Basic first-order model theory in Mizar

MARCO CAMINATI
Dipartimento di Matematica "Guido Castelnuovo",
Sapienza - Università di Roma
caminati@mat.uniroma1.it
http://www.mat.uniroma1.it/~caminati

The author has submitted to Mizar Mathematical Library a series of five articles introducing a framework for the formalization of classical first-order model theory. In them, Gödel's completeness and Lowenheim-Skolem theorems have also been formalized for the countable case, to offer a first application of it and to showcase its utility. This is an overview and commentary on some key aspects of this setup. It features exposition and discussion of a new encoding of basic definitions and theoretical gears needed for the task, remarks about the design strategies and approaches adopted in their implementation, and more general reflections about proof checking induced by the work done.

1. MOTIVATION: MODEL THEORY IN MIZAR

Inside Mizar Mathematical Library (MML), there are at least three strains hosting articles of content suitable for the treatment of first-order logic:

(1) A series of articles supplying a language apt to describe set theory according to Zermelo-Fraenkel axioms, started with [Ban90].
(2) A series of articles supplying a general language for first-order logic, started with [RT90].
(3) A series of articles supplying terminology and results about universal algebras, started with [KMK92].

Most of the classical results of first order logic have, during the years, found their way in strain (2): building on those articles a fairly equipped gear of formalizations has been created.
There are treatments about the most elementary syntactical properties (those of variables and free variables in a formula (`QC_LANG3`), of subformulas (`QC_LANG2`, `QC_LANG4`), of substitution (`CQC_LANG,SUBSTUT1,SUBSTUT2`), of similarity between formulas (`CQC_SIM1`)), which in turn allow for less and less elementary results, regarding: propositional calculus (`PROCAL_1`, `LUKASI_1`), interpretation and satisfiability (`VALUAT_1`), Gentzen-style sequent calculus (`CALCUL_1`, `CALCUL_2`), up to a basic version of Gödel's completeness theorem (`HENMODEL,GOEDELCP`).

Unfortunately, the coding of the first order language adopted from the very beginning in [RT90] is somewhat rigid: roughly sketching the situation, strings of first-order language are represented as tuples of couples of natural numbers, with special symbols (quantifiers, connectives, truth symbol) represented by couples in which the first component is a reserved (small) natural.

This inherently prevents treating uncountable languages, which, alas, would be quite the point for developing even the most fundamental results of model theory, starting with Löwenheim-Skolem and compactness theorems.

Also, the completeness theorem currently present in MML has some limitations that look hardly removable in the established framework. For example, it is restricted to equality-lacking languages, while it would be of interest to talk about languages with equality: Mizar first-order language itself is furnished with equality, and the option of possibly applying results worked out to Mizar itself is desirable.

This paper is an account of how a fairly developed codebase for model theory in Mizar has been laid down, given the considerations above. They imposed reformulating things from scratch with a hopefully more flexible approach.

This codebase culminates, as a testbed for itself, with formalizations of the fundamental Gödel's completeness and Lowenheim-Skolem theorems, restricted to the case of a generic countable language, and has been submitted to MML Library Committee for peer-reviewing; meanwhile it is accessible at the author's homepage. More precisely, among the many flavours of Lowenheim-Skolem theorem, the one checked is the 'downward' flavour, like VI.1.1 from [EFT84]. Its Mizar statement sounds like:

```
for
  U2 being non empty set, S being Language,
  X being countable Subset of AllFormulasOf S,
  I2 being Element of U2-InterpretersOf S st X is I2-satisfied
ex U1 being countable non empty set,
   I1 being Element of U1-InterpretersOf S st
X is I1-satisfied
```

while the completeness theorem runs thus:

```
for C being countable Language, phi wff string of C, X being set st
  X c= AllFormulasOf C & phi is X-implied
holds
  phi is X-provable
```

Note that this last restriction to countable languages is a mere matter of convenience: the whole work was set up to treat an arbitrary language up to satisfiability theorem (see section 1.1), which is the cornerstone result; on the other hand, reducing to the least-cardinality case was desirable in order to have the job done more quickly (under the urge of demonstrating its usability), without having to handle complications related to the axiom of choice and the likes. Besides, the role of Konig's lemma in countable case might deserve some investigation in which Mizar could help (see section 7).

Those two theorems are here regarded as a significant goal because of their fundamental role in model theory. In particular, the family of Lowenheim-Skolem theorems have a fruitful interplay with the cardinality of the language, which the ability to deal with, as said, was a starting, motivating point for the present work. Moreover, this latter kind of results seem to be underrepresented in the global repository of mechanically checked mathematics: the only work sharing the aims of the present which the author is aware of is [Har98]; both the checker and the proof

techniques used there are entirely different than what we are going to deploy here, however.

1.1 A birds-eye's view of the proof

This paper focuses on design choices and encodings adopted in formalization, rather than on the proofs themselves. However, since indeed the goal all along the way was ultimately proving a theorem and this proof presents some departing points from literature standards (e.g. [EFT84]), let us sketch how the proof of completeness theorem works (Lowenheim-Skolem is then a conceptually easy step). A more detailed account of the proof techniques deployed in this formalization can be found in [Cam09].

A Henkin-style proof works its way towards completeness by setting a first intermediate result known as satisfiability theorem. The path to this first result is constructive, passing through the introduction of the so called Henkin's model. In turn, here the construction of Henkin's model is staged (see section 3) into a construction here called free interpretation (`-freeInterpreter`, see section 4.3) and into quotienting it by a given equivalence relation, as in the following checklist displaying what needs to be proven (and where (1) corresponds to `FOMODEL4:Lm11a` and `FOMODEL4:Lm53a`, (2) to `FOMODEL4:Lm12`, and (3) to `FOMODEL4:15`):

(1) if D is a set of rules including at least rules (2), (3), (4), (5) (see section 2.2), and X is a D-closed theory, that is containing all formulas provable from X itself using the rules of D, then the relation $\sim$ between the terms

$$t_1 \sim t_2 \Leftrightarrow X \vdash_D \equiv t_1 t_2$$

—is an equivalence relation on the set of all terms
—is compatible with (also 'is respected by') `(S,X)-freeInterpreter`,
that is, called f_s the free interpreter for the single symbol s of absolute arity $n \in \mathbb{N}$, the request

$$\forall\, (t_1, \ldots, t_n), (t'_1, \ldots, t'_n) \text{ being } n\text{-tuples of terms holds}$$
$$\begin{cases} t_1 \sim t'_1, \ldots, t_n \sim t'_n \Rightarrow f_s(t_1, \ldots, t_n) \sim f_s(t'_1, \ldots, t'_n) & \text{if } s \text{ is operational} \\ t_1 \sim t'_1, \ldots, t_n \sim t'_n \Rightarrow f_s(t_1, \ldots, t_n) = f_s(t'_1, \ldots, t'_n) & \text{if } s \text{ is relational} \end{cases}$$

is satisfied. This permits to define unambiguously the quotient of every f_s, as a function on the n-tuples of $\sim$-equivalence classes of terms in the natural way:

$$\frac{f_s}{\sim}\left([t_1], \ldots, [t_n]\right) := f_s\left(t_1, \ldots, t_n\right) \tag{1}$$

(2) if the above happens for every compounder (i.e. relational or functional, see section 2.1) symbol s, the resulting quotient interpretation $H_{D,X}$ enjoys the property

$$H_{D,X} \vDash \phi \iff \phi \in X \qquad \forall \phi \text{ atomic formula of } S$$

(3) if, moreover, D includes rules (7), (6) (left rule), (8) (left rule) and X is maximal and includes witnesses, then one can generalize the previous point to all formulas, which is the wanted satisfiability theorem.

Afterwards, completeness, that is, provability of any formula entailed by a given consistent theory Y, can be achieved by expanding Y to a superset X to which satisfiability theorem is applicable: then said provability is easily granted. Obviously the delicate part in passing from satisfiability to completeness is to make sure that the superset X contains the right formulas, so as to supply witnesses and be maximal as dictated by point (3). This is exactly the passage in which the present work restricts to countable languages, and where the couple of rules of section 2.2 yet to be used are employed.

1.2 A word about reformulation

Looking back at all the preliminary work that had already been done and included in MML, and that could not be used led, after initial despair, to sketch somewhat of a strategy to plan the work.

Effort has been devoted to reducing to a minimum the work, cutting out the formalization of every result not strictly needed, at the cost of rephrasing the theory a bit, too (see section 2). Care has been taken in putting forth definitions in the neatest possible way, with the idea that good definitions should provide the soil to introduce new results with relatively little effort, just when they are needed; and save the effort if one finds a way to avoid their introduction.

Perhaps the most distinctive example of this approach is how the point of free occurrences of a variable in a formula has been faced (rules (6) in section 2.2): rather than re-inventing all the battery of results to deal with the concept of free occurrences, attention has been focused on changing the rules of sequent calculus in a way which permits largely to delay the issue to the days in which its treatment will be strictly needed.

Instead of the one-way dynamics (from human to machine) one could expect when starting digging into formalization, this turned into a sort of feedback leading the human to rethink and rephrase along the way what he is formalizing. Every time this happened, the final outcome was always tidier and more neat of the initial idea; sometimes, the process leading to the change was itself instructive and thought-provoking.

This process was applied all along the formalization work, resulting in the end in a treatment differing in a bunch of crucial spots from the one taken as a starting point [EFT84].

2. A MIZAR-FRIENDLY REFORMULATION

For the reasons just exposed, standard definitions of first-order language, syntax, and derivation rules have been dedicatedly tweaked. Here we informally explain how.

2.1 Encoding of language

The first design choice is to use polish notation: for example $x > y + z$ becomes $> x{+}yz$. This is a common choice in software and in formalization for its simplicity; both [RT90] and [Ban90] adopt it as well.

Secondly, there is no native distinction between free and bound variables. What's more, there is not even a distinction between variables and constants symbols. There are only symbols of arity zero, which are called literals, and symbols of non zero arity, called compounders. To be more precise, the distinction is left to the semantics, in the sense that a constant becomes a variable exactly when it is caught by quantification inside a formula.

Third, there is no quantification symbol. This does not mean that we will not be able to quantify, of course: *existential* quantification is indicated by heading a formula with a literal symbol, and this creates no confusion.
Of course, universal quantification will be rendered via existential and negation constructs, as is customarily done; we shall soon an applied instance of this in the example about group axioms below.

Fourth, arity will yield *signed* integers, with the convention that negative arity symbols will be relational (predicate) compounders and positive arity symbols will be operational compounders. The absolute value of the arity will indicate the actual arity of the compounder. In [RT90] there are no operational symbols, which can always semantically be emulated by relational (predicate) symbols, but this makes the definition of well-formed formulas (wff) lengthy.

Lastly, there is only one logical connector, that is NOR ($\downarrow$). This suffices since NOR is universal (as is NAND). So, in this reformulation, we will be able to take advantage of a language having only two special symbols: equality and NOR (particularly in treating wff formulas and evaluation (see 4.2.1) this will be a life-saving simplification). However, they will not be hard-coded as fixed sets, rather will be brought out via how the `Language` type will be defined. This way one can choose any infinite set as the symbol set. More precisely, to us a language is a quadruple consisting of an infinite set X, two distinct yet arbitrary elements $\equiv$ and $\downarrow$ of it, and a function from $X\backslash\{\downarrow\}$ into $\mathbb{Z}$, called adicity (the keyword 'arity' is already taken inside MML, which will not prevent us from using it here outside of Mizar code), with the added constraint that the adicity of $\equiv$ must be -2. This constraint is part of the type itself, because we want to do all the proofs for first-order languages with equality, as said in section 1.

To give the simplest illustration, let us rephrase in this language the group axioms, using $\mathbb{N}$ as a symbol set, 1 as $\equiv$, 0 as $\downarrow$ and arity $f\colon \mathbb{Z}^+ \to \mathbb{Z}$ given by

$$f(n) := \begin{cases} -2 & \text{if n=1} \\ 2 & \text{if n=2} \\ 0 & \text{otherwise} \end{cases}$$

Direct translation might result bewildering, so first let us list axioms in standard human-friendly form and in an intermediate jargon made by combining polish notation with shortcut symbols $\exists, \forall, =, +$ for quantifiers and compounders:

$$\begin{aligned} \forall a, b, c\ a(bc) = (ab)c \qquad & \forall 3\forall 4\forall 5\ = +3 + 45 + +345 \\ \forall a\ ea = a \qquad & \forall 4\ = +344 \\ \forall a \exists b\ ba = e \qquad & \forall 4\exists 5\ = +543 \end{aligned}$$

Finally, we pass to the final coding by rendering $\forall x\phi$ as $\neg\exists x\phi$, $\neg\phi$ as $\downarrow\phi\phi$, $\exists x\phi$ as $x\phi$ and substituting $=$, $+$ respectively with 1, 2, obtaining some nasty strings:

$$0304051232452342551232452342540512324523425512324523425 3$$
$$040512324523425512324523425405123245234255123245234 25$$
$$0412344412344$$
$$045124534512453,$$

where the first, exceedingly long axiom has been split across two lines.

This shows how the absence of auxiliary boolean connectors and quantifiers makes even trivial formulas go wildly verbose. Note that none of the three axioms uses more than seven literals, so we have been able to unambiguously use decimal representation for $\mathbb{N}$. Also compare the role of the symbol '3' in expressing first and second axioms: in the first it is quantified and thus used as a variable, in the second it acts as a constant (the unity of the group) since it is not quantified. Not having distinguished between constants and variables permits reusing a literal symbol in both ways, as long as the corresponding constant does not appear in the formula in which the symbol is used as a variable. Given our goals, we do not care much about readability of the language: all that matters is that any first-order theory is expressible in the language, and that a set of sequent derivation rules which are both *correct* (that is, yielding correct sequents from correct sequents) and *complete* (that is, powerful enough to prove any consequence of a first-order theory) is given. As long as these constraints are respected, we seek for the design which maximize simplicity and neatness of formalization.

2.2 Sequent calculus

In this spirit we now face the choice of sequent derivation rules. The ones usually causing hindrance in formalization are those involving quantifiers, because they introduce notions like term substitution and free occurrence of variables, requiring a bulk of preparatory work (see `QC_LANG3`, `QC_LANG2`, `QC_LANG4`, `CQC_LANG`, `SUBSTUT1`, `SUBSTUT2`, `CQC_SIM1`).

To escape the catch, we reformulate rules so as to eliminate the need to talk of free occurrences, while for term substitution we resort to constructions already introduced for other reasons, thus enforcing a sort of code reusing. Let us explain this concretely: in what follows, φ, ϕ, ψ stand for formulas, Γ, Γ' for any finite set of formulas, t, t' for terms, s for a generic symbol of first-order language, and v for a literal symbol of it. Possibly, numerical subscripts can be used to indicate different represented entities.

As said, the rules we fear are the ones dealing with quantification; in our case, they appear at row (6) below. The first implies only the checking of generic (not *free*) occurrence of a single symbol among a finite set of formulas, which is easily done, and the substitution of a literal with a literal, not dealing with terms at all, which makes its formalization easy too.

The second rule introduces the operator $\psi \mapsto \psi\frac{t}{v}$ substituting a literal with a term. The idea is that, at least when applied to another term, this operator can be expressed by plugging together other Mizar constructs we needed in previous work,

$$\frac{}{\Gamma \vdash \varphi} \text{ where } \{\varphi\} \subseteq \Gamma \qquad \frac{\Gamma \vdash \varphi}{\Gamma' \vdash \varphi} \text{ where } \Gamma \subseteq \Gamma' \tag{2}$$

$$\frac{}{\equiv t_1 t_2 \vdash \equiv t_2 t_1} \qquad \frac{}{\vdash \equiv tt} \qquad \frac{\Gamma \vdash \equiv tt_1}{\Gamma \equiv t_1 t_2 \vdash \equiv tt_2} \tag{3}$$

$$\frac{}{\equiv t_1 t_1' \ \dots \ \equiv t_n t_n' \vdash \equiv st_1 \dots t_n st_1' \dots t_n'} \text{ where } n = \operatorname{arity}(s) \in \mathbb{N} \tag{4}$$

$$\frac{}{st_1 \dots t_n \equiv t_1 t_1' \ \dots \ \equiv t_n t_n' \vdash st_1' \dots t_n'} \text{ where } n = -\operatorname{arity}(s) \in \mathbb{Z}^+ \tag{5}$$

$$\frac{\Gamma \quad \frac{v_2}{v_1}\phi \vdash \psi}{\Gamma \quad v_1\phi \vdash \psi} \text{ where } v_2 \text{ does not occur in } \Gamma, \phi, \psi \qquad \frac{}{\psi \frac{t}{v} \vdash v\psi} \tag{6}$$

$$\frac{\Gamma \vdash \downarrow \phi\psi}{\Gamma \vdash \downarrow \psi\phi} \qquad \frac{\Gamma \vdash \downarrow \phi\phi \quad \Gamma \vdash \downarrow \psi\psi}{\Gamma \vdash \downarrow \phi\psi} \tag{7}$$

$$\frac{\Gamma \quad \phi \vdash \phi_1 \quad \Gamma \quad \phi \vdash \downarrow \phi_1\phi_2}{\Gamma \vdash \downarrow \phi\phi} \qquad \frac{\Gamma \vdash \downarrow\downarrow \phi}{\Gamma \vdash \phi} \tag{8}$$

Fig. 1. Gentzen-style derivation rules adopted

by reducing $t_1 \mapsto t_1 \frac{t_2}{v}$ to (see definition of `AtomicSubst` in `FOMODEL3`)

$$\left(\frac{t_2}{v}\, \mathcal{F}_{\{\}}\right)(t_1)$$

where

(1) Operator $\frac{u}{v}\mathcal{I}$ transforms an interpretation $\mathcal{I}$ into a different interpretation in which *only* the assignment of *one* literal v changes into the element u of the universe. This is an easy formalization (see section 5) and is anyway needed to evaluate the truth of a quantified formula.

(2) Given an interpretation $\mathcal{I}$ and a term t of a language, $\mathcal{I}(t)$ yields the value of t in that interpretation; this is of course a fundamental construction for even just talking about a first-order language.

(3) $\mathcal{F}_\Phi$ denotes the free interpretation of a language according to a set of formulas Φ of that language. Here the adjective "free" is to be meant as in "free object", not as in "free occurrence" (of a literal). As already said, this is a fundamental player in a Henkin-style proof. See section 4.3.

The leap to term substitution inside a formula is elementary and not discussed here.

Having given some motivation for them, the full set of rules can be seen in Figure 1. (4) and (5) are a bit clumsy to write down, but their proof-theoretical weakness turned out to be quite helpful in easing formalization.

Anyway, writing derivation rules in the manner above is like drawing diagrams, in that their goal is to communicate to another human how the rule works; what matters is the formalizability, and maybe the computability (which is likely to be good if the former is), so we should not worry about the appearance of (4) and (5).

3. ORGANIZATION OF THE CODEBASE

With a total of about $700k$ bytes and $19k$ lines of Mizar code, this turned out to be a fairly complex project, so care has been constantly taken to orderly arrange the various results according to their scope into five separate Mizar articles, each depending on the previous ones and hosting affine themes:

—`FOMODEL0.MIZ` is the receptacle of all results of broader scope stemmed during the various formalizations, with results and registrations about objects already in MML and quite few dependencies.

—`FOMODEL1.MIZ` introduces the type `Language`, the classification of symbols according to their arity and of terms according to their depth, and the functor to extract subterms from a term or an atomic formula. The bulk of syntax is done here and in next article.

—`FOMODEL2.MIZ` deals with syntax of non atomic formulas and all the semantics by giving the following constructions: the definition of an interpreter I relative to a non empty set U (universe), the constructions saying how to evaluate a term in U, how to evaluate an atomic formula in $\{0, 1\}$, what can be regarded as a generic wff formula, how to evaluate it in $\{0, 1\}$ according to I, and how to evaluate its depth. Also, the functor to obtain another interpreter in the same universe U from I by changing the evaluation of a single literal symbol of the language, and the definitions of satisfaction and of entailment are given.

—`FOMODEL3.MIZ` supplies a toolkit of constructions to work with languages and interpretations, and results relating them: the free interpretation of a language, having as a universe the set of terms of the language itself, is defined; the quotient of an interpretation with respect to an equivalence relation is built, and shown to remain an interpretation when the relation respects it. Both the concepts of quotient and of respecting relation are defined in broadest terms, with respect to objects as general as possible. This is arguably the most 'technical' article in the tier.

—`FOMODEL4.MIZ` introduces the proof-theoretical notions and binds all together. As a first more general task, it defines what a sequent and a rule are, and what means for a rule to be correct. Then, using these definitions, it builds the particular set of derivation rules we chose in section 2.2. Among many other results, satisfiability theorem is proven. Finally, restricting to countable languages, completeness and downward Lowenheim-Skolem are proved.

Having sketched the themes dealt with in each article, now the idea is that each formalized result should be placed in the lowest article in which the entities to enunciate it are available, so to give a precise criterion for the arraying of Mizar code among the five articles.

About one sixth of the code dwells in `FOMODEL0.MIZ`, thus applying to already-defined Mizar entities; also, the results located there tend to be shorter and more numerous than the lemmas showing up in subsequent articles. This is a clue of a general separation and modularization design policy pursued across the whole work, aiming at

—stating results in terms of the most general possible Mizar entities.

—breaking specific statements into smaller lemmas, especially if the latter as a result get applicable to a broader class of objects and/or if the smaller lemmas can be put together in more than a way to get significant theorems. The same applies to definitions.

As an example, take the construction of the already discussed Henkin model. In [EFT84], it is introduced just before the proof of the satisfiability theorem, and so, given the rather instrumental nature of its role, its definition is quite condensed. Here, on the other hand, it has been split into the pair of definitions of free interpretation and of quotient interpretation, with a twofold benefit. First, the former object gets reused to construct one of the deduction rules (section 2.2). On the other hand, the latter applies not only to the former, but to any interpretation. What's more, the quotient functor is defined more generally as quotient of a relation by a pair of equivalence relations. Relations are more general than equivalence relations, which are in turn more general than functions, which finally are more general than interpretations, if one call an entity more general than another when the latter is defined in terms of the former.

Accordingly, the various results needed for the Henkin interpretation break into smaller and more general statements, sometimes of interest themselves, or occurring more than once in building further theorems, or maybe just hopefully useful to a possible coder in the future: having stated them in less restrictive terms increases the probability that this will be the case.

This process of separation and modularization may provide a further benefit: in breaking a statement into smaller steps, a fine-grained analysis of which assumptions are needed for each step is encouraged. This blatantly occurs in chopping down satisfiability theorem: in section 1.1 each step specifies which derivation rules are needed for it to hold. Indeed, keeping track of which result traces back to which rules did provide the main guidance in enouncing ruleset given in section 2.2. In the sequel, other, more specific occurrences of this attitude will be given: see especially sections 4.4 and 5.

Here, another aspect of this policy is discussed: closely related to the just discussed tendency to predicate about as less specialized entities as possible is the choice of encoding formulas in simple strings of symbols. Another method could have been the use of parse trees, which directly model the semantics and thus inherently dispense one from specifying the syntax rules for well-formedness and give an elementary way to attach a meaning to each formula. This is surely a strong plus for them. We maintain that using 'plain text', as done here, has advantages, too. A first advantage is readability: as strings require little assumed knowledge to be understood and have simple notations, the results worked out here are themselves very readable. This is of importance especially for a project like Mizar which, besides verifying, also aims at building a library of mathematical knowledge accessible to humans. Secondly, in the same vein of what has just been discussed, all the results worked out here are likely to produce sublemmas of interest to more Mizar coders than if we assume we chose parse trees: indeed, there is a series of Mizar articles supplying the machinery of parse trees in the context of formal languages ([]), and in this assumption, many of the general results in `FOMODEL0` would have been in a form available only to the users of that machinery. This is a two-way phenomenon,

of course: the author, using plain sequences instead of parse trees, has been able to take advantage of the massive amount of pre-existing results about the mode `FinSequence`. As an example of a 'by-product' of the present formalization which could be of more general interest, and which has been brought out because of the choice of using strings instead of parse trees, we pick a result regarding monoid and prefixes (see (9) in section 3.1); it is one of the numerous results got by treating subterms.

3.1 Dealing with subterms

As exposed in 2.2, some standard elementary definitions regarding the language have been diverted or reformulated into fewer entities. We missed the same goal for subterms: a way to avoid an explicit definition could not be devised; since their role is fundamental in evaluating both terms and formulas, we have to build a functor SubTerms yielding the subterms of a term. It is used crucially in the definition of `TermEval` and `TruthEval` functors, see section 4.2.1. Its coding will not be explicitly shown here for space reasons.

Here, we want to discuss how its construction slightly departs from standard treatments. The task at hand is plain dull: one usually does it recursively starting from literals and iterating through operational symbols, and there is not much room from alternative approaches. However, since the language is presently constructed in terms of strings and concatenation, we tried to do the job at the more general level of monoids and associative operations. We discuss briefly the idea, without displaying Mizar code.

Take a monoid $(M, \circ)$. One can easily extend the operation $\circ$ to any finite number of arguments iteratively, for example setting

$$\circ\,(a,b,c) := (a \circ b) \circ c, \qquad \circ\,(a,b,c,d) := (\circ\,(a,b,c)) \circ d,$$

and so on. To do this in Mizar we introduced the functor `MultPlace`, which actually takes any binary operation (associativity is not needed yet). Consider any $X \subseteq M$, and call it *unambiguous* (similarly to [Lot02], 1.2.1) if the restriction of $\circ$ to $X \times M$ is injective:

$$\circ\,(x_1,m_1) = \circ\,(x_2,m_2) \Rightarrow x_1 = x_2, m_1 = m_2 \qquad x_1,x_2 \in X, m_1,m_2 \in M$$

Now associativity comes into play for the fundamental result:

$$\circ \text{ associative and } X \text{ unambiguous } \Rightarrow \circ|_{X^n} \text{ is injective} \qquad \forall n \in \mathbb{N}, \qquad (9)$$

that is, unambiguity is sort-of preserved for n-tuples. Now, let us indicate with T^m the terms of depth not greater than $m \in \mathbb{N}$, with T the set of the terms altogether, $T = \bigcup_{m\in\mathbb{N}} T^m$, and with O the set of operation symbols. It is easy to show that $T^0 \cup M$ is unambiguous, now identifying $\circ$ with string concatentation, which is obviously associative (indeed, any one-letter strings subset of a language is trivially unambiguous with respect to concatenation). Starting from that, and using (9), it is easily shown by induction that any T^m is unambiguous, too; and finally:

THEOREM 3.1. *T^S is unambiguous.*

PROOF. Suppose $t, t' \in T$ and $y, y' \in M$ are such that $ty = t'y'$. Call m the greater among the depths of t and t'. Since $t, t' \in T^m$ and T^m is unambiguous, it must be $t = t'$ and $y = y'$. □

This permits defining subterms of a term t as the n-tuple of terms $t_1, \ldots, t_n$ such that:

$$t = \circ(o, t_1, \ldots, t_n),$$

where o is the first operation symbol, of arity n, of the string t. Since we know that $t_1, \ldots, t_n$ all belong to T, which is unambiguous, we can again apply (9) to decree their uniqueness, which is the point. We have discussed the general idea, the exact formulation is contained inside Mizar articles.

4. ENCODING IN MIZAR

In reporting here Mizar formalizations, some minor typographic changes to the original code have been made to accommodate and make it more readable; thus the snippets reported here should not be expected to compile correctly. For the real code, please refer to Mizar articles.
For an overview of Mizar, refer to appendix A. For a concise reminder of the Mizar notations used below, refer to table I.

4.1 The Language type

Here the ground mode `Language` we will be talking about all the time is defined, to circumvent limitations cited in section 1. There is good support in MML for finite sequences (articles `FINSEQ_1` through `FINSEQ_8`), so it is natural to identify the strings of the language we are defining with the finite sequences over its carrier. The same was done originally in [RT90]. The difference is that there it has been imposed to use exclusively sequences of Kuratowski pairs of natural numbers. Moreover, the encoding of special logical symbols is "hardwired" into that scheme. Then a layer of functors and modes definitions is added to be able to refer to these pairs with more suggestive names instead of using directly the encoding. However, there's no apparent need to impose preemptively how a first-order language should be encoded into sets, rather it seems more sensible to work only at the level of Mizar types, leaving freedom to choose what actual symbol set to use to the instantiator of the type. So let us start by introducing a preparatory type named `Language-like`:

```
definition
  struct (ZeroOneStr) Language-like
  (#carrier->set, ZeroF, OneF->Element of the carrier,
  adicity->Function of the carrier\{the OneF}, INT#);
end;
```

In this definition there appears yet another provision of Mizar to cope with types. `struct` is a "structured type", similar in spirit to the ones found in many programming languages (called something like aggregates, records, structures, as appropriate). It is a concise way to group a finite number of types into one entity which becomes a new type. Each entry, or selector, of the new type is denoted by an arbitrary type name. In our case, we took a pre-defined (see `STRUCT_0`) structure type, called `ZeroOneStr`, inherited all of its fields and added one more. So we end up with a quadruple consisting of an alphabet (the carrier), two distinguished symbols of it, and a arity (adicity) function. For brevity, a couple of devices are introduced here: first, the OneF will serve as our logical connective (Nor), and it

will turn out convenient not to have the arity defined on it; secondly, we agree that a negative arity will denote a relation symbol, a positive arity an operation symbol, and a zero arity a literal; these two points had been already introduced in section 2. With this in mind, the following definitions are obvious shorthands:

```
definition
  let S be Language-like;
  func AllSymbolsOf S equals the carrier of S;
  func LettersOf S equals (the adicity of S) " {0};
  func OpSymbolsOf S equals (the adicity of S) " (NAT \ {0});
  func RelSymbolsOf S equals (the adicity of S) " (INT \ NAT);
  func TermSymbolsOf S equals (the adicity of S) " NAT;
  func LowerCompoundersOf S equals (the adicity of S) " (INT \ {0});
  func TheEqSymbOf S equals the ZeroF of S;
  func TheNorSymbOf S equals the OneF of S;
  func OwnSymbolsOf S equals
  (the carrier of S)\{the ZeroF of S,the OneF of S};
end;
definition
  let S be Language-like;
  mode Element of S is Element of (AllSymbolsOf S);
  func AtomicFormulaSymbolsOf S equals
    AllSymbolsOf S\{TheNorSymbOf S};
  func AtomicTermsOf S equals 1-tuples_on (LettersOf S);
end;
```

This almost suffices to encode any first-order language. We only add a couple of further features we wish to endow our new type with:

```
definition
  let S be Language-like;
  attr S is eligible means LettersOf S is infinite &
  (the adicity of S).(TheEqSymbOf S)=-2;
end;
```

These two requests impose to have access to an infinite number of letters (we do not know the length of the terms and formulas we will need to write down), and that the arity of the equality symbol is -2, as already discussed in section 2. This automatically equates equality symbol to any other predicate symbol. However, this is true only at this stage of syntax. The equality symbol acquires of course special meaning in evaluation, as discussed in section 4.3. Finally, `Language` type is:

```
definition
  mode Language is eligible (non degenerated Language-like);
end;
```

`degenerated` is an attribute inherited from the type `ZeroOneStr`, and means that the `ZeroF` and the `OneF` coincide. So we are requesting that the equality symbol and the logical connective symbol are distinguishable. For what will be more than mere convenience, we also translate definitions in 4.1 cluster-wise:

```
definition
  let S be Language-like;
  let s be Element of S;
  attr s is literal means s in LettersOf S;
  attr s is low-compounding means  s in LowerCompoundersOf S;
  attr s is operational means  s in OpSymbolsOf S;
  attr s is relational means s in RelSymbolsOf S;
  attr s is termal means  s in TermSymbolsOf S;
  attr s is own means  s in OwnSymbolsOf S;
  attr s is ofAtomicFormula means s in AtomicFormulaSymbolsOf S;
end;
```

4.2 Syntax and semantics

The main objects introduced in this section are the three functors `-termsOfMaxDepth`, `-formulasOfMaxDepth`, `-TruthEval` and the type `Interpreter`. They have the fundamental roles of describing the sets of terms and formulas of a given (or smaller) depth, of defining what is an interpretation, and of evaluating a term or a formula in a given interpretation. For the sake of convenience, let us introduce a dedicated type for the generic S-string:

```
definition
  let S be Language;
  mode string of S is Element of ((AllSymbolsOf S)*\{{}});
end;
```

The present construction will be split in stages: first atomic terms (already introduced in 4.1), then terms inductively, and finally atomic formulas. Let us start with an auxiliary function performing the basic construction for polish notation, that is, appending an n-tuple of strings to a leading symbol according to its arity:

```
definition
  let S be Language,s be ofAtomicFormula Element of S,Strings be set;
  func ar(s) -> Element of INT equals (the adicity of S).s;
  func Compound(s,Strings) -> Subset of (AllSymbolsOf S)*\{{}}
  equals
    {<*s*> ^ ((S-multiCat).StringTuple) where
    StringTuple is Element of (AllSymbolsOf S)**:
    rng StringTuple c= Strings & StringTuple is (abs(ar(s)))-long};
end;
```

Here, `S-multiCat` is a dedicated function which concatenates tuples of strings. Roughly speaking, it is the finite iteration of the functor `^`. Now recursive construction of terms is straightforward:

```
definition
  let S be Language;
  func S-termsOfMaxDepth ->
  Function of NAT,bool((AllSymbolsOf S)*\{{}})
  means dom it=NAT & it.0 = (AtomicTermsOf S) & for n being Nat holds
    it.(n+1) = (union {Compound(s,it.n)
```

```
    where s is ofAtomicFormula Element of S:s is operational}
    ) \/ it.n;
  func AllTermsOf S equals union rng (S-termsOfMaxDepth);
end;
```

Again, let us rephrase above definitions in terms of attributes:

```
definition
  let m be Nat, S be Language, w be string of S;
  attr w is m-termal means  w in S-termsOfMaxDepth.m;
  let w be string of S;
  attr w is termal means w in AllTermsOf S;
  attr w is atomic means
    ex s being relational Element of S,
    V being abs(ar(s))-long Element of (AllTermsOf S)* st
      w=<*s*>^(S-multiCat.V);
end;
```

4.2.1 *Saving work: completing syntax and doing semantics, concurrently.*
Definitions in 4.2 are quite standard, and have been reported just for reference, being used in ones to come. Now, instead of proceeding with the syntax of non-atomic formulas, we digress to start concurrently putting forth some building blocks of semantics. We will then be able to define both syntax and semantics of non-atomic formulas in one shot, taking advantage of the fact that, in contrast to the building of terms, the compounders to derive higher-level formulas from lower-level ones are fixed and well-known. The fact of having reduced them to just two types (that is, one logical connective and one existential quantifier) will ease the job. This strategy saves a good deal of work for our purpose. First, we start with defining what is an interpretation of a `Language S` in a non empty set `U` (standing for universe). The definition is similar to the one given in [EFT84], only since we don't make distinction between 0-arity compounders (constants) and variables symbols, the distinction made there between interpretation, structure and assignment vanishes too. Also, we separate the universe from the interpretation (the corresponding type is called `Interpreter`; in informal talking we will use both words), more precisely, we make the latter a type dependent on the former. Here, too, we proceed gradually:

```
definition
  let S be Language, U be non empty set,
  s be ofAtomicFormula Element of S;
  mode Interpreter of s, U ->
  Function of (abs(ar(s)))-tuples_on U, U\/BOOLEAN means
    it is Function of (abs(ar(s)))-tuples_on U, BOOLEAN
  if s is relational otherwise
    it is Function of (abs(ar(s)))-tuples_on U, U;
end;
```

It is worth noting that in case of a literal (0-arity) symbol `s`, the interpreter of s,U reduces to a function from $\{\{\}\}$ into an element of U. So, the assignment of a literal, instead of being directly a constant of u of U, is rendered as a function `{{}} --> u`.

This is convenient for reducing the cases in subsequent proofs and definitions from three (positive, negative and zero arity) to two (negative and non negative arity). Now the definition of an interpreter of the whole alphabet is straightforward:

```
definition
  let S be Language, U be non empty set;
  mode Interpreter of S, U -> Function means
  for s being own Element of S holds it.s is Interpreter of s, U;
end;
definition
  let S be Language, U be non empty set, f be Function;
  attr f is (S,U)-interpreter-like means
  f is Interpreter of S,U & f is Function-yielding;
:: Function-yielding not fundamental; added for technical convenience
end;
definition
  let S be Language, U be non empty set;
  func U-InterpretersOf S equals {f where f is
  Element of Funcs(OwnSymbolsOf S, PFuncs(U*,U\/BOOLEAN)):
  f is (S,U)-interpreter-like};
end;
```

Before going on we quickly introduce two constructs: the first is the standard Mizar functor (`FUNCT_4:def 1`) `+*` which 'pastes' two function `f` and `g` into a function `f +* g` defined on the union of their domains, with `g` (the right term) prevailing in case of conflicts.

The second is the functor `ReassignIn` which implements the operator changing the assignment of a single literal in a given interpretation, already needed in (1) of section 2.2 and examined thoroughly in section 5.

Now, building a functor `I-AtomicEval phi` yielding the truth value of the *atomic* formula `phi` in the interpretation `I` is standard practice, and the corresponding code is omitted here. As anticipated, we rather want to indulge on the interpretation of non atomic formulas. Usually, one has to do first a recursive definition of the set of wffs, then another recursive definition to evaluate a wff in a given interpretation. The idea here is to do both in one single recursive definition. This technically can be done by having, as an object of the recursive definition, a partial function, here called `F` provisionally for brevity, such that, for any natural `mm`, `F.mm`

—has as a domain exactly the cartesian product of `U-InterpretersOf S` with the set of wff of depth not exceeding `mm`.

—on that domain maps a pair (interpretation, string) into the right truth value.

So we are working on a higher level, where also the interpreter `I` is a variable which gets evaluated together with a wff to return a truth value; only `L` and `U` are fixed parameters. For this reason, we first need a tedious but necessary step to transform `I-AtomicEval phi` from a functor into a *function* of `I` and `phi`, named `S-TruthEval U` (its name is regretfully not too descriptive):

```
definition
```

```
  let S,U;
  func S-TruthEval(U) ->
  Function of [: U-InterpretersOf S, AtomicFormulasOf S :],BOOLEAN
  means :DefTruth6: for I being Element of U-InterpretersOf S,
  phi being Element of AtomicFormulasOf S holds
    it.(I,phi)=I-AtomicEval(phi);
end;
```

For the same reason, in Mizar code the name of the functor `F` contains only `S` and `U`, and is `(S,U)-TruthEval`; so we can get the expected behaviour for it via the fundamental definition:

```
definition
  let S be Language, U be non empty set;
  func (S,U)-TruthEval -> Function of NAT,
  PFuncs([:U-InterpretersOf S, (AllSymbolsOf S)*\{{}}:], BOOLEAN)
  means it.0=S-TruthEval(U) & for mm being Element of NAT holds
  it.(mm+1)=G(it.mm) +* it.mm;
end;
```

At each step the partial function `(S,U)-TruthEval.mm`, which applied to the generic pair `[:I, phi:]` yields a defined, and correct, truth value if and only if `phi` is of depth not exceeding `mm`, is extended by the operator `G`, which of course must yield a partial function of domain extended to the wffs of depth `mm+1`. So the task is now the construction of `G`. We divide the problem in two simpler parts, taking care respectively of the existential symbol and of the NOR simbol separately, so that `G(it.mm)` in the actual Mizar definition is written as

```
ExIterator(it.mm) +* NorIterator(it.mm)
```

Let us illustrate only the construction of `ExIterator g` alone: the idea behind the other half is the same. Here `g` is a generic, appropriate `PartFunc`. We said that `ExIterator` has to take care simultaneously that the `PartFunc` it returns has both the right domain and the right output on it, based on `g`. This does not mean that we cannot further divide the problem into simpler parts: the definition of `ExIterator g` will actually specify only the correct domain, delegating the evaluation to yet another functor `-ExFunctor`:

```
definition
  let S be Language, U be non empty set;
  let g be Element of
  PFuncs([:U-InterpretersOf S, (AllSymbolsOf S)*\{{}}:], BOOLEAN);
  func ExIterator(g) -> PartFunc of
  [:U-InterpretersOf S, (AllSymbolsOf S)*\{{}}:],BOOLEAN means
    (for x being Element of U-InterpretersOf S,
    y being Element of (AllSymbolsOf S)*\{{}} holds
    ([x,y] in dom it iff (
    ex v being literal Element of S, w being string of S st
    [x,w] in dom g & y=<*v*>^w
     ))) &
```

```
  (for x being Element of U-InterpretersOf S,
  y being Element of (AllSymbolsOf S)*\{{}} st [x,y] in dom it holds
  it.(x,y)=g-ExFunctor(x,y));
end;
```

We have indented the part of definition which actually does something (i.e. the specification of the domain, as we were just saying); it does that something quite trivially, too. Also trivial is the action of the functor `-ExFunctor(x,y)` to which we delegated the semantical part:

```
definition
  let S be Language, U be non empty set, f be PartFunc of
  [:U-InterpretersOf S, (AllSymbolsOf S)*\{{}}:], BOOLEAN;
  let I be Element of U-InterpretersOf S;
  let phi be Element of(AllSymbolsOf S)*\{{}};
  func f-ExFunctor(I,phi) -> Element of BOOLEAN equals
  TRUE if ex u being Element of U, v being literal Element of S st
    (phi.1=v & f.((v,u) ReassignIn I, phi/^1)=TRUE)
  otherwise FALSE;
end;
```

Just notice that this functor is expected to be accurate only when yielding `TRUE`, since otherwise it could yield `FALSE` when actually it is supposed to be undefined. This is not a problem anymore, since the previous definition already took care of that matter.

Now the significant part of the work is done: all the syntactical and semantical knowledge is thus stored in `(S,U)-TruthEval`, we just may want to rearrange it in a more accessible way, a task with which we end this section. First, we can go back to the lower level and get a function of just the string we want to evaluate:

```
definition
  let S be Language, U be non empty set, m be Nat;
  let I be Element of U-InterpretersOf S;
  func (I,m)-TruthEval ->
  Element of PFuncs((AllSymbolsOf S)*\{{}},BOOLEAN)
  equals (curry ((S,U)-TruthEval.m)).I;
end;
```

Information about both syntax and semantics is now carried by (I,m)-TruthEval in respectively its domain and its return value, so:

```
definition
  let S be Language, m be Nat, w be string of S;
  func S-formulasOfMaxDepth m -> Subset of ((AllSymbolsOf S)*\{{}})
  means for U being non empty set,
  I being Element of U-InterpretersOf S holds
    it=dom (I,m)-TruthEval;
  attr w is m-wff means  w in S-formulasOfMaxDepth m;
  attr w is wff means ex m st w is m-wff;
  func AllFormulasOf S equals
```

```
  {x where x is string of S: ex m st x is m-wff};
end;

definition
  let S be Language, U be non empty set;
  let I be Element of U-InterpretersOf S, w be wff string of S;
  func I-TruthEval w -> Element of BOOLEAN means
  for m being Nat st w is m-wff holds it=((I,m)-TruthEval).w;
end;
```

Here only the independence of `dom (I,m)-TruthEval` on `I` and `U` needs to be shown to finally be able to evaluate the truth value of a wff formula, which is omitted here. Let us end this part with stating the remaining semantical definitions implied in the statement of Lowenheim-Skolem and completeness theorems, both traditionally indicated by the double turnstile $\models$; the satisfaction relation (cmp. [EFT84], III.3.1):

```
definition
  let U be non empty set, S be Language;
  let I be Element of U-InterpretersOf S; let X be set;
  attr X is I-satisfied means
  for phi being wff string of S st phi in X holds I-TruthEval phi=1;
end;
```

and the logical implication (entailment):

```
definition
  let X be set, S be Language, phi be wff string of S;
  attr phi is X-implied means
  for U being non empty set, I being Element of U-InterpretersOf S st
  X is I-satisfied holds I-TruthEval phi=1;
end;
```

4.3 Free interpretation

The free interpreter of a given operational symbol s of arity n of a Language S is the operation on the set of n-tuples of terms of S obtained by concatenating the tuple and appending it to the symbol s. Obviously the result is again an element of the set of all terms of S, which now acts as a universe and makes this operation an interpreter as of 4.2.1.

If we add to the picture an arbitrary set X of formulas of S we can talk also of the free interpreter of a relational symbols r of S, of arity $-n \in \mathbb{Z}^-$. In this case an n-tuple of terms is evaluated TRUE if and only if the atomic formula obtained by concatenating and appending to r (the same job done in previous case) belongs to X.

```
definition
  let S be Language, s be ofAtomicFormula Element of S, X be set;
  func X-freeInterpreter(s) -> Interpreter of s,(AllTermsOf S) equals
    s-compound |(abs(ar(s))-tuples_on(AllTermsOf S))
  if not s is relational otherwise
    chi(X,AtomicFormulasOf S) *
```

```
    (s-compound | (abs(ar(s))-tuples_on (AllTermsOf S)));
end;
```

It is worth noting that this definition is also applicable to the equality symbol. This does not matter since, for *any* interpreter, the evaluation of any $\equiv$ atomic formula is overridden at the level of the definition of `-TruthEval` to give the correct value. This is indeed what is meant when talking about a language with equality. The functor `-compound` appearing above is introduced to aid the typing and has a trivial definition (see 4.2 for `-multiCat`):

```
definition
  let S be Language, s be Element of S;
  func s-compound -> Function of ((AllSymbolsOf S)*\{{}})*,
  (AllSymbolsOf S)*\{{}} means for V being Element of
  ((AllSymbolsOf S)*\{{}})* holds it.V = <*s*>^(S-multiCat.V);
end;
```

And finally here is the free interpretation over all the symbols of S, with `AllTermsOf S` as universe.

```
definition
  let S be Language, X be set;
  func (S,X)-freeInterpreter ->
  Element of (AllTermsOf S)-InterpretersOf S means
  dom it=OwnSymbolsOf S & for s being own Element of S holds
    it.s=X-freeInterpreter(s);
end;
```

4.4 Encoding of sequents and of sequent calculus

We end our review by discussing the encoding of sequent calculus. We define what sequents are in just a plain way:

```
definition
  let S be Language; func S-sequents equals
  {[premises,conclusion] where premises is Subset of AllFormulasOf S,
  conclusion is wff string of S: premises is finite};
end;
```

Only notice that `premises` is an (unsorted) finite set, not a n-tuple or a bag.

Since the common way of representing Gentzen-style derivation rules, as already noticed, has more the nature of a diagram rather than that of a precise formulation, encoding them has presented a number of fundamental design choices. When starting from scratch, as in this case, one should put an effort in laying down a structure with enough flexibility and generality to last in time and possibly be reused for other purposes.

The first decision regarded modularization: the framework specifying what a rule is and its general properties has been separated from the description itself of the single rule. This brings some benefits:

—Definitions are terse and readable, compared with other approaches (cmp. e.g. `CALCUL_1`, definition of `is_a_correct_step`).

—The effect of allowing or forbidding the use of a rule can be studied. Indeed, here for each result proved the single rules needed are resolved.
—Possible expansion upon this schemes would be feasible; eg for applying logic flavours other than classical one.

So we first define a framework in which to deal with rules by specifying an abstract `Rule` type:

```
definition
  let S be Language;
  mode Rule of S is
    Element of Funcs (bool (S-sequents), bool (S-sequents));
  mode RuleSet of S is
    Subset of Funcs (bool (S-sequents), bool (S-sequents));
end;
```

One should think of a `Rule` as the function mapping a set X of sequents into the set of all sequents obtainable by applying the Gentzen derivation rule to all the sequents in X).

Having to do generally with deductions using several rules in succession, we introduce the functor `OneStep` to specify all the sequents derivable from some starting sequents using only one rule of a given `RuleSet D`.

```
definition
  let D be RuleSet of S;
  func OneStep(D) -> Rule of S means
  dom it = bool (S-sequents) &
  for Seqs being set st Seqs in dom it holds
    it.Seqs = union ((union D) .: {Seqs});
end;
```

With that, we have started specifying how to pass from rules to deductions, and the next definition will complete the job. Sequent calculus separates the concepts of formal deducibility and of correct proof, so we have two attributes as well; the first is applied to a sequent and certifies it to be derivable from an initial set of sequents, while the second applies to a formula and witnesses it is the tail of a sequent derivable from no assumptions and whose premises are given:

```
definition
  let S be Language, D be RuleSet of S, Seqs1, Seqs2 be set;
  attr Seqs2 is (Seqs1,D)-derivable means
    Seqs2 c= union (((OneStep D) [*]) .: {Seqs1});
  let X,phi be set;
  attr phi is (X,D)-provable means
    ex seqt being set st
      (seqt‘1 c= X & seqt‘2 = phi & {seqt} is ({},D)-derivable)
end;
```

Now we want to code the rules given in section 2 in this framework. We try to separate the jobs of typing from that of actually specifying how a rule works,

by proceeding in stages. Let us take as an example the encoding of the last rule appearing at row (7) in section 2.2 (note that inside the code it is referred to as `Rule6`). First we specify the core of the rule as a Mizar predicate:

```
definition
  let S be Language, x be set;
  attr x is S-null means not contradiction;
end;
definition
  let Seqts be set, S be Language, seqt be S-null set;
  pred seqt Rule6 Seqts means
  ex y1,y2 being set,phi1, phi2 being wff string of S st
    y1 in Seqts & y2 in Seqts & y1‘1 = y2‘1 & y2‘1=seqt‘1 &
    y1‘2= <*TheNorSymbOf S*> ^ phi1 ^ phi1 &
    y2‘2=  <*TheNorSymbOf S*> ^ phi2 ^ phi2 &
    seqt‘2 = <*TheNorSymbOf S*> ^ phi1 ^ phi2;
end;
```

We want at this stage to reduce at a minimum the role of types, to concentrate on the mechanics of the rule, so we declare the starting sequents, represented by `Seqts`, as an untyped variable (a set); at the same time, to do the correct typing later, we need to preserve a link to the type of the specific language `S` we are referring to, so we introduce a fake attribute `-null`, and save it in the variable seqt, which represents the derived sequent (the “denominator”) of the rule.

Now we pass from the predicate `Rule6` to a rule as specified by `Rule` type:

```
definition
  let S be Language, R be Relation of bool (S-sequents), S-sequents;
  func FuncRule(R) -> Rule of S means
  for inseqs being set st inseqs in bool (S-sequents) holds
    it.inseqs={x where x is Element of S-sequents:[inseqs,x] in R};
end;

registration
  let S be Language;
  cluster -> S-null Element of S-sequents;
end;
definition
  let S be Language;
  func P6(S) -> Relation of bool (S-sequents), S-sequents means
  for Seqts being Element of bool (S-sequents), seqt being
  Element of (S-sequents) holds
    [ Seqts, seqt ] in it iff seqt Rule6 Seqts;
end;

definition
let S be Language;
func R6(S) -> Rule of S equals FuncRule(P6(S));
end;
```

When having to code many rules this scheme is convenient because one needs only to define a Mizar predicate without much worrying about typing; afterwards, the rule is easily, and standardly, converted into a `Relation` and finally applied `FuncRule`.

A very last addition has to be made to this encoding scheme for sequent calculus, though. The definition of type `Rule` is arguably very liberal; as a matter of fact, it is indeed a bit too liberal, since it misses prescribing a very reasonable property a rule should have:

```
definition
  let S be Language, R be Rule of S;
  attr R is monotonic means
  Seqts1 c= Seqts2 implies R.Seqts1 c= R.Seqts2;
end;
```

Dealing with one or two sequents in the "higher part of a rule", one usually overlooks this property. We gave a much looser definition of rule, so we have to worry about this property, which turns out to be needed in order to establish very natural passages usually silently agreed on, like ones based on the following result:

```
for D1, D2 being RuleSet of S st
  D1 c= D2 & (D2 is monotonic or D1 is monotonic)
  & Y is (X,D1)-derivable holds
Y is (X,D2)-derivable,
```

where a RuleSet could be called monotonic if every rule in it is such; actually, it suffices an even weaker definition to prove the result above:

```
definition
  let S be Language, D be RuleSet of S;
  attr D is monotonic means
  for Seqts1,Seqts2 being Subset of S-sequents, f being Function st
    Seqts1 c= Seqts2 & f in D
  ex g being Function st g in D & f.Seqts1 c= g.Seqts2;
end;
```

As a side note, it might be interesting to study rulesets which are monotonic in the above weak sense, but not all rules of which are themselves monotonic.

We end this section stating that monotonicity, as such a humble request, is readily proven true for all the rules of section 2.2.

5. CONSIDERATIONS ON SOME PROOF DESIGN ISSUES

Awareness that thoroughly calibrating types when enouncing definition is a key factor for a well-structured proof grew steadily during the work. If one goes too strong, by being too fussy in specifying what type of arguments a functor takes, and at some point faces the need, for example, to apply the same functor to two arguments which differ little, but do not have the same type, in this case he is forced to do double work; also, sometimes a job can be made lighter by adapting an existing type to an affine situation, and base on ready-made formalizations, instead of creating a brand new world of types and having to re-invent the wheel.

On the other hand, being too light with typing one loses the advantages of a tidy formalization given by Mizar. As an example, compare the definitions of atomic wff in [RT90] and in the present work:

```
definition
  let F be Element of QC-WFF;
  attr F is atomic means
:
```

```
definition
let S be Language;
let phi be string of S;
attr phi is atomic means
:
```

The latter definition applies to any string, and not to anything less only because inside the body of the definition there are functors requiring a string (a `FinSequence`) as arguments; on the other hand the first definition restricts the objects to which atomic attribute can be applied. This is likely to complicate forthcoming treatments. One could object that the first solution has the strength of ensuring that 'atomic' implies 'wff'. But this can be attained also in the second case by clustering (see appendix A), which is indeed done in the formalization:

```
registration
let S be Language;
cluster 0-wff -> atomic string of S;
cluster atomic -> 0-wff string of S;
let m be Nat;
cluster m-wff -> wff string of S;
let n be Nat;
cluster (m+0*n)-wff -> (m+n)-wff (string of S);
end;
```

The heavy adoption of attributes and clusters is a trait of the present formalization. Their use has a few advantages: first, a technical one, for they permit to automatically and implicitly reach conclusions which otherwise should be made explicit with a `by` statement; this also brings an advantage in terms of terseness and legibility; finally, they make type-trimming easier, allowing rich typing with relative ease.

In the present case, this is especially true for the classification of the various types of alphabet symbols: literal, compounder, relational, etc..., see 4.1.

A further character of this formalization is the effort to find definitions based on `equals` and `is` rather than those based on `means` when possible. It seems that the former encourage the reusing of pre-existent objects (functors, modes or attributes), at the price of doing the preparatory work of translating the definition to be expressed in terms of those other objects. Definitions thus obtained are arguably more neat and readable, although sometimes less immediate. For sure "`equals`" definitions have a technical advantage resembling that of attributes: they are grasped automatically by Mizar if included in the `definitions` directive, again making life easier and code terser. See [Kor09], section 3. Good examples of this method could be the definitions of the functors `===` (not reviewed here, needed

in construction of `-TruthEval`), `X-freeInterpreter` (see 4.3), `(I,m)-TruthEval` (see 4.2.1), and `ReassignIn` (see sections 4.2.1 and 2.2).

The last example is interesting because it also honors the ideas introduced in section 3: indeed, besides having a clean, `equals`-based definition, it is first introduced for arguments of more general types than we need for our particular case:

```
definition
let x,y be set, f be Function;
func (x,y) ReassignIn f -> Function equals
f +* (x .--> ({} .--> y));
end;
```

Recalling the action of `+*` functor and how we encoded the interpretation of a literal symbol (section 4.2.1), its way of working should be clear. We are leaning of course on a definition (`+*`) given elsewhere, but this permits to use more general tools, avoid restating things, reduce the length of the definition, and, above all, reuse possible results already proven about `+*`. Even if these results were not already available in MML, proving them for a more general, pre-defined object is always better than providing a specialized result framed in a narrower context: somebody else could take advantage of them for developing possibly different areas of MML. Again, as in the first example of this section, we adapt this general definition to our needs by showing this functor returns the expected type when applied to the types we will feed it, using the powerful tool of functorial clustering (see appendix A):

```
registration
  let S be Language,U be non empty set,
  I be (S,U)-interpreter-like Function;
  let x be literal Element of S, u be Element of U;
  cluster (x,u) ReassignIn I -> (S,U)-interpreter-like;
end;
```

Indeed, as noted in section 3, some developments needed in the present work produced results regarding only pre-existing, more general objects: as examples, one could consider the introduction of the `-unambiguous` attribute for generic binary operations, and the related results for the generic monoids, sketched in section 3.1. Here, two more examples, taken again from `FOMODEL0` and which were missing from MML, are exhibited in view of their concise and general statement; they both derived from investigations on how to formalize sequent calculus.

The first regards the transitive closure `R[*]` of a relation `R` and states that it is both transitive and reflexive:

```
registration
  let R be Relation;
  cluster R[*] -> transitive Relation;
  cluster R[*] -> reflexive Relation;
end;
```

The second binds the transitive closure and the iteration of a function:

```
rng f c= dom f implies
```

```
f[*]=union {iter(f,mm) where mm is Element of NAT: not contradiction}
```

6. NUMERICALLY CHARACTERIZING THE FORMALIZATION

We want to estimate formalization cost and de Bruijn factor ([Wie00, ASC10, Nau06]).
There are huge spaces of discretionality, which will be discussed below, in both calculations, so we will make some arbitrary choices, hoping they will result sensible and acceptable.

There are two figures to estimate to trigger calculations: the amount of man hours devoted to formalization and a number measuring the size of a non-formal, human-targeted mathematical text carrying information grossly equivalent to the one formalized.

6.1 Estimating formalizing time

A significant amount of work regarded preliminary reformulation ([Cam09]) rather than Mizar formalization, as discussed in section 2. This portion of work was carried on largely before Mizar formalization even started, however its results were revised 'dynamically' during the formalization as a result of the 'feedback' cited in section 1.2, and as confirmed by the differences noticeable between Mizar code and [Cam09]. Thus, formalization time assessment will be affected by some excess due to this auxiliary work subtracting time to effective coding, and to the fact that the workflow was rather irregular and interleaved with idle periods due to extraneous activities; this last point is probably common to any formalization time estimation.

With the foregoing cautionary remarks, evolution of the codebase is as follow, using Mizar public repository on author's homepage as a development history record. The first Mizar file ever written by the author dates back to 24th January 2010, and, since then, formalization and Mizar learning efforts went on concurrently; the first codebase including Gödel's completeness theorem was successfully checked on 12th October 2010.

Lowenheim-Skolem was first successfully compiled on 5th November 2010. As a conclusion, formalizing time can be estimated in 284 days.

6.2 Establishing a non-formal, equivalent mathematical source text

For the reasons exposed in 6.1, choosing a denominator to compute de Bruijn factor is not so straightforward in this case. The nearest treatment would obviously be [Cam09], which, however, merely highlights the points in the proof which are novel and less trivial, and silently assumes a lot of prerequisites. Instead, the low starting point of this formalization demands we choose a more thorough treatment as a fairer reference, with an exposition starting from scratch (alphabets, strings, etc...) as this formalization does, and not omitting the tedious and 'trivial' details. Since [EFT84], being an undergraduate text book, arguably satisfies these requirements and was the original source of inspiration, it seems a good candidate. Specifically, we OCRed its scans and selected the excerpt going from section II.1 ('Alphabets', page 10) through section VI.1 ('The Lowenheim-Skolem Theorem', ending on page 89), taking the resulting ASCII text as our non-formal source text. It is available on author's home page for reference. We have not removed the dispensable bits occurring in this source (exercises, historical notes, examples); first, they can be

considered quantitatively negligible for our purposes, especially if one consider how arbitrary the whole matter is; secondarily, if one regards de Bruijn factor as a fundamental ratio between how much information is needed for a machine to accept statements and how much information is needed for a human to accept the same statements, rather than a totally empirical indicator to practically compare formalization verbosities, he could consider those bits as effectively useful for that human reader to accept (assimilate, he would say) those statements.

6.3 Results

The formalization cost is then calculated to be

$$\frac{\frac{284}{7}}{89-10+1} = 0.5 \text{ weeks per page}$$

The de Bruijn factor is shown below:

	informal (bytes)	formal (bytes)	de Bruijn factor	
uncompressed	132495	710144	5.4	apparent
gzipped	46839	153399	3.3	intrinsic

7. CONCLUSIONS AND FURTHER DEVELOPMENTS

We discussed the reasons to introduce an alternative construction for first-order languages in MML. We exposed a model for a first-order language oriented toward simplicity, conceived with Mizar formalization in mind.
Then we listed a set of rules devised in the same spirit.
This set has been shown to be both correct and complete: using these constructions, both Gödel's completeness and Lowenheim-Skolem theorems have been Mizar-verified.
The discussion proceeded with a general, algebraic-oriented treatment, based on the notion of an unambiguous set with respect to a generic binary operation, permitted an alternative definition of the subterms of a term.
After that, we passed to displaying the main points of the Mizar formalization of these objects, and the formalization, in this framework, of the basic theoretical tools for syntax and semantics. At this stage, an alternative design unifying the formalizations of syntax and evaluation of non-atomic formulas, charted to decrease the amount of Mizar code needed to introduce them, has been proposed.
Continuing in the illustration of our Mizar formulations, we passed to the definition of a sequent and of a derivation rule. In doing that, we pointed out that our definition of rule has been split in stages, allowing to "plug" additional rules if needed, by separating the definition of a rule from its actual specification, and by defining derivability and provability for a generic, unspecified set of rules. Also, we noticed how the specification of a specific rule has been separated in stages as well, with a first stage describing the very mechanics of the rule, and later stages to easily accommodate its typing inside the framework introduced for sequent calculus. Finally, considerations on some design choices faced during the work were given, and numerical estimations to characterize the formalization were calculated, resulting in an intrisic de Bruijn factor of 3.3 and in a cost factor of 0.5.

Basing on what already done, a further step could be the verification of Lowenheim-Skolem theorem for higher cardinalities and in the 'upward' direction, by employing some form of choice. Usually in the present countable case, its place can be taken by König's lemma (see e.g. [Jer96, Avi10]): so it could also be of interest to investigate, tracing axiom dependency via Mizar, the role of König's lemma in this formalization.

ACKNOWLEDGMENTS

I wish to thank Marco Pedicini and Mario Piazza for kind suggestions and useful discussions.

References

[ASC10] A. Asperti and C. Sacerdoti Coen. Some Considerations on the Usability of Interactive Provers. In *Intelligent Computer Mathematics: 10th International Conference, Aisc 2010, 17th Symposium, Calculemus 2010, and 9th International Conference, Mkm 2010, Paris, France, July 5-10, 2010. Proceedings*, page 147, 2010.

[Avi10] J. Avigad. Gödel and the metamathematical tradition. *Kurt Gödel: Essays for His Centennial*, page 45, 2010.

[Ban90] G. Bancerek. A model of ZF set theory language. *Formalized Mathematics*, 1(1):131–145, 1990.

[BR02] G. Bancerek and P. Rudnicki. A compendium of continuous lattices in mizar. *Journal of Automated Reasoning*, 29(3):189–224, 2002.

[Cam09] M.B. Caminati. Yet another proof of Goedel's completeness theorem for first-order classical logic. *Arxiv preprint arXiv:0910.2059*, 2009.

[EFT84] H.D. Ebbinghaus, J. Flum, and W. Thomas. *Mathematical Logic*. Undergraduate Texts in Mathematics. Springer-Verlag, second edition, 1984. ISBN 0387908951.

[Har98] J. Harrison. Formalizing basic first order model theory. *Theorem Proving in Higher Order Logics*, pages 153–170, 1998.

[Jaś34] S. Jaśkowski. *On the rules of suppositions in formal logic*. Nakładem Seminarjum Filozoficznego Wydziału Matematyczno-Przyrodniczego Uniwersytetu Warszawskiego, 1934.

[Jer96] H.R. Jervell. Thoralf Skolem Pioneer of Computational Logic. *Nordic Journal of Philosophical Logic*, 1(2):107–117, 1996.

[KMK92] J. Kotowicz, B. Madras, and M. Korolkiewicz. Basic notation of universal algebra. *Journal of Formalized Mathematics*, 4, 1992.

[Kor09] A. Kornilowicz. How to Define Terms in Mizar Effectively. *Studies in Logic, Grammar and Rhetoric*, 18(31):67–77, 2009.

[Lot02] M. Lothaire. *Algebraic combinatorics on words*. Cambridge Univ Pr, 2002.

[Nau06] A. Naumowicz. An example of formalizing recent mathematical results in Mizar. *Journal of Applied Logic*, 4(4):396–413, 2006.

[Ono62] K. Ono. On a practical way of describing formal deductions. *Nagoya Mathematical Journal*, 21:115–121, 1962.

[RSN63] H. Rasiowa, R. Sikorski, and P.A. Nauk. *The mathematics of metamathematics*. PWN Warszawa, 1963.

[RT90] P. Rudnicki and A. Trybulec. A first order language. *Formalized Mathematics*, 1(2):303–311, 1990.

[RT99] P. Rudnicki and A. Trybulec. On equivalents of well-foundedness. *Journal of Automated Reasoning*, 23(3):197–234, 1999.

[Rud92] P. Rudnicki. An overview of the Mizar project. In *Proceedings of the 1992 Workshop on Types for Proofs and Programs*, pages 311–332. Citeseer, 1992.

[Wie00] F. Wiedijk. The De Bruijn Factor. *preprint*, 2000. http://www.cs.ru.nl/~freek/factor/factor.pdf.

[Wie07] F. Wiedijk. Mizar's soft type system. In *Proceedings of the 20th international conference on Theorem proving in higher order logics*, pages 383–399. Springer-Verlag, 2007.

A. A QUICK OVERVIEW OF MIZAR

What is nowadays customarily referred to as Mizar (`www.mizar.org`) [Rud92], [RT99], is an enduring mathematical knowledge formalization project consisting of a language to code first-order logic with schemes, a small core of fixed axioms (Tarski-Grothendieck, see [RT99]) expressed in this language, a checker executable (PC Mizar) to certify the correctness according to the latter of natural deduction (in a flavour adhering to those described in [Jaś34] and [Ono62]) proofs, and a library (MML) of certified results which has grown quite massive and diverse throughout the decades the project has seen. Currently, MML is the only part of the project open to external contributions.

Being based on set theory, the Mizar language is an untyped one. Virtually, every (first-order) Mizar formula corresponds to a string of $\{, \}, \in, \forall, \exists, \wedge, \vee, \neg, \implies, \iff, =$ and variables symbols. For users' convenience, however, there are facilities to overlay those untamed set-theoretic strings with orderly structures. Given the aspiration of Mizar to be a universal library aiming to be referenced to by any mathematician, besides a mere proof-certifying entity, these facilities have a fundamental role also for the readability of MML, apart from being decisive tools for the coder of the proofs. Indeed, they make Mizar language very close to mathematicians' common way of expressing mathematics, e.g. the one instinctively put down when they face chalk and blackboard. The present paper itself leans on this marked readability of Mizar language: we extensively exhibited pieces of code as commentary of themselves, ensured of course by the fact that Mizar checker already took care of proving the corresponding mathematics correct.

First, one can define a type as a family of sets satisfying a given condition. Given that, one then can also define functions (called *functors* in Mizar parlance, though no link with the notion of category theory exists; as explained in [BR02], the name is drawn from a passage on [RSN63], V.1, and serves to emphasize that they map between strings of the Mizar meta-language, and are different from the first-order `function` type) operating on a finite list of arguments with specified types, much like one defines functions in a procedural programming language (check table I for some occurring in this paper). Along with functors, one can define predicates,

`f"X`	preimage of the set X through f	$f^{-1}(X)$
`X/\Y`, `X\/Y`, `X\Y`	basic set-theoretical operations	$X \cup Y, X \cap Y, X \backslash Y$
`[x,y]`	Kuratowski ordered pair	(x, y)
`[:X,Y:]`	cartesian product of sets	$X \times Y$
`NAT, INT`		$\mathbb{N}, \mathbb{Z}$
`X*, n-tuples_on X`	sets of finite and n-long X-words	X^*, X^n
`<*s*>`	the string made of the lone char s	`^s$`
`p^q`	concatenation of strings p and q	$p \vert\vert q$
`dom f, rng f`	domain and range of a relation f	
`p/^n`	the string p with the first n chars removed	
`bool X`	the power set of X	2^X
`f.x`	the value of the function f in x	$f(x)$
`f +* g`	the pasting of the functions f, g	
`curry`	currying	$((x, y) \mapsto f(x, y)) \mapsto (x \mapsto (y \mapsto f(x, y)))$
`f * g`	functional composition	$f \circ g$
`f.:X`	image of the set X through f	$f[X]$
`[x,y]‘1=x` `[x,y]‘2=y`	projectors for Kuratowski pairs	
`Funcs(X,Y)`	the set of functions from X to Y	Y^X
`PFuncs(X,Y)`	the set of partfunctions from X to Y	$\bigcup_{x \subseteq X} Y^x$
`R[*]`	finite iterations of a relation R, aka the transitive closure of R	
`X --> y`	the constant y-valued function on X	$X \mapsto \{y\}$
`x.-->y = {x}-->y`	function between two singlets	
`chi(Y,X)`	characteristic function of $Y \subseteq X$	$1_Y : X \mapsto \{0, 1\}$

Table I. Notable Mizar definitions used in the paper, pre-existing in MML

which works the same way, except that the former return a term of first-order language, while the latter return a truth value. Reasoning in the same manner, a third very special kind of constructor has been introduced in Mizar, called a mode. It does not return a term neither a truth value from its arguments: it returns a new type, a *dependent* type. Finally, any type can be appended an attribute to further subdivide it into subtypes. The pair (type, attribute) becomes a new type and so on. A nice feature of attributes is that Mizar provides a mechanism to automatically attach an attribute to a given type once proven this type indeed possessing the corresponding property. This mechanism, called clustering, has a quite powerful application scheme, and was extensively exploited in this work.

It is important to stress that all these overlays are added at the level of the meta-language, not at the level of first-order, set theory language (sometimes it is said that Mizar has a *soft* type system, [Wie07]). All the same, the difficulty of formalizing in Mizar without taking advantages of them could be vaguely compared to the one of coding in machine language without using a higher-level language.