High-Level Theories*

Jacques Carette and William M. Farmer **

Department of Computing and Software
McMaster University
Hamilton, Ontario, Canada

29 April 2008

Abstract. We introduce high-level theories in analogy with high-level programming languages. The basic point is that even though one can define many theories via simple, low-level axiomatizations, that is neither an effective nor a comfortable way to work with such theories. We present an approach which is closer to what users of mathematics employ, while still being based on formal structures.

1 Introduction

The mission of *mechanized mathematics* is to develop software systems that support the process people use to create, explore, connect, and apply mathematics. There are historically two main kinds of *mechanized mathematics systems (MMSs)*: *theorem proving systems (TPSs)* and *computer algebra system (CASs)*. Both kinds of systems encapsulate a body of mathematical knowledge and a collection of tools for using this knowledge. The tools of TPSs tend to be primarily deductive, while those of CASs tend to be primarily computational.

MMSs have two kinds of users. *End users* use the tools of an MMS to help them do mathematics, whatever that involves. *Developers* use the tools of an MMS to produce new mathematical knowledge and new tools to facilitate the work of end users. Developers need to have a deep understanding of the logical and mathematical foundation of their MMS of choice. They are interested in the structure of mathematics, the problems involved in formalizing mathematics, and the MMS. End users need much less depth in their understanding of the MMS. They are computer scientists, engineers, and other scientists who are primarily interested in an MMS as a (mathematical) tool to solve a problem.

Mathematics is usually done in informal high-level reasoning environments that include a rich set of concepts and practical tools. The tools

 Published in S. Autexier, J. Campbell, J. Rubio, V. Sorge, M. Suzuki, and F. Wiedijk, eds., *Intelligent Computer Mathematics, LNCS*, 5144:232–245, 2008. This research was supported by NSERC.

** {carette,wmfarmer}@mcmaster.ca.

involve a mixture of computation and deduction and are highly integrated with each other. While the setting appears informal, enough rigor is applied that, in theory, the results *could* be made formal. MMS end users want to work in similar high-level environments, and MMS developers want to build such environments. But contemporary MMSs do not provide such environments, nor do they provide the tools to build them. We would like to provide something akin to the ease-of-use of *Theorema* [4] with the computational correctness and efficiency provided by *Focal* [27] and the soundness of Coq [9].

Contemporary TPSs provide low-level reasoning environments based on axiomatic theories. An *axiomatic theory* consists of a set of formulas (called *axioms*) in a formal language. An axiomatic theory encodes a body of mathematical knowledge declaratively; the "truths" of the body of knowledge are the logical consequences of the axioms. In a TPS, new concepts are expressed by writing definitions, and the consequences of the concepts are explored by stating and proving conjectures. Making definitions and proving theorems is the primary emphasis; performing computations and introducing derived reasoning rules is secondary. As a result, doing mathematics in a TPS requires working at a very low conceptual level—much like programming in an assembly language. TPSs give developers the access they need to low-level details, but generally not the capability to build the kind of high-level reasoning tools that end users want. As a result, TPSs are almost useless for end users who are interested in "getting work done" and lack the necessary understanding of mathematics at the level of axiomatic theories.

Contemporary CASs provide high-level computational environments based on algorithmic theories. An *algorithmic theory* consists of a set of algorithms that perform symbolic computations over a formal language. An algorithmic theory encodes a body of mathematical knowledge procedurally; the "truths" of the body of knowledge are the results that can be obtained by running the algorithms on the range of inputs. In a CAS, new algorithms can be expressed in the theory by writing programs, usually in a special, system-supplied programming language. Reasoning is narrowly focused on computation; there is usually little support for deductive reasoning. Moreover, not only is the background theory of the algorithms largely hidden from the user, these background theories are (unfortunately) inconsistent from algorithm to algorithm, which makes the results obtained from such computations frequently untrustworthy. As a result, doing mathematics in a CAS is like programming in a high-level programming language with an inaccessible, untrustworthy compiler. CASs offer

end users a high-level reasoning environment, but one that supports only computation and provides untrustworthy results. By not giving developers access to their logical foundations, CASs have very limited use for developers of mechanized mathematics.

We argue in this paper that, in order for mechanized mathematics to achieve its potential, MMSs must provide end users with high-level environments for reasoning and computation, similar to the informal environments they are used to, in which they can work in a sound and convenient fashion. MMSs must also provide developers with the capabilities to build high-level environments for end users that are derived from a solid logical foundation. Toward this goal, we introduce the notion of a *high-level theory*, a semi-formal high-level environment for reasoning and computation that is analogous to a high-level programming language.

The paper is organized as follows. Section 2 discusses what a high-level reasoning environment is. Section 3 introduces the notion of a high-level theory and explores the analogy between high-level theories and high-level programming languages. How mathematics is performed within a high-level theory is the subject of section 4. How high-level theories are created and connected is covered in section 5. Two examples of high-level theories are briefly discussed in section 6. We comment in section 7 on some related issues and conclude in section 8 with some remarks and a recommendation that MMS builders design and implement systems that offer high-level theories instead of just axiomatic or algorithmic theories.

2 High-Level Reasoning Environments

As we have mentioned above, mathematics practitioners work in high-level reasoning environments that offer integrated sets of concepts and deductive, computational, visual, and other kinds of tools. Working in them is more convenient and practical than working in an axiomatic theory or even in a network of axiomatic theories.

For example, consider the informal reasoning environment of *natural number arithmetic* (which is also called *number theory*). Even though an axiomatization of natural number arithmetic is relatively simple, the informal environment that people actually work in is quite sophisticated. For instance, it includes a set of algebraic operators, a linear order, several lattice structures, a collection of induction principles, a collection of algorithms for adding, multiplying, dividing (with remainder), etc., and various connections to set theory, analysis, and abstract algebra. An axiomatization of natural number arithmetic—even one augmented with

```
Theory Nat:
  concepts 0,1 : Nat.
  transformers +, * : Nat -> Nat -> Nat. total, commutative, associative.
  transformers <, = : Nat -> Nat -> Bool.
  theorem: Nat is an ordered semi-ring.
  language AE(Nat) = {0,1,+,*}         // Arithmetic Expressions over Nat
  derive transformer: eval : AE -> Nat             // Evaluation, derived
  theorem: total(eval).
  derive transformer induction : Prop(AE) -> InductiveProof
  ...                                   // Structural induction principle
```

Fig. 1. `Nat` as a sample high-level theory

many definitions and theorems—is an enfeebled reasoning environment in comparison to this standard informal high-level environment. Figure 1 gives a taste of what we mean. It is important to notice that the "signature" of `Nat` does not export the implementation details of the representation of `Nat`. We are thus free to provide "implementations" via Peano's axioms or via some other (more efficient) means.

Another good example of an informal high-level environment is what is often called *group theory*. It includes the basic definitions of the algebraic structure called a group, machinery connecting groups via homomorphisms, tools such as the Sylow theorems and the orbit-stabilizer theorem for analyzing the structure of groups, and standard applications of groups to various symmetric structures and problems in mathematics. Group theory cannot be naturally derived from an axiomatization of a group. It is based on a set of axiomatic theories that includes a theory of a single group, several copies of this theory for homomorphisms, a theory of a group action, a theory of natural number arithmetic, and a theory of sets and functions. In common use, group theory is more about its connections to other theories than about groups themselves.[1]

Reasoning in one of these high-level environments is analogous to programming in a high-level programming language like Java or ML. The reasoning *can* be reduced to the level of axiomatic theories, but this is rarely necessary or even desirable.

3 High-Level Theories

So what exactly are high-level theories? Informally, they should be to mechanized mathematics what high-level programming languages are to

[1] There are exceptions, naturally.

1. Convenient, human-oriented, sound, and precise.
2. Supports deduction, computation, and mixtures of the two.
3. Allows the end user to work at a high conceptual level.
4. Includes a well-constructed, highly integrated set of tools.
5. Constructed modularly.
6. Efficiently implemented with respect to resources.
7. Enables multiple modes of interaction (e.g., graphical).

Fig. 2. The pragmatic properties of a high-level theory

programming. This analogy is quite rich, and deserves to be expanded upon. But first, we will explicitly list in Figure 2 the pragmatic properties that we want a high-level theory to have. While these might all sound quite desirable, each is a nontrivial constraint. Furthermore, if they are not *designed into* a system, they are rather unlikely to be emergent properties of an implementation.

3.1 The high-level programming analogy

Taking a wide, top-down view of high-level programming languages, we first encounter programming *paradigms*, namely procedural, functional, object-oriented, and logical/relational. Mathematicians also have styles. Some like to prove purely existential theorems, others are engaged in giant computations, whilst others like to find relations between various theories; there are entire books and active conferences dedicated to studying these topics. While our goal is to support as many of these activities as possible, we will focus on deduction and computation.

If we look at most modern high-level programming languages, we get a large set of features, even though we know that we could achieve much the same by directly programming in an assembly language (or a Turing machine or the λ-calculus). The same reasons that drove programming languages to include such high-level features should guide our search. Furthermore we should not look merely at language advances or even programming systems (languages combined with a standard library). It is important to realize that some languages thrive because they inhabit a complete ecosystem, with rich IDEs offering non-ASCII based methods of interaction, project management features, etc. It is sobering to remember that N. G. de Bruijn had foreseen some of this 40 years ago [12].

At the simplest level, we want to combine oft-used chains of primitive deductions into new transformers which are meant to be used as units.

For example, loops are so pervasive in programming that all languages offer high-level constructs for this, with a range of semantics. This varies from the one-size-fits-all `while` loop of early imperative languages to the semantically richer `foreach` loop, and the even richer `fmap` and `foldl` (from Haskell). In an MMS, once we have theorems that prove the correctness of algorithms for addition and multiplication over `Nat`, we should simply add these algorithms as new "fundamental" tools.

Between the two extremes of programming paradigm and low-level primitives, programming languages offer further tools, for "programming in the large", like classes, modules, or functors. The analogy extends: we want structuring mechanisms for our MMSs. We favor using *theories* and *parameterized theories* for that purpose. And even though our aim is to present to users rich high-level theories, we still firmly believe in the *little theories method* [19]. These can be assembled in a principled and modular fashion, and implemented atop a module system like Mei [29, 30].

Perhaps the biggest difference is that a high-level theory needs to support more than just computation, it also needs to support deductive reasoning (and vice versa). These activities should not just co-exist: they should be tightly integrated with each other as they are in mathematical practice. Furthermore, reasoning and computation should not be restricted to objects of a particular theory: they should be applicable to theories and their interconnections [7].

3.2 Informal definition

In this subsection, we give a preliminary, informal definition of a high-level theory, while in the next section we show how to effectively make this definition precise. In other words, we give an abstract specification now and then an implementation later. The aims of this subsection is to convey the intuition behind our ideas.

Definition 1 *A* high-level theory (HLT) *is a tuple* $(\mathcal{C}, \mathcal{T})$ *of* concepts *and* transformers *that possesses the pragmatic properties given in Figure 2. Concepts* $\mathcal{C}$ *are the basic objects of discourse and transformers* $\mathcal{T}$ *are* n*-ary functions on expressions concerning concepts.*

Implicit in the above definition is the notion of a *language* and a *base theory* over which everything is defined, which enters more directly in the next definition.

A *concept* is a pair (s, d) of a new symbol s and a definition d for s. In other words, a concept consists of a name and its meaning—defined in

a formal language over a theory. This can be a basic object like 0 or 1, a particular group G, the definition of the fundamental group π_1 of a surface, the Gaussian elimination algorithm, an algorithm for integer factorization, the set theory NBG, or an abstract 2-category. The definition d of a concept can be given implicitly as a certain set of properties or explicitly as an expression that denotes an object.

A *transformer* is a function that maps expressions to expressions. It can embody computations or deductions. So a transformer can be as simple as the modus ponens deduction rule or integer arithmetic, up through induction or Gröbner basis computations, to proof rules for applying tactics, or an algorithm for solving PDEs symbolically. It is important to note that all these are maps from some pieces of syntax to other pieces of syntax, although our chief interest in all of them is what that syntax denotes. This point is worth emphasizing because this is a principal difference between theorem proving systems and computer algebra systems: both implement transformers, but they differ greatly in what *meaning* is a priori given to each transformer.

Another characteristic of mathematics which is important to model is the pervasive use of conceptual layers and abstraction. Concepts often appear in various guises: for example, (computable) functions can both be used directly as transformers and can also be a concept of study. Directed graphs with labeled edges and nodes can be studied directly or can be used to conveniently represent other concepts like commutative diagrams, which themselves represent equations in a theory. An HLT should give us the tools to draw a commutative diagram (as a labeled graph) and have the MMS properly interpret the result.

3.3 Semi-formal definition

In the previous subsection we defined an HLT to be a collection of concepts and transformers that satisfy certain pragmatic requirements. Concepts are names representing mathematical ideas and objects, while transformers are functions mapping expressions to expressions that represent deduction and computation rules. In this section we will introduce a semi-formalization of an HLT, based on the notion of a biform theory [16, 20].

We will begin by formalizing the notion of a transformer. Fix a set $\mathcal{E}$ of *expressions* that includes a set of *formulas*. For $n \geq 0$, an n-ary *transformer for* $\mathcal{E}$ is $\Pi = (\pi, \hat{\pi})$ where π is a symbol and $\hat{\pi}$ is an algorithm that implements a (possibly partial) function $f_{\hat{\pi}} : \mathcal{E}^n \to \mathcal{E}$. The symbol π serves as a name for the algorithm $\hat{\pi}$. There is no restriction on how the

algorithm is presented. For example, it could be a lambda-expression in $\mathcal{E}$ or a program written in a high-level programming language.

Definition 2 *A* biform theory T *is a triple* $(\mathcal{E}, \mathcal{T}, \Gamma)$ *where* $\mathcal{E}$ *is a set of expressions,* $\mathcal{T}$ *is a set of transformers for* $\mathcal{E}$*, and* Γ *is a set of formulas in* $\mathcal{E}$*.*

The set $\mathcal{E}$ is generated from a set of symbols. Each symbol is either the name of a concept of T or is the name of a transformer of T. The members of Γ are the *axioms* of T. They specify the meanings of the concepts and transformers of T. Implicit in the above definition is the notion of a *background logic* that provides a semantic foundation for the meaning of a biform theory. We may define a biform theory $T = (\mathcal{E}, \mathcal{T}, \Gamma)$ to be an *axiomatic theory* if $\mathcal{T}$ is empty and an *algorithmic theory* if Γ is empty. Thus a biform theory is a generalization of both an axiomatic theory and an algorithmic theory.

A *rule for* $\mathcal{E}$ is a formula A in $\mathcal{E}$ of form

$$\forall\, e_1 : \mathcal{E}, \ldots, e_m : \mathcal{E}\ .\ B$$

where B contains one or more occurrences of an expression of the form $\pi(a_1, \ldots, a_n)$, which represents an application of the algorithm $\hat{\pi}$ to n expressions denoted by $a_1, \ldots, a_n$. The application of A to an input list $E_1, \ldots, E_m$ of expressions in $\mathcal{E}$ is the formula A' obtained by replacing each occurrence of the form $\pi(E'_1, \ldots, E'_n)$ in

$$B[e_1 \mapsto E_1, \ldots e_m \mapsto E_m]$$

with the result of applying $\hat{\pi}$ to the list $E'_1, \ldots, E'_n$ of expressions. A rule can be a rule of computation, deduction, or a mixture of the two.

Declaratively, a rule is a formula that specifies the set of transformers whose names occur in the formula. Functionally, a rule maps a list of expressions to a formula that relates the expressions as inputs to the expressions that are produced as outputs.

Definition 3 *A* high-level theory (HLT) *is a biform theory* $T = (\mathcal{E}, \mathcal{T}, \Gamma)$ *such that* Γ *includes a set of rules that have the pragmatic attributes listed in Figure 2.*

Since rules are statements about both the syntax of expressions and what the expressions mean, nontrivial biform theories are not easy to

formalize in traditional logics such as first-order logic or simple type theory [16]. A logic is needed in which reasoning about the syntax of expressions (normally performed outside the logic) is integrated with reasoning about the semantics of expressions (normally performed in the logic itself). We have proposed a logic of this kind named *Chiron* [17, 18] that we believe is exceptionally well-suited for formalizing biform theories. A derivative of von-Neumann-Bernays-Gödel (NBG) set theory, Chiron supports several reasoning paradigms by integrating set theory with elements of type theory, a scheme for handling undefinedness, and a facility for reasoning about the syntax of expressions.

3.4 Extended example

To give a taste of what we would like to do, we present a "simple" example of what we can express in this setting.

Suppose that we define the concept *commutative* as the property $\forall\, a, b \,.\, a \circ b \simeq b \circ a$ of a (possibly) partial binary operation $\circ : A \times A \rightharpoonup A$. As expected, commutativity is a property of binary operators; less usual is the generalization to the partial setting.

Now consider the theory of Abelian groups. We would obtain a rather unusual theory if we used the above definition of commutativity. Yet, that is exactly the definition that we wish to use. Will we then need to rebuild all of group theory for a generalization that does not seem important? No, as we also have the following property:

Proposition 31 *Let $* : A \times A \to A$ be a total binary operation. Then $*$ is commutative if and only if $\forall\, a, b : A \,.\, a * b = b * a$.*

This property is really an immediate corollary of a more fundamental equivalence, namely

$$\forall\, a, b \in \mathcal{E} \,.\, (a\!\downarrow \wedge b\!\downarrow) \to (a \simeq b \leftrightarrow a = b)\,.$$

($a\!\downarrow$ means a is defined.)

We recall that a binary operation $*$ is total if and only if $\forall\, a, b : A \,.\, (a * b)\!\downarrow$. Combining this with the property above allows us to create a *theory level* transformer which (in Chiron with syntactic sugar) reads

$$\lambda\, e : \mathcal{E} \,.\, \texttt{if}(\texttt{match}(e, e_1 \simeq e_2) \wedge [\![e_1]\!]\!\downarrow \wedge [\![e_2]\!]\!\downarrow, e_1 = e_2, e_1 \simeq e_2).$$

In other words, when we instantiate the concept of commutativity in the process of creating the theory of Abelian groups, our theory building op-

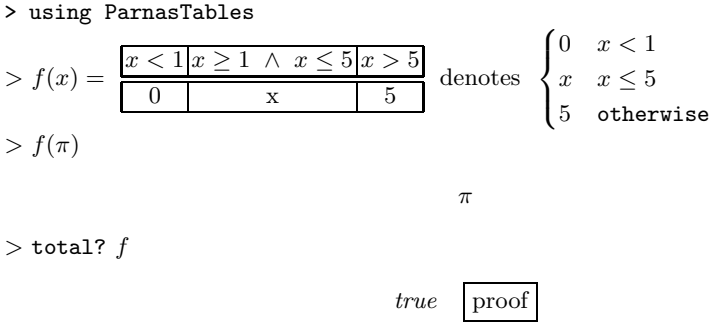

Fig. 3. (Mock up of) working in an HLT of Parnas tables

erators can use the above as a *simplification* rule[2]. This rule belongs in the HLT dedicated to building theories, which is a computation on the syntactic representation of theories but relies on intermediate deductions (and computations) for its proper application. Another aspect is to note the different quantifications we are using above—universal for the definition of commutativity, over expressions for the equivalence of $\simeq$ and $=$, and over values for the definition of total.

4 Exploring High-Level Theories

The ultimate purpose of an HLT is to provide a convenient environment for end users to formulate mathematical problems and to explore possible solutions using deduction, computation, and other techniques such as visualization. An HLT is intended to be self-contained in the sense that everything the end user needs to reason within the HLT is available in the HLT. The end user should not have to introduce many new concepts, construct many new transformers, or look for many results in other HLTs. Not only does the end user have the tools he or she needs, the tools are designed to work together efficiently.

Since the tools of an HLT should usually involve both deduction and computation, an HLT-based MMS should provide a *derivation facility* for proving conjectures and performing computation in which proving and computing are mixed. For example, in derivation a computation can involve the proof of side conditions and a proof can involve the computation of expressions. Figure 3 shows an idealized end user interaction with

[2] It is important to note that we are implicitly using deep inference and inference "in context" in the statement of this rule.

the HLT of Parnas tables [21] (as a particularly useful visual metaphor for piecewise functions [6]). A notable "feature" is the first-class proof object in the last interaction, while we retain the dubious feature of Parnas tables using only first-order logic to specify how to partition the domain of f. Note the difference with Figure 1 which gives the developer's view of the definition of an HLT, whereas Figure 3 is the end user's view. The vocabulary is the same, but an end user will typically interact via (the names of) transformers on expressions and see their results pretty-printed, while the developer gets a more theory-centric view.

5 Creating and Connecting High-Level Theories

The job of an MMS developer is to create HLTs. There are several approaches for doing this. First, an HLT can be created from scratch. This will be difficult and labor intensive if the subject matter of the HLT is complex or unfamiliar. On the other hand, this could be feasible if the HLT mirrors a well-understood and well-tested high-level reasoning environment such as elementary calculus. In either case, there is a significant danger that the concepts and transformers of the HLT may be inconsistent with each other. Another danger is that it could be extremely difficult to connect an HLT developed from scratch to another HLT so that results shared between the HLTs can be trusted, as witnessed by CASs.

A second approach is to construct an HLT incrementally starting from a set of very low-level axiomatic theories. Using concept and transformer definition techniques as well as module building techniques such as extension, union, renaming, and parameter instantiation [30], a network of interconnected biform theories can be build on top of the starting set of axiomatic theories. As one of the biform theories in the network, the HLT has a modular construction that is recorded in the structure of the theory network. The HLT is thus derived in a structured fashion from its underlying set of axiomatic theories.

The concepts and transformers of the HLT can be viewed as its *interface* and the theory network that records its construction can be viewed as its *implementation*. Just like the interface of a software module, the interface should not include everything in the implementation. Many low-level tools that are needed by developers to construct an HLT are either of no use to the end user or are subsumed by the high-level tools in the HLT's interface. Moreover, it is certainly possible that the same HLT can be derived from several different sets of axiomatic theories. That is, an HLT can have more than one implementation.

The implementation of an HLT is crucial for connecting one HLT to another HLT. Suppose we would like to connect an HLT T_1 to an HLT T_2. Let us assume that T_i is derived from a set S_i of axiomatic theories for $i = 1, 2$. We first construct translations from the axiomatic theories of S_1 to the axiomatic theories of S_2. Next we show that these translations are meaning preserving, i.e., are *theory interpretations* [14, 15]. The last step is to use the constructions of T_1 and T_2 as a guide to lift and merge these axiomatic theory interpretations to a theory interpretation of T_1 in T_2. This last step would ideally be performed automatically. This has very natural categorical semantics in terms of limits of diagrams (in the category of biform theories).

A third approach is to construct an HLT, or least part of an HLT, automatically from an axiomatic theory. The work by R. McCasland on mechanical theorem discovery is an interesting step in this direction [24]. Two classical examples are the use of the Knuth-Bendix completion algorithm to automatically generate a terminating term rewrite system from a set of equational axioms [23] and the use of Buchberger's algorithm to construct a Gröbner basis for a system of polynomials [3]. For the algorithmic aspects, one can instantiate generic algorithms and still get efficient implementations [5, 8].

We will finish this section with an important observation. Mathematical knowledge as a whole is a network of interconnected smaller bodies of mathematical knowledge. The interconnectivity of mathematics allows problems to be expressed and solved in a general context (e.g., metric spaces) and then their solutions to be applied in more specialized contexts (e.g., real analysis). The *little theories method* [19] models the network of interconnected bodies of mathematical knowledge as a network of separate, but interconnected axiomatic theories. The *big theory method* [19] models mathematical knowledge as one big theory. The little theories method is central in the creation and connection of HLTs, but the big theory method is followed in the exploration of HLTs.

6 Examples of High-Level Theories

We further examine how we might formalize the two examples of high-level reasoning environments given in section 2.

An HLT formalization of natural number arithmetic would be a biform theory T_1 containing several hundred concepts, transformers, and axioms. It would be carefully constructed in two ways.

First, T_1 would be constructed in a modular fashion from a well-understood, low-level axiomatization of natural number arithmetic, like Peano's axioms, and a set of supporting low-level theories about such things as sets and real numbers. Its construction would demonstrate that each of its axioms is a theorem derived from the low-level theories. How the low-level theories are axiomatized is not important as long as the axioms of T_1 can be derived from the theories.

Second, the concepts, transformers, and axioms of T_1 would be carefully chosen. They would not include every known concept, transformer, and theorem of natural number arithmetic. In particular, T_1 would not be simply the sum of the low-level theories from which it is constructed. Instead the constituents of T_1 would have a high level of coverage and a low level of redundancy. For example, the concepts of T_1 might well not include the successor function since it can be easily expressed using the addition function. And there would be no need for the recursive definitions of addition and multiplication if T_1 includes transformers for computing sums and products. The axioms of T_1 would be high-level theorems such as the fundamental theorem of arithmetic and the Chinese remainder theorem, high-level deduction rules such as various induction principles, and high-level computation rules such as those for computing greatest common divisors and factoring natural numbers into primes.

To end users, T_1 would look like a formalization of what mathematicians call "number theory"; the concepts, transformers, and axioms of T_1 would be basic ideas, tools, and assumptions of the theory. To developers, T_1 would look like the end result of a large, complicated theory development; the concepts, transformers, and axioms of T_1 would be the high-level ideas, tools, and theorems derived from the underlying low-level theories.

An HLT formalization of group theory would be a biform theory T_2 that contains, like T_1, several hundred concepts, transformers, and axioms, and constructed in the same careful manner to embody "group theory". In particular, "group theory" does not care whether an inverse function is provided axiomatically or as a derived property.

7 Related Work

As we have already mentioned, the working environment of a mainstream CAS (like Maple and Mathematica) gives the impression of working in a HLT, but in reality only achieves properties (3) and (7) (of those in

Figure 2), and somewhat implements (5) and (6). Theorema [4] adds (2). But since they are unsound, it is unclear if that all amounts to much.

Current large TPSs (like Coq [9], Isabelle [26], and PVS [25]) also seem to be moving in this direction, with their strength being properties (4) and frequently (5), and slowly moving in the direction of (2). Mizar [28] is hobbled by being nonmodular. There is a lot of work being done to integrate computation into deduction [1, 2, 13]. This work will certainly have an effect on ours, but we feel that it is too asymmetric compared to the symmetry between computation and deduction in biform theories.

The most direct implementation of something akin to HLTs is in Focal [27]. Unfortunately, this system is only really comfortable for dedicated developers and does not yet enable multiple modes of interaction.

It is possible to build a safe computational system atop a theorem prover, as Kaliszyk and Wiedijk show [22]. While a definite achievement, this seems to embody (part of) one handbuilt HLT.

Lastly, we should note that it is possible to encode biform theories in the Calculus of Inductive Constructions [10, 11], and thus in Coq. It is however our current feeling that Chiron is better suited for this task.

8 Conclusion

A high-level theory (HLT) is a model of the high-level reasoning environments employed in mathematical practice. Roughly speaking, it consists of a well-crafted set of concepts and transformers. More precisely, it is a biform theory with certain pragmatic properties. In particular, it includes high-level tools for deduction, computation, and a mixture of the two. An HLT is to a low-level axiomatic theory or algorithmic theory as a high-level programming language is to an assembly language. Working in an HLT is much more effective and convenient than working in a low-level theory.

We recommend that the ultimate goal of an MMS should be to provide a library of HLTs that are useful and accessible to a wide range of mathematics practitioners. The library's HLTs should include the best of the features currently found in the axiomatic and algorithmic theories of contemporary TPSs and CASs. Low-level axiomatic and algorithmic theories should be considered as part of the supporting infrastructure of the library, not as the end product of the system. To facilitate the development and expansion of the library, the MMS should include a facility with a powerful set of tools for developers to construct HLTs from low-level theories. We believe that an MMS that offers HLTs to end users and

the tools for building HLTs to developers has the best chance of realizing the immense potential of mechanized mathematics.

References

1. F. Blanqui, J.-P. Jouannaud, and P.-Y. Strub. Building decision procedures in the calculus of inductive constructions. In J. Duparc and T. A. Henzinger, editors, *CSL*, volume 4646 of *Lecture Notes in Computer Science*, pages 328–342. Springer, 2007.
2. F. Blanqui, J.-P. Jouannaud, and P.-Y. Strub. From formal proofs to mathematical proofs: A safe, incremental way for building in first-order decision procedures. In *TCS 2008: 5th IFIP International Conference on Theoretical Computer Science*. Springer-Verlag, 2008.
3. B. Buchberger. Theoretical basis for the reduction of polynomials to canonical forms. *SIGSAM Bulletin*, 39:19–24, 1976.
4. B. Buchberger, A. Craciun, T. Jebelean, L. Kovacs, T. Kutsia, K. Nakagawa, F. Piroi, N. Popov, J. Robu, M. Rosenkranz, and W. Windsteiger. Theorema: Towards computer-aided mathematical theory exploration. *Journal of Applied Logic*, 4:470–504, 2006.
5. J. Carette. Gaussian Elimination: a case study in efficient genericity with MetaOCaml. *Science of Computer Programming*, 62(1):3–24, 2006. Special Issue on the First MetaOCaml Workshop 2004.
6. J. Carette. A canonical form for piecewise defined functions. In *Proceedings of the 2007 International Symposium on Symbolic and Algebraic Computation (ISSAC)*, pages 77–84, New York, NY, USA, 2007. ACM Press.
7. J. Carette, W. M. Farmer, and V. Sorge. A rational reconstruction of a system for experimental mathematics. In *Proceedings of MKM/Calculemus 2007*, volume 4573 of *LNCS*, pages 13–26. Springer Verlag, 2007.
8. J. Carette and O. Kiselyov. Multi-stage programming with Functors and Monads: eliminating abstraction overhead from generic code. Accepted. Special issue for GPCE '04 and '05, 2008.
9. Coq Development Team. *The Coq Proof Assistant Reference Manual, Version 7.4*, 2003. Available at `http://pauillac.inria.fr/coq/doc/main.html`.
10. T. Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 76:95–120, 1988.
11. T. Coquand and C. Paulin-Mohring. Inductively defined types. In P. Martin Löf and G. Mints, editors, *COLOG-88: Proceedings of the International Conference on computer Logic*, volume 417 of *Lecture Notes in Computer Science*, pages 50–66. Springer-Verlag, 1990.
12. N. G. de Bruijn. Automath, a language for mathematics. In J. Siekmann and G. Wrightson, editors, *Automation of Reasoning 2: Classical Papers on Computational Logic 1967-1970*, pages 159–200. Springer, Berlin, Heidelberg, 1983.
13. G. Dowek, T. Hardin, and C. Kirchner. Theorem proving modulo. *J. Autom. Reasoning*, 31(1):33–72, 2003.
14. H. B. Enderton. *A Mathematical Introduction to Logic*. Academic Press, second edition, 2000.
15. W. M. Farmer. Theory interpretation in simple type theory. In J. Heering et al., editor, *Higher-Order Algebra, Logic, and Term Rewriting*, volume 816 of *Lecture Notes in Computer Science*, pages 96–123. Springer-Verlag, 1994.

16. W. M. Farmer. Biform theories in Chiron. In M. Kauers, M. Kerber, R. R. Miner, and W. Windsteiger, editors, *Towards Mechanized Mathematical Assistants*, volume 4573 of *Lecture Notes in Computer Science*, pages 66–79. Springer-Verlag, 2007.
17. W. M. Farmer. Chiron: A multi-paradigm logic. In R. Matuszewski and A. Zalewska, editors, *From Insight to Proof: Festschrift in Honour of Andrzej Trybulec*, volume 10(23) of *Studies in Logic, Grammar and Rhetoric*, pages 1–19. University of Białystok, 2007.
18. W. M. Farmer. Chiron: A set theory with types, undefinedness, quotation, and evaluation. SQRL Report No. 38, McMaster University, 2007. Revised 2008.
19. W. M. Farmer, J. D. Guttman, and F. J. Thayer. Little theories. In D. Kapur, editor, *Automated Deduction—CADE-11*, volume 607 of *Lecture Notes in Computer Science*, pages 567–581. Springer-Verlag, 1992.
20. W. M. Farmer and M. von Mohrenschildt. An overview of a formal framework for managing mathematics. *Annals of Mathematics and Artificial Intelligence*, 38:165–191, 2003.
21. R. Janicki, D. L. Parnas, and J. Zucker. Tabular representations in relational documents. In C. Brink, W. Kahl, and G. Schmidt, editors, *Relational Methods in Computer Science*, pages 184–196. Springer-Verlag, 1997.
22. C. Kaliszyk and F. Wiedijk. Certified computer algebra on top of an interactive theorem prover. In *Calculemus/MKM*, pages 94–105, 2007.
23. D. E. Knuth and P. B. Bendix. Simple word problems in universal algebra. In J. Leech, editor, *Computational Problems in Abstract Algebra*, pages 263–297. Pergamon Press, 1970.
24. R. L. McCasland, A. Bundy, and P. F. Smith. Ascertaining mathematical theorems. *Electronic Notes in Theoretical Computer Science*, 151:21–38, 2006.
25. S. Owre, S. Rajan, J. M. Rushby, N. Shankar, and M. Srivas. PVS: Combining specification, proof checking, and model checking. In R. Alur and T. A. Henzinger, editors, *Computer Aided Verification: 8th International Conference, CAV '96*, volume 1102 of *Lecture Notes in Computer Science*, pages 411–414. Springer-Verlag, 1996.
26. L. C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.
27. V. Prevosto. Certified mathematical hierarchies: The FoCal system. In Thierry Coquand, Henri Lombardi, and Marie-Françoise Roy, editors, *Mathematics, Algorithms, Proofs*, number 05021 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2005. Internationales Begegnungs- und Forschungszentrum (IBFI), Schloss Dagstuhl, Germany.
28. P. Rudnicki. An overview of the MIZAR project. Technical report, Department of Computing Science, University of Alberta, 1992.
29. J. Xu. Mei — A module system for mechanized mathematics systems. In *Programming Languages for Mechanized Mathematics Workshop*, Hagenberg, Austria, 2007.
30. J. Xu. *Mei — A Module System for Mechanized Mathematics Systems*. PhD thesis, McMaster University, January 2008.