Representation of mathematical objects in interactive books [1]

André M.A. van Leeuwen
CWI, Dept. AM
Kruislaan 413
1098 SJ Amsterdam
The Netherlands
e-mail: leeuw@cwi.nl

Abstract

We present a model for the representation of mathematical objects in structured electronic documents, in a way that allows for interaction with applications such as computer algebra systems and proof checkers. Using a representation that reflects only the intrinsic information of an object, and storing application-dependent information in so-called *application descriptions*, it is shown how the translation from the internal to an external representation and *vice versa* can be achieved. Hereby a formalisation of the concept of *context* is introduced. The proposed scheme allows for a high degree of application integration, e.g., parallel evaluation of subexpressions (by different computer algebra systems), or a proof checker using a computer algebra system to verify an equation involving a symbolic computation.

[1]This work is part of the MathViews project, supported by SION (The Netherlands Foundation for Computer Science Research), project number 612-317-410. MathViews is embedded in the ACELA project, which (amongst others) aims at producing an interactive book on the topic of Lie Algebras.

1 Introduction

Nowadays powerful workstations with high resolution graphical displays are quite common. This, and the continuing development of high level mathematical software such as computer algebra systems and proof checkers, have led us to the idea of developing a new (instructional) tool, the *interactive book*[2].

An interactive book combines the functionality of introducing mathematics to the reader with offering him a playground to test and explore his understanding of the topics just learned. As opposed to reading an ordinary paper copy of a book, an interactive book offers the reader an environment in which he can actively manipulate parts of the book. This manipulation is not limited to actions the author has catered for, nor is it restricted to the use of a fixed set of applications. The reader is free to experiment with the material offered and is allowed, and indeed encouraged, to use whatever functionality is present. This functionality consists of the combined functionality offered by the set of applications that are connected to the system.

The natural hypertext structure of mathematical knowledge will be reflected in the structure of interactive books, allowing the reader to travel through a book in several directions. Thus, the reader can develop insight into the nature of mathematics by seeing how things fit into each other and by noticing the interdependency of matters.

We feel that this approach is more promising than building upon existing paradigms like Mathematica Notebooks; an introduction to notebooks can be found in [Fultz93]. A more extensive description of interactive books can be found in [Cohen94].

The following list of examples give us an idea of the possibilities offered by interactive books:

- The ability to perform any kind of meaningful action on elements of the book, such as evaluating an expression, plotting a graph of a function, checking a step in a proof.
- Adaptive level of treatment. For example, let the reader decide to which extent the steps of a proof will be broken down into smaller steps.
- Active examples. The reader is allowed to change certain values in an example, which results in the automatic recomputation of all dependent values.
- User-defined (2 dimensional) notation for mathematical concepts. This notation is not only used for displaying information, but also pertains to user input.
- Interpretation of mathematical notation as part of a programming language. A description of an algorithm using pseudo code can be executed on a user supplied problem.
- Guided exercises. Hints, like showing the paragraphs containing the necessary theorems and algorithms, or listing the available operations (directed at solving the problem) can be provided at the click of a mouse.

[2]In the context of this paper the notion of interactive book is restricted to the discipline of mathematics.

All the actions to be applied on (parts of) an interactive book are carried out by a dynamic set of applications. The software that is used to run an interactive book, from now on tentatively called *the system*, basically provides the infrastructure needed to connect these applications to the book. Moreover, it includes an editor/viewer that takes care of all user interaction, and comprises a set of databases, which contain the models used to represent knowledge as well as the knowledge thus represented.

As we will see in the sequel, the way mathematical knowledge is represented in the system is different from that used by an application. Hence we can speak about an *internal* representation, to distinguish from the various *external* representations that are required to communicate with the applications.

The main problem of interaction with the reader or with these so-called external applications is how to transform the internal representation into an external one and *vice versa*. Obviously the system needs to contain an adequate description of these external formats. This and other knowledge will be stored in so-called *application descriptions*. However, we will see that more is needed. Our solution involves a formalisation of the concept of *context*. Context represents the active set of knowledge that is available at a certain moment; it is the key that allows the system to correctly interpret the information it is offered.

By extending the system's knowledge about applications to include qualitative information besides functional information, it can decide which application to use for a specific task, or break down a task in a number of subtasks each suited for a specific application. As the system will thus be able to show some intelligent behaviour, it can also offer its services to the connected applications. A proof checker encountering a symbolic computation can recognise this as a hard task to perform by itself, and ask the system to relegate its evaluation to an application that is better suited for the task, viz. a computer algebra system.

The paper is organised as follows. Section 2 introduces our use of the notions *object* and *view*. Section 3 describes the notions of internal representation, *application description* and *mathematical database*. Section 4 then explains how *context* fits into our model, and section 5 presents an example, showing how the presented model can be put to use. Finally, section 6 lists some of our ideas that still need more elaboration to be incorporated into our model.

2 Preliminary notions

2.1 Objects

Books on mathematics contain many different entities, which we all designate as objects. In this paper we use the term *objects* to refer to mathematical objects only, i.e., we exclude objects such as chapters, sections, paragraphs, and so on. Examples of mathematical objects are expressions, formulae, functions, definitions, theorems, proofs, algorithms, ...

The idea that our notion of objects includes items like definitions and proofs, at first thought might seem somewhat strange. However, keep in mind that the system should enable us to use all kinds of mathematical applications. This includes programs like formal proof checkers; the expressions to be evaluated by this type of program are proofs to be checked, definitions that need to be expanded, etc.

Remark: our view on formal mathematics is influenced by ideas used in the development and implementation of the Automath system; a survey of the Automath project is presented in [De Bruijn80]. As an example, consider Automath's view on the concept `proof of the implication` $A \Rightarrow B$. It is seen as a map taking a proof of A as argument and returning a proof of B as a result. The analogy with ordinary functions acting on ordinary objects is clear: proof classes serve as domains, logical derivations result from evaluating function calls.

2.2 Views

When writing a book authors choose a notation to refer to these objects that suits their needs. Depending on their background and the intended use of an object, they may use different conventions to denote the objects and display the structure of an object. In other words, we use a *visual representation* of an (abstract) object. Figure 1 shows some examples of different representations for the same abstract object. We call each representation of an object a *view* on the object. Though people have preferences regarding their own view on a certain object, they are able to deal with other views quite well.

$\bar{A}$	A^c	`set-complement-of(`A`)`

Figure 1: Some user views on an expression denoting the complement of a set A

This is not true of an application: it needs to be addressed in its own well-defined language, which is often rigid in the sense that it accepts only one unique representation of an object. We call this representation the *application's view* on the object. Figures 2 and 3 show some applications' views on an equation and on a theorem. As can be seen from these examples the user's view on an object is in general quite different from that of the applications. Different applications need different information to perform their task, and applications rely more heavily on the structure of objects, which is expressed in their views. This observation leads us to the main topics to be addressed in this paper:

- What information is needed to be able to generate each view?
- How do we generate a specific view using this information?
- How do we infer this information from a specific view?

$$\sum_{i=0}^{n} x^i = \frac{1-x^{n+1}}{1-x}$$

```
\[ \sum_{i=0}^{n} x^i = \frac{1-x^{n+1}}{1-x} \]
```

```
sum(x^i, i=0..n) = (1-x^(n+1)) / (1-x)
```

Figure 2: Display[3], LaTeX, and Maple views on an equation.

Let $\langle A, +, 0, -, *, \cdot\rangle$ be an associative algebra over a field k. Define a bracket operator $[\,.\,,\,.\,] : A \times A \to A$ as follows:

$$[x, y] := x * y - y * x$$

Then $\langle A, +, 0, -, [\,.\,,\,.\,], \cdot\rangle$ is a Lie algebra over k.

```
[A:SetType]; [+:BinOpType A];
[0:ElementOf A]; [-:UnOpType A];
[*:BinOpType A]; [k:FieldType];
[M:MapType (SetOf k) A A];

(IsAlgebra A + 0 - * k M) ->
IsLieAlg A + 0 - (brkt A + - *) k M;
```

Figure 3: Display and (simplified) Lego views on a theorem.

3 Representing mathematical objects

The first topic mentioned above concerns the representation of objects. In order to decide what a good representation of an object looks like one usually analyses what type of data the object belongs to and what actions one wants to perform on objects of that type. As we don't intend to give a complete description here of each and every mathematical object one can think of, we restrict ourselves to a qualitative description of what we think a good representation must conform to. Let us first look at a list of global demands posed on such a representation.

[3]Actually, the Display view only captures the visual layout and structure of the formula. The bitmap shown is generated from that view.

- Every view on an object reflects a subset of the total information that constitutes the object. In order to be able to generate each view, the internal representation must at least comprise all of this information.
- The representation must allow for translation to any application that knows how to deal with the object involved. E.g., evaluating the expression $\frac{2}{\sqrt{\pi}} \int_0^\infty e^{-x^2} dx$ can be done by most computer algebra systems, though they may require different names for the operations involved and expect arguments in different orders. The internal representation must be independent of these application-specific features.
- The representation must allow for selection of meaningful subobjects. This is not only required for the system to be able to have things evaluated in parallel, the user might want to perform actions on subobjects too.
- As different views on an object can exist at the same time, the danger of having inconsistent versions of an object appears. The system must treat the objects in a way that consistency is guaranteed.
- We want the system to be easily extensible, therefore the representation must be chosen in a way that an extension leads to local changes only:
  - *Adding new functionality* must be possible without interfering with existing functionality. Think, e.g., of programming for a specific computer algebra system; making the implemented routines available to a reader must only involve introducing the logical functionality to the system's knowledge and extending its knowledge about the computer algebra system.
  - *Adding a new application* must be independent of other applications.

These demands strongly suggest the separation of object-specific and application-specific information. The first type of information will be stored in the *internal representation*, the latter in the *application description*.

3.1 The internal representation

The list of demands given above leads us to the following characterisation of the internal representation of objects:

- **Completeness**
  All *intrinsic* information that constitutes an object must be captured by its internal representation. This means not only the value but also the type of the object is part of the representation.
- **Uniqueness**
  Though there can exist several views on an object, side by side, there must only be one mastercopy, which embodies the actual object. All views are generated from this

mastercopy and every piece of information that is present in a view can be tracked down to a unique (set of) source(s).

- **Syntax abstraction**
  The fact that we want to express ourselves independently of the applications we are using, raises the question of which language to use. Observing the fact that the notation we use to designate objects does not influence the way we pronounce what we read — each of the views from figure 1 will be read aloud as: "the complement of (the set) A" — we are able to choose well-known names for actions and constructions[4]. Parameters to an action must be supplied in conjunction with their names, so the order will be irrelevant. Because we allow the user to bind his own preferred names and notation to concepts (including ordering of parameters), the problem of having to use long and tedious names can be resolved; see also section 4 on the use of context.

- **Reflection of structure**
  The semantic meaning of an object is not only deduced from the semantic meaning of the words used to describe an object, but also from the way these words are structured into sentences. The internal structure of an object, i.e., what subobjects can be recognised and how they relate to one another, is part of the intrinsic information of the object and so must be reflected by the internal representation.

The demands on uniqueness and reflection of structure, imply that information which is shared between objects, e.g., the domain an object belongs to, will be stored as a reference. A more elaborate description of how information is accessed and used (activated) is postponed until section 4.

3.2 The application description

For the system to be able to use the functionality offered by (external) applications, and in order to do this wisely, the system needs to comprise knowledge of the applications. This knowledge is stored in so-called *application descriptions*. Essential parts of an application description (without which the translation between internal format and the application's format is impossible) are[5]:

- A mapping from internal names to the names used by the application to address a certain operation. This mapping is domain-dependent: when translating the expression (displayed as) `Factorise(`x^2+1`)` to Maple, the result is `factor(`x^2+1`)` if the ring of coefficients of the polynomial's domain is `Integer`, while it is `Factor(`x^2+1`)` `mod` 2 if the ring is $\mathbf{Z}/2\mathbf{Z}$.

[4]In those cases where several names are applicable, the user will be able to find the internal name bound to the subject sought for via the additional keywords field of its database entry. (Knowledge is internally organised in databases, see subsection 3.3).

[5]Trivial matters such as the names of hosts on which an application is available, the sequence of commands that needs to be issued for starting up an application (including logging on), etc., are intentionally left out.

The system will construct the inverse mapping from the application's names to the internal names. Since it will not always be possible to infer the domain from the application's view, this mapping may be only partially defined. In such cases the system will rely on the use of context (see section 4) to complete the mapping when used. If the application does not deliver the results in a suitable format, a normalising procedure might be necessary too. For instance, if we use an untyped computer algebra system to compute the product of two polynomials from $\mathbf{Z}[x][y]$, collecting and reordering of terms of the returned polynomial might be necessary. Of course whenever possible the computer algebra system involved will be put to use to perform this task.

- A mapping from the internal way of treating arguments to the one used by the application; in general this involves building an ordered list from the set of named arguments.

- An optional mapping which defines priority and associativity of the binary operators offered by the application. When the application generates flat expressions using infix notation and a minimal number of parentheses, this is required so the system is able to build the internal tree structure.

- A formal description of the grammar used to build syntactically correct expressions in the language accepted by the application.

Considering the fact that the operations on this information only involve term rewriting techniques, we believe that a general way of representing it can be conceived. Developing a sufficient and satisfactory model is currently being investigated.

Next to this information, the application description might also contain qualitative information like the space and time complexity of the operations it offers. This information can be used by the system to decide which application to choose for performing the requested operations. Other data upon which such a decision will depend are statistics that are to be gathered by the system itself. These will belong to a specific process running an application and may involve network throughput, host machine load, the amount of memory available to that process, and such.

Remark: when implementing the model described in this paper, it will probably turn out that we need information concerning the state of a specific instance of an application as well. Many applications allow the loading of modules which add functionality or change the meaning of keywords. In such cases the system needs to keep track of which modules are active and the application's description must contain information concerning what modules are required to be active and what commands to issue in order to activate a module. The state of an instance of an application also includes information such as which variables are up to date; a further discussion on this matter is considered beyond the scope of this paper.

Using a representation as described thus far, it becomes clear how part of the functionality that was mentioned in the introduction, is realised.

- **User-configurable notation**
  As the user interface of the system is treated the same way as any other application, the system will comprise a *user interface description*. What is needed to attain user defined notation (for input as well as output) is a tool that allows the user to change the mappings in that description.

- **Parallel evaluation**
  A mathematical expression is a structured object, in which we can recognise a top level function identifier and a set of arguments, which are legal expressions themselves. Because the internal representation reflects this structure, the system is able to decide to have subexpressions evaluated in parallel, by different applications if that is efficient, before having a (possibly third) application perform the final evaluation step. This scheme is depicted in figure 4.

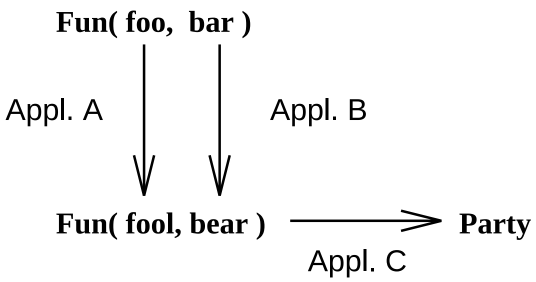

Figure 4: Parallel evaluation of an expression

3.3 The mathematical database

We have already mentioned the fact that we could choose good names to denote mathematical functionality. These names and the objects they refer to must reside somewhere in the system, which brings us to the subject of the mathematical database. The mathematical database serves the following purposes:

- We need a mathematical dictionary describing which words are known to the system and explaining their meaning. The maps in the application descriptions draw their arguments from the entries in this dictionary.

- The database provides the information needed for checking well-formedness of expressions. This includes checking whether arguments belong to the correct domains: if P belongs to the domain `ProofOf(Assert(`$x^2 + y^2 \geq 0$`))` and `MatrixDeterminant` is an operator only defined on domains generated by the `SquareMatrix` domain constructor, the expression `MatrixDeterminant(`P`)` is considered syntactically invalid.

- The database offers the means to extend the system's knowledge. The tools that manipulate it comprise ways to define new mathematical objects such as categories, domains, properties, axioms and their constructors.

- The database captures the natural hypertext structure of mathematics. Defining a new type of mathematical object is often a matter of introducing an abbreviation for a structured object involving previously defined objects and notions. Based on certain properties, operations can be defined on instances of the object type. The way this information is stored in the database exposes the interrelationships.

The knowledgeable reader probably has already noticed our use of Axiom [Jenks92] terminology. Indeed, the structure of our mathematical database resembles the way Axiom treats mathematical objects. A notable difference lies in the fact that in our approach a domain has to provide the default names for the required special elements and operators, when proclaiming it belongs to a certain category. This way we can express the category `AbelianGroup` as the category `Group` extended with the property `Commutative` for the binary operator. In Axiom these two categories reside in different branches of the category hierarchy.

Remark The price we have to pay is the repeated binding of actual and formal parameters the category author has to provide. However, when the author selects a super category into which the current category is embedded, the tool used for adding new categories presents the author with a suggested list of bindings based on type and name of the parameters. Furthermore, the author can indicate the binding through point-and-click operations.

Let us continue with defining the components of our mathematical database.

- **Categories.**
  A category represents a class of domains and describes (part of) the structure of the domains belonging to it. It presents an overview of the special elements and operators of such a domain and gives a list of the properties that must hold for these. It does so by stating in which other categories the category is embedded, and giving additional properties or axioms that must hold. See figure 5 for a tentative example.

  Categories serve to expose mathematical interrelationships, to store the properties to be used in constructing proofs, and to allow the use of polymorphic algorithms. The validity of such an algorithm depends on certain properties only and is not bound to a specific domain. Binding it to a category therefore seems natural and allows it to be applied to any domain belonging to the category.

- **Domains and domain constructors.**
  A domain is essentially an *abstract data type*; it models the objects belonging to the domain and the operations that can be performed on them. From a mathematical point of view a domain represents a *typed* set. A domain specification defines the default representation of its elements, identifies special elements and defines operations on its elements. Finally, by stating to which categories the domain belongs, the structure of the typed set is expressed.

Domains are generated using domain constructors, which can take other domains as parameters. Some examples:

`RealNumber, Integer, UniVariatePolynomial( Ring, Symbol).`

The system automatically creates a generic abstract domain for each known category. This domain can be used for abstract reasoning and to express the polymorphic operations mentioned in the previous item. New domains can also be constructed as subdomains of existing ones by adding predicates that must hold for objects in the base domain. Unless redefined, all operations will be inherited from the basedomain; when used their results will be checked against the predicates to see if they belong to the subdomain.

- **Functions.**
  Functions need not be defined as part of a domain constructor. They can also be defined in groups called packages. A function definition will in general consist only of stating the domain it belongs to and giving the properties that show the relation with other functions in the same package. For example, the functions `RadianSine` and `RadianCosine` might be defined in the package `Trigonometry`, which would contain properties like the sum of their squares equals the constant (function) one. The domain they belong to could be $C^\infty(\mathbf{R}, \mathbf{R})$ (where $\mathbf{R}$ denotes `RealNumbers`) as opposed to, say, `UnaryOperator(`$\mathbf{R}$`)`, which would be their type if they were defined by the domain `RealNumbers`. On the other hand, default expansions (templates) can be given, allowing one to give a polymorphic definition for an operation (in such a case the operation will be bound to an abstract domain or category).

- **Properties or axioms.**
  The properties are used by categories and domain constructors as the primary means to describe the structure of domains. Each one represents a valid operation on the trees used to represent mathematical objects. Translating a property to a proof checker's view may result in a rewrite rule to be used in a proof. We can also think of a user applying operations to prove equality between two expressions represented as trees; by pointing and clicking he would select the action, connect actual to formal parameters and apply the action. The rule for associativity might then be represented to the user as shown in figure 6.

- **Domain conversions.**
  It is often the case that a domain can be naturally embedded in another domain, even though it is not explicitly constructed as a subdomain of that other domain. For instance, the rational numbers can be seen as a subfield of the field of real numbers. If we want the system to accept rational numbers where formally only real numbers would be valid, we need to provide the system with a conversion operation that maps rational numbers to the corresponding real numbers. Such a conversion can be given as a function — in which case the user explicitly needs to apply it — or it

Category identifier: `Ring`

Arguments:

```
Basetype:          ObjectType
zero:              Basetype
add:               (left:Basetype, right:Basetype) → Basetype
minus:             (argument:Basetype) → Basetype
multiply:          (left:Basetype, right:Basetype) → Basetype
```

Inherited Properties:

```
AbelianGroup(Basetype=Basetype, E=zero, operator=add, inverse=minus)
SemiGroup(Basetype=Basetype, operator=multiply)
```

Additional Properties:

```
DistributeLeftBranch(OldTop=multiply, NewTop=add)
DistributeRightBranch(OldTop=multiply, NewTop=add)
```

Note: these two properties check whether the types of the operators match and create equivalences of:

$\forall a, b, c \in$ Basetype: $a * (b + c) = a * b + a * c$
$\forall a, b, c \in$ Basetype: $(a + b) * c = a * c + b * c$

Figure 5: Part of description of the category `Ring`

can be added to the domain conversions section of the database. In the latter case the system will transparently add the domain conversions where applicable. When explicitly creating a domain as a subdomain, this section of the database will be updated automatically.

All entries in the mathematical database carry additional information, such as a field containing a verbose description of the entry, extra keywords by which the entry can be found by the database browser, pointers to related issues, classification codes, etc.

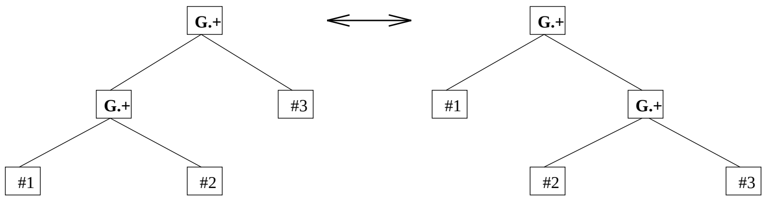

Figure 6: Tree representation of property `Associative(Domain=G,Operator=+)`

4 Context

Now that we have given a description of the internal representation of mathematical objects, and have shown the viability of generating an application's view, we are still left with the following problem:

- How do we generate the internal representation from these views?

This translation does not only involve parsing expressions written in an application's language — we will need the information from the application's description for that part of the task — as many an application's view lacks part of the information needed. Maple's view on polynomials for instance, completely ignores the concept of *ring of coefficients*. Even after feeding Maple the expression `Factor(`x^2+1`) mod` 2 it forgets the domain information in subsequent computations. What we need is a mechanism that allows the system to deduce the missing information; this is where *context* starts playing a rôle. We think of context as the set of knowledge that is *actively* available at a certain moment in time. In a way it temporarily restricts our knowledge to a small dedicated set, thus allowing us to correctly interpret otherwise ambiguous statements. This characteristic is retained by our formal model of the concept. Our formalisation of context offers the following functionality:

- It provides the living-room for the objects which are currently active — an object is called active if it is allowed to be engaged in computations. To each object in the context a name can be bound, which can be used to refer to the object. Thus, the author can attach a name to an expression that is displayed to the reader, say `Expr`, and use `Evaluate(Expr)`[6] to create a piece of text in which the latter object will get updated as soon as the reader changes the first.

- Before an object can become active (by entering the context) its internal representation must be generated/completed. In many cases this will involve the use of the current contents of the context: if it contains a variable x belonging to the domain

[6]The formal parameter names may be omitted if the order of the parameters is irrelevant (as for any unary or commutative binary operation, or when all actual parameters are identical), or when the formal association can be deduced unambiguously from the context, e.g. by type considerations.

$\mathbf{Z}/7\mathbf{Z}$ and we want to activate the expression $x + 3$ (appearing as an untyped expression in the interactive book sources), the system infers the domains for $+$, 3 and $x + 3$. (`BinaryOperator(`$\mathbf{Z}/7\mathbf{Z}$`)`, $\mathbf{Z}/7\mathbf{Z}$ and `Expression(`$\mathbf{Z}/7\mathbf{Z}$`)`, for instance). Using this mechanism the system interprets objects at the time and place they are used. The meaning of an expression is thus dependent on material directly preceding it (from the reader's point of view!). So, if there are several routes via which a reader can arrive at a certain point in an interactive book, the contents is interpreted according to the route taken. It is this behaviour that justifies the term *context*.

- As the internal representation of an object is complete (see subsection 3.1), all the objects related to an active object reside in the context as well, i.e., the part of the mathematical database containing the object's domain and the operations defined therein. Using this information the system can complete partial internal representations that, e.g., are delivered as a result computed by a computer algebra system (see subsection 3.2). The system only uses *active* knowledge when interpreting (parsing) objects to their internal representation. This has the advantage that otherwise ambiguous expressions can now be correctly understood without the aid of the reader.

- If an object in an interactive book is editable, the editor — the application used for editing — can question the context about the type of the object allowed as input. Using this type and the operations allowed on objects of this type (also extracted from the context) the editor is able to accept only valid expressions and aid the user in doing so. (Showing a list of available functions/operators, auto-completing partial function names, etc.).

- If we think of the system as an interpreter and the interactive book sources as programs, then the context serves as the set of visible data. Introducing new objects into the context can temporarily hide or permanently destroy previous objects and objects can be declared local to a part of a book, such as a section or a paragraph. Using these mechanisms the author has complete control on the lifetime and visibility of objects. For example, in a problem solving section of a book, a careful setup of the context by the author can provide the reader with a limited set of objects and actions, that are especially suited to the task at hand.

5 Outline of an interactive example

In order to get an idea of how everything works together we will look at a simple example. Suppose the following box shows an extract from an interactive book:

This example shows us the results of applying the schoolbook division algorithm on two polynomials with integer coefficients. You can change the dividend and the divisor and see how this affects the quotient and the remainder.			
Dividend:	$x^2 + 3x + 2$	Divisor:	$x + 5$
Quotient:	$x - 2$	Remainder:	12

In this example boxes surrounded by a double outline denote the parts that are editable by the reader, the singly outlined boxes contain the non-editable results and the items that have a hyperlink attached to them are printed in Sans Serif.

When the reader visits this paragraph — the paragraph gets displayed — the context needs to be initialised so that the needed computations will take place. Therefore a special object is attached to the paragraph object, that contains the instructions that activate the necessary mathematical objects. Figure 7 shows what the contents for this *context initialising object* might look like. In that figure we use four different types of assignment:

Type assignment (:)
: connects a domain to an object, if necessary a new object will be created. In this particular case it is used mainly to ensure that any previous value x might have will be invisible in this paragraph.

Name assignment (=)
: defines an alias. All subsequent references to `POL` will be replaced by references to the given polynomial ring.

Direct assignment (:=)
: replaces the previous value of an object by the given one. Here we have used it in conjunction with the type assignment, which ensures us only objects of the given type may be assigned to `P1` and `P2`.

Delayed assignment (==)
: binds the given expression to the object, i.e., no evaluation of the right hand side will take place. Furthermore, we haven't restricted the pair to a type. As the members of the pair are used in the book as read-only objects, there is no need to impose such a restriction.

The text part of the paragraph then looks like[7] figure 8. Concerning the horrid look of this piece of text one should keep in mind that the authoring system takes care of providing the mark-up. As the intended meaning of the mark-up is clear, we return to our original pursuit: looking at how the system handles the mathematical objects involved.

[7] As the development of the system is still in the design phase, the actual format may differ from the shown one.

x:	`Symbol`
`POL =`	`UnivariatePolynomial( CoefficientRing=Integer, Dummy=`x` )`
`P1 :=`	$x^2 + 3x + 2$ `: POL`
`P2 :=`	$x + 5$ `: POL`
`(Quotient, Remainder) ==`	`SchoolbookDivision(RingObject=POL) (Dividend=P1, Divisor=P2)`

Figure 7: A context initialising object

When the reader visits the paragraph the first thing that happens is the initialisation of the context. A new local context block is created and the following sequence of actions takes place:

- A new name x is added to the newly created block and, if it wasn't already in the context, the `Symbol` domain is loaded from the domain section of the mathematical database. An object of the domain is then created and bound to the name x.
- A new name `POL` is added. If the polynomial and the integer domains weren't in the context, the `Integer` domain and the domain constructor for `UnivariatePolynomial` are loaded and, after checking the types from `Integer` and x to be `Ring` and `Symbol`, expanded. The resulting domain object is then bound to the name `POL`.
- The new names P1 and P2 are added and two polynomial objects with the given values are created. These values then get bound to the corresponding names.
- If not already there, the names `Quotient` and `Remainder` are added, the `Schoolbook-Division` is loaded from the packages section of the mathematical database and, after checking the type of `POL` to be `EuclideanRing`, expanded. Finally, the types of `P1` and `P2` are checked and the *invariant* that binds the values of `Quotient` and `Remainder` is created.

After the context initialisation has taken place, the system proceeds with actually displaying the paragraph on the screen. To this end, using the information stored in the display description, a logical visualisation of the paragraph is built. When the system encounters `Evaluate(Quotient)` in the source it will consult the context to get the internal representation of the object. The system then needs to evaluate the right hand side (RHS) of the invariant for the first time. It will select an appropriate computer algebra system (CAS), i.e., the translation of the RHS using the CAS's application description is successful, and send the necessary expressions to the CAS. Again using the CAS's application description and the knowledge that the returned expression is a pair of univariate polynomials with integer coefficients[8], the results are converted to the internal representation

[8]Expansion of the definition of the schoolbook algorithm includes its signature.

```
/SHOW
  /SENTENCE
    This example shows us the results of applying the
    /DICT schoolbook division algorithm \DICT
    on two /DICT polynomials \DICT with
    /DICT integer \DICT /DICT coefficients \DICT
  \SENTENCE
  /SENTENCE
    You can change the /DICT dividend \DICT and the
    /DICT divisor \DICT and see how this affects the
    /DICT quotient \DICT and the /DICT remainder \DICT
  \SENTENCE
  /DISPLAY
    /DICT Dividend \DICT : /INOUT /MATH P1 \MATH \INOUT
    /HFILL
    /DICT Divisor \DICT : /INOUT /MATH P2 \MATH \INOUT
  \DISPLAY
  /DISPLAY
    /DICT Quotient \DICT : /OUT /MATH Evaluate(Quotient) \MATH \OUT
    /HFILL
    /DICT Remainder \DICT : /OUT /MATH Evaluate(Remainder) \MATH \OUT
  \DISPLAY
\SHOW
```

Figure 8: Sample of text part of a paragraph

and bound to `P1` and `P2`. The system will also set a flag to note that the current value conforms to the invariant. This has the advantage that a subsequent request for the value of the evaluated remainder will be handled without recomputation taking place. When the logical visual image is complete, the actual visual image is generated and the paragraph finally gets displayed on the screen.

The last thing to describe is what happens when the reader changes the values of `P1` and/or `P2`. When the reader selects an object to edit, the editor first finds out the allowed type of the object. This information is provided by the context. Then it loads from the display description the rules that describe how such objects are to be displayed. In this case the display's view on `Integers, UnivariatePolynomials` and x are loaded. Using all this only valid polynomials are accepted as input. Note: as the context does not contain any special operations which deliver the required polynomials as a result (e.g., taking the derivative with respect to x), the reader is only allowed to use the operations defined in the domain to construct polynomials. Once the reader has finished editing, the context gets updated. The system then notes the invariant is out of date, and will initiate recomputation of the dependent values and update the display as was described in the preceding paragraph. Figure 9 depicts the main dataflow resulting from changing `P1` or `P2`.

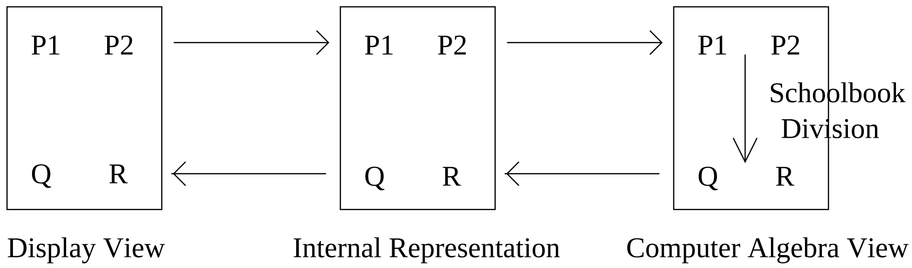

Figure 9: Main dataflow when the example is updated

6 Future ideas and conclusion

The way the system handles user input allows us to achieve a higher degree of application integration by treating applications likewise. This can be particularly useful if we consider the case where a proof checker encounters a step requiring symbolic evaluation. Consider, for example, a proof step consisting of evaluating the validity of $2^{10} > 10^3$, where the natural numbers are implemented using Peano's laws. For a computer algebra system this requires much fewer evaluation steps than for the proof checker, so if the proof checker could consult our system, a significant gain in speed could be achieved. Another advantage lies in a gain in simplicity when presenting the proof to the reader, who will probably be satisfied with presenting the equation to be equivalent with $1024 > 1000$.

As the mathematical database is never complete and authors will surely want to use their own code from within an interactive book, interactive book deliverables need to comprise ways of automatically extending the mathematical database as well as transfering, say, computer algebra specific code to a particular instance of the CAS. Furthermore, at the same time the application's description needs to be updated to reflect the new functionality. The final product will include such a mechanism.

If the system proves to be successful, the internal representation may serve as part of a protocol to exchange mathematical expressions amongst different computer algebra systems. Expanding on this idea a protocol for exchanging mathematical documents might eventually arise. While it is not one of our main goals, we do believe such a protocol will be conceived in the near future.

Allthough we have spoken about the mathematical database as being part of our system, it is probably useful to have it implemented as a stand-alone application. It is the part of our system which best allows itself to be shared among applications, after all, it is mainly accessed to retrieve information, which easily allows for concurrent use. Thus, we can imagine research groups setting up networkwide mathematical database servers.

Of course our system will only be useful if there is enough material — interactive books — available. As many mathematical documents are written using a TEX dialect such as

LaTeX or AMSTeX, a special interactive tool for converting such documents, requiring human assistance to fill in the gaps, might be of great help in producing new material.

References

[Cohen94] *The Acela Project: Aims and plans.*
by A.M. Cohen & L.G.L.T. Meertens.
To appear in *HISC proceedings*, 1994.

[De Bruijn80] *A survey of the project AUTOMATH.*
by N.G. de Bruijn.
Published in *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, edited by J.P. Seldin - J.R. Hindley, Academic Press, 1980.

[Fultz93] *Mathematica User's Guide For the X Front End.*
by J. Fultz and J. Grohens.
Published by Wolfram Research, Inc., 1993.

[Jenks92] *Axiom. The scientific Computation System.*
by R.D Jenks and R.S. Sutor.
Published by Springer-Verlag, 1992.