How to Make AXIOM Into a Scratchpad

Richard D. Jenks Barry M. Trager
IBM Thomas J. Watson Research Center
P.O. Box 218, Yorktown Heights, NY 10598 USA
{jenks,bmt}@watson.ibm.com

1 INTRODUCTION

Scratchpad [GrJe71] was a computer algebra system developed in the early 1970s. Like M&M (Maple [CGG91ab] and Mathematica [WolS92]) and other systems today, Scratchpad had one principal representation for mathematical formulae based on "expression trees". Its user interface design was based on a pattern-matching paradigm with infinite rewrite-rule semantics, providing what we believe to be the most natural paradigm for interactive symbolic problem solving. Like M&M, however, user programs were interpreted, often resulting in poor performance relative to similar facilities coded in standard programming languages such as FORTRAN and C.

Scratchpad development stopped in 1976 giving way to a new system design ([JenR79], [JeTr81]) that evolved into AXIOM [JeSu92]. AXIOM has a strongly-typed programming language for building a library of parameterized types and algorithms, and a type-inferencing interpreter that accesses the library and can build any of an infinite number of types for interactive use.

We suggest that the addition of an expression tree type to AXIOM can allow users to operate with the same freedom and convenience of untyped systems without giving up the expressive power and run-time efficiency provided by the type system. We also present a design that supports a multiplicity of programming styles, from the Scratchpad pattern-matching paradigm to functional programming to more conventional procedural programming. The resulting design seems to us to combine the best features of Scratchpad with current AXIOM and to offer a most attractive, flexible, and user-friendly environment for interactive problem solving.

Section 2 is a discussion of design issues contrasting AXIOMwith other symbolic systems. Sections 3 and 4 is an assessment of AXIOM's current design for building libraries and interactive use. Section 5 describes a new interface design for AXIOM, its resulting paradigms, and its underlying semantic model. Section 6 compares this work with others.

2 DESIGN ISSUES

Symbolic mathematical systems are software systems that manipulate formulae and perform algebraic computations. The first such systems to do this were FORMAC [SamJ66] and ALTRAN [BrnS73], both initiated in 1962. The 1971 SIGSAM conference introduced MACSYMA [MaFa71], a new REDUCE [HrnT71], and Scratchpad. Later came CAYLEY [CanJ84], DERIVE [RiSt92], SMP [CoWo81], Maple, Mathematica, and AXIOM. Scratchpad and ALTRAN stopped development in the mid-70s. Others became commercial products.

Most computer algebra systems have a General Representation for Expressions As Trees (GREAT) as their principal datatype for user data.[1] FORMAC was the first GREAT computer algebra system. MACSYMA had a GREAT representation among others. More recent examples of GREAT systems are M&M and DERIVE.[2]

In contrast, AXIOM provides a large number of canonical datatypes but no user-accessible GREAT representation. AXIOM Version 2 has two languages: a new strongly-typed programming language $A^\sharp$ [WaBD94ab] for developing library code and an interpreter language for interactive use. The AXIOM interpreter does type inferencing for ease-of-use and to permit compilation of user programs for improved run-time performance.

AXIOM's design is radically different from systems with GREAT designs. Its library is a hierarchically organized collection of currently over 1200 *abstract datatypes* (LosR74, [LiGu77]). AXIOM can build any of an infinite number of types in response to user input. When a user enters the expression `%i + x/2`, ($\%i = \sqrt{-1}$), for example, AXIOM builds the type `Polynomial Complex Fraction Integer`[3] A user asking to factor an expression may trigger the use of an algorithm requiring computations with, say, matrices with integer-mod-7 coefficients. To run this algorithm, AXIOM dynamically builds the type `Matrix IntegerMod(7)`.

Users can use the $A^\sharp$ in Version 2 to create stand-alone applications or to extend the existing AXIOM library. $A^\sharp$ is an object-based programming language that handles functions and parameterized types as first class values.

[1]An *expression tree* is recursively defined as either an atom (one of several basic objects) or else a list of one or more expression trees.

[2]The internal representation of REDUCE and Scratchpad is not so much GREAT as it is canonical. LISP [McCa60] is a good example of a general purpose GREAT programming language.

[3]This type is expressed in the AXIOM syntax. Here juxtaposition means function application and associates to the right. The type is equivalently written **Polynomial (Complex (Fraction (Integer)))**.

Performance

GREAT systems interpret their expression trees from the bottom-up, effectively replacing each successive interior node by a value. AXIOM-type systems interpret parse trees in a similar way but in addition must build types at each node [JeSu87].

AXIOM is relatively slow on start-up when type constructors are first loaded from the library and new types are built and cached for later reference. Its design however has an advantage of *decisiveness* over GREAT systems: knowing the type of user data avoids needless run-time testing.[4] That plus the speed inherent from compilation makes AXIOM more run-time efficient than GREAT systems for large computations running comparable algorithms.

Strategy

M&M define a *kernel*, a subset of the system hand-coded in C to be both compact and efficient for customary use. Other code is written in the M&M language and is interpreted with marginal loss in efficiency for *small* problems.

In contrast, all of AXIOM's library code is compiled. Is compilation necessary? It depends. Kernels necessarily provide small coverage over the potential range of computer algebra applications. M&M systems have yet to demonstrate performance relative to the compiled algorithms of AXIOM on, for example, large Groebner basis computations [BucB91]. Performance on relatively unusual applications can be much slower relative to compiled code.[5].

What distinguishes AXIOM from GREAT systems is not so much its use of types internally but rather its use of dynamically constructible parameterized types and "categories" (see below). GREAT systems *could* or *do* use types internally, allow static type declarations, and provide a compiler to produce efficient code for algebraic computations.[6] On the other hand, AXIOM code does *not* have to run compiled. $A^\sharp$ has a compact internal representation with an interpreter [WaSM94]. Also AXIOM users can use $A^\sharp$ to design and implement their own specialized M&M-like kernels and interpret other code.

Size

For general purpose computer algebra, GREAT systems can be much smaller than AXIOM-like systems.[7] AXIOM's dynamic types are run-time objects that take up space that can grow large for applications requiring significant portions of its library. Maple recommends 4MB and Mathematica 8MB on a 486.[8] AXIOM requires 16MB on a 486 to avoid expensive paging.

While a smaller footprint means greater accessibility to users, size is becoming less of a problem year-by-year. We now see 486 machines growing in increasing numbers relative to 386 machines. Users who could previously run only DERIVE can now run M&M if not AXIOM.

3 AXIOM LIBRARY LANGUAGE

A principal aim of AXIOM's library language design is the ability of users and system implementers alike to use the same language (now $A^\sharp$) to build a rich library of type-safe and reusable modules of datatypes and algorithms.

Datatypes

Mathematics has a richness of objects that today's computer algebra systems barely touch upon. There are many kinds of numbers: integers, integers mod p, rational numbers, floats, complex numbers, quaternions, radicals, factored integers, p-adic numbers, and numbers that are roots of specific polynomial equations, all of which are implemented in AXIOM. Many algorithms require "bignums" (integers of indefinite size) and "bigfloats" (floats with arbitrarily large mantissa and exponent). For others, single- or double-precision (machine) floats, full- or half-word integers, even bits or byte strings for small non-negative-integers, can be critical for performance.

Once you have numbers you will want to form aggregates of them like lists, arrays, finite sets, and mathematical structures such as polynomials, matrices, power series, and tensors. Data structures such as hash tables, extensible arrays, doubly-linked lists, and balanced binary trees each offer optimal efficiency for particular applications. All of these types are defined by AXIOM source code available for user modification and extension.

Algorithms

A key contribution of AXIOM to computer algebra is its language and compiler for algorithms and algebraic properties (now $A^\sharp$):

> AXIOM provides a language for parameterized *categories*, for describing types and algebraic properties of objects that ensure type-safety. Algorithms are run *if* and *only if* its parameters have certain prescribed properties.

The *if* addresses generality and code reuse. A linear equation solver defined over any field F is parameterized by F. $A^\sharp$ compiles this algorithm to produce code that runs with *any* field F such as rational numbers, rational functions, algebraic extensions, or any new field later added to the system.[9] Also, the efficiency of Groebner basis computations depends upon the choice the underlying polynomial representation, the coefficient domain, and term ordering. AXIOM can compile an algorithm which efficiently handles any choice of these three kinds of parameters at once.[10]

[4]Experience with GAUSS, a system that implements AXIOM-like concepts in Maple, supports this point. Although GAUSS runs interpreted, it runs faster than comparable Maple code "because no time is wasted analyzing what kind of expression was input." [GrMo93]

[5]Martin Schoenert built GAP [SchM93] with its own kernel for group theoretic computations that runs 2-3 orders of magnitude faster than comparable Maple code [SchM94]

[6]No current GREAT system approaches Common LISP in its richness of underlying types. Common LISP implementations have compilers which in one instance has demonstrated FORTRAN-like efficiency for numerical applications [SteG84].

[7]MAPLE's computer algebra kernel consists of about 25,000 lines of C code, not counting graphics, I/O, and user interface code [Mona94b].

[8]At the time of writing this paper, a 486 version of AXIOM is not commercially available.

[9]Other designs must implement multiple versions of algorithms. For example, MACSYMA once had 6 different linear equation solvers.

[10]Maple has 4 different Groebner basis implementations which handle only 2 specific kinds of polynomials and 2 specific term orderings. [GruD93]

The *only if* addresses type safety and extensibility through static category declarations:

- Categorical declarations ensure that only objects with the correct algebraic properties are passed to algorithms, e.g. to disallow the passing of a finite field element to an algorithm expecting one from a field of characteristic zero.

- Categories are structures that help ensure that new code will integrate smoothly both with existing and future code in the system.[11]

- Categories provide a high-level understanding of the current and future structure of the library.

AXIOM's ability to handle parameterized types and categories also give it an advantage over object-oriented languages such as C++ [StrB92] which do not have dynamic types nor allow type parameters to have types; type parameter errors must be detected by user code at run-time.[12] In constrast, $A^\sharp$ can perform such type-checking at compile time.

4 AXIOM INTERFACE LANGUAGE

AXIOM's library organization as a large collection of types and categories presents an interesting challenge for the design of its user interface for interactive problem solving.

AXIOM has a type-inferencing engine designed to minimize the need for beginning users to declare types for simple problems. Type declarations are however often useful and preferred by experienced user. Type declarations can be used to define a *type context* much as in mathematics. For example,

```
K := Fraction Polynomial Integer
E := CliffordAlgebra(3,K, quadraticForm 0)
e: E
dual(e) == ...
```

in English means "Let K denote the field of rational functions. Let E denote an exterior algebra on a three space over K. Let e be an element of E. Define the dual of e as.."

Type contexts allow common names like norm and operations like * to be reused (overloaded), to have different meanings in different contexts. Given a type context, names become unambiguous so that users can interactively use names and natural notations to which they are accustomed. Conversely GREAT systems have one principal type with one set of operation names; as a typeless system grows, users and system designers are forced to use long, unnatural, or qualified names to avoid ambiguity.

As each AXIOM object has a type with certain prescribed algebraic properties, it is possible to rather precisely describe to the user what library operations are applicable to a given result. These include not only operations exported by the type of the object but also "categorical" operations.

AXIOM's current interpreter converts each user expression immediately into a specific system datatype. This conversion often implies considerable restructuring of the user's input. Users sometimes want to manipulate their input forms and output results as formulas without being forced into type-specific representations.

Algebraic algorithms are typically designed to work over structures of homogeneous type. In mathematics one talks about "polynomials over a field". Users may want to create a list of objects of different values, say, integers, integers mod p, and rational numbers, without having to require them to come from a common place. But the elements of lists in AXIOM must all have a common type. Thus adding a rational number as a new element to an existing list of integers requires all elements of the list to be converted to rational numbers.

A more serious problem concerns *mutable types*, types whose values contain updateable components. For example, the attempt to replace an element of a list of integers with a rational number would force all other elements to become rational numbers. Since AXIOM does not provide back pointers for embedded objects, it will refuse to replace a component by an object of a different type.

5 A NEW AXIOM INTERFACE

The existing AXIOM interface language was originally designed to be as close as possible to its original programming language used to develop the AXIOM library. Here we propose a new user interface language having among other features an untyped interface. The new language proposed is neither AXIOM nor $A^\sharp$. We call it here $B^\natural$ (pronounced "B-natural"), a higher-level language serving as a bridge between AXIOM and $A^\sharp$. New users begin with type User. Advanced users can "break-out" to normal AXIOM by issuing declarations.

The principal differences between the $B^\natural$ and the existing user interface language is the introduction of a new type User and various syntax and semantic conventions.

5.1 TYPE User

A new type User provides a GREAT component to AXIOM. The new design still allows full access to the AXIOM library as before with this difference: all user results are converted to type User. Also aggregate types have type User as their underlying domain. This allows users to build structures with different kinds of objects and for mutable structures to be updated without the need of changing type as mentioned above.

All domains in AXIOM accessible from type User export a coercion to convert objects of the domain into objects of type User, and coercions the other way that may fail. This simplifies the task of the interpreter that often has to find a coercion path between two given types.

[11] Other system designs have problems with "implied assumptions" that a new kind of object may violate. A former long-standing bug of Maple involved the introduction of complex arithmetic by use of the symbol i for $sqrt(-1)$. All appropriate arithmetic routines checked for this symbol. Yet the computation of the rank of the matrix $\begin{pmatrix} 1 & i \\ -i & 1 \end{pmatrix}$ gave the wrong answer.

[12] It should not be surprise the reader that AXIOM's object-based design differs fundamentally from those of object-oriented languages. While our design goals (code reuse, data abstraction, and polymorphism) are much the same, AXIOM's design featuring dynamic parameterized types and categories seems to us to be a natural consequence of our attempt to model "constructive mathematics" from the past 200 years. In contrast, the design C++ has evolved from SmallTalk [GoRo83], CLU [LiGu77], and others, and guided largely by business and computer science applications, e.g. graphical user interfaces.

Type User has exported operations that allow the user to do general formula manipulation, to transform parse forms and results as symbolic formulae under rather complete user control. Transformations can be done by pattern-matching or direct manipulation. Expression trees provide a simple and intuitive model of a formula.[13] Also TEX-formatted display of an expression can generally map directly into a GREAT internal representation. Such transparency makes a well-designed pattern-matcher easy to use and understand.

Type User makes the user interface appear simpler to beginning users by making it type-less. Beginners can first use AXIOM as a type-less system, then later as they need to do more, they can learn about types.

Type User could serve as a gateway to the AXIOM library for use either as a client or server for other systems and languages. For example, type User might represent and implement a protocol language for communication between the AXIOM library and an external language such as OpenMath [VorS94].

Finally, type User provides an excellent basis for an expression editor where users can select, drag, and drop subexpressions using a mouse or a pen. Type User is represented internally by an attributed tree, a tree having a property list at its nodes. The AXIOM interpreter does a bottom-up traversal of the parse tree caching computed values in the property lists at the nodes.

5.2 CONVENTIONS

We now present various syntax and semantic conventions which we believe are useful for interactive problem solving.

Names

Names are case sensitive. By convention, type names begin in uppercase; library function names usually begin in lowercase. Full names are preferred; users can `)include` their own alias file for abbreviations.

System default type aliases have 3 or more uppercase letters (e.g. `INT` for `Integer`). Other aliases are used for functions (e.g. `int` for `integrate`, `D` for `differentiate`, `N` for `numeric`). Names with multiple words are run together with successive words capitalized (e.g. `nullSpace`).

Names can have `?` and `%` as special characters. Names of Boolean valued functions (*predicates*) end with a question-mark (?). Characters `?` and `%` are also used in pattern-variable names (see below).

TEX-like Conventions

As TEX has become a standard for typesetting mathematics, we propose to extend the AXIOM language to enhance interactions with TEX formulae in two ways, first to provide for TEX output of results, second to incorporate linear notations that allow a full range of scripting and accents similar to those provided by TEX.[14] A percent prefix is used to denote special constants, e.g. `%e` for $exp(1)$ and `%i` for complex(0, 1). TEX mathematical symbols and accents are similarly marked. Symbols `%pi` and `%Pi` (\pi and \Pi in TEX) denote π and Π respectively.[15] Suffixes to names beginning with `%` give accents to names, e.g. **`x%hat`** for $\hat{x}$. **`%alpha%underline%tilde`** for $\underline{\tilde{\alpha}}$. TEX symbols used as operators have prefix &, e.g. **`x &cup y`** for $x \cup y$. **`x &not&equiv y`**, as $x \not\equiv y$.

The notations `x_i`, `x_{i_j}`, and `x_{i,j}` denote subscripted variables and display x_i, x_{i_j}, and $x_{i,j}$ respectively. For more complicated scripting, operators **`&n`**, **`&s`**, **`&ne`**, **`&nw`**, etc., can be used to place scripts at locations above, below, and around the operation name for TEX display. For example, `x_{i &nw j}` displays as ${}^{j+1}x_i$ and `%Sigma_{&s i = 0 &n n}` as

$$\sum_{i=0}^{n}$$

$B^\natural$ allows two separators within braces: commas and semi-colons, thereby offering a convenient shorthand notation when scripts are confined to the four corner positions. Inside script braces, commas separate arguments at the same location whereas semi-colons move the script position counter-clockwise from SE to NE to NW to SW. Thus `f_{i;j}` is an equivalent way to write `f_{i &ne j}` which displays as f_i^j. Likewise `f_{&ne i,j &sw k + 1}` can also be written as `f_{;i,j;;k+1}` and displays as ${}_k f^{i,j}$. Using semi-colons, users have five positions in which to place arguments to symbols, e.g. `f_{1;2;3;4}(5)` means ${}^3_4 f^2_1(5)$. Scripts and accents can occur to any depth and in any position. Thus `x%vec%prime_{f_j(x)}` displays as $\vec{x'}_{f_j(x)}$. Other useful notations are D_x for $\frac{\partial}{\partial x}$, $D^{1,2}_{x,y}$ for $\frac{\partial}{\partial x}\frac{\partial^2}{\partial y}$, $\sum_i$ and $\int_x$.

To refer to a variable with a given arrangement of scripts, a dot is placed in each script position. Thus the syntax `f`, `f_.` and `f_{.,.}` is used to respectively refer to the distinct variables f (no scripts), $f_\bullet$ (one subscript), and $f_{\bullet,\bullet}$ (two subscripts).

Scripted variables can be either functions or variables. If functions, they are functions of the scripts as well as their normal arguments (see the Laguerre polynomial example below).

Operators

New operator conventions are designed to offer convenient notations for interactive use.

Quote. A prefix quote-mark (') combines with other standard operators to form a new operator with the same precedence. This operator is used to construct expression tree of type User. Whereas `x + y` will apply the operator `+` to the operands `x` and `y`, `x '+ y` creates an expression tree with root `+` and two branches `x` and `y`.

[13] User studies have shown that mathematically-trained users naturally visualize mathematical formulae as expression trees.Those with little math backgrounds however tend to visualize math formulas as strings [AviR94].

[14] Regrettably direct use of TEX syntax is not appropriate. As TEX is a mathematical text language, it does not distinguish superscripts from powers nor operator names from identifiers. $B^\natural$ distinguishes between operators and identifiers as does $A^\sharp$. Also / and \ are useful mathematical operators and thus available for user definition in $A^\sharp$ either alone or in combination with other symbols.

[15] Symbols pi and %pi are distinct. A user may want an identifier name *pi* distinct from the corresponding Greek letter.

New Operators. Three new operators offer user convenience and expressive power for dealing with aggregates such as lists.

- **Strip**. The prefix operator `:` is used within list constructor brackets to strip off one level of parenthesis. The notation `[:x, :y]` is equivalent to `append(x,y)` and creates a list that joins lists `x` and `y` end-to-end.
- **Over**. Any infix operator θ can be applied *over* a list `u` using the notation θ`/u`. Thus `+/u` will sum the elements of `u`. Also if `p` is a list of boolean values, then `and/p` returns `true` if all of the elements of `p` are `true`, and `false` otherwise.
- **Each**. The prefix operator `!` means *each*, e.g. `absval ! u` applies the function `absval` to *each* element of the list `u`. Likewise, `and/(x > !u)` returns `true` if `x` is greater than *each* value of `u`. If more than one list arguments to a function is given a `!` prefix, the arguments are iterated over in parallel, e.g. `min(!u, !v)` is short for
  ```
  [min(x, y) for x in u for y in v]
  ```
  and returns a list of *each* minimum pair of corresponding values from `u` and `v`.

Juxtaposition

The meaning of juxtaposition is delayed until evaluation or compile time. When AXIOM or user-defined names are used as left-hand arguments, juxtaposition means application. Otherwise, juxtaposition means multiplication. This convention allows users to write expressions mostly without parentheses, e.g. `n cos x log a b x` is equivalent to `n*cos(x)*log(a*b*x)`. Parentheses can always be used for clarity. Juxtaposition associates to the right, e.g. `f g x` is equivalent to `f(g(x))`. The infix dot (.) operator always means application and associates to the left, e.g. `f.g.x` is equivalent to `((f.g).x)`.

Afterthoughts

Every user command in $B^\natural$ has the general form `eval(e, ...)`. The function `eval` takes one or more arguments. The first argument `e` is mandatory: it is the expression to evaluate. All remaining arguments are optional: they indicate how `e` is to be transformed during evaluation.

This convention makes every user command in $B^\natural$ take the form:

expression, after-thought1, after-thought2, ...

After-thoughts have the form of rules or lists of rewrite-rules that are used to transform the (left-most) expression.

Here is a sample conversation with afterthoughts, using the notion of rewrite-rules discussed more fully below.

```
u == sum_{i in 0..n}a_i x^i
u, x == t
u, a_i == sum_{j in 0..n | j ^= i}a_{i,j} y^j
```

The simplest form of a rewrite-rule is one where the left-hand side is a variable.The first line is a rewrite-rule telling AXIOM how to compute the value of u: "u is computed by evaluating the right-hand side sum." The second line asks for that sum to be evaluated, but with a temporary definition for x. The result is $\sum_{i=0}^{n} a_i t^i$. The third line evaluates u, instead with a temporary definition for a_i, producing

$$\sum_{i \in 0..n} \sum_{j \in 0..i \mid j \neq i} a_{i,j} x^i y^j$$

Pattern-matching

Our pattern-matching conventions follow those of Scratchpad and AXIOM, and seem to us to be simpler and more natural yet provide equivalent function as those in M&M.

Patterns are expressions that appear on the left-hand side of rewrite rules $L == R$, where L is a pattern and R is an expression, or in "rulesets" (see below). An example use of a rule is to give a function definition:

$f(x) == y$ meaning "for all x, rewrite f of x by y"

Identifiers in patterns infrequently need to be prefixed by either a quote-mark (`'`), an underscore (`_`), or question-mark (`?`). The pattern `'x` matches the symbol `x` and none other. The pattern `?x` marks an *optional* expression as described below. Here are the $B^\natural$ rules for forming patterns.

- If a pattern is a single identifier, the identifier is implicitly quoted. The pattern `x` matches the symbol `x` but not `y` or any other expression. Thus `x == y` means "rewrite the symbol `x` by `y`";
- An operator name is implicitly quoted. Arguments are not and, without further qualification, match any value. Thus `f(x)` matches `f(a + 1)` but not `g(a)` or `f(a,b)`. On rare occasions when it is necessary to let an operator match any name, a prefix underscore (`_`) is given before the name. Thus the pattern `_f(x)` matches all functions of one argument, for example, `u(x)` and `v(y)` but not `u(x,y)`.
- Literals such as numbers, strings, and constants (for example, `true`, `false`, and `[]`) match only identical literals. The pattern `f(0)` matches `f(0)` but not `g(0)`, `f(0.0)` or any other. Also `f("joe",true)` matches `f("joe",true)` and none other.
- An exception to the last convention occurs when `f` has the property `commutative` in which case `f("joe", 1)` also matches `f(1, "joe")`. In general, symbols may have "properties" as in Mathematica that affect matching.
- Repeated variables in patterns indicate equal values: the pattern `f(x,x)` matches `f(0,0)` but not `f(0)` or `g(0,0)`.
- Operations `[]`, `+` and `*` take multiple arguments. Pattern variables prefixed by a colon (`:`) match a sequence of 1 or more expressions. For example, `x + :y` matches `a + b` with `x=a` and `y=b`, `a + b + c` with `x=a` and `y=b + c`, in general, any sum of two or more terms. The pattern `[a,:b]` matches any list of length 1 or more with `a` matching its `first` element and `b`, the `rest` of the list.[16]
- Functions with "default" arguments use a pattern variable with prefix `?` to match an optional argument value. For example, `+` has default value 0, and so the pattern `x + :?y` matches any expression possibly with `y=0`.

[16] Only one colon (:) is allowed at a given level in patterns. For example, the pattern [:x,a,:y] is not allowed.

- Patterns are qualified by the use of vertical bar (meaning "such that") followed by a predicate, e.g. `'x + :?a | freeOf?(a,x)` matches any expression of the form `x + a` such that `a` (possibly 0) does not contain `x`.

Pattern-matching always reduces to expressions of the form `x is p | pred` where `x` is a variable, `p` is a pattern, and `pred` is a predicate. The function `isLinearIn` tests if an expression `e` is linear in `x`:

```
isLinearIn(e,x) == freeOf?(e,x) or
   (e is x*:?a + :?b | freeOf?(a,x) and freeOf?(b,x))
```

An **is** predicate either returns `false` or else a local ruleset of "bindings" for pattern variables. The function `pal` tests if a list `x` is a palindrome (that is, `reverse(x) = x`).

```
pal(u) == {
  #u < 2          => true;
  u is [x,:v,x]   => pal v;
  false;
}
```

5.3 7 PARADIGMS

Wolfram remarks in [DDoJ93] that a language supporting a number of different programming paradigms is useful for computer algebra. We agree. By making $B^\natural$ independent of $A^\sharp$, AXIOM is able to fully support 7 paradigms:

Rules

(AXIOM, SMP, and Mathematica). With this paradigm, `==` is always used to associate a value with a variable. A program consists of a set of rewrite-rules. A simple example is a piecewise definition of a recurrence relation.

```
p_0 == 1
p_1 == x
p_n == x*p_{n-1} - n/2*p_{n-2} when n > 1
```

Each rule has the general form $L == R$ if p which means "replace L by R if p".[17] Rewrite-rules are "lazy": no computation is performed until you demand a "value" by issuing an expression to the interpreter.

> The value of a given expression is obtained by scanning the rules in search of an applicable rule. If one is found, the rule is applied to obtain a new expression. This process is repeated again and again until no changes are possible. The resulting expression produced is called the *value* of the given expression.

For example,

```
p_n for n in 1..5
```

asks for the first five values of p_i. The last of these is

$$p_5 = x^5 - 7x^3 + \frac{33}{5}x$$

[17]The first of the above three rules means, for example, "replace p_n by 1 if $n = 0$".

If none of the rules overlap, as is the case above, their order doesn't matter and program steps can be shuffled with no effect. If one or more rules do overlap, newer rules take precedence. The only exception is a rule of the special form $L == R$ **otherwise** which is given least precedence.

A user's interactive session consists of the user entering rules, asking for values, entering new rules, asking for more values, and so on. Every value is produced by scanning the rules that exist *at that time.*

It is often convenient to introduce a temporary rule as an afterthought to cause substitution. For example,

```
p_5, x == 1/2
```

introduces a *temporary* rule for x whereby x is replaced by 1/2 producing the result $\frac{105}{32}$.

Assignments

(C, FORTRAN, and most other languages). With this paradigm, expressions use `:=` (assignment) rather than `==` (rewrite) to associate values with symbols. Unlike rewrite-rules where the right-hand side is evaluated each time you ask for a value, the right-hand side of an assignment is evaluated immediately and never again. If `b` has no assigned value, then `a := b + 1` assigns to `a` the symbolic value `b + 1`. To replace the value of `b` by 2, the user must issue `substitute(2,b,a)`. The command `a, b := 2` won't work since the value of `a` remains `b + 1` until a new assignment is given for `a`.

To define a recurrence relation analogous to p_i above, an assignment is made to a parameterized form.

```
q_0 := 1
q_1 := x
q_n := x q_{n-1} - n/2 q_{n-2} for n in 2..5
```

These statements define a "table" of values for $q_\bullet$ as in M&M. Each value of q_i is stored under the key i and is accessible by evaluating q_i. Like variable assignments using `:=`, the value of the right-hand side is computed immediately when the assignment is made. Then `substitute(1/2, x, q_5`, not `q_5, x := 1/2`, is used to substitute the value of 1/2 for `x` in the value of q_5.

Procedures

(C, FORTRAN, and most other languages). Procedural programming is by far the most common programming paradigm used today. Most all programming languages provide constructs such as blocks, conditionals, and iterations found in C for writing programs. So does $B^\natural$.

The procedural definition of r_n below is semantically equivalent to that of p_n:

```
r_n == {
   n = 0 => 1;
   n = 1 => x;
   x r_{n-1} - n/2 r_{n-2}
}
```

The *block* `{a; b; ... ;c}` means "do `a` then do `b` then ... do `c`"; the *conditional statement* `a => b` means "if `a` then return the value of `b`". When a value of r_n is requested, the right-hand side is rewritten to produce a value (see section 5.7).

An alternate way to define r_n is via a **cases** statement:

```
r_n == cases {
   0 if n = 0;
   1 if n = 1;
   x r_{n-1} - n/2 r_{n-2}} otherwise
}
```

This notation directly corresponds to a TeX construct which we use as the preferred way of displaying piecewise definitions:

$$r_n == \begin{cases} 0 & \text{if } n = 0; \\ 1 & \text{if } n = 1; \\ xr_{n-1} - n/2r_{n-2} & \text{otherwise} \end{cases}$$

The rewrite-rule

```
take(n,u) == [u i for i in 1..n]
```

defines **take**(n, u) to return the first $n \geq 0$ elements of a list u.[18] An alternate and more efficient recursive procedural definition shows how patterns can be expressed by conditional expressions of the form `x is p` where `x` is an identifier and `p` is a pattern.

```
take(n,u) == {
   n > 0 and u is [x,:y] => [x,:take(n - 1,y)];
   []
}
```

An equivalent piecewise definition illustrates an alternate syntax for giving conditions for pattern variables.

```
take(n | n > 0,[x,:y]) == [x,:take(n - 1,y)]
take(n,u)              == [] otherwise
```

Pattern-matching

(REDUCE, Scratchpad, and M&M). Pattern-match programming in computer algebra was introduced by REDUCE, featured in Scratchpad, later commercialized by Mathematica. Rulesets are expressed as lists of rewrite-rules. They give users rather complete control over how and when pattern-matching is done.[19] Here are two examples:

```
logRules := rules
   [log x + log y = log x y
    y log x       = log x^y]

trigRules := rules
   [sin x sin y = (cos(x-y) - cos(x+y))/2,
    sin x cos y = (sin(x-y) + sin(x+y))/2]
```

Rulesets behave as functions that are applied to expressions to make transformations. For example, `trigRules u` applies ruleset `trigRules` *once* to u applying each rule of the ruleset successively at each node. If a match occurs, the indicated substitution is made and the transformed result is returned. The notation `trigRules* u` applies the ruleset *repeatedly* until the result no longer changes. Similarly, the command

```
u, logRules*, trigRules
```

first evaluates `u`, then applies `logRules` repeatedly, and finally applies `trigRules` once.

Equations or lists of equations given as qualifiers are converted to anonymous rulesets, e.g. `q_5, x = 1/2` has the same meaning as `q_5, [x == 1/2]`. The operation "solve" for a system of equations returns a list of solutions, each a list of equations. When used as qualifiers, this list of equations produces a list of values as in Mathematica and MACSYMA. For example,

```
solve([x^2 - 2y^2 - 19, x y - y - 5x + 5],0.01)
[x,y], %
```

returns the value

```
[[5.0, -8.30859375], [5.0, 8.30859375]]
```

Left-to-right

(SML [MiTH90], Haskell [HuJo92], C++, Smalltalk [GoRo83]). In a left-to-right programming language, expressions are evaluated from left-to-right exactly as they are entered interactively. Left-to-right expressions in $B^\natural$ are constructed using infix operator dot (.).

This programming style is supported directly by $A^\sharp$ as new types can be created having exported operations designed for left-to-right use. The following command illustrates this style for drawing a circle of radius 10 at location (50, 20) inside a window:

```
cursor.forward(50).right(20).draw(circle,10)
```

This style allows users to "think left-to-right" as they enter commands from a keyboard: "Start with the cursor. Move forward 50 units. Then right 20 units. Now draw a circle of radius 10." Compare the naturalness of the above expression with that of the more conventional programming style that uses a right-to-left (or "depth-first") syntax:

```
draw(circle,10,right(forward(cursor,50),20))
```

Functional

(SML, Haskell, and others). The function programming paradigm is enabled by $A^\sharp$'s handling of functions as first-class objects. In functional programming, functions typically take one argument and produce other functions as values. For example, suppose **plus** has type $S \rightarrow S \rightarrow S$. Evaluating *((***plus** *1) 2)* involves first applying **plus** to 1 producing a function that is then applied to 2 to produce 3. In Haskell, juxtaposition means composition with association to the left. $B^\natural$ syntax for function definitions is similar to Haskell except that dot is used for application. A favorite example operation of Haskell is **fold**: $((S, S) \rightarrow S) \rightarrow S \rightarrow$ *List* $S \rightarrow S$ defined in $B^\natural$ as follows:

```
fold.op.init []      == init
fold.op.init [x,:xs] == op(x,fold.op.init xs)
```

Two example uses of **fold** are **sum** to sum the elements of a list and **append** to join two lists end-to-end.

```
sum        == fold.+.0
append.u v == fold.cons.v u
```

[18] The application u i to a list u to an index i returns the ith element of u. A more efficient iterative definition is [x for x in u for i in 1..n] which counts and collects elements simultaneously.

[19] Matching on expressions however is always done "inside-out" in contrast to logic programming that does *resolution* matching from "outside-in". An interesting challenge is to smoothly integrate PROLOG-like matching rules within AXIOM. We have not done this.

Types

(AXIOM). This style refers to that used in current AXIOM. The simplest way of accessing other AXIOM or $A^\sharp$ types is via $B^\natural$ "constructors". For example, **matrix [[a,b],[c,d]]** builds a 2 by 2 matrix of elements of type User. To break out of type User, a user simply makes type declarations. For example, the following is a display of a $B^\natural$ definition for Laguerre polynomials $L_n^a(x)$ in TEX:

$$
\begin{array}{l}
n : \text{NonNegativeInteger} \\
a : \text{FractionInteger} \mid a > -1 \\
L_n^a(x) == \begin{cases} 1 & \text{if } n = 0; \\ -x + 1 + a & \text{if } n = 1; \\ \frac{2n+a-1-x}{n} L_{n-1}^a(x) - \frac{n+a-1}{n} L_{n-2}^n(x) & \text{otherwise} \end{cases}
\end{array}
$$

By omitting the type of **x** in the above definition, type inferencing is used to determine the full signature of the function $L_\bullet^\bullet$ when the user asks for a value with given arguments. This may result in multiple compilations of $L_\bullet^\bullet$ if the function is called with different types of arguments.

Alternately, a user may declare the type of $L_\bullet^\bullet$ rather than that of its arguments **n** and **a**, e.g.

$$L_\bullet^\bullet : (\text{NNI}, \text{RF}, \text{RF}) \to \text{RF}$$

where **NNI** and **RF** are used here to abbreviate the AXIOM types **NonNegativeInteger** and **Fraction Polynomial Integer** respectively.

5.4 SEMANTIC MODEL

The semantic model for a computer algebra system should be simple, consistent, and flexible. Our notion of property lists and evaluation is based on LISP.

Property Lists

All user variables have a *property list*, a list of name-value pairs. The AXIOM interpreter uses property lists to accumulate information for each user variable. User code and library code alike can use this facility to query properties of user-defined names. Three important properties are:

type This is User by default but may be any AXIOM type. This property is set for a symbol x by a declaration, e.g. **x: Integer**.

value This property is set for a symbol x by evaluating an assignment or rewrite-rule definition for x. The **value** property of a symbol is always consistent with the **type** and **predicate**.

predicate The predicate for a symbol x describes conditions that **values** of x must satisfy in terms of operations on its **type**.

Functions **get** and **put** are used to read and write properties. Putting the property **cache** on a function name f causes values computed for f to be "remembered" as in Maple.

Unlike CommonLISP, a variable cannot simultaneously denote a function and a variable: a symbol can have at most one value as given by its **value** property. The rewrite-rule `f(x) == x + 1` gives a function **value** to f equivalent to that defined by `f == x +-> x + 1`. Piecewise definitions are converted to procedures and treated similarly.

Evaluation

Evaluation is the universal mechanism for producing output results from input forms. Evaluation in $B^\natural$ corresponds to that of REDUCE, Scratchpad, and Common LISP, using symbols and their property lists.

Here now are the rules for producing μf, the *value* of f:

- If f is a constant (a number, string, a symbol with no binding, or a mapping form), $\mu f \longrightarrow f$ itself.
- If f has the quoted form $'g$, $\mu f \longrightarrow g$. If f has the doubly-quoted form $''g$, $\mu f \longrightarrow f$.[20]

μf is otherwise defined recursively as follows.

- If f is a variable with bound value g, then $\mu f \longrightarrow \mu g$.
- If f is not a special form, it has the form $g(a, ..)$ and $\mu f \longrightarrow \texttt{apply}(\mu g, [\mu a, ..])$. As to special forms, $\mu(g\texttt{==}S)$ gives g the binding S (S remains unevaluated) whereas $\mu(g\texttt{:=}\ S)$ gives g the quoted binding $'\mu S$ (S is immediately evaluated).
- All pattern-matching reduces to evaluation of expressions of the form e **is** p where p is a pattern that in general has an accompanying predicate on the variables. Evaluation of an **is** expression produces either false or else a ruleset giving values for the pattern variables in p.

Except for other special forms such as blocks, iterators, and conditions, the above cases completely define evaluation.

6 COMPARISONS

The design of $B^\natural$ resembles M&M which both use use ideas originating in the earlier systems MACSYMA, REDUCE, and Scratchpad. The notion of afterthoughts was first used in Scratchpad and MACSYMA. The notion of rulesets comes from Scratchpad and is similar to that used in Mathematica. A major difference is the syntax used in writing pattern-match rules that seem to us to read more like mathematics that those of Mathematica.

The approach of $B^\natural$ is similar to that of GAUSS [Mona94b] but with opposite aims. GAUSS's goal was to build an AXIOM-like facility to create domains and categories starting with MAPLE. $B^\natural$'s goal is to build a MAPLE-like typeless interface starting with AXIOM. As all GAUSS domains provide coercions to MAPLE with partial coercions the other way, so do all AXIOM domains provide analogous coerces to type User.

As GAUSS is written on top of MAPLE, however, there is no means for static type-checking. Since type User is written on top of AXIOM however, type-inferencing [JeSu87] can be used to statically type-check and compile user programs as in the current AXIOM interpreter.

[20]This concerns the "noun/verb" question that has perplexed system designers throughout history [MosJ71]: if a user enters *int*, does it mean "do it" (*int* is a *verb*) or not (*int* is a *noun*). MAPLE makes "sum" a verb and "Sum" a noun. In $B^\natural$, a single quoted expression always evaluates to itself. Thus **'integrate(y,x)** produces the formal integral $\int_x y$. A second evaluation of a singly-quoted form causes it to evaluate normally. Doubly-quoted forms are "locked" against evaluation and must be *unlocked* to evaluate normally.

References

[AviR94] Avitzur, R., private communication

[BrnS73] Brown, W.S., **ALTRAN Users Manual**, Bell Laboratories, Murray Hill, 1973.

[BucB91] Buchberger, B., Groebner Bases in Mathematica: Enthusiasm and Frustration, Technical Report, RISC-Linz Series no. 91-11, 1991.

[CanJ84] Cannon, J.J., An Introduction to the Group Theory Language CAYLEY, in Computational Group Theory, ed. M. Atkinson, Academic Press NY 1984.

[CoWo81] Cole, C.A. and Wolfram, S., SMP: A Symbolic Manipulation Program, *Proceedings of SYMSAC '81*, P. Wang, ed., ACM 1981.

[CGG91a] Char, B.W., Geddes, K.O., Gonnet, G.H., Leong, B.L., Monagan, M.B., and Watt, S.M., **Maple V Library Reference Manual**, Springer-Verlag, 1991

[CGG91b] Char, B.W., Geddes, K.O., Gonnet, G.H., Leong, B.L., Monagan, M.B., and Watt, S.M., **Maple V Language Reference Manual**, Springer-Verlag, 1992

[DDoJ93] The Multi-Paradigm Man, interview with S. Wolfram, Dr. Dobbs Journal, 1993.

[GoRo83] Goldberg, A. and Robson, D., **Smalltalk-80: The Language and its Implementation**, Addison-Wesley, 1983.

[GrMo93] Gruntz, D. and Monagan, M., Introduction to Gauss, *MapleTech: The Maple Technical Newsletter*, Issue 9, Spring 1993, pp. 23-45.

[GruD93] Gruntz, D., Groebner Bases in Gauss, *MapleTech: The Maple Technical Newsletter*, Issue 9, Spring 1993, pp. 36-46.

[FaIv68] Falkoff, A.D., and Iverson, K.E., **APL/360 User's Manual**, IBM Thomas J. Watson Research Center, 1968.

[GrJe71] Griesmer, J.H. and Jenks, R.D., SCRATCHPAD/1: An interactive facility for symbolic mathematics, *Proceedings of the Second Symposium for Symbolic and Algebraic Manipulation*, S.R. Petrick, Ed., ACM, 1971.

[HrnT71] Hearn, A.C., Reduce 2, A System and Language for Algebraic Manipulation, *Proceedings of the Second Symposium for Symbolic and Algebraic Manipulation*, S.R. Petrick, Editor, ACM, 1971.

[HuJo92] Hudak, P., Jones, S. P., Wadler, P., et al, Report on the Programming Language Haskell: A Non-strict, Purely Functional Language, Version 1.2, March 1, 1992.

[JenR79] Jenks, R. D. MODLISP: An Introduction, **EUROSAM '79, Lecture Notes in Computer Science**, #72, G. Goos and J. Hartmanis, Editors, Springer-Verlag, NY, 1979.

[JeSu87] Jenks, R. D. and Sutor, R. S., The Type Inference and Coercion Facilities in the Scratchpad Interpreter, *Proceedings of the SIGPLAN '87 Symposium on Interpreters and Interpretive Techniques*, SIGPLAN Notices 22, 7, pp. 56-63.

[JeSu92] Jenks, R.D., and Sutor, R.S., **AXIOM: The Scientific Computation System**, Springer-Verlag and NAG, Ltd, 1992.

[JeTr81] Jenks, R.D., and Trager, B.M., A Language for Computational Algebra, *Proceedings of SYMSAC '81*, ACM, 1981 (also published in SIGPLAN Notices, November 1981 and IBM Research Report RC 8930)

[LiGu77] Liskov, B. and Guttag, J., **Abstraction and Specification in Program Development**, MIT Press, 1986.

[LosR74] Loos, R., Toward a Formal Implementation of Computer Algebra, *Proceedings of EUROSAM '74*, SIGSAM Bulletin, Vol. 8, Number 3, 1974.

[MaFa71] Martin, W.A. and Fateman, R.J., The MACSYMA System, *Proceedings of the Second Symposium for Symbolic and Algebraic Manipulation*, S.R. Petrick, Editor, ACM, 1971.

[McCa60] McCarthy, J., et al, **LISP 1.5 Programmers Manual**, Cambridgy, MA, The MIT Press, 1965.

[MiTH90] Milner, R., Tofte, M., and Harper, R., The Definition of Standard ML, MIT Press, 1990.

[MonM94a] Monagan, M., Gauss: a Parameterized Domains of Computation System with Support for Signature Functions, *Proceedings of DISCO '93.*

[MonM94b] Monagan, M., private communication

[MosJ71] Moses, J., Algebraic Simplification: A Guide for the Perplexed, *Proceedings of the Second Symposium for Symbolic and Algebraic Manipulation*, S.R. Petrick, Editor, ACM, 1971.

[SteG84] Steele, G.L. Jr., **Common LISP: The Language**, Digital Press, 1984

[StrB92] Stroustrup, B., **The C++ Programming Language**, Second Edition, Addison-Wesley, April, 1992.

[Vor94] Vorkoetter, S.M., OpenMath: Preliminary Report, Waterloo Maple Software, 1994.

[WaDo94] Watt, S. M., Dooley, S. S., Morrison, S. C., Steinbach, J. M., and Sutor, R. S., $A^\sharp$ User's Guide, 1993.

[WaBD94a] Watt, S.M., Broadbery, P.A., Dooley, S.S., Iglio, P., Morrison, S.C., Steinbach, J.M., Sutor, R.S., A First Report on the $A^\sharp$ Compiler, this proceedings.

[WaBD94b] Watt, S.M., Broadbery, P.A., Dooley, S.S., Iglio, P., Morrison, S.C., Steinbach, J.M., Sutor, R.S., $A^\sharp$ User's Guide, v35.0, NAG Ltd, 1994.

[WaSM94] Watt, S. M., Steinbach, J. M., Morrison, S. C., and Broadbery, P.A. FOAM: A First Order Abstract Machine, v3.5, IBM Research Report RC 19528, 1994.

[WolS92] Wolfram, S., **Mathematica: A System for Doing Mathematics By Computer** Second Edition, Addison-Wesley, 1991.