An Approach to Object-Oriented Programming in C*

W. Windsteiger

December 29, 1990

1 Introduction

In pure C language no tools for object-oriented programming are available. This paper shows one possibility how to achieve some aspects of object-oriented programming in C. Since this should be used in Algebraic Computation, the most important feature one desires to have, is the possibility to use the same names for algorithms that perform the same actions. For example the multiplication algorithm should always be called *"Product"* regardless from which domain its arguments are (operator overloading).

One approach for a solution is the *Simulated Package Principle (SPP)*, which is treated detailed in [Buchberger 90]. The main difference to our solution is that in *SPP* all actions that have to be done to guarantee an "object-oriented behaviour" happen at compile-time whereas in this method a dispatch is done at run-time. Roughly spoken, this is done by wrapping a structure around every object that contains information about all functions that are applicable to this particular object.

Since huge computations should be carried out (e.g. Gröbner bases), storage management has to be considered, too. The problem that occurs is the following: When one allocates memory one does not know when it may be freed again. This is mainly due to the fact that in Algebraic Computation there is a lot of overlap of different data domains. Let us illustrate this fact by a short example: Consider polynomials as a list of monomials. If one wants to split a polynomial into *Leading Monomial* and *Remaining Polynomial* it is not preferable to copy the whole list. So only pointers to the already existing list elements are set. This results in the fact that during computation there are a lot of elements with more than one pointer to themselves. Therefore it is a problem when to free these elements.

In section 3 one proposal how to solve these problems is discussed, in section 4 one can find a description how this can then be used for including own domains into the whole environment with generic functions and storage management. In section 5 this

*This research work was done in the frame of the Gröbner Bases project sponsored by the *Bundesministerium für Wissenschaft und Forschung*, Austria.
Approved by: Dr.Bruno Buchberger and Dr.Franz Lichtenberger.

is illustrated guided by the implementation of *one* function, in section 7.1 the code for integers, power-products represented by integer-exponent-arrays monomials and lists is listed.

2 The Problem

Mathematicians use the same names for algorithms over different data domains because by inspecting the arguments it can be decided which algorithm has to be used. This is not possible in conventional programming languages where two different routines must have different names. The desire to overload names has mainly two reasons:

- Some algorithms can be implemented *independently of the data domain*, e.g. the multiplication algorithm for polynomials does not depend on the domain of coefficients and

- the algorithms should be *independent of the concrete data representation* that is chosen, e.g. the multiplication algorithm for polynomials should be the same regardless whether monomials are represented as lists, arrays, structures or whatever.

The advantages are obvious:

- The amount of code is reduced because algorithms do not have to be implemented separately for each combination of domains (polynomials over the rationals, over finite fields, over rational functions,...) and

- if once a hierarchy of domains is established, algorithms are independent of changes in the underlying domains, i.e. the "interface" between the domains can be defined easily.

2.1 Restrictions To Be Fulfilled

The main restriction is that the source code should be as readable and as natural as possible regarding the fact that the produced software should be distributed together with a book and therefore the source code should be intimately connected with the written text. Furthermore, the code should be easily extendable by customers who do not know all the details about the system. This implies that the mechanism, how generic behavior is guaranteed must not be too complicated, i.e. the overhead for the programmer, who wants to add own routines, has to be minimal.

Nevertheless, speed must not suffer all to much from the advantages one gains. It is clear that one will not get all this for free, but it is one of the reasons not to take a complete object-oriented system (e.g. *PCL*) because it turned out that the price one has to pay is too high (see [Windsteiger 90] for timings with *PCL*). We try to minimize the loss of velocity by just implementing these aspects of generic programming, which we really need.

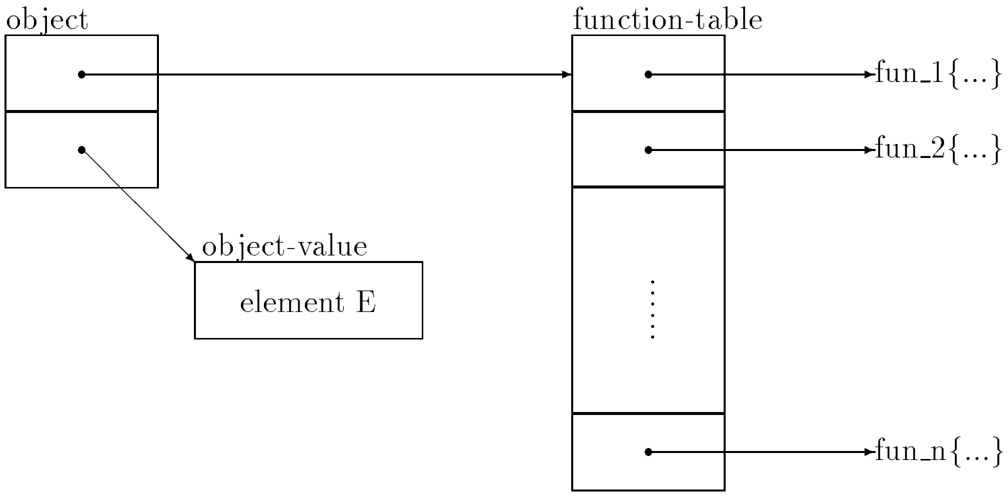

Figure 1: Object Structure

3 The Solution

3.1 First Approach

A solution for the problems stated above is to wrap a structure around every element containing two pieces of information as shown in Figure 1:

1. a pointer to an array of functions (in fact it is an array of pointers to functions) that are applicable to the element and
2. a pointer to the element itself, which has to be of type `void*` because it may point to various types.

The types are declared as follows:

```
#define MAXFUN 20

typedef void* ((*PF)()); /* Pointer to function returning (void*) */

typedef PF Operations[MAXFUN]; /* Array of pointers to functions
                                  returning (void*)  */

typedef struct   /* abstract data type for all objects */
  {
  PF  *type;
  void *value;
  } object;
```

The functions f_1 to f_n in Figure 1 are those functions that are applicable to the element E and are called *element-functions*. Note that the array of function-pointers has to be stored only once per class of elements. In the sequel this array is called *function-table*, the element E is called *object-value* and the structure wrapped around is called *object*.

The following conditions must hold:

- The index of the pointer to an element-function has to be fixed within the function-table, e.g. the index of the pointer to the *addition-function* must be 2 in *every* domain. These indices have to be fixed by the programmer.
- If in a domain a function does not exist, the pointer at the corresponding index in the function-table must be the null-pointer, e.g. in the domain *exponent array* there is no addition and therefore the second element in the function-table has to be the null-pointer.
- For every function that should be used generic a *Generic Function* has to be supplied. All these Generic Functions have a common pattern. When getting an object as an argument, the corresponding pointer in the function-table is evaluated and stored in the variable *fun*. In the case where this is not the null-pointer, *fun* is called with the *object-value* as the argument otherwise an error is detected.
- The element-functions must have `void*` arguments because they are called with the *object-values* as parameters, which are declared as `void*`.

Therefore a typical Generic Function looks as follows:

```
#define SUM 2

object *Sum( object *o1, object *o2)
  {
  PF fun;
  if( fun = (o1->type)[ SUM])
    return fun( o1->value, o2->value);
  printf("\nNo function Sum available for this object\n");
  }
```

3.2 Advanced Solution

As already discussed in section 1, one has the problem when to free memory. We chose a mechanism of *Reference Counting* for solving this problem. All objects as well as all types that are defined by the user (arrays, structures,...) must have an additional slot for a reference counter. The reference counter is an integer containing the number of pointers that have a reference to the particular object, array or structure.
Advanced object declaration:

```
typedef struct   /* abstract data type for all objects */
  {
  PF  *type;
  void *value;
  int ref_count;
  } object;
```

In addition, the assignment-operation for objects has to be defined. Each time an object gets a new value,

1. the old value can be freed and
2. the reference counter of the new value has to be increased by one.

Therefore, each object has to be initialized by zero to indicate that there is no old value to be freed.
Assignment operation for objects:

```
void Set( object **o1, object *o2)
  {
  if( *o1) Free( *o1);
  *o1 = o2;
  o2->ref_count++;
  }
```

Freeing an object is done as follows:

1. Decrease the reference counter by one and
2. if the reference counter is zero then, free the object-value recursively otherwise nothing has to be done.

Moreover, memory is not freed immediately but the pointer to the memory-space to be freed is put into an array of such pointers, the *free-array*. Hence, if new memory has to be allocated, this can then be easily done by just taking a pointer from the free-array, which has to be supplied for each type one defines. We illustrate this for *objects*. The global variable `next_object` stores the index in the free-array where a pointer to available memory-space can be taken. If no pointer is in the free-array, memory must be allocated explicitly by `malloc`.

```
object *allocate_object( void)
  {
  if( next_object >= 0)
    return free_object_array[ next_object--];
  else
    return (object*)malloc( sizeof( object));
  }
```

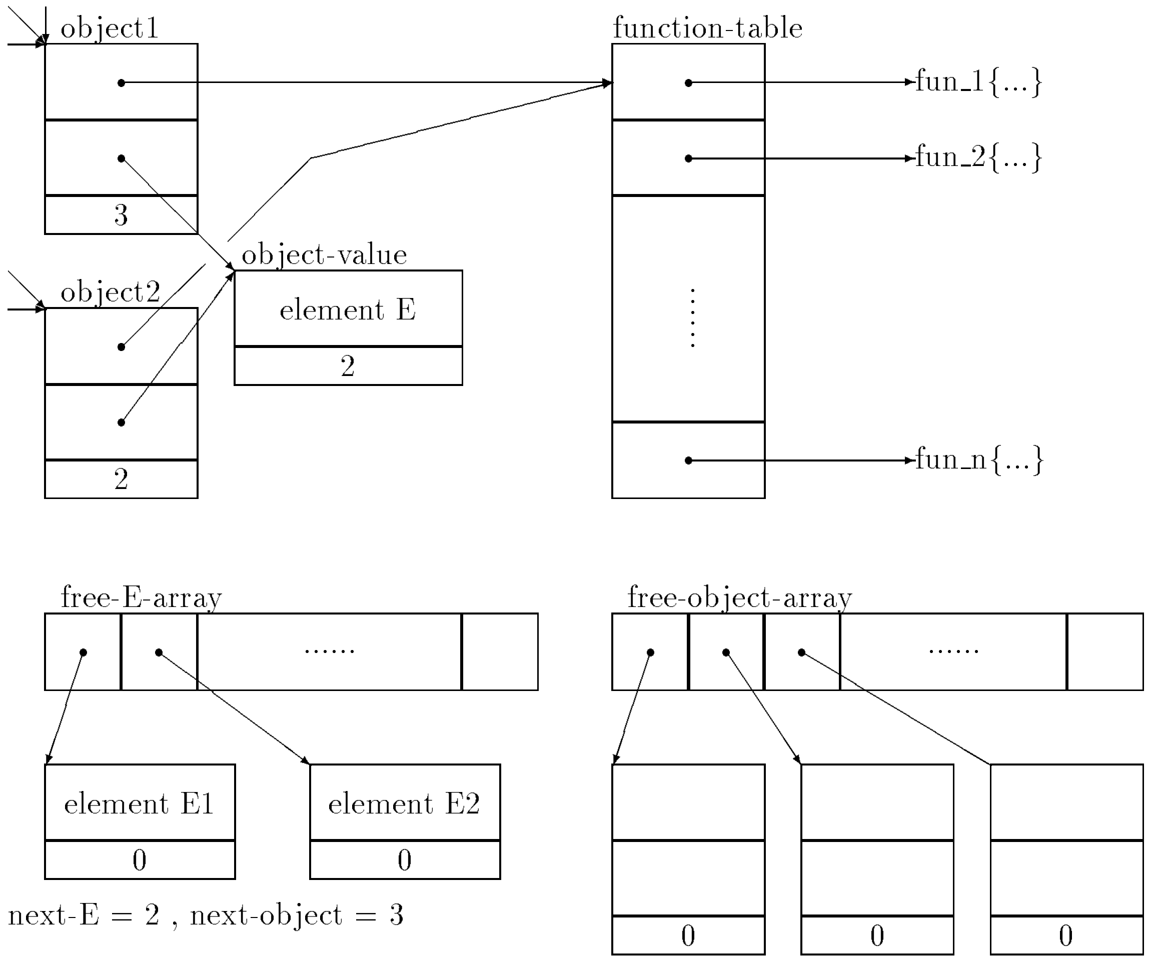

Figure 2: Advanced Object Structure

The size of the free-array is specified by the global variable `MAXFREE`, which is defined in the header file "generic.h". If the free-array for a particular type is already full, space must be freed explicitly by `free`.

```
void Free( object *o)
  {
  if( !( -- o->ref_count))
    {
    (o->type)[ FREE]( o->value);
    if( ++next_object < MAXFREE)
      free_object_array[ next_object] = o;
    else free(o);
    }
  }
```

4 Description for the User

This section describes, which functions the programmer has to make available for self-defined domains in order to guarantee that this domain fits into the whole system, i.e. that reference counting as well as generic function calls are provided for this domain. Moreover, it contains an "algorithm", how to implement self-defined domains. For details on a concrete implementation see section 5.

4.1 Type Definitions

The only restriction concerning the type definition is that it has to contain a reference counter. It is the decision of the user to implement the reference counter as a slot in a structure or an additional array element or whatever.

4.2 Function Prototypes

Element-functions must return a pointer to an object or `void` if they are used as procedures. The arguments have to be of type `void*` and must afterwards be cast to the respective type.

4.3 Global Variables

For each domain one has to provide a global variable of type `Operations` (the function-table for this domain), an array of size `MAXFREE` of pointers to elements of the domain (the free-array for this domain) and the integer variable, which contains the index of the next free element that can be allocated. The free-array and the integer variable are only needed if one decides to use the allocate-free-mechanism described in section 3.2.

4.4 Assignment Operation

The generic assignment operation `Set` must be used whenever an object is assigned to another object in order to invoke reference counting.

4.5 Additional Functions

The following functions must be available for each domain:

1. A *generation function*: This function gets a pointer to a domain element as an argument and returns a pointer to the whole object. In the object the pointer to the function-table and the pointer to the object-value have to be initialized, the reference counter of the produced object must be set to zero and the reference counter of the object-value has to be increased by one. Since this function is local to each domain it need not be put into the function-table.

2. An *initialization function* that initializes the function-table and the global integer variable mentioned in section 4.3.

Furthermore, for the allocate-free-mechanism are necessary:

- An *allocate-function* as described in section 3.2.
- A *free-function* as described in section 3.2. The index in the function-table must be `FREE`, a global constant that is defined in the header file "generic.h".

4.6 "Algorithm" for Implementing New Domains

In this section we start from the following situation: The element-functions are already available but we still have to adapt them to fit into our system. We consider the example of multiplication of monomials. The implementation we start from is the following:

```
typedef int *iea;

typedef struct
  {
  int coef;
  iea pp;   /* concrete representation of power-products */
  } monomial;

typedef monomial *mon;

mon new_m()
  {
  return (mon)malloc( sizeof( monomial));
  }

mon product_m( mon m1, mon m2)
  {
  mon m0;

  m0 = new_m();

  m0->coef = m1->coef * m2->coef;
  m0->pp   = product_iea( m1->pp, m2->pp);

  return m0;
  }
```

The disadvantages are obvious: The type-definition of `monomial` depends on the type of the coefficients and the concrete representation of power-products. Hence, if one wants to study another domain of coefficients or another representation of power-products one has to change the definition as well as the code of all functions. The multiplication of the coefficients (*) and the power-products (product_iea) must be altered to the multiplication functions for the new coefficient domain and the new representation of power-products. The following changes are necessary to avoid these disadvantages:

1. Change type definition of monomial into

```
typedef struct
  {
  object *coef;
  object *pp;
  int ref_count;
  } monomial;
```

2. Change the function prototype:

```
object *product_mon( void*, void*);
```

3. Introduce the global variables as stated in section 4.3.

4. Substitute every occurrence of the assignment operator '=' by the generic procedure Set (if '=' was used with objects.).

5. Implement the additional functions as stated in section 4.5. On implementing the initialization function one has to consult the header file "generic.h". In this file one finds a list of all generic functions that are available. If the considered generic functions already exist nothing more has to be done. Otherwise one has to introduce a new generic function as described in all detail in section 3.1 and add this function to these in the file "generic.c". If necessary the global variable MAXFUN has to be increased.

5 The Resulting Code

All functions that are not needed for the product of monomials are left out.

```
/* *************** GENERIC.H ***************** */

/* ************ Global Constants ************* */

#define MAXFUN 20
#define MAXFREE 200

typedef void* ((*PF)()); /* Pointer to function returning (void*) */
```

```
typedef PF Operations[MAXFUN]; /* Array of pointers to functions
                                  returning (void*)  */

typedef struct   /* abstract data type for all objects */
  {
  PF  *type;
  void *value;
  int ref_count;
  } object;

/* ************ Global Constants ************* */

#define PRODUCT 5
#define FREE 14

/* ************ Global Variables ************* */

object* free_object_array[ MAXFREE]; /* free-array */
int next_object;

/* ************ Function Declarations ************* */

void Set( object**, object*); /* Assignment Operation */

object *Product( object*, object*);
void Free( object*);
object *allocate_object( void);

/* *************** GENERIC.C **************** */

void Set( object **o1, object *o2)
  {
  if( *o1) Free( *o1);
  *o1 = o2;
  o2->ref_count++;
  }

object *Product( object *o1, object *o2)
  {
  PF fun;
  if( fun = (o1->type)[ PRODUCT])
    return fun( o1->value, o2->value);
  printf("\nNo function Product available for this object\n");
  }
```

```
void Free( object *o)
  {
  if( !( -- o->ref_count))
    {
    (o->type)[ FREE]( o->value);
    if( ++next_object < MAXFREE)
      free_object_array[ next_object] = o;
    else free(o);
    }
  }

object *allocate_object( void)
  {
  if( next_object >= 0)
    return free_object_array[ next_object--];
  else
    return (object*)malloc( sizeof( object));
  }

/* *************** MONOMIAL.H ***************** */

typedef struct
  {
  object *coef;
  object *pp;
  int ref_count;
  } monomial;

typedef monomial *mon;

Operations mon_funct; /* function-table */

object *make_mon( mon);  /* generation function */
mon allocate_mon( void);
mon new_mon( void);
void free_mon( mon);
object *product_mon( void*, void*);
void init_mon( void);

/* *************** MONOMIAL.C ***************** */

/* Global Variables */

mon free_mon_array[ MAXFREE]; /* free-array  */
int next_mon;
```

```
object *make_mon( mon m)
  {
  object *obj_ptr;
  obj_ptr = allocate_object();

  obj_ptr->type  = mon_funct;
  obj_ptr->value = (void*)m;
  obj_ptr->ref_count = 0;
  m->ref_count++;

  return obj_ptr;
  }

mon allocate_mon()
  {
  if( next_mon >= 0)
    return free_mon_array[ next_mon--];
  else
    return (mon)malloc( sizeof( monomial));
  }

mon new_mon()
  {
  mon m0;
  m0 = allocate_mon();
  m0->coef = (object*)0;
  m0->pp = (object*)0;
  m0->ref_count = 0;

  return m0;
  }

void free_mon( mon m)
  {
  if( !( -- m->ref_count))
    {
    Free( m->coef);
    Free( m->pp);

    if( ++next_mon < MAXFREE)
      free_mon_array[ next_mon] = m;
    else free( m);
    }
  }

object *product_mon( void *v1, void *v2)
```

```
  {
  mon m0, m1, m2;

  m0 = new_mon();
  m1 = (mon)v1;
  m2 = (mon)v2;

  Set( &m0->coef, Product( m1->coef, m2->coef));
  Set( &m0->pp, Product( m1->pp, m2->pp));

  return make_mon( m0);
  }

void init_mon( void)
  {
  next_mon = -1;

  mon_funct[ PRODUCT] = product_mon;
  mon_funct[ FREE] = free_mon;
  }
```

6 Summary

The code given in the previous section for the multiplication of monomials seems to be very long but all the work has to be done only once per new domain. Implementing the next element-function for monomials needs no more additional routines. In addition, if some domains are implemented most of the generic functions one needs are already available. The generic function calls could be improved because in the actual version it is assumed that all arguments of a function are of the same domain and due to this one only has to call the appropriate element-function stored in the function-table of the first argument. In an improved version one could check whether the arguments are of the same type and one could think about a strategy how to find the correct element-function in the case where the arguments do not have the same type.

7 Appendix

7.1 Source Code

Generic functions can be easily recognized in the code because these are the only function names beginning with an upper-case letter. The generic functions (except `Sum` and `Print`) do not offer the error handling as described in section 3.1. The pattern how this can be done for all generic functions can be seen in `Sum` and `Print` as well.

```
/* *************** GENERIC.H **************** */
```

```
/* ************ Global Constants ************* */

#define MAXFUN 20
#define MAXFREE 200

/* ************ Type Definitions ************* */

typedef void* ((*PF)()); /* Pointer to function returning (void*) */

typedef PF Operations[MAXFUN]; /* Array of pointers to functions
                                  returning (void*)  */

typedef struct   /* abstract data type for all objects */
  {
  PF  *type;
  void *value;
  int ref_count;
  } object;

/* ************ Global Constants ************* */

#define FALSE ((object *)0)
#define TRUE ((object *)1)

#define ZERO 0
#define ONE 1
#define SUM 2
#define NEGATIVE 3
#define DIFFERENCE 4
#define PRODUCT 5
#define INVERSE 6
#define QUOTIENT 7
#define ISEQUAL 8
#define L_C_M 9
#define ISMULTIPLE 10
#define TDEG 11
#define TDEGGR 12
#define PRINT 13
#define FREE 14

/* ************ Global Variables ************* */

object* free_object_array[ MAXFREE];
int next_object;
```

```
/* ************ Function Declarations ************* */

void Set( object**, object*); /* assignment operation */
object *allocate_object( void);

object *Zero( object*);
object *One( object*);
object *Sum( object*, object*);
object *Negative( object*);
object *Difference( object*, object*);
object *Product( object*, object*);
object *Inverse( object*);
object *Quotient( object*, object*);
object *Is_Equal( object*, object*);
object *LCM( object*, object*);
object *Is_Multiple( object*, object*);
object *Total_Degree( object*);
object *Is_Total_Degree_Greater( object*, object*);
void Print( object*);
void Free( object*);

/* *************** ADDITIVE_GROUP.H ***************** */

object *difference_ag( object*, object*);

/* *************** MULTIPLICATIVE_GROUP.H ***************** */

object *quotient_mg( object *, object *);

/* *************** INTEGER.H ***************** */

Operations int_funct;

object *make_int( int);
object *zero_int( void*);
object *one_int( void*);
object *sum_int( void*, void*);
object *negative_int( void*);
object *product_int( void*, void*);
object *is_equal_int( void*, void*);
object *lcm_int( void*, void*);
object *read_int( void);
void print_int( void*);
void free_int( int);
void init_int( void);
```

```
/* *************** IEAW.H ***************** */

#define max( a, b) ( (a) > (b) ? (a) : (b) )
#define min( a, b) ( (a) < (b) ? (a) : (b) )

typedef int *iea;

Operations iea_funct;

int length( iea);
object *make_iea( iea);
iea new_iea( int);
object *one_iea( void*);
object *product_iea( void*, void*);
object *inverse_iea( void*);
object *is_equal_iea( void*, void*);
object *lcm_iea( void*, void*);
object *is_multiple_iea( void*, void*);
object *total_degree_iea( void*);
object *is_total_degree_greater_iea( void*, void*);
object *read_iea( int);
void print_iea( void*);
void free_iea( void*);
void init_iea( void);

/* *************** MON.H ***************** */

typedef struct
  {
  object *coef;
  object *pp;
  int ref_count;
  } monomial;

typedef monomial *mon;

Operations mon_funct;

object *make_mon( mon);
mon allocate_mon( void);
mon new_mon( void);
object *one_mon( void*);
object *sum_mon( void*, void*);
object *negative_mon( void*);
object *product_mon( void*, void*);
object *inverse_mon( void*);
```

```
object *is_equal_mon( void*, void*);
object *lcm_mon( void*, void*);
object *is_multiple_mon( void*, void*);
object *total_degree_mon( void*);
object *is_total_degree_greater_mon( void*, void*);
object *read_mon( int);
void print_mon( void*);
void free_mon( mon);
void init_mon( void);

/* *************** LIST.H ***************** */

typedef struct list_item   /* list of objects ! */
  {
  struct list_item *next;
  object *value;
  int ref_count;
  }list_element;

typedef list_element *list;

Operations list_funct;

object *make_list( list);
list new_list_element( object*);
list allocate_list_element( void);
object* first( object*);
object* rest( object*);
object* cons( object*, object*);
object* empty_list( void);
object *is_equal_list( void*, void*);
void print_list( void*);
void free_list( void*);
void init_list( void);

/* *************** GENERIC.C ***************** */

#include <stdio.h>
#include "generic.h"

#pragma begin_inline

/*
  Assignment operation for objects with
  reference-counting.
*/
```

```
void Set( object **o1, object *o2)
  {
  if( *o1) Free( *o1);
  *o1 = o2;
  o2->ref_count++;
  }

object *allocate_object( void)
  {
  if( next_object >= 0)
    return free_object_array[ next_object--];
  else
    return (object*)malloc( sizeof( object));
  }

object *Zero( object *o)
  {
  return (o->type)[ ZERO]( o->value);
  }

object *One( object *o)
  {
  return (o->type)[ ONE]( o->value);
  }

object *Sum( object *o1, object *o2)
  {
  PF fun;
  if( fun = (o1->type)[ SUM])
    return fun( o1->value, o2->value);
  printf("\nNo function Sum available for this object\n");
  return (object*)0;
  }

object *Negative( object *o)
  {
  return (o->type)[ NEGATIVE]( o->value);
  }

object *Difference( object *o1, object *o2)
  {
  return (o1->type)[ DIFFERENCE]( o1, o2);
  }

object *Product( object *o1, object *o2)
```

```
  {
  return (o1->type)[ PRODUCT]( o1->value, o2->value);
  }

object *Inverse( object *o)
  {
  return (o->type)[ INVERSE]( o->value);
  }

object *Quotient( object *o1, object *o2)
  {
  return (o1->type)[ QUOTIENT]( o1, o2);
  }

object *Is_Equal( object *o1, object *o2)
  {
  return (o1->type)[ ISEQUAL]( o1->value, o2->value);
  }

object *LCM( object *o1, object *o2)
  {
  return (o1->type)[ L_C_M]( o1->value, o2->value);
  }

object *Is_Multiple( object *o1, object *o2)
  {
  return (o1->type)[ ISMULTIPLE]( o1->value, o2->value);
  }

object *Total_Degree( object *o)
  {
  return (o->type)[ TDEG]( o->value);
  }

object *Is_Total_Degree_Greater( object *o1, object *o2)
  {
  return (o1->type)[ TDEGGR]( o1->value, o2->value);
  }

void Print( object *o)
  {
  PF fun;

  if( fun = (o->type)[ PRINT])
      fun( o->value);
  else
```

```
    printf("\nNo function Print available for this object\n");
  }

void Free( object *o)
  {
  if( !( -- o->ref_count))
    {
    (o->type)[ FREE]( o->value);
    if( ++next_object < MAXFREE)
      free_object_array[ next_object] = o;
    else free(o);
    }
  }

#pragma end_inline

/* *************** ADDITIVE_GROUP.C ***************** */

#include "generic.h"
#include "additiveGroup.h"

object *difference_ag( object *o1, object *o2)
  {
  object *o;

  o = allocate_object();

  Set( &o, Negative( o2));
  Set( &o, Sum( o1, o));
  o->ref_count = 0;
  return o;
  }

/* *************** MULTIPLICATIVE_GROUP.C ***************** */

#include "generic.h"
#include "multiplicativeGroup.h"

object *quotient_mg( object *o1, object *o2)
  {
  object *o;

  o = allocate_object();

  Set( &o, Inverse( o2));
  Set( &o, Product( o1, o));
```

```
  o->ref_count = 0;
  return o;
  }

/* *************** INTEGER.C **************** */

#include "generic.h"
#include "additiveGroup.h"
#include "integer.h"

object *make_int( int a)
  {
  object *obj_ptr;

  obj_ptr = allocate_object();

  obj_ptr->type = int_funct;
  obj_ptr->value = (void*)a;

  return obj_ptr;
  }

object *zero_int( void *a)
  {
  return make_int( 0);
  }

object *one_int( void *a)
  {
  return make_int( 1);
  }

object *sum_int( void *a1, void *a2)
  {
  int sum;

  sum = (int)a1 + (int)a2;

  return make_int( sum);
  }

object *negative_int( void *a)
  {
  int neg;

  neg = - (int)a;
```

```
  return make_int( neg);
  }

object *product_int( void *a1, void *a2)
  {
  int prod;

  prod = (int)a1 * (int)a2;
  return make_int( prod);
  }

object *is_equal_int( void *a1, void *a2)
  {
  return ( (int)a1 == (int)a2 ? TRUE : FALSE);
  }

object *lcm_int( void *a1, void *a2)
  {
  return make_int( (int)a1 * (int)a2);  /* for test purposes */
  }

object *read_int()
  {
  int a;

  scanf("%d", &a);
  return make_int( a);
  }

void print_int( void *a)
  {
  printf("%d ",(int)a);
  }

void free_int( int a){}

void init_int( void)
  {
  int i;

  for( i=0; i<MAXFUN; i++) int_funct[ i] = (PF)0;

  int_funct[ ZERO] = zero_int;
  int_funct[ ONE] = one_int;
  int_funct[ SUM] = sum_int;
  int_funct[ NEGATIVE] = negative_int;
```

```
  int_funct[ DIFFERENCE] = difference_ag;
  int_funct[ PRODUCT] = product_int;
  int_funct[ ISEQUAL] = is_equal_int;
  int_funct[ L_C_M] = lcm_int;
  int_funct[ PRINT] = print_int;
  int_funct[ FREE] = free_int;
  }

/* *************** IEAW.C ***************** */

/*
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

  Domain: Integer Exponent Arrays

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

Data Structure:
---------------

An Integer Exponent Array is an array of the following
form:

    ( l, a1, ..., al)

    where l ("length") is a natural number and
    the ai's are integers.
*/

#include "generic.h"
#include "multiplicativeGroup.h"
#include "ieaW.h"

int ref_count_index;

int length( iea a)
  {
  return a[0];
  }

object *make_iea( iea a)
  {
  object *obj_ptr;

  obj_ptr = allocate_object();
```

```
  obj_ptr->type  = iea_funct;
  obj_ptr->value = (void*)a;
  obj_ptr->ref_count = 0;
  a[ ref_count_index]++;

  return obj_ptr;
  }

iea new_iea( int l)
  {
  iea a0;

  a0    = (iea)malloc( sizeof( int) * (l+2));
  a0[0] = l;
  a0[ ref_count_index] = 0;   /* reference counter */

  return a0;
  }

object *one_iea( void *v)
  {
  int i, l;
  iea a0, a;

  a  = (iea)v;
  l  = length( a);
  a0 = new_iea( l);

  for( i = 1; i <= l; i++) a0[ i] = 0;

  return make_iea( a0);
  }

object *product_iea( void *v1, void *v2)
  {
  int i, l;
  iea a0, a1, a2;

  a1 = (iea)v1;
  a2 = (iea)v2;

  l  = length(a1);
  a0 = new_iea( l);

  for( i = 1; i <= l; i++) a0[i] = a1[i] + a2[i];
```

```
  return make_iea( a0);
  }

object *inverse_iea( void *v)
  {
  int i, l;
  iea a0, a;

  a = (iea)v;
  l  = length(a);
  a0 = new_iea( l);

  for( i = 1; i <= l; i++) a0[i] = -a[i];

  return make_iea( a0);
  }

object *is_equal_iea( void *v1, void *v2)
  {
  int i, l;
  iea a1, a2;

  a1 = (iea)v1;
  a2 = (iea)v2;
  l = length( a1);

  for( i = 1; i <= l; i++)
    if( a1[ i] != a2[ i]) return FALSE;

  return TRUE;
  }

object *lcm_iea( void *v1, void *v2)
  {
  int i, l;
  iea a0, a1, a2;

  a1  = (iea)v1;
  a2  = (iea)v2;
  l = length( a1);
  a0 = new_iea( l);

  for( i = 1; i <= l; i++)
    a0[ i] = max( a1[ i], a2[ i]);

  return make_iea( a0);
```

```
  }

object *is_multiple_iea( void *v1, void *v2)
  {
  int i, l;
  iea a1, a2;

  a1  = (iea)v1;
  a2  = (iea)v2;
  l = length( a1);

  for( i = 1; i <= l; i++)
    if( a1[ i] < a2[ i]) return FALSE;

  return TRUE;
  }

object *total_degree_iea( void *v)
  {
  int i, l;
  int td;
  iea a;

  a  = (iea)v;
  l  = length( a);

  td = 0;

  for( i = 1; i <= l; i++)
    td += a[ i];

  return (object*)td;
  }

object *is_total_degree_greater_iea( void *v1, void *v2)
  {
  int td1, td2;
  iea a1, a2;

  a1  = (iea)v1;
  a2  = (iea)v2;
  td1 = (int)total_degree_iea( a1);
  td2 = (int)total_degree_iea( a2);

  return ( td1 > td2 ? TRUE : FALSE);
  }
```

```
object *read_iea( int l)
  {
  int i;
  iea a0;

  ref_count_index = l+1;
  a0 = new_iea( l);

  printf("integer exponent array: ");
  for( i = 1; i <= l; i++) scanf("%d", a0 + i);

  return make_iea( a0);
  }

void print_iea( void *v)
  {
  int i, l;
  iea a;

  a = (iea)v;
  l = length( a);

  if( a)
    {
    printf(" (");
    for( i = 1; i <= l; i++) printf("%d  ", a[ i]);
    printf(")");
    }
  else
    printf("Argument not printable\n");
  }

void free_iea( void *v)
  {
  iea a;

  a = (iea)v;
  if( !(--a[ ref_count_index])) free( a);
  }

void init_iea( void)
  {
  int i;

  for( i=0; i<MAXFUN; i++) iea_funct[ i] = (PF)0;
```

```
  iea_funct[ ONE] = one_iea;
  iea_funct[ PRODUCT] = product_iea;
  iea_funct[ INVERSE] = inverse_iea;
  iea_funct[ QUOTIENT] = quotient_mg;
  iea_funct[ ISEQUAL] = is_equal_iea;
  iea_funct[ L_C_M] = lcm_iea;
  iea_funct[ ISMULTIPLE] = is_multiple_iea;
  iea_funct[ TDEG] = total_degree_iea;
  iea_funct[ TDEGGR] = is_total_degree_greater_iea;
  iea_funct[ PRINT] = print_iea;
  iea_funct[ FREE] = free_iea;
  }

/* *************** MON.C ***************** */

/*
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

  Domain: Monomials

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

Data Structure:
---------------

A Monomial is a structure of the following
form:

    { c, pp}

    where c ("coefficient") is a (void) pointer to
                            a coefficient
    and pp ("power-product") is a (void) pointer to
                             a power-product.
*/

#include "generic.h"
#include "additiveGroup.h"
#include "multiplicativeGroup.h"
#include "integer.h"
#include "ieaW.h"
#include "mon.h"

/* Global Variables */
```

```
mon free_mon_array[ MAXFREE];
int next_mon;


object *make_mon( mon m)
  {
  object *obj_ptr;

  obj_ptr = allocate_object();

  obj_ptr->type  = mon_funct;
  obj_ptr->value = (void*)m;
  obj_ptr->ref_count = 0;
  m->ref_count++;

  return obj_ptr;
  }

mon allocate_mon()
  {
  if( next_mon >= 0)
    return free_mon_array[ next_mon--];
  else
    return (mon)malloc( sizeof( monomial));
  }

mon new_mon()
  {
  mon m0;

  m0 = allocate_mon();
  m0->coef = (object*)0;
  m0->pp   = (object*)0;
  m0->ref_count = 0;

  return m0;
  }

object *one_mon( void *v)
  {
  mon m0, m;

  m0 = new_mon();
  m  = (mon)v;

  Set( &m0->coef, One( m->coef));
```

```
  Set( &m0->pp, One( m->pp));

  return make_mon( m0);
  }

object *sum_mon( void *v1, void *v2)
  {
  mon m0, m1, m2;

  m0 = new_mon();
  m1 = (mon)v1;
  m2 = (mon)v2;

  if( Is_Equal( m1->pp, m2->pp))
    {
    Set( &m0->coef, Sum( m1->coef, m2->coef));
    Set( &m0->pp, m1->pp);
    return make_mon( m0);
    }
  else
    {
    printf("Error in sum_mon\n");
    return (object*)0;
    }
  }

object *negative_mon( void *v)
  {
  mon m0, m;

  m0 = new_mon();
  m = (mon)v;

  Set( &m0->coef, Negative( m->coef));
  Set( &m0->pp, m->pp);

  return make_mon( m0);
  }

object *product_mon( void *v1, void *v2)
  {
  mon m0, m1, m2;

  m0 = new_mon();
  m1 = (mon)v1;
  m2 = (mon)v2;
```

```
  Set( &m0->coef, Product( m1->coef, m2->coef));
  Set( &m0->pp, Product( m1->pp, m2->pp));

  return make_mon( m0);
  }

object *inverse_mon( void *v)
  {
  mon m0, m;

  m0 = new_mon();
  m = (mon)v;

  Set( &m0->coef, Inverse( m->coef));
  Set( &m0->pp, Inverse( m->pp));

  return make_mon( m0);
  }

object *is_equal_mon( void *v1, void *v2)
  {
  mon m1, m2;

  m1 = (mon)v1;
  m2 = (mon)v2;
  return ( Is_Equal( m1->coef, m2->coef) &&
           Is_Equal( m1->pp, m2->pp));
  }

object *lcm_mon( void *v1, void *v2)
  {
  mon m0, m1, m2;

  m0 = new_mon();
  m1 = (mon)v1;
  m2 = (mon)v2;

  Set( &m0->coef, LCM( m1->coef, m2->coef));
  Set( &m0->pp, LCM( m1->pp, m2->pp));

  return make_mon( m0);
  }

object *is_multiple_mon( void *v1, void *v2)
  {
```

```
  mon m1, m2;

  m1 = (mon)v1;
  m2 = (mon)v2;

  return Is_Multiple( m1->pp, m2->pp);
  }

object *total_degree_mon( void *v)
  {
  mon m;

  m = (mon)v;
  return Total_Degree( m->pp);
  }

object *is_total_degree_greater_mon( void *v1, void *v2)
  {
  mon m1, m2;

  m1 = (mon)v1;
  m2 = (mon)v2;
  return Is_Total_Degree_Greater( m1->pp, m2->pp);
  }

object *read_mon( int l)
  {
  object *pp;
  mon m0;

  m0 = new_mon();

  printf("Coefficient: ");
  Set( &m0->coef, read_int());
  Set( &m0->pp, read_iea( l));

  return make_mon( m0);
  }

void print_mon( void *v)
  {
  mon m;

  m = (mon)v;

  if( m)
```

```
    {
    printf("( ");
    Print( m->coef);
    Print( m->pp);
    printf( ")");
    }
  else
    printf("Argument not printable\n");
  }

void free_mon( mon m)
  {
  if( !( -- m->ref_count))
    {
    Free( m->coef);
    Free( m->pp);

    if( ++next_mon < MAXFREE)
      free_mon_array[ next_mon] = m;
    else free( m);
    }
  }

void init_mon( void)
  {
  int i;

  for( i=0; i<MAXFUN; i++) mon_funct[ i] = (PF)0;

  next_mon = -1;

  mon_funct[ ONE] = one_mon;
  mon_funct[ SUM] = sum_mon;
  mon_funct[ NEGATIVE] = negative_mon;
  mon_funct[ DIFFERENCE] = difference_ag;
  mon_funct[ PRODUCT] = product_mon;
  mon_funct[ INVERSE] = inverse_mon;
  mon_funct[ QUOTIENT] = quotient_mg;
  mon_funct[ ISEQUAL] = is_equal_mon;
  mon_funct[ L_C_M] = lcm_mon;
  mon_funct[ ISMULTIPLE] = is_multiple_mon;
  mon_funct[ TDEG] = total_degree_mon;
  mon_funct[ TDEGGR] = is_total_degree_greater_mon;
  mon_funct[ PRINT] = print_mon;
  mon_funct[ FREE] = free_mon;
  }
```

```
/* *************** LIST.C ***************** */

#include "generic.h"
#include "list.h"

/* Global Variables */

list free_list_array[ MAXFREE];
int next_list_element;

object *make_list( list l)
  {
  object *obj_ptr;

  obj_ptr = allocate_object();

  obj_ptr->type  = list_funct;
  obj_ptr->value = (void*)l;
  obj_ptr->ref_count = 0;
  if( l) l->ref_count++;

  return obj_ptr;
  }

list new_list_element( object* o)
  {
  list l0;

  l0 = allocate_list_element();

  l0->value = o;
  l0->ref_count = 0;
  o->ref_count ++;

  return l0;
  }

list allocate_list_element()
  {
  if( next_list_element >= 0)
    return free_list_array[ next_list_element--];
  else
    return (list)malloc( sizeof( list_element));
  }
```

```
object *first( object *o)
  {
  list l;

  l = (list)o->value;

  return l->value;
  }

object *rest( object *o)
  {
  list l0, l;

  l = (list)o->value;
  l0 = l->next;

  return make_list( l0);
  }

object *cons( object *o1, object *o2)
  {
  list l0, l;

  l = (list)o2->value;

  l0 = new_list_element( o1);
  l0->next  = l;
  if( l) l->ref_count++;

  return make_list( l0);
  }

object *empty_list()
  {
  return make_list( (list)0);
  }

object *is_equal_list( void *v1, void *v2)
  {
  list l1, l2;
  object *o1, *o2;

  l1 = (list)v1;
  l2 = (list)v2;

  while( o1 = l1->value)
```

```
    {
    if( (o2 = l2->value) && Is_Equal( o1, o2))
      {
      l1 = l1->next;
      l2 = l2->next;
      if( l1 == l2) return TRUE;
      }
    else
      return FALSE;
    }
  if( o2 = l2->value) return FALSE;
  else return TRUE;
  }

void print_list( void *v)
  {
  list l1, l;

  l = (list)v;

  printf("(");
  if( l1=l)
    do
      {
      Print( l1->value);
      printf(",");
      }
    while( l1 = l1->next);
  printf(")\n");
  }

void free_list( void *v)
  {
  list l1, l;

  l = (list)v;

  if( !l) return;
  do
    if( !( -- l->ref_count))
      {
      Free( l->value);
      l1 = l;
      l  = l->next;
      if( ++next_list_element < MAXFREE)
        free_list_array[ next_list_element] = l1;
```

```
      else
        free( l1);
      }
    else return;
  while( 1);
  }

void init_list( void)
  {
  int i;

  for( i=0; i<MAXFUN; i++) list_funct[ i] = (PF)0;

  next_list_element = -1;

  list_funct[ ISEQUAL] = is_equal_list;
  list_funct[ PRINT] = print_list;
  list_funct[ FREE] = free_list;
  }
```

References

[Buchberger 90] B. Buchberger: *A Simple Method for Elementary Polymorphism in C*, Technical Report, RISC-Series 90-58 , University of Linz, Austria, 1990.

[Windsteiger 90] W. Windsteiger: *An Implementation of Rational Functions in PCL*, Technical Report, RISC-Series 90-56 , University of Linz, Austria, 1990.