UNIVERSITÄT KAISERSLAUTERN
Zentrum für Computeralgebra

REPORTS ON COMPUTER ALGEBRA
NO. 10

A proposal for syntactic data integration for math protocols

by

O. Bachmann, S. Gray, and H. Schönemann

January 1997

The Zentrum für Computeralgebra (Centre for Computer Algebra) at the University of Kaiserslautern was founded in June 1993 by the Ministerium für Wissenschaft und Weiterbildung in Rheinland-Pfalz (Ministry of Science and Education of the state of Rheinland-Pfalz). The centre is a scientific institution of the departments of **Mathematics, Computer Science, and Electrical Engineering** at the University of Kaiserslautern.
The goals of the centre are to advance and to support the use of Computer Algebra in industry, research, and teaching. More concrete goals of the centre include

- the development, integration, and use of software for Computer Algebra
- the development of curricula in Computer Algebra under special consideration of interdisciplinary aspects
- the realisation of seminars about Computer Algebra
- the cooperation with other centres and institutions which have similar goals

The present coordinator of the Reports on Computer Algebra is:
Olaf Bachmann (email: `obachman@mathematik.uni-kl.de`)

Zentrum für Computeralgebra
c/o Prof. Dr. G.-M. Greuel, FB Mathematik
Erwin-Schrödinger-Strasse
D-67663 Kaiserslautern; Germany
Phone: 49 - 631/205-2850 Fax: 49 - 631/205-5052
email: `greuel@mathematik.uni-kl.de`
URL: `http://www.mathematik.uni-kl.de/~zca/`

A Proposal for Syntactic Data Integration for Math Protocols*

Olaf Bachmann Hans Schönemann

Centre for Computer Algebra
Department of Mathematics
University of Kaiserslautern
Kaiserslautern, Germany
{obachman, hannes}@mathematik.uni-kl.de

Simon Gray

Department of Mathematics and Computer Science
Kent State University
Kent, OH 44242, USA
sgray@mcs.kent.edu

Abstract

The problem of providing connectivity for a collection of applications is largely one of data integration: the communicating parties must agree on the semantics and syntax of the data being exchanged. In earlier papers [15, 5], it was proposed that dictionaries of definitions for operators, functions, and symbolic constants can effectively address the problem of semantic data integration. In this paper we extend that earlier work to discuss the important issues in data integration at the syntactic level and propose a set of solutions that are both general, supporting a wide range of data objects with typing information, and efficient, supporting fast transmission and parsing.

1 Introduction

It is widely accepted that providing connectivity between software tools, in general, and between Computer Algebra and other mathematical software systems, in particular, is an inevitable development. Among others, the need for software tool integration can be seen from several recent developments. First, the design of distributed Problem Solving Environments (PSEs) that connect symbolic, numeric, text processing, and visualization packages. There has been increased research in this area lately, with workshops and white papers (see, for example, [12, 22]), a NSF-funded project [21], as well as explorations of the use of the Web for building wide-area, loosely-coupled systems [8, 24]. Second, with the maturity of tightly-coupled parallel machines and message passing systems such as PVM [13] and MPI [19], there is renewed interest in the development and implementation of parallel algorithms for symbolic computation (see, for example, [9, 20, 4, 3]). Finally, there is an increasing need to extend the capabilities of a system through a *communication link* with one or more other systems (see, for example, [7, 11]). The "communication" approach has several well-known advantages over the more traditional approach of linking in additional modules to produce a single executable image.

Several groups are currently working on different aspects of protocols for exchanging mathematical data (MathLink [23], ASAP [10], Multi Protocol (MP) [14, 15], PoSSoXDR [1], OpenMath [2], and MathBus [25]). The central problem facing these efforts is that of data integration: Mathematical objects have a semantics and a syntax which must be shared between communicating partners.

Semantic data integration requires a shared understanding of the meaning of mathematical data. Until recently, math protocols provided no support for shared semantics beyond the meaning of the primitive data types and simply assumed that the communicating partners "knew" each other. An important task of the Computer Algebra community is to close this semantic gap. Several initiatives addressing this problem are underway (MP, OpenMath, MathBus) and we hope that more experience and a careful evaluation of the proposals will lead to a unifying solution.

Syntactic data integration requires a shared understanding of the encoding and structure of the objects exchanged. At first glance, achieving syntactic data integration might seem easier and, therefore, somewhat less worthy of discussion than semantics. However, our experience [15, 5] is that this problem becomes increasingly non-trivial if one requires the protocol to be general as well as efficient.

Briefly, generality refers to the applicability of the protocol to mathematical software systems and efficiency refers to the productivity of data communications managed by the protocol.

This paper concentrates on the syntactic aspects of data integration, but we recognize that the syntactic and semantic sides of data integration are not mutually exclusive. We outline the major issues that must be addressed if syntactic data integration is to be achieved in a general and efficient manner and propose solutions which have worked in practice.

1.1 Syntactic data integration

We distinguish two levels of syntactic data integration. At the lowest level is the set of basic (atomic) types supported (such as numbers, strings, identifiers, symbolic constants, etc.) and their encoding. At a higher level are the rules which govern the composition of basic types to produce structured objects.

Syntactic generality implies that the basic types are encoded using a common, well-defined format applicable to a majority of mathematical software systems, and that the rules governing the composition of structured data are relatively simple and result in *dynamically typed* data, i.e. the representation of an object includes sufficient type information to unambiguously decode it without any additional information. Examples of syntactically general math protocols

*Work reported herein has been supported in part by the National Science Foundation under Grant CCR-9503650, by the Deutsche Forschungsgemeinschaft, and by the Stiftung Innovation des Landes Rheinland-Pfalz.

include MathLink and ASAP.

Syntactic efficiency implies that syntactic transformations (intermediate transformations between different representations of the same object, e.g., conversions between different encodings – ascii, hex, binary – for numeric data, transformations between different formats for arbitrary precision numbers, or transformations between sparse and dense matrices or polynomial representations) and the amount of "overhead" data (e.g., type information) are minimal. Consequently, if the encodings of basic and structured objects are close to the internal representation of the target system(s), and if the rules governing the composition of structured data are object specific and *statically* defined (e.g., in a document and hardcoded into the interface code), then efficient syntactic data integration is easily accomplished. Examples of syntactically efficient math protocols include PoSSoXDR and the protocol used for communications between Macaulay2's front end and its compute engine [17].

The discussion above suggests a tension between generality and efficiency. Among others, generality implies that explicit type and syntactic information be provided *with* the object, while efficiency implies that the responsibility for supplying this information be shifted from the object itself to the interface code that will parse it. More simply, until now, the more explicit the type information, the more general, but less efficient, the data integration, and vice versa.

Is it important to have a protocol that provides general *and* efficient syntactic data integration? Yes! Often, mathematical software systems require communication facilities that are general for some applications and efficient for others. Consider the CAS SINGULAR [18], for example. If SINGULAR is to be widely used as a component in a PSE to provide specialized and state-of-the-art standard bases computations or if SINGULAR requires the services of another CAS, then we clearly need a general protocol – being able to easily establish basic connectivity is of upmost importance for this purpose. On the other hand, there is the desire to use SINGULAR-based parallel and distributed computations to expedite the execution of (time-) complex algorithms. So, there is a real need for an efficient protocol, since for parallel, and particularly distributed parallel, computations, the protocol's efficiency affects overall computation time as well as the grain size that can be supported. Without both, SINGULAR, as well as many other systems, would be required to have separate interfaces for general and efficient communications, which is clearly an undesirable duplication of development effort.

The next question follows naturally: Is it possible to have both? As we will show, the answer is *yes*! The main idea behind our solution is to provide mechanisms which implement the design philosophy of "making the common case fast"; that is, support efficient data integration of "common case" data. On the lower level of basic object encodings, we realize this through negotiations of the basic data formats and on the higher level of structure encodings, we use a flexible and expressive data (type) description mechanism which supports very compact and natural encodings of a standard set of common data structures.

The next section introduces MP and discusses the problem of general and efficient data integration within the context of MP. Sections 3 and 4 give details of our solutions for low-level basic object encodings and high-level structure encodings, respectively. Section 5 introduces a general and efficient MP parser which is based on the mechanisms described in sections 3 and 4 and whose goal is to facilitate the development of MP interfaces for software systems. Finally, the last section summarizes our work.

2 The Multi Protocol

The purpose of the Multi Protocol is to support general and efficient communication of mathematical data among scientific computing systems. MP defines a set of basic types and mechanisms for constructing structured data. Numeric data (fixed and arbitrary precision floats and integers) are transmitted in a binary format (2's complement, IEEE float, etc.). Composite data (such as general expressions, polynomials, matrices, etc.) are represented as a linearized, annotated syntax tree (*MP tree*) which is transmitted as a sequence of *node* and *annotation packets*, where each node packet transmits a node from the syntax tree. The node packet has fields giving the *type* of the data carried in the packet, the number of arguments (for operators) that follow, a dictionary tag, the number of annotations attached to the node.

A *data packet* is a block of data unemcumbered by node packet headers and may refer to a single data item or a collection of, possibly heterogeneous, data items.

Annotations carry additional information which is either supplementary and can be safely ignored by the receiver, or may contain information essential to the proper decoding of the data. Each annotation is tagged in such a way that the receiver always knows whether it can safely ignore the annotation content or not.

In a layer above the data exchange portion of the protocol, MP supports collections of definitions for annotations and mathematical symbols (operators and symbolic constants) in *dictionaries*. Dictionaries address the problem of data integration by defining standardized representation(s) and semantics for mathematical objects. They are identified within packets through a dictionary tag field. Applications that communicate according to definitions provided in dictionaries do not need to have direct knowledge of each other.

Applications send ("Put" interface) and receive ("Get" interface) *messages*, each containing a MP tree which is (typically) created by calling routines from the MP Application Programming Interface (API). An application communicates with other applications through a *MP link*, which is simply an abstraction of an underlying data transport mechanism that is bound to the link at the time of its creation.

If MP's major goal were generality, then this goal would be achieved best by defining one standard representation for each basic and structured object and by requiring that each datum be communicated as a node packet (i.e., each datum is communicated together with type and other information). Consequently, a 100x100 integer matrix A which has zero entries everywhere except for $A[1,1] = A[2,1] = A[2,4] = 1$ should be communicated as a "array of array of integer node packets" as shown in 1. Notice that of the total 804,004 bytes communicated, 404,004 are used for node headers and that of the 400,000 data bytes communicated, only 12 are non-null – certainly not a very efficient format to communicate A.

On the other hand, if MP's major goal were efficiency, then this goal would be achieved best by having a large variety of statically defined representations for basic and structured objects. Consequently the matrix A would be communicated as a "list of integer data packets" as shown

Figure 1: A encoded as general expression tree matrix

Type	Dict	Value	#Arg:Annot	Remarks
COP	Basic	Array	100:0	
COP	Basic	Array	100:0	1st row
MP_Sint32		1		[1,1]=1
MP_Sint32		0		[1,2]=0
		....		
COP	Basic	Array	100:0	2nd row
MP_Sint32		1		[2,1]=1
MP_Sint32		0		[2,2]=0
MP_Sint32		0		[2,3]=0
MP_Sint32		1		[2,4]=1
		....		

Figure 2: A encoded as statically defined sparse matrix

Type	Dict	Value	#Arg:Annot	Remarks
COP	Matrix	SpIntMat	3:0	3 non-zero entries
		100		1st dim
		100		2nd dim
		1		
		1		
		1		[1,1] = 1
		2		
		1		
		1		[2,1] = 1
		2		
		4		
		1		[2,4] = 1

in Figure 2. Notice that it only takes 48 bytes to encode A. Furthermore, notice that a correct decoding of A presupposes explicit (i.e., static) knowledge that the operator `SpIntMat` is followed by integer data packets (i.e., headerless integers) specifying the dimensions and non-zero entries of A – certainly not a very *general* format to communicate A.

Although somewhat simple, these examples illustrate important aspects of the tension of generality and efficiency within the context of MP. In the next two sections we will present our solutions to this problem: a restricted form of negotiation of basic data formats and powerful mechanisms to dynamically describe the type and structure of homogenous data using the `prototype` annotation. The main ideas behind these principles can be found in the first papers about MP [14, 15], and in [5] we focused on solutions of the generality versus efficiency problem within the context of communicating polynomial data. Realizing that this problem is far more generally applicable and complex than in [5] led to the research and results described in this paper.

3 MP low level data integration – Negotiations

On its lowest level, MP communicates data as unsigned 32 bit integers in a 2'complement binary representation. However, already on this lowest level, one has to make an encoding choice: Should integers be communicated in Big Endian encoding (also called network byte order) or in Little Endian encoding? Fixing the "endianness" (say, to network byte order as XDR does) supports generality, but introduces unnecessary computational overhead if the two communicating parties are Little Endian.

Similar problems occur for the communication of fixed precision floating point numbers (possible encoding: IEEE, Cray, VAX, etc.), and arbitrary precision integers (GnuMP, PARI, SacLIB, BigNum, etc.) and floating point numbers (GnuMP, PARI): again, restricting the communication to one format supports generality, but significantly and unnecessarily decreases efficiency if both sides use the same non-MP format.

The solution we propose is based on a restricted form of format negotiation. The main idea is the following: At link creation time, a MP application writes a record to the link, which, for each of the critical basic data types, specifies a list of encodings which the particular application can handle, with a score in the range of 0 to 255 reflecting the efficiency with which an encoding can be handled – the native encoding is typically assigned the highest score. Furthermore, a record of the same type is read from the link containing the peer's preferences. Based on these two records, the encoding of each of the basic data types is independently and uniquely determined based on the following rule:

> Choose the encoding whose sum of the scores of both records is maximal. In case of a tie, use a statically defined order as tie-breaker.

Furthermore, each list must contain at least the MP-defined default member, which guarantees that it is always possible to find at least one "common" encoding for communication. In addition, if such an exchange of format-specifying records is impossible for a particular type of MP link (e.g., for a write-only link to a file) then the default encoding is used.

Table 3 shows the currently supported encodings of numeric MP types. We might add that these particular encodings were motivated by our practical needs and could easily be extended to include other encodings.

Data Type	MP default format	MP alternative format
32 bit word	BigEndian	LittleEndian
fixed precision float	IEEE	VAX
arb prec float	GnuMP	
arb prec integer	GnuMP	PARI, BigNum

Table 1: Currently supported MP encodings

While the "negotiation" scheme outlined above might seem complex at first sight, it is totally hidden from the user of the MP API. For each application, the list of the possible encoding formats together with their respective scores are determined at compile time of the MP library. The determination of the encoding takes place without the user realizing it when a MP link is opened.

4 MP high level data integrations – Prototypes

In MP, a datum is usually communicated as a node packet and composite data as an expression tree. While this approach works well for objects having an heterogeneous and tree-like syntactic structure, it is unnecessarily inefficient for objects which have a more homogenous format (like vectors, matrices, polynomials, etc.). For such objects, we would like to get away from an expression tree representation and support representations which more naturally (and efficiently) reflect the object's internal representation and which avoid the overhead incurred by having to communicate a node packet header for each datum. In addition to the obvious advantage that there would be less data to transmit and

to buffer inside an application's interface, parsing is simpler and more efficient, for now blocks of homogeneous data (without headers) can be moved between the interface and the application's internal data structure using efficient memory move operations.

However, communicating headerless data requires establishing mechanisms which ensure that the data can still be correctly parsed. Within the context of MP we considered the following approaches:

1. In addition to statically defining the semantics of an operator in a dictionary, we give the static definition of an operator's syntax. Figure 2 gives an example. It assumes that we explicitly defined in the `Matrix` dictionary that the `SpIntMat` operator is followed by 3*#Arg+2 headerless integer arguments, where the first two integers encode the dimensions and the remaining data are triples encoding the non-zero entries of the matrix.

2. A MP-defined "type description" (i.e., prototype) dictionary specifies constructs (i.e., operators, annotations, etc.) and rules used to dynamically describe the structure of homogenous data. MP and user-defined dictionaries are free to specify the "expected" syntax of operators identifying structured objects using these mechanisms. The corresponding prototypes (i.e., type descriptions), however, are always sent with the object at transmission time, so interfaces that know the dictionary may use very efficient routines to parse the objects, and interfaces that do not know the dictionary may still (at least) parse the object. It is in this sense, that we can consider prototyped objects as "self-describing".

Approach (1) has the advantage of being most efficient (the amount of type information sent is minimal), but has the disadvantage that the task of parsing requires *built-in* knowledge of the definitions in a referenced dictionary. Clearly, this approach is inappropriate for a general protocol. Consequently, we decided to pursue (2). As we will show, a careful design of the grammar for the self-describing encoding of headerless data results in mechanisms which are almost as compact as (1) and have the additional advantage that the communicated data can always be syntactically manipulated (e.g., stored, displayed, duplicated, etc.). In other words, the realization of approach (2) is our solution to the efficiency versus generality problem on this syntactic level.

4.1 The prototype annotation

The main construct to specify the type and structure of homogenous data is a `MP_Prototype` annotation. It may be attached to an operator and its value (which is again a MP Tree, often referred to as prototype tree) specifies the syntax, and, possibly some semantics, of *all* arguments of the operator. Subsequently, the entire collection of argument data items can be placed in a *data packet*. A receiver side would retrieve the prototype and use it to properly decode the (headerless) data.

For describing prototype trees, we more formally define the syntax of a prototype annotation by

<Prototype AP> ::= AP(MP_Prototype)<MP TypeSpec>

<MP TypeSpec> ::= <Basic TypeSpec> |
<Operator TypeSpec>

Figure 3: An array of 1000 IMP_Real32 numbers

Type	Dict	Value	#Arg:Annot	Remarks
COP	Proto	MP_Array	1000:1	1) CommonOp
AP	Proto	MP_Prototype		2) Proto Annot
CMBP	Proto	IMP_Real32		3) type spec
		1.0		4) *data packet*
		2.0		data only, no
		3.0		extra pkt
		...		info

where AP(x) is an Annotation Packet of type x,. Furthermore, we make a distinction between *prototype specification time*, the time at which we create (send) the prototype, and *data communication time*, the time at which we actually send the data described by the prototype.

4.2 Basic type specifications

Basic type specifications denote that the type of the data to be read is one of the MP atomic types or an extension supplied in a dictionary as a Basic TypeSpec. The atomic MP types are the non-operator types: those that cannot have arguments and can only appear as the leaves of a tree. A distinction is made between a datum sent with and without the packet header. The former are prefaced with `MP_` and the latter with `IMP_`. For example, `MP_Sint32` corresponds to a node packet containing a signed 32-bit integer, requiring 4 bytes for the packet header and 4 bytes for the data. Sending a complete node packet is useful if we need to attach *different* annotations to that node at different points in the data. `IMP_Sint32` indicates that the integer is to be sent without a node packet header, that is, as (part of) a data packet. It is not possible to attach annotations directly to data in a data packet.

The specification of MP basic types is done using CommonMetaBasicPackets (CMBPs) whose value specifies the *type* of some data that will appear later. The value of a Common Meta Basic Type is an integer encoding identifying a basic type. Section 4.3.3 discusses user-defined types defined in a dictionary.

The following simple example makes this more concrete and helps motivate the discussion. We encode an array of 1000 `IMP_Real32` numbers as shown in Figure 3. COP indicates a Common Operator Packet. Note that the prototype specifies that the elements of the array were transmitted *without* corresponding node packets (`IMP_Real32` is used instead of `MP_Real32`) on line 3, so only the actual data from the array is transmitted in the MP tree that follows the prototype.

Line 1 tells the receiver that what follows is an array of 1000 items. The prototype on lines 2 - 3 indicate that *each* element of the array is of the type `IMP_Real32`. Recall that "IMP" indicates an instance of the data type and not a complete node packet of that type. The data follows (line 4 on), but without node packets to individually specify the type of each element. For an array of 1,000 32-bit floats, the data requires 4,000 bytes and an overhead for the node packet headers of 4,000 bytes. Using the prototype to specify the element type of the array reduces the total size of the array's encoding from 8,008 bytes (4,008 bytes total overhead) to 4,016 bytes (16 bytes of overhead). An important point

to make about prototypes is that the size of the prototype is largely independent of the size of the data. In the example above, the overhead for the prototype stays at 16 bytes even if, for example, the size of the data doubles (4 to 8 bytes for an `IMP_Real64`) or the length of the array increases.

4.3 Operator type specifications

The most simple of prototypes consist of a single Basic TypeSpec, as in the example above. For more complex data structures such as matrices, arrays of structures, and recursive data structures, operators must appear within the prototype tree. These more complex objects can be constructed from four basic structuring operators: structures, unions, arrays, and pointers. User-defined structures are also supported. Syntactically, we define:

<Operator TypeSpec> ::=
  <General Operator TypeSpec>
  | <Meta Operator TypeSpec>

<General Operator TypeSpec> ::=
  MP_CommonOperatorPkt(MP_Struct)$_n$ <MP TypeSpec>$_n$
  | MP_CommonOperatorPkt(MP_Union)$_n$ <MP TypeSpec>$_n$
  | <OperatorPacket>(OP)$_n$ <MP TypeSpec>$_n$

<Meta Operator TypeSpec> ::=
  MP_CommonMetaOperatorPkt(MP_Array) <Prototype AP>
  | MP_CommonMetaOperatorPkt(MP_Pointer) <Pointer AP>

<Pointer AP> ::=
  <Prototype AP> | AP(MP_RecursiveStructAnnot)
  | AP(MP_RecursiveUnionAnnot)

<OperatorPacket> ::=
  MP_OperatorPkt | MP_CommonOperatorPkt

where OP is any operator. An index n to a packet specification indicates that the packet has n arguments within the prototype tree (all MP TypeSpecs).

4.3.1 General Operator TypeSpec

As indicated above, the arguments to a "general" operator appearing in a Prototype tree are all MP TypeSpecs. Two obvious cases where the need for this arises are in the specification of structures and unions.

Struct TypeSpec

<Struct TypeSpec> ::=
  MP_CommonOperatorPkt(MP_Struct)$_n$ <MP TypeSpec>$_n$

A collection of (possibly heterogeneous) objects to be treated as logically related is described using a `MP_Struct`. The number of arguments field specifies both the number of actual arguments the operator has in the Prototype tree *and* the number of data items to be read from the data packet that follows the Prototype tree. The arguments to the struct operator specify the types of the arguments individually using MP TypeSpecs. Figure 4 shows how a sparse matrix can be represented (see also Figures 1 and 2 for comparison). Note that the statically defined representation in Figure 1 required 48 bytes and the prototyped version requires only 76 bytes, but the size of the prototype remains constant.

Figure 4: A sparse matrix

Type	Dict	Value	#Arg:Annot	Remarks
COP	Matrix	SparseMat	1:3	
AP	Proto	MP_Prototype		
COP	Proto	MP_Struct	3:0	
CMP	Proto	IMP_Uint32		rowspec
CMP	Proto	IMP_Uint32		columnspec
CMP	Proto	IMP_Sint32		entry
AP	Matrix	SpMatRows		
MP_Uint32		100		#rows
AP	Matrix	SpMatCols		
MP_Uint32		100		#cols
		1		
		1		
		1		[1,1]=1
		2		
		1		
		1		[2,1]=1
		2		
		4		
		1		[2,4]=1

Union TypeSpec

<Union TypeSpec> ::=
  MP_CommonOperatorPkt(MP_Union)$_n$ <MP TypeSpec>$_n$

`MP_Union` is a union of n prototypes (MP Typespecs). The index n to a packet specification indicates that this packet has n arguments, each of which is a prototype (MP TypeSpec) and the indices 1 to n are used for union discrimination. The index 1 refers to the first prototype, index 2 to the second prototype, and so on. At communication time, when the sender is preparing to put a value described by a `MP_Union` operator, the sender must first put an `IMP_Uint32` in the range 1 to n in the data packet. This discriminator is followed by the data. In parsing the prototype, the receiver must remain aware of `MP_Union` operators and store the argument prototypes in the proper order. In reading the incoming data for a node described by a `MP_Union`, the parser must first read in an `IMP_Uint32` identifying the particular prototype to use, then read in the data according to that prototype. Obviously, if the receiver reads an index greater than n, an error must be returned. An example is given at the end of the section.

4.3.2 Meta Operator TypeSpecs

For objects such as structures and unions, the sender knows at prototype specification time how many arguments the structuring operator has. For these objects, the type of each field is given quite easily within the prototype by specifying them through *the operator's arguments*. However, this approach does not work for objects which are a repetition of a single type specification (an array, for example), or for which the actual number of arguments is not known at prototype specification time or for which the number of arguments may be different for individual instances of the object within the tree (a pointer or ragged array, for example).

These cases are handled using a variant of the Common Basic Meta type: `MP_CommonMetaOperator`. This Meta operator may only appear in a Prototype tree. They have two defining characteristics which distinguish them from general operators. First, the type of each element in the object is

defined through a nested prototype annotation. Second, the Meta operators have *no* actual arguments within the Prototype tree. Instead, the number of arguments field gives information about the number of arguments to expect in the data packet that follows the prototype. There are two cases to consider:

1. A meta operator packet with a non-zero number of arguments field. This is the number of elements to read from the data packet.

2. A meta operator packet with 0 in the number of arguments field. The number of arguments to the operator could not be given at prototype specification time and the actual number must be read from the data packet.

First we consider arrays, then pointers and recursion.

Array TypeSpec

<Array TypeSpec> ::=
CommonMetaOperatorPkt(MP_Array) <Prototype AP>

In the first case, the prototype consists of a common meta operator whose value is `MP_Array` and whose number of arguments field is *non-zero*. The common meta operator has a prototype specifying the type of each argument to the `MP_Array` operator. Here the number of arguments field specifies the number of data items that will *appear later in the data packet* and *not* the number of arguments that immediately follow the operator within the prototype. Consider, for example, a matrix of reals given as an array of arrays.

In the second case, a `MP_Array` operator with 0 in the number of arguments field appears in a prototype. As in the first case, the nested prototype gives the type of each of the arguments of the top-level operator. The 0 in the number of arguments field of the meta operator indicates that the actual number of arguments the operator has was not known at prototype specification time and will be given as an `IMP_Uint32` in the data packet *in the place where this operator would have appeared* (this value could be 0). Consider an array of arrays of different lengths. For example, $((-1,-2,-3),(-4,-5))$ in Figure 5. The prototype operator on line 1 is an array with no arguments, indicating that at this node's position in the linearized tree (data packet) the receiver should look for an `IMP_Uint32` giving the number of arguments for the subarray, as seen on lines 4 and 5. The receiver knows from the prototype which operator should appear here (`MP_Array`). What the sender cannot say at prototype specification time is how many arguments it actually has. The head of the prototype, line 1, is an `MP_Array` operator with 0 in the #args field, so the first thing the receiver would look for is an `IMP_Uint32` giving the length of the first subarray. The receiver would read in the "3" on line 4 and then read in three `IMP_Sint32`s, as specified by the nested prototype on line 3. After having read in the first subarray, the receiver would know to look for the next `IMP_Uint32` giving the length of the next subarray, and so on.

Pointer TypeSpec and recursive data structures

It is important to be able to handle the idea of a "pointer", especially since this is how recursive data structures are done. This is accomplished with the `MP_Pointer` common operator appearing in a common meta operator packet.

Figure 5: An uneven array of arrays

Type	Dict	Value	#Arg:Annot	Remarks
COP	Proto	MP_Array	2:1	2 elems in the array
AP	Proto	MP_Prototype		
CMOP	Proto	MP_Array	0:1	1) diff lengths
AP	Proto	MP_Prototype		2) subarray elem type
CMBP	Proto	IMP_Sint32	0:0	3) are ints
		3		4) 3 elem array
		-1		1st subarray
		-2		
		-3		
		2		5) 2 elem array
		-4		2nd subarray
		-5		

<Pointer TypeSpec> ::=
CommonMetaOperatorPkt(MP_Pointer) <Pointer AP>

<Pointer AP> ::=
AP(MP_PrototypeAnnot) | AP(MP_RecursiveStructAnnot)
| AP(MP_RecursiveUnionAnnot)

The number of arguments to the operator within the Prototype tree *must* be 0. Consequently, the number of arguments to the pointer type must be given explicitly in the data packet as an `IMP_Uint32` whose value is either 0, indicating a NULL pointer, or 1, indicating that the pointer "points to" a block of data. This data immediately follows the "1" in the data packet. For non-recursive objects, the structure of this data is given by a prototype to the `MP_Pointer` operator.

Recursive data objects are just a special case of structures or unions. The annotations `MP_RecursiveStructAnnot` (`MP_RecursiveUnionAnnot`) are attached to the struct (union) operator packet and to the CMOP(MP_Pointer) packet to indicate their recursive nature. A recursive pointer always points back to the most closely nested recursive structure or union within the prototype tree. This approach admittedly provides a limited "namespace", but it is our opinion that it will support the majority of needs, and, in keeping with the KISS principle[1], we chose not to go immediately to a true naming mechanism. However, using an annotation to indicate recursion is a *general* approach that can easily be extended simply by adding a naming annotation.

As before, `MP_Pointer` always has 0 in its number of arguments field. Consequently, the actual number of arguments must be read from the data packet which follows the prototype tree. This value must be 0 (NULL pointer) or 1 (non-NULL pointer). An accurate mental model of such a structure is a linked list.

A last example uses `MP_Union`, `MP_Struct`, `MP_Pointer`, in `MP_RecUnion` to represent a sparse recursive polynomial (see Figure 6). Notice how naturally the definition below is described in the prototype between lines 1 and 2.

```
union SparseRecPoly  {
  Sint32;                          // coefficient
  struct exponent {
   String;                         // varname
   Uint32;                         // exponent
   Pointer struct SparseRecPoly; // multiplic subpoly
   Pointer struct SparseRecPoly; // additive subpoly
```

[1] Keep It Simple Stupid.

Figure 6: A sparse recursive polynomial

Type	Dict	Value	#Arg:Annot	Remarks
COP		SparseRecPoly	1:1	
AP	Proto	MP_Prototype		
COP	Proto	MP_Union	2:1	1) recursive
AP	Proto	MP_RecUnion		
CMP	Proto	IMP_Sint32		
COP	Proto	MP_Struct	4:0	
CMP	Proto	IMP_String		var name
CMP	Proto	IMP_Uint32		exponent
CMOP	Proto	MP_Pointer	0:1	mult. subpoly
AP	Proto	MP_RecUnion		
CMOP	Proto	MP_Pointer	0:1	add. subpoly
AP	Proto	MP_RecUnion		2)
		2		use proto 2
		x		the outermost var
		4		its exponent
		2		use proto 2
		y		the current var
		1		its exponent
		1		use proto1
		3		coef
		1		use proto 1
		2		coef
		2		use proto 2
		x		var x
		2		and the value 2
		1		use proto 1
		1		value 1
		0		ptr: NULL

```
  };
};
```

The sample data is:

```
3*x^4(y+2) + x^2  =  x^4     +     x^2
                     *
                     (y      +     2)
                     *
                     3
```

4.3.3 user-defined types

The previous sections described operators defined in a MP dictionary. Users can define their own structured types in non-MP dictionaries and use them as the value of a Meta Operator in a Prototype tree following the fashion outlined in the previous sections (see [6] for further details). This will be convenient in a number of circumstances and provides the protocol with extensibility.

5 The MPT library

MPT (Multi Protocol Tree) is a library for parsing and manipulating MP trees. Its purpose is to support the implementation of general and efficient MP interfaces for mathematical software systems.

Development of MPT was mainly motivated by the following problem: While the flexibility to represent an object in a variety of formats eases the implementation of an application's "Put" interface (the sender can choose to "Put" the data in the format which is closest to its internal representation) and is one of the main sources for achieving high efficiency, it complicates the implementation of a receiving application's "Get" routines: Instead of having to "understand" just one, well-defined representation of an object, a receiver needs to be able to understand the same object in a variety of representations. For example, an integer matrix might be received as a prototyped sparse or dense matrix, or as an ordinary (unprototyped) sparse or dense matrix in an expression tree format. Realizing that it is a cumbersome requirement that every MP interface needs to understand every possible representation of every relevant object, the MPT library provides routines which allow an application to have to understand only one representation of an object. The current MPT library might be considered as a generalization of the MPP library described in [5]: the data representation used and the prototype mechanisms provided are more general (i.e., not polynomial specific), and a greater variety of tree manipulating routines are provided (see below).

The main data structure MPT operates on is `MPT_Tree` – a memory representation of a MP tree. A MPT tree closely reflects the structure of the MP encoding: It consists of a `MPT_Header` and a `MPT_Data`. A `MPT_Header` is a pointer to a structure used for storing properties of a MP node header (type, dictionary tag, number of arguments annotations, etc.) and a `MPT_Data` is a `(void *)` pointer which is used for storing the respective node data (e.g., value of a number, arguments of an operator, characters of a string, etc.). The explicit separation of the header from the actual data stems from the fact that MPT uses a principle similar to MP for handling prototyped data: the same header structure is used for specifying the type (and, possibly structure) of more than one data item (i.e., prototyped data is stored as compactly as it is communicated).

The main routines provided by MPT are `MPT_GetTree()`, which reads a MP tree from a MP link and transforms it into a `MPT_Tree`, and various primitive routines for manipulating `MPT_Tree`s (such as transforming all prototyped data into fully type-specified data, or initializing, copying, deleting, and comparing `MPT_Tree`'s). Additionally, collections of various subject-specific transformations of dictionary-defined objects are provided as add-on libraries (such as the MPP library, which provides transformations between the different polynomial representations [5], or the MPM library which provides matrix manipulations).

In its most straightforward form, MPT can be used as shown in the top row of arrows of Figure 7. Instead of reading in the data directly from a MP link into its internal representation, an application uses MPT to first read the data into a `MPT_Tree`, which is subsequently manipulated using the appropriate library or external manipulation routines (e.g., all polynomials are transformed into a sparse recursive representation), and finally transformed from a `MPT_Tree` into the application's internal representation. This is the easiest approach to implement a MP "Get" interface: Not only can one rely on the various manipulation routines for MPT trees to transform a `MPT_Tree` object into a unique representation, but also the transformation from a `MPT_Tree` into an application's internal representation is greatly facilitated by the ability to repeatedly walk through the tree in an arbitrary order (which is largely impossible, if we were to read the data directly from a MP link).

However, for objects which could be read directly from the link into an application's internal representation (i.e., objects whose format is nearly identical to the application's "native" format), the intermediately generated `MPT_Tree` representation implies an unnecessary loss of efficiency. To

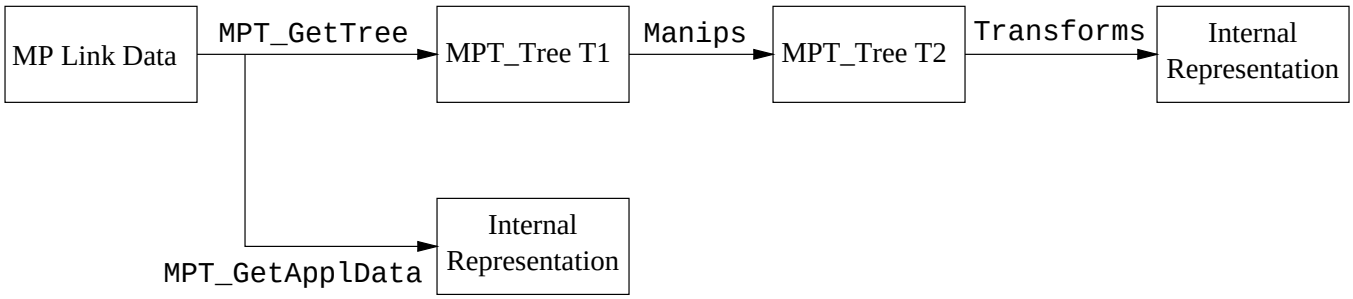

Figure 7: Getting data with the MPT library

avoid this inefficiency without losing the convenience of the straightforward approach, we use the following solution: *After* `MPT_GetTree()` has read the header of a node (including reading all attached annotations), but *before* reading the actual data packet, the routine `MPT_GetApplData()` is called with the node header as an argument. `MPT_GetApplData()` is a pointer to a function which is to be redefined by each application. The intended functionality is as follows: An application checks the type and/or structure of the object as specified by the node header. If the object is in a format which is close to the application's "native" format, then the data is read in directly, bypassing `MPT_GetTree`'s data read routines and stored as encapsulated opaque data within a `MPT_Tree`. Otherwise, the data is read in by the "normal" `MPT_GetTree` routines and subsequently manipulated and transformed as outlined above.

Unfortunately, this optimization approach introduces a duplication of development effort since we need to have two nearly identical set of routines which transform "native" objects into the application's internal representation: One set which accomplishes that from MP link data and another set which accomplishes the same from `MPT_Tree` data. To overcome this duplication, we take advantage of MP's abstract device interface (ADI) [16], and define a new MP *transport device* – a MP internal memory device – and can consequently bind this internal memory device to a new type of MP link: an internal memory link. The ADI operations are implemented based on the `MPT_Tree` data structures and on the routines of the MPT library such that writing/reading a `MPT_Tree` to/from a internal memory MP link merely amounts to pointer assignments. Consequently, instead of having to implement a second set of routines which transform native objects from a `MPT_Tree` representation, we "Put" ("associate" might be a better verb here) the `MPT_Tree` to an internal MP memory link, and then can use the *same* set of "Get" routines which read MP link data directly into an applications internal representation (see also the "back loop" of Figure 7).

We wish to point out that, based on the MPT library, one is able to implement an MP "Get" interface which is nearly optimal w.r.t. efficiency (it can basically be as efficient as specialized "canned" "Get" routines) and simplicity/convenience (as an API programmer, one basically needs to implement one set of Get routines and can let the MPT library routines do the rest). In other words, MPT is our answer to the generality versus efficiency problem on the programming level.

6 Conclusion

Achieving connectivity between mathematical software packages presents many challenges. A protocol which expects to be useful to a broad range of applications that may be connected in a variety of different computing paradigms *must* be general as well as efficient. Generality allows applications to easily communicate data with a shared understanding of its syntax and semantics. Different applications have different needs, but efficiency is, of course, always welcome, and in some situations (for example for parallel computations) it is critical.

Our approach has been to attack the syntactic side of this problem: we discussed it on several levels and proposed solutions which evolved from our experiences with MP.

At the lowest level, that of the protocol's primitive data objects (like numbers), a protocol should support encodings which are widely used and efficient to communicate. Further, the protocol should not fix a standard encoding for numeric data, but provide means to dynamically determine the most appropriate data encoding. In this way, costly conversions which would otherwise occur can be minimized.

At a higher level, dynamic type specifications, or prototypes, should be provided as a powerful mechanism for expressing structure and type information for commonly used mathematical objects. While our proposed prototypes are relatively simple, they provide great flexibility to model the structure of a collection of homogeneous data items. This approach provides complete syntactic and type information without noticeable loss of efficiency.

Finally, tools should be provided which simplify the interface programmer's task, both conceptually and in terms of implementation. (A protocol that is powerful, but difficult to grasp and/or to implement will have a short lifespan.) We address this problem by providing a flexible and efficient MP parser library which, in combination with the link and abstract device concepts, eases the implementation of MP interfaces. Although these features have not received lavished attention here, their importance should not be underestimated.

We do not expect that all the details of our solutions will stand the test of time. However, we are convinced that their principles lead us in the right directions and, together with enough stamina and cooperative weather, will enable us to reach our long-sought destination.

References

[1] Abbott, J., and Traverso, C. Specification of the POSSO External Data Representation. Technical report, Sept. 1995.

[2] Abbott, J., van Leeuwen, A., and Strotmann, A. Objectives of OpenMath. Technical Report 12, RIACA, Amsterdam, June 1996.

[3] Ajwa, I., and Wang, P. Using PVM to Speedup Gröbner Bases Computations. In *Proceedings of the Eighth IASTED International Conference on Parallel and Distributed Computing Systems (PDCS '96)* (1996), K. Li, T. S. Abdelrahman, and L. E., Eds., pp. 457–461.

[4] Amrhein, B., Gloor, O., and Küchlin, W. A Case Study of Multi-Threaded Gröbner Basis Completion. In *Proc. of the International Symposium on Symbolic and Algebraic Computation (ISSAC'96), Zurich, Switzerland* (July 1996), ACM Press.

[5] Bachmann, O., Schönemann, H., and Gray, S. A Framework for Distributed Polynomial Systems Based on MP. In *Proc. of the International Symposium on Symbolic and Algebraic Computation (ISSAC'96), Zurich, Switzerland* (July 1996), ACM Press.

[6] Bachmann, O. and Gray, S. and Schönemann, H. MP Prototype Specification. In *Reports On Computer Algebra*, no. 10. Centre for Computer Algebra, University of Kaiserslautern, January 1997.

[7] Bronstein, M. σ^{IT} – A Strongly-Typed Embeddable Computer Algebra Library. In *Proc. of the Intl. Symp. on Design and Implementation of Symbolic Computation Systems (DISCO'96)* (Karlsruhe, Germany), J. Calmet and C. Limongelli, Eds., vol. 1128 of *Lecture Notes in Computer Science*, Springer Verlag.

[8] Chen, M., Cowie, J., Fox, G., Furmanski, W., and Rebbi, C. WebWork: Integrated Programming Environment Tools For National and Grand Challenges. Technical Report CRPC-TR95614, Center for Research on Parallel Computation (CRPC), June 1994.

[9] Cooperman, G. STAR/MPI: Binding a Parallel Library to Interactive Symbolic Algebra Systems. In *Proc. of the International Symposium on Symbolic and Algebraic Computation (ISSAC'95), Montreal, Canada* (July 1995), A. H. M. Levelt, Ed., ACM Press, pp. 126 – 132.

[10] Dalmas, S., Gaëtano, M., and Sausse, A. ASAP: a protocol for symbolic computation systems. INRIA Technical Report 162, Mar. 1994.

[11] Dewar, M. C. Manipulating Fortran Code in AXIOM and the AXIOM-NAG Link. In *Proceedings of the Workshop on Symbolic and Numeric Computing* (1994), H. Apiola, M. Laine, and E. Valkeila, Eds., University of Helsinki, Finland, pp. 1–12. Available as Technical Report B10, Rolf Nevanlinna Institute.

[12] Gallopoulos, E., Houstis, E., and Rice, J. Computer as Thinker/Doer: Problem-Solving Environments for Computational Science. *IEEE Computational Science and Engineering* (1994), 11–23.

[13] Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R., and Sunderam, V. PVM3 User's Guide and Reference Manual. Technical Report ORNL/TM-12187, Oak Ridge National Laboratory, September 1994.

[14] Gray, S., Kajler, N., and Wang, P. S. MP: A Protocol for Efficient Exchange of Mathematical Expressions. In *Proc. of the International Symposium on Symbolic and Algebraic Computation (ISSAC'94), Oxford, GB* (July 1994), M. Giesbrecht, Ed., ACM Press, pp. 330–335.

[15] Gray, S., Kajler, N., and Wang, P. S. Design and Implementation of MP, a Protocol for Efficient Exchange of Mathematical Expressions. Forthcoming in *Journal of Symbolic Computing*.

[16] Gray, S., Kajler, N., and Wang, P. S. Pluggability Issues in the Multi Protocol. In *Proc. of the Intl. Symp. on Design and Implementation of Symbolic Computation Systems (DISCO'96)* (Karlsruhe, Germany, Sept. 1996), J. Calmet and C. Limongelli, Eds., vol. 1128 of *Lecture Notes in Computer Science*, Springer Verlag.

[17] Grayson, D., and Stillman, M. *Macaulay 2*, 1996. For further information contact the author s at `dan@math.uiuc.edu` or `mike@math.cornell.edu`.

[18] Greuel, G.-M., Pfister, G., and Schönemann, H. *Singular*: A system for computation in algebraic geometry and singularity theory. University of Kaiserslautern, Dept. of Mathematics, 1995. Available via anonymous ftp from `helios.mathematik.uni-kl.de`.

[19] Gropp, W., Lusk, R., and Skjellum, A. *Using MPI.* MIT Press, 1994.

[20] Hong, H., Ed. *Proc. of the 1st Intl. Symp. on Parallel Symbolic Computation (PASCO'94)* (Sept. 1994), vol. 5, World Scientific.

[21] PSEware Group. The PSEware Project: A Toolkit for Building Problem-Solving Environments. http://www.extreme.indiana.edu/pseware, 1996.

[22] Rice, J. Scalable Scientific Software Libraries and Problem Solving Environments. Technical Report CSD TR-96-001, Department of Computer Science, Purdue University, Jan. 1996.

[23] Wolfram Research, Inc. MathLink Reference Guide (version 2.2). Mathematica Technical Report, 1993.

[24] Yau, H., Leung, D., Furmanski, W., and G., F. Exploration of Emerging HPCN Technologies for Web-Based Distributed Computing. Technical Report CCS-755, Northeast Parallel Architectures Center (NPAC).

[25] Zippel, R. MathBus. Available from `http://dave.cs.cornell.edu:8002/Simlab/papers/mathbus/mathTerm.htm`, 1996.

List of papers published in the Reports on Computer Algebra series

[RCA:29] Greuel, G.-M. Computer Algebra and Algebraic Geometry – Achievements and Perspectives. Feb 2000.

[RCA:28] Siebert, T. Recursive Computation of Free Resolutions and a Generalized Koszul Complex. Jan 2000.

[RCA:27] Bachmann, O. and Gräbe, H.-G. The SymbolicData Project: Towards an Electronic Repository of Tools and Data for Benchmarks of Computer Algebra Software. Jan 2000.

[RCA:26] B. Reinert. Solving One-Sided Equations in Integer Monoid Rings. Jan 2000.

[RCA:25] B. Reinert and D. Zeckzer. Coset Enumeration using Prefix Gröbner Bases in Mrc – An Experimental Approach. Nov 1999.

[RCA:23] Birgit Reinert. Observations on coset enumeration. Nov 1998.

[RCA:22] K. Madlener and F. Otto. Some Applications Of Prefix-Rewriting In Monoids, Groups, And Rings. November 1998.

[RCA:21] G.-M. Greuel, G. Pfister, and H. Schönemann. Singular version 1.2 user manual. June 1998.

[RCA:20] B. Reinert and D. Zeckzer. MRC – A System for Computing Gröbner Bases in Monoid and Group Rings. July 1998.

[RCA:19] B. Reinert, K. Madlener, and T. Mora. A note on nielsen reduction and coset enumeration. February 1998.

[RCA:18] O. Bachmann and H. Schönemann. Monomial Representations for Gröbner Bases Computations. January 1998.

[RCA:17] Thomas Siebert. An algorithm for constructing isomorphisms of modules. January 1998.

[RCA:16] K. Madlener and B. Reinert. String Rewriting and Gröbner Bases – A General Approach to Monoid and Group Rings. October 1997.

[RCA:15] B. Martin and T. Siebert. Splitting Algorithm for vector bundles. September 1997.

[RCA:14] K. Madlener and B. Reinert. Relating rewriting techniques on monoids and rings: Congruences on monoids and ideals in monoid rings. September 1997.

[RCA:12] O. Bachmann, S. Gray, and H. Schönemann. MP Prototype Specification. Dec 1997.

[RCA:11] O. Bachmann. Effective simplification of cr expressions. January 1997.

[RCA:10] O. Bachmann, S. Gray, and H. Schönemann. A proposal for syntactic data integration for math protocols. January 1997.

[RCA:09] B. Reinert. Introducing reduction to polycyclic group rings – a comparison of methods. October 1996.

[RCA:08] T. Siebert. On strategies and implementations for computations of free resolutions. September 1996.

[RCA:02] H. Schönemann. Algorithms in singular. June 1996.

[RCA:01] H. Grassmann, G.-M. Greuel, B. Martin, W. Neumann, G. Pfister, W. Pohl, H. Schönemann, and T. Siebert. Standard bases, syzygies and their implementation in singular. July 1996.