UNIVERSITÄT KAISERSLAUTERN
Zentrum für Computeralgebra

REPORTS ON COMPUTER ALGEBRA
NO. 12

MP Prototype Specification

by

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

Dec 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/`

MP Prototype Specification

Olaf Bachmann Simon Gray Hans Schönemann

December 1997

1 Introduction

Briefly, a prototype is a syntax tree providing type and structuring information about a composite data object. This report expands and explains in detail the definition and use of prototypes within the Multi Protocol (MP). Readers are referred to [1, 3] for a general description of MP and to [2] for a more general overview of the ideas and philosophy behind MP prototypes.

An important requirement that drives many of the design decisions discussed here is that an MP Tree should *always* be parsable at the syntactic level. This means that a receiver may have no understanding of the dictionaries (i.e., semantics) involved, but should still be able to parse, manipulate, and echo the MP Tree (that is, process the tree at the syntactic level).

An implication of this decision is that MP does not support purely static definitions of composite data objects in a dictionary. By contrast, this is exactly the approach taken by XDR [4] in which an object is statically defined in a file and an object-specific filter routine is responsible for encoding it on the sending side. In this scenario, only data is transmitted and no type or structuring information is supplied; rather, this information is hardcoded into the filter. The sender assumes the receiver knows the type of data arriving and has a corresponding object-specific filter which can decode the incoming data.

In MP, on the other hand, a composite object must be transmitted either as an MP Tree using only node packets or using a prototype to describe the structure and content of the object, with the (usually headerless) ”pure” data from the object following the prototype. The use of prototypes does not preclude the possibility of writing efficient routines for sending and receiving mathematical objects. Instead of decoding each datum by interpreting the prototype, a receiver can simply check the entire prototype to ensure that the incoming data meets the expectations of a compiled read routine. The sending side is simpler: An object-specific routine may efficiently marshall the user's data structure, *including* sending the prototype describing the object.

There are several competing goals in specifying the syntax and semantics of prototypes:

1. Efficiency - This is the motivation for including protoypes within MP; to overcome the cost of communicating full node packets when the data has a homogeneous format. In addition to reducing the amount of data transmitted, this approach also allows the use of efficient compiled routines for encoding and decoding the data.

2. Flexibility - A prototype must be able to represent a wide variety of data structures, including arrays, lists, structures (records), pointers (and recursive structures using pointers), and should do so in a way that naturally reflects the organization of the data structure.

3. Simplicity - The concept of prototypes must be simple, as must the process of creating prototypes. The interpretation of a prototype must be unambiguous. Relatedly, the design of prototypes must not greatly complicate the implemenation of the protocol-application interface.

Section 2 briefly reviews the general syntax rules governing data communicated by MP and motivates the need for prototypes. The formal definition of prototypes is given in section 3. A number of examples are also given to make the definitions more concrete and to further illustrate the advantages of this approach. A short discussion of the use of tree specifications with respect to prototypes is given in section 3.3.

Notice that we do *not* define in this report how particular mathematical objects such as polynomials or matrices should be represented. This is done in the dictionaries covering the mathematical areas where these objects occur. We furthermore do not define in this report formal mechanisms for mapping syntactic data structures to MP encodings. This is done in the report describing the MP Data Definition Language (DDL) [**?**]. Instead, what we describe here are the *means* which MP provides to develop efficient and compact encodings of these objects.

2 Syntax of MP and Prototype Annotations

The syntax of data transmitted by MP may be described by five levels:

1. Messages
2. Trees
3. Node Packets
4. Data limbs[1]
5. Annotation Packets

On its highest level, an MP message is the unit of communication between applications, bracketed by BeginMsg and EndMsg markers. On the MP API level, messages are separated by a concluding `MP_EndMsgReset()` call on the sending side, and by a `MP_SkipMsg()` or `MP_InitMsg()` call on the receiving side.

A message consists of zero or more MP Trees, although usually one tree per message is communicated. An MP Tree is built from node packets, annotations packets, and data limbs and might be thought of as the MP equivalent of an expression used by most CASs.

A data limb consists of "pure" data; no type information is directly attached to data limbs. Instead, their type is implictly encoded by the *place* at which a data limb appears in an MP Tree. Using the MP API, data limbs are usually sent using `IMP_Put<type>()` routines (e.g., `IMP_PutSint32()`) and received using `IMP_Get<type>()` routines (e.g., `IMP_GetSint32()`). In the style of the MP API, we will use the notation IMP_<type> in this report to refer to a data limb (e.g., IMP_Sint32 refers to a data limb that is four bytes long and encodes an Sint32 value).

An annotation packet consists of a header which, for valuated annotations, is immediately followed by an MP Tree. Whether or not an annotation is valuated is encoded in the annotation header. A node contains exactly as many annotation packets as is specified in the node packet header. Annotation packets are usually sent using `MP_PutAnnotationPacket()` and received using `MP_GetAnnotationPacket()`.

A node packet consists of a header, and, possibly, a data limb. The header encodes the type of the node (indicating the type of the data carried in its data field), a dictionary tag field identifying the dictionary in which the node's value is defined (for relevant types), the number of annotation packets attached to the node, and, for operator node packets (see below), the number of arguments of the node. On the MP API level, node packets are usually sent using `MP_Put<type>Packet()` routines (e.g., `MP_PutSint32Packet()`) and received using `MP_Get<type>Packet()` routines (e.g., `MP_PutGetPacket()`). Following the style of the MP API, we will use the notation MP_<type> in this report to refer to a node packet (e.g., MP_Sint32 refers to an Sint32 node packet). From a syntactic point of view, node packets are furthermore categorized w.r.t. two criteria:

1. basic versus operator node packets
   Basic node packets appear at the leaves of an MP Tree and may not be followed by argument packets. In contrast, operator packets have arguments and are therefore interior nodes of an MP Tree.

2. common versus ordinary node packets
   Unlike an "ordinary" node packet, a common node packet does not have a separate data limb containing the node's value, but instead has the node's value carried in the node packet header. In contrast, the header of an ordinary node packet is immediately followed by a data limb which encode's the node's value.

[1] What we call here a "data limb" used to be called a "data packet" in previous descriptions of MP. However, to make our description clearer, we will generally use the term "packet" to refer to a datum that has its type information directly attached (node packets) and use the term "limb" to refer to a datum that has no type information attached.

By default, the arguments of an operator node packet are again MP Trees. However, when all arguments of an operator have a homogeneous syntactic format, it is possible to minimize the overhead incurred by the node packet headers of the arguments. An array of floating point numbers is a good example, but the data could be more structured - an array of complex or rational numbers, or a polynomial or generators of an ideal.

To enable such minimizations, we extend the valid syntax of MP Trees by providing special annotations and operators in the prototype dictionary (`MP_ProtoDict`) which define mechanisms for communicating the arguments of an operator node packet as data limbs.

To specify that the arguments of an operator node packet are data limbs (i.e., are not normal MP Trees), a prototype annotation must be used:

- The prototype annotation is defined in the prototype dictionary (as `MP_AnnotProtoPrototype`). The annotation must have the valuated and required flags set. For clarity and to distinguish the MP Tree for an expression from the MP Tree following a prototype annotation, we will refer to the latter as a *prototype tree*. Furthermore, the block of data described by a prototype tree is referred to as a *prototyped data tree*.

  More formally, the syntax of a prototype annotation packet is:

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

  where AP(x) is an Annotation Packet of type x.

- A prototype annotation may only be attached to an operator node packet.

- If a <Prototype AP> appears as an annotation to an operator node packet and the <MP TypeSpec> that follows denotes a valid prototype tree, then all of the operator's arguments are of the type specified in <MP TypeSpec>.

- A valid prototype tree may only consist of meta types, meta operators, and four designated operators defined in the prototype dictionary (see the next section for details), and otherwise satisfies the syntactic rules of an MP Tree.

- If an annotation is attached to a node in the prototype tree, it applies to every instance of the data in the prototyped data tree described by that node. This is a very cost effective mechanism for providing supplemental information about a potentially large block of data.

We make a distinction between *prototype specification time*, the time at which the prototype is transmitted, and *data communication time*, the time at which the data described by the prototype is transmitted.

3 MP Type Specifications

An MP type specification is an MP Tree giving the structure and content type of a block of data. The prototype tree may specify that the data block is of a single basic type or that it consists of a tree built from a combination of operators and basic types.

Table 1 gives a summary of the rules governing the relationship between a prototype tree and the data transmitted at data communication time. The remainder of this report expands on this table. The TypeSpec column of the table uses the following notation: <type>:<dictionary>::<value>:<nargs>, where <type> specifies an MP node packet type ("Cmt" for CommonMetaType, "Cop" for CommonOperator, "Cmop" for CommonMeta-Operator, etc.), <dictionary> is the name of the dictionary where <value> is defined, and <nargs> gives the number of arguments for operators. For example, "Cmt:Proto::IMP_Sint32" specifies the Common Meta Type `MP_CmtProtoIMP_Sint32` defined in the prototype dictionary.

More formally, we define an MP type specification by:

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

TypeSpec	Corresponding data	Section
	Basic TypeSpec	
Cmt:Proto::IMP_*	1 IMP_* data limb	§ 3.1
Cmt:Proto::RecStruct	as specified by the preceeding Cop:Proto:RecStruct TypeSpec	§ 3.2.3
Cmt:Proto::RecUnion	as specified by the preceeding Cop:Proto:RecUnion TypeSpec	§ 3.2.3
	Tree Typespec	
Any other (Common) MetaType	1 MP Tree	§ 3.1, § 3.3
	Prototype Operator TypeSpec	
Cop:Proto::Struct:n <exactly n TypeSpecs follow>	n data limbs follow i^{th} limb is as specified by i^{th} TypeSpec	§ 3.2.1
Cop:Proto::Union:n <exactly n TypeSpecs follow>	1 IMP_Uint32(m) and exactly 1 data limb follows which is as specified by the $(m-1)^{th}$ TypeSpec	§ 3.2.1
Cop:Proto::RecStruct:n <exactly n TypeSpecs follow>	n data limbs follow i^{th} limb is as specified by i^{th} TypeSpec	§ 3.2.2
Cop:Proto::RecUnion:n <exactly n TypeSpecs follow>	1 IMP_Uint32(m) and exactly 1 data limb follows which is as specified by the $(m-1)^{th}$ TypeSpec	§ 3.2.1
	Meta Operator TypeSpec	
Mop:Any::Any:n or Cmop:Any::Any:n	if (n == 0) then IMP_Uint32(m) else set m = n m data limbs which are as specified by the prototype annot attached to the meta operator	§ 3.2.2
Any other Op or Cop	syntax error	§ 3.2.1

Table 1: MP TypeSpec summary

3.1 Basic type specifications

Basic type specifications specify that the type of a transmitted data limb is one of the MP basic types.

The specification of basic types is done using CommonMetaType (Cmt) node packets whose value

- is defined in the prototype dicitionary (`MP_ProtoDict`)
- specifies the *type* of some data limb that will appear in the prototyped data tree that follows.

More precisely, a basic type specification is defined by:[2]

<**Basic TypeSpec**> ::=
<IMP Integer Cmt> |
<IMP Real Cmt> |
<IMP StringBased Cmt> |
<Recursion Cmt> | Cmt:Proto::IMP_Raw

<**IMP Integer Cmt**> ::= Cmt:Proto::IMP_Sint32 | Cmt:Proto::IMP_Uint32 | Cmt:Proto::IMP_ApInt

<**IMP Real Cmt**> ::= Cmt:Proto::IMP_Real32 | Cmt:Proto::IMP_Real64 | Cmt:Proto::IMP_ApReal

<**IMP StringBased Cmt**> ::= Cmt:Proto::IMP_String | Cmt:Proto::IMP_Identifier | Cmt:Proto::IMP_Constant

[2]Notice, that we do not currently support the type specification of data limbs for the common basic types (i.e., there are no <IMP Common Cmt> definitions). Due to internal implementation constraints, the length of a data limb needs to be a multiple of four bytes, and common basic data types have a length of only one byte.

<**Recursion Cmt**> ::= Cmt:Proto::RecUnion | Cmt:Proto::RecStruct

For a first simple example we encode an array of 1000 `MP_Real32_t` numbers. The resulting MP Tree is shown in a stylized fashion in figure 1. The actual encoding is binary and not easily human readable. A fragment of the code to produce the tree is shown in figure 2.

Figure 1: An array of 1000 IMP_Real32 numbers

Type	Dict	Value	#Annot:Arg	Remarks
Cop	Proto	Array	1:1000	(1) CommonOperatorPacket
AP	Proto	Prototype		(2) Prototype AnnotationPacket
Cmt	Proto	IMP_Real32	0:	(3) MP type specification
IMP_Real32		1.0		(4) beginning of prototyped data tree:
IMP_Real32		2.0		data only, no extra
IMP_Real32		3.0		packet information
		...		

Line (1) contains the common operator `Array` defined in the Proto dictionary. It has a single annotation and 1000 arguments. The annotation is a prototype annotation and is given on lines (2 - 3). It specifies that *each* element of the array is of the type IMP_Real32. Recall that the common meta type packet (Cmt) specifies the type of data that will appear in this position of the MP Tree that follows. The prototyped data tree follows, beginning on line (4). Note that there are no node packet headers to *individually* specify the type of each element of the prototyped data tree (this is the job of the prototype). Recall that the IMP_<type>s represent data only - there is no attached type information. An unprototyped version of figure 1 requires 8,008 bytes: 8 bytes for the Array node packet and 8,000 bytes for the array elements, 4,000 bytes of which is the actual data (1,000 32-bit floats) and the remaining 4,000 bytes are the overhead for the node packet headers. Using the prototype mechanism 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 doubles.

3.2 Operator type specifications

The most simple of prototypes consist of a single Basic TypeSpec, as in figure 1. To specify the type of composite data such as structures, union, or arrays of structs or unions, (meta) operators must appear within the prototype tree. Syntactically, we define:

<**Operator TypeSpec**> ::= <Prototype Operator TypeSpec>
| <Meta Operator TypeSpec>
<**Prototype Operator TypeSpec**> ::= Cop:Proto::Struct:n <MP TypeSpec>$_n$
| Cop:Proto::RecStruct:n <MP TypeSpec>$_n$
| Cop:Proto::Union:n <MP TypeSpec>$_n$
| Cop:Proto::RecUnion:n <MP TypeSpec>$_n$
<**Meta Operator TypeSpec**> ::= Mop:Any::Any<Prototype AP> | Cmop:Any::Any<Prototype AP>

where the index n to <MP TypeSpec> specifies n arguments, each a prototype tree. The structuring operators `MP_CopProtoStruct`, `MP_CopProtoRecStruct`, `MP_CopProtoUnion`, and `MP_CopProtoRecUnion` used in Prototype Operator TypeSpecs are defined in the prototype dictionary (`MP_ProtoDict`). These are the *only* operators which may appear in a syntactically valid prototype tree. That is, a valid prototype may only consist of (common) meta operators, (common) meta types, and these four common operators (in addition to annotation packets attached to these nodes). In particular, other operators defined in the prototype dictionary, such as Cop:Proto::Array (`MP_CopProtoArray`) or Cop:Proto::Pointer (`MP_CopProtoPointer`), are only provided as convenient means for mapping data structures to prototyped data trees and play *no* special role within prototype trees.

```
MP_Real32_t r32_array[1000];

/* ... Fill the array ... */

/* (1) Put the operator node packet */
MP_PutCommonOperatorPacket(link,              /* MP_Link_pt */
                           MP_ProtoDict,      /* the dictionary defining */
                           MP_CopProtoArray,  /* the array operator */
                           1,                 /* One annot */
                           1000);             /* 1000 args */
/* (2) Put the prototype annotation packet */
MP_PutAnnotationPacket(link,
                       MP_ProtoDict,          /* the dictionary defining */
                       MP_AnnotProtoPrototype,/* the prototype annot */
                       MP_AnnotValuatedFlag & MP_AnnotRequiredFlag);
/* (3) Put TypeSpec for the data limbs */
MP_PutCommonMetaTypePacket(link,
                           MP_ProtoDict,
                           MP_CmtProtoIMP_Real32,
                           0);         /* no annots to Cmt */
/* (4) Put the data limbs */
for (i=0; i<1000; i++) IMP_PutReal32(link, r32_array[i]);

/* A more efficient way would actually be to use:
 * IMP_PutReal32Vector(link, r32_array, 1000);
 * instead of putting each data limb separately
 */
```

Figure 2: Code fragment for putting an array of MP_Real32's

3.2.1 Prototype Operator TypeSpec

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

Struct TypeSpec

<**Struct TypeSpec**>::= Cop:Proto::Struct:n <MP TypeSpec>$_n$
| Cop:Proto::RecStruct:n <MP TypeSpec>$_n$

A collection of (possibly heterogeneous) objects that are to be treated as logically related is described using a Cop:Proto::Struct:n or a Cop:Proto::RecStruct:n operator node packet. The only difference between these operators is that a Cop:Proto::RecStruct:n operator is used for designating a structure as a target for a recursive type specification (see section 3.2.3). The number-of-arguments field specifies both the number of actual arguments the operator has in the prototype tree *and* the number of corresponding data items to be transmitted at data communication time. The arguments to the Cop:Proto::Struct:n operator specify the types of the struct's fields individually using MP TypeSpecs, where the i^{th} argument TypeSpec specifies the type of the i^{th} field.

Figure 3 shows how an array of, say, three sparse matrix elements, is represented using a Struct TypeSpec.

```
struct {
   MP_Uint32    row_num;
   MP_Uint32    col_num;
   MP_Real64    val;
} array[3] = {{2,2,1.0},{2,3,1.0},{2,4,1.0}};
```

Figure 3: A sparse matrix

Type	Dict	Value	#Annot:Arg	Remarks
Cop	Proto	Array	1:3	(1) 1 annot and 3 elements
AP	Proto	Prototype		(2) start of the prototype
Cop	Proto	Struct	3:0	(3) each element has 3 args, describe them individually
Cmt	Proto	IMP_Uint32	0:	(4) 1st arg is the row spec
Cmt	Proto	IMP_Uint32	0:	(5) 2nd arg is the column spec
Cmt	Proto	IMP_Real64	0:	(6) 3rd arg, matrix element type
IMP_Uint32		2		(7) 1st struct, beginning of protoyped data tree
IMP_Uint32		2		
IMP_Real64		1.0		[2,2] = 1.0
IMP_Uint32		2		(8) 2nd struct
IMP_Uint32		3		
IMP_Real64		1.0		[2,1] = 1.0
IMP_Uint32		2		(9) 3rd struct
IMP_Uint32		4		
IMP_Real64		1.0		[2,4] = 1.0

The top-level operator on line 1 indicates that this is a three element array. The prototype (lines 2 - 6) specifies the type of each element. Line 3 says that each element is a 3-field structure and the arguments to this Struct operator (lines 4 - 6) specify the individual types of the arguments using common meta type (Cmt) packets. The prototyped data trees begin on line 7 with the first field of the first structure. The data for the second structure begins on line 8 and the third on line 9.

Union TypeSpec

<**Union TypeSpec**>::= Cop:Proto::Union:n <MP TypeSpec>$_n$ | Cop:Proto::RecUnion:n <MP TypeSpec>$_n$

A Cop:Proto::Union:n or Cop:Proto::RecUnion:n operator node packet is used to specify a union of n prototype trees (<MP TypeSpec>'s). Again, the only difference between these operators is that a Cop:Proto::RecUnion:n operator designates a union as a target for a recursive type specification (see section 3.2.3 for details). As for Struct and RecStruct, the index n indicates that each of the n arguments following the operator is again a prototype tree (MP TypeSpec). At data communication time, a union discriminator is first transmitted as an IMP_Uint32 followed by a prototyped data tree. The valid range for the union discrimination index is $1 \ldots n$. An index i between 1 and n specifies that the corresponding prototyped data tree is as specified by the i^{th} argument of the Cop:Proto::Union:n operator node packet.

The example in figure 4 illustrates the use of the Cop:Proto::Union:n.

```
union {
   MP_Uint32    uint;
   MP_Sint32    sint;
   MP_ApInt     apint;
} array[3];
```

Figure 4: An array of integers using the union operator

Type	Dict	Value	#Annot:Arg	Remarks
Cop	Proto	Array	1:3	
AP	Proto	Prototype		
Cop	Proto	Union	0:3	(1) 3 prototypes follow
Cmt	Proto	IMP_Uint32	0:	(2) 1st prototype: index 1
Cmt	Proto	IMP_Sint32	0:	(3) 2nd prototype: index 2
Cmt	Proto	IMP_ApInt	0:	(4) 3rd prototype: index 3
IMP_Uint32		2		(5) use prototype with index 2
IMP_Sint32		-245		1st data value IMP_Sint32
IMP_Uint32		1		(6) use prototype with index 1
IMP_Uint32		1057		2nd data value (IMP_Uint32)
IMP_Uint32		3		(7) use prototype with index 3
IMP_ApInt		1234567890		3rd data value (IMP_ApInt)

The union operator on line 1 indicates that there are three prototypes (fields) in the union. These are given on lines 2 - 4. The prototyped data tree begin on line 5 with an IMP_Uint32 serving as union discriminator. The value is 1, indicating that the prototype with index 1 (defined on line 3) specifies the type of the following data limb. Once a receiver has consumed the data according to the prototype, it is positioned to read the next discriminator on line 6. This would proceed until all 3 arguments had been read.

3.2.2 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 tree by specifying them through *the prototype operator's arguments*. Unfortunately, this approach does not work for objects which are a repetition of a single type specification (as in 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 separate instances of the object within the tree (a pointer or ragged array, for example).

These cases are handled using a combination of the CommonMetaType and CommonOperator node packets: `MP_MetaOperator` (Mop) and `MP_CommonMetaOperator` (Cmop). Meta operator node packets may only meaningfully appear in a prototype tree where they serve as a "place holder" for an operator which otherwise would appear at data communication time. Meta operators have two defining characteristics which distinguish them from operator and meta type packets. First, unlike a meta *type* packet, a meta *operator* packet does not specify a leaf in the corresponding prototyped data tree. Instead, a meta operator specifies an inner node (operator) in the prototyped data tree where the communication of the node (operator) itself is shifted from data communication time to prototype specification time. Second, unlike an operator packet, a meta operator packet has *no* actual arguments. Instead, the number-of-arguments-field encodes *information about* the *number of arguments* which are communicated at the corresponding place in the prototyped data tree. There are two cases to consider:

1. A meta operator node packet with a non-zero number-of-arguments field. This is the number of arguments transmitted at data communication time.

2. A meta operator node packet with 0 in the number-of-arguments field. The number of arguments to the operator is delayed from prototype communication time to data communication time. The actual number of arguments is transmitted at data communication time by an IMP_Uint32 data limb before the arguments are transmitted (i.e., at the place where the operator would have appeared at data communication time).

Furthermore, annotations given to a meta operator at prototype specification time fullfill the role of the annotations which otherwise would have been given to the actual operator at data communication time. This applies in particular to the prototype annotation: If a meta operator appearing in a prototype tree has an attached prototype annotation, then the corresponding arguments appearing at data communication time are of the type specified by the prototype tree following the prototype annotation. An example where annotations are attached to meta operators is shown in figure 10.

The concept of type specifications based on meta operators is very powerful and flexible. It can be used, among others, to specify the structure and content of data structures built from pointers and (even- and uneven–length) arrays, which we illustrate in the following two examples.

Example 1: prototyped arrays of even-length arrays

As a first example, consider a matrix of reals given as an array of arrays[3]. In this case, the prototype consists of a common meta operator whose value is `MP_CopProtoArray` and whose number-of-arguments field is non-zero. The common meta operator has a prototype annotation specifying the type of each argument to the `MP_CopProtoArray` operator. Notice again, that the number-of-arguments field of the meta operator specifies the number of data values that *appear at data communication time* and *not* the number of arguments that follow the meta operator within the prototype tree. Figure 5 gives prototyped and non-prototyped versions side-by-side for comparison. The top-level operator on line 1 specifies an array with 2 arguments. The prototype annotation starting on line 2 specifies the element type of the top-level array to be a 3 element array (line 3). So each element of the array from line 1 is another array. The meta operator on line 3 also has a prototype (lines 4 - 5) specifying that the elements of each subarray is an `IMP_Real32`. Note that the meta operator on line 3 does not have any actual arguments within the prototype tree. The number-of-arguments field specifies the number of arguments to expect in the prototyped data tree that follows.

Example 2: prototyped arrays of uneven-length arrays

As a second example, let us suppose that the subarrays in the example above have different lengths, say one has length 2 and one has length 3. In this case we need to use a meta operator node packet with 0 in the number-of-arguments field as shown in figure 6.

As in the first example, the prototype (lines 2 – 5) specifies that each element of the top-level array is again an array of IMP_Real32's. However, the 0 in the number-of-arguments field of the meta operator in line 3 specifies that the actual number of elements of the subarrays is not given at prototype specification time but is instead given at data communication time by an `IMP_Uint32` data limb *in the place where the operator would have appeared* (lines 6 and 7).

[3]This example is illustrative only. Two-dimensional matrices should be communicated as defined in the matrix dictionary. For example, using the `MP_CopMatrixDenseMatrix` with an attached prototype annotation specifying the type of the basic matrix elements and a matrix-dimension annotation giving the dimensions of the matrix.

Figure 5: An array of even-lengths arrays

```
MP_Real32_t array[2][3] =
{
  { -1.0, -2.0, -3.0 },
  { -4.0, -5.0, -6.0 }
};
```

Prototyped version					Non-Prototyped version			
Type	Dict	Value	#Arg:Annot	Remarks	Type	Dict	Value	#Annot:Arg
Cop	Proto	Array	1:2	(1) a 2 element array	Cop	Proto	Array	0:2
AP	Proto	Prototype		(2) element type				
Cmop	Proto	Array	1:3	(3) elements are arrays				
AP	Proto	Prototype		(4) subarray elem type				
Cmt	Proto	IMP_Real32	0:	(5) is IMP_Real32				
					Cop	Proto	Array	0:3
IMP_Real32		-1.0		1st elem, 1st subarray	MP_Real32		-1.0	0:
IMP_Real32		-2.0			MP_Real32		-2.0	0:
IMP_Real32		-3.0			MP_Real32		-3.0	0:
					Cop	Proto	Array	0:3
IMP_Real32		-4.0		1st elem, 2nd subarray	MP_Real32		-4.0	0:
IMP_Real32		-5.0			MP_Real32		-5.0	0:
IMP_Real32		-6.0			MP_Real32		-6.0	

Example 3: pointer type specifications

As a third example, let us consider how the concept of pointers can be realized by type specifications. For this, we use a meta operator (typically, we would use the operator `MP_CopProtoPointer` provided in the prototype dictionary[4]) which has 0 in its number-of-argument field and a prototype annotation attached to the meta operator specifying the type of the data "pointed to". The corresponding IMP_Uint32 which is transmitted at data communication and which precedes the data "pointed to", may then have the value zero (pointer is `NULL`) or one (pointer is not `NULL`).

Consider an array of two `struct1` structures as defined below:

```
struct struct2 {
  MP_String   a;
  MP_Uint32   b;
}

struct struct1 {
  MP_Sint32         x;
  MP_Real32         y;
  struct struct2 *  struct2ptr;  // a ptr to struct2
}
```

and the following data:

```
{   {456, 90.12, NULL        },        - First structure
  {71 ,    2.1, &{"Blue", 2}}        - Second structure
}
```

[4]Recall that the `MP_CopProtoPointer` and `MP_CopProtoArray` operator values are only provided in the `MP_ProtoDict` as a convenience for the users of MP and that they play no special role within prototype trees, unlike the `MP_CopProtoStruct` and `MP_CopProtoUnion` prototype operators described in section 3.2. Hence, it is a syntactic error if a **Cop**:Proto::Array or **Cop**:Proto::Pointer operator node packet appears in a prototype tree. Instead, only **Cmop**:Proto::Array or **Cmop**:Proto::Pointer common *meta* operator packets may appear in a valid prototype tree.

Figure 6: An array of uneven-lengths arrays

```
MP_Real32_t array[2][] =
{
  { -1.0, -2.0, -3.0 },
  { -4.0, -5.0}
};
```

Prototyped version					Non-Prototyped version			
Type	Dict	Value	#Annot:Arg	Remarks	Type	Dict	Value	#Annot:Arg
Cop	Proto	Array	1:2	(1) a 2 element array	Cop	Proto	Array	0:2
AP	Proto	Prototype		(2) element type				
Cmop	Proto	Array	1:0	(3) elements are arrays				
AP	Proto	Prototype		(4) subarray elem type				
Cmt	Proto	IMP_Real32	0:	(5) is IMP_Real32				
IMP_Uint32		3		(6) 1st length	Cop	Proto	Array	0:3
IMP_Real32		-1.0		1st elem, 1st subarray	MP_Real32		-1.0	0:
IMP_Real32		-2.0			MP_Real32		-2.0	0:
IMP_Real32		-3.0			MP_Real32		-3.0	0:
IMP_Uint32		2		(7) 2nd length	Cop	Proto	Array	0:2
IMP_Real32		-4.0		1st elem, 2nd subarray	MP_Real32		-4.0	0:
IMP_Real32		-5.0			MP_Real32		-5.0	0:

where the structure that is preceeded by the `&` sign indicates a "pointer to" that structure.

Figure 7 shows what this would look like in MP. The first prototype on line 1 indicates that each element of the array is a 3-field structure. Line 3 indicates that the third field is a pointer to an object. Lines 4 - 7 give the prototype describing the object pointed to on line 3. The structuring prototype on line 5 says that the object pointed to on line 3 is a 2-field structure. The prototyped data tree follows the prototype tree beginning on line 8. The third field of the first element of the array appears on line 9. This is the field corresponding to the structure pointer from line 3. The value here is 0, indicating a `NULL` pointer, so the receiver skips the nested prototype (lines 4 - 7) describing the structure pointed to. However, the value for the pointer field for the second array element on line 10 is 1 (non-`NULL`), so the receiver uses the prototype from lines 4 - 7 to read the object pointed to: Line 6 tells the receiver to read a String (found on line 11), and line 7 says to read a Uint32 (found on line 12). There were only two elements to the array, so the end of the MP Tree has been reached.

3.2.3 Recursive type specifications

To realize recursion in its most general form, we would need mechanisms to attach a "label" to any node in a prototype tree so that we could refer to it by name from other points in the prototype tree. While being very powerful, this approach makes parsing of MP data considerably more complicated. It would require a receiver to watch for "labeled" prototype nodes and to dynamically maintain a name space of "labels" and their associated prototype nodes. Since the data objects we have considered in practice do not require such a general mechanism to express their recursive structure, we currently support only a restricted (static) form of recursion. Instead of attaching a "label" (name) to any node in a prototype tree, we only allow "static labeling" of structures and union prototype operators and provide common meta type values to refer back to the previosuly designated targets of a recursive union or structure specification.

More precisely, if a **Cmt**:Proto:RecStruct (resp. Cmt:Proto:RecUnion) common meta type node packet appears in a prototype tree, then the corresponding data (transmitted at data communication time) is of the type as specified by the **Cop**:Proto:RecStruct (resp. Cop:Proto:RecUnion) common operator packet which, under lexical scope within the same prototype tree, preceded the recursive meta type node packet. By "lexical scope within the same prototype tree" we mean that a recursive meta type node packet must have been preceded by a corresponding recursive prototype operator packet within the same prototype tree, and that the "back reference" is made to the

Figure 7: Array of structures containing pointers

Type	Dict	Value	#Annot:Arg	Remarks
Cop	Proto	Array	1:2	A 2 element array
AP	Proto	Prototype		(1) prototype #1
Cop	Proto	Struct	0:3	(2) each element is a structure (`struct1`)
Cmt	Proto	IMP_Sint32	0:	
Cmt	Proto	IMP_Real32	0:	
Cmop	Proto	Pointer	1:0	(3) 3rd element is a pointer
AP	Proto	Prototype		(4) prototype #2 describing structure of thing pointed to
Cop	Proto	Struct	2:0	(5) it is a structure of 2 things (`struct2`)
Cmt	Proto	IMP_String	0:	(6)
Cmt	Proto	IMP_Uint32	0:	(7)
IMP_Sint32		456		(8) 1st instance of `struct1` - retrieve using prototype #1
IMP_Real32		90.12		
IMP_Uint32		0		(9) Value for Pointer is NULL, so end of 1st `struct1`
IMP_Sint32		71		2nd instance of `struct1`
IMP_Real32		2.1		
IMP_Uint32		1		(10) Value for Pointer is non-NULL, get the struct pointed to (`struct2`)
IMP_String		“Blue”		(11) retrieve using prototype #2
IMP_Uint32		2		(12)

one which is most closely nested under under lexical scoping rules. So, a **Cmt**:Proto::RecStruct points back to the most closely nested **Cop**:Proto::RecStruct, and similarly for RecUnion.

Using recursive metatype node packets and recursive prototype operator node packets we obtain sufficiently powerful means to realize recursive type specification for most cases without having to introduce a dynamic name space for node labels and prototype nodes.

The simple example of a linked-list below illustrates the concepts (also see figure 8).

```
struct recurse_ex {
  MP_Sint32             a;
  MP_Real32             b;
  struct recurse_ex *   recurse_ptr;  // self-referencing pointer
}
```

Figure 8: Recursive structure

Type	Dict	Value	#Annot:Arg	Remarks
Cop	Proto	Array	1:1	
AP	Proto	Prototype		(1)
Cop	Proto	RecStruct	0:3	(2) defining a recursive structure
Cmt	Proto	IMP_Sint32	0:	(3)
Cmt	Proto	IMP_Real32	0:	(4)
Cmop	Proto	Pointer	1:0	(5) pointer
AP	Proto	Prototype		(6)
Cmt	Proto	RecStruct	0:	(7) points to struct from line (2)
IMP_Sint32		10		(8) use prototype from lines (1 - 7)
IMP_Real32		2.3		
IMP_Uint32		1		(9) recursive pointer, 1 indicates another struct follows
IMP_Sint32		20		(10) use prototype from lines (1 - 7)
IMP_Real32		6.5		(11)
IMP_Uint32		0		(12) 2nd struct ends here, 0 indicates no more follow

The prototype on lines 1 - 7 describes a recursive data structure. The Cop:Proto::RecStruct operator on line 2 indicates each argument of the top-level Cop:Proto:Array operator is a structure of three elements and designates this structure for further recursive references (i.e., gives it a "static" recursion label to the structure). The third field of this structure on line 5 is a meta operator node packet (used as a pointer), whose prototype on lines 6 - 7 specifies that this is a recursive pointer "pointing to" the most closely nested `MP_RecStruct` within the prototype tree, which is on line 2. Note that the pointer's number-of-arguments field is 0 and that omitting lines 5 and 6 would result in an incorrect "endless recursive" type specification. The prototyped data tree immediately follows the prototype tree (line 8 on). The first occurrence of the recursive pointer field appears on line 9. Here the actual number-of-arguments of the Cmop:Proto::Pointer is found. Since it is 1 (non-`NULL`), another instance of this structure follows (lines 10 - 12). The recursive pointer field of this second structure is shown on line 12. Here the value is 0 indicating that the pointer is `NULL`. Since this is a recursive data structure, we have reached the end of the "linked list" and no more data follows.

Also see section 3.4 for a more complex examples illustrating the use of recursive type specifications.

3.3 Tree TypeSpecs

Prototyped data which is specified using only Basic and Operator TypeSpec's consist of a collection of data limbs. In contrast, a Tree Typespec specifies that the corresponding prototyped data is a "normal" MP tree (i.e., consists of node and annotation packets). A Tree Typespec is a useful means to indicate that the corresponding data trees have certain (most often syntactic) properties.

More precisely, a Tree TypeSpec is accomplished by a (Common) Meta Type which can be defined in any dictionary and which indicates that the corresponding prototyped data is communicated as an MP Tree that has a certain property, as defined in the respective dictionary.

The restriction to allow the data corresponding to a Tree TypeSpec to only consist of *full node packets* (of *syntactically correct MP Trees*) stems from the fact that we require MP data to always be parsable on a syntactic level. Only the data corresponding to the meta types defined in the prototype dictionary may be transmitted as data limbs.

Tree TypeSpec's can nevertheless be used to communicate the corresponding data objects more efficiently: First, we can attach annotations to the meta types at prototype specification time, which then apply to all instances of the corresponding node packets at data communication time; and, second, a receiver may use the additional provided syntactic meta information to parse incoming data more efficiently.

As an example, we show in table 9 the definition of Rational and Integer common meta types, as done in the "Numbers" dictionary (`MP_Number`).

Figure 9: Commom meta number types and their communication time equivalents

Cmt	Possible node packet types at communication time
`MP_CmtNumberInteger`	MP_Sint32 | MP_Uint32 | MP_Sint8 | MP_Uint8 | MP_ApInt
`MP_Rational`	<MP_Integer> | Cop:Basic::Div:2 <MP_Integer> <MP_Integer>

The common meta types given in figure 9 can conveniently be used in a prototype tree to specify that the corresponding data are rational numbers. Furthermore, if they all have certain properties (such as being normalized), then we can attach this information as an annotation to the meta type in the prototype tree. The example in figure 10 shows how this mechanism is used to send an array of rationals. If a receiver "knows" what rational numbers are, then it could use compiled routines which read them more efficiently based on the prototype specification in line 1. Furthermore, if a receiving application requires all rational numbers to be internally stored as normalized numbers, then by means of the annotation in line 2, it does not need to normalize the rationals received.

Figure 10: An array of rationals using user-defined meta types

Type	Dict	Value	#Annot:Arg	Remarks
Cop	Proto	Array	1:10	
AP	Proto	Prototype		
Cmt	Number	Rational	1:	(1) from the number dictionary – corresponding data are full node packets
AP	Number	Normalized		(2) all rational numbers are normalized
Cop	Basic	Div	0:2	1st rational
MP_Sint32		-2	0:	numerator
MP_Uint32		3	0:	denominator
MP_Apint		245	0:	2nd rational
MP_Apint		4593922	0:	3rd rational
MP_Uint32		1257	0:	4th rational
MP_Uint32		994	0:	5th rational
MP_Uint8		90	0:	6th rational
		...		and so on

3.4 Summarizing example: transmission of a sparse recursive polynomial

The following example combines `MP_RecUnion`, `MP_Struct`, recursive, and user-defined type specifications. The considered data structure is a recursive representation of a sparse polynomial. One possible declaration looks like:

```
union SparseRecPoly  {
  Rational_t;                  // coefficient - prototype index 0
  struct inner_poly{           //               prototype index 1
    MP_String_t;               // varname
    MP_Uint32_t;               // exponent
    union SparseRecPoly *;     // multiplic subpoly
    union SparseRecPoly *;     // additive subpoly
  };
};
```

where `Rational_t` is supposed to be a previously defiend type representing a rational number.

The sample data is:

```
9876321098x^4(y+2/3) + x^2  =  x^4    +     x^2
                               *
                               (y     +     2/3)
                               *
                               9876321098
```

Its encoding in MP is shown in the usual table form in figure 11 and as a source-code fragment using the MP API in figure 12.

Based on the recursive poly example and the rules governing meta operator type specifications, it is easy to develop the encodings of more complex polynomial data, such as arrays or matrices of polynomials, or the generators of a polynomial ideal. In these cases, the top-level Cop:Poly::SparseRecPoly:1 operator of the example above appears as a common meta operator in the TypeSpec of the top-level enclosing operator and the rest of the type specification is exactly as above. As a last example, we show how this is accomplished in figure 13.

References

[1] O. Bachmann, S. Gray, and H. Schönemann. 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.

[2] O. Bachmann, S. Gray, and H. Schönemann. A proposal for syntactic data integration for math protocols. In *to appear in the Proceedings of the International Symposium on Parallel Symbolic Computation (PASCO'97)*, Hawaii, USA, July 1997. .

[3] S. Gray, N. Kajler, and P. S. Wang. Design and Implementation of MP, a Protocol for Efficient Exchange of Mathematical Expressions. *Journal of Symbolic Computing*, 1997. Forthcoming.

[4] Sun Microsystems, Inc., Mountain View, CA. *Network Programming Guide (revision A)*, 1990. Part number 800-3850-10.

Figure 11: A sparse recursive polynomial

Type	Dict	Value	#Annot:Arg	Remarks
Cop	Poly	SparseRecPoly	1:1	top-level operator
AP	Proto	Prototype		
Cop	Proto	RecUnion	0:2	recursive union
Cmt	Number	MP_Rational	0:	user-defiend coeficient TypeSpec
Cop	Proto	Struct	0:4	inner_poly struct TypeSpec
Cmt	Proto	IMP_String	0:	var name TypeSpec
Cmt	Proto	IMP_Uint32	0:	exponent TypeSpec
Cmop	Proto	Pointer	1:0	pointer to
AP	Proto	Prototype		
Cmt	Proto	RecUnion	0:	mult. subpoly recursive TypeSpec
Cmop	Proto	Pointer	1:0	pointer to
AP	Proto	Prototype		
Cmt	Proto	RecUnion	0:	add. subpoly recursive TypeSpec
IMP_Uint32		2		use proto 2
IMP_String		x		the outermost var is x
IMP_Uint32		4		its exponent is 4
IMP_Uint32		1		mult. subpoly ptr is non-NULL
IMP_Uint32		2		mult. subpoly to `x^4` – use proto 2
IMP_String		y		the current var is y
IMP_Uint32		1		its exponent is 1
IMP_Uint32		1		mult. subpoly ptr is non-NULL
IMP_Uint32		1		mult. subpoly to y – use proto 1
MP_ApInt		9876321098	0:	coeff node packet
IMP_Uint32		1		add. subpoly ptr is non-NULL
IMP_Uint32		1		add. subpoly y – use proto 1
Cop	Basic	Div	0:2	coeff tree
MP_Sint32		2	0:	numerator
MP_Sint32		3	0:	denominator
IMP_Uint32		1		add. subpoly ptr is non-NULL
IMP_Uint32		2		add. subpoly to `x^4` – use proto 2
IMP_String		x		var is again x
IMP_Uint32		2		its exponent is now 2
IMP_Uint32		1		mult. subpoly ptr is non-NULL
IMP_Uint32		1		mult subpoly to `x^2` – use proto 1
MP_Sint32		1	0:	coeff node packet
IMP_Uint32		0		add. subpoly ptr is NULL `x^2` – use proto 0

```
MP_PutCommonOperatorPacket(link,          /* top-level operator */
                           MP_PolyDict, MP_CopPolySparseRecPoly, 1,1);
MP_PutAnnotationPacket(link,              /* prototype annot */
                       MP_ProtoDict, MP_AnnotProtoPrototype,
                       MP_AnnotValuatedFlag & MP_AnnotRequiredFlag);
MP_PutCommonOperatorPacket(link,          /* recursive union */
                           MP_ProtoDict, MP_CopProtoRecUnion, 0, 2);
MP_PutCommonMetaTypePacket(link,          /* coefficient TypeSpec */
                           MP_NumberDict, MP_CmtNumberRational, 0);
MP_PutCommonOperatorPacket(link,          /* inner_poly struct TypeSpec */
                           MP_ProtoDict, MP_CopProtoStruct, 0, 4);
MP_PutCommonMetaTypePacket(link,          /* var name TypeSpec */
                           MP_ProtoDict, MP_CmtProtoIMP_String, 0);
MP_PutCommonMetaTypePacket(link,          /* exponent TypeSpec */
                           MP_ProtoDict, MP_CmtProtoIMP_Uint32, 0);
MP_PutCommonMetaOperatorPacket(link,      /* mult. subpoly */
                           MP_ProtoDict, MP_CopProtoPointer, 1, 0);
MP_PutAnnotationPacket(link,              /* pointer to */
                       MP_ProtoDict, MP_AnnotProtoPrototype,
                       MP_AnnotValuatedFlag & MP_AnnotRequiredFlag);
MP_PutCommonMetaTypePacket(link,          /* recursivly preceeding union */
                           MP_ProtoDict, MP_CmtProtoRecUnion, 0);
MP_PutCommonMetaOperatorPacket(link,      /* add. subpoly */
                           MP_ProtoDict, MP_CopProtoPointer, 1, 0);
MP_PutAnnotationPacket(link,              /* pointer to */
                       MP_ProtoDict, MP_AnnotProtoPrototype,
                       MP_AnnotValuatedFlag & MP_AnnotRequiredFlag);
MP_PutCommonMetaTypePacket(link,          /* recursively preceeding union */
                           MP_ProtoDict, MP_CmtProtoRecUnion, 0);
/* Poly data begins */
IMP_PutUint32(link, 2);   /* use proto 2 */
IMP_PutString(link, "x"); /* the outermost var is x */
IMP_PutUint32(link, 4);   /* its exponent is 4 */
IMP_PutUint32(link, 1);   /* mult subpoly to x^4 exists */
IMP_PutUint32(link, 2);   /* mult subpoly to x^4 is poly -- use proto 2 */
IMP_PutString(link, "y"); /* the current var is y */
IMP_PutUint32(link, 1);   /* its exponent is 1 */
IMP_PutUint32(link, 1);   /* mult subpoly to y exists */
IMP_PutUint32(link, 1);   /* mult subpoly to y is number -- use proto 1 */
MP_PutApIntPacket(link, apint, 0); /* coeff apint which equals 9876321098 */
IMP_PutUint32(link, 1);   /* add subpoly to y exists */
IMP_PutUint32(link, 1);   /* add. subpoly to y is number -- use proto 1 */
MP_PutOperatorPacket(link,/* coeff tree corresponding to 2/3 */
                     MP_ElemAlgDict,  MP_CopBasicDiv, 0, 2);
MP_PutSint32Packet(link, 2, 0);
MP_PutUint32Packet(link, 3, 0);
IMP_PutUint32(link, 1);   /* add subpoly to x^4 exists */
IMP_PutUint32(link, 2);   /* add subpoly to x^4 is poly -- use proto 2 */
IMP_PutString(link, "x")  /* var is again x */
IMP_PutUint32(link, 2);   /* its exponent is now 2 */
IMP_PutUint32(link, 1);   /* mult subpoly to x^2 exists */
IMP_PutUint32(link, 1);   /* mult subpoly to x^2 is number -- use proto 1  */
MP_PutSint32(link, 1);    /* coeff node packet corresponding to 1 */
IMP_PutUint32(link, 0);   /* no add. subpoly to x^2 -- use ``NULL'' ptr */
```

Figure 12: Sending the polynomial $9876321098x^4(y+2/3)+x^2$ as a Sparse Recursive Polynomial using the MP API

Figure 13: An Ideal

Type	Dict	Value	#Annot:Arg	Remarks
Cop	Poly	Ideal	1:1	the ideal consists of
AP	Proto	MP_Prototype		
Cmop	Basic	Array	1:10	an array of 10 polys
AP	Proto	MP_Prototype		where each poly is a
Cmop	Poly	SparseRecPoly	1:1	sparse-rec poly – notice Cmop here
AP	Proto	Prototype		
Cop	Proto	RecUnion	0:2	recursive union
Cmt	Number	MP_Rational	0:	user-defiend coeficient TypeSpec
Cop	Proto	Struct	0:4	inner_poly struct TypeSpec
Cmt	Proto	IMP_String	0:	var name TypeSpec
Cmt	Proto	IMP_Uint32	0:	exponent TypeSpec
Cmop	Proto	Pointer	1:0	pointer to
AP	Proto	Prototype		
Cmt	Proto	RecUnion	0:	mult. subpoly recursive TypeSpec
Cmop	Proto	Pointer	1:0	pointer to
AP	Proto	Prototype		
Cmt	Proto	RecUnion	0:	add. subpoly recursive TypeSpec
data of the 10 polynomials goes here, starting with the prototype to be used for the first poly, etc				

List of papers published in the Reports on Computer Algebra series

[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:13] O. Bachmann. MPT – a library for parsing and Manipulating MP Trees. January 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:07] G.M. Greuel. Description of SINGULAR: A computer algebra system for singularity theory, algebraic geometry, and commutative algebra. 1996.

[RCA:06] G.M. Greuel and G. Pfister. Advances and improvements in the theory of standard bases and szyzygies. 1996.

[RCA:05] O. Bachmann, S. Gray, and H. Schönemann. MPP: A Framework for Distributed Polynomial Computations. 1996.

[RCA:04] O. Bachmann and H. Schönemann. A Manual for the MPP Dictionary and MPP Library. 1996.

[RCA:03] R. Stobbe. FACTORY: a C++ class library for multivariate polynomial arithmetic. 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.