Treating Sets as Types in a Proof Assistant for Ordinary Mathematics

Sebastian Reichelt

September 4, 2010

Abstract

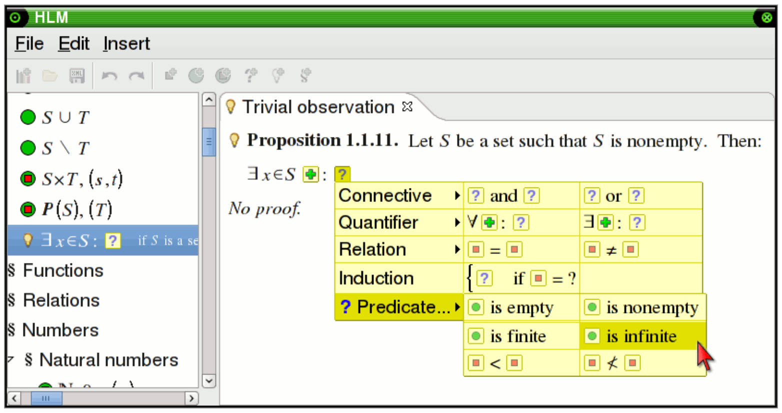
I present a proof assistant with a novel graphical user interface, which not only renders user input in a familiar mathematical style, but completely abandons textual input in favor of a menu-driven approach. In order to fill each menu precisely with the set of intuitively meaningful choices, the software employs a custom formal system which appears set-theoretic from the user's point of view but is based on types internally.

1 Introduction

The user interaction of all major proof assistants, whether declarative or procedural, is inherently text-based and focused on *input*. After mentally formalizing definitions and theorems in the language of the underlying formal system, the user needs to encode the result into a program-specific syntax. Efforts to increase user-friendliness have thus dealt primarily with streamlining this encoding, by introducing syntactical features such as symbol overloading [1] and automatically inferred arguments [10], by adding more sophisticated automation [23], or simply by making the syntax more readable [19]. While this strategy has led to notable successes in hardware and software verification [9, 13], the formalization of ordinary mathematics has not progressed equally well yet [21].

Two of the most frequently cited reasons are the high overhead of formalization ("de Bruijn factor"), especially in terms of time, and a lack of participation by the mathematical community (attributed in part to said overhead and in part to the remoteness of proof assistants from mathematical practice) [21]. I believe that further syntax or automation improvements cannot significantly reduce the time overhead or make the input resemble ordinary mathematics more closely. Instead, I would like to propose an entirely different proof assistant design, covering both the software and the formal system it is based on — and present a prototype implementation.

The main idea is to seize all opportunities for interaction and feedback that arise in a graphical user interface (GUI). While current proof assistant GUIs are essentially wrappers around text-based interfaces with (very valuable) additional features [1, 2], an all-encompassing GUI can let the user operate directly on the



Proposition 1.1.11. Let S be a set such that S is nonempty. Then:

Figure 1: Inserting a formula within a new theorem.

mathematical data structures instead of parsing text. Thus, no keyboard input is required except for names; everything else can be selected from context-specific menus (see figure 1). As a result, the focus automatically shifts from input to *output*: The struggle for the most efficient syntax is superseded by a quest for the most intuitive menus and the most efficient navigation and search facilities. For example, if the user can specify freely how a definition should be rendered, and the custom rendering is used even in menus, finding the right definition can become a matter of seconds.

As a consequence of this design, user-friendliness depends unusually strongly on how well the data structures and the underlying formal system match the user's expectations. For a proof assistant intended for all areas of mathematics, two seemingly competing requirements emerge:

First, to achieve the most "mathematical" rendering, the internal data representation must itself resemble mathematical practice as closely as possible. From this perspective, a first-order axiomatic set theory like Zermelo-Fraenkel would be the obvious choice, given today's widespread use of set-theoretical principles and notation. But second, there is suddenly a substantial benefit in *restricting* the possible choices to those that are meaningful in a particular situation. De Bruijn once remarked that ZF lets us "ask whether the union of the cosine function and the number e contains a finite geometry" [5, 7]. Unfortunately, no set theory rejects all such nonsense questions.

Instead, that task falls squarely within the domain of type theory [11]. Indeed, many proof assistants are based on some variant of type theory, though mostly for different reasons such as its connection to computer science. Alas, type theory carries an even greater potential to confuse users with an ordinary mathematical background than meaningless choices in menus could possibly acquire, simply for not being like set theory. The most obvious show-stopper is the non-trivial correspondence between subsets in set theory and functions yielding user-defined Booleans or Curry-Howard-style propositions in type theory.

To meet both criteria at the same time, I have developed a formal system that merely *interprets* ordinary mathematics in terms of types. Although this idea is hardly new [5, 14] and arguably dates back to the beginnings of type theory, the particular formal system appears to be unique in satisfying all of the following properties:

- There is a simple mechanical translation from the "internal" language of the system to common set-theoretical notation.
- Almost the entire body of existing mathematics can be formalized in the system, including general reasoning about sets. In other words, the formal system qualifies as a set theory, for a sufficiently broad definition of "set theory."
- Given a rigorously but informally formulated definition or theorem, the proper formalization is immediately obvious. No obscure "coding" is required.
- Validity in the formal system coincides extremely closely with intuitive meaningfulness. The rules of the system prevent all constructs that would be considered "type errors." In other words, the formal system qualifies as a type theory, for a sufficiently broad definition of "type theory."

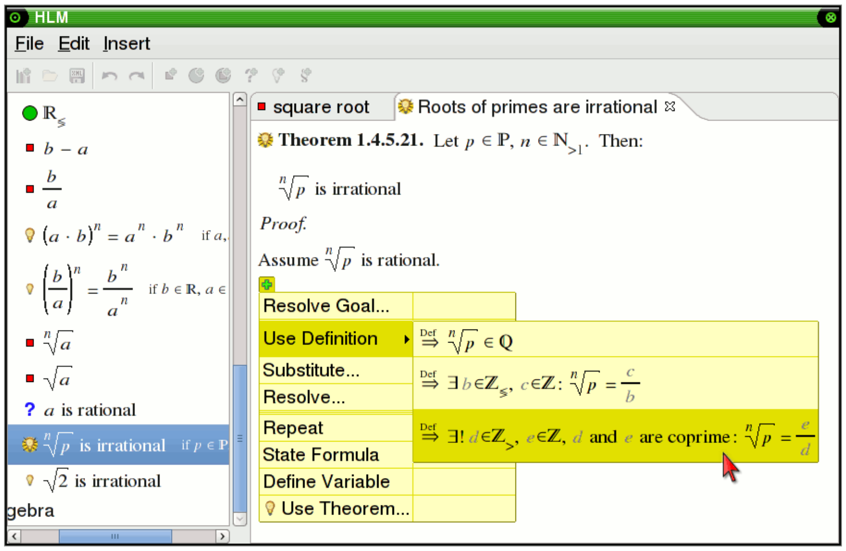
The rest of this paper is organized as follows: In section 2, I discuss the relationship of the presented work to some existing proof assistants and formal systems. Section 3 contains a description of the formal system along with its connection to the proof assistant, with a focus on the type mechanism. Section 4 reflects on some surprising properties that emerge during the development of the system, as well as current weak spots that call for extensions.

2 Related Work

Proof assistants. Among existing proof assistants, Mizar [18] is probably closest in spirit to the software presented in this paper. Like Mizar, my proof assistant is designed for mathematics rather than computer science (although using it for program verification is certainly possible). Especially, the Mizar syntax and proof style are intended to resemble ordinary mathematical notation, which is one of my primary goals as well. The biggest difference is that like all major proof assistants, Mizar takes text as input.

Although Mizar is based on ZF-style set theory, it also features a type system [20]. Its types serve several purposes related to user-friendliness, such as symbol overloading, guarded quantification, and automatic inferences. Nevertheless, since they are optional, such types would be insufficient for the purpose of restricting choices in a GUI.

Proofs in Mizar are declarative and therefore comparatively readable, which has inspired declarative add-ons to other proof assistants such as HOL [8], Coq [4], and Isabelle [17]. Most other systems natively use a procedural style,



$\mathbb{R}_{\leq}$

$b - a$

$\frac{b}{a}$

$(a \cdot b)^n = a^n \cdot b^n$ if a,

$\left(\frac{b}{a}\right)^n = \frac{b^n}{a^n}$ if $b \in \mathbb{R}$, $a \in$

$\sqrt[n]{a}$

$\sqrt{a}$

? a is rational

$\sqrt[n]{p}$ is irrational if $p \in \mathbb{P}$

$\sqrt{2}$ is irrational

Theorem 1.4.5.21. Let $p \in \mathbb{P}$, $n \in \mathbb{N}_{>1}$. Then:

$\sqrt[n]{p}$ is irrational

Proof.

Assume $\sqrt[n]{p}$ is rational.

$\overset{\text{Def}}{\Rightarrow} \sqrt[n]{p} \in \mathbb{Q}$

$\overset{\text{Def}}{\Rightarrow} \exists\, b \in \mathbb{Z}_{\neq}, c \in \mathbb{Z} : \sqrt[n]{p} = \frac{c}{b}$

$\overset{\text{Def}}{\Rightarrow} \exists!\, d \in \mathbb{Z}_{>}, e \in \mathbb{Z}$, d and e are coprime $: \sqrt[n]{p} = \frac{e}{d}$

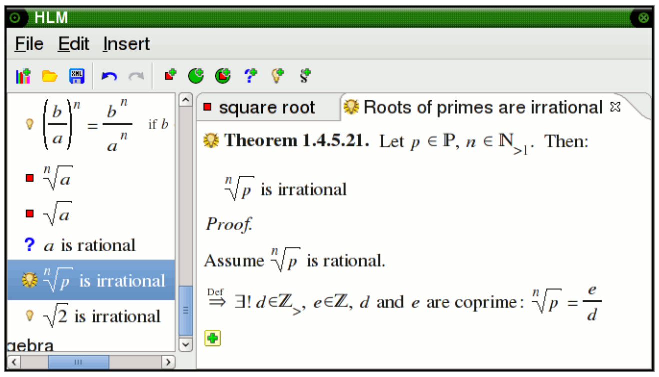
Figure 2: Inserting a proof step.

which has the advantage of smaller proof sizes, immediate feedback, and better programmability [6, 8]. There also exist efforts to merge both styles [22], since from a user's point of view, the proof style constitutes the main difference between programs.

In a GUI, the dichotomy between declarative and procedural input can be avoided. At any point in a proof, my program presents a list of possible next steps, which roughly correspond to the tactics of procedural proof assistants (see figure 2). A similar input method is found in Papuq [16], an IDE for the Coq proof assistant. The crucial difference is that every step selected by the user is inserted into the proof in a declarative style (see figure 3). Thus, the user gets immediate feedback (and cannot even select anything that is not a correct inference), while at the same time the result is a human-readable proof.

Formal systems. Although the differences in input method and proof style are the most visible, the essence of my contribution lies in the formal system that makes all advances possible in the first place. To be able to compare it with established systems, it is necessary to highlight a crucial divergence in the approach taken to arrive at its rules. The result of this approach is a formal system based neither on (whatever-order) predicate logic nor on (typed/untyped) lambda calculus.

All formal systems that serve as a foundation of mathematics can be regarded as an abstraction of mathematical practice. Typically, one strives to eliminate as many individual concepts as possible while keeping the same strength, by translating the removed concepts into the remaining ones. For example, the concept of 'definition' is usually understood to lie outside of the scope of a formal system, as a definition can be eliminated by substituting its contents at


HLM
File Edit Insert
square root
Roots of primes are irrational
$\left(\frac{b}{a}\right)^n = \frac{b^n}{a^n}$ if b
$\sqrt[n]{a}$
$\sqrt{a}$
? a is rational
$\sqrt[n]{p}$ is irrational
$\sqrt{2}$ is irrational
gebra

Theorem 1.4.5.21. Let $p \in \mathbb{P}$, $n \in \mathbb{N}_{>1}$. Then:

$\sqrt[n]{p}$ is irrational

Proof.

Assume $\sqrt[n]{p}$ is rational.

$\overset{\text{Def}}{\Rightarrow} \exists!\, d \in \mathbb{Z}_{>}, e \in \mathbb{Z}, d$ and e are coprime: $\sqrt[n]{p} = \frac{e}{d}$

Figure 3: The proof step of figure 2 has been inserted.

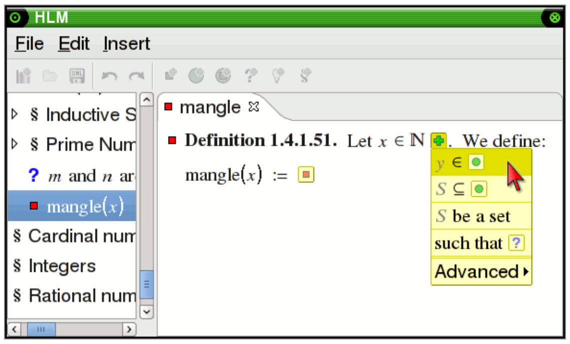
all places where it is instantiated (though formal systems with 'definitions' exist, e.g. DZFC and PST [12], as well as the Calculus of Inductive Constructions [15]). Similarly, when working in a set theory, notions such as 'function' or 'number' (that could be considered primitive in informal mathematics) are defined in terms of sets. Even the concept of 'set' becomes irrelevant if all objects of discourse are in fact sets, as is the case in ZF.

In the introduction, I argued that my formal system must be sufficiently rich to be able to render its contents in mathematical notation, and to offer a selection between meaningful alternatives in every situation. Therefore, it contains a number of concepts that are normally abstracted away. These include 'definition' and 'set' (as a primitive notion, not described by axioms) as well as some new concepts. The primary motivation for this design is that it enables another central concept, that of a 'type,' to be hidden from the user in the proof assistant.

Nevertheless, functions, relations, numbers, groups, vector spaces, categories, etc. can all be defined on top of the formal system as usual. (That is, in the proof assistant, they are part of the library.) However, the construction of such objects requires far less arbitrary coding than in pure set theories, due to a 'construction' concept that resembles inductive type declarations in type-theoretic proof assistants.

3 The Formal System

In this paper, I present both a proof assistant and its underlying formal system, with an emphasis on the latter. Both are strongly related, perhaps unusually so: While the standard idea is that a proof assistant is *based* on a formal system, in this case the formal system can alternatively be regarded as an *abstraction* of the *data structures* of the program. Since these data structures are also clearly represented in the graphical user interface, I will frequently use screenshots of the proof assistant as a more concrete exposition of abstract concepts.



Definition 1.4.1.51. Let $x \in \mathbb{N}$. We define:

$\text{mangle}(x) :=$

Figure 4: Inserting a second parameter (after first parameter x).

The user-defined mathematical content of the proof assistant is organized in a single ‘library,’ which is essentially a large hierarchical data structure without any dependencies on the rest of the program. A library consists of an unordered list of ‘definitions’ and ‘theorems.’ As indicated, these three concepts carry over verbatim to the formal system, which distinguishes this system from most others.

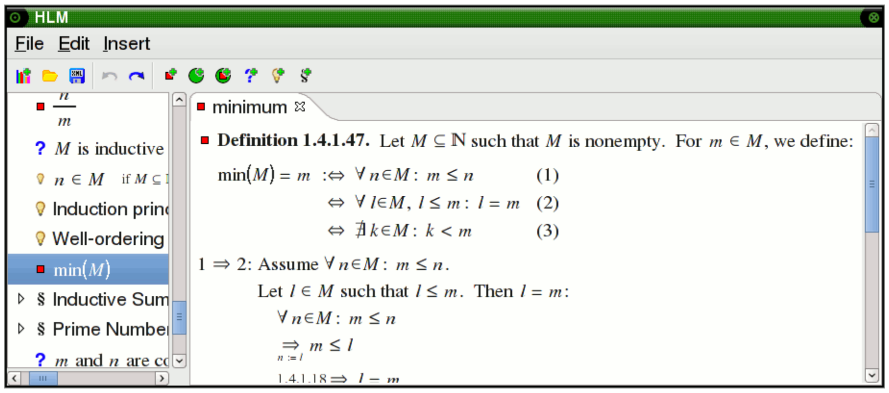
Definitions and theorems start with a list of ‘parameters.’ Intuitively, these parameters play a similar role as function parameters in type-theoretic proof assistants and programming languages, and indeed, they

- introduce variables that are used in the body of the definition or theorem,
- are substituted by ‘arguments’ whenever the definition/theorem is used, and
- set the stage for the type mechanism.

It is important, however, *not* to think of definitions with parameters as functions in the mathematical sense, as the latter are defined in the library in the usual manner as subsets of Cartesian products. In this regard, my system differs strongly from existing type theories, which tend to have built-in function types.

Figure 4 shows a screenshot of a definition with one parameter and a menu for adding a second parameter. The menu contents depict the four basic kinds of parameters that exist:

- The first is called an ‘element parameter’ and rendered as “Let $x \in \ldots$” To avoid any confusion, I would like to stress that the ‘$\in$’ symbol is considered part of the parameter and thus *not* identical to the same symbol in a *formula* “$x \in \ldots$” (Since symbols are never entered but merely rendered, overloading them is entirely unproblematic as long as it cannot cause any confusion.)
- The second is called a ‘subset parameter’ and rendered as “Let $S \subseteq \ldots$”



Definition 1.4.1.47. Let $M \subseteq \mathbb{N}$ such that M is nonempty. For $m \in M$, we define:

$$\min(M) = m \;:\Leftrightarrow\; \forall\, n{\in}M:\ m \le n \qquad (1)$$
$$\Leftrightarrow\; \forall\, l{\in}M,\ l \le m:\ l = m \qquad (2)$$
$$\Leftrightarrow\; \nexists\, k{\in}M:\ k < m \qquad (3)$$

$1 \Rightarrow 2$: Assume $\forall\, n{\in}M:\ m \le n$.

Let $l \in M$ such that $l \le m$. Then $l = m$:

$\forall\, n{\in}M:\ m \le n$

$\underset{n := l}{\Rightarrow}\ m \le l$

$1.4.1.18 \Rightarrow\ l = m$

Figure 5: An implicit definition with multiple equivalent alternatives.

- The third is called an ‘arbitrary set parameter’ and rendered as “Let S be a set.”

- The fourth is called a ‘constraint parameter’ and rendered as “Assume . . .” or ”such that . . .”

The reason for introducing these specific kinds of parameters is closely related to the type system and will become apparent in the next section. However, the existence of ‘element,’ ‘subset,’ and ‘arbitrary set’ parameters already highlights a special property of the formal system: the role of sets as a concept, rather than as objects. Not only is set-related terminology built into the system at a central place where one might not expect it (in particular, the introduction of variables), it also serves to classify variables in a way that would not be possible if sets were treated as objects: We can call each variable introduced by an element parameter an ‘element variable,’ and each variable introduced by a subset or arbitrary set parameter a ‘set variable.’ This distinction becomes a key ingredient of the “sets as types” paradigm.

What follows after the parameters depends on whether one is stating a theorem or a definition, and in the latter case, what kind of definition. The details are somewhat complex and arbitrary; consider for instance implicit definitions (see figure 5). Therefore, I will confine myself to the most important aspects of formulae and terms:

- The distinction between element variables and set variables carries over to terms. (Power sets, which intuitively seem to break this scheme, can in fact be handled easily using an additional “indirection” – see section 3.2.)

- Formulae of the form “$x = y$,” “$x \in S$,” and “$S \subseteq T$” are considered primitive. Here, x and y denote element terms, whereas S and T denote set terms.

- Quantifiers are followed by parameters matching those described above, instead of just variable names. (Of course, words like "let" are omitted here – see figure 5.)

- There is a primitive set term of the form "$\{x \in S : \ldots\}$." The "$x \in S$" part is actually an element parameter.
- Since definitions are part of the language, so is their instantiation. An element term must be provided as an 'argument' for each element parameter, and a set term for each subset or arbitrary set parameter. Moreover, all constraints must be provably satisfied. Depending on the kind of definition, the result is an element or set term, or a formula.

This rough overview should suffice to convey at least a vague intuition about what can and cannot be said in the language of the system. For example, a formula like "$x \in x$" is never syntactically valid because x would have to be an element variable and a set variable at the same time. Still, so far, nothing would prevent one from asking e.g. whether a given natural number is in the set of finite directed graphs. This is where types enter the scene.

3.1 Types

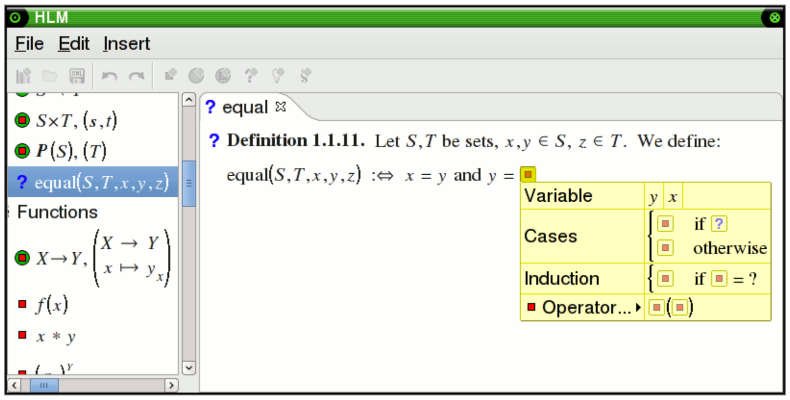
To explain the idea behind the type system, I will first state its rules in an informal fashion.

- The formula "$x = y$," introduced as primitive above for element terms x and y, is valid if and only if there exists a set term T such that both $x \in T$ and $y \in T$ can be determined to hold on a purely syntactical basis.
- Similarly, the formula "$x \in S$" is valid if and only if x can be determined to be a member of some superset of S.
- Finally, the formula "$S \subseteq T$" is valid if and only if S and T can be determined to have a common superset.

The above rules describe the essence of the "sets as types" paradigm: The language of the system is inherently set-theoretic; in particular, the $\in$ and $\subseteq$ symbols are used just like in ordinary mathematics. However, they have a radically different status compared to their role in axiomatic set theory; they are both governed by and play a part in the type rules.

It is easy to see that the rules, when made sufficiently precise, ensure that all such formulae are meaningful. It certainly makes sense to ask whether two members of a single given set are equal, or whether a certain member of a set is also in one of its subsets. To support the reverse claim – that every ordinary mathematical statement can be phrased in accordance with the given rules – I can only point to my proof assistant and its present library.

To make the informal rules precise, each element or set term is recursively assigned a 'type.'


Definition 1.1.11. Let S, T be sets, $x, y \in S$, $z \in T$. We define:

$\mathrm{equal}(S,T,x,y,z) :\Leftrightarrow x = y$ and $y =$

Figure 6: Type rules prohibit the use of z.

- The type of an element term x referring to a parameter introduced as "let $x \in T$" is defined to be the same as the type of T. (There is no need to introduce an "element of T" type because element terms and set terms are disjoint.)
- The type of a set term S referring to a parameter introduced as "let $S \subseteq T$" is also defined to be the same as the type of T.
- The type of a set term of the form "$\{x \in T : \ldots\}$" is, again, defined to be the same as the type of T.
- The type of a term that references a definition is determined by the contents of the definition, with all occurrences of variables substituted by their arguments.

This definition reduces every type to that of a set term S referring to an 'arbitrary set' parameter introduced as "let S be a set." Any two such types are considered distinct, so formally, the type of such an S is defined to be S itself. Then, the informal principle can be rephrased as:

> Formulae of the form "$x = y$," "$x \in S$," and "$S \subseteq T$" are valid if and only if x, y, S, and T have the same type.

Since the type of a term can be determined algorithmically, there is no need to even mention the word 'type' in the proof assistant. Instead, types work in the background to eliminate meaningless choices. For example, in figure 6, the variable z is simply omitted from the menu. More usefully, in figure 7, the most-recently-used list is filled with precisely the items that (can) yield functions, and in the case of the identity function, the software infers automatically that it must be the identity on X, the domain of f. (For the same reason, the variable f cannot be inserted at the given position.)

The concepts and primitives described thus far are not sufficient to deal with concrete objects (like those seen in figure 7), but do permit the statement of

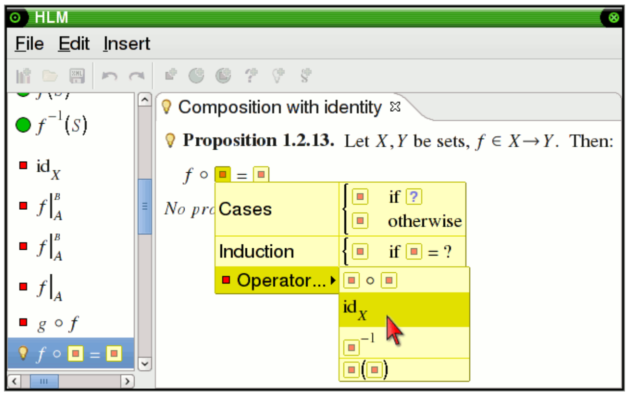

HLM
File Edit Insert
Composition with identity
Proposition 1.2.13. Let X, Y be sets, $f \in X \to Y$. Then:
Cases
Induction
Operator...
if
otherwise
id_X

Figure 7: Appropriately filled most-recently-used list, and automatic type inference (X). (Also note how the software can render every symbol according to a user-defined notation and layout.)

some abstract definitions and theorems. One example would be the intersection or union of two sets S and T that have the same type:

> Let U be a set, $S, T \subseteq U$. We define:
>
> $$S \cup_U T := \{x \in U : x \in S \vee x \in T\}$$

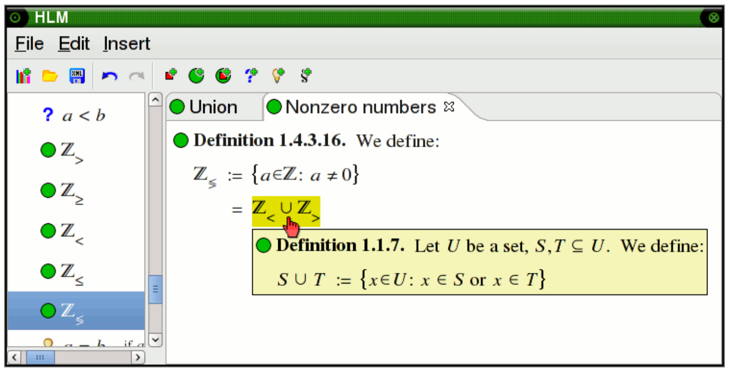
The occurrence of an arbitrary set U in "$S \cup_U T$" would obviously be irritating, but since every valid argument will provably yield the same result, it can be inferred automatically and therefore omitted (see figure 8). However, U needs to be part of the definition to ensure that S and T have the same type.

The last major missing piece is the introduction of concrete objects to get the system off the ground.

3.2 Constructions

The usual approach for the construction of sets and objects is to postulate axioms that guarantee their existence and uniqueness. However, since axioms vary from system to system, they are seldom referred to in practice. Thus, a more high-level mechanism for set construction can be beneficial from a usability standpoint. There is also the technical issue that "the unique set S satisfying property $P(S)$" is not a valid term, as two arbitrary sets are considered uncomparable.

Therefore, the notion of 'axioms' is abandoned in favor of set construction rules that directly yield the sets occurring in practice, such as Cartesian products, power sets, sets of numbers, etc. In particular, 'constructions' are one of the several kinds of definitions that can occur in a library. In contrast to other definitions, a construction defines a set and its elements at the same time.



Definition 1.4.3.16. We define:

$$\mathbb{Z}_\lessgtr := \{a \in \mathbb{Z} : a \neq 0\} = \mathbb{Z}_< \cup \mathbb{Z}_>$$

Definition 1.1.7. Let U be a set, $S, T \subseteq U$. We define:

$$S \cup T := \{x \in U : x \in S \text{ or } x \in T\}$$

Figure 8: Omission of superset U from the union symbol.

A construction consists of one or more 'constructors,' each of which contains its own parameter list, in addition to the parameter list of the entire construction. In general, it is written as follows:

construction parameters

$$\text{construction} :=: \left\{ \begin{array}{c|c} \mathsf{constructor}_1 & \text{constructor}_1 \text{ parameters} \\ \mathsf{constructor}_2 & \text{constructor}_2 \text{ parameters} \\ \vdots & \vdots \end{array} \right\}$$

For example, a construction with two arbitrary set parameters and a single constructor with two appropriate element parameters yields the Cartesian product of the two sets:

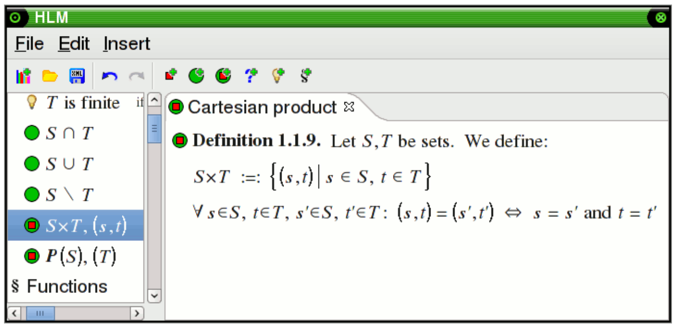
Let S, T be sets. We define:

$$S \times T :=: \left\{ \mathsf{pair}_{S \times T}(s, t) \;\middle|\; s \in S,\ t \in T \right\}$$

The usual notation is obtained by omitting everything but the parentheses from the pair constructor (see figure 9).

With appropriate rules concerning circularities, the set of natural numbers can be defined easily using two constructors (see figure 10). Although the idea is similar to inductive types in functional programming languages and proof assistants based on type theory [3], the use of parameter lists makes constructors more versatile. For example, to define power sets or sets of functions, one merely needs to add a single constructor with a subset parameter (see figure 11). Therefore, in contrast to existing type theories, functions do not need to have any special status.

For constructors using element and subset parameters, two instances of the same constructor can be considered equal if their arguments are. For arbitrary set parameters, the sets given as arguments do not necessarily have the same



Definition 1.1.9. Let S,T be sets. We define:

$$S \times T \;:=:\; \left\{(s,t) \,\middle|\, s \in S,\, t \in T\right\}$$

$$\forall\, s \in S,\, t \in T,\, s' \in S,\, t' \in T:\; (s,t) = (s',t') \;\Leftrightarrow\; s = s' \text{ and } t = t'$$

Figure 9: Definition and custom rendering of Cartesian product.

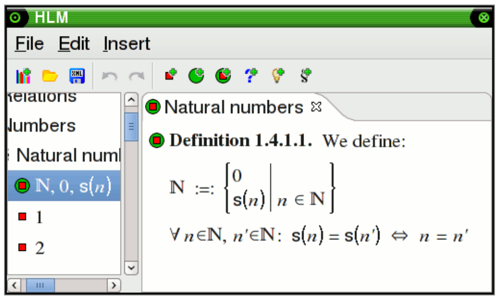
type, and thus cannot be compared. A general solution is to let the user specify when two instances should be equal, subject to the requirement of reflexivity, symmetry, and transitivity. This is useful even if no arbitrary set parameters are involved; for example when defining integers (see figure 12). With arbitrary set parameters, it leads to some interesting constructions:

- A single arbitrary set parameter defines cardinal numbers (or equivalence classes of cardinal numbers).
- A set with a well-order relation defines ordinal numbers.
- A set with a binary operation defines isomorphism classes of magmas.
- In general, it is possible to define all mathematical structures up to isomorphism.

Although the collections of such structures are usually not regarded as sets, in this system there is no reason to treat them specially. Inconsistencies like the Burali-Forti paradox can be avoided by prohibiting certain circular uses.

Thus, the objects obtained from the four different kinds of parameters correspond very well to the structures used in everyday mathematics. (Due to some details of the system beyond the scope of this paper, it is always possible to work with structures up to isomorphism only.)

Every instance of a construction is regarded as a separate type. In particular, natural numbers are separate from integers, which in turn are separate from rationals, reals, etc. At first glance, this seems problematic because in ordinary mathematics, every natural number is treated as an integer as well, and so on. Upon further inspection, the property that e.g. no natural number can be entered where an integer is expected (and vice versa) enables a novel 'embedding' feature. There is no need for strange hacks like in Mizar [18] or explicit casts like in HOL [10]; instead the user can simply specify an embedded set when defining a construction (see figure 12). Due to the aforementioned property that instances of different constructions would normally be separate, no ambiguities can arise. (In a more limited sense, the same feature would be possible in other type theories as well.)



Definition 1.4.1.1. We define:

$$\mathbb{N} := \left\{ \begin{matrix} 0 \\ \mathsf{s}(n) \end{matrix} \,\middle|\, n \in \mathbb{N} \right\}$$

$$\forall\, n \in \mathbb{N},\, n' \in \mathbb{N}: \ \mathsf{s}(n) = \mathsf{s}(n') \Leftrightarrow n = n'$$

Figure 10: Definition of natural numbers.

3.3 Consistency

The issue of consistency matters especially because the formal system does not obviously resemble any other existing system, and even admits some sets that are considered problematic in mathematics. For a proof assistant, it is desirable to achieve as much certainty as possible while being able to formalize essentially all existing mathematics. According to Gödel's second incompleteness theorem, the system cannot formalize its own consistency proof if it is consistent, so it is definitely impossible to achieve both goals at the same time (not that it would otherwise be obvious how to achieve absolute certainty).

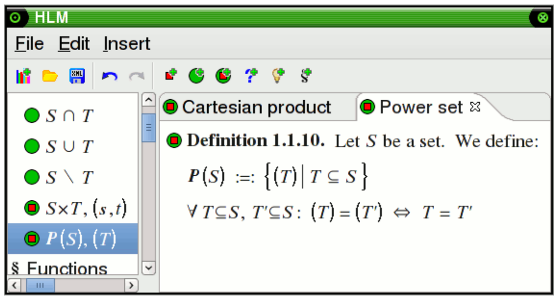
Nevertheless, a consistency proof of the system is possible within ordinary mathematics. It works by constructing a (meta-level) truth predicate for propositions and proving soundness with respect to this predicate, and can be formalized in Zermelo-Fraenkel set theory as well as some weaker systems. The most special property of the proof is that it requires arbitrarily-but-finitely many iterations of the power set operation, which currently cannot be formalized in the system because the rules concerning circularities are too strict. While it is certainly possible to modify them to permit this construction, for practical purposes it is sufficient to ignore the circularity rules in particular cases that are easily shown to be non-circular.

I believe that such an "unconditional" consistency proof provides slightly better security than one which relies on the consistency of another strong formal system, as it does not use any principles beyond everyday mathematics.

4 Conclusions and Future Work

I have presented a proof assistant and underlying formal system designed to capture mathematical practice as accurately as possible. Although it employs set-theoretic language, it accepts exactly the formulae and terms that would be considered meaningful in a given situation. This property significantly increases user-friendliness in the GUI-based proof assistant because the rules of the formal system map directly to the possible choices in graphical menus.

A crucial ingredient is that sets are built into the formal system in such



Definition 1.1.10. Let S be a set. We define:

$P(S)$:=: $\{(T) \mid T \subseteq S\}$

$\forall\, T \subseteq S,\ T' \subseteq S:\ (T) = (T') \Leftrightarrow T = T'$

Figure 11: Definition of power set.

a way that they can implicitly act as types. Two sets have the same type if they are syntactically declared to share a common superset. 'Constructions' introduce new types by defining sets along with their elements; they are general enough to define Cartesian products, functions, numbers, and all mathematical structures up to isomorphism.

The "up to isomorphism" part seems particularly intriguing because it provides a metamathematical characterization of 'isomorphisms' without having to encode structures as categories or even specify what 'morphisms' are. Moreover, since category theory ultimately deals with structures up to isomorphism, one might expect the system to be especially well-suited for categorial reasoning. For a start, defining the concept of a category is straightforward.

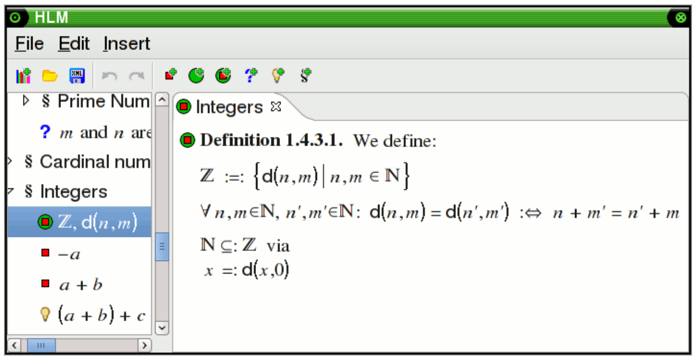
Alas, isomorphism classes of structures do not carry enough information to define morphisms in a meaningful way, so categories of all sets, all groups, etc. are not available. Although certain special categories can be defined, such as a category of ordinals or of all subsets of a given set, a more complete formalization of category theory will require some changes to the formal system (or to the definition of a category).

This would not be the only desirable extension: In the current system, one cannot even define the ordinal ε_0 given the construction of ordinals as described in section 3.2. Due to the non-circularity rules, a second construction of ordinals would be needed, so that a subset of the ordinals defined in the first construction could be used to define an ordinal in the second construction. Higher ordinals would require even more constructions, so that a mechanism of "merging" these constructions seems desirable. The same principles apply to categories of categories.

The system avoids the usual problems with undefinedness by allowing arbitrary constraints on definitions. For example, the simplest definition of the limit of a sequence includes the constraint that the sequence is actually convergent. However, in practice, the statement

$$\lim_{k \to \infty} a_k = x$$

is actually meant to *imply*, not *require*, that a is convergent. Of course, one can



Definition 1.4.3.1. We define:

$\mathbb{Z}$:=: $\{\mathrm{d}(n,m) \mid n,m \in \mathbb{N}\}$

$\forall\, n,m \in \mathbb{N},\, n',m' \in \mathbb{N}:\ \mathrm{d}(n,m) = \mathrm{d}(n',m') \ :\Leftrightarrow\ n + m' = n' + m$

$\mathbb{N} \subseteq: \mathbb{Z}$ via

x =: $\mathrm{d}(x,0)$

Figure 12: Definition of integers as superset of natural numbers.

simply write "a converges to x" instead, but it would be more convenient if the constraint in the definition of a limit could be marked "implied."

Moreover, I would like the formal system to be more modular, and in particular to accommodate both classical and constructive logic. The difference between these would also have a non-obvious impact on the user interface. For example, in the current classical setting, it is never necessary, and therefore not possible, to enter a doubly negated formula.

Concerning the user interface, the widest gap at the moment is the lack of automation even for very simple inferences. Most other proof assistants can search for proofs automatically at least in special cases (for example by providing advanced tactics), and this is usually necessary to avoid an explosion of extremely small inference steps. In a GUI, certain complex inferences can be input more easily (e.g. substitution at specific places according to given theorems), and certain recurring aspects (such as associativity and commutativity) can be dealt with on a higher level, mitigating the lack of automation in some cases. However, there remain a lot of small proofs, especially sub-proofs, that a human would consider too trivial to even mention. To automate these, a proof search should run in the background, and be updated whenever the user adds a proof step, as well as when a new sub-proof is opened. As soon as the current (sub-)proof can be finished automatically, the software should simply fill in the result, so the user can continue at the next step that requires manual interaction.

But above all, I would like the proof assistant to serve as a testing ground for all user interface features that can potentially make the input of definitions and theorems easier, faster, and more fun.

The current prototype is available at http://hlm.sourceforge.net/.

References

[1] Andrea Asperti, Claudio Sacerdoti Coen, Enrico Tassi, and Stefano Zacchiroli. User interaction with the Matita proof assistant. *Journal of Automated Reasoning*, 39(2):109–139, 2007.

[2] Janet Bertot and Yves Bertot. CtCoq: A system presentation. In *AMAST '96: Proceedings of the 5th International Conference on Algebraic Methodology and Software Technology*, volume 1101 of *Lecture Notes in Computer Science*, pages 600–603. Springer-Verlag, 1996.

[3] Thierry Coquand and Peter Dybjer. Inductive definitions and type theory: an introduction. In *Proceedings of the 14th Conference on Foundations of Software Technology and Theoretical Computer Science*, volume 880 of *Lecture Notes in Computer Science*, pages 60–76. Springer-Verlag, 1994.

[4] Pierre Corbineau. A declarative language for the coq proof assistant. In *TYPES '07: Proceedings of the 2007 international conference on Types for proofs and programs*, volume 4941 of *Lecture Notes in Computer Science*, pages 69–84. Springer-Verlag, 2008.

[5] N.G. de Bruijn. On the roles of types in mathematics. In P. de Groote, editor, *The Curry-Howard isomorphism*, pages 27–54, Louvain-la-Neuve, Belgium, 1995. Academia-Erasme.

[6] Herman Geuvers. Proof assistants: history, ideas and future. *Sadahana Journal*, 34:3–25, February 2009.

[7] Thomas C. Hales. Formal proof. *Notices of the AMS*, 55(11):1370–1380, December 2008.

[8] John Harrison. A Mizar mode for HOL. In Joakim von Wright, Jim Grundy, and John Harrison, editors, *TPHOLs '96: Proceedings of the 9th international conference on Theorem proving in higher order logics*, volume 1125 of *Lecture Notes in Computer Science*, pages 203–220. Springer-Verlag, 1996.

[9] John Harrison. Floating-point verification. In John Fitzgerald, Ian J. Hayes, and Andrzej Tarlecki, editors, *FM 2005: Formal Methods, International Symposium of Formal Methods Europe, Proceedings*, volume 3582 of *Lecture Notes in Computer Science*, pages 529–532. Springer-Verlag, 2005.

[10] John Harrison. HOL Light tutorial (for version 2.20). http://www.cl.cam.ac.uk/users/jrh/hol-light/tutorial_220.pdf, December 2007.

[11] John Harrison. Formal proof – theory and practice. *Notices of the AMS*, 55(11):1395–1406, December 2008.

[12] Steven Kieffer, Jeremy Avigad, and Harvey Friedman. A language for mathematical knowledge management. In A. Grabowski and A. Naumowicz, editors, *Computer Reconstruction of the Body of Mathematics*, volume 18 of *Studies in Logic, Grammar and Rhetoric*, pages 51–66, 2009.

[13] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, pages 207–220. ACM, October 2009.

[14] Agnieszka Kozubek and Paweł Urzyczyn. In the search of a naive type theory. In *TYPES '07: Proceedings of the 2007 international conference on Types for proofs and programs*, volume 4941 of *Lecture Notes in Computer Science*, pages 110–124. Springer-Verlag, 2008.

[15] Frank Pfenning and Christine Paulin-Mohring. Inductively defined types in the calculus of constructions. In *Proceedings of the 5th International Conference on Mathematical Foundations of Programming Semantics*, volume 442 of *Lecture Notes in Computer Science*, pages 209–228. Springer-Verlag, 1990.

[16] Jakub Sakowicz and Jacek Chrzaszcz. Papuq: a Coq assistant. In H. Geuvers and P. Courtieu, editors, *Proceedings of PATE'07*, pages 79–96, 2007.

[17] Markus Wenzel. Isar – a generic interpretative approach to readable formal proof documents. In Yves Bertot, Gilles Dowek, André Hirschowitz, C. Paulin, and Laurent Théry, editors, *TPHOLs '99: Proceedings of the 12th international conference on Theorem proving in higher order logics*, volume 1690 of *Lecture Notes in Computer Science*, pages 167–184. Springer-Verlag, 1999.

[18] Freek Wiedijk. Mizar: An impression. http://www.cs.ru.nl/~freek/mizar/mizarintro.pdf, 1999.

[19] Freek Wiedijk. A proposed syntax for binders in Mizar. http://www.cs.ru.nl/~freek/mizar/binder.pdf, 2003.

[20] Freek Wiedijk. Mizar's soft type system. In *TPHOLs '07: Proceedings of the 20th international conference on Theorem proving in higher order logics*, volume 4732 of *Lecture Notes in Computer Science*, pages 383–399. Springer-Verlag, 2007.

[21] Freek Wiedijk. The QED Manifesto revisited. *Studies in Logic, Grammar and Rhetoric*, 10(23):121–133, 2007.

[22] Freek Wiedijk. A synthesis of the procedural and declarative proof styles of interactive theorem proving. http://www.cs.ru.nl/~freek/miz3/miz3.pdf, 2010.

[23] Sean Wilson, Jacques Fleuriot, and Alan Smaill. Inductive proof automation for Coq. In Yves Bertot, editor, *Second Coq Workshop*, 2010.