Mizar's Soft Type System

Freek Wiedijk

Institute for Computing and Information Sciences
Radboud University Nijmegen
Toernooiveld 1, 6525 ED Nijmegen, The Netherlands

Abstract. In Mizar, unlike in most other proof assistants, the types are not part of the foundations of the system. Mizar is based on untyped set theory, which means that in Mizar expressions are typed but the values of those expressions are not.
In this paper we present the Mizar type system as a collection of type inference rules. We will interpret Mizar types as *soft types*, by translating Mizar's type judgments into sequents of untyped first order predicate logic. We will then prove that the Mizar type system is *correct* with respect to this translation in the sense that each derivable type judgment translates to a provable sequent.

1 Introduction

1.1 Problem

The activity of checking mathematical proofs for correctness using a computer is called 'formalization of mathematics'. Systems for doing this are called proof assistants [12]. There are three main classes of proof assistants: the ones based on Church's *higher order logic* (e.g., HOL, Isabelle/HOL, ProofPower, and maybe also systems like PVS), the ones based on Martin-Löf's *type theory* (e.g., Coq, NuPRL, Agda, Epigram), and the proof assistants based on Cantor's *set theory* (e.g., Mizar, Metamath, Isabelle/ZF, the B method). These three 'cultures' are quite different, and people from different cultures sometimes find it hard to get a clear view on what people from the other cultures are doing. The primary aim of this paper is to make the set theoretical system Mizar (and in particular its type system) more understandable to people from the type theoretical and higher order logic communities.

The Mizar proof assistant [7] has been in development in Białystok, Poland, from the seventies until today, by a team led by Andrzej Trybulec. The current version of the system is called PC Mizar and is at version 7. It was originally released in 1989, and is still being actively developed. The library of formalized mathematics that accompanies the Mizar system is called MML (for Mizar Mathematical Library). It is the largest library of formalized mathematics that is currently in existence. At the time of writing it consists of 1.9 million lines of Mizar text, and is the collaborative work of almost two hundred people (the so-called 'Mizar authors.') The large size of the MML library is partly a reflection of the fact that Mizar only gives one moderate automation. Also the Mizar

library contains quite a bit of rather obscure mathematics. Still the Mizar library is undoubtedly one of the most interesting libraries of formalized mathematics in the world. The MML contains a formalization of a graduate-level textbook on the theory of continuous lattices, which is being translated quite faithfully into the Mizar language. (This project, which is being led by Grzegorz Bancerek, is currently two thirds finished.)

The Mizar system is based on first order predicate logic with on top of that a slight extension of Zermelo-Fraenkel set theory (ZF). To be precise: the *system* only implements first order predicate logic (together with 'schematic' axioms and theorems[1] – to be able to deal with the 'replacement' axiom scheme of ZF – and with a Hilbert choice operator that is hard-wired into the way the system deals with underspecified functions), but the MML *library* is based on set theory (all of the 1.9 million of lines are fully checked for correctness against only a handful of set theoretical axioms.) It is irrelevant for the rest of the paper exactly which set theory the Mizar library is based on, but to be specific: Mizar's set theory is called Tarski-Grothendieck set theory. It is ZF set theory with one extra axiom added (where '$\sim$' means that the two sets have the same cardinality):

$$\begin{aligned}\forall N \exists M \big(& N \in M \wedge \\ & \forall X \forall Y (X \in M \wedge Y \subseteq X \Rightarrow Y \in M) \wedge \\ & \forall X (X \in M \Rightarrow \exists Z (Z \in M \wedge \forall Y (Y \subseteq X \Rightarrow Y \in Z))) \wedge \\ & \forall X (X \subseteq M \Rightarrow X \sim M \vee X \in M) \big)\end{aligned}$$

This axiom states that there are arbitrarily large universes, or, in other words, that there are arbitrarily large strongly inaccessible cardinals. From it one can derive the Axiom of Choice (even without that axiom being hard wired into the logic.) I do not know why Mizar has selected exactly *this* set theory for its foundation. Maybe one of the reasons was that the universes are useful when one wants to do category theory.

Set theory in the style of Cantor, like ZF set theory or von Neumann-Bernays-Gödel set theory (NBG), customarily is an *untyped* theory. Most often there is just one 'sort', the sort of sets. At most there will be two sorts, with the second sort being the sort of 'proper classes'. However, the Mizar system is a *typed* system: it implements a type system on top of its first order predicate logic. And not only does it have a type system, its type system is very powerful, and has many interesting features. In particular: it does support *dependent types*, which are *very* useful if one wants to formalize abstract mathematics.

This raises the question of what the type system of Mizar 'means': what the relation is between the typed version of set theory that one gets in Mizar and the untyped set theories like ZF that one finds in logic textbooks. This is the question that this paper will address.

1.2 Approach

There are at least two ways in which to give an interpretation to Mizar's types:

[1] Mizar's notation for set comprehension – in the Mizar community called the 'Fraenkel-operator' – is a alternative version of this.

1. Either one interprets the types of Mizar as *classes of sets*. In that case the set theory of Mizar will be taken to be a set theory like NBG set theory, where proper classes are first class objects.
   In this approach the interpretation of the Mizar type '`Ring`' will be the class of rings, while the (dependent) type '`LeftMod of`' will be interpreted by the 'class function' that maps each element of this class of rings to the class of the left modules over it. This interpretation of '`LeftMod of`' therefore maps *each* element of a proper class to a *different* proper class.
2. Alternatively one can interpret the types of Mizar as *predicates*[2] that the system will keep track of automatically. This approach sometimes is called *soft typing*[3], which is the terminology that we will adopt for it in this paper. In this approach each reference to the type '`Ring`' will be interpreted as *actually* being about a unary predicate '`is_Ring`', while the type '`LeftMod of`' will be interpreted as *actually* talking about a binary predicate '`is_LeftMod_of`'.

In this paper we will focus on the second approach, and will not pursue the first possibility. This second approach has the attractive property that it does not need set theory. It allows one to give a meaning to the types of typed first order predicate logic (and even with types that are as powerful as the ones that one find in Mizar), even in the case that the theory in that logic is not set theory.

Mizar uses types in various ways. First, they annotate all variables that occur in a Mizar text: both for bound variables in formulas and for variables in a proof[4] there is a type. For instance, one writes

```
for n being Nat holds ...
```

when one universally quantifies over a variable `n` that has the type `Nat` (the type of natural numbers), and one might write

```
let n be Nat;
```

if one is reasoning in a proof about a variable of type `Nat`.

But types also can be used to create a formula, that states that an expression has a certain type. For instance one can write that

```
i is Nat;
```

Here the keyword '`is`' is put between a term and a type. This specific example means that the value of the expression '`i`' (which for instance could be a variable

[2] More generally: a dependent type with n parameters will be interpreted as an $n+1$-ary predicate.

[3] This term originates in the programming languages community [2], where it is used with a somewhat different meaning. In that context it means that you statically type as much as possible, and insert run-time checks for what cannot be statically typed.

[4] Although in logic these are considered variables, in the Mizar community they are often looked at as constants that are local to the proof. In the Isabelle community these are called *fixed variables*.

of type '`Integer`') satisfies the defining property of the type '`Nat`', i.e., this states that `i` is non-negative.

This second way of using types does not have a direct counterpart in most of the proof assistants from the higher order logic and type theory traditions. There a type judgment is on a different level from the statements that one reasons about: in those systems type judgments are *outside* the logic, while statements are *in* the logic. But 'soft types' cross this boundary.

Apart from using types for reasoning, Mizar also uses types to disambiguate expressions that involve overloaded notation. Although this feature is very useful (for instance in Mizar a notation like $X + Y$ can mean many different things, depending on what the types of X and Y are) we will not go into this use of Mizar types in this paper.

To reiterate: the interpretation that we give to Mizar types will be to associate a 'hidden' predicate which each type, and to consider references to types to really be about these hidden predicates. We will consider the Mizar type system as being an engine that automatically – behind the scenes – generates statements involving those predicates. For instance, if we call the predicate that corresponds to the '`Nat`' type '`is_Nat`', then the three example lines of Mizar on the previous page will be interpreted as being an abbreviation of:

```
for n being set holds (is_Nat(n) implies ...)

let n be set; assume is_Nat(n);

is_Nat(i)
```

Now we have statements in single sorted predicate logic where all variables are in the sort '`set`'.

1.3 Related Work

There already is a paper about the Mizar type system by Grzegorz Bancerek [1]. However, the presentation of the type system in that papers is quite different from the one that is used here. Also, the interpretation of the types in an 'algebra of types' is closer to the first approach that we mentioned on page 3 than to the second approach that we are pursuing here. Andrzej Trybulec also has such an approach, using the notion of what he calls *dependent type algebras*.

The work of Ingo Dahn [3] and the work of Josef Urban [9, 8, 10] on translating Mizar into a form that is digestible by first order theorem provers, is based on the same interpretation of Mizar types as soft types that we present in Section 5. In the work of Josef Urban [10] also occurs the view of the Mizar type checker in 'logic programming' style, which is how we interpreted our typing rules in Section 4.

There are other theorem provers that use soft types as a way of pushing the type system of first order predicate logic beyond the customary 'many-sorted' variant of first order predicate logic, where one only has a finite number of types without any further structure. For instance it is used in the SPASS first order theorem prover, see [11].

Joe Hurd showed how to have a layer of soft typing on top of a higher order system [5]. PVS also can be considered to be such a system. However, in PVS the soft typing is not actually a separate *layer*.

1.4 Contribution

The contribution of this paper is threefold:

- It gives a *presentation* of the Mizar type system as a collection of type inference rules, which is the style that is standard in the type theoretical research community. This presentation of the Mizar type system is new.
- It gives an *interpretation* of Mizar's types by systematically translating type judgments to sequents of untyped first order predicate logic. This interpretation was already implicitly present in the work of Ingo Dahn and Josef Urban, but the way it is made explicit in this paper, and the focus that we have on interpreting it as the *semantics* of the Mizar type system, is new.
- It shows that this interpretation is *correct*, in the sense that for any type judgment that is generated by the Mizar type system, the corresponding translation is provable.

1.5 Outline

The structure of this paper is as follows. In Section 2 we present the notation that we will use for Mizar's type judgments. In Section 3 we give a non-trivial example of such a judgment, which we both present in our notation and in the usual Mizar syntax. In Section 4 we present the type inference rules of the Mizar type system. In Section 5 we prove that these rules are correct with respect to a translation into untyped logic. Finally in Section 6 we conclude, and mention some questions for further research.

2 Judgments

In Mizar types have various uses. In particular, they also play an important rôle in the logical language of the system, because type judgments are one of the kinds of atomic formula. However, from now on we will focus on the type system on its own, separately from the logic. That is, we will now just focus on the mechanism that ascribes types to terms.[5] In the Mizar system, this is

[5] This might seem to be a strong restriction. In particular the reader might wonder how the type system fits in with the underlying set theory. (In fact, the Mizar type system is completely independent of set theory. The Mizar system and its type theory can be used with *any* first order theory.)

If one translates a Mizar formalization into standard 'unsorted' first order predicate logic (as is done in [9]), then basically three kinds of statements are involved. First, there are the statements that actually occur in the Mizar text, which are 'translated' by relativizing all quantifiers to the predicates that correspond to

implemented in the part of the system that is called ANALYZER. (The logic is implemented in the part called CHECKER.[6])

We will now describe the syntax that we will use for Mizar type judgments. The notation that we will use is specific to this paper. It will make the typing rules of Section 4 much more compact than if we had used the Mizar input syntax. For a non-trivial example of a typing sequent, both in our syntax as well as in the Mizar input syntax, see Section 3.

In our judgments we will have four kinds of variables. Actually, three of the four kinds are not considered to be variables in the Mizar system, but because we bind these symbols in the context part of the judgments[7] we will consider them to be variables of the type judgment. These kinds of variables are:

x	term variable
f	'functor' = function symbol
α	'attributes' = type modifier
M	'mode' = type symbol

(The words 'functor', 'attributes[8]', 'mode' and (below) 'radix type' are Mizar-specific terminology.[9])

Apart from these variables, we will use one special symbol, the asterisk, for the root type of Mizar, of which all other types are subtypes:

---

the types. Second, there are the statement that give the definition of the various predicates, functors, modes and attributes. In the Mizar file these are implicitly given by the keywords '`means`' or '`equals`'. Third, there are the statements that originate in the type system: the statements occurring in the translations of the type judgments as described in Section 5. Apart from the statements of the second kind, *all* these statements have a proof in the Mizar file.

This means that although it might seem in Section 4 that one is allowed to arbitrarily build a Mizar context, if one wants to have that context in an actual Mizar text, then one will need to *prove* the translation in the style of Section 5 of that context.

[6] Note that neither ANALYZER nor CHECKER has a small 'kernel', as is common in LCF-style systems. To trust the Mizar system, one will need to trust the full code base.

[7] Actually, many of these variables will occur multiple times in the context, because of the presence of redefinitions and clusters. The first occurrence of the variable – corresponding to its original introduction in the Mizar text – is the binding one.

[8] Most systems do not have anything like the attributes of Mizar, which are type modifiers that behave a bit like intersection types. See Section 3 below for an example that shows how attributes are used. In that example the type '`int`' of the integers has attributes '`pos`' and '`neg`' that allow one to talk about the positive and the negative integers. For example the type '`pos int`' contains the positive integers. The elements of that type are the objects that *both* satisfy the defining statement of the '`int`' type and that of the '`pos`' attribute.

[9] Words like 'functor' and 'radix' are used with quite different meanings outside of the context of Mizar, so our use of these words might be confusing. Maybe 'function symbol' or 'operator' for 'functor' and 'base type' for 'radix type' would be more appropriate. However, we decided not to depart from standard Mizar terminology.

$*$ the root type

In Mizar input syntax this type is called '`set`' because it is the type of all sets. It also used to be called '`Any`'. (In our interpretation of types as unary predicates it corresponds to the predicate that is constantly true.[10])

Mizar also has *predicate symbols*, which are not treated in this paper. The reason for leaving them out is that although the arguments to the predicates are typed, the predicates will not play any part in generating the types.

Next, here is the grammar that gives the various notions that occur in our type judgments:

$$
\begin{aligned}
t &::= x \mid f(\vec{t}) \\
R &::= * \mid M(\vec{t}) \\
a &::= \alpha \mid \bar{\alpha} \\
T &::= \vec{a}\, R \\
D &::= x : T \\
J &::= \cdot \mid t : T \mid T \leq T \mid \exists T \mid \alpha / T \\
\Delta &::= \vec{D} \\
\Gamma &::= \overrightarrow{[\Delta](J)}
\end{aligned}
$$

In this grammar t are the *terms*, R are the *radix types*, a are the *adjectives* (these are either an attribute α or the negation of an attribute $\bar{\alpha}$), T are the *types*, D are the *declarations*, J are the *judgment elements*, and Δ and Γ are the two parts of the context of a type judgment. The full type judgments that we will derive with our typing rules will have the shape:

$$\Gamma; \Delta \vdash J$$

In this judgement Γ corresponds to the type information of a series of Mizar definitions and registrations. The 'local context' Δ corresponds to the types of the variables that have been introduced *inside* a Mizar definition or proof. We call Γ the global context and Δ the local context of the judgment.

In the grammar from this section we use vector notation. For instance, an expression like $[\vec{x} : \vec{T}](f(\vec{x}) : T')$ should be read as $[x_1 : T_1, \ldots, x_n : T_n]$ $(f(x_1, \ldots, x_n) : T')$. Furthermore all expressions should only be considered 'modulo α-equivalence', so $[x : T](f(x) : T')$ and $[y : T](f(y) : T')$ are really considered to be the same. Finally the 'clusters of adjectives' $\vec{a}$ in front of a type are considered only as sets. In other words, the expression $\alpha_1\bar{\alpha}_2$ is considered to be equivalent to $\bar{\alpha}_2\alpha_1$, $\alpha_1\alpha_1\bar{\alpha}_2$, $\alpha_1\bar{\alpha}_2\bar{\alpha}_2$, and so on.

The instances of the notion of judgment elements J have the following interpretations:

[10] In the typing rules of Section 4 the root type plays an essential rôle. One can only introduce new radix types to the context if one already *has* a type (which will be its supertype). So without the root type the rules of Section 4 would not work, as there would not be a starting point for introducing types.

$t : T$	t has type T
$T_1 \leq T_2$	T_1 is a subtype of T_2
$\exists T$	T is a non-empty type
α/T	α is an attribute of the type T

In Mizar input language $t : T$ is written as 't `be` T' or 't `being` T', and $T_1 \leq T_2$ is written as 'T_1 `->` T_2'.

The centered dot '$\cdot$' is used if one just wants to state that a context is well-formed. (To indicate that a type T is well-formed one either uses $T \leq T$ or $\exists T$, depending on whether one already knows that the type is non-empty or not.)

3 Example

We now present an example that demonstrates the various features of the Mizar type system. The following type judgment[11] is derivable using the rules of Section 4:

$$
\begin{array}{c}
\mathsf{int} \leq *,\ \exists\,\mathsf{int},\\
\mathsf{pos}/\mathsf{int},\\
\mathsf{neg}/\mathsf{int},\\
\mathsf{pos\ int} \leq \overline{\mathsf{neg}}\ \mathsf{int},\\
\exists\,\mathsf{pos\ int},\\
\exists\,\overline{\mathsf{pos}}\ \overline{\mathsf{neg}}\ \mathsf{int},\\
\mathsf{z} : \mathsf{int},\\
\mathsf{z} : \overline{\mathsf{neg}}\ \overline{\mathsf{pos}}\ \mathsf{int},\\
[n : \mathsf{int}](\mathsf{S}(n) : \mathsf{int}),\\
[n : \mathsf{int}](\mathsf{P}(n) : \mathsf{int}),\\
[n : \overline{\mathsf{neg}}\ \mathsf{int}](\mathsf{S}(n) : \mathsf{pos\ int}),\\
[n : \mathsf{pos\ int}](\mathsf{P}(n) : \overline{\mathsf{neg}}\ \mathsf{int}),\\
[n : \overline{\mathsf{neg}}\ \mathsf{int}](\mathsf{list}(n) \leq *),\ [n : \overline{\mathsf{neg}}\ \mathsf{int}](\exists\,\mathsf{list}(n)),\\
\mathsf{nil} : \mathsf{list}(\mathsf{z}),\\
[n : \overline{\mathsf{neg}}\ \mathsf{int},\ x : *,\ l : \mathsf{list}(n)](\mathsf{cons}(n, x, l) : \mathsf{list}(\mathsf{S}(n))),\\
[n : \mathsf{pos\ int},\ l : \mathsf{list}(n)](\mathsf{car}(n, l) : *),\\
[n : \mathsf{pos\ int},\ l : \mathsf{list}(n)](\mathsf{cdr}(n, l) : \mathsf{list}(\mathsf{P}(n)))\\
;\\
x : *\\
\vdash\\
\mathsf{cdr}(\mathsf{S}(\mathsf{z}), \mathsf{cons}(\mathsf{z}, x, \mathsf{nil})) : \mathsf{list}(\mathsf{P}(\mathsf{S}(\mathsf{z})))
\end{array}
$$

This example involves types int and list for integers and lists, the latter being a dependent type with the length of the list as the argument. It has attributes pos and neg on the type of integers for positive and negative integers. Furthermore it has functions S and P on the integers for successor and predecessor and functions

[11] We put the first two judgment elements '$\mathsf{int} \leq *$' and '$\exists\,\mathsf{int}$' on a single line, as in the Mizar text it corresponds to a single definition.

nil, cons, car and cdr for the lists.[12] The local context of the judgment is just the variable declaration $x : *$, and the statement of the judgment is that the term '$\mathsf{cdr}(\mathsf{S}(\mathsf{z}), \mathsf{cons}(\mathsf{z}, x, \mathsf{nil}))$' has among its collection of types the type '$\mathsf{list}(\mathsf{P}(\mathsf{S}(\mathsf{z})))$'.

The judgment that we just presented has the shape

$$\Gamma; \Delta \vdash J$$

of which the part Γ corresponds to a series of definitions and registrations. Here is what this Γ looks like in Mizar syntax:

```
definition
 mode int -> set means ...; existence ...
end;

definition let n be int;
 attr n is pos means ...
 attr n is neg means ...
end;

registration
 cluster pos -> non neg int; coherence ...
end;

registration
 cluster pos int; existence ...
 cluster non pos non neg int; existence ...
end;

definition
 func z -> int means ...
end;

registration
 cluster z -> non neg non pos; coherence ...
end;

definition let n be int;
 func S n -> int means ...
 func P n -> int means ...
end;

registration let n be non neg int;
 cluster S n -> pos; coherence ...
```

[12] These are not the names that are used in the MML for these notions. There int is called `Integer`, list is called `FinSequence` (although the version that depends on the length of the list like in the example is called `Tuple`, and that type also depends on the set from which the elements are taken). The attributes pos and neg are called `positive` and `negative`, and nil is called `{}`. The functions S and P do not exist in the MML, $\mathsf{S}(n)$ is just written as $n + 1$ and $\mathsf{P}(n)$ is just $n - 1$. The list functions do not exist either: $\mathsf{cons}(x, l)$ is written `<*`x`*>^`l, $\mathsf{car}(l)$ is written l`.1`, and $\mathsf{cdr}(l)$ is written l`/^1`.

The name `neg` in the MML is not an attribute symbol but a functor symbol, for intuitionistic negation.

```
end;

definition let n be pos int;
 redefine func P n -> non neg int; coherence ...
end;

definition let n be non neg int;
 mode list of n -> set means ...; existence ...
end;

definition
 func nil -> list of z means ...
end;

definition let n be non neg int; let x be set; let l be list of n;
 func cons(x,l) -> list of S n means ...
end;

definition let n be pos int; let l be list of n;
 func car l -> set means ...
end;

definition let n be pos int; let l be list of n;
 func cdr l -> list of P n means ...
end;
```

For a version of these definitions that has been completed with statements and proofs in the place of the dots see <http://www.cs.ru.nl/~freek/mizar/example.miz> and <http://www.cs.ru.nl/~freek/mizar/example.voc>.[13]

Note that most of the items in this list of definitions and registrations have proof obligations. This means that they do not, like in the judgments that we show in this paper, behave like assumptions, but that they really are statements that are proved from the existing theory. For example, the part of the context

$$[n : \mathsf{pos\ int}](\mathsf{P}(n) : \overline{\mathsf{neg}}\ \mathsf{int}),$$

that is the 'redefinition' of the predecessor function to give its value the more specific type '$\overline{\mathsf{neg}}\ \mathsf{int}$' when it is known that the argument has the type '$\mathsf{pos\ int}$' is in the Mizar formalization written as

```
definition
 let n be pos int;
 redefine func P n -> non neg int;
 coherence
 proof
  ...
  hence P n is non neg int by ...
 end;
end;
```

[13] Note that from the point of view of the MML, this is a *very* silly example as it just defines a lot of stuff that is present already, only using different names.

where the '`P n`' in the `hence` line refers to the earlier definition of the P function. This means that the dots will have to be a proof that amounts to showing that from $n > 0$ it follows that $n - 1 \geq 0$. Under the translation from Section 5 this part of the typing context turns into

$$\forall n\big(\mathsf{pos}(n) \wedge \mathsf{int}(n) \Rightarrow \neg\mathsf{neg}(\mathsf{P}(n)) \wedge \mathsf{int}(\mathsf{P}(n))\big)$$

and in a real Mizar formalization that statement will have to be proved.

Similarly the other definitions and registrations have appropriate proof obligations (the only judgment elements in the context of the example that do not have proof obligations are the attribute definitions.)

4 Rules

We will now present the Mizar type system as a collection of typing rules. These rules present a slightly simplified version of the Mizar type system:

- There are no structure types.
  Structure types are the 'record types' of Mizar. (For an interesting discussion of how structure types can be understood in terms of the underlying set theoretical foundations of the Mizar system, see [6].)
- In the real Mizar system, symbols can be overloaded. Here we suppose that all symbols have been sufficiently disambiguated. That is, the typing rules that we are presenting should only be instantiated according to the Barendregt convention (variables should never be 'hidden' by later variables.)
- In the real Mizar type system, a redefinition takes priority over the definitions that come before it in the context. However, in this version of the type system the order of the definitions is not taken into account when generating type judgments.
- When reasoning, the set of types of a Mizar expression will be closed under available equalities. For instance, if one of the statements that justifies the inference is $n = n'$ and an expression has type $\mathsf{list}(n)$, then it also will be assigned type $\mathsf{list}(n')$. This kind of equality reasoning is not present in our version of the Mizar type system.

Note that a Mizar term generally does not have just one type (for instance, every well typed expression will always *also* have the type $*$), and can even get a quite large collection of types.

One can consider these rules to be something like a 'logic programming' description of the Mizar type checker:

$$\frac{}{;\vdash \cdot}$$

$$\frac{\Gamma;\Delta \vdash \vec{t} : \vec{T}[\vec{x} := \vec{t}]}{\Gamma;\Delta \vdash J[\vec{x} := \vec{t}]}\ [\vec{x} : \vec{T}](J) \in \Gamma;\Delta \qquad \frac{\Gamma;\Delta \vdash \exists T}{\Gamma;\Delta, x : T \vdash \cdot}$$

$$\frac{\Gamma;\Delta\vdash\cdot}{\Gamma;\Delta\vdash *\leq *}\qquad\frac{\Gamma;\Delta\vdash\cdot}{\Gamma;\Delta\vdash\exists *}$$

$$\frac{\Gamma;\Delta\vdash T\leq T'}{\Gamma;\Delta\vdash T\leq T}\qquad\frac{\Gamma;\Delta\vdash T\leq T'\quad\Gamma;\Delta\vdash T'\leq T''}{\Gamma;\Delta\vdash T\leq T''}$$

$$\frac{\Gamma;\Delta\vdash t:T\quad\Gamma;\Delta\vdash T\leq T'}{\Gamma;\Delta\vdash t:T'}\qquad\frac{\Gamma;\Delta\vdash T\leq T'\quad\Gamma;\Delta\vdash\exists T}{\Gamma;\Delta\vdash\exists T'}$$

$$\frac{\Gamma;\Delta\vdash\alpha/T}{\Gamma;\Delta\vdash\alpha T\leq T}\qquad\frac{\Gamma;\Delta\vdash\alpha/T}{\Gamma;\Delta\vdash\bar{\alpha}T\leq T}$$

$$\frac{\Gamma;\Delta\vdash T\leq T'\quad\Gamma;\Delta\vdash a\,T'\leq T'}{\Gamma;\Delta\vdash a\,T\leq T}\qquad\frac{\Gamma;\Delta\vdash T\leq T'\quad\Gamma;\Delta\vdash a\,T'\leq T'}{\Gamma;\Delta\vdash a\,T\leq a\,T'}$$

$$\frac{\Gamma;\Delta\vdash T\leq aT'\quad\Gamma;\Delta\vdash T\leq a'T'}{\Gamma;\Delta\vdash T\leq aa'T'}$$

functor definition:
$$\frac{\Gamma;\,\vec{x}:\vec{T}\vdash\exists T'}{\Gamma,\,[\vec{x}:\vec{T}](f(\vec{x}):T');\vdash\cdot}\;f\notin\Gamma$$

mode definition:
$$\frac{\Gamma;\,\vec{x}:\vec{T}\vdash\exists T'}{\Gamma,\,[\vec{x}:\vec{T}](M(\vec{x})\leq T'),\,[\vec{x}:\vec{T}](\exists\,M(\vec{x}));\vdash\cdot}\;M\notin\Gamma$$

attribute definition:
$$\frac{\Gamma;\Delta\vdash\exists T'}{\Gamma,\,[\Delta](\alpha/T');\vdash\cdot}\;\alpha\notin\Gamma$$

existential cluster:
$$\frac{\Gamma;\Delta\vdash\vec{a}\,T'\leq T'\quad\Gamma;\Delta\vdash\exists T'}{\Gamma,\,[\Delta](\exists\,\vec{a}\,T');\vdash\cdot}$$

conditional cluster:
$$\frac{\Gamma;\Delta\vdash\vec{a}\,T'\leq T'\quad\Gamma;\Delta\vdash\vec{a}'\,T'\leq T'}{\Gamma,\,[\Delta](\vec{a}\,T'\leq\vec{a}'\,T');\vdash\cdot}$$

functorial cluster:
$$\frac{\Gamma;\Delta\vdash t:T'\quad\Gamma;\Delta\vdash\vec{a}\,T'\leq T'}{\Gamma,\,[\Delta](t:\vec{a}\,T');\vdash\cdot}$$

functor redefinition:
$$\frac{\begin{array}{c}\Gamma;\,\vec{x}:\vec{T}\vdash\exists T''\quad\Gamma;\,\vec{x}:\vec{T}\vdash T''\leq T'''\\ \Gamma;\,\vec{x}':\vec{T}',\,\vec{x}:\vec{T}\vdash f(\vec{x}):T'''\end{array}}{\Gamma,\,[\vec{x}':\vec{T}',\,\vec{x}:\vec{T}](f(\vec{x}):T'');\vdash\cdot}$$

mode redefinition:
$$\frac{\begin{array}{c}\Gamma;\,\vec{x}:\vec{T}\vdash\exists T''\quad\Gamma;\,\vec{x}:\vec{T}\vdash T''\leq T'''\\ \Gamma;\,\vec{x}':\vec{T}',\,\vec{x}:\vec{T}\vdash M(\vec{x})\leq T'''\end{array}}{\Gamma,\,[\vec{x}':\vec{T}',\,\vec{x}:\vec{T}](M(\vec{x}):T'');\vdash\cdot}$$

The side-conditions '$\ldots \notin \Gamma$' of the three definition rules mean that the symbol does not occur anywhere in the context Γ. The 'extra arguments' $\vec{x}'$ in the redefinition rules and generally are the empty vector.

In Mizar the types of actual terms always have to be non-empty.[14] This explains all the assumptions of the form $\exists\, T$ that occur in these rules. However, note that clusters sometimes involve types that have not been shown to be non-empty.

5 Correctness

We will now translate the typing judgments of our Mizar type system into sequents of single sorted first order predicate logic.[15] We will do this in two phases:

- First we introduce an *annotated* version of the Mizar type system, in which all the attributes have explicit arguments. This is exactly the same system that we already presented, but each attribute α now gets a list of arguments. These arguments are the arguments that determine the type on which the attribute was defined.
  The grammar of the judgments stays exactly like it was, except that the rule for an adjective becomes:

$$a ::= \alpha(\vec{t}) \mid \bar{\alpha}(\vec{t})$$

  The derivation rules also stay all the same, apart from the rules that explicitly involve an attribute symbol:

$$\frac{\Gamma; \vec{x} : \vec{T} \vdash \exists\, T'}{\Gamma,\ [\vec{x} : \vec{T}](\alpha(\vec{x})/T');\ \vdash \cdot}\ \alpha \notin \Gamma$$

$$\frac{\Gamma; \Delta \vdash \alpha(\vec{t})/T}{\Gamma; \Delta \vdash \alpha(\vec{t})\, T \leq T} \qquad \frac{\Gamma; \Delta \vdash \alpha(\vec{t})/T}{\Gamma; \Delta \vdash \bar{\alpha}(\vec{t})\, T \leq T}$$

  We now have the following theorem, that we present here without proof:

[14] There is no good mathematical reason for this restriction. It *does* lead to higher quality formalizations, as the formalizer will need to think about whether the types are empty or not. Also, it of course simplifies the implementation of the inference checker. On the other hand it sometimes leads to unnatural mode definitions (like the infamous definition of '`Element of`').

[15] In fact the Mizar typing rules can be motivated from this translation. The Mizar type system is designed to combine two opposite goals: to have the system automatically infer as many type judgments as possible for which the translation is provable; but on the other hand to have a type system in which all typing judgments can be efficiently inferred.

Theorem 5.1. *For every derivable type judgment of the non-annotated version of the Mizar type system, there exists a corresponding type judgment of the annotated version of the Mizar type system which only differs in that after the attributes arguments have been added.)*

Note that the annotated version of the judgment is not always unique. This is the reason that the Mizar implementation internally keeps track of these 'hidden arguments' of the attributes. (There has been discussion in the Mizar community about whether these arguments maybe also should be allowed to be explicit.)

– Now that we have annotated versions of the Mizar type judgments, translating them into first order logic is easy. We define a translation $|\cdot|$ that maps our system into first order logic:

$$\begin{aligned}
|*|(t) &:= \top \\
|M(\vec{t'})|(t) &:= M(t, \vec{t'}) \\
|\alpha(\vec{t'})|(t) &:= \alpha(t, \vec{t'}) \\
|\bar{\alpha}(\vec{t'})|(t) &:= \neg\alpha(t, \vec{t'}) \\
|a_1 \ldots a_n\, R|(t) &:= |a_1|(t) \wedge \ldots \wedge |a_n|(t) \wedge |R|(t) \\
\\
|t : T| &:= |T|(t) \\
|T \leq T'| &:= \forall x\, (|T|(x) \Rightarrow |T'|(x)) \\
|\exists\, T| &:= \exists x\, (|T|(x)) \\
|\alpha/T| &:= \top \\
\\
|[x_1 : T_1, \ldots, x_n : T_n](J)| &:= \forall x_1 \ldots \forall x_n (|T_1|(x_1) \wedge \ldots \wedge |T_n|(x_n) \Rightarrow |J|) \\
|\overrightarrow{[\Delta](J)}; \overrightarrow{J'} \vdash J''| &:= \overrightarrow{|[\Delta](J)|}, \overrightarrow{|J'|} \vdash |J''|
\end{aligned}$$

In the first order logic, we take the symbols for types and attributes to be predicate symbols, where the arity as a predicate symbol is one more than the original arity. The idea is that if T is a type, then $|T|(t)$ corresponds to the Mizar statement 't `is` T'. This explains the order that we chose for the arguments of the symbol taken as a predicate.
(The translation of α/T as just 'true' might be unexpected. One might expect the translation to be:

$$|\alpha/T| := \forall x\, (|\alpha|(x) \Rightarrow |T|(x))$$

However, we decided not to do this, as it would be strange to require $|\alpha|(t)$ to imply $|T|(t)$, but not to require the same property for $|\bar{\alpha}|(t)$ ($\bar{\alpha}$ also being an adjective of T). Our translation is interpreting adjectives in the spirit of intersection types.[16])

[16] It might be surprising that an attribute can be given whatever radix type one likes. This corresponds to the fact that in Mizar an attribute definition does not need any proof (it does not have a 'correctness condition'): the defining statement of an attribute can be any formula, without any restriction. Note also that there is no such thing as an 'attribute redefinition'.

The following theorem now also is easy:

Theorem 5.2. *If $\Gamma; \Delta \vdash J$ is a judgment of the annotated version of the Mizar type system, then $|\Gamma; \Delta \vdash J|$ is a provable sequent of first order predicate logic.*

Proof. Induction on the derivation of the judgment: each rule of the type system corresponds to a small derivation in first order logic. □

6 Conclusion

6.1 Discussion

One of the nicest things about interpreting the Mizar type system as a system of *soft* types, is that it shows that the Mizar type system, with all its power, can easily be added 'on top' of any existing proof assistant.[17] It is mainly a matter of defining selected predicates to take the rôle of the types and then to:

- Implement automatic inference of statements about these predicates along the lines of the rules in Section 4.
- Add a layer of parsing and pretty-printing to the system to have syntax for the predicates as types. In this layer the *implicit arguments* that are common in dependently typed systems can be implemented.

This way even the systems from the HOL family of proof assistants [4] can have dependent types!

In fact, even in systems like Coq that have a rather powerful type system already, one often sees the tendency to use 'soft' types on top of the 'hard' types that are already in the system. For instance, in the C-CoRN library from Nijmegen, there is a formalization of integration theory, where instead of using a type for continuous partial functions, there are lines in the development that look like:

```
Variables a b : IR.
Hypothesis Hab : a [<=] b.
Variable F : PartIR.
Hypothesis contF : Continuous_I Hab F.
```

and then the operation called '`Integral`' takes the predicate '`contF`' as one of its arguments. Clearly this is a rough version of soft typing.

Similarly, a development of Galois theory by Georges Gonthier and Sean McLaughlin (developed in the Microsoft/INRIA institute in Paris) is using a style of having 'soft types on top of Coq'.

[17] The work of Joe Hurd [5] already is a version of this (although it of course does not implement the full Mizar type system.)

6.2 Future work

There are various interesting questions that might be pursued as a continuation of the research presented in this paper:

- A type system really should be *decidable*, but we have not established this property for the type system that we presented here. The implementation of the type system in the Mizar type checker clearly seems to have this property, but then the rules that we presented in this paper do not *exactly* match the type system as it is implemented in the actual system.
  It is clear that a naive implementation of our version of the Mizar type system will not work. The redefinition:

  ```
  redefine mode Element of n -> Element of n + 1;
  ```

  (which is allowed in the actual Mizar system!) will with our version of the typing rules give rise to infinite collections of types.[18] This example also shows that a direct Prolog-style implementation of the rules from Section 4 will not always terminate. However, it is might be the case that a less naive algorithm than just generating all possible types for an expression still can implement a type checker for the typing rules from Section 4.
- We have proved that the type system that we present in this paper is *correct*, but one also could consider the question of whether it is *complete*. The theorem to be proved for this would be that if a sequent is a translation of a typing judgment in our system of which the context is already a correct context, and if that sequent is provable in first order predicate logic, that then in fact the typing judgment is derivable in the type system.
  We have not yet tried to prove this property, but thus far we do not know of a counter-example either.
- It might be interesting to make the rules that we presented here more realistic, by for example adding structure types or equalities between terms. Maybe even more interesting would be to precisely analyze the differences between our version of the type system, and the corresponding fragment of the type system the way it is implemented in the actual Mizar system. The interesting question would be whether the differences might easily be removed (either by adapting the typing rules, or by changing the implementation), or whether the differences are essential for giving the implementation of the type checker a reasonable performance.

Acknowledgments. It will be clear that this paper was inspired by the papers of Fairouz Kamareddine. When I wrote this paper I had Herman Geuvers in mind as my target audience: I hope that this paper managed to communicate the Mizar type system to him. Furthermore many thanks to Josef Urban, Makarius and the anonymous referees of the TPHOLs conference for their helpful comments on a draft version of this paper.

[18] We got this example from Josef Urban.

References

1. Grzegorz Bancerek. On the structure of Mizar types. *Electronic Notes in Theoretical Computer Science*, 85, 2003.
2. R. Cartwright and M. Fagan. Soft typing. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 278–292, 1991.
3. Ingo Dahn. Interpretation of a Mizar-Like Logic in First-Order Logic. In *FTP (LNCS Selection)*, pages 137–151, 1998.
4. M.J.C. Gordon and T.F. Melham, editors. *Introduction to HOL*. Cambridge University Press, Cambridge, 1993.
5. Joe Hurd. Predicate subtyping with predicate sets. In Richard J. Boulton and Paul B. Jackson, editors, *14th International Conference on Theorem Proving in Higher Order Logics: TPHOLs 2001*, volume 2152 of *LNCS*, pages 265–280. Springer-Verlag, 2001.
6. Gilbert Lee and Piotr Rudnicki. Alternative Aggregates in Mizar, 2007. To be published in Mathematical Knowledge Management 2007.
7. Michał Muzalewski. *An Outline of PC Mizar*. Fondation Philippe le Hodey, Brussels, 1993. ⟨`http://www.cs.ru.nl/~freek/mizar/mizarmanual.ps.gz`⟩.
8. Josef Urban. MPTP 0.1: System Description. *ENTCS*, 86(1), 2003.
9. Josef Urban. Translating Mizar for First Order Theorem Provers. In *MKM*, volume 2594 of *Lecture Notes in Computer Science*, pages 203–215. Springer, 2003.
10. Josef Urban. MoMM – Fast Interreduction and Retrieval in Large Libraries of Formalized Mathematics. *International Journal on Artificial Intelligence Tools*, 15(1):109–130, 2006.
11. C. Weidenbach. SPASS: Combining superposition, sorts and splitting, 1999.
12. Freek Wiedijk, editor. *The Seventeen Provers of the World*, volume 3600 of *LNCS*. Springer, 2006. With a foreword by Dana S. Scott.