MIZAR'S TARSKI-GROTHENDIECK AS A THEORY IN PROOFGOLD

CHAD E. BROWN

Abstract. We describe a Proofgold theory inspired by Mizar's formulation of Tarski-Grothendieck set theory. We further discuss ways to mimic various Mizar features within the Proofgold theory.

1. Introduction

Mizar [2] is one of the oldest and longest lived systems for formalizing mathematics. It is also one of the few systems based on set theory instead of some variant of type theory. The Mizar Mathematical Library (MML) [1] at one time contained more formalized mathematics than any other system and is still one of the largest. If we wish to translate the MML into the Proofgold blockchain, we first need to give an appropriate Proofgold theory. A Proofgold theory consists of finitely primitive typed constants (extending the language of typed terms) and finitely many axioms (extending provability of intuitionistic higher-order logic). More details about Proofgold theories can be found in [4].

The Proofgold theory we need should form a foundation giving a form of Tarski-Grothendieck set theory (TG), upon which the MML is based. Different versions of TG in simple type theory have been discussed in recent publications. In [9] a version of TG is given as an Isabelle object logic. In this case the object logic is intended to model Mizar closely. A different version of TG forms the foundation of Egal [6, 5, 3]. An easy way to distinguish between Mizar's style and Egal's style is the treatment of universes. In Mizar (Tarski-style) universes exist due to Tarski's Axiom A [14]. In Egal (Grothendieck-style) universes [8] exist due to a primitive operator giving a universe (closed under the ZF constructors) containing a given set. As discussed in [6] the Mizar style axioms can be proven in Egal and variants of the Egal axioms can be proven in Mizar. This does not quite make the systems equivalent since the Egal version is properly higher-order (so that, for example, Fraenkel's Replacement axiom holds for all higher-order formulas, not just first-order formulas), but does indicate that the two systems behave similarly for most practical purposes.

When giving TG as a Proofgold theory we could use either formulation.[1] We have chosen to use a formulation close to Mizar's simply due to Mizar being more widely

Date: August 30, 2020.

Czech Technical University in Prague.

[1]To be more precise, the Egal foundation has a choice operator at every type. This is not possible in a Proofgold theory since Proofgold does not support polymorphism. Instead one could only give a choice operator at the base type of sets, which is arguably enough for practical use cases.

known and used than Egal. Unlike [9] we do not try to mimic Mizar too closely. For example, we do not have a type of Mizar types. Instead we use the type of predicates over sets as Mizar types.

2. Mizar's TG as a Proofgold Theory

In this section we give a presentation of the Mizar TG theory in Proofgold. We will use Megalodon's syntax for the presentation instead of Proofgold's, as Megalodon's syntax is more human readable. (Megalodon is a new generation of the Egal prover [6, 3].) Megalodon can be used to author documents that can be translated into Proofgold's format.

Let ι be the base type of individuals which are intended to range over sets. In ASCII we write ι as `set`. As always we have a type o of propositions which we write in ASCII as `prop`.

The Mizar system includes a `the` operator which chooses a member of a given Mizar type. This is a form of the axiom of choice.[2] Note that Mizar types must be nonempty. In our Proofgold theory we give a primitive `the` operator.

```
Parameter the : (set->prop)->set.
```

The axiom for this operator needs to have a condition that the predicate is nonempty since unlike Mizar types there is (of course) an empty predicate.

```
Axiom the_ax : forall P:set->prop, forall x:set, P x -> P (the P).
```

The logic of Proofgold includes implication and universal quantification by default. Furthermore, there is notation for equations and existential quantification at each type. For a type α and terms s, t of type α, $s = t$ is notation for $\forall p : \alpha\alpha o.pst \to pts$, i.e., symmetric Leibniz equality [4]. For a type α, variable x of type α and a term s of type o, $\exists x.s$ is notation for $\forall p : o.(\forall x : \alpha.s \to p) \to p$. In order to state the remaining axioms more naturally, we make the follow Russell-Prawitz style definitions [13, 11] of the logical connectives $\bot$ (`False`), $\neg$ (`not`), $\wedge$ (`and`), $\vee$ (`or`) and $\leftrightarrow$ (`iff`). In addition we declare infix notation for $\wedge$, $\vee$ and $\leftrightarrow$. (The notation only affects the Megalodon presentation and is note part of the Proofgold file.)

```
Definition False : prop := forall p:prop, p.
Definition not : prop -> prop := fun A:prop => A -> False.
Definition and : prop -> prop -> prop
   := fun A B:prop => forall p:prop, (A -> B -> p) -> p.
Infix /\ 780 left := and.
Definition or : prop -> prop -> prop
   := fun A B:prop => forall p:prop, (A -> p) -> (B -> p) -> p.
Infix \/ 785 left := or.
Definition iff : prop -> prop -> prop
   := fun A B:prop => (A -> B) /\ (B -> A).
Infix <-> 805 := iff.
```

[2] Without the `the` operator, Mizar already has a form of the axiom of choice as consequence of Tarski's Axiom A.

The next axiom does not have a counterpart in Mizar and could be omitted. The main reason we include it is so that we can prove an η-law for structures (see Section 6). The axiom is propositional extensionality and says two propositions are equal if they are equivalent. Since Mizar is essentially first-order, it is not possible to express an equation between to propositions.

```
Axiom prop_ext : forall p q:prop, (p <-> q) -> p = q.
```

The next typed primitive is membership of type ιo. We will write $\in$ (in infix) mathematically and `:e` (in infix) in ASCII. We also write $\forall x \in s.t$ as shorthand for $\forall x.x \in s \to t$ and write $\exists x \in s.t$ as shorthand for $\exists x.x \in s \wedge t$. We also use $\notin$ and `/:e` for the negated versions of membership.

```
Parameter In:set->set->prop.
```

From this point on the primitives, definitions and axioms closely mimic Mizar's Tarski-Grothendieck Set Theory article [15] (although we present them in a different order).

We define $\subseteq$ of type ιo using membership in the obvious way. We write $\subseteq$ in infix and write `c=` (in infix) in ASCII.

```
Definition Subq : set -> set -> prop
  := fun A B => forall x :e A, x :e B.
```

The third primitive is of type $\iota\iota$ and takes two sets x and y and forms the unordered pair $\{x, y\}$. We use the notation $\{x, y\}$ (also in ASCII) for this primitive applied to x and y.

```
Parameter UPair : set -> set -> set.
Notation SetEnum2 UPair.
```

We define the singleton operation of type $\iota\iota$ to take x and return $\{x, x\}$. We write $\{x\}$ (also in ASCII) for the singleton operation applied to x.

```
Definition Sing : set -> set := fun x => {x,x}.
Notation SetEnum1 Sing.
```

The final primitive is the union operation and has type $\iota\iota$. We write $\bigcup X$ for this primitive applied to X.

```
Parameter Union : set->set.
```

In order to state Axiom A, we will need a notion of equipotence. In the MML this is defined using Kuratowski pairs as ordered pairs. For this reason we define a Kuratowski pair operation of type $\iota\iota\iota$ taking two sets x and y to the set $\{\{x, y\}, \{x\}\}$.

```
Definition KPair : set -> set -> set := fun x y => {{x,y},{x}}.
```

We now define the equipotence relation between two sets X and Y as holding when there is a set Z that acts as the graph encoding of a bijection between X and Y.

```
Definition equip : set -> set -> prop := fun X Y =>
  exists Z,
      (forall x :e X, exists y :e Y, KPair x y :e Z)
   /\ (forall y :e Y, exists x :e X, KPair x y :e Z)
   /\ (forall x y z u, KPair x y :e Z -> KPair z u :e Z
             -> (x = z <-> y = u)).
```

This completes the set of typed constants and definitions. From now on we only state axioms.

Set extensionality states two sets are equal if they have the same elements.

```
Axiom setext : forall X Y:set, (forall x, x :e X <-> x :e Y) -> X = Y.
```

The axiom of unordered pairs states that $x \in \{y, z\}$ if and only if $x = y$ or $x = z$.

```
Axiom UPair_ax : forall x y z, x :e {y,z} <-> x = y \/ x = z.
```

The union axiom states that $x \in \bigcup X$ if and only if there is a set Y such that $x \in Y$ and $Y \in X$.

```
Axiom Union_ax : forall X x, x :e Union X <-> exists Y, x :e Y /\ Y :e X.
```

The regularity axiom states that if X is nonempty then there is a member Y of X such that X and Y are disjoint.

```
Axiom Regularity : forall X x, x :e X
  -> exists Y :e X, not (exists x :e X, x :e Y).
```

Fraenkel's replacement axiom is a scheme in first-order set theories (and in particular in Mizar [15]) but can be stated as a single axiom in a higher-order setting. It says for every set A (of type ι) and every binary relation P (of type $\iota\iota o$), if P is functional, then there is a set X such that $x \in X$ if and only if there is a $y \in A$ such that P relates y to x.

```
Axiom Replacement : forall A, forall P:set -> set -> prop,
    (forall x y z, P x y -> P x z -> y = z)
 -> exists X, forall x, x :e X <-> exists y :e A, P y x.
```

The final axiom is Tarski's Axiom A. Axiom A is powerful enough to replace the usual ZFC axioms of infinity, power set and choice. It states that for every set N there is a Tarski universe M such that $N \in M$.

```
Axiom Tarski_A : forall N, exists M,
    N :e M
 /\ (forall X Y, X :e M -> Y c= X -> Y :e M)
 /\ (forall X, X :e M -> exists Z :e M, forall Y, Y c= X -> Y :e Z)
 /\ (forall X, X c= M -> equip X M \/ X :e M).
```

This completes the set of axioms for the theory and so completely describes the intended Proofgold theory. Since the underlying logic is intuitionistic, it may be surprising that there is no explicit axiom making the theory classical. The theory is provably classical via a standard Diaconescu style argument (making use of `the` and `the_ax`) [12, 10].

When publishing a theory into the Proofgold chain, a certain number of Proofgold bars must be burned (the amount depends linearly on the serialized size of the theory). The Mizar-inspired theory described here was published in a transaction[3] on August 4, 2020, that also burned 5.586 bars.

[3]Txid: f69d8c1530488832c630b6274ca86e16fc740d866575fa58012a032c05075d0a

3. Definitions

There are several types of definitions supported by Mizar. We discuss the definitions of attributes and functions, leaving modes (basic Mizar types) and adjectives (modifiers on types) to Section 4.

Definitions of attributes are easily mapped to the contexts of Megalodon and Proofgold. Assume the Mizar attribute is defined by giving n typed variables $x_1, \ldots, x_n$ each of Mizar type $\varphi_1, \ldots, \varphi_n$. Here φ_i is considered as a predicate over sets (i.e., of type ιo) and φ_i may contain free occurrences of $x_1, \ldots, x_{i-1}$. The `means` keyword separates the attribute being defined (on the left) from its definition. Its definition is formed from preexisting language and so we can assume it can be translated as ψ of type o. For the corresponding Megalodon/Proofgold definition, we define the attribute as having type $\iota \cdots \iota o$ (with n arguments) with definition $\lambda x_1 \cdots x_n.\psi$. Note that the types φ_i are omitted entirely. The fact that the predicates must be provable in any context in which the attribute is used is an invariant that would need to be externally maintained.

An early example of an attribute is `empty` defined in [7]. In Mizar syntax this is given as follows:

```
definition
  let X be set;
  attr X is empty means
  :Def1:
  not ex x being set st x in X;
end;
```

An analogous Megalodon definition looks as follows:

```
Definition empty : set -> prop
 := fun X => forall x, x /:e X.
```

Obviously it would be closer to use `not (exists x, x :e X)`, but the variants are equivalent.

Some defined functions can be handled the same way as attributes, namely those using the `equals` keyword. Functions can also be defined using the `means` keyword where to the right of `means` is a proposition depending on a keyword `it`. These can be handled by the `the` operator in Megalodon/Proofgold. Suppose there are n input variables $x_1, \ldots, x_n$ (of type ι). We can translate the proposition to the right of `means` as a proposition ψ depending on $x_1, \ldots, x_n$ and an extra variable y of type ι (playing the role of `it`). Now the new function can be defined as

$$\lambda x_1 \cdots x_n.\texttt{the}\ (\lambda y.\psi).$$

In Mizar one would need to prove this definition is well-formed. In Megalodon/Proofgold, the definition can be made without proving it has the properties associated with being well-formed. However, proving those properties in Megalodon/Proofgold would be essential in order to use the definition as intended.

4. Types and Sethood

Modes (defining basic Mizar types) can be handled in much the way attributes are handled. Suppose the mode depends on variables $x_1, \ldots, x_n$ (of Mizar types $\varphi_1, \ldots, \varphi_n$) and holds when ψ holds for a variable y. (The y may be playing the role of the `it` keyword in some mode definitions.) The mode itself would be translated as $\lambda x_1 \cdots x_n y.\psi$. An extra invariant of proving nonemptiness would need to be (externally) enforced. To be precise, assuming the Mizar type of each x_i corresponds to the predicate φ_i, the nonemptiness property would be

$$\forall x_1.\varphi_1\ x_1 \to \cdots \forall x_n.\varphi_n\ x_n \to \exists y.\psi.$$

One of the most used (dependent) types in the MML is `Element of` defined in [16]. By applying `Element of` to a set X, we obtain a type of elements of X (assuming X is nonempty). Since all Mizar types must be inhabited, `Element of` X is defined to contain "every" empty set if X is empty. In Mizar the definition looks as follows (omitting proofs of correctness):

```
definition
  let X;
  mode Element of X means :Def1:
  it in X if X is non empty otherwise it is empty;
  ...
end;
```

A corresponding Megalodon definition looks as follows:

```
Definition Element_of : set -> set -> prop
 := fun X x => x :e X \/ empty X /\ empty x.
```

It would perhaps be closer to include a conjunct saying X is not empty in the first disjunct, but the two variants are easily seen to be equivalent. The key theorem to prove to maintain an invariant that predicates corresponding to Mizar types are inhabited is the following:

```
Theorem Element_of_nonempty : forall X, exists x, Element_of X x.
```

This theorem is easy to prove by a case analysis (using excluded middle).

Adjectives take a Mizar type and return a refined Mizar type. This can be translating by defining a corresponding operation of type $(\iota o)\iota o$ that takes a predicate and returns a subpredicate (by conjoining the new property). In case the adjective depends on some variables, the type may have the form $\iota \cdots \iota(\iota o)\iota o$.

In some cases Mizar types are "small" in the sense that there is a set containing all the objects satisfying the type as elements. Since Mizar types are considered as predicates, the notion of being "small" can be defined as a predicate on predicates. We call this `setlike` (of a predicate) here, but it is called `sethood` (of a type) in Mizar.

```
Definition setlike : (set -> prop) -> prop
 := fun P => exists A, forall x, x :e A <-> P x.
```

Every predicate can be coerced into being a set using the `the` operator.

```
Definition toset : (set -> prop) -> set
 := fun P => the (fun A => forall x, x :e A <-> P x).
```

The set given by `toset` satisfies the expected property if the predicate satisfies the `setlike` property.

```
Theorem setlike_toset : forall P:set -> prop, setlike P ->
 forall x, x :e toset P <-> P x.
```

This theorem has been proven in Megalodon (starting from the theory in Section 2) and published as part of a document in the Proofgold blockchain. All the remaining definitions and theorems we describe have also been published into the Proofgold blockchain.

Key examples of Mizar types that are small are those given by `Element of`. This is recorded by the following Megalodon theorem:

```
Theorem setlike_Element_of : forall X, setlike (Element_of X).
```

It is also easy to prove that for nonempty sets X, `toset` is a one-sided inverse to `Element_of`.

```
Theorem setof_Element_of : forall X, ~empty X -> toset (Element_of X) = X.
```

5. Fraenkel Terms

One way Mizar extends the traditional language of first-order logic is by allowing some term level binders. In particular, so-called *Fraenkel terms* are permitted as terms (of Mizar type `set`). These are written as

$$\{s \texttt{ where } x_1 \texttt{ is } \varphi_1, \ldots, x_n \texttt{ is } \varphi_n \texttt{ : } \psi\}.$$

Here the types φ_i of the bound variables must be small. Considering φ_i as a predicate, this means we need it to satisfy the `setlike` property. In Section 4 we have defined a coercion `toset` that turns a predicate into a set, giving the expected set when the predicate is small. We will make use of this here to translate each Fraenkel term as

$$\mathsf{ReplSep}_n\ (\texttt{toset}\ \varphi_1)\ (\lambda x_1.\texttt{toset}\ \varphi_2)$$
$$\cdots\ (\lambda x_1 \cdots x_{n-1}.\texttt{toset}\ \varphi_n)\ (\lambda x_1 \cdots x_n.\psi)\ (\lambda x_1 \cdots x_n.s).$$

For practical purposes we limit ourselves to $1 \leq n \leq 7$, but the pattern makes it clear that the definition of $\mathsf{ReplSep}_n$ generalizes to arbitrary values of n.

We first need a replacement operator. To define this we first prove sets given by replacing the elements of A with the values given by a meta-function $F : \iota\iota$ exist. This easily follows from Fraenkel's replacement axiom.

```
Theorem Repl_ex : forall A:set, forall F:set -> set, exists B:set,
 forall y, y :e B <-> exists x :e A, y = F x.
```

With the previous theorem we are justified in defining a Repl operator of type $\iota(\iota\iota)\iota$ using `the`.

```
Definition Repl : set -> (set -> set) -> set := fun A F =>
 the (fun B => forall y, y :e B <-> exists x :e A, y = F x).
```

We use the notation $\{s|x \in t\}$ for $\mathsf{Repl}\ t\ (\lambda x.s)$. It is easy to prove (using `the_ax` and `Repl_ex`) that the members of $\{Fx|x \in A\}$ are the expected members.

```
Theorem Repl_iff : forall A:set, forall F:set->set,
 forall y:set, y :e {F x|x :e A} <-> exists x :e A, y = F x.
```

Fraenkel's replacement axiom also allows us to easily prove Zermelo's separation axiom.

```
Theorem Sep_ex : forall A:set, forall P:set -> prop, exists B:set,
 forall x, x :e B <-> x :e A /\ P x.
```

Using `the` we can define a Sep operator of type $\iota(\iota o)\iota$

```
Definition Sep : set -> (set -> prop) -> set
 := fun A P => the (fun B => forall x, x :e B <-> x :e A /\ P x).
```

We use the notation $\{x \in s|\psi\}$ for Sep s $(\lambda x.\psi)$. It is easy to prove $\{x \in s|\psi\}$ has the right members.

```
Theorem Sep_iff : forall A, forall P:set -> prop,
 forall x, x :e {x :e A|P x} <-> x :e A /\ P x.
```

We next combine the replacement and separation operators as ReplSep of type $\iota(\iota o)(\iota\iota)\iota$. We write $\{s|x \in t, \psi\}$ for ReplSep t $(\lambda x.\psi)$ $(\lambda x.s)$.

```
Definition ReplSep : set->(set->prop)->(set->set)->set
 := fun A P F => {F x|x :e {z :e A|P z}}.
```

It should be clear that ReplSep can play the role of $\mathsf{ReplSep}_1$.

We can define $\mathsf{ReplSep}_2$ by combining ReplSep with Repl and the union operation as follows:

```
Definition ReplSep2 : set -> (set -> set) -> (set -> set -> prop)
                          -> (set -> set -> set) -> set
 := fun A B P F => Union {{F x y|y :e B x, P x y} | x :e A}.
```

This generalizes to a scheme for defining $\mathsf{ReplSep}_{n+1}$ from $\mathsf{ReplSep}_n$, Repl and union. For example, here is the definition of $\mathsf{ReplSep}_7$ assuming the definition of $\mathsf{ReplSep}_6$ has already been made:

```
Definition ReplSep7 : set -> (set -> set) -> (set -> set -> set)
 -> (set -> set -> set -> set) -> (set -> set -> set -> set -> set)
 -> (set -> set -> set -> set -> set -> set)
 -> (set -> set -> set -> set -> set -> set -> set)
 -> (set -> set -> set -> set -> set -> set -> set -> prop)
 -> (set -> set -> set -> set -> set -> set -> set -> set) -> set
 := fun A B C D E G H P F =>
 Union {ReplSep6 (B x) (C x) (D x) (E x) (G x) (H x) (P x) (F x)|x :e A}.
```

6. Structures

Structure types are a different kind of Mizar type as those considered in Section 4. A structure type in Mizar is given by sequence of typed labels. Each of these labels gives a function, called a *selector*, from structures to a value of the type corresponding to the label. Structures generally are allowed to have more information than specified in the structure type, allowing for future structure types to inherit from multiple previous structure types. Structures may also be *strict*, essentially meaning they have no extra information.

In [9] Mizar structures are represented by partial functions (represented by sets) from labels to values. Finite ordinals are used as labels, providing an infinite set of labels that are provably distinct, but any such infinite set would suffice. Here we describe a generalization of the approach to representing structures (and structure types) given in [9].[4]

Let us begin by considering four simple Mizar structure types.

Example 6.1. *1-sorted structures have only one label* `carrier` *associated with type set. This means a 1-sorted structure determines a set and this set can be obtained by applying the selector function* `carrier` *to a 1-sorted structure. In Mizar notation, the 1-sorted structure type is defined as follows:*

```
definition
  struct 1-sorted(# carrier -> set #);
end;
```

Example 6.2. *A ZeroStr structure is a 1-sorted structure with an additional piece of information: an element of the carrier (or, more properly, a value of type* `Element of` *applied to the carrier set of the structure). The label corresponding to this label is* `ZeroF` *and* `ZeroF` *is also the name of the selector taking a ZeroStr structure X to the corresponding value of type* `Element of` X*. In Mizar notation the ZeroStr structure type is defined as follows:*

```
definition
  struct (1-sorted) ZeroStr
  (# carrier -> set, ZeroF -> Element of the carrier #);
end;
```

Example 6.3. *A OneStr structure is the same as a ZeroStr structure except the label and selector function is given by* `OneF` *instead of* `ZeroF`*. The Mizar definition is the obvious one:*

```
definition
  struct (1-sorted) OneStr
  (# carrier -> set, OneF -> Element of the carrier #);
end;
```

It may be suprising that ZeroStr and OneStr structures are defined separately, as clearly the two kinds of structures are essentially the same. However, by defining them independently, we can combine them with multiple inheritance in the next example.

Example 6.4. *A ZeroOneStr structure is both a ZeroStr and a OneStr. That is, we have a carrier set and two elements (both of which might be the same). The selector for the carrier set is* `carrier`*, the selector for the zero element is* `ZeroF` *and the selector for the one element is* `OneF`*. The Mizar definition looks as follows:*

[4]Karol Pąk considered different ways of emulating Mizar structures in Isabelle and some of the ideas presented here are inspired by ideas discussed by Pąk in a talk in Prague in 2016 predating the publication of [9].

```
definition
  struct (ZeroStr,OneStr)
    ZeroOneStr(# carrier -> set,
                 ZeroF -> Element of the carrier,
                 OneF -> Element of the carrier #);
end;
```

For ease of explanation, from now on we only consider structures whose carrier set is nonempty when convenient (so that having type `Element of` the carrier set is the same as being a member of the carrier set). This allows us to essentially omit references to `Element of` below.

Let us consider (mathematically) how these four kinds of structures would be represented in the manner considered in [9]. Labels correspond to finite ordinals, so suppose we take 2 to represent the label `carrier`, 3 to represent the label `ZeroF` and 5 to represent the label `OneF`. Structures are partial functions from labels to appropriate values. Let us assume that partial functions are represented as sets of Kuratoski pairs, written $[u, v]$. That is, if the Kuratowski pair $[u, v]$ is in the partial function, then the partial function maps u to v. A 1-sorted structure would be any partial function X on ω with $[2, U] \in X$ for some U. A ZeroStr structure would be any partial function X on ω with $[2, U] \in X$ and $[3, y] \in X$ for some U and $y \in U$. A OneStr structure would be any partial function X on ω with $[2, U] \in X$ and $[5, z] \in X$ for some U and $z \in U$. A ZeroOneStr structure would be any partial function X on ω with $[2, U] \in X$, $[3, y] \in X$ and $[5, z] \in X$ for some U and $y, z \in U$. It is easy to see that every ZeroOneStr structure is both a ZeroStr and a OneStr and that every ZeroStr structure and every OneStr structure is a 1-sorted structure. The three selector (meta-)functions can be written as follows:

$$\texttt{carrier} := \lambda S.\texttt{ap}\ S\ 2$$

$$\texttt{ZeroStr} := \lambda S.\texttt{ap}\ S\ 3$$

$$\texttt{OneStr} := \lambda S.\texttt{ap}\ S\ 5$$

Here `ap` is application of a set level function to an argument. We assume `ap` gives some arbitrary (unimportant) value (e.g., $\emptyset$) when applied to an argument outside its domain.

If we were to follow the same approach to representing structures, we would start by defining finite ordinals and the `ap` function of type $\iota\iota$. Each of our selectors would then have type $\iota\iota$. We instead will examine what abstract properties selectors of the form $\lambda S.\texttt{ap}\ S\ n$ have and use this to define an *orthogonal collection of selectors*. We will then use orthogonal collections of selectors to define structures and structure types.

Consider the two functions $\texttt{carrier} := \lambda X.\texttt{ap}\ X\ 2$ and $\texttt{ZeroStr} := \lambda X.\texttt{ap}\ X\ 3$ above. For every pair of values u and v, there is a partial function S (namely $\{[2, u], [3, v]\}$) such that $\texttt{carrier}\ S = u$ and $\texttt{ZeroStr}\ X = v$. Likewise, for every triple of values u, v and w, there is a partial function X such that $\texttt{carrier}\ X = u$, $\texttt{ZeroStr}\ X = v$ and $\texttt{OneStr}\ X = w$. These previous two properties are clearly similar, although one mentions pairs and the other mentions triples. We can unify them by specifying the "tuple" of values we want the selectors to return by giving the selectors themselves as

arguments of the "tuple" (making use of the fact that we are in a higher-order setting). In other words, instead of giving a pair of values u and v, we give a single value x of type $(\iota\iota)\iota$ such that x `carrier` is u and x `ZeroStr` is v. Now the relevant property of the two selectors states that for every such x, there is a partial function X such that `carrier` $X = x$ `carrier` and `ZeroStr` $X = x$ `ZeroStr`. In general, given a collection $\mathcal{F}$ of type $(\iota\iota)o$, we say $\mathcal{F}$ is an *orthogonal collection of selectors (ocs)* if for every x of type $(\iota\iota)\iota$ there is a set X such that $f\ X = x\ f$ for every $f \in \mathcal{F}$. Intuitively, we consider $(x\ f)_{f\in\mathcal{F}}$ to be an $\mathcal{F}$-indexed family of values and the ocs condition means we can always glue this family together into a single value X (of type ι). It should be clear that every collection of functions of the form $\lambda X.\mathsf{ap}\ X\ n$ (where n ranges over ω) will be an ocs. However, we do not need to commit to using finite ordinals as labels or, indeed, to using partial functions to represent structures at all.

In Megalodon we define ocs as follows:

```
Definition ocs : ((set -> set) -> prop) -> prop :=
  fun F : (set -> set) -> prop =>
    forall x : (set -> set) -> set, exists X : set,
    forall f:set -> set, F f -> f X = x f.
```

We can now define a generic constructor for structures mk_struct that takes an ocs $\mathcal{F}$ and a $\mathcal{F}$-indexed family x and returns some set gluing the family together. We use the `the` operator to choose the representative.

```
Definition mk_struct : ((set -> set) -> prop) -> ((set -> set) -> set) -> set
  := fun F:(set -> set) -> prop => fun x:(set -> set) -> set =>
     the (fun X => forall f:set -> set, F f -> f X = x f).
```

Assuming $\mathcal{F}$ is an ocs, we can prove that each selector $f \in \mathcal{F}$ does return $x\ f$ when f is applied to the value given by mk_struct.

```
Theorem ocs_mk_struct_prop : forall F:(set -> set) -> prop, ocs F ->
 forall f:set -> set, F f ->
 forall x:(set -> set) -> set, f (mk_struct F x) = x f.
```

For strictness of structures, we simply need to know that it is a unique representive giving the appropriate values. This can be defined by saying X is a strict $\mathcal{F}$-structure if it is the $\mathcal{F}$-structure given by mk_struct applied to the $\mathcal{F}$-indexed family $\lambda f.f\ X$. In Megalodon, the definition is given as follows:

```
Definition strict_struct_p : ((set -> set) -> prop) -> set -> prop
 := fun F:(set -> set) -> prop =>
    fun X:set =>
      X = mk_struct F (fun f => f X).
```

We can prove the following η law, making use of propositional extensionality in the proof.

```
Theorem mk_struct_eta : forall F:(set -> set) -> prop, ocs F ->
   forall X:set, mk_struct F (fun f => f X)
               = mk_struct F (fun f => f (mk_struct F (fun f => f X))).
```

From the η law it is easy to prove that structures of the form $\mathsf{mk\_struct}\ \mathcal{F}\ (\lambda f.f\ X)$ are always strict. In essence, we can make every structure X into a strict $\mathcal{F}$-structure by applying this operation.

```
Theorem strict_mk_struct : forall F:(set -> set) -> prop, ocs F ->
 forall X:set, strict_struct_p F (mk_struct F (fun f => f X)).
```

We consider two $\mathcal{F}$-structures equivalent whenever they give the same values on all the selectors $f \in \mathcal{F}$. It is easy to prove this is an equivalence relation (given $\mathcal{F}$).

```
Definition equ_struct : ((set -> set) -> prop) -> set -> set -> prop :=
  fun F:(set -> set) -> prop => fun X Y:set =>
   forall f:set -> set, F f -> f X = f Y.
```

Let us finally consider how Examples 6.1, 6.2, 6.3 and 6.4 can be represented. We start by opening a Megalodon section and declaring the *carrier* selector as a variable (instead of defining it as above).

```
Variable carrier : set -> set.
```

Because we will reuse this variable in many contexts, we open a subsection.

```
Section struct_1sorted_sec_b.
```

We locally define *sels* as the singleton collection containing *carrier*. This will play the role of $\mathcal{F}$ for the definition of 1-sorted structures. We will assume as a local hypothesis that *sels* is an ocs.[5]

```
Let sels : (set -> set) -> prop
  := (fun f:set -> set => f = carrier).
Hypothesis ocs_carrier : ocs sels.
```

We can now define the notions of 1-sorted structures and strict 1-sorted structures as follows.

```
Definition struct_1sorted_p : set -> prop := fun X => True.
Definition strict_1sorted_p : set -> prop := fun X =>
   struct_1sorted_p X /\ strict_struct_p sels X.
```

We now close the section, removing the local definition of *sels* and local hypothesis *ocs_carrier* from the context.

```
End struct_1sorted_sec_b.
```

We next turn to a unified version of Examples 6.2 and 6.3 as pointed structures. We start by opening a new subsection and adding a selector variable *pt*. We also add a local hypothesis that *pt* is a different function from *carrier*. We then locally define sels to contain exactly these two selectors and add a local hypothesis that this forms an ocs.

```
Section struct_Ptd_sec.

Variable pt : set -> set.
Hypothesis carrier_not_pt: ~(carrier = pt).
Let sels : (set -> set) -> prop
```

[5]The local hypotheses play no role in the definitions made in this section, but are included to make clear what assumptions would need to be made or proven when making use of such structures.

```
  := (fun f:set -> set => f = carrier \/ f = pt).
Hypothesis ocs_Ptd : ocs sels.
```

We can now define pointed structures and strict pointed structures as follows.

```
Definition struct_Ptd_p : set -> prop := fun X => pt X :e carrier X.
Definition strict_Ptd_p : set -> prop := fun X =>
  struct_Ptd_p X /\ strict_struct_p sels X.
```

We then close the subsection.

```
End struct_Ptd_sec.
```

The reader may be concerned that we may have trouble representing Example 6.4 since we combined Examples 6.2 and 6.3. However, the fact that we can vary the collection $\mathcal{F}$ means we do have separate representations of Example 6.2 and Example 6.3 on demand.

To represent Example 6.4 we open a new subsection, declare two new selector function variables, *ZeroF* and *OneF*, and assume they are different from *carrier* and each other.

```
Section struct_ZeroOne_sec.

Variable ZeroF : set -> set.
Variable OneF : set -> set.
Hypothesis carrier_not_ZeroF: ~(carrier = ZeroF).
Hypothesis carrier_not_OneF: ~(carrier = OneF).
Hypothesis ZeroF_not_OneF: ~(ZeroF = OneF).
```

We now locally define a collection of the three selectors.

```
Let sels : (set -> set) -> prop
  := (fun f:set -> set => f = carrier \/ f = ZeroF \/ f = OneF).
```

We add a local hypothesis that the set of all three is an ocs.

```
Hypothesis ocs_ZeroOne : ocs sels.
```

We can now define ZeroOneStr structures and precisely those that are both ZeroStr structures and OneStr structures. Since $\mathsf{struct\_Ptd_p}$ was defined in a section with a variable *pt* that is now closed, we must explicitly give the argument to fill in this value. We can use either *ZeroF* or *OneF* and we make use of both in the definition below.

```
Definition struct_ZeroOne_p : set -> prop
  := fun X => struct_Ptd_p ZeroF X /\ struct_Ptd_p OneF X.
Definition strict_ZeroOne_p : set -> prop
  := fun X => struct_ZeroOne_p X /\ strict_struct_p sels X.
```

We finally close the subsection and section.

```
End struct_ZeroOne_sec.
End struct_1sorted_sec.
```

7. Conclusion

We have described a version of higher-order Tarski-Grothendieck set theory published as a Proofgold theory. The axioms are relatively faithful to the Mizar formulation.

Acknowledgment

This work has been supported by the European Research Council (ERC) Consolidator grant nr. 649043 *AI4REASON*.

References

[1] Bancerek, G., Bylinski, C., Grabowski, A., Kornilowicz, A., Matuszewski, R., Naumowicz, A., Pak, K.: The role of the Mizar Mathematical Library for interactive proof development in Mizar. Journal of Automated Reasoning **61**(1-4), 9–32 (2018)

[2] Bancerek, G., Byliński, C., Grabowski, A., Korniłowicz, A., Matuszewski, R., Naumowicz, A., Pąk, K., Urban, J.: Mizar: State-of-the-art and Beyond. In: Kerber, M., Carette, J., Kaliszyk, C., Rabe, F., Sorge, V. (eds.) Intelligent Computer Mathematics - International Conference, CICM 2015. LNCS, vol. 9150, pp. 261–279. Springer (2015). https://doi.org/10.1007/978-3-319-20615-8_17

[3] Brown, C.E.: The Egal manual (Sep 2014)

[4] Brown, C.E.: A theory supporting higher-order abstract syntax. Tech. rep., Czech Technical University in Prague (Aug 2020), http://grid01.ciirc.cvut.cz/~chad/hoas/pfghoas.pdf

[5] Brown, C.E., Kaliszyk, C., Pąk, K.: Higher-Order Tarski Grothendieck as a Foundation for Formal Proof. In: Harrison, J., O'Leary, J., Tolmach, A. (eds.) 10th International Conference on Interactive Theorem Proving (ITP 2019). Leibniz International Proceedings in Informatics (LIPIcs), vol. 141, pp. 9:1–9:16. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany (2019), http://drops.dagstuhl.de/opus/volltexte/2019/11064

[6] Brown, C.E., Pąk, K.: A tale of two set theories. In: Kaliszyk, C., Brady, E.C., Kohlhase, A., Coen, C.S. (eds.) Intelligent Computer Mathematics - 12th International Conference, CICM 2019, Prague, Czech Republic, July 8-12, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11617, pp. 44–60. Springer (2019)

[7] Committee, L.: Boolean properties of sets — definitions (April 2002), http://mizar.org/JFM/EMM/xboole_0.html

[8] Grothendieck, A., Verdier, J.L.: Théorie des topos et cohomologie étale des schémas - (SGA 4) - vol. 1, Lecture Notes in Mathematics, vol. 269. Springer-Verlag (1972)

[9] Kaliszyk, C., Pąk, K.: Semantics of Mizar as an Isabelle object logic. Journal of Automated Reasoning **63**(3), 557–595 (2019). https://doi.org/10.1007/s10817-018-9479-z

[10] N. D. Goodman and J. Myhill: Choice Implies Excluded Middle. Zeitschrift fur Mathematische Logik und Grundlagen der Mathematik **24**, 461 (1978)

[11] Prawitz, D.: Natural deduction: a proof-theoretical study. Dover (2006)

[12] R. Diaconescu: Axiom of choice and complementation. Proceedings of the American Mathematical Society **51**, 176–178 (1975)

[13] Russell, B.: The Principles of Mathematics. Cambridge University Press (1903)

[14] Tarski, A.: Über Unerreichbare Kardinalzahlen. Fundamenta Mathematicae **30**, 68–89 (1938)

[15] Trybulec, A.: Tarski Grothendieck Set Theory. Journal of Formalized Mathematics **Axiomatics** (2002), released 1989

[16] Trybulec, Z.: Properties of subsets. Formalized Mathematics **1**(**1**), 67–71 (1990)