : & \{x : \mathit{be}\ X\}\,(\mathit{proof}\ \ A\ x) \to \mathit{be}\ (\mathit{adjective}\ X\ \ A) & \\
\quad \mathit{adjE} & : & \{x : \mathit{be}\ (\mathit{adjective}\ X\ \ A)\}\ \ \mathit{be}\ X & \\
\quad \mathit{adjE}' & : & \{x : \mathit{be}\ (\mathit{adjective}\ X\ \ A)\}\,\mathit{proof}\ \ A\ (\mathit{adjE}\ x) & \\
\quad \vdots & & & \\
\} & & &
\end{array}
$$

Fig. 10. LF Signature for Mizar

6.1.2. *Tarski–Grothendieck set theory.* TG is similar to ZFC but uses Tarski's axiom asserting that for every set there is a universe containing it. It implies the axioms of infinity, choice, power set and large cardinals. Mizar/TG is defined in the Mizar article `Tarski` (Trybulec 1989), which contains primitives for elementhood, singleton, unordered pair, union, the Fraenkel scheme and the Tarski axiom, as well as a definition of ordered pairs following Kuratowski.

6.2. *Formalising Mizar and TG set theory*

6.2.1. *Mizar.* The LF signature that encodes Mizar's logic is given in Figure 10, where we omit the declarations of definable constants, such as equivalence, *iff* and the existential quantifier *ex*. The general form of the encoding of Mizar expressions in LF is given in Figure 11. Mizar types, formulas and proofs of F are represented as LF terms of the types *tp*, *prop* and *proof* $\ulcorner F \urcorner$, respectively. The judgment *expand* encodes Mizar's type expansion relation.

Mizar's use of a type system within an untyped foundation is hard to represent in a logical framework. We mimic it by using an auxiliary type constructor *be* with the intended meaning that $t : \mathit{be}\ T$ encodes a Mizar term t of type T. Consequently, if T

https://doi.org/10.1017/S0960129511000144 Published online by Cambridge University Press

Expression	Mizar	LF
type	ϑ	$\ulcorner\vartheta\urcorner : tp$
formula	α	$\ulcorner\alpha\urcorner : prop$
proof	P proving α	$\ulcorner P\urcorner : proof\ \ulcorner\alpha\urcorner$
typed term	t *be* ϑ	$\ulcorner t\urcorner : be\ \ulcorner\vartheta\urcorner$
type expansion	ϑ expands to ϑ'	$expand\ \ulcorner\vartheta\urcorner\ \ulcorner\vartheta'\urcorner$ inhabited

Fig. 11. Encoding of expressions

Mizar	LF
`let` x_i `be` ϑ_i	
`mode` M `of` $x_1,\dots,x_n$ `is` ϑ	$M : \{x_1 : be\ \ulcorner\vartheta_1\urcorner\} \dots \{x_n : be\ \ulcorner\vartheta_n\urcorner\}\ tp$ $= [x_1] \dots [x_n]\ \ulcorner\vartheta\urcorner$
`mode` M `of` $x_1,\dots,x_n \to \vartheta$ `means` α `existence` P	$M : \{x_1 : be\ \ulcorner\vartheta_1\urcorner\} \dots \{x_n : be\ \ulcorner\vartheta_n\urcorner\}\ tp$ $= [x_1] \dots [x_n]\ mode\ ([it]\ \ulcorner\alpha\urcorner)\ \ulcorner P\urcorner$
`func` $f(x_1,\dots,x_n)$ `equals` t t expands to ϑ	$f : \{x_1 : be\ \ulcorner\vartheta_1\urcorner\} \dots \{x_n : be\ \ulcorner\vartheta_n\urcorner\}\ be\ \ulcorner\vartheta\urcorner$ $= [x_1] \dots [x_n]\ \ulcorner t\urcorner$
`func` $f(x_1,\dots,x_n) \to \vartheta$ `means` α `existence` P `uniqueness` Q	$f : \{x_1 : be\ \ulcorner\vartheta_1\urcorner\} \dots \{x_n : be\ \ulcorner\vartheta_n\urcorner\}\ be\ \ulcorner\vartheta\urcorner$ $= [x_1] \dots [x_n]\ func\ ([it]\ \ulcorner\alpha\urcorner)\ \ulcorner P\urcorner\ \ulcorner Q\urcorner$
`let` x `be` ϑ `attr` x `is` $(x_1,\dots,x_n)V$ `means` α	$V : \{x_1 : be\ \ulcorner\vartheta_1\urcorner\} \dots \{x_n : be\ \ulcorner\vartheta_n\urcorner\}\ attr\ \ulcorner\vartheta\urcorner$ $= [x_1] \dots [x_n]\ ([x]\ \ulcorner\alpha\urcorner)$
`theorem` $T : \alpha\ P$	$T : proof\ \ulcorner\alpha\urcorner = \ulcorner P\urcorner$

Fig. 12. Encoding of declarations

expands to T', terms of type T must be explicitly cast to obtain terms of type T' by applying *cast*.

Attributes on a type ϑ are represented as LF terms of type *attr* ϑ. In effect, they are represented as LF functions *be* $\vartheta \to$ *prop*. A type $\vartheta = A_1\ \dots\ A_m\ R$ is encoded as *adjective* $(\dots(\mathit{adjective}\ R\ A_m)\ \dots\ )\ A_1$. Attributes $A = (t_1,\dots,t_n)V$ are encoded as $V\ \ulcorner t_1\urcorner\ \dots\ \ulcorner t_n\urcorner$. Finally, types M `of` $t_1,\dots,t_n$ (radix types in Mizar) are encoded as $M\ \ulcorner t_1\urcorner \dots \ulcorner t_n\urcorner$.

To this, we add LF constant declarations that represent the primitive formula and proof constructors of Mizar's first-order logic. For formulas and proofs, this is straightforward, and the only subtlety is to identify exactly which constructors are primitive. For example, *or* and *imp* are defined using *and* and *not*. We omit the constructors for type expansion. This induces an encoding of Mizar terms, types, formulas and proofs as LF terms. The only remaining subtlety is that applications of *cast* must be inserted whenever the well formedness of a type depends on the type expansion relations.

Then we can represent Mizar declarations according to Figure 12. Explicit functor and mode definitions are easily represented as defined LF constants. Implicit definitions are represented using special constants *func* and *mode*. *func* $([x : be\ \vartheta]\ \alpha\ x)$ *ex unq* encodes the uniquely existing object of type ϑ that satisfies α. Like δ in our ZFC encodings, it takes proofs of existence and uniqueness as arguments. *mode* $([x : be\ \vartheta]\ \alpha\ x)\ P$ encodes the

https://doi.org/10.1017/S0960129511000144 Published online by Cambridge University Press

necessarily non-empty subtype of ϑ containing the objects satisfying α. Attribute definitions are encoded easily. In all three cases, the arguments $x_1, \ldots, x_n$ of Mizar functors, modes and attributes are represented directly as LF arguments. Finally, theorems are encoded in the same way as for Isabelle.

Finally, we can encode a Mizar article $Art_1, \ldots, Art_n\ TP$ in file A as the following LF signature, where the *Text-Proper* part TP is encoded declaration-wise:

$$\begin{array}{l}
\texttt{sig}\ A\ =\ \{ \\
\quad \texttt{include}\ Mizar \\
\quad \texttt{include}\ Art_1 \\
\quad \vdots \\
\quad \texttt{include}\ Art_n \\
\quad \ulcorner \mathrm{TP} \urcorner \\
\}
\end{array}$$

6.2.2. *Adequacy*. Intuitively, our Mizar encoding should be adequate in the sense that Mizar articles that stay within our simplified grammar are well formed in Mizar if and only if their encoding is well formed in LF.

We cannot state or even prove the adequacy because there is no reference semantics of Mizar that would be rigorous and complete enough for that. This is partially due to the fact that Mizar is justified more through mathematical intuition than through a formal semantics.

6.2.3. *TG set theory*. The encoding of TG set theory given in Figure 13 is fairly straightforward. The use of LF's $\{\}$ binder for Mizar's axiom schemes is the only subtlety. The definitions for *singleton*, *uopair* and *union* are given using *func*, and their existence and uniqueness conditions are stated as axioms. We only give the case for *singleton*. The Tarski axiom is easy to encode but requires some auxiliary definitions.

6.3. *Interpreting Mizar/Tarski in ZFC*

In a similar way to the interpretation of Isabelle/HOL in ZFC, we give corresponding views for Mizar (the view from $Tarski$ to ZFC is dashed because it is partial as it omits the Tarski axiom, which goes beyond ZFC):

$$\begin{tikzcd}
Mizar \arrow[d, hook] \arrow[dr, "MizarZFC"] & \\
Tarski \arrow[r, dashed, "TarskiSem"] & ZFC
\end{tikzcd}$$

6.3.1. *Mizar*. The general idea of the interpretation of Mizar in ZFC is given in Figure 14. In particular, a type ϑ is interpreted as a unary predicate (the intensional description of ϑ), and the auxiliary type *be* ϑ as the class of sets in ϑ (the extensional description of ϑ).

https://doi.org/10.1017/S0960129511000144 Published online by Cambridge University Press

$$
\begin{array}{lll}
\multicolumn{3}{l}{\texttt{sig}\ \mathit{Tarski}\ =\ \{} \\
\quad \texttt{include}\ \mathit{Mizar} & & \\
\quad \vdots & & \\
\quad \mathit{singleton}_{ex} & : & \{y : \mathit{be\ set}\}\ \mathit{proof\ ex}\ [\mathit{it} : \mathit{be\ set}]\ (\mathit{for}\ [x : \mathit{be\ set}] \\
 & & (x\ \mathit{in\ it})\ \mathit{iff}\ (x\ \mathit{eq}\ y)) \\
\quad \mathit{singleton}_{unq} & : & \{y\}\mathit{proof\ for}\ [\mathit{it}]\ \mathit{for}\ [\mathit{it}']\ (\mathit{for}\ [x]\ (x\ \mathit{in\ it\ iff}\ x\ \mathit{eq}\ y) \\
 & & \mathit{and\ for}\ [x]\ \ (x\ \mathit{in\ it}'\ \mathit{iff}\ x\ \mathit{eq}\ y))\ \mathit{implies\ it\ eq\ it}' \\
\quad \mathit{singleton} & : & \mathit{be\ set} \to \mathit{be\ set} = [y]\ \mathit{func}\ ([\mathit{it}]\ \mathit{for}\ [x]\ x\ \mathit{in\ it\ iff}\ x\ \mathit{eq}\ y) \\
 & & (\mathit{singleton}_{ex}\ y)\ (\mathit{singleton}_{unq}\ y) \\
\quad \vdots & & \\
\quad \mathit{fraenkel} & : & \{A : \mathit{be\ set}\}\ \{P : \mathit{be\ set} \to \mathit{be\ set} \to \mathit{prop}\} \\
 & & \mathit{proof}\ (\mathit{for}\ [x : \mathit{be\ set}]\ \ \mathit{for}\ [y : \mathit{be\ set}]\ \ \mathit{for}\ [z : \mathit{be\ set}] \\
 & & ((P\ x\ y)\ \mathit{and}\ (P\ x\ z))\ \mathit{implies}\ y\ \mathit{eq}\ z) \to \mathit{proof}\ (\mathit{ex}\ [X] \\
 & & \mathit{for}\ [x]\ \ ((x\ \mathit{in}\ X)\ \mathit{iff}\ (\mathit{ex}\ [y]\ \ y\ \mathit{in}\ A\ \mathit{and}\ (P\ y\ x)))) \\
\quad \vdots & & \\
\quad \mathit{subset}_{closed} & : & \{m\}\ \ \mathit{prop} = [m]\ \ \mathit{for}\ [x]\ \ (\mathit{for}\ [y] \\
 & & (((\ x\ \mathit{in}\ m)\ \mathit{and}\ (y\ \subseteq\ x))\ \mathit{implies}\ (y\ \mathit{in}\ m))) \\
\quad \mathit{powerset}_{closed} & : & \{m\}\ \ \mathit{prop} = [m]\ \ \mathit{for}\ [x]\ \ (x\ \mathit{in}\ m\ \mathit{implies}\ (\mathit{ex}\ [z]\ \ z\ \mathit{in}\ m \\
 & & \mathit{and}\ (\mathit{for}\ [y]\ \ y\ \subseteq\ x\ \mathit{implies}\ y\ \mathit{in}\ z))) \\
\quad \mathit{tarski\_ax} & : & \mathit{proof\ for}\ [n]\ \ (\mathit{ex}\ [m]\ \ (\\
 & & n\ \mathit{in}\ m\ \mathit{and}\ \mathit{subset}_{closed}\ m\ \mathit{and} \\
 & & \mathit{powerset}_{closed}\ m\ \mathit{and\ for}\ [x] \\
 & & (x\ \subseteq\ m\ \mathit{implies}\ ((\mathit{isomorphic}\ x\ m)\ \mathit{or}\ x\ \mathit{in}\ m)))) \\
\quad \vdots & & \\
\} & &
\end{array}
$$

Fig. 13. Encoding TG set theory

Mizar	ZFC
$\vartheta : tp$	$MizarZFC(\vartheta) : set \to prop$
$\alpha : prop$	$MizarZFC(\alpha) : prop$
$P : proof\ \alpha$	$MizarZFC(P) : ded\ MizarZFC(P)$
$\alpha : be\ \vartheta$	$MizarZFC(\alpha) : Class\ MizarZFC(\vartheta)$

Fig. 14. Mizar/TG declarations in ZFC

Technically, we should interpret types as non-empty predicates, that is, as pairs of a predicate and an existence proof. We avoid doing this because it would complicate the encoding even more than in the case of Isabelle/HOL. This is possible because no part of our restricted Mizar language relies on the non-emptiness of type.

Type expansion is interpreted as a subclass relationship, and the interpretation of *cast* maps a set to itself, but treated as an element of a different class. This is formalised by the first declarations in the view *MizarZFC* in Figure 15.

$$
\begin{array}{lll}
\multicolumn{3}{l}{\texttt{view}\ MizarZFC : Mizar \to ZFC = \{} \\
\quad tp & := & set \to prop \\
\quad prop & := & prop \\
\quad proof & := & ded \\
\quad be & := & [f]\ Class\ f \\
\quad set & := & [x]\,\top \\
\quad is & := & [a]\,[F]\,F\ (cwhich\ a) \\
\quad in & := & [a]\,[b]\,(cwhich\ a)\ \in\ (cwhich\ b) \\
\quad \vdots & & \\
\quad func & := & [F]\,[EX]\,[UNQ]\,\delta\ F\ (andI\ EX\ UNQ) \\
\quad mode & := & [F : Class\ A \to prop]\,[EX]\,([x]\,(A\ x)\ \wedge'\ [p]\,F\ (celem\ x\ p)) \\
\quad adjective & := & [P : set \to prop]\,[Q : Class\ T \to prop] \\
 & & \quad [x]\,(P\ x)\wedge'\ [p]\,(Q\ (celem\ x\ p)) \\
\quad \vdots & & \\
\}
\end{array}
$$

Fig. 15. Interpreting *Mizar* in *ZFC*

func is interpreted using the description operator from ZFC, and the interpretation of *mode* is trivial.

Finally, attributed modes *adjective* ϑ A are interpreted using the conjunction of the interpretations P : *set* → *prop* of ϑ and Q : *Class* P → *prop* of A.

Note how sequential conjunction is needed to use the truth of $P\ x$ in the second conjunct. This is necessary because in Mizar A only has to be defined for terms of type ϑ, which corresponds to Q only being applicable to sets satisfying P.

We omit the straightforward but technical remaining cases for formula and proof constructors.

6.3.2. *TG.* The view *TarskiZFC* from *Tarski* to *ZFC* is straightforward, and we omit the details. However, the view is only partial because it omits the Tarski axiom.

Partial views in LF simply omit cases. Consequently, their homomorphic extensions are partial functions. For our view, this means that no definition or theorem that depends on the Tarski axiom can be translated to ZFC. This is more harmful than it sounds because the Tarski axiom is used in Mizar to prove the existence of power set, infinity and choice, so almost all definitions depend on it.

However, we have already designed an elegant extension of the notion of Twelf views in Dumbrava and Rabe (2010) that solves this problem as it enables us make *TarskiZFC* undefined for the Tarski axiom, but map Mizar's theorems of power set, infinity and choice, which depend on the Tarski axiom, to their counterparts in ZFC. We say that power set, infinity and choice are *recovered* by the view. Then Mizar expressions that are stated in terms of the recovered constants can still be translated to ZFC, and the preservation of truth is still guaranteed. With this amendment, most theorems in the Mizar library can be translated. Only theorems that directly appeal to the Tarski axiom remain untranslatable, and that is intentional because they are likely to be unprovable over ZFC.

https://doi.org/10.1017/S0960129511000144 Published online by Cambridge University Press

6.4. *Related work*

Mizar is notorious for being impenetrable as a logic, and previous work has focused on making the syntax and semantics of Mizar more accessible. The main source of complexity is the type system.

Wiedijk (2007) gives a comprehensive account of the syntax and semantics of the Mizar type system. It interprets types as predicates in the same way as we have done here. A translation to first-order logic is given that is similar in spirit to our translation to ZFC. An alternative approach using type algebras was given in Bancerek (2003).

Urban (2003) gives a translation of Mizar into TPTP-based first-order logic, which also interprets types as predicates.

7. Conclusions and future work

We have represented three foundations of mathematics and two translations between them in a formal framework, namely Twelf. The most important feature is that the well-definedness and soundness of the translations are verified statically and automatically by the Twelf type checker. In particular, the LF type system guarantees that the translation functions preserve provability. Our work is the first systematic case study of statically verified translations between foundations.

Our foundations are ZFC, Mizar's Tarski–Grothendieck set theory (TG) and Isabelle's higher-order logic (HOL). We chose ZFC as the most widespread foundation of non-formalised mathematics, and our formalisation stays notably close to textbook developments of ZFC. (However, we have had to add a global choice function for both Isabelle/HOL and for Mizar/TG.) We chose Isabelle/HOL and Mizar because they are two of the most advanced foundations of formalised mathematics in terms of library size and (semi-)automated proof support. They are also foundationally very different – higher-order logic and untyped set theory, respectively – and represent the whole spectrum of foundations. Moreover, our formalisations make the foundational assumptions of these systems explicit, and thus contribute to their documentation and systematic comparison.

We have formalised translations from Isabelle/HOL and Mizar/TG into ZFC as indicated below:

$$\begin{tikzcd}
Mizar \arrow[d, hook] \arrow[dr, dashed] & & Pure \arrow[dl] \arrow[d, hook] \\
Tarski \arrow[r, dashed] & ZFC^{+} & HOL \arrow[l]
\end{tikzcd}$$

These translations can be seen as giving two foundations used in formalised mathematics a semantics in terms of the foundation dominant in traditional mathematics. Actually, the translation from Mizar/TG to ZFC^{+} is only partial because the former is stronger than the latter, but this is not a serious concern, as we discussed in Section 6.3. We did not give the inverse translation from ZFC to Mizar/TG, but that would be straightforward.

However, a corresponding translation from ZFC to Isabelle/HOL remains a challenge (translations such as the one in Aczel (1998) would not be inverse to ours).

Future work will focus on two research directions:

(1) We will formalise more foundations and translations. This is an on-going effort in the LATIN project (Kohlhase *et al.* 2009), which will provide a large library of statically verified foundation translations and for which this work provides the theoretical basis and seed library. Examples of further systems are Coq (Coquand and Huet 1988) and PVS (Owre *et al.* 1992).

(2) A major drawback of statically verified translations is that the extracted translation functions cannot be directly applied to the libraries of the foundations, since they must first be represented in the foundational framework. This is a conceptually trivial, but in practice requires a long-term research effort, which is still under way.

References

Aczel, P. (1998) On Relating Type Theories and Set Theories. In: Altenkirch, T., Naraschewski, W. and Reus, B. (eds.) *Types for Proofs and Programs* 1–18.

Bancerek, G. (2003) On the structure of Mizar types. *Electronic Notes in Theoretical Computer Science* **85** 69–85.

Bourbaki, N. (1964) Univers. In: *Séminaire de Géométrie Algébrique du Bois Marie – Théorie des topos et cohomologie étale des schémas*, Springer-Verlag 185–217.

Brown, C. (2006) Combining Type Theory and Untyped Set Theory. In: Furbach, U. and Shankar, N. (eds.) International Joint Conference on Automated Reasoning. *Springer-Verlag Lecture Notes in Computer Science* **4130** 205–219.

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

Constable, R. *et al.* (1986) *Implementing Mathematics with the Nuprl Development System*, Prentice-Hall.

Coquand, T. and Huet, G. (1988) The Calculus of Constructions. *Information and Computation* **76** (2/3) 95–120.

de Bruijn, N. (1970) The Mathematical Language AUTOMATH. In: Laudet, M. (ed.) Proceedings of the Symposium on Automated Demonstration. *Springer-Verlag Lecture Notes in Computer Science* **25** 29–61.

de Nivelle, H. (2010) Classical Logic with Partial Functions. In: Giesl, J. and Hähnle, R. (eds.) Automated Reasoning. *Springer-Verlag Lecture Notes in Computer Science* **6173** 203–217.

Dumbrava, S. and Rabe, F. (2010) Structuring Theories with Partial Morphisms, Workshop on Abstract Development Techniques.

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

Fraenkel, A. (1922) The notion of ‘definite’ and the independence of the axiom of choice.

Gordon, M. (1988) HOL: A Proof Generating System for Higher-Order Logic. In: Birtwistle, G. and Subrahmanyam, P. (eds.) *VLSI Specification, Verification and Synthesis*, Kluwer-Academic Publishers 73–128.

Gordon, M., Milner, R. and Wadsworth, C. (1979) Edinburgh LCF: A Mechanized Logic of Computation. *Springer-Verlag Lecture Notes in Computer Science* **78**.

https://doi.org/10.1017/S0960129511000144 Published online by Cambridge University Press

Gordon, M. and Pitts, A. (1993) The HOL Logic. In: Gordon, M. and Melham, T. (eds.) *Introduction to HOL, Part III*, Cambridge University Press 191–232.

Hales, T. (2003) The flyspeck project. (Available at `http://code.google.com/p/flyspeck/`.)

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

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

Harrison, J. (1996) HOL Light: A tutorial introduction. In: Srivas, M. and Camilleri, A. (eds.) Proceedings of the First International Conference on Formal Methods in Computer-Aided Design (FMCAD'96). *Springer-Verlag Lecture Notes in Computer Science* **1166** 265–269.

Hurd, J. (2009) OpenTheory: Package Management for Higher Order Logic Theories. In: Reis, G. D. and Théry, L. (eds.) *Programming Languages for Mechanized Mathematics Systems*, ACM 31–37.

Iancu, M. and Rabe, F. (2010) Formalizing Foundations of Mathematics, LF Encodings. (Available at `https://latin.omdoc.org/wiki/FormalizingFoundations`.)

Jaśkowski, S. (1934) On the rules of suppositions in formal logic. *Studia Logica* **1** 5–32.

Keller, C. and Werner, B. (2010) Importing HOL Light into Coq. In: Kaufmann, M. and Paulson, L. (eds.) Proceedings Interactive Theorem Proving, ITP 2010. *Springer-Verlag Lecture Notes in Computer Science* **6172** 307–322.

Klein, G., Nipkow, T. and Paulson, L. (2004) Archive of Formal Proofs. (Available at `http://afp.sourceforge.net/`.)

Kohlhase, M., Mossakowski, T. and Rabe, F. (2009) The LATIN Project. (Available at `https://trac.omdoc.org/LATIN/`.)

Krauss, A. and Schropp, A. (2010) A Mechanized Translation from Higher-Order Logic to Set Theory. In: Kaufmann, M. and Paulson, L. (eds.) Proceedings Interactive Theorem Proving, ITP 2010. *Springer-Verlag Lecture Notes in Computer Science* **6172** 323–338.

Landau, E. (1930) *Grundlagen der Analysis*, Akademische Verlagsgesellschaft.

Lovas, W. and Pfenning, F. (2009) Refinement Types as Proof Irrelevance. In: Curien, P. (ed.) Typed Lambda Calculi and Applications. *Springer-Verlag Lecture Notes in Computer Science* **5608** 157–171.

Matuszewski, R. (1990) Formalized Mathematics. (Available at `http://mizar.uwb.edu.pl/fm/`.)

McLaughlin, S. (2006) An Interpretation of Isabelle/HOL in HOL Light. In: Shankar, N. and Furbach, U. (eds.) International Joint Conference on Automated Reasoning. *Springer-Verlag Lecture Notes in Computer Science* **4130** 192–204.

Mizar (2009) Grammar, version 7.11.02. (Available at `http://mizar.org/language/mizar-grammar.xml`.)

Naumov, P., Stehr, M. and Meseguer, J. (2001) The HOL/NuPRL proof translator – a practical approach to formal interoperability. In: Boulton, R. and Jackson, P. (eds.) 14th International Conference on Theorem Proving in Higher Order Logics. *Springer-Verlag Lecture Notes in Computer Science* **2152** 329–345.

Nipkow, T., Paulson, L. and Wenzel, M. (2002) Isabelle/HOL – A Proof Assistant for Higher-Order Logic. *Springer-Verlag Lecture Notes in Computer Science* **2283**.

Norell, U. (2005) The Agda WiKi. (Available at `\url{http://wiki.portal.chalmers.se/agda}`.)

Obua, S. and Skalberg, S. (2006) Importing HOL into Isabelle/HOL. In: Shankar, N. and Furbach, U. (eds.) International Joint Conference on Automated Reasoning. *Springer-Verlag Lecture Notes in Computer Science* **4130** 298–302.

Owre, S., Rushby, J. and Shankar, N. (1992) PVS: A Prototype Verification System. In: Kapur, D. (ed.) 11th International Conference on Automated Deduction (CADE). *Springer-Verlag Lecture Notes in Computer Science* **607** 748–752.

https://doi.org/10.1017/S0960129511000144 Published online by Cambridge University Press

Paulson, L. (1994) Isabelle: A Generic Theorem Prover. *Springer-Verlag Lecture Notes in Computer Science* **828**.

Paulson, L. and Coen, M. (1993) Zermelo-Fraenkel Set Theory. Isabelle distribution, ZF/ZF.thy.

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

Pfenning, F., Schürmann, C., Kohlhase, M., Shankar, N. and Owre, S. (2003) The Logosphere Project. (Available at `http://www.logosphere.org/`.)

Poswolsky, A. and Schürmann, C. (2008) System Description: Delphin – A Functional Programming Language for Deductive Systems. In: Abel, A. and Urban, C. (eds.) International Workshop on Logical Frameworks and Metalanguages: Theory and Practice. *Electronic Notes in Theoretical Computer Science* **228** 135–141.

Rabe, F. (2010) Representing Isabelle in LF. In: Crary, K. and Miculan, M. (eds.) Logical Frameworks and Meta-languages: Theory and Practice. *EPTCS* **34** 85–99.

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

Schürmann, C. and Stehr, M. (2004) An Executable Formalization of the HOL/Nuprl Connection in the Metalogical Framework Twelf. In: Hermann, M. and Voronkov, A. (eds.) 11th International Conference on Logic for Programming Artificial Intelligence and Reasoning. *Springer-Verlag Lecture Notes in Computer Science* **4246** 150–166.

Hales, T., Gonthier, G., Harrison, J. and Wiedijk, F. (2008) A Special Issue on Formal Proof. *Notices of the AMS* **55** (11).

Tarski, A. (1938) Über Unerreichbare Kardinalzahlen. *Fundamenta Mathematicae* **30** 176–183.

Trybulec, A. (1989) Tarski Grothendieck Set Theory. *Journal of Formalized Mathematics, Axiomatics* **1** (1) 9–11.

Trybulec, A. and Blair, H. (1985) Computer Assisted Reasoning with MIZAR. In: Joshi, A. (ed.) *Proceedings of the 9th International Joint Conference on Artificial Intelligence*, Morgan Kaufmann Publishers 26–28.

Urban, J. (2003) Translating Mizar for First Order Theorem Provers. In: Asperti, A., Buchberger, B. and Davenport, J. (eds.) Mathematical Knowledge Management: Second International Conference, MKM 2003. *Springer-Verlag Lecture Notes in Computer Science* **2594** 203–215.

van Benthem Jutting, L. (1977) *Checking Landau's 'Grundlagen' in the AUTOMATH system*, Ph.D. thesis, Eindhoven University of Technology.

Wenzel, M. (2009) The Isabelle/Isar Reference Manual. (Available at `http://isabelle.in.tum.de/documentation.html`.)

Whitehead, A. and Russell, B. (1913) *Principia Mathematica*, Cambridge University Press.

Wiedijk, F. (2006) Is ZF a hack? Comparing the complexity of some (formalist interpretations of) foundational systems for mathematics. *Journal of Applied Logic* **4** (4) 622–645.

Wiedijk, F. (2007) Mizar's Soft Type System. In: Schneider, K. and Brandt, J. (eds.) Theorem Proving in Higher Order Logics. *Springer-Verlag Lecture Notes in Computer Science* **4732** 383–399.

Wiener, N. (1967) A Simplification of the Logic of Relations. In: van Heijenoort, J. (ed.) *From Frege to Gödel*, Harvard University Press 224–227.

Zermelo, E. (1908) Untersuchungen über die Grundlagen der Mengenlehre I. *Mathematische Annalen* **65** 261–281. (English title: Investigations in the foundations of set theory I.)

https://doi.org/10.1017/S0960129511000144 Published online by Cambridge University Press