The Weyl Computer Algebra Substrate

Richard Zippel*
Cornell University
Ithaca, NY 14853
rz@cs.cornell.edu

January 5, 1993

This is a working draft of a paper on the lessons learned while developing Weyl. A number of sections are missing text. Beware!!

1 Introduction

In the last twenty years the algorithms and techniques for manipulating mathematical quantities symbolically have improved dramatically. These techniques have been made available through a number of algebraic manipulation systems. Among the most widely distributed systems are Reduce [5], Macsyma [10], Maple [7] and Mathematica [11]. These systems all contain comparable mathematical functionality and possess relatively similar designs. They are typical of most of the computer algebra systems currently being developed. Nonetheless, we feel that these systems are deficient in three respects:

- They are monolithic structures intended for use as self-contained computational systems in a largely interactive fashion.
- The internal algorithms of these systems are not designed in a "functorial" fashion, so their existing algorithms and mathematical data structures cannot be easily extended to work over new types of algebraic structures.
- The systems cannot represent mathematical "domains" like groups, rings and spaces of functions.

Weyl is a new computer algebra substrate designed to address each of these deficiencies. The rest of this section discusses each of these points in a bit more detail.

Computer algebra systems like Macsyma, Maple and Mathematica are complete systems within which one is supposed to do mathematical computations. In addition, to the basic symbolic algorithms each of these systems contain a complete, and often sophisticated, user interface and a programming language in which the user is expected describe their computations and extend the system.

*This research was supported in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research Contract N00014–88–K–0591, the National Science Foundation through grant IRI–9006137, the Office of Naval Research through contract N00014–89–J–1946 and in part by the U.S. Army Research Office through the Mathematical Science Institute of Cornell University.

This situation seems somewhat odd when compared with what has occurred in other areas of scientific computation such as linear algebra. Most practitioners use linear algebra libraries like LINPACK [4] and EISPACK [9], that can be incorporated in larger systems (like fluid dynamics codes or circuit analyzers). Stand alone linear algebra systems like Matlab [8] are used for research, but large scale applications such as those in computation fluid dynamics tend to be built using libraries rather than on top of Matlab. Furthermore, notice that Matlab was developed after the computational linear algebra field had matured and after the creation of a large number of applications that used linear algebra. In contrast Reduce and Macsyma were among the first symbolic mathematics packages created and they had a major impact on the creation of the field of computer algebra.

We argue that what is needed now is symbolic computing libraries that can be linked with existing numerical and graphical libraries. The results of symbolic computations should be as easy to manipulate as the matrices of linear algebra, which are so often used in scientific computing. This type of organization would make symbolic computation far more palatable to the general scientific computing community.

The second limitation of current symbolic mathematics systems is that they deal with a relatively limited and fixed set of algebraic types, which the user is not expected to extend significantly. For instance, most systems implement a type for representing polynomials in several variables with integer coefficients. In some problems, it is very convenient to deal with polynomials whose coefficients are rational functions (quotients of polynomials). To add this new type usually requires a new implementation of each of the arithmetic operations for polynomials with rational function coefficients, even though the algorithms are no different than those that are already implemented. Experimenting with algorithms for dealing with mathematical objects that are not already provided, such as polynomials with power series coefficients or Poisson series, often requires a great deal of effort, even if the system's source code is available.

The alternative is to implement the basic algorithms of the system in a "functorial" fashion. That is, rather than implementing polynomials with integer coefficients, the functorial approach would implement polynomials over an arbitrary ring[1] where the particular arithmetic operations used for computing with coefficients would come from the coefficient ring. This implementation could then be used for polynomials over the integers, over algebraic function fields, over rings matrices, etc.

The third limitation of current systems is their lack of "domains." Consider the ring of polynomials in x, y and z with integer coefficients, $\mathbb{Z}[x, y, z]$. While most algebraic systems can represent and manipulate the elements of this ring, they cannot represent the ring itself. Objects like $\mathbb{Z}[x, y, z]$ are called "domains" and have proven to be a very useful framework on which to hang mathematical properties.

Axiom [6] is a notable exception to this trend in that the algebra code has been developed in a "functorial" fashion that makes the whole system quite extensible. In addition, Axiom does provide a mechanism for representing mathematical domains.

Section 2 gives several examples of where the concept of a domain is useful. Section 3 explains the type system that is used in Weyl and how coercion is implemented. Section 4 explains how domains are created. Sections 5 and 6 illustrate how numbers and polynomials are implemented in Weyl.

Two appendices are included. Appendix A gives an example of how Weyl can be used used to perform a simple computation. An introduction to the features of the Common Lisp Object

[1] A ring is a mathematical term for a set of elements and two operations $+$ and $\times$ that satisfies certain properties. The prototypical example of a ring is the set of integers, $\{\ldots, -1, 0, 1, 2, \ldots\}$, which we denote by $\mathbb{Z}$

System we use is provided in Appendix B.

2 The Domain Concept

One of the novel concepts in Weyl is that of a *domain*. This section gives three examples that illustrate the need for domains.

Consider the problem of integrating rational functions with integer components. Even though coefficients of the integral are elements of $\mathbb{Z}$, the form of the integral depends upon the zeroes of the rational function. Sometimes, one gets lucky and all the zeroes lie in the ground field, *e.g.*,

$$\int \frac{dx}{x^3 - 2x^2 - x + 2} = \int \frac{dx}{(x-1)(x+1)(x-2)} = -\frac{1}{2}\log(x-1) + \frac{1}{6}\log(x+1) + \frac{1}{3}\log(x-2)$$

But sometimes new algebraic quantities must be added. For instance,

$$\int \frac{dx}{x^3 - 2} = \frac{\sqrt[3]{2}}{12}\log\frac{(x-\sqrt[3]{2})^2}{x^2 + \sqrt[3]{2}x + \sqrt[3]{4}} + \frac{\sqrt[3]{2}\sqrt{3}}{6}\arctan\frac{-\sqrt{3}x}{2\sqrt[3]{2} + x}.$$

Although the original problem involved rational functions with integer coefficients ($\mathbb{Z}$), notice that the final answer requires the use of $\sqrt{3}$ and $\sqrt[3]{2}$. These radicals are needed to express the zeroes of $x^3 - 2$.

Now consider a routine called `solve` that returns one of the zeroes of a polynomial in one variable. Given $x^2 - 5$ it should return $\sqrt{5}$. But how should the symbols $\sqrt{5}$ be interpreted, *i.e.*, how does one know that $\sqrt{5}^2 = 5$, and does $\sqrt{5} = 2.2361$ or $\sqrt{5} = -2.2361$? Furthermore, after `solve` has been asked to solve $x^2 - 5$, what is returned for $x^2 - 20$? Either $\sqrt{20}$ or $2\sqrt{5}$ could be correct, although the latter is probably better than the former.

These issues are addressed by using domains. Domains are collections of objects that possess some underlying mathematical organization. For instance, the set of rational integers $\mathbb{Z}$ is a domain, as is the set of rational numbers $\mathbb{Q}$. The polynomials $x^2 - 5$, $x^2 - 20$ and $x^2 - x - 1$ could all be elements of the domain $\mathbb{Q}[x]$, the set of polynomials in x with rational number coefficients. Other common domains are the real numbers $\mathbb{R}$ and the complex numbers, $\mathbb{C}$.

The first invocation of `solve` is given $x^2 - 5$, which is an element of $\mathbb{Q}[x]$. We will use the notation $(x^2 - 5)_{\mathbb{Q}[x]}$ when we want to emphasize the domain that contains an algebraic element. When `solve` returns $\sqrt{5}$ it must also create a domain in which $\sqrt{5}$ is an element. This domain is $\mathbb{Q}[\alpha]/(\alpha^2 - 5)$, the ring of polynomials in α with coefficients in $\mathbb{Q}$ reduced modulo the relation $\alpha^2 - 5 = 0$. Notice that α can correspond to either of the zeroes of $x^2 - 5$, so `solve` must also set up an embedding of $\mathbb{Q}[\alpha]/(\alpha^2 - 5)$ into $\mathbb{C}$—that is, a map that sends each element of $\mathbb{Q}[\alpha]/(\alpha^2 - 5)$ to a complex number. Since there is a natural embedding of $\mathbb{Q}$ into $\mathbb{C}$ it is only necessary to indicate the image of α in $\mathbb{C}$.

This final algebraic structure, along with the embedding into $\mathbb{C}$ is abbreviated as $\mathbb{Q}[\sqrt{5}]$. `Solve` returns $(\sqrt{5})_{\mathbb{Q}[\sqrt{5}]}$ thus conveying all the simplification rules that $\sqrt{5}$ must obey as well as the embedding in $\mathbb{C}$. Notice that this information is associated with the domains, rather than attached it directly to $\sqrt{5}$. We find this to be the mathematically more natural approach.

The second invocation of `solve` is interesting. Is `solve` invoked on $(x^2 - 20)_{\mathbb{Q}[x]}$ or $(x^2 - 20)_{\mathbb{Q}[\sqrt{5}][x]}$? It is not possible to determine which is the case by just examining the polynomial. Some reference to the enclosing domain is needed. If `solve` is called on $(x^2-20)_{\mathbb{Q}[x]}$ then $(\sqrt{20})_{\mathbb{Q}[\sqrt{20}]}$ is returned, just as in the previous discussion.

When `solve` is called on $(x^2-20)_{\mathbb{Q}[\sqrt{5}][x]}$, the coefficient field need to be extended and $(2\sqrt{5})_{\mathbb{Q}[\sqrt{5}]}$ is returned. This eliminates the potential future need of determining the relationship between $\sqrt{5}$ and $\sqrt{20}$.

One might expect that there is never a need for two isomorphic domains in a single computation. Why, for instance, would one want to have two copies of $\mathbb{Z}$ present during a computation? One answer to improve the safety of the programs and is discussed a bit later. Algebraic extensions though provide another motivation. The three zeroes of x^3-2 are

$$\sqrt[3]{2}, \quad \frac{1+\sqrt{-3}}{2}\sqrt[3]{2} = \zeta_3\sqrt[3]{2}, \quad \zeta_3^2\sqrt[3]{2}.$$

Each generates an algebraic extension of $\mathbb{Q}$ of degree 3. Algebraically each extension is of the form $\mathbb{Q}[\alpha]/(\alpha^3-2)$: *i.e.*,

$$\begin{aligned}\mathbb{Q}[\sqrt[3]{2}] &= \mathbb{Q}[\alpha]/(\alpha^3-2),\\ \mathbb{Q}[\zeta_3\sqrt[3]{2}] &= \mathbb{Q}[\beta]/(\beta^3-2),\\ \mathbb{Q}[\zeta_3^2\sqrt[3]{2}] &= \mathbb{Q}[\gamma]/(\gamma^3-2).\end{aligned}$$

That is, the polynomials that represent elements in the three different fields are manipulated in precisely the same fashion. The only difference between these three fields is their embedding in $\mathbb{C}$, *i.e.*,

$$\begin{aligned}\alpha &\longrightarrow 1.259921,\\ \beta &\longrightarrow -0.629960+1.091124i,\\ \gamma &\longrightarrow -0.629960-1.091124i.\end{aligned}$$

Finally, having multiple isomorphic domains allows us to check "functorial" code for missing or incorrect conversions while still using simple examples. For instance, consider the representation of polynomials over the integers, $\mathbb{Z}[x]$. Internally, such polynomials may be represented as lists of exponent-coefficient pairs:

$$2x^3+5 \approx ((3,2),(0,5)).$$

The coefficients are elements of $\mathbb{Z}$, as are the exponents. For most operations, the exponents never come in contact with the coefficients. For instance, when multiplying polynomials, the coefficients are multiplied with each other, and the exponents are compared and added to each other, but there are no operations that mix exponents and coefficients.

Many implementations use the same representation for both coefficients and exponents. Instead, we suggest that there should be a domain of integers for the exponents in addition to the domain associated with the coefficients. More generally, $k[x]$ would have two associated domains, k for the coefficients and a copy of $\mathbb{Z}$ for the exponents.

The value of this approach is apparent when examining polynomial differentiation, where exponents must be multiplied by coefficients. By having two isomorphic, but different, copies of $\mathbb{Z}$ we are forced to coerce an exponent into the domain of the coefficients even though it may not initially appear to be necessary for polynomials in $\mathbb{Z}[x]$, thus catching the need for coercion immediately.

3 Weyl's Type Structure

The objects manipulated by Weyl include *domains*, *domain elements* and *morphisms* between domains. Domain elements are the objects with which algebraic manipulations systems normally deal: integers, polynomials, algebraic numbers, etc. Examples of domains are the rational integers ($\mathbb{Z}$), polynomial rings ($\mathbb{Q}[x,y]$, $\mathbb{F}_p[t]$) and vector spaces ($\mathbb{R}^3$, $P^3(\mathbb{Z})$). The elements of these domains are domain elements.[2] Each domain element is the element of a single domain and this domain can be determined from the element. Morphisms are maps between domains. There exist predicates to determine if a morphism is a homomorphism, automorphism, etc.

The "type" of a domain element consists of two components, the domain of which the object is a member and the type of Lisp structure used to represent the object. For instance, the polynomial $x+2$ might be implemented as a list but it is an element of the domain $\mathbb{Z}[x]$. We say that the *structural type* of $x+1$ is `list` and that its *domain type* is $\mathbb{Z}[x]$. Elements of $\mathbb{Z}[x]$ could be represented as vectors or in other ways, and lists could be used to represent objects in other domains besides $\mathbb{Z}[x]$. Thus the structural type is truly orthogonal to the domain type of an object. Additional examples are given in Sections 5 and 6.

more on representation types and abstract types.

Both domains and domain elements are implemented as instances of CLOS classes. These classes are part of a conventional, multiple inheritance class hierarchy. The class hierarchy used for domains is completely separate and orthogonal to the class hierarchy for domain elements. The domain class hierarchy includes classes corresponding to groups, rings, fields and many other familiar types of algebraic domains. A section of the domain class hierarchy is shown in Figure 1.

This approach allows us to provide information about domains that is often difficult to indicate if we could only specify information about the type of the domain's elements. For instance, in addition to the class `Ring` we also provide classes like `Integral-Domain`. Whether a domain is an integral domain or just a ring does not change how the domain's elements are represented or the operations that can be performed on them. What it may affect is the algorithms that are used to implement the operations.

In addition, domains provide information about the permissible operations involving their elements. For instance, an `Group` must have a `*` operation that can be applied to pairs of its elements. The result will be an element of the abelian group. This parallels the mathematical definition of a group where a group is a pair $(G,\times)$ consisting of a set of elements (G) and a binary operation ($\times$) that maps pairs of elements of G into G. Contrast this to the typical definition of a type as a set of objects that obey a predicate.

Finally, the domains are objects that may be actively computed with. There are (constructive) functors that take an integral domain and produce its quotient field, take a ring and produce its spectrum, take a field and produce a vector space. The characteristic of a ring R is determined by applying the `characteristic` function to R. For implementing certain algorithms, the presence of domains seems to be a big improvement over the organization of systems like Macsyma.

3.1 Domain Hierarchy

The domain hierarchy of Weyl follows that of algebra fairly closely and is modeled on the category hierarchy used in Axiom [6]. The root of the hierarchy is the class `Set`. A section of the hierarchy is shown in Figure 1. As was done in Axiom, we have duplicated the definitions for semigroups, monoids and groups in the definitions of their abelian counterparts. This allows us to be a bit more explicit about the operations they use. That is, the binary operation of a `Group` is `*` while the operation of an `Abelian-Group` is `+`.

[2] To be strictly correct here we are speaking of the Lisp objects that represent the different domains and domain elements. Thus the Lisp object that represents $\mathbb{Z}$ or $\mathbb{Q}[x,y]$ are domains, and the Lisp object that represents x^2+2y+1 is a domain element.

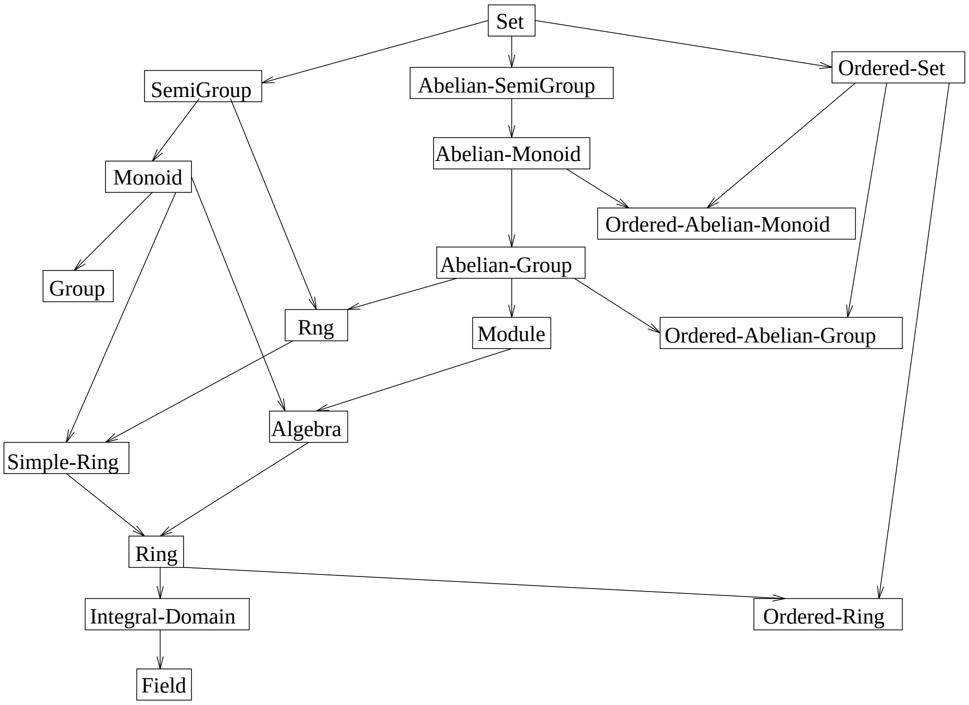

Figure 1: Section of Weyl's Domain Hierarchy

Actually, the *binary* operations are `binary+` and `binary*`. To match Common Lisp, where `+` and `*` are n-ary functions, `+` and `*` are macros that expand into repeated applications of `binary+` and `binary*` respectively. In particular,

`(+ a b c)` = `(binary+ a (binary+ b c))`.

Similar transformations are done for `*`, `-`, `/` and other binary functions.

Domains are defined using three forms. The CLOS `defclass` form is used to define the class used for the domain. Thus the following code is used to define a `Semigroup` and a `Simple-Ring`.

```
(defclass Semigroup (set)
  ())

(defclass Simple-Ring (rng monoid)
  ((characteristic :initarg :characteristic
                   :reader ring-characteristic)))
```

The `Semigroup` definition is quite simple. It merely includes the root class `set`. The `Simple-Ring` definition is a bit more involved. It inherits from both the `Rng` and `Monoid`, and in addition, includes a slot for the characteristic of the ring.

Once the classes have been defined additional information is provided about the operations associated with the domain and the axioms those operations must obey. The first form below indicates that the `binary*` operation is closed, and the second that `binary*` is associative.

```
(define-operations semigroup
  (binary* (element self) (element self)) -> (element self))
```

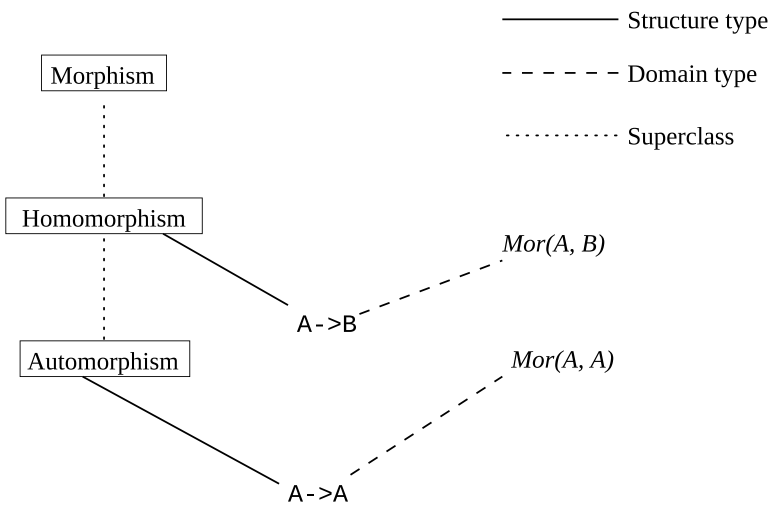

Figure 2: Piece of Morphism Domain Hierarchy

```
(define-axioms semigroup
  (associative binary*))
```

This information is used when new domains are created and when choosing certain algorithms. A simple example of the latter arises in the implementation of exponentiation to an integer power. Two algorithms that might be used are repeated multiplication and repeated squaring. However, the repeated squaring algorithm can only be used when the underlying multiplication algorithm is associative. This information can be determined from the domain.

Weyl currently does not include any theorem proving mechanisms like that contained in Nuprl [2]. Their inclusion would make the reasoning processes of the previous paragraph much more powerful. Till now we have not needed anything more significant, but this will change as more sophisticated mathematical structures and algorithms are added to the system.

None of the domains in Figure 1 can actually be instantiated. They are "abstract domains" that are intended to be included in instantiable domains. For instance, the domain used for polynomial rings inherits from the `Ring` domain and the domain used for the rational integers includes the `Unique-Factorization-Domain` class.

3.2 Morphisms

Morphisms are maps between domains (or more mathematically, the arrows of the category of Weyl domains). They are essential to the coercion mechanisms used in Weyl.

Though we could have used regular Lisp functions for this purpose, we have decided to encapsulate the mapping functions in an object so that additional information can be attached to the map itself. This also allows us to take advantage of the polymorphism and overloading mechanisms provided by CLOS.

Morphisms have two user visible components, the domain of the morphism and the range of the morphism. The printed representation of a morphism from `A` to `B` is `A->B`. Like other objects in Weyl, morphisms have a structural type and domain type. The domain type of a homomorphism `A->B` is $\mathrm{Mor}(A, B)$, the set of morphisms of A to B. The structural type depends on how it is implemented. At the time of this writing there are only three choices, `morphism`, `homomorphism` and `automorphism`. This is illustrated in Figure 2.

A few abstract classes of morphisms have been provided to indicate that more is known about the map than that is a simple morphism. In particular, a morphism can be a homomorphism, isomorphism or automorphism. Finer structure, such as if the morphism is injective, bijective, continuous, exact, etc awaits further work.

Morphisms are often created automatically when new domains are created. For instance, when the polynomial ring $k[x, y, z]$ is created from k, a homomorphism is crated that embeds elements of k in $k[x, y, z]$, *i.e.* `k->k[x, y, z]`. Similarly, when $\mathbb{F}_{163}$ (the finite field of 163 elements) is created the homomorphism `Z->GF(163)` is also created.

The function `apply-morphism` applies a morphism to an element of its domain and yields an element of its range. The main operation on morphisms is `compose`, which constructs a new morphism that maps elements of the domain of its first argument into the range of its second. Thus,

```
> (compose Z->GF(163) GF(163)->GF(163)[x, y, z])
Z->GF(163)[x, y, z]
```

In the future, we hope to have `image`, `kernel` and `co-kernel` routines. These routines would create new algebraic domains. Needless to say, computing the image and kernel of an arbitrarily defined morphism can be difficult or impossible. However, when the morphism is defined by reducing by an ideal, by localization or some combination of these two operations image, kernel and co-kernel computations can often be made effective.

3.3 Coercions

In most algebraic manipulation systems, when two objects from different domains are combined, they are automatically coerced into a common domain. Which domain is used can be ambiguous. For instance, the sum of $(1/2)_{\mathbb{Q}}$ and $(x+1)_{\mathbb{Z}[x]}$ could be either

$$\left(x + \frac{3}{2}\right)_{\mathbb{Q}[x]} \quad \text{or} \quad \left(\frac{2x+3}{2}\right)_{\mathbb{Z}(x)}.$$

problem is typical of computer algebra, but is exacerbated by the typing system of Weyl.

The wrong choice could cause significant problems because of the added precision introduced by the use of Weyl domains.

Because of this type of ambiguity we feel that algebraic algorithms should be coded in a way that makes explicit the coercions take place. That is, when building system code, only explicit coercions should be performed and operations like adding $(1/2)_{\mathbb{Q}}$ and $(x+1)_{\mathbb{Z}[x]}$ should be illegal. The user of a computer algebra system, however, should have more flexibility.

Two mechanisms are available for working with elements of different domains. The simplest is to explicitly coerce the elements into the proper domain and then combine them, *i.e.*,

$$\texttt{(+ (coerce } (1/2)_{\mathbb{Q}}\ \mathbb{Q}[x]\texttt{) (coerce } (x+1)_{\mathbb{Z}[x]}\ \mathbb{Q}[x]\texttt{))}$$

The `coerce` functions searches the set of homomorphisms that have been defined to determine a set of homomorphisms whose composition goes from the domain of the first argument to the target domain. If only one such set is found, then the composition is applied. If more than one such set is found, then an error is signaled because of the ambiguity.

$$
\begin{tikzcd}
\mathbb{Q} \arrow[r] & \mathbb{Z}(x) \\
\mathbb{Z} \arrow[r] \arrow[u] & \mathbb{Z}[x] \arrow[u]
\end{tikzcd}
$$

Figure 3: A Sample Computational Context

The second approach is used by applications developers. When a binary operation is applied to two elements of different domains, and a certain switch is turned on, Weyl will attempt to find a common domain into which both elements can be coerced. If such a domain can be uniquely determined, the operation will be done there.

The set of domains and homomorphisms that are in force at any point in time is called a *computational context*. Recalling the previous problem of adding $(1/2)_{\mathbb{Q}}$ and $(x+1)_{\mathbb{Z}[x]}$, if the user has established the the computational context shown in Figure 3 then the value of the sum will be well defined. We expect that application specific systems that use the Weyl substrate will use different computational contexts for different problems.

4 Domain Creation

Domains are created in Weyl using functions called *functors*. Examples of these functors are `get-polynomial-ring`, which was used in Section A, and `get-quotient-field`, which generates a domain that is the quotient field of its argument. Functors are coded as regular CLOS methods. For instance, the following forms are used to define `get-quotient-field`.

```
(defmethod get-quotient-field ((ring Field))
  ring)

(defmethod get-quotient-field ((ring Integral-Domain))
  (let ((qf (make-instance 'quotient-field :ring ring)))
    (with-slots (zero one) qf
      (setq zero (make-quotient-element qf (zero ring) (one ring)))
      (setq one (make-quotient-element qf (one ring) (one ring))))
    qf))
```

The first form takes care of the trivial case of the quotient field of a field. There is no `get-quotient-field` constructor for more general rings, since the resulting structure is not necessarily a field. The second form creates quotient fields from integral domains. In addition, we have decided to cache the values of zero and one in the quotient field for efficiency reasons.

If there is a `gcd` routine for the ring, then the following, more efficient routine is used. The elements of `efficient-quotient-field` use the `gcd` routine to reduce the numerator and denominator of quotient elements to lowest terms.

```
(defmethod get-quotient-field ((ring Gcd-Domain))
  (let ((qf (make-instance 'efficient-quotient-field :ring ring)))
    (with-slots (zero one) qf
      (setq zero (make-quotient-element qf (zero ring) (one ring)))
```

```
  (setq one (make-quotient-element qf (one ring) (one ring))))
qf))
```

Algebraic extensions provide additional complications. Consider the case of extending $\mathbb{Q}[\sqrt{2}]$ by a zero of $f(x) = x^4 - 10x^2 + 1$. Although $f(x)$ is irreducible over $\mathbb{Q}$, over $\mathbb{Q}[\sqrt{2}]$ it factors into

$$f(x) = (x^2 - 2\sqrt{2}x - 1)(x^2 + 2\sqrt{2}x - 1).$$

(The zeroes of $f(x)$ are $\pm\sqrt{2} \pm \sqrt{3}$.) There are several potential problems here. The algebraic extension constructor could return one of several different results:

$$\begin{aligned}
A &= \mathbb{Q}[\sqrt{2}][\alpha]/(\alpha^4 - 10\alpha^2 + 1), \\
B &= \mathbb{Q}[\sqrt{2}][\beta]/(\beta^2 - 2\sqrt{2}\beta - 1), \\
C &= \mathbb{Q}[\gamma]/(\gamma^4 - 10\gamma^2 + 1), \\
B' &= \{\mathbb{Q}[\sqrt{2}][\beta]/(\beta^2 - 2\sqrt{2}\beta - 1), \mathbb{Q}[\sqrt{2}][\beta]/(\beta^2 + 2\sqrt{2}\beta - 1)\}.
\end{aligned}$$

A has the problem that it is a ring with zero divisors not a field. B is a field, but includes one of two possible zeroes of $x^4 - 10x^2 + 1$. Furthermore, B is not generated by a primitive element over $\mathbb{Q}$. This can make some algorithms somewhat complex. C is generated by primitive element over $\mathbb{Q}$, and it has the proper degree. However, it also as selected a particular pair of zeroes of $f(x)$. Which zeroes are chosen depends upon the embedding of $\mathbb{Q}[\sqrt{2}]$ in C. B' returns not one field, but two; one for each pair of zeroes of $f(x)$. This is essentially the approach taken by Duval [3].

The presence of domains in Weyl provides an interesting opportunity to develop some relatively sophisticated computations on domains themselves such as computing the the Hilbert class field of an algebraic number field, the cohomology groups of ring acted on by a group and the topology of the spectrum of a commutative ring. Thus far we have not

Thus Weyl should be able to provide a sound computational environment for investigating many questions in commutative algebra and algebraic geometry as well as in applied mathematics.

5 Number Types

The different numerical types provided by Weyl provide a number of simple and illuminating examples of the issues that arise in symbolic computations as we have organized it. The domains we will consider are the rational integers, $\mathbb{Z}$, the rational numbers, $\mathbb{Q}$, the real numbers, $\mathbb{R}$, and the complex numbers $\mathbb{C}$. The domain $\mathbb{Z}$ is a `Ring`, (*i.e.*, commutative ring with unit), while $\mathbb{Q}$, $\mathbb{R}$ and $\mathbb{C}$ are `Fields`.

Each element of the rational integers, $\mathbb{Z}$, is represented by instances of the CLOS class `Rational-Integer`:

```
(defclass rational-integer (numeric)
     ((value :initarg :value
             :reader integer-value)))
```

One slot is provided to hold the Lisp number that is used to represent the integer.[3] The class `Numeric` is the base class for all domain elements that are numbers.

[3]In a language that does not have built-in variable precision integers, this structure would be more complex.

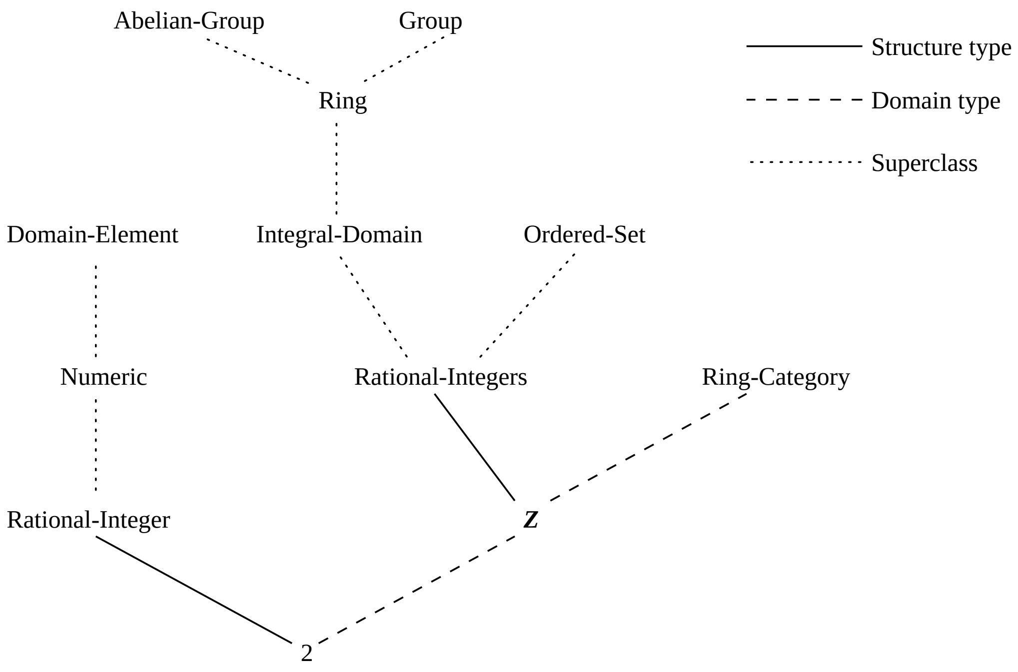

Figure 4: Structure/Domain hierarchy for $2_{\mathbb{Z}}$

Figure 4 illustrates the structural and domain type hierarchies that lie over $2_{\mathbb{Z}}$. The structure types are indicated by solid lines and the domain types by dashed lines. Thus the integer $2_{\mathbb{Z}}$ has the structure of a `Rational-Integer`, while $\mathbb{Z}$ has the structure of `Rational-Integers`. The domain of $2_{\mathbb{Z}}$ is $\mathbb{Z}$, while the domain of $\mathbb{Z}$ is the category of rings.[4]

Lying above `Rational-Integer` and `Rational-Integers` is the CLOS class hierarchy that is used to implement them. The most interesting part of this hierarchy is that lying over `Rational-Integers`.

Methods are provided for the basic operations with rational integers. For instance, a simple version of the binary addition operator might be defined as defined as

```
(defmethod binary+ ((x Rational-Integer) (y Rational-Integer))
  (make-element (domain-of x)
                (lisp:+ (integer-value x) (integer-value y))))
```

This method returns an element of the domain of `x`. The function `lisp:+` refers to the built in Lisp routine for adding variable precision integers.

However, since `x` and `y` are not necessarily elements of the same domain, this approach is not strictly correct. Instead, we have chosen to use the following approach

```
(defmethod binary+ :around ((x Rational-Integer) (y Rational-Integer))
  (let ((domain (domain-of x)))
    (cond ((eql domain (domain-of y))
           (make-element domain
                         (lisp:+ (integer-value x) (integer-value y))))
```

[4]The category of rings is not currently implemented. Numerous issues arise in this sort of situation that we have not yet resolved.

```
            (t (call-next-method x y)))))
```

This :around method ensures that x and y are elements of the same domain. If they are not, then this routine "passes the buck" to whoever else might have an idea on how to add two rational-integers.

The buck stops with the following default routine:

```
 1 (defmethod binary+ ((x domain-element) (y domain-element))
 2   (when (null *coerce-where-possible*)
 3     (error "No way to add ~S and ~S" x y))
 4   (let ((domain-x (domain-of x))
 5         (domain-y (domain-of y))
 6         common-domains)
 7     (when (eql domain-x domain-y)
 8       (error "No way to add ~S and ~S" x y))
 9     (setq common-domains (find-common-domains domain-x domain-y))
10     (cond ((null common-domains)
11            (error "No way to add ~S and ~S" x y))
12           ((null (rest common-domains))
13            (binary+ (coerce x (first common-domains))
14                     (coerce y (first common-domains))))
15           (t (error "Ambiguous coercion for addition (~S, ~S)"
16                     x y)))))
```

Notice that the specializers are domain-element, so this method is the default method for addition of any two elements of Weyl domains.

The switch *coerce-where-possible* enables automatic coercion between domains, as described in Section 3.3. In most system code this switch is turned off and an error is signaled, indicating that one cannot add two integers from different domains (line's 2 and 3).

The routine find-common-domains tries to find the smallest domains into which both x and y can be coerced. If there is just one such domain, then both are coerced into the common domain and the addition is tried again (lines 9–14). If there is more than one common domain, than an error is signaled (lines 15–16).

Finally, if this routine is invoked when two elements that are already in the same domain, so finding a common domain in which to perform the operation will not work, then an error is signaled (lines 7 and 8). These lines eliminate the possibility of infinite recursion.

Similar routines are present for most binary operations.

5.1 Structural Type and Domain Type Orthogonality

Now consider the elements of $\mathbb{Q}$. Rational numbers like 1/3 and 3/7 are represented using the following CLOS class:

```
(defclass rational-number (numeric)
  ((numerator :accessor qo-numerator
              :initarg :numerator)
   (denominator :accessor qo-denominator
                :initarg :denominator)))
```

However, we have chosen to represent the integers in $\mathbb{Q}$ using the Rational-Integer class. Thus elements of $\mathbb{Q}$ can have one of two different representations: as Rational-Integers or as Rational-Numbers. Thus we could have two elements, $2_{\mathbb{Z}}$ and $2_{\mathbb{Q}}$, that use the same representation but that are elements of different domains.

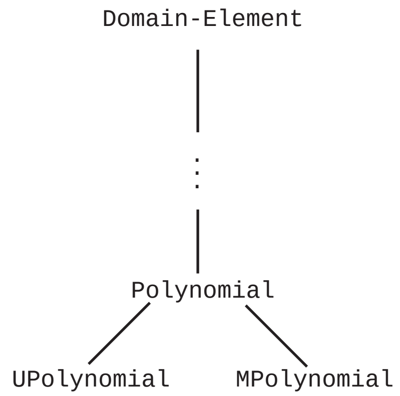

Figure 5: Structural type hierarchy for polynomials

However, we now have an interesting problem. If we try to divide $2_{\mathbb{Q}}$ by $3_{\mathbb{Q}}$ we should get $2/3_{\mathbb{Q}}$. But, under most circumstances, the quotient of $2_{\mathbb{Z}}$ by $3_{\mathbb{Z}}$ should signal an error. In this case, the class based dispatching used by CLOS fails us, since it only pays attention to the structural types of $2_{\mathbb{Z}}$ and $3_{\mathbb{Z}}$. Both of these quantities have the same structural types as $2_{\mathbb{Q}}$ and $3_{\mathbb{Q}}$, for which division is proper.

This problem is dealt with in Weyl by the somewhat *ad hoc* approach of including additional checks in the quotient routine.

```
(defmethod binary/ :around ((x rational-integer) (y rational-integer))
  (let ((domain (domain-of x)))
    (cond ((not (eql domain (domain-of y)))
           (call-next-method x y))
          ((1? y) x)
         ((typep domain 'field)
          (make-element domain
                        (lisp:/ (integer-value x) (integer-value y))))
         (t (call-next-method x y)))))
```

It would be far better if the domain type could be used in the generic function dispatch.

6 Polynomial Package

The two dimensional type structure used by Weyl does not naturally map into the one dimensional type system used by CLOS. We have chosen to make the "CLOS type" of an object be the structural type. The domain type of of a Weyl object is kept in the `domain` slot of the object and is managed by code in Weyl.

The basic polynomial code provides a good vehicle for demonstrating how the Weyl structure is mapped into the CLOS structure. Polynomials can be represented using either a recursive, multivariate, sparse representation called `mpolynomial` or using a univariate dense representation called `upolynomial`. `Mpolynomial` and `upolynomial` are both CLOS classes and Weyl structural types. In addition there is a CLOS class called `polynomial` from which both classes inherit. At the root of the CLOS class hierarchy is the class `domain-element`, as shown in Figure 5.

Elements of a univariate polynomial ring can exist in either representation, and two upolynomials or mpolynomials can be elements of different rings. Operations with domain elements like polynomials are performed by generic functions. The generic function to add two domain elements is binary+. This generic function dispatches on the CLOS class of its two arguments, to a method that examines the domain's of its two arguments for compatibility. For instance, the code to add two polynomials looks like this:

```
(defmethod binary+ ((x mpolynomial) (y mpolynomial))
  (let ((domain (domain-of x)))
    (cond ((eql domain (domain-of y))
           (make-mpolynomial domain
                             (mpoly-plus (poly-form x) (poly-form y))))
          (t (error "Trying to add two polynomials from different domains.")))))
```

In addition, methods are needed for the three other combinations of mpolynomials with upolynomials.

The real work of adding polynomials is performed by the function mpoly-plus. Notice that this is a true Lisp function and not a CLOS generic function. This done for efficiency reasons. When one is building a new polynomial algorithm the bulk of the code is written using functions like mpoly-plus and then wrapped in a generic function like binary+. One retains full generality by using generic functions like binary+ for operations with the polynomial coefficients.

Not all algorithms are so simple. For instance, for polynomial GCD there are several algorithms. The subresultant GCD algorithm is valid for polynomials over any ring, while the sparse modular algorithm requires that the coefficient domain be sufficiently large and that there be a function for producing a random element of the coefficient domain.

In this situation the gcd method needs to perform some additional analysis of the coefficient domain of the polynomial ring before choosing an algorithm. The following code illustrates this. Notice that we can determine the coefficient domain of the polynomial ring, from the domain of x.

```
(defmethod gcd ((x mpolynomial) (y mpolynomial))
  (let ((domain (domain-of x)))
    (cond ((eql domain (domain-of y))
           (make-mpolynomial domain
             (let ((cdomain (coefficient-domain domain)))
               (cond ((and (has-operation? cdomain 'random)
                           (zerop (characteristic cdomain)))
                      (sparse-modular-gcd (poly-form x) (poly-form y)))
                     (t (subresultant-gcd (poly-form x) (poly-form y)))))))
          (t (call-next-method x y)))))
```

The scheme given above is not completely satisfactory as it stands. First, the decision about which algorithm should be used is made every time one compute the GCD of two polynomials. This is a little silly, since the decision never changes (in this case). Thus we should cache this decision in the polynomial domain itself. Second, this decision is only available if we go through the method gcd, when just a moment ago we indicated that this should be avoided for performance reasons. Thus the decision making code given above, and the caching code should be moved into a function like mpoly-gcd. All of this, as well as the process of defining the methods can be eliminated by the use of macros.

The most severe problem with this scheme is that when new algorithms are introduced several pieces of code need to be rewritten. This is the problem that the Capsules system [12] solves. We have not yet integrated the Capsule ideas into Weyl.

6.1 Expanded Polynomials

To efficiently implement the Gröbner basis algorithm, which is central to many problems in computational algebraic geometry, a special representation is used for polynomials. In an expanded representation, a polynomial is represented as a *sorted* list of terms where each terms is represented as a vector. For instance, the polynomial

$$x^3 + 7xy^3z + 12yz^5$$

would be represented by the list

```
(<1, 3, 0, 0> <7, 1, 3, 1> <12, 0, 5, 7>)
```

By far the most common operation in the Gröbner basis algorithm is that of comparing the degrees that occur in terms. Many different comparison predicates are used in computational algebraic geometry, where each gives a different mathematical result. For this exposition, we focus on one of the simplest, the "lexicographical" ordering.

In Common Lisp this would be coded as:

```
(defun lexical> (vect1 vect2)
  (loop with dimension = (array-dimension 0 domain)
        for i upfrom 1 below dimension
        do (when (> (aref vect1 i) (aref vect2 i))
             (return t))
           (when (< (aref vect1 i) (aref vect2 i))
             (return nil))))
```

With some additional declarations this produces quite good code.

However, a substantial speedup can be achieved if know the dimension of the vectors (*i.e.*, the number of variables in the Gröbner basis calculation), which is typically less than 10, and completely unroll the loops. In this case the code would be

```
(defun lexical>-3 (vect1 vect2)
  (cond ((> (aref vect1 1) (aref vec2 1))
         t)
        ((< (aref vect1 1) (aref vec2 1))
         nil)
        ((> (aref vect1 2) (aref vec2 2))
         t)
        ((< (aref vect1 2) (aref vec2 2))
         nil)
        ((> (aref vect1 3) (aref vec2 3))
         t)
        ((< (aref vect1 3) (aref vec2 3))
         nil))
        (t nil))
```

With a few additional declarations, this code can be twice as fast as the previous version. This is especially true on the new pipelined RISC processors.

To facilitate this approach, Weyl automatically generates functions like `lexical>-3` from a specification of the comparison function provided by the user. These functions are then compiled and loaded into the running Weyl environment. This has resulted in a dramatic speedup in the time to compute Gröbner bases.

7 Conclusions

Weyl's current implementation use runtime typing for all objects. Some decisions that might be performed at compile time are delayed until run time. Some of this is due to using the Common Lisp Object System instead of building our own polymorphism and overloading scheme as is done in Axiom.

Though the type system used by Weyl is supported at run time, there is no reason why it could not be supported by the compiler. This would provide all the advantages of strong typing, and might improve performance somewhat. We have opted for the run time version used here for convenience in development.

Much of this work is the direct result of discussions with members of the IBM symbolic computing group, particularly Barry Trager and Dick Jenks, whose assistance I gratefully acknowledge.

References

[1] D. G. Bobrow, L. G. DeMichiel, R. P. Gabriel, K. Kahn, S. E. Keene, G. Kiczales, L. Masinter, D. A. Moon, M. Stefik, and D. L. Weinreb, *Common Lisp object system specification*, Tech. Rep. 88–002, X3J13, ANSI Common Lisp Standardization Committee, July 1988.

[2] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith, *Implementing Mathematics with the Nuprl Proof Development System*, Prentice-Hall, Englewood Cliffs, NJ, 1986.

[3] C. Dicrescenzo and D. Duval, *Algebraic extensions and algebraic closure in scratchpad ii*, in ISSAC '88, P. Gianni, ed., vol. 358 of Lecture Notes in Computer Science, Berlin-Heidelberg-New York, 1988, Springer-Verlag, p. ??

[4] J. Dongarra, J. R. Bunch, C. B. Moler, and G. W. Stewart, *LINPACK User's Guide*, SIAM Publications, Philadelphia, PA, 1978.

[5] A. C. Hearn, *Reduce 3 user's manual*, tech. rep., The RAND Corp., Santa Monica, CA, 1986.

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

[7] Maple Group, *Maple*, Waterloo, Canada, 1987.

[8] C. B. Moler, *Matlab user's guide*, Tech. Report CS81–1, Dept. of Computer Science, University of New Mexico, Albuquerque, NM, 1980.

[9] B. T. Smith, J. M. Boyle, Y. Ikebe, V. C. Klema, and C. B. Moler, *Matrix Eigensystem Routines: EISPACK Guide*, Springer-Verlag, New York, NY, second ed., 1976.

[10] Symbolics, Inc., *MACSYMA Reference Manual*, Burlington, MA, 14th ed., 1989.

[11] S. Wolfram, *Mathematica: A System for Doing Mathematics by Computer*, Addison-Wesley, Redwood City, CA, 1988.

[12] R. E. Zippel, *Capsules*, SIGPLAN Bulletin, 18 (1983), pp. 166–169.

A A Session with Weyl

This section gives a short example of how Weyl can be used to perform a simple algebraic calculation, calculating the coefficients of the "F and G series." These coefficients are defined by the following recurrences:

$$f_0 = 0$$

$$f_n = -\mu g_{n-1} - \sigma(\mu + 2\epsilon)\frac{\partial f_{n-1}}{\partial \epsilon} + (\epsilon - 2\sigma^2)\frac{\partial f_{n-1}}{\partial \sigma} - 3\mu\sigma\frac{\partial f_{n-1}}{\partial \mu}$$

$$g_0 = 1$$

$$g_n = f_{n-1} - \sigma(\mu + 2\epsilon)\frac{\partial g_{n-1}}{\partial \epsilon} + (\epsilon - 2\sigma^2)\frac{\partial g_{n-1}}{\partial \sigma} - 3\mu\sigma\frac{\partial g_{n-1}}{\partial \mu}$$

Each of the f_i and g_i are polynomials in μ, ϵ and σ with integer coefficients and can be viewed as elements of the domain $\mathbb{Z}[\mu, \epsilon, \sigma]$.

Our first task is to generate the *domain* $\mathbb{Z}[\mu, \epsilon, \sigma]$. In this problem, not much is required of the domain, it is only used to generate the desired polynomials. This done by first getting a copy of the rational integers, $\mathbb{Z}$, using the function `get-rational-integers`. The polynomial ring $\mathbb{Z}[\mu, \epsilon, \sigma]$ is the created using `get-polynomial-ring`:

```
> (setf R (get-polynomial-ring (get-rational-integers) '(m e s)))
Z[m, e, s]
```

where we have used the symbols `m`, `e` and `s` to represent the greek letters μ, ϵ and σ. The line proceeded by the "`>`" is what is typed into Lisp by the user. The following line is Weyl's response.

At this point we can create polynomials that are elements of the domain `R`. To do this it is convenient to bind the Lisp variable `mu` to μ, `eps` to ϵ and `sigma` to σ. This is done by coercing the symbols `m`, `e` and `s` into polynomials in the domain `R`.

```
> (setf mu (coerce 'm R))
m
> (setf eps (coerce 'e R))
e
> (setf sigma (coerce 's R))
s
```

The coefficients of the first terms of the both recurrences are the same, so rather than computing them afresh each time, they are stored in global variables

```
> (setq x1 (- mu))
-1 m
> (setq x2 (* (- sigma) (+ mu (* 2 eps))))
-1 s m + -2 s e
> (setq x3 (+ eps (* -2 (expt sigma 2))))
e + -2 s^2
> (setq x4 (* -3 mu sigma))
-3 s m
```

Notice that the usual Lisp functions for performing arithmetic operations are used and normal Lisp variables can have polynomials as values. This is one of the convenient features of Weyl. We have simply extended the basic Lisp functions to deal with the algebraic objects of Weyl.[5] Also note that the values are printed in a linear infix notation so that can be easily interpreted, although the output form could be improved upon.

We can now write the recursion formulae.

[5]Actually, the symbols "+", "-" and "*" are in a different package so that a clean Common Lisp world can be preserved.

```
(defun f (n)
  (if (= n 0) (coerce 0 R)
      (+ (* x1 (g (1- n)))
         (* x2 (partial-deriv (f (1- n)) eps))
         (* x3 (partial-deriv (f (1- n)) sigma))
         (* x4 (partial-deriv (f (1- n)) mu)))))
```

Notice that the function `partial-deriv` is used to compute the partial derivative of a polynomial. The recursion formula for g_n is similar.

Finally, we can compute the coefficients by invoking the `f` function.

```
> (f 3)
mu^2 + (3 eps + -15 sigma^2) mu
> (f 4)
-15 sigma mu^2 + (-45 sigma eps + 105 sigma^3) mu
```

This simple example illustrates that writing programs that use the Weyl substrate is not much different from writing ordinary Lisp programs. We have extended some of the data types and we have introduced the concept of domains, but the same control structures, abstractions and programming tools will continue to work.

B Common Lisp Object System

Weyl is based on the Common Lisp Object System (CLOS) [1] programming model. Since this model of object oriented programming differs significantly from the more common, message based paradigm used in Smalltalk and C++, we give a short introduction here.

CLOS is an object oriented extension to Common Lisp that has been adopted as part of ANSI standard Common Lisp during the standardization process. A portable implementation of CLOS is available[6] and most commercial Common Lisp vendors provide efficient implementations. This introduction hits only the highlights needed for Weyl. Many other features are present are described in the standard.

CLOS extends Lisp with two pairs of new concepts: *instances* and *classes*, and *methods* and *generic functions*. Instances and classes are an extension of the "structure" concept that is already present in Common Lisp (and most other modern languages). Methods and generic functions are completely new to Common Lisp and provide most of the new functionality.

Instances are structures that contain zero or more *slots* that can each hold arbitrary Lisp values. Each instance is an instance of *class* of similar objects. For example, we may want to construct the class of complex numbers $\mathbb{C} = \{a + bi\}$. Each instance of these objects contains two values, which we indicate by `a` and `b`. The class of such instances might be called `Zsqrt7` and would be defined as:

```
(defclass complex-number ()
  ((a :initarg :real :accessor real-part)
   (b :initarg :imag :accessor imag-part)))
```

A particular instance of the class `complex-number` can be created by calling the function `make-instance`. The keywords provided with the `:initarg` options in the class definition are used to indicate the initializing values for each slot. A particular `point` can be created as:

[6] It can be transferred from `parcftp.xerox.com` in the `pub/pcl` directory.

```
(setq z (make-instance complex-number :real 1.0 :imag 2.0))
```

The `:accessor` options used in the class definition define forms that can be used to read the values in each slot. The forms can also be used as arguments to `setf`. For example, we can modify `z` to be its complex conjugate by:

```
(setf (imag-part z) (- (imag-part z)))
```

One can define a class so that each of its instances contain the slots associated with another class. For example, if we wished to associate another property with a complex number, *e.g.*, the units, we would define a class like

```
(defclass dimensioned-complex-number (point)
  ((units :initarg :units :accessor units)))
```

An instance of a `dimensioned-complex-number` would contain three slots, `a` and `b` slots that are *inherited* from `complex-number` and `units` slot that is unique to instances of `dimensioned-complex-number`. We say that the class `dimensioned-complex-number` *inherits* from `complex-number`, and that `complex-number` is a *superclass* of `dimensioned-complex-number`.

Classes can inherit from more than one superclass, in which case their instance's set of slots contains the union of the slots of the superclasses. This introduces an potential problem when the same slot name is used for more than one superclass. How this dealt with, is discussed in detail in CLOS specification. For our purposes, we will assume that this situation never occurs. If it occurs, it is to be treated as an error.

The dimensioning example given above is a good example of where multiple inheritance would be naturally used. There are many different types of numerical objects for which we would like to associate units. To simplify our definitions and to ensure consistent names are used for the initialization keyword and for the accessing routines, we create a second superclass for units, *viz.*

```
(defclass dimensioned-number ()
  ((units :initarg :units :accessor units)))
```

Now the definition of `dimensioned-complex-number` becomes just

```
(defclass dimensioned-complex-number (complex-number dimensioned-number)
  ())
```

This is all that needs to be known about classes and instances in CLOS.

The basic building block of Lisp is functional invocation. In CLOS this mechanism is extended via polymorphism. This is allows the same function name to be used for arguments of different types and with different pieces of code invoked. For example, assume we want to use the function `binary+` to add two numbers, and in particular, we want `binary+` to work for two complex numbers. This can be accomplished by defining the following *method*:

```
(defmethod binary+ ((z1 complex-number) (z2 complex-number))
  (make-instance 'complex-number
                 :real (lisp:+ (real-part z1) (real-part z2))
                 :imag (lisp:+ (imag-part z1) (imag-part z2))))
```

The `lisp:+` function used above is the addition function of Common Lisp.

A method is a chunk of code that is to be applied for arguments that belong to certain classes. Thus when `binary+` is applied to two instances, both of whose classes are subclasses of `complex-number`, the method given above is invoked. Thus if two `dimensioned-complex-numbers` are passed to `binary+` a `complex-number` will be returned.

This can be addressed, and an error message generated when appropriate, by adding the following method:

```
(defmethod binary+
    ((z1 dimensioned-complex-number) (z2 dimensioned-complex-number))
  (cond ((eql (units z1) (units z2))
         (make-instance 'dimensioned-complex-number
                        :real (lisp:+ (real-part z1) (real-part z2))
                        :imag (lisp:+ (imag-part z1) (imag-part z2))
                        :units (units z1)))
        (t (error "Combining dimensioned numbers with different units."))))
```

When two `dimensioned-complex-number`'s are combined using `binary+`, this method is invoked instead of the first method.

Behind the scenes, CLOS creates a *generic function* and puts it in the function cell of the symbol `binary+`. When invoked, the generic function examines the classes of each of its arguments, chooses the most specific, appropriate method and invokes it.

Notice that methods are not associated with particular classes, but with generic functions. This is the major difference between CLOS and most other objected oriented programming environments. It allows CLOS to treat the arguments to generic functions symmetrically.

As with multiple inheritance of classes, it is possible for ambiguities to arise when deciding, which method is most appropriate for a given set of arguments. The CLOS specification provides an precise mechanism for eliminating this ambiguity.

This approach has a problem, namely each time we define a new subclass of `dimensioned-objects` we will have to define a new, specialized version of the `binary+` method. Furthermore, the central code of the `complex-number` `binary+` method is repeated. This problem can be avoided by using the a special type of method, "wraps itself around all other methods." Using this mechanism, we could replace the `dimensioned-complex-number` `binary+` method by the following `:around` method for `dimensioned-object`.

```
(defmethod binary+ :around ((z1 dimensioned-object) (z2 dimensioned-object))
  (cond ((eql (units z1) (units z2))
         (call-next-method z1 z2))
        (t (error "Combining dimensioned numbers with different units."))))
```

When `:around` methods are provided, each of the applicable around methods are collected into a list with the regular or primary method placed at the end of the list. The generic function then invokes the first method in the list. The value returned by this method is the value of the generic function. However, each `:around` method can pass control to the next method on the list by invoking `call-next-method`. The arguments to `call-next-method` are the arguments to the next method on the list, so `:around` methods can be used to modify, or canonicalize the arguments to other methods.

The process of building up methods from pieces, is called *method combination* and can be a very powerful tool. However, it can lead to extremely opaque and hard to follow code if not used judiciously.

There are many other features in CLOS, including more sophisticated forms of method combination and the metaobject protocol, that we have not touch on. However, we hope that this quick introduction makes the discussion of Weyl's internal organization easier to understand.