Noname manuscript No.
(will be inserted by the editor)

Semantics of Mizar as an Isabelle Object Logic

Cezary Kaliszyk · Karol Pąk

the date of receipt and acceptance should be inserted later

Abstract We formally define the foundations of the Mizar system as an object logic in the Isabelle logical framework. For this, we propose adequate mechanisms to represent the various components of Mizar. We express Mizar types in a uniform way, provide a common type intersection operation, allow reasoning about type inhabitation, and develop a type inference mechanism. We provide Mizar-like definition mechanisms which require the same proof obligations and provide same derived properties. Structures and set comprehension operators can be defined as definitional extensions. Re-formalized proofs from various parts of the Mizar Library show the practical usability of the specified foundations.

1 Introduction

Mizar [8] is one of the longest-lived proof assistants today. Its set theoretic foundations [27] together with an intuitive soft type system make the system attractive for mathematicians [68] and its formal library – the Mizar Mathematical Library (MML) [7] – became one of the largest libraries of formalized mathematics today. It includes many results absent from those derived in other systems, such as lattices [9], topological manifolds [63], and random-access Turing machines [42]. The unique features of the system include:

- A soft type system which allows for dependent types and intersection types,
- Declarative proofs in Jaśkowski-style natural deduction [33],
- Convenient structure types with (multiple) inheritance,

The paper has been financed by the resources of the Polish National Science Center granted by decision n°DEC-2015/19/D/ST6/01473 and the ERC starting grant no. 714034 *SMART*.

Cezary Kaliszyk
Department of Computer Science, University of Innsbruck, Innsbruck, Austria
cezary.kaliszyk@uibk.ac.at

Karol Pąk
Institute of Informatics, University of Białystok, Białystok, Poland
pakkarol@uwb.edu.pl

– A term language that allows for disambiguation based on types, in some cases allowing hundreds of meanings of a single symbol,
– Proof checking corresponding to the research on the notion of "obviousness" of a single proof step, without the need to explicitly reference proof procedures or tactics.

Nevertheless, the system has been developed since the seventies, and some of the decisions taken at that time still affect the current implementation. The following issues have served as a motivation for the work presented here.

Semantics. The complete semantics of Mizar has only been described in papers. The Mizar reference manual [27] introduces the underlying first-order logic variant by explaining the behavior of the CHECKER, discusses the soft type system, definitional mechanisms, and automation mechanisms. Further extensions of the system are described only in the articles introducing these extensions. Examples include the handling of ellipsis and flexary connectives [41] and the integration of SAT solvers [53].

Knowledge. As Mizar has a highly overloaded language, it is hard to access the contents of its library. Even with the various attempts at extracting the contents of the library to other formats (discussed in Section 12.3) some of the knowledge contained in the library has never been completely available outside of the system. In particular, the MML content has been hard to access for users of other proof assistants and authors of tools that manage mathematical knowledge.

Monolithic Kernel. Issues in the implementation of a proof assistant kernel can result in invalid proofs being accepted. With the source code of the large kernel of Mizar available only to the members of the Association of Mizar Users the situation may be more worrying for the community. Allowing such portability of proofs across systems provides a good way of proof certification [2].

User Interface. The main user interface for verifying Mizar proofs is a batch tool. The tool processes complete Mizar *articles* – that is collections of Mizar definitions, statements, and proofs gathered in a single file (this corresponds to the Isabelle notion of a theory). The Emacs mode provides some additional functionality, but it is still limited when compared to the modern interfaces of many other proof assistants [12, 80].

1.1 Contributions

In this paper we introduce the semantics of Mizar and discuss the possible ways to express this semantics in a logical framework. We further implement the Mizar foundations as an object logic in the Isabelle logical framework [61] and re-formalize a number of articles from the MML that include all the important techniques used in Mizar to practically demonstrate the adequacy of our encoding. In particular:

– We specify the semantics of the Mizar core in a concise way mostly abstracting from the Mizar syntax and from the naming scheme used in the system (Sect. 3).
– We discuss the possible approaches to encode the Mizar semantics in a logical framework. We propose our encoding as an Isabelle object logic that can inherit either from Isabelle/FOL or from Isabelle/HOL (Sect. 4). The former allows a closer correspondence with the semantics of Mizar, while the latter allows the use of the packages developed only for HOL, such as learning-reasoning [13] and counterexample-search [14].

- We specify the Mizar type system in the object logic covering all its peculiarities, including intersection types, inhabitation, and sethood. We make all the basic operations (such as quantifiers and choice) rely on the type system, which allows us to express the set theory axiomatization similarly to the Mizar one (Sect. 5).
- We propose mechanisms to introduce all the concepts that can be defined in Mizar. We develop derived mechanisms based on standard Isabelle definitions (Sect. 6).
- We discuss the background knowledge available in Mizar, in particular reserve that globally assigns given types to selected variable names, `ty` that collects types of term, and `cluster` that provides user-defined type inference rules. We develop an Isabelle/ML mechanism that imitates Mizar type inference. (Sect. 7).
- We introduce structures (Sect. 8) and set comprehension operators (Sect. 9).
- To practically show the usability of the defined Mizar foundations, we re-formalize a number of Mizar articles. The re-formalizations are performed manually, however are able to use minimal Mizar-like automation. The formalizations include set theoretic notions, algebraic structures, ordinals, and Mizar computers. Some of these rely on the Tarski axiom and its basic consequences (Sect. 11 and 10 respectively). A single example of a re-formalized Mizar proof will be presented in Fig. 3.
- When writing the paper we decided to mostly avoid introducing the vocabulary used in Mizar jargon and some Mizar literature. A small dictionary of such terms is given in Appendix A. We recommend readers who are either familiar with the Mizar terminology or those who like to further read the Mizar literature to start with this Appendix.

The current paper includes certain contents from the papers presented at CPP 2016 [37] (only an initial axiomatization of the Mizar foundations in Isabelle remains in the current version), CICM 2017 [35] (where conditional definitions were introduced), MACIS 2017 [34] (where we proposed the semantics of Mizar structures and set comprehensions) and FedCSIS 2017 [36] (where we proposed partial automation for structure proofs).

The new content in the current paper, which was not covered in any of the previous works includes automating type inference, the specification of the background information and the mechanisms for its gathering, and explicit inhabitation predicate. Furthermore, we simplified the axiomatization of the Mizar foundations to rely on fewer axioms, while at the same time imitating more closely the semantics of some of the definitions (such as `non`) and introduced Mizar-like mother-types and result types for all type definitions. We performed the corresponding changes to all aspects of the system, which in particular involved significant changes in the proofs (inhabitation proofs are now necessary for every quantifier introduction and elimination). The final new content are some of the case studies: ordinals, natural numbers, and the basic definitions of category theory.

2 Logical Frameworks and Isabelle

Most interactive proof assistants implement a fixed foundational logic. However, it is also interesting to model logical systems themselves. *Logical frameworks* have been proposed for this purpose. Today some of them serve not only as tools for modeling logics, but also allow users to work in such specified logics, referred to as *object logics*. The two major logical frameworks today are LF (with its most current implementations

being Twelf [66] and MMT [64]) and Isabelle [82]. Various other systems are also useful for modeling logics and working with them, for example λ-Prolog [25] used to implement proof systems for the calculus of inductive constructions and higher-order logic [23].

The foundations of Isabelle, called the meta-logic *Pure* are a variant of simple type theory with shallow polymorphism. Pure together with the Isabelle kernel are implemented in Standard ML. The framework provides functionality that makes it convenient to define components of object logics, their notations, and the necessary procedures. Isabelle/HOL is today the most developed Isabelle object logic. Further Isabelle object logics include untyped set theory [62], constructive type theory, and Lamport's temporal logic of actions [49]. Paulson [61] discuses the various object logics implemented in Isabelle. Isabelle has been the only logical framework so far to support a complete declarative proof language – Isar. As Isar is inspired and therefore quite similar to Mizar [83], it is particularly attractive for representing Mizar.

The three main syntactic categories of Isabelle/Pure are types, terms, and theorems. Encoding a logic in Isabelle therefore consists of axiomatically specifying the new constructs in each of these categories, namely introducing the new types, constants, and axioms respectively. We will illustrate this, by giving excerpts of the encoding of first-order logic in Isabelle/Pure. As for most object logics, this starts with the introduction of the type of propositions:

typedecl o

The propositions of the object logic are related with the propositions of the meta-logic by specifying a judgement operator, traditionally called `Trueprop`. In some object logics it is denoted by ⊢, however in FOL and HOL statements are usually distinguishable from terms by their contests, it will be mostly hidden.

judgment
  Trueprop :: o ⇒ prop ((_) 5)

Next, the basic logical constants can be defined. This is usually done together with their introduction and elimination rules. First-order conjunction and implication together with their corresponding introduction and elimination rules are as follows:

axiomatization
  conj :: o ⇒ o ⇒ o (**infixr** ∧ 35) **and**
  imp :: o ⇒ o ⇒ o (**infixr** ⟶ 25) **and**
  False :: o
where
  conjI: P ⟹ Q ⟹ P ∧ Q **and**
  conjunct1: P ∧ Q ⟹ P **and**
  conjunct2: P ∧ Q ⟹ Q **and**
  impI: (P ⟹ Q) ⟹ P ⟶ Q **and**
  mp: ⟦P ⟶ Q; P⟧ ⟹ Q

As the foundations of Isabelle are polymorphic, it is also possible to specify constants and axioms parametrized by types. This is for example done for equality:

axiomatization
  eq :: α ⇒ α ⇒ o (**infixl** = 50)
where
  refl: a = a **and**
  subst: a = b ⟹ P(a) ⟹ P(b)

In order to specify first-order quantifiers, it is possible to relate them to the meta universal quantifier:

axiomatization
  All :: (a ⇒ o) ⇒ o (**binder** ∀ 10) **and**
  Ex :: (a ⇒ o) ⇒ o (**binder** ∃ 10)
where
  allI: (⋀x. P(x)) ⟹ (∀x. P(x)) **and**
  spec: (∀x. P(x)) ⟹ P(x) **and**
  exI: P(x) ⟹ (∃x. P(x)) **and**
  exE: ∃x. P(x) ⟹ (⋀x. P(x) ⟹ R) ⟹ R

Not all constants need to be axiomatized: it is possible to define some of them. The definition mechanisms available in Isabelle have been recently shown consistent even with overloaded constants [44] and with the user-level type definition mechanisms available in HOL [45]. Therefore specifying parts of object logics as constants and deriving their properties reduces the size of the code one needs to trust. One example of this is truth, which we can define and prove its introduction rule:

definition True ≡ False ⟶ False
lemma TrueI: True
  unfolding True_def **by** (rule impI)

The encoding of higher-order logic in Isabelle/Pure follows the same steps. The major difference is a type constructor for function spaces declared early. The presence of functions naturally requires a somewhat different axiomatization of quantifiers etc. In the remainder of the paper we assume basic knowledge of Isabelle. For further information on the syntax of Isabelle and its features see [81].

3 Mizar Semantics

The core of Mizar is based on first-order predicate logic. Mizar proofs are proofs in natural deduction, where the primitive inference rules are those of the Jaśkowski system [33] (the rules proposed by Jaśkowski are only slightly different from the ones by Fitch [26] and the ones by Ono [60]). The formulations of individual proof steps correspond to valid first-order predicate calculus formulae. Any first-order predicate logic statement for which a proof in natural deduction exists, can be justified in Mizar and conversely all individual steps which Mizar accepts correspond to steps provable in natural deduction.

When it comes to top-level statements, Mizar also allows formulating and justifying selected statement of second-order predicate logic, namely these where all the second-order variables are universally quantified at the top-level. These correspond to axiom schemes and theorem schemes. It is not possible to formulate (or define) properties of higher orders, including properties whose arguments are not sets. Various properties that go beyond first-order logic are handled explicitly in specific ways: set comprehension (Sect. 9), properties assigned to types (sethood), and properties of meta-level functions and predicates (properties).

Mizar verifies the given proof script in multiple stages. The verification of each individual proof step first makes sure that the expression is well-formed. This is the case if the article environment and the needed references are imported successfully [27] and the following stages succeed:

- *Parsing.* The parsing stage verifies that the whole article is written according to the Mizar syntax. In this paper we will mostly abstract from the concrete syntax of Mizar, we will therefore generally ignore this aspect.
- *Disambiguation.* Because of the high level of overloading in the Mizar syntax it is necessary to disambiguate expressions. In particular, for every expression, Mizar checks if the properties given to its arguments uniquely identify it. By properties we mean typing information, as discussed in the next section 3.1. This allows overloading, hidden arguments, flexary functions, etc. There are almost 30 kinds of Mizar errors corresponding to expression disambiguation. As imitating the syntax of Mizar is not the focus of this paper, we will again mostly ignore this aspect and rely on the logical framework's syntax and disambiguation mechanisms leaving Mizar-like syntax to future work. In the Mizar implementation the types of expressions are inferred in this stage. We will postpone the discussion of type inference and our approximation thereof until Sect. 7.1.
- *Inhabitation.* The next verification step ensures that for every expression that requires an inhabited type, the inhabitation of that type has been proven directly by the user, or follows from the known type inhabitation relations. The inhabitation check is performed for the types used in quantifiers, for the introduction of locally fixed variables, for the use of the choice operator, and for explicit typing judgements with intersection types `term` is `type1|type2`. We discuss this in more detail in Section 3.2.
- *Sethood.* For each set comprehension expression, it is necessary to verify that the covering type has the sethood property. This is necessary to avoid Russell's paradox and its variants. More details on sethood are given in Section 9.

Given a well-formed expression, Mizar proceeds with the actual proof step verification. A classical disprover is used for this. The list of assumptions together with the conjecture are transformed to disjunctive normal form. Finally a form of resolution using background information together with selected information about equalities (in the form of congruence closure [54]) is used in the Mizar implementation. There are various limitations to the implemented resolution procedure, both for the written proofs to correspond to the human notion of obviousness [21,65] and to allow for faster verification. Since we are not interested in these limitations, we will assume that the last phase corresponds to first-order validity of formulas in disjunctive normal form [84].

3.1 Mizar Type System

Types are not foundational in the Mizar logic. Semantically, Mizar types correspond to first-order predicates. Note however, that the proof checking will handle types differently from predicates. This was already indicated in the previous subsection, here we will discuss the handling of types in more detail. The fact that types are predicates means that any object can be of multiple types. To avoid ambiguity, Mizar makes sure that each object is assigned a *basic inhabited type* (referred to as radix in Mizar) and optionally a number of additional types (referred to as adjectives). These can later be extended or changed using Mizar redefinitions: using the type-widening relation terms can be interpreted as being instances of a more general type.

The mechanism is similar to idea of templates in generic programming, as well as to the use of base classes in object-oriented programming. In fact, most definitions

in the MML are formulated with few argument requirements. The meaning of objects with more advanced types as arguments is formulated using `redefinition`, where the more involved arguments can be used in (additional) hidden argument positions. Such redefined objects can be used anywhere, where the non-redified ones were accepted.

Type inference [52] can automatically derive some of the (adjective) types assigned to an object. Additional type inference information that is proved by the user is given in the form of Horn clauses [72]. Type information present in terms combined with the one provided in the assumptions is propagated in a bottom-up way. Note, that Mizar types can directly or indirectly depend on other types as well as on term arguments and their types (e.g. the type `being_homeomorphism Function` of `S,T` is defined for `S, T` being `TopStruct`). This corresponds to dependent types and polymorphism in other systems. Therefore, by bottom-up we mean both in the subterm order and considering the terms present in the types.

As in other typed systems, Mizar also uses types for early type checking, i.e., prohibiting expressions where the arguments of a term do not fulfill the properties used in the definition of the concept. Mizar treats the basic inhabited types differently from other types. First, the inhabitation proof needs to be provided before such types can be used (syntactically it is already in their definitions). Second, for each basic inhabited type[1] there is precisely one inhabited type that the current one directly extends to (this restriction will be relaxed for structure types with multiple inheritance). We will call this the *parent type* (it is sometimes called *mother type* in Mizar literature). Inhabited types can also be introduced as a restriction of a given inhabited type by a list of (adjective) types. For example, the type `Function` in its definition is the inhabited type `Relation` restricted by the (adjective) type `Function-like`, where `Relation` is the inhabited type `set` restricted by the (adjective) type `Relation-like`.

Other (adjective) types do not have further constraints, but specify the inhabited type for which the given type can serve as a further restriction (e.g. the type Function-like in its definition specifies that it can refer to objects of the inhabited type set).

3.2 Inhabitation

Certain proof verification optimizations in Mizar assume types to be inhabited. This is for example the case for quantifiers. Quantifiers that bind variables not present in the subformula can be dropped by the CHECKER. To avoid inconsistencies every quantifier must therefore range over an inhabited type. The same mechanism is also used for justifying the semantics of defined objects, as well as the reservation of variable types (the reserved types will need to be explicitly added assumptions). This does not restrict Mizar formalization to types that must be non-empty. For example the Mizar formalization of category theory [17] discusses possibly empty types.

Consider the inhabited type Θ. If the user employs the intersection type $\tau|\Theta$, for example by writing the formula:

for `x` being τ | Θ holds `P[x]`

and Mizar is not able to derive the existence of an object of the type it reports an error ("136 Non registered cluster"). It is however always possible to formulate it as:

for `x` being Θ st `x` is τ holds `P[x]`

Concrete examples of such constructions will be presented in Section 6.

[1] Except the root of the type hierarchy, the basic inhabited type object, discussed in Sect. 5.

3.3 Choice

Since 2008 (Mizar version 7.10.01) an explicit choice operator the is available in Mizar. It is used for example to define the empty set. Similar to quantifier terms, the choice operator can be used for any inhabited type.

In fact a variant of the axiom of choice is already implied by the formulation of the Tarski axiom (Sect. 10). Furthermore, the strong axiom of choice is implied by the conditional definitions present in Mizar. Therefore the explicit choice operator is just a convenience. Indeed, it is possible to define a meta-level function taking a single set as an argument, which is defined under the assumption $\bot$, and returns an element of the set that satisfies $\top$. Then, the result is a value of that type, with no other information provable about it. This corresponds to the choice operator.

3.4 Definitions

There are five kinds of new concepts that can be added by a Mizar user without extending the foundational logic. Each may depend on a number of arguments. The user needs to explicitly give the arguments and their types in the definition. In this section we discuss definitions of predicates, two kinds of type definitions (basic and adjective types), and meta-level function definitions. The semantics of structure definitions will be postponed until Sect. 8. Depending on the kind of definition, Mizar introduces new information (derived definitional theorems) about the concept. All the derived information is added to the background knowledge. A number of examples in both Mizar and Isabelle syntax will be given in Fig. 1.

Predicates and Types. Defining predicates and (adjective) types is straightforward. Mizar adds to the background information the expansion given by the definition (excluding conditional definitions). Mizar is able to automatically expand the definition at any point in the declarative proof. The definition can be used in two ways. First, it can be unfolded to match the user-given proof outline with the proof obligation. Second, the properties for the predicates allow for their corresponding automation.

Basic Inhabited Types. Since every basic type needs to be inhabited, Mizar ensures this at the time of definition. Furthermore, Mizar distinguishes basic inhabited types that are type abbreviations (Mizar name: expandable mode) from atomic types. A type abbreviation must only mention the parent type and the inhabitation information needs to be available in the environment. The definition of an atomic type can use the special variable it, which in the context of the definition has the parent type. Then, the user must show that there exists an object of the parent type which has the defining property. Mizar adds to the background information two facts. First, that the new type is inhabited. Second, that any value of the new type is also of the parent type (in particular that the new type can be used in any context where the parent type could be used). Finally, an introduction rule for a proof of the object being of the new type becomes available.

Meta-level Functions. Mizar allows two kinds of meta-level function definitions: definitions by equals and definitions by means. The result type of each defined function needs to be specified and the role of the result type is similar to that of a parent type for basic inhabited type definitions.

Function definitions by equals play a role similar to abbreviations for expressions and only require specifying the denoted term. Such a definition also allows modifying

the default type of an expression, when this cannot be expressed with a user-specified typing rule, which serves as type casting operators in Mizar. For example, the Mizar type of n-element sequences of real numbers (`n-element FinSequence of REAL`) and the type of vectors in n-dimensional Euclidean space (`Vector of TOP-REAL n`) are isomorphic but not equal, and it is possible to define casting functions in both directions. For any type-valid occurrence of the newly defined function, Mizar adds the result type information and the equality to the definition body.

Function definitions by means need a predicate in the definition body, which can use the special variable it to refer to the defined object. This is very similar to definitions of functions by the choice operator in other systems. Note however, that contrary to a direct use of the choice operator, for such definitions by means it is possible to specify additional conditions and it is necessary to provide the result type. Two user proof obligations need to be fulfilled: a proof of existence and of the uniqueness of the result of the function. The definitional theorem is used by Mizar only when the user explicitly refers to the definition.

4 Our Model of Mizar Semantics

In this section we discuss our semantic model of Mizar alongside relevant parts of its formalization in Isabelle and discuss the adequacy of this model. Our semantic model of Mizar has evolved in various ways since the initial one [37]: we reduced the number of needed constants and axioms, consider only one Isabelle type for Mizar types, simplified the handling of choice and type membership. We added predicates about these concepts (inhabited, sethood, ...) which allow closer mimicking of the logic of Mizar. We made it more flexible with respect to the underlying Isabelle object logic and finally made definitions much closer to those of Mizar. In the remainder of this section, we will focus on the current, improved model and will not highlight differences to the previously considered models. We start with the basics of FOL or HOL as implemented by the respective Isabelle object logics (only providing a theory file for each, which makes the syntax the same). An overview of all the introduced axioms and constants in the object logic together with their roles is given in Table 1.

4.1 Types, Inhabited Types, and their Declarations

Semantically, (adjective) types and inhabited types are predicates over sets. However Mizar distinguishes adjectives from inhabited types already syntactically. Namely, types can only occur as arguments of selected constructions (is, be, being, the, as, qua), while Mizar predicates must be applied to terms.

We therefore distinguish Mizar types from predicates, by introducing a new Isabelle meta type of Mizar types `Ty`. This syntactically distinguishes them from predicates, which allows for earlier error reporting, but most importantly allows a different processing of the two. This will be essential for efficient type inference (Sect. 7). Mizar also distinguishes inhabited types from other (adjective) types syntactically. We have considered separating the two in the logical framework before [35]. Such separation would indeed allow further control over the order in which intersection types can be built, to closer imitate the syntax of Mizar, however does complicate the semantics. We decided to simplify the model, by adding a meta-level predicate that corresponds

Isabelle Constant	Kind	Role	Details
If	definition	if-then-else construction	
Set, Ty	types	Types of Mizar sets and types	p. 9
be, define_ty, def_ty_property	axiomatization	Type System Specification	p. 10
inhabited	definition	Type Inhabitation	p. 10
choice, choice_ax	axiomatization	Axiom of choice	p. 11
object, in, object_root, object_exists	axiomatization	Axiomatic article HIDDEN	p. 11
Ball, Bex	definitions	Bounded quantifiers	p. 11
ty_intersection, non	definitions	intersection and complement types	p. 11
Struct, TheSelectorOf, field	definitions	structures and their fields	p. 25
domain_of, strict, the_restriction_of	definitions	domains, strictness, restrictions	p. 26
well_defined	definition	structure declaration	p. 27
sethood	definition	type sethood	p. 29
Fraenkel1, Fraenkel2, the_set_of_all	definitions	set comprehension variants	p. 30

Table 1 Overview of the axioms and constants introduced to specify Isabelle/Mizar. All other definitions and the Tarski-Grothendieck axioms have been already specified in the MML and are only translated by us to Isabelle.

to type inhabitation. Combining the two allows us to introduce fewer axioms to specify the logic.

typedecl Set
typedecl Ty
inhabited(D) $\longleftrightarrow$ ($\exists_L$x. x be D)

The $\exists_L$ is the standard existential quantifier of the logic (either FOL or HOL), as opposed to the Mizar bounded quantifier, for which we will later use the plain $\exists$ notation. Note that both Isabelle/FOL and Isabelle/HOL are polymorphic logics, so we can use the quantifiers present there with the type of Mizar sets.

We next introduce the constants specifying the Mizar foundations in Isabelle. The first two constants, ty_membership and define_ty, specify the relation between types and predicates. The former, written in infix form `S be T`[2], given a set `S` and a type `T` returns a proposition, which is valid if the set belongs to the Mizar type. This check is equivalent to the application of the predicate represented by the Mizar type. The latter allows creating types based on one-argument predicates. Since Mizar types can be conditional and since basic inhabited types need to have a parent type, it needs three arguments: the parent type, the condition, and the predicate. Whenever the parent type is not necessary (e.g. for adjective types) the root of the type hierarchy will be used. Our previous experiments with define_ty without the parent type information, would make type inference much more involved and would not allow expressing the information that applying a type to a term may require some basic properties. Consider that a Mizar term can only be `one-to-one` if it is of the `Function` type. Now if a term is not a function, it is impossible to prove either that it is or that it is not one to one.

[2] Mizar syntax allows the keywords be, being, and is for type membership in various contexts, trying to imitate English. Semantically they are the same. We will use be in this paper wherever possible.

consts
ty_membership :: Set ⇒ Ty ⇒ o (**infix** be 90)
define_ty :: Ty ⇒ (Set ⇒ o) ⇒ (Set ⇒ o) ⇒Ty

These two constants will be specified by a single axiom which specifies all the consequences of a type definition. The below definition appears unnecessarily complicated, and it seems it could be replaced by a type equivalent to a predicate, as discussed earlier. However, the Mizar semantics is weaker, therefore we consider only selected properties of the formula it be parent ∧ (cond(it) ⟶ property(it)). First, from type membership, we can infer its parent type and that the property holds under the condition. Second, if the parent type, condition, and property hold, we get membership. The final conjunct corresponds to the semantics of Mizar, when the property does not hold. It is necessary to show inhabitation for function definitions and conditional definitions when the condition does not hold.

axiomatization where
def_ty_property: T ≡ define_ty(parent, cond, property) ⟹
(x be T ⟶ x be parent ∧ (cond(x) ⟶ property(x))) ∧
(x be parent ∧ cond(x) ∧ property(x) ⟶ x be T) ∧
(x be parent ∧ ¬cond(x) ⟶ inhabited(T))

The choice constant is the choice operator specified for inhabited types only with choice_ax specifying the axiom of choice.

consts
choice :: Ty ⇒ Set (the _)
axiomatization where
choice_ax: inhabited(M) ⟹ (the M) be M

The define_ty constant allows specifying the complement (negation) of a type only under the assumption that it is of the parent type. It becomes a meta-level function that given a type returns a type. Note that `object` will be the root of type hierarchy, and will be specified in the next section.

non A ≡ define_ty(object, λ_. True, λx. ¬ x be A)

From the definition we immediately derive the following theorem which will be used to interpret non types.

x be non A ⟷ ¬ x be A

Similarly the `ty_intersection` constant with the infix syntax $T_1 \mid T_2$ (with a derived definition of the same name) allows the construction of a type which is the intersection of T_1 and T_2. The intersection operation will not require inhabited types. We only show the derived definition:

x be t1 | t2 ⟷ x be t1 ∧ x be t2

4.2 Bounded Quantifiers

Universal and existential quantification in Mizar requires specifying the Mizar type of the introduced bound variable. We introduce two constants `Ball` and `Bex`. Their

definitions will be very similar to these of bounded quantifiers in Isabelle/HOL or Isabelle/ZF, however instead of predicates we use the given Mizar types.

An additional assumption that we make is that the quantified types must be inhabited. By specifying this as an assumption of the definition, we get the semantics of Mizar: If the type is empty, it is not possible to prove neither the quantified formula nor its negation. We additionally introduce standard automation for the bounded quantifiers (Isabelle attributes [intro!], [dest], etc).

$$\mathsf{inhabited(D)} \Longrightarrow \mathsf{Ball(D,\ P)} \longleftrightarrow (\forall_L \mathsf{x}.\ \mathsf{x\ be\ D} \longrightarrow \mathsf{P(x)})$$
$$\mathsf{inhabited(D)} \Longrightarrow \mathsf{Bex(D,\ P)} \longleftrightarrow (\exists_L \mathsf{x}.\ \mathsf{x\ be\ D} \wedge \mathsf{P(x)})$$

These two will use the standard notations ∀x:D. P(x) and ∃x:D. P(x) respectively. Furthermore, the notations ∀x. P(x) (same for ∃) will also refer to the bounded quantifier, and will only be usable when the implicit type of x will be specified using reserve (Sect. 7.1).

4.3 Mizar Schemes

Mizar schemes require special treatment in Mizar, as they go beyond first-order logic. Namely, the statement can range over second-order variables, predicates and functions. All of these must be universally bound on the top-level. In Isabelle, whether we use FOL or HOL, it is possible to (even implicitly) meta-universally quantify a theorem statement over free second order variables, which makes expressing schemes straightforward. Therefore specifying a Mizar scheme only requires explicitly specifying the types of the quantified variables. Consider our Isabelle re-formalization of the Mizar ordinal induction:

theorem ordinal2_sch_1:
  assumes A1: P({})
    and A2: ∀A:Ordinal. P(A) ⟶ P(succ A)
    and A3: ∀A:Ordinal. A ≠ {} ∧ A be limit_ordinal ∧
          (∀B:Ordinal. B in A ⟶ P(B)) ⟶ P(A)
  shows ∀A:Ordinal. P(A)

The corresponding statement in Mizar needs to explicitly specify the argument types of the predicate P (its argument must be an Ordinal), while in our approach it is necessary to specify such argument types at each use (if it is not always satisfied by the argument, as is the case here because of the quantifiers ranging over ordinals).

Mizar requires the user to provide the number of arguments of each quantified predicate and their types. These are used to ensure that the arguments used in each application of the predicate extend to the given ones. This is also used as part of the parsing mechanism, to identify the expressions in the predicate defining formula. In this paper we do not focus on imitating the syntax of Mizar, therefore we assume that the syntax is sufficiently disambiguated.

When it comes to quantification over second-order functions, Mizar requires not only the types of the arguments but also the return type. This type information is often used in the proofs of the schemes. The following scheme requires the assumption T0 in the Isabelle re-formalization:

theorem funct_2_sch_4:
assumes [ty]: C be non empty|set D be non empty|set
and T0: ∀x:Element-of C. F(x) be Element-of D
shows ∃f:Function-of C,D. ∀x:Element-of C. (f . x) = F(x)

where `F` is explicitly given the type `F(Element` of `C)` → `Element` of `D` in Mizar.

4.4 Adequacy of the Model

The proposed model allows formulating all Mizar statements, definitions, and proofs. Here we discuss the adequacy of the basic constructions, while the extension of the logic with new definitions will be discussed in Sect. 6, and more advanced Mizar constructions, namely set comprehension operators and structures, as well as the adequacy of their representations will be discussed in Sections 8 and 9. Finally, our re-formalization of parts of the MML (Sect. 11) will show the adequacy of the model in practice.

With the help of explicit typings, every basic Mizar construction, first-order statement or Mizar typing statement can be expressed in Isabelle/FOL (or HOL). We believe Isabelle/Mizar is consistent, and its consistency is equivalent to the other works that add the axioms of set theory on top of simple type theory [3, 15, 58]. This is also equivalent to the existence of a suitably large cardinal. The only additional component of Isabelle/Mizar is the addition of types that correspond to predicates, which is equivalent to adding sets in Isabelle/HOL.

Our semantics is however slightly more liberal than that of Mizar. We relax the Mizar typing discipline, both in the sense that inhabited types are not distinguished from other (adjective) types and therefore can be combined in arbitrary ways, but also, because we allow the user to prove statements that would not be Mizar theorems (but we believe to be meaningful). Consider a set that has the one-to-one property in the sense of functions. Using the fact that a one-to-one function exists, we can use the choice operator for this type to get an element. As it is a set, the following statement is provable, and because the type is inhabited it can be used with quantifiers allowing us to prove facts about the type:

theorem [ex]: inhabited(set|one-to-one)
theorem ∀X : set|one-to-one. X be one-to-one|set

We can also reason about an object not being of a certain type, which is not possible in Mizar. Since every function transforms its domain to its range, we can prove that if the domain of x is not mapped to its range, then x cannot be a function. Again, this is reasonable, and does not lead to inconsistency, but would not be provable in Mizar (could not even be stated):

theorem ∀x : object. ¬ x be Function-of dom x, rng x ⟶ ¬ x be Function
using funct_2_th_2 **by** mauto

5 Set Theory Axiomatization

Given the foundation of the logic underlying Mizar and its type system, we can proceed with the foundations of set theory. In Mizar these are defined in three axiomatic articles: `HIDDEN`, `TARSKI_0`, and `TARSKI_A`. In this section we discuss the first two of these

articles. The third axiomatic article, which corresponds to the set theoretic foundations that go beyond the Zermelo-Fraenkel set theory will be discussed in Sect. 10.

The Mizar article `HIDDEN` introduces the inhabited types `object` and `set`, equality, and set membership. `object` is the root of the type hierarchy. This means that everything is an object and that this type is inhabited. This corresponds to two Isabelle/Mizar axioms that state these two properties. The types object and set are equivalent in Mizar. Mizar distinguishes them to restrict some of the definitional expansions (an example of such a restriction is preserving equality between numbers rather than to expand it to two set inclusions). We can define (rather than axiomatize) `set` to be `object` and immediately get its inhabitation. Using this equivalence requires explicit references to the definition. The next component is equality. Since equality is already handled by the Mizar checker internally, its definition in the MML is only for automated testing and documentation purposes. We re-use the equality of the underlying Isabelle logic restricted to sets. Set membership becomes a new uninterpreted function.

The Mizar article `TARSKI_0` introduces the core axioms of set theory. In Mizar, the reserve command allows specifying the default types of variables. These are used both in explicit quantifiers and to implicitly quantify the free variables in statement. In the Isabelle representation we want to have the axioms stated as clearly as possible, we therefore use the axiomatization command which means that the types of quantifiers can be handled automatically but the types of free variables need to be provided explicitly. We will alleviate this limitation in Sect. 7.1 by providing a new theorem command that can use such implicitly associated types. The Tarski-Grothendieck axioms can be seen in our formalization, for space reasons we do not repeat them in the paper.

6 Definitions

Mizar definition are specified in blocks. However, since we want to abstract from the Mizar syntax, we will ignore the blocks and focus on imitating each kind of concept that can be defined individually.

6.1 Type Definitions

There are four major kinds of type definitions in Mizar (we ignore the definitions by cases, which with higher-order constructions can be directly translated to one of the four major kinds). Three of these define basic inhabited types.

Type Defining (adjective) types is straightforward. It consists of the application of the define_ty constant to a predicate and an original type. We introduce an abbreviation (an Isabelle term abbreviation is a notation that is unfolded in the input process after type checking, therefore it is never seen by the reasoning core [81]) that allows a more Mizar-like construction this kind of type introductions, and show as an example the Isabelle/Mizar definition of `empty` which is the type of `set`s that do not contain any object. The attr[3] command introduces a definition of a type and derives a few basic consequences:

[3] An adjective in the Mizar language is an attribute or its negation.

abbreviation (input) attr_prefix (attr _ for _ means _)
where attr df for ty means prop ≡
(df ≡ define_ty(object, λit .it be ty, prop))

attr xboole_0_def_1 (empty)
attr empty for set means (λit. ¬ (∃x : object. x in it))

The only theorems introduced to the Mizar environment given a type definition are the stated equivalence and the introduction and elimination rules for the type. We provide minimal automation that automatically derives and makes such theorems available to the user.

Type Abbreviations Type abbreviations need to be expanded before the proof checking phase. This is done by the Mizar ANALYZER. A type abbreviation has to be inhabited, however this information does not need to be given together with the definition, but can be stated as a user-defined typing rule. We can therefore immediately imitate Mizar type abbreviations by Isabelle abbreviations (of type Ty). The following abbreviation introduces the dependent type of functions from X to Y.

abbreviation funct_2_mode_1 (Function-of _, _)
where Function-of X,Y ≡ (X,Y: quasi-total) | (PartFunc-of X,Y)

Basic Inhabited Types We define basic inhabited types (Mizar name: non-expandable modes) using the define_ty constant. The Mizar-like definitions are as follows.

abbreviation (input) mode_prefix (mode _ → _ means _)
where mode df → ty means prop ≡ (df ≡ define_ty(ty, λ_.True, prop))

Many definitions in Mizar not only introduce a concept but also ensure that it fulfills various properties, and store only derived properties of the concept. For this, we introduce a definitional mechanism in Isabelle (mdefinition) which given a statement of a definition together with an optional environment specified by mlet and a definitional theorem after the definition statement, asks the user to fulfill the necessary proof obligations and stores only the derived definitions. A definitional theorem must be an implication. Its first assumption is discharged with the raw definition. The result is of the form: $C_1 \Longrightarrow C_2 \Longrightarrow \ldots \Longrightarrow C_n \Longrightarrow D_1 \wedge \ldots \wedge D_m$, where C_i are the conditions to prove and D_i are the derived definitions. The following is a definitional theorem for the introduction of basic inhabited types, which can be proved with the help of the set axiom.

lemma mode_property:
assumes df: mode df → ty means prop
and m: inhabited(ty)
and ex: ∃x:ty. prop(x)
shows (x be df ⟷ (x be ty ∧ prop(x))) ∧ inhabited(df)

With this, we can define a new basic inhabited type by providing a raw definition df, a proof of the inhabitation of the parent type m and the existence of an element that satisfies the predicate ex. The derived definition states the conditions required for an object to be of the newly defined type M and stores the information that M is inhabited. Note that the inhabitation of the parent type p and the existence condition ex are used only to show that there exists at least one element of the defined type. We can now give an example of a concrete basic type definition, namely the dependent type of elements of the given set X:

```
mdefinition subset_1_def_1                                  (Element-of _) where
  mlet X be set
  mode Element-of X → set means
   (λit. (it in X if X be non empty otherwise it be empty)) : mode_property
```

Note, that the information about the inhabitation of the type is indeed necessary for conditional definitions, as we will show in the next subsection (it was missing in our previously considered approaches [35]).

Conditional Types Definitions with explicitly stated assumptions are very frequent in Mizar [28]. Such definitions are referred to as permissive in Mizar. The assumptions are often used to express the relations between the arguments in cases where the types are not powerful enough to capture all the conditions.

We will refer to basic inhabited types with assumptions as conditional types. These would be called permissive modes in Mizar. Consider the type of morphisms between `a` and `b` in the category `C`. This is an (inhabited) type if `a` and `b` are objects of `C`, `C` is non empty, and `Hom(a,b)` is non-empty. The adjective non `empty`–struct refers to the carrier being non-empty. In our Isabelle/Mizar re-formalization, the definition can be as follows (we use an abbreviation with an assumption, analogous to the previous abbreviation. It also uses an analogous lemma assume_mode_property, presented below):

```
mdefinition cat_1_def_5                                     (Morphism-of) where
  mlet C be non empty-struct | non void-struct | CatStr,
        a be Object-of C, b be Object-of C
  assume Hom_C(a,b) ≠ {}
  mode Morphism-of(a,b,C) → Morphism-of C means
     (λit. it in Hom_C(a,b)): assume_mode_property
```

The conditional definition allows accessing the definition body only when the condition is fulfilled. This means that the assumptions need to be repeated at every use of the definition, for example in the below (we already use the `mtheorem` command, rather than the standard Isabelle theorem. This allows the automated processing of Mizar background knowledge and type inference, and will be explained in 7.3):

```
mtheorem cat_1_th_5:
  ∀f : Morphism-of (a,b,C). Hom_C(a,b) ≠ {} ⟶ dom_C f = a ∧ cod_C f = b
```

and in all the definitions of concepts that rely on a conditional type. We will present such example in Sect. 6.3 (cat-1-def-13). The lemma used to automatically obtain the derived properties of such definitions is as follows:

```
lemma assume_mode_property:
  assumes df: assume as mode df → ty means prop
    and m: inhabited(ty)
    and assume_ex: as ⟹ ex x being ty st prop(x)
  shows
    (x be df ⟶ (x be ty ∧ (as ⟶ prop(x))))
     ∧ (x be ty ∧ as ∧ prop(x) ⟶ x be df)
     ∧ inhabited(df)
```

6.2 Meta-level Functions

As mentioned in Sect. 3.4 meta-level functions are defined by equals or by means. A definition by equals can be considered as a special case of one by means where the property of the defined object has the form it = ... and it does not occur on the right-hand side. We will therefore focus on the definitions by means, later adding infrastructure for definitions by equals for convenience. A first component that simplifies defining meta-level functions is a choice operator working at the level of first-order predicates:

abbreviation (input) theProp
  where theProp(ty, prop) ≡ the define_ty(ty, λ_. True, prop)

With this operator we can define simple functions by means using a combination of the information of the result type with the user given predicate. We show an abbreviation introduced for such definitions together with an example definition by means in Isabelle/Mizar, namely the definition of the power set of the set `X`:

abbreviation (input) means_prefix (func _ → _ means _)
  where func df → ty means prop ≡ df = theProp(ty, prop)

func zfmisc_1_def_1 (bool _) **where**
  mlet X be set
  func (bool X) → set means
    λit. (∀Y. Y in it ⟷ Y ⊆ X)

Note the func keyword. As meta-level functions are the most common kind of introduced objects, we introduce this keyword instead of mdefinition. It additionally uses the appropriate definitional theorem and stores some of the derived information in various theorem lists (Sect. 7.1). The first definitional theorem for functions by means is proved using the correctness conditions `existence` and `uniqueness`, as well as the fact that the result type is inhabited. Only the result type is automatically known in Mizar, the other two derived theorems require explicit references to the definition.

lemma means_property:
  assumes df: func df → ty means prop
    and m: inhabited(ty)
    and ex: ∃x:ty. prop(x)
    and un: ⋀x y. x be ty ⟹ y be ty ⟹
        prop(x) ⟹ prop(y) ⟹ x = y
  shows df be ty ∧ prop(df) ∧ (x be ty ∧ prop(x) ⟶ x = df)

The main reason why uniqueness is important for meta-level functions defined in Mizar is efficiency. Uniqueness is used implicitly in the congruence closure algorithm at the base of Mizar's proof checker to make processing of terms more efficient. Equivalent meta-level functions (including these defined by means) are assumed to return same results on same inputs [55]. This is clearly true for mathematical functions, even these defined by choice, as it is in our model. Even so, an easy way to avoid uniqueness proof obligations in Mizar would be to explicitly use the axiom of choice on inhabited types. However, such definitions would prohibit subsequent redefinitions. Namely, without uniqueness conditions, it would not be possible to show that choice applied to the new condition is equal to choice applied to the previous one.

The definitions by equals are a special form of these by means:

abbreviation (input) equals_prefix (func _ → _ equals _)
where func df → ty equals term ≡
  func df → ty means λit. it = term

where additionally the `uniqueness` condition is simplified out and the `existence` condition is reduced to `coherence`, that is showing that the term is of the declared return type.

6.3 Conditional Functions

Mizar meta-level functions can be conditional. Conditions imply more complex definitional theorems: they need to specify the properties satisfied when the arguments satisfy the assumptions and when it is not the case. If the conditions are not satisfied, the only information the Mizar CHECKER knows about the term is its result type. We imitate this by the following notation and definitional theorem which corresponds exactly to the Mizar semantics:

abbreviation (input) as_means (assume _ func _ → _ means _)
where assume as func df → ty means prop ≡
  df = the define_ty(ty, λ_. as, prop)

lemma assume_means_property:
assumes df: assume as func df → ty means prop
  and assume_ex: as ⟹ ∃x:ty. prop(x)
  and assume_un: ⋀x y. as ⟹ x be ty ⟹ y be ty ⟹
    prop(x) ⟹ prop(y) ⟹ x = y
  and mode_ex: inhabited(ty)
shows
  df be ty ∧ (as ⟶ prop(df)) ∧ (as ∧ x be ty ∧ prop(x) ⟶ x = df)

The following definition of morphism composition requires a conditional definition.

func cat_1_def_13 (_ $\circ_{\_,\_,\_,\_}$ _) **where**
  mlet C be Category, a be Object-of C, b be Object-of C,
    c be Object-of C, f be Morphism-of (a,b,C), g be Morphism-of (b,c,C)
  assume Hom_C(a,b)≠{} ∧ Hom_C(b,c)≠{}
  func g $\circ_{C,a,b,c}$ f → Morphism-of(a,c,C) equals g $*_C$ f

Note however, that not all Mizar library uses of conditional definitions necessitate the assumption to define the function. Often the conditions are only there to ensure the non-emptiness of the argument types. Furthermore, it also enforces a particular way in which the Mizar checker verifies reasoning steps, making proofs more efficient.

The next part of the MML that we re-formalize are all the basic meta-level functions in set theory. We show the Mizar definitions and the corresponding Isabelle code in Figure 1. Note, that all these function definitions have proofs of existence, uniqueness, as well as a number of properties which are a part of our formal proofs.

6.4 Remaining Mizar Definitions

Mizar predicates directly correspond to predicate definitions in Isabelle. Mizar definitions by cases require an `if` construct. We needed and provided definitions with two

```
  let y be object;
  func { y } → set means
:: TARSKI:def 1
    for x holds x in it iff x = y;

  let y, z be object;
  func { y, z } → set means
:: TARSKI:def 2
    x in it iff x = y or x = z;

  let X be set;
  func union X → set means
:: TARSKI:def 4
    x in it iff ex Y st x in Y & Y in X;

  let x,y be object;
  func [ x , y ] → object equals
:: TARSKI:def 5
    {{ x,y } , { x }};

   func {} → set equals
:: XBOOLE_0:def 2
     the empty set;

  let X,Y be set;
  func X \/ Y → set means
:: XBOOLE_0:def 3
    for x holds x in it iff x in X or x in Y;
```

```
func tarski_def_1 ({_}) where
  mlet y be object
  func {y} → set means λit.
    ∀x. x in it ⟷ x = y

func tarski_def_2 ({_ , _}) where
  mlet y be object, z be object
  func {y, z} → set means λit.
    ∀x. x in it ⟷ (x = y ∨ x = z)

func tarski_def_4 (union _) where
  mlet X be set
  func union X → set means λit.
    ∀x. x in it ⟷ (∃Y. x in Y ∧ Y in X)

func tarski_def_5 ([_ , _]) where
  mlet x be object, y be object
  func [x,y] → object equals
    {{x, y} , {x}}

func xboole_0_def_2 ({}) where
  func {} → set equals
    the empty|set

func xboole_0_def_3 (infixl ∪ 65) where
  mlet X be set, Y be set
  func X ∪ Y → set means λit.
    ∀x. x in it ⟷ x in X ∨ x in Y
```

Fig. 1 Excerpts of the Mizar articles TARSKI and XBOOLE_0 and their corresponding Isabelle formalizations.

cases, however the Mizar library contains for example `sign` which is defined using three cases. All such non-standard definitions can be reformulated to the basic definitions we provide. Furthermore, extended Mizar language constructs (for example ellipsis [41]) could be expanded to the provided Isabelle basic definitions.

7 Type Inference

All the type knowledge (as well as other kinds of background knowledge available in Mizar) can be formulated as Isabelle theorems. Certain kinds of knowledge are automatically considered by Mizar reasoning steps [27]. Since some type reasoning is necessary in almost every Mizar proof step, in this section we discuss the kinds of background information available in Mizar (mostly type information) and propose a type inference mechanism. We also mention automation for other background knowledge, such as introduction rules for proofs of predicates or types.

7.1 Background Type Information

Type information can be introduced in Mizar in three ways. In this section we discuss these ways together with the procedures we use to gather this information.

Explicit Type Information. Whenever a variable is introduced in a line of reasoning using one of the basic constructs (let, consider, reconsider) its type is either given explicitly at the declaration or a type associated with that variable name (by the reserve command) is used. We introduce a reserve command in Isabelle, that lets a user associate default types with variable names. We also introduce an Isabelle attribute `[ty]` which stores a list of theorems of the shape `term` is Θ in the local proof context. The theorems from this list will be automatically used by specific tactics.

Definitional Theorems. There are various kinds of type information that originate from definitional theorems, as discussed in the previous Section 6:

1. Function result types. These are given in definitions and can be stored by a designated theorem list (ty_func_cluster).
2. Parent types. Both the extension of types to their parent (inhabited) types and to the ancestors in case of structures can be again stored in a theorem list (ty_parent).
3. Inhabitation information. A list of types that are inhabited is stored (ex).

The three lists store type inference rules (Horn clauses), however they are separate, as they will be used differently. For the first one, the conclusion can be a larger term than the terms in the premises (so they can only be safely used backwards). For the second one, this is not the case: the right-hand side terms are simpler than the ones in the premises. For the last one, these are inhabitation information. Re-definitions which modify functor result types can be directly stored in the first list. As re-definitions are mostly syntactic, we will not discuss these in this paper.

User-given Type Inference Rules. There are three kinds of such rules, referred to as cluster registrations in Mizar.

- *existential* retain type non-emptiness information for types. These can again be stored in the ex list. For example the expansion of `Function of X,Y`:
  theorem [ex]:
  X be set ∧ Y be set ⟹
  inhabited(X,Y: quasi-total | PartFunc-of X,Y)
- *conditional* extend the list of types assigned to a term, given that particular types have already been established. These are again similar to ty_parent. Examples include:
  theorem [ty_cond_cluster]:
  Y be set ⟹ F be Y-valued | Function ⟹
  F be Y -bijective ⟹ F be Y -onto | one-to-one
  theorem [ty_cond_cluster]:
  Y be set ⟹ F be Y-valued | Function ⟹
  F be Y -onto | one-to-one ⟹ F be Y -bijective
- *functorial* automatically provide type information for compound terms expanding the information given in the definitions. For example the composition operation is defined for relations, however when applied to two functions it is also a function:
  theorem [ty_func_cluster]:
  f be Function ∧ g be Function ⟹ (g ∘ f) be Function_like

Mizar stores two lists of types for each variable (upper and lower cluster). The explicit information about all variables is expanded (rounded-up) by all the user-given inference rules to obtain the second list.

In our type inference mechanism, we will imitate the information passed between the Mizar ANALYZER and the CHECKER by the theorem list ty. That list will include all the rounded up information about all the terms in the current goal-state [52]. Furthermore, to imitate the Mizar super-cluster automation [52], we add the type information selected from the premises and the antecedents of the implications (since the facts are in the implication form) rounding-up the ty list repeatedly. Mizar uses the type information in the main verification process as part of unification, to add new type information to terms [84]. Details of our procedure follow.

7.2 Automating Type Inference

To support the user of the Isabelle/Mizar object logic, we develop a number of proof methods and other automation mechanisms that try to imitate the Mizar type inference using the logical framework infrastructure.

As information about any variable is added to the background knowledge using the ty attribute, we try to derive the consequences of this information and (recursively) add it to the same knowledge base. For example, given the information that `x be empty|set` and the user proved typing rule that any `empty|set` is also `finite`, we want to derive the information that `x is finite` and possibly any other consequences of this information. In particular, such user type information can include dependent types. Note that using the user-given typing rules in an uncontrolled way would often loop, deriving typing information about larger and larger terms.

Our type inference procedure consists of two functions. The function $\mathcal{J}$ attempts to derive consequences of a typing judgement without generating new terms and the function $\mathcal{T}$ tries to derive all types of a given term. A simplified overview of the procedure in pseudo-code is given in Fig. 2. The functions `infer_judgement` and `infer_term` correspond to $\mathcal{J}$ and $\mathcal{T}$ respectively.

The function $\mathcal{J}$ first separates the intersection types to individual typing judgements. All are added to the set `ty` and processed one by one. For each judgement, we first process its subterms. This includes both the subterms of the left-hand side and of the right-hand side. For example `x be Subset of ⟨A,A⟩` would also have `A` and the pair `⟨A,A⟩` as subterms. For each such subterm t we call $\mathcal{T}(t)$. Next, we look at the user-given forward typing rules. For each typing rule we attempt to discharge the last assumption with the added individual typing judgement. If this succeeded, all previous assumptions are discharged with theorems from the set `ty` from the right (the order is important to resolve type dependencies correctly). If this returned a new typing judgement j, we call $\mathcal{J}(j)$ recursively.

The function $\mathcal{T}(t)$ first calls itself recursively on all the subterms of t. Then it looks at all the user-given backward typing rules. We try to unify the term t with the left-hand side of the judgement in the conclusion of the typing rule. If this succeeds, we discharge all the remaining assumptions from the right using the theorems in the set `ty`. If this succeeds, we get a single new typing judgement; we apply $\mathcal{J}$ to that new judgement.

The two defined functions are used as follows in Isabelle/Mizar. First, when a user declares a typing judgement with the attribute `ty` the function $\mathcal{J}$ is called on the judgement. Second, we provide a proof method `mty`, which looks at all the propositions involved in the current proof goal and calls $\mathcal{T}$ on these propositions. To get even closer to the Mizar by, we implement a method that attempts to instantiate all the user

```
fun store_judgement(j) =
  for j' <- normalized conjuncts of j do
    add j' to [ty];
    infer_term (proposition_of (j'));
    for r <- user_forward_typing_rule
      if can biresolve last assumption of r by j'
      and can biresolve other assumptions by [ty] as j'' then
        store_judgement(j'')
    done
  done

fun infer_judgement(tm) =
  if can biresolve all assumptions of tm with [ty] as j' then
    store_judgement (j')
  else if can resolve some of the assumptions of tm as j' then
    add j' to user_forward_typing_rules

fun infer_subterm(tm) =
  for r <- all user typing rules do
    if can unify conclusion(r) with tm as r' then
      infer_judgement(r')
  done

fun infer_term(tm) =
  if not (tm considered already) then
    iterate_subterms(infer_subterm, tm)
```

Fig. 2 A pseudo-code overview of the type inference automation in Isabelle/Mizar.

given helper theorems and instantiates them exhaustively with the typing information derived from the goal. Finally the proof method `mauto` does call `mty` to derive all type consequences, `mby` to instantiate the Mizar typing information, and calls `auto` [81] on the goal. We will use it as a replacement of the Mizar by for most of the goals in the re-formalizations.

The behavior of our typing procedure is very similar to the typing procedure implemented in Mizar, however we do not impose any limits on the typing.

7.3 Type Inference Example

To illustrate our type inference, consider the proof of the group inverse property presented in Figure 3. The variable `G` is reserved to be a group, while `g` and `h` are its elements. For reserved variables, their uses are automatically bound with an outermost universal quantifier, and the variable is introduced into the line of reasoning.

The `mtheorem` command realizes this on the top-level in our representation. However, eliminating the quantifier requires the type to be inhabited – we ensure this using ex. The information about `G`, `g`, `h` are added to the list ty. The procedure discussed in the previous subsection computes the consequences of this information for all the subterms of the goal. This generates 23 theorems on the ty list. For each inference step, the list is locally extended, up to 35 typing judgements in this proof. Examples of derived information include G is unital, $(\mathsf{h} \otimes_G \mathsf{g})^{-1}{}_G$ is Element-of-struct G, and the carrier of G is non empty.

As illustrated in Fig. 3 the similarity between Mizar and Isar allows us to reformulate the proof outlines. Such a similarity is however not always possible. Minor issues include the lack of take in Isar (which requires creating a subproof after each take),

```
reserve G for Group;
reserve h,g for Element of G;

theorem Th16:
  (h * g)" = g" * h"
proof
  (g" * h") * (h * g)
   = g" * h" * h * g
     by Def3
  .= g" * (h" * h) * g
     by Def3
  .= g" * 1_G * g
     by Def5
  .= g" * g
     by Def4
  .= 1_G
     by Def5;
 hence thesis
     by Th11;

end;
```

```
reserve G for Group
reserve h,g for Element-of-struct G

mtheorem group_1_th_16:
  (h ⊗_G g)^-1_G = g^-1_G ⊗_G h^-1_G
proof-
  have (g^-1_G ⊗_G h ^-1_G) ⊗_G (h ⊗_G g)
              = (g^-1_G ⊗_G h^-1_G) ⊗_G h ⊗_G g
    using group_1_def_3E[of _ _ h] by mauto
  also have ... = g^-1_G ⊗_G (h^-1_G ⊗_G h) ⊗_G g
    using group_1_def_3E by mty auto
  also have ... = g ^-1_G ⊗_G 1._G ⊗_G g
    using group_1_def_5 by mauto
  also have ... = (g ^-1_G)  ⊗_G g
    using group_1_def_4 by mauto
  also have ... = 1._G
    using group_1_def_5 by mauto
  finally show ?thesis
    using group_1_th_11[of _ h ⊗_G g,
    THEN conjunct1] by mauto
qed
```

Fig. 3 A proof of a property of group inverses in Mizar and corresponding Isabelle/Mizar.

or the fact that Mizar can change the thesis within one proof block, which in Isabelle corresponds to the use of a very involved initial proof method. Despite the automation mechanisms we provide, there are many cases where the Mizar proofs can be shorter than these done in our re-formalization. The final difference is the explicit need to state the proof methods.

8 Structures

In this section we introduce the semantics of Mizar structure and present our Isabelle model thereof.

8.1 Mizar Structure Semantics

Mizar structures represent mathematical tuples, where the fields of the tuple are additionally equipped with labels. Such labels allow referring to particular fields of a tuple even as tuples are extended or restricted. We will refer to the association between the labels and the types of the corresponding fields as the *structure type*. This is equivalent to a class signature in object oriented languages or in some proof assistants. An actual object of a particular structure type will be referred to as a *structure instance*. The Mizar approach does not remember the positions of the elements in the tuples, but ensures that the labels of the elements are unique (the labels are called `selectors` in Mizar). Mizar structures are quite similar to the work of Naraschewski and Wenzel [51] which uses parametric polymorphism to define records with labels, however the Mizar type system allows multiple inheritance and requires no coercions between structure types and their subtypes.

Structure instances can be modeled by partial functions from the labels to the values on that labels. Structure types can be modeled by functions from the labels to the allowed types of values on those labels. In order to ensure structure inheritance (including multiple inheritance), structure types only specify the minimal set of labels that must be contained in the function definition and not the whole domain. Every assignment must be of the form *sel* $\rightarrow$ *spec*, where *sel* is a selector and *spec* is the specification of the type of values that of the functions applied to that selector. Note, that the type of a particular field can depend of the values on the other selectors.

Consider the tuple $\langle R, +, \mathbf{0}, \cdot, \mathbf{1}\rangle$. It can be interpreted as a ring, where the structure type should specify that it must contain a set R, two binary operations $+$, $\cdot$ and two selected elements $\mathbf{0}$, $\mathbf{1}$ of the set. The following Mizar structure `doubleLoopStr` is used in the MML as the structure type for a ring:

```
struct (multLoopStr_0, addLoopStr) doubleLoopStr (#
  carrier → set, addF → BinOp of the carrier,
  ZeroF → Element of the carrier, multF → BinOp of the carrier,
  OneF → Element of the carrier #)
```

where the list `(multLoopStr_0, addLoopStr)` specifies the structures the current one inherits from. `multLoopStr_0` corresponds to a multiplicative group with a selected zero $\langle R, \cdot, \mathbf{0}, \mathbf{1}\rangle$ and `addLoopStr` corresponds to an additive group without a selected element $\langle R, +\rangle$.

The structure type only corresponds to the signature of the structure. Further properties that make the above type correspond to rings are added by further intersecting this type with (adjective) types that correspond to those properties. The structure type definition in Mizar ensures the inhabitation of the newly defined type. For this, the checker verifies that all the defined structure specifications are non-empty. The selectors occurring in one specification cannot repeat. Furthermore, the order of the selectors in Mizar is important and the use of a selector in a specification is only allowed after providing its specification. Finally, Mizar verifies that the new type inherits from its ancestors, i.e. the assignments of all ancestors appear in newly defined structure type.

Mizar provides the following definitional theorems for any user-given structure type definition `str`. First, the structure type is inhabited. Second, a `strict str` version of the type is created. An object is of the strict structure type, if it contains only the selectors specified in the definition of `str`. Third, for each assignment from selector `sel` to its specification `spec` and for each `X : str` the term the `sel` of `X` becomes available. It corresponds to an application of `sel` to `X` and its type is `spec(X)`. Fourth, a constructor of the structure type is introduced (Mizar name: *aggregate*). An example constructor in Mizar is:

```
definition
  func F_Real → strict doubleLoopStr equals
    doubleLoopStr
      (# REAL, addreal, multreal, In(1, REAL), In(0, REAL) #);
end;
```

This constructor is of the type `strict str` and all its fields are as given, for example the `addF` of `F_Real` = `addreal`. Fifth, if two objects of the type `str` are both `strict str` and all their fields are equal, than the objects are equal. Finally a restriction of an object to an ancestor structure is possible. For example:

```
the addLoopStr of F_Real = addLoopStr(# REAL, addreal #)
```

8.2 Formal Structure Preliminaries

We will represent structure types by schemes of functions. Like in Mizar, these structure types can be further restricted by the user (using the intersection type system) with further given properties. The structure type will specify the domain of every structure instance of that type. Additionally, it will specify the types of the elements in the co-domain. To represent such (partial) functions, we first re-formalize the first few articles of the MML that formally define Mizar functions. As these do not use Mizar structures, they can be directly used to represent both structure types and structure instances without adding any new axioms to the foundations.

Our structure definition starts by defining the `selector` of. This corresponds to a partial function application, but does not return the default value when the argument is outside of the domain of the function (which is the behavior of function application in the MML). The meta-level function definition infrastructure presented in Sect. 6 can be used directly. To fulfill the correctness conditions `uniqueness` and `existence` we need the condition that `selector` in `dom Str`.

definition TheSelectorOf (the _ of _) **where**
  func the selector of Str → object means λit.
    ∀ T : object. [selector,T] in Str ⟶ it = T

To specify the assignments, it is necessary to have an infinite set of distinct labels. Strings could be used for this purpose, but to further reduce the part of the MML formalization required to specify structures, we chose to use the set theoretic natural numbers `0 = {}`, `succ(X) = X ∪ {X}`. The only property that we need to derive about these labels is distinctness. We define the label "carrier" to 0 and all subsequent labels as distrinct numbers. All labels are currently manually defined in the `mizar_string` theory.

As shown in the examples in Sect. 8.1, parts of the specification may refer to other fields of the structure. For this reason, we make each selector a meta-level function, which given the current structure instance returns the type of that assignment. Now we can define a single assignment from a selector to its specification as a meta-level function again:

definition field (**infix** → 9) **where**
  selector → spec ≡ define_ty(object, λ_. True,λit.
    the selector of it be spec(it) ∧ selector in dom it)

To illustrate this in practice, consider the double loop structure which serves as the signature of various algebraic structures including rings and fields. The `carrier` is a `set` (it can ignore the argument `S`), while for example the `zero` uses the argument `S` to specify its type as an `Element` of the `carrier` of `S`. The definition finally requires the derived definition lemma given the explicitly given domain (last line of the below `mdefinition`) discussed further in Sect. 8.3.

mdefinition doubleLoopStr_d(doubleLoopStr) **where**
struct doubleLoopStr (#
  carrier → (λS. set);
  addF → (λS. BinOp-of the carrier of S);
  ZeroF → (λS. Element-of the carrier of S);
  multF → (λS. BinOp-of the carrier of S);

```
  OneF → (λS. Element-of the carrier of S)
#) : well_defined_property[of _ _ {carrier}∪{addF}∪{ZeroF}∪{multF}∪{OneF}]
```

To correctly imitate the semantics of each assignment, is it sufficient to ensure the conditions the `selector` of it be `spec(it)` and `selector` in `dom` it, where the latter allows us to globally specify the minimal set contained in the domain of any instance of a structure type. Now the domain of the structure can also be defined globally. Note however, that the result can only be evaluated for a particular defined structure type. We give the definition of `domain_of` together with an example:

```
definition domain_of                                   (domain_of _) where
  func domain_of M → set means (λit.
     (∃X:M. it = dom X) ∧ (∀X : M.  it ⊆ dom X))

lemma domain_of doubleLoopStr =
    {carrier} ∪ {addF} ∪ {ZeroF} ∪ {multF} ∪ {OneF}
```

In a similar way we can globally define `strict` as a meta-level construction that given a structure type returns its strict version.

```
definition strict where
  strict(M) ≡ define_ty(object,λ_.True, λX. X be M ∧ dom X = domain_of M)
```

We finally define the restriction of an instance to a structure type. A restriction must be `strict` by construction. The definition directly uses the restriction of a function to a smaller domain, which we denote by $f\restriction_A$:

```
definition the_restriction_of                    (the_restriction_of _ to _) where
  func the_restriction_of X to Str → strict(Str)
    equals X↾domain_of Str
```

8.3 Partial Automation for Structures

For every user-defined structure we would like to automatically derive the same properties which Mizar assumes. In this subsection we will present a general lemma which will make individual structure definitions simpler. This lemma, when instantiated with the raw structure definition, shall return all the derived facts. The first approach to stating such a lemma is as follows:

```
lemma struct_scheme:
  assumes df: struct S Fields
    and exist: ∃_L X . X be Fields|Struct ∧ dom X = D
    and monotone: ∀_L X1. X1 be Fields|Struct ⟶ D ⊆ dom X1
    and restriction: ∀_L X1. X1 be Fields|Struct ⟶ X1↾_D be Fields
  shows (x be S ⟷ x be Fields|Struct) ∧
      inhabited(S) ∧ inhabited(strict(S)) ∧
      domain_of S = D ∧
      (E be S ⟶ the_restriction_of E to S be strict(S))
```

where `Struct` is equivalent to `Funct`, however we will not want to unfold this definition outside of structure definitions to avoid structures being fully expanded to

`Function-like Relation-like set`. Now, to define a concrete structure type, one needs to specify the domain D of a structure. We give an outline of the proof, for more details see the formalization. The existence and monotonicity assumptions (ex and `monotone`) in the lemma are necessary to show the correctness of the definition of `domain_of` (its existence and uniqueness) to obtain the equality `domain_of S = D`. Next, the-restriction-of `X to S` is defined by equals, therefore the correctness of this definition requires a proof that the right hand side has the correct return type, i.e. $\texttt{X}\restriction_{\texttt{domain\_of S}}$ is of the type `strict(S)`. To show that it is of the type `S` we use the equation `domain_of S = D`. To show that it is also `strict` we need to show `dom` $(\texttt{X}\restriction_{\texttt{domain\_of S}})$ `= domain_of S`. This follows by definition of function restriction and the `monotone` assumption. This completes the proof.

To further simplify the definitions of structures, we note that the choice operator can be used to fulfill the existence condition. An example use of the choice operator the to show that the double loop structure is inhabited is:

term {[carrier, the set]} ∪
{[addF, the BinOp-of (the set)]} ∪
{[ZeroF, the Element-of (the set)]} ∪
{[multF, the BinOp-of (the set)]} ∪
{[OneF, the Element-of (the set)]}

however using it directly would be quite tedious. For `doubleLoopStr` the manual proof in Isabelle requires about 100 lines. We will instead create structures recursively. This is similar to what is implemented in Mizar.

A naive approach to constructing structures recursively which would directly use the `struct_scheme` lemma, fails for the following reasons. The assumptions of the `struct_scheme` lemma can be used to show the non-emptiness of a defined structure type `S`. However, the assumptions about the ancestors of `A` are insufficient to prove the other properties of `S`. In particular, there is no condition that would correspond to the `restriction` condition, which would in turn give the necessary information as to which extensions of `A` satisfy all the assignments of `A`.

For this reason, we extend the `struct_scheme` lemma assumptions by an additional condition, which will allow using the properties of the ancestor `A` in the proofs of the currently defined structure. This condition, `well_defined`, is as follows:

definition well_defined_prefix (**infix** well defined on 50)
where
Fields well defined on D ≡
$(\exists_L$ X . X be Fields|Struct ∧ dom X=D) ∧
$(\forall_L$ X1. X1 be Fields|Struct ⟶ D ⊆ dom X1 ∧ $\text{X1}\restriction_D$ be Fields) ∧
$(\forall_L$ X1 X2. X1 be Fields|Struct ∧
X2 be Struct ∧ D ⊆ dom X1 ∧ X1 ⊆ X2 ⟶ X2 be Fields)

We can now derive a defining lemma with stronger assumptions (they are stronger, as `Fields well defined on D` implies `exists`, `monotone`, and `restriction`):

lemma well_defined_property:
assumes df: struct S Fields
and well: Fields well defined on D
shows (x be S ⟷ x be Fields|Struct) ∧
inhabited(S) ∧ inhabited(strict(S)) ∧

```
domain_of S = D ∧
(E be S ⟶ the_restriction_of E to S be strict(S)) ∧
(Fields well defined on D)
```

The list of existing assignments specified for the domain `D` can be augmented with a new assignment `selector` → `specification` given that the `selector` is not present in `D`, and that any selectors the `specification` uses are present in `D`.

As the maximum number of extended selectors used in a new specification in the MML is three, we formulate the following lemma that can be used to create new specifications. Note, that it can directly be used for the cases of one and two new selectors and a version without selectors is even simpler. We skip these in the paper.

```
theorem Fields_add_3_arg_Mode:
  assumes Fields well defined on D
      sel_1 in D    sel_2 in D    sel_3 in D    ¬ sel in D
    and ⋀X1. X1 be Fields|Struct ⟹
      inhabited (M1(the sel_1 of X1, the sel_2 of X1, the sel_3 of X1))
  shows Fields | (sel → (λS. M1 (the sel_1 of S, the sel_2 of S, the sel_3 of S)))
      well defined on D ∪ {sel}
```

Using this lemma we can provide a partial automation for defining structures: a single Isabelle tactic (the use of the lemma followed by auto with a number added introduction and elimination rules) can now automate the obligations necessary for a structure definition.

Mizar structures also allow inheritance. In our approach the inheritance information always follows in a straightforward way from the first conjunct of `struct_scheme`. We can show such a proved inheritance lemma:

```
theorem doubleLoopStr_inheritance[ty_parent]:
  assumes X be doubleLoopStr
  shows X be multLoopStr_0 ∧ X be addLoopStr
using assms doubleLoopStrA addLoopStrA multLoopStr_0A by auto
```

The final property needed to adequately imitate the Mizar semantics of structures, is the fact that structure constructors are of structure types. Such typing rules are again proved for every structure. For example:

```
lemma doubleLoopStr_AG[ty_func]:
  assumes X be set   A be BinOp-of X   Z be Element-of X
   M be BinOp-of X   E be Element-of X
  shows
   [#carrier↦X; addF↦A; ZeroF↦Z; multF↦M; OneF↦E#] be doubleLoopStr
```

where the constructor `[# ... #]` is a set-theoretic function, that is a union of pairs.

8.4 Structure Adequacy

The proposed model of structures is very similar to that of Lee-Rudnicki [47] imitating the Mizar model closely. When it comes to the actual interaction with the structures our implementation is slightly more liberal: it allows providing inheritance information

after the point of the definition, as well as allows to prove the inhabitation of a structure type directly.

The automation we provide is similar to that given by Mizar, but it is also possible to directly show structure extensions that would require an indirect definition with further type restrictions in Mizar. Consider a structure similar to `1-sorted`, but where the `carrier` must be additionally non-empty. We cannot indicate `1-sorted` as an ancestor of the structure, because Mizar reports the error `92: Type of the field must be equal to the type in prefix`.

theorem
X be Inhabited_1-sorted $\Longrightarrow$ X be 1-sorted
X be non empty-struct|1-sorted $\Longrightarrow$ X be Inhabited_1-sorted

Furthermore, the correctness of definitions in Mizar relies on the order of the fields, whereas in our model it is even possible to define a structure whose selector specifications refer to each other mutually:

definition
struct Test (# testA $\rightarrow$ (λT. Element-of the testB of T);
testB $\rightarrow$ (λT. Element-of the testA of T) #)

It is possible to prove the correctness of this definition, since `{}` is `Element` of `{}`, see `subset_1_def_1` on page 16. For this simple example we can even show properties, for example that this structure uniquely determines its fields:

theorem $\forall$ T:Test. the testA of T = {} $\wedge$ the testB of T = {}

9 Set Comprehension

Mizar set comprehensions allow defining sets which satisfy the given predicate (see [27, Fraenkel]). The Mizar syntax for set comprehension is:

$\{\mathtt{t}(\mathtt{v}_1,\mathtt{v}_2,\ldots,\mathtt{v}_n)$ where $\mathtt{v}_1$ is Θ_1, $\mathtt{v}_2$ is $\Theta_2,\ldots,\mathtt{v}_n$ is Θ_n:$\mathtt{P}[\mathtt{v}_1,\mathtt{v}_2,\ldots,\mathtt{v}_n]\}$

Semantically, such a set comprehension has the type set in Mizar. The system does not allow a further specification of the type (as opposed to, e.g., function definitions, where the result type can be explicitly given). To avoid Russell's paradox, set comprehension expressions are well-formed only when all the types $\Theta_1, \Theta_2, \ldots, \Theta_n$ have the sethood property. If not, Mizar reports the *"It is only meaningful for sethood property"* error [28].

Definition 1 A Mizar-type Θ has the sethood property if all objects of the type Θ are elements of some set.

Set membership applied to well-formed set comprehension terms is automatically expanded as follows:

$\mathtt{x}$ in $\{\mathtt{t}(\mathtt{v}_1,\mathtt{v}_2,\ldots,\mathtt{v}_n)$ where $\mathtt{v}_1$ is Θ_1, $\mathtt{v}_2$ is $\Theta_2,\ldots,\mathtt{v}_n$ is $\Theta_n : \mathtt{P}(\mathtt{v}_1,\mathtt{v}_2,\ldots,\mathtt{v}_n)\}$
$\longleftrightarrow$
$\exists \mathtt{v}_1 : \Theta_1,\ \mathtt{v}_2 : \Theta_2,\ \ldots,\mathtt{v}_n : \Theta_n.\ \mathtt{x} = \mathtt{t}(\mathtt{v}_1,\mathtt{v}_2,\ldots,\mathtt{v}_n) \wedge \mathtt{P}[\mathtt{v}_1,\mathtt{v}_2,\ldots,\mathtt{v}_n]$

To faithfully represent Mizar set comprehensions in Isabelle, we first define the sethood property, and derive its basic consequence. Like in the MML, we show the sethood property for various major types. We provide a tactic to automate sethood inheritance proofs.

definition sethood(M) ≡ ∃X:set. ∀x:object. x be M ⟶ x in X
theorem sethood:
sethood(M) ⟷ (∃X:set. ∀x:object. x be M ⟷ x in X)

We can now show the existence of set comprehensions globally. For this we introduce a Frankel definition:

definition Fraenkel1 **where**
func Fraenkel1 (F, D, P) → set means λit.
∀x : object. x in it ⟷ (∃y : D. x = F(y) ∧ P(y))

and show that it has all the necessary properties:

lemma Fraenkel_A1:
fixes F :: Set ⇒ Set **and** P :: Set ⇒ o
assumes [ex]:inhabited(L) sethood(L)
shows Fraenkel1(F, L, P) be set ∧
(∀x : object. x in Fraenkel1(F, L, P) ⟷ (∃y : L. x = F(y) ∧ P(y))) ∧
(x be set ∧ (∀xa : object. xa in x ⟷ (∃y : L. xa = F(y) ∧ P(y)))
⟶ x = Fraenkel1(F, L, P))

The main step in the proof, is to use the sethood property to show:

show ∃x : set. ∀xa : object. xa in x ⟷ (∃y : L. xa = F(y) ∧ P(y))

The comprehension operator Frankel1 defined above is limited to unary predicates and functions. We can extend it to more arguments recursively using Cartesian products. For this we re-formalize parts of the Mizar article `ZFMISC_1` which introduces such set theoretic products and use this to define set comprehensions for n-ary predicates and functions.

The formalized Theorem `Frankel_A1` together with the usual set comprehension notation allows the following example, which gives the image of the function f on the set X.

term {f. x where x be Element-of dom f: x in X}

Mizar additionally introduces an abbreviation for set comprehensions used without any property. the set of all $\mathtt{t}(\mathtt{v}_1,\mathtt{v}_2,\ldots,\mathtt{v}_n)$ where $\mathtt{v}_1$ is Θ_1, $\mathtt{v}_2$ is $\Theta_2,\ldots,\mathtt{v}_n$ is Θ_n abbreviates $\{\mathtt{t}(\mathtt{v}_1,\mathtt{v}_2,\ldots,\mathtt{v}_n)$ where $\mathtt{v}_1$ is Θ_1, $\mathtt{v}_2$ is $\Theta_2,\ldots,\mathtt{v}_n$ is Θ_2: not contradiction$\}$. We only show the corresponding Isabelle abbreviation in the following example:

theorem funct_1_th_110:
assumes [ty]:B be non empty|functional|set f be Function
and f = union B
shows
dom f = union the set-of-all dom g where g be Element-of B
rng f = union the set-of-all rng g where g be Element-of B

10 Beyond ZF

Many formalizations in the MML rely on the extension of the standard ZF axiomatization by the Tarski axiom. To complete the Mizar foundations in Isabelle, we express

the Tarski-Grothendieck axiom and re-derive a number of its consequences which will be used by the more advanced case studies.

The Tarski axiom states that for every set N, there exists a set M that contains N, is closed under subsets, power sets and each subset of M is a member of M or is equipotent with M. Its first statement is done in the last axiomatic Mizar article TARSKI_A. The statement refrains from using most derived definitions. Here, we only present the derived version of the axiom, reformulated using a power set construction:

reserve X,Y,N,M **for** set
mtheorem zfmisc_1_th_112:
∃M. N in M ∧ (∀X,Y. X in M ∧ Y ⊆ X ⟶ Y in M) ∧
(∀X. X in M ⟶ bool X in M) ∧
(∀X. X ⊆ M ⟶ X,M areequipotent ∨ X in M)

To actually obtain the universe M of a given set N, the Mizar article CLASSES1 [6] introduces a meta-level function Tarski-Class which returns the smallest set that satisfies the above predicates. We have not re-formalized the results from the CLASSES articles yet, as this relies on a substantial part of the Mizar library. We do, however, discuss some of its consequences here to explain the adequacy of our model when it comes to Mizar's set theoretical universes.

One of the consequences of the formulation of the axiom introduced by Tarski is the axiom of choice. This is also the case in Mizar. The Mizar theorems WELLORD2:17 and WELLORD2:18 use the Tarski axiom to derive a well-ordering of any set. Note, that it is possible to reformulate the axiom, so it does allow universes but without implying the axiom of choice. This is done for example by Brown in Egal [15], however in Isabelle/Mizar we avoid this modification to stay faithful to Mizar.

Furthermore, Tarski's axiom also implies the generalized axiom of infinity, which can be used to derive ordinals, as we will do in one of our case studies (Sect. 11.2), as well as the von Neumann hierarchy and universal classes in the Mizar articles CLASSES1, 2 by Bancerek [6]. First, one shows that it is possible to construct arbitrary ordinal-based sequences. Using their uniqueness it is possible to define the (von Neumann) rank of a set. Finally, various relations between such defined Ranks and universes (the function Tarski-Class) were proven. Another frequent use of the Tarski axiom in the MML is a simplified scheme for defining a function based on the function graph (relation) and the domain but without specifying the range (CLASSES1:sch 1). The proof of the scheme again needs to use the minimal ordinal assigned to any x in the domain.

11 Case studies

We have already shown excerpts from various Mizar articles re-formalized in Isabelle including the examples from set theory (Fig. 1), groups (Fig. 3), and categories. In this section we further want to argue that our model does not only correctly represent Mizar's foundations, but is also usable for practical Mizar-like formalization. For this we show small excerpts of various re-formalization case studies we have done.

Having performed these case studies, we can also give some observations about the usability of Isabelle/Mizar in comparison with the original Mizar system for practical formalization of mathematics. A major benefit of using Isabelle/Mizar is the fact, that there is only one article environment. Setting up article environments is a big effort for Mizar users hindering the start of formalizations. Isabelle also allows nice

Unicode notations and the configuration of notations in the formalization process. Currently, the Isabelle overloading mechanism is much weaker than that of Mizar. Also, a priority based grammar means that one needs more parentheses in Isabelle. Derived type information available in a theorem list makes it very easy to inspect the typing procedure. This is only available in the semantic Mizar form, which is not normally available to a user. The Mizar proof outlines are more flexible, making them easier to write, however they are harder to verify than the Isabelle ones. Some of the Mizar steps (notably `obtain`) involve a large syntax burden, when translating to Isabelle, as the Isar existential elimination is weaker than that of Mizar. Finally, the Mizar automation is still incomparable with the one we provide: in some cases it can be found by calls to Sledehammer, however in many cases manual instantiations are necessary.

11.1 Algebra

A field is defined in the Mizar library as a structure that inherits from nine other structures (e.g. additive and multiplicative groups). Inheritance between these structures concerns the signatures (independent types corresponding to A, $\oplus_A$, $\ominus_A$, 0_A, $\otimes_M$, and $/_M$) the actual group axioms (again types corresponding to the structure satisfying particular axioms), and the background information (reductions etc.).

We tested the usability of the proposed foundations by re-formalizing a selection of 15 structure types present in the articles `STRUCT_0`, `GROUP_1`, `RLVECT_1`, `ALGSTR_0`, `VECTSP_1`, which define all the ancestors of a field (`doubleLoopStr`), and the 50 main adjectives used in the field definitions, as well as the 16 basic binary and unary operations. This involved 146 formal proofs. Here we present a single statement which shows when a product of two field elements is zero. More properties and their proofs are included in the Isabelle/Mizar formalization.

reserve F **for** Field
reserve x,y **for** Element-of-struct F
mtheorem $\mathsf{x} \otimes_F \mathsf{y} = 0_F \longleftrightarrow \mathsf{x} = 0_F \vee \mathsf{y} = 0_F$

11.2 Ordinals and Natural Numbers

The Tarski axiom can be used to define the type of ordinals. Indeed in the MML and in our re-formalization ordinals are epsilon-transitive and epsilon-connected sets. We next show that the ordinals are ordered and that the ordinal successor returns an ordinal:

mtheorem ordinal1_th_10:
$\neg \mathsf{A}$ in B $\wedge$ A $\neq$ B $\longrightarrow$ B in A

We can then derive the transfinite induction (the scheme was already presented on page 12). The smallest ordinal that contains the empty set can be denoted as `omega`. By showing that `omega` has the sethood property, we finally obtain the natural numbers with the standard zero and successor operators. Natural number induction can be derived as a consequence of ordinal induction:

theorem ordinal_2_sch_19:
assumes [ty]: a be Nat

```
  and A1: P({})
  and A2: ∀ n : Nat. P(n) ⟶ P(succ n)
shows P(a)
```

11.3 Random Access Turing Machines

The *Simple Concrete Model* (SCM) [42] is a quite specific model of computers, designed to be nicely expressible in set theory. The Mizar formalization introduces a formal model corresponding to random-access Turing machines [24], their instructions, and programs. It has been considered to be a more realistic model than Turing machines for physical computers [50]. 66 Mizar articles concern SCMs and the proofs of various properties of this model.

The definition of `AMI-Struct` is the only structure in the MML where the specification uses three different selectors of the defined structure at the same time. All other structures use at most two selectors directly. The formalization uses both structures and set comprehension operators in non-trivial ways. We re-formalized the first few articles formalizing Mizar computers. We reached the theorem that shows that a `Trivial-AMI N` is of the computer type and that it does halt, which shows the non-emptiness of the Mizar type of computers:

```
theorem extpro_1:
  assumes [ty]: N be with_zero | set
  shows halt Trivial-AMI N be halting Trivial-AMI N, N
```

12 Related Work

Various other proof assistant foundations are based on set theory. Also certain Mizar features have been imitated in other systems and exports between Mizar and other systems were considered. In this section we discuss related works, dividing it in these categories.

12.1 Formalizations of Set Theory

Paulson [62] developed the Isabelle/ZF object logic based on the Zermelo-Fraenkel set theory. The foundational axioms used in ZF are different from the Tarski-Grothendieck ones used in Mizar. Furthermore, the logic is untyped, while Mizar relies on a precise type system. Finally, the Isabelle/ZF library and the automation provided are quite different from the Isabelle/Mizar emulation.

Zhan is developing an object logic based on untyped set theory together with the auto2 [87] automation package for Isabelle. The library tries to support even stronger automation that of Mizar, with no references to previously proved theorems at all. However, explicit instantiations for certain (especially second-order) theorems are necessary. Again the foundations, the library, and the automation are different from ours.

Agerholm and Gordon [3] were the first to represent set theory in HOL. The approach of adding set theory axioms on top of HOL was further extended by Obua first

in HOLZF [57] in Isabelle and later in the ProofPeer system, where the basic higher-order logic is minimal [58]. There, the support for automated distinguishing of object logic and meta-logic functions has been considered [59].

Kunčar [46] attempted to recover the Mizar type system in HOL Light. The considered approach worked well for the first few articles for the MML, but would require changes to accommodate for the more advanced concepts, such as dependent types.

The Egal [15] system starts with intuitionistic type theory adding a set theory variant that is very close to that of Mizar. It includes a formulation of the Tarski axiom as a function that explicitly returns a Grothendieck universe for a set. There is however no type system and the automation is very limited. Nevertheless, Brown was able to re-formalize large formal libraries in Egal, such as the Grundlagen [31].

Other large formalization systems based on set theory include the B-Method toolchain [1] as well as the Metamath [48] system.

12.2 Mizar Features in Other Systems

Harrison [30] has developed the first Mizar-style proof mode for a HOL-based proof assistant. He also created an imitation of the Mizar prover by in HOL Light without the Mizar types. There have been many attempts to create Mizar-like declarative proof modes since. Syme developed the *Declare* [67] prototype proof language for multiple higher-order provers. Wenzel's PhD work [79] brought the *Isar* declarative proof language for Isabelle. Today, it is not only a language for proofs, but also for complete interaction with the prover, with most users rarely interacting with the read-eval-print loop of the underlying LCF prover. Wiedijk developed *Mizar-light* [85] and *miz3* [86] for the HOL Light theorem prover.

In intuitionistic type theory, Corbineau [18] proposed the *C-zar* declarative proof language for Coq. The first author with Wiedijk discussed automated conversion of tactical proofs to declarative ones for Coq [39]. Lean [22] allows switching to the declarative proof mode in a tactical proof and back.

Theoretical properties of type systems with intersection types have been a field of study since the seventies [40]. This is however quite different from soft intersection types as the ones used in Mizar and our model of it.

12.3 Exports between Mizar and Other Systems

The first attempt at Mizar interpretation and use for first-order theorem proving was done by Dahn, Wernhard, and Byliński in the ILF project [20, 19].

The most comprehensive exports of Mizar data have been developed by Urban. Its first version (MPTP 0.1) [69] exported Mizar directly to first-order logic targeting the SPASS prover [78] using the DFG format [29]. This was followed by a complete XML export [70] of the Mizar semantic layer, which included an adaptation of Mizar to internally use XML. The next version of MPTP (0.2) [73] – has been based on the XML export. It natively uses the Prolog-based MPTP format, which extends the first-order TPTP by dependent types and other constructs of the Mizar logic. Recently a higher-order version of the data-set has been created, which includes all the schemes and set comprehension operators as simple type theory statements and is compatible with TH0 provers [16].

These have served as a basis for a number of experiments and tools in automated theorem proving and machine learning [75,76,4,38], ATP-based cross-verification [77] and explanation of Mizar proofs [74]. Custom exports of MML have been developed by Urban and Bancerek for semantic searching tools such as MoMM [56] and MML Query [10] and their integration in the MizarMode Emacs interface [71,11].

Iancu et al. [32] have exported all the statements from the Mizar Mathematical Library to the LF logic in the MMT logical framework. The export does not include the proofs or the semantics of the more advanced concepts, and its main purpose is to allow the use of various mathematical knowledge management services for the Mizar library. This includes browsing, search, and proof advice.

Another export of the Mizar library has been created from the Mizar parser intermediate representation, the Weakly Strict Mizar (WSX), to allow accessing the Mizar knowledge [56]. The use of this export combined with other stages of Mizar processing could be part of an automated translation from Mizar to Isabelle in future work.

Krauss and Schropp [43] have developed a proof-preserving translation from Isabelle/HOL to Isabelle/ZF. The interpretation included all basic Isabelle/HOL concepts including type classes and translated HOL type checking to explicit reasoning about set membership. This would in principle allow combining all the HOL libraries with the possibility to reason with the full power of set theory, however its performance was discouraging for larger HOL developments in practice.

13 Conclusions

We have defined the semantics of Mizar as an Isabelle object logic, developed a Mizar-like type inference mechanism and showed that it allows re-formalizing various parts of the Mizar Mathematical Library. Our formalization can be found at:

`http://cl-informatik.uibk.ac.at/cek/isabellemizar-jar2017.tgz`

It totals 662KB of Isabelle proofs and its re-formalized part of the MML includes 185 Mizar definitions (including types and meta-level functions) and 433 proofs.

In this paper we have not focused on the Mizar syntax, and a natural next step would be to also allow Mizar-like overloading, disambiguation, and other convenience mechanisms for reasoning with its type system. Furthermore, we would like to imitate other Mizar automation mechanisms, such as derivation of inhabitation information or background processing of properties. A further step would be to imitate the Mizar by proof justification, or provide a similar mechanism that would also correspond to the notion of obviousness in mathematics, including reasoning modulo equalities rather than the use of a simplifier. Finally, our re-formalization has been done manually so far. Extracting and combining the information from the various Mizar processing stages could allow (partially) automating the proof translation.

Acknowledgments

We would like to thank the anonymous reviewers, as well as Josef Urban, Chad Brown, and Julian Parsert for their comments on the previous versions of this paper. This work has been supported by the European Research Council (ERC) grant no. 714034 *SMART*, OeAD Scientific & Technological Cooperation with Poland grant, and the Polish National Science Center granted by decision n°DEC-2015/19/D/ST6/01473.

References

1. J. Abrial. *Modeling in Event-B - System and Software Engineering*. Cambridge University Press, 2010.
2. M. Adams. Proof auditing formalised mathematics. *J. Formalized Reasoning*, 9(1):3–32, 2016.
3. S. Agerholm and M. J. C. Gordon. Experiments with ZF set theory in HOL and Isabelle. In E. T. Schubert, P. J. Windley, and J. Alves-Foss, editors, *Higher Order Logic Theorem Proving and Its Applications*, volume 971 of *LNCS*, pages 32–45. Springer, 1995.
4. J. Alama, T. Heskes, D. Kühlwein, E. Tsivtsivadze, and J. Urban. Premise selection for mathematics by corpus analysis and kernel methods. *J. Autom. Reasoning*, 52(2):191–213, 2014.
5. A. Asperti, G. Bancerek, and A. Trybulec, editors. *Mathematical Knowledge Management (MKM 2004)*, volume 3119 of *LNCS*. Springer, 2004.
6. G. Bancerek. Tarski's classes and ranks. *Formalized Mathematics*, 1(**3**):563–567, 1990.
7. G. Bancerek, C. Byliński, A. Grabowski, A. Korniłowicz, R. Matuszewski, A. Naumowicz, and K. Pąk. The role of the Mizar Mathematical Library for interactive proof development in Mizar. *Journal of Automated Reasoning*, 2017.
8. G. Bancerek, C. Byliński, A. Grabowski, A. Korniłowicz, R. Matuszewski, A. Naumowicz, K. Pąk, and J. Urban. Mizar: State-of-the-art and Beyond. In M. Kerber, J. Carette, C. Kaliszyk, F. Rabe, and V. Sorge, editors, *Intelligent Computer Mathematics - International Conference, CICM 2015*, volume 9150 of *LNCS*, pages 261–279. Springer, 2015.
9. G. Bancerek and P. Rudnicki. A Compendium of Continuous Lattices in MIZAR. *J. Autom. Reasoning*, 29(3-4):189–224, 2002.
10. G. Bancerek and P. Rudnicki. Information retrieval in MML. In A. Asperti, B. Buchberger, and J. H. Davenport, editors, *Mathematical Knowledge Management, MKM 2003*, volume 2594 of *LNCS*, pages 119–132. Springer, 2003.
11. G. Bancerek and J. Urban. Integrated semantic browsing of the Mizar Mathematical Library for authoring Mizar articles. In A. Asperti, G. Bancerek, and A. Trybulec, editors, *Mathematical Knowledge Management (MKM 2004)*, volume 3119 of *LNCS*, pages 44–57. Springer, 2004.
12. B. Barras, C. Tankink, and E. Tassi. Asynchronous processing of Coq documents: From the kernel up to the user interface. In C. Urban and X. Zhang, editors, *Interactive Theorem Proving, ITP 2015*, volume 9236 of *LNCS*, pages 51–66. Springer, 2015.
13. J. C. Blanchette, D. Greenaway, C. Kaliszyk, D. Kühlwein, and J. Urban. A learning-based fact selector for Isabelle/HOL. *J. Autom. Reasoning*, 57(3):219–244, 2016.
14. J. C. Blanchette and T. Nipkow. Nitpick: A counterexample generator for higher-order logic based on a relational model finder. In M. Kaufmann and L. C. Paulson, editors, *Interactive Theorem Proving, ITP 2010*, volume 6172 of *LNCS*, pages 131–146. Springer, 2010.
15. C. E. Brown. *The Egal Manual*, 2014.
16. C. E. Brown and J. Urban. Extracting higher-order goals from the Mizar Mathematical Library. In M. Kohlhase, M. Johansson, B. R. Miller, L. de Moura, and F. W. Tompa, editors, *Intelligent Computer Mathematics (CICM 2016)*, volume 9791 of *LNCS*, pages 99–114. Springer, 2016.
17. C. Byliński. Introduction to categories and functors. *Formalized Mathematics*, 1(**2**):409–420, 1990.
18. P. Corbineau. A declarative language for the Coq proof assistant. In M. Miculan, I. Scagnetto, and F. Honsell, editors, *Types for Proofs and Programs, International Conference, TYPES 2007*, volume 4941 of *LNCS*, pages 69–84. Springer, 2007.
19. I. Dahn. Interpretation of a Mizar-like logic in first-order logic. In R. Caferra and G. Salzer, editors, *First-Order Theorem Proving (FTP 1998)*, volume 1761 of *LNCS*, pages 137–151. Springer, 1998.
20. I. Dahn and C. Wernhard. First order proof problems extracted from an article in the Mizar Mathematical Library. In M. P. Bonacina and U. Furbach, editors, *First-Order Theorem Proving (FTP 1997)*, RISC-Linz Report Series No. 97-50, pages 58–62. Johannes Kepler Universität, Linz (Austria), 1997.
21. M. Davis. Obvious logical inferences. In P. J. Hayes, editor, *International Joint Conference on Artificial Intelligence (IJCAI 1981)*, pages 530–531. William Kaufmann, 1981.
22. L. M. de Moura, S. Kong, J. Avigad, F. van Doorn, and J. von Raumer. The Lean theorem prover (system description). In A. P. Felty and A. Middeldorp, editors, *Conference on Automated Deduction, CADE 2015*, volume 9195 of *LNCS*, pages 378–388. Springer, 2015.

23. C. Dunchev, C. S. Coen, and E. Tassi. Implementing HOL in an higher order logic programming language. In G. Dowek, D. R. Licata, and S. Alves, editors, *Logical Frameworks and Meta-Languages Theory and Practice, LFMTP 2016*, pages 4:1–4:10. ACM, 2016.
24. C. C. Elgot and A. Robinson. Random-access stored-program machines, an approach to programming languages. *J. ACM*, 11(4):365–399, 1964.
25. A. P. Felty, E. L. Gunter, J. Hannan, D. Miller, G. Nadathur, and A. Scedrov. Lambda-Prolog: An extended logic programming language. In E. L. Lusk and R. A. Overbeek, editors, *International Conference on Automated Deduction, CADE*, volume 310 of *LNCS*, pages 754–755. Springer, 1988.
26. F. B. Fitch. *Symbolic Logic. An Introduction.* The Ronald Press Company, 1952.
27. A. Grabowski, A. Korniłowicz, and A. Naumowicz. Mizar in a nutshell. *J. Formalized Reasoning*, 3(2):153–245, 2010.
28. A. Grabowski, A. Korniłowicz, and A. Naumowicz. Four decades of Mizar. *Journal of Automated Reasoning*, 55(3):191–198, 2015.
29. R. Hähnle, M. Kerber, and C. Weidenbach. Common syntax of the DFGSchwerpunktprogramm deduction. Technical Report TR 10/96, Fakultät für Informatik, Universität Karlsruhe, Karlsruhe, Germany, 1996.
30. J. Harrison. A Mizar mode for HOL. In J. von Wright, J. Grundy, and J. Harrison, editors, *Theorem Proving in Higher Order Logics: TPHOLs 1996*, volume 1125 of *LNCS*, pages 203–220. Springer, 1996.
31. D. Hilbert. *Foundations of Geometry.* Open Court, 1971.
32. M. Iancu, M. Kohlhase, F. Rabe, and J. Urban. The Mizar Mathematical Library in OMDoc: Translation and applications. *J. Autom. Reasoning*, 50(2):191–202, 2013.
33. S. Jaśkowski. On the rules of suppositions. *Studia Logica*, 1, 1934.
34. C. Kaliszyk and K. Pąk. Isabelle formalization of set theoretic structures and set comprehensions. In J. Blamer, T. Kutsia, and D. Simos, editors, *Mathematical Aspects of Computer and Information Sciences, MACIS 2017*, volume 10693 of *LNCS*. Springer, 2017.
35. C. Kaliszyk and K. Pąk. Presentation and manipulation of Mizar properties in an Isabelle object logic. In H. Geuvers, M. England, O. Hasan, F. Rabe, and O. Teschke, editors, *Intelligent Computer Mathematics - CICM 2017*, volume 10383 of *LNCS*, pages 193–207. Springer, 2017.
36. C. Kaliszyk and K. Pąk. Progress in the independent certification of Mizar Mathematical Library in Isabelle. In M. Ganzha, L. A. Maciaszek, and M. Paprzycki, editors, *Proceedings of the 2017 Federated Conference on Computer Science and Information Systems, FedCSIS 2017*, pages 227–236, 2017.
37. C. Kaliszyk, K. Pąk, and J. Urban. Towards a Mizar environment for Isabelle: Foundations and language. In J. Avigad and A. Chlipala, editors, *Proc. 5th Conference on Certified Programs and Proofs (CPP 2016)*, pages 58–65. ACM, 2016.
38. C. Kaliszyk and J. Urban. MizAR 40 for Mizar 40. *J. Autom. Reasoning*, 55(3):245–256, 2015.
39. C. Kaliszyk and F. Wiedijk. Merging procedural and declarative proof. In S. Berardi, F. Damiani, and U. de'Liguoro, editors, *Types for Proofs and Programs, International Conference, TYPES 2008*, volume 5497 of *LNCS*, pages 203–219. Springer, 2008.
40. N. Kobayashi, editor. *Proceedings Eighth Workshop on Intersection Types and Related Systems, ITRS 2016*, volume 242 of *EPTCS*, 2017.
41. A. Korniłowicz. Flexary connectives in Mizar. *Computer Languages, Systems & Structures*, 44:238–250, 2015.
42. A. Korniłowicz and C. Schwarzweller. Computers and algorithms in Mizar. *Mechanized Mathematics and Its Applications*, 4(1):43–50, 2005.
43. A. Krauss and A. Schropp. A mechanized translation from higher-order logic to set theory. In M. Kaufmann and L. C. Paulson, editors, *Interactive Theorem Proving (ITP 2010)*, volume 6172 of *LNCS*, pages 323–338. Springer, 2010.
44. O. Kuncar and A. Popescu. A consistent foundation for Isabelle/HOL. In C. Urban and X. Zhang, editors, *Interactive Theorem Proving - 6th International Conference, ITP 2015*, volume 9236 of *LNCS*, pages 234–252. Springer, 2015.
45. O. Kuncar and A. Popescu. Safety and conservativity of definitions in HOL and Isabelle/HOL. *PACMPL*, 2(POPL):24:1–24:26, 2018.
46. O. Kunčar. Reconstruction of the Mizar type system in the HOL Light system. In J. Pavlu and J. Safrankova, editors, *WDS Proceedings of Contributed Papers: Part I – Mathematics and Computer Sciences*, pages 7–12. Matfyzpress, 2010.

47. G. Lee and P. Rudnicki. Alternative aggregates in Mizar. In M. Kauers, M. Kerber, R. Miner, and W. Windsteiger, editors, *Proc. 6th International Conference on Mathematical Knowledge Management (MKM 2007)*, volume 4573 of *LNCS*, pages 327–341. Springer, 2007.
48. N. D. Megill. *Metamath: A Computer Language for Pure Mathematics*. Lulu Press, Morrisville, North Carolina, 2007.
49. S. Merz. Mechanizing TLA in Isabelle. In R. Rodošek, editor, *Workshop on Verification in New Orientations*, pages 54–74, Maribor, 1995. Univ. of Maribor.
50. Y. Nakamura and A. Trybulec. A mathematical model of CPU. *Formalized Mathematics*, 3(**2**):151–160, 1992.
51. W. Naraschewski and M. Wenzel. Object-oriented verification based on record subtyping in higher-order logic. In J. Grundy and M. C. Newey, editors, *Theorem Proving in Higher Order Logics, 11th International Conference, TPHOLs'98*, volume 1479 of *LNCS*, pages 349–366. Springer, 1998.
52. A. Naumowicz. Enhanced processing of adjectives in Mizar. In A. Grabowski and A. Naumowicz, editors, *Computer Reconstruction of the Body of Mathematics*, volume 18(31) of *Studies in Logic, Grammar and Rhetoric*, pages 89–101. University of Białystok, 2009.
53. A. Naumowicz. Automating boolean set operations in Mizar proof checking with the aid of an external SAT solver. *J. Autom. Reasoning*, 55(3):285–294, 2015.
54. A. Naumowicz and C. Byliński. Improving Mizar texts with properties and requirements. In Asperti et al. [5], pages 290–301.
55. A. Naumowicz and A. Korniłowicz. A brief overview of Mizar. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *Theorem Proving in Higher Order Logics, TPHOLS*, volume 5674 of *LNCS*, pages 67–72, Berlin, Heidelberg, 2009. Springer-Verlag.
56. A. Naumowicz and R. Piliszek. Accessing the Mizar library with a weakly strict Mizar parser. In M. Kohlhase, M. Johansson, B. R. Miller, L. de Moura, and F. W. Tompa, editors, *Intelligent Computer Mathematics, CICM 2016*, volume 9791 of *LNCS*, pages 77–82. Springer, 2016.
57. S. Obua. Partizan games in Isabelle/HOLZF. In K. Barkaoui, A. Cavalcanti, and A. Cerone, editors, *Theoretical Aspects of Computing - ICTAC 2006*, volume 4281 of *LNCS*, pages 272–286. Springer, 2006.
58. S. Obua, J. D. Fleuriot, P. Scott, and D. Aspinall. ProofPeer: Collaborative theorem proving. *CoRR*, abs/1404.6186, 2014.
59. S. Obua, J. D. Fleuriot, P. Scott, and D. Aspinall. Type Inference for ZFH. In M. Kerber, J. Carette, C. Kaliszyk, F. Rabe, and V. Sorge, editors, *Intelligent Computer Mathematics - International Conference, CICM*, volume 9150 of *LNCS*, pages 87–101. Springer, 2015.
60. K. Ono. On a practical way of describing formal deductions. *agoya Mathematical Journal*, 21, 1962.
61. L. C. Paulson. Isabelle: The next 700 theorem provers. In P. Odifreddi, editor, *Logic and Computer Science (1990)*, pages 361–386, 1990.
62. L. C. Paulson. Set theory for verification: I. From foundations to functions. *J. Autom. Reasoning*, 11(3):353–389, 1993.
63. K. Pąk. Topological manifolds. *Formalized Mathematics*, 22(**2**):179–186, 2014.
64. F. Rabe. A logical framework combining model and proof theory. *Mathematical Structures in Computer Science*, 23(5):945–1001, 2013.
65. P. Rudnicki. Obvious Inferences. *J. Autom. Reasoning*, 3(4):383–393, 1987.
66. C. Schürmann. The Twelf Proof Assistant. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009*, volume 5674 of *LNCS*, pages 79–83. Springer, 2009.
67. D. Syme. Three tactic theorem proving. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin-Mohring, and L. Théry, editors, *Theorem Proving in Higher Order Logics, TPHOLs 1999*, volume 1690 of *LNCS*, pages 203–220. Springer, 1999.
68. A. Trybulec, A. Korniłowicz, A. Naumowicz, and K. T. Kuperberg. Formal mathematics for mathematicians - special issue. *J. Autom. Reasoning*, 50(2):119–121, 2013.
69. J. Urban. MPTP - motivation, implementation, first experiments. *J. Autom. Reasoning*, 33(3-4):319–339, 2004.
70. J. Urban. XML-izing Mizar: Making semantic processing and presentation of MML easy. In M. Kohlhase, editor, *Mathematical Knowledge Management (MKM 2005)*, volume 3863 of *LNCS*, pages 346–360. Springer, 2005.
71. J. Urban. MizarMode - an integrated proof assistance tool for the Mizar way of formalizing mathematics. *J. Applied Logic*, 4(4):414–427, 2006.

72. J. Urban. MoMM - Fast interreduction and retrieval in large libraries of formalized mathematics. *Int. J. on Artificial Intelligence Tools*, 15(1):109–130, 2006.
73. J. Urban. MPTP 0.2: Design, implementation, and initial experiments. *J. Autom. Reasoning*, 37(1–2):21–43, 2006.
74. J. Urban and G. Bancerek. Presenting and explaining Mizar. *Electr. Notes Theor. Comput. Sci.*, 174(2):63–74, 2007.
75. J. Urban, K. Hoder, and A. Voronkov. Evaluation of automated theorem proving on the Mizar Mathematical Library. In K. Fukuda, J. van der Hoeven, M. Joswig, and N. Takayama, editors, *International Congress on Mathematical Software (ICMS 2010)*, volume 6327 of *LNCS*, pages 155–166. Springer, 2010.
76. J. Urban, P. Rudnicki, and G. Sutcliffe. ATP and presentation service for Mizar formalizations. *J. Autom. Reasoning*, 50(2):229–241, 2013.
77. J. Urban and G. Sutcliffe. ATP-based cross-verification of Mizar proofs: Method, systems, and first experiments. *Math. in Computer Science*, 2(2):231–251, 2008.
78. C. Weidenbach, B. Afshordel, U. Brahm, C. Cohrs, T. Engel, E. Keen, C. Theobalt, and D. Topić. System description: SPASS version 1.0.0. In *Automated Deduction - CADE-16*, volume 1632 of *LNCS*, pages 378–382. Springer, 1999.
79. M. Wenzel. Isar - A generic interpretative approach to readable formal proof documents. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Théry, editors, *Theorem Proving in Higher Order Logics, 12th International Conference, TPHOLs 1999*, volume 1690 of *LNCS*, pages 167–184. Springer, 1999.
80. M. Wenzel. Asynchronous user interaction and tool integration in Isabelle/PIDE. In G. Klein and R. Gamboa, editors, *Interactive Theorem Proving, ITP 2014*, volume 8558 of *LNCS*, pages 515–530. Springer, 2014.
81. M. Wenzel. *The Isabelle/Isar Reference Manual*, 2017.
82. M. Wenzel, L. C. Paulson, and T. Nipkow. The Isabelle framework. In O. A. Mohamed, C. A. Muñoz, and S. Tahar, editors, *Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008*, volume 5170 of *LNCS*, pages 33–38. Springer, 2008.
83. M. Wenzel and F. Wiedijk. A comparison of Mizar and Isar. *J. Autom. Reasoning*, 29(3-4):389–411, 2002.
84. F. Wiedijk. CHECKER - notes on the basic inference step in Mizar. available at http://www.cs.kun.nl/∼freek/mizar/by.dvi, 2000.
85. F. Wiedijk. Mizar Light for HOL Light. In R. J. Boulton and P. B. Jackson, editors, *Theorem Proving in Higher Order Logics, TPHOLs 2001*, volume 2152 of *LNCS*, pages 378–394. Springer, 2001.
86. F. Wiedijk. A synthesis of the procedural and declarative styles of interactive theorem proving. *Logical Methods in Computer Science*, 8(1), 2012.
87. B. Zhan. Formalization of the fundamental group in untyped set theory using auto2. In M. Ayala-Rincón and C. A. Muñoz, editors, *Interactive Theorem Proving - ITP 2017*, volume 10499 of *LNCS*, pages 514–530. Springer, 2017.

A Dictionary of Mizar Jargon

In this paper we tried to express Mizar concepts using common type-theoretic expressions. Here, we provide a dictionary of expressions used in the Mizar jargon and some Mizar literature.

- **Fraenkel operator.** set comprehension operator
- **permissive definition.** a definition with a precondition
- **Mizar type.** a type in an intersection type system
- **mother type.** parent type
- **attribute.** a (soft) type
- **adjective.** an attribute or its complement (negation)
- **mode.** an inhabited type
- **radix type.** a basic inhabited type
- **parametric type.** a dependent type
- **functor.** a meta-level function
- **cluster registration.** a type inference rule proved by the user
- **rounding-up adjectives.** type inference using the available clusters
- **expandable mode.** a type abbreviation
- **background knowledge.** a typing environment