The Weyl Computer Algebra Substrate*

Richard Zippel

TR 90-1077
January 1990

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

*This work was supported by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research Contract N00014-88-K-0591, ONR Grant N00014-89-J-1946, and NSF Grant DMC-86-17355.

The Weyl Computer Algebra Substrate

Richard Zippel
Cornell University
Ithaca, NY 14853

October 25, 1989

1 Introduction

In the last twenty years the algorithms and techniques for manipulating symbolic mathematical quantities have improved dramatically. These techniques have been made available to practitioners through a number of algebraic manipulation systems. Among the most widely distributed systems are Macsyma [13], Reduce [6], Maple [8] and Mathematica [15]. These systems are all quite similar—they are designed to be used as self-contained computational systems and are not intended to be incorporated in larger, more specialized systems. Thus, these systems are awkward to use on problems that require a mixture of computer algebra, numerical techniques and sophisticated user interfaces and the leverage that can be applied by computer algebra techniques is lost.

This situation seems somewhat odd when compared with what has occurred with linear algebra. Most practitioners use linear algebra libraries like Linpack [5] and Eispack [10], which can be incorporated in larger systems (like fluid dynamics simulators or circuit analyzers). Stand alone linear algebra systems like Matlab [9] are used for research in linear algebra, but relatively few applications are built with them. Furthermore, notice that Matlab was developed after the field of computational linear algebra was mature and after a large number of applications that used linear algebra packages existed. 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.

Another 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. In most systems, the basic mathematical objects dealt with are rational functions in several variables over either finite fields or algebraic extensions of the rational integers. Often univariate power series are also added. To experiment with algorithms for dealing with mathematical objects that are not already provided, like polynomials with power series coefficients or Poisson series, often requires a great deal of effort. This situation is exacerbated by the unavailability of the code that implements the algebraic algorithms of the symbolic manipulation systems.

Scratchpad [7] is a noticeable exception to this trend in that the developers plan to make the algebra code for the system widely available and the internal data typing mechanisms provided

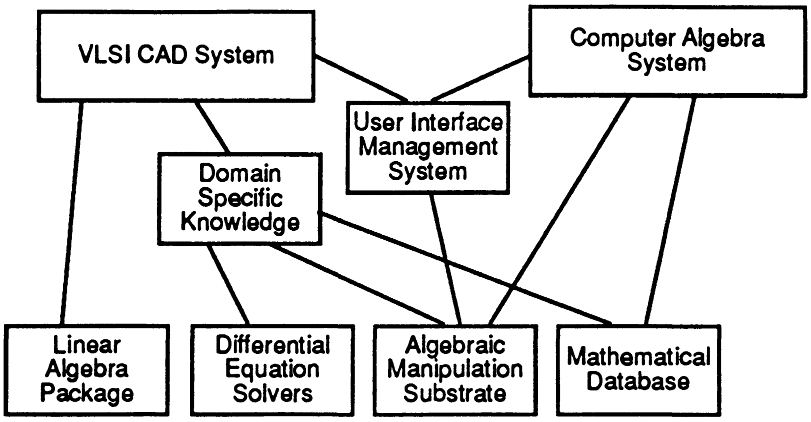

Figure 1: Organization of an application that uses symbolic mathematics

by Scratchpad are designed with the extension mechanisms just mentioned in mind. However, Scratchpad is only available on some IBM equipment and, like the other systems, is designed to be used as a stand alone system.

There do exist a few systems that use symbolic mathematics integrally in their computations such as the VLSI design system Schema [2], the geometric modeling system Geometer [4] and several systems for constructing differential equation solvers [12, 14].

As an example of how these systems integrate symbolic computation into their environment consider how the Schema is used to analyze the transfer characteristics of a linear circuit. The designer uses a schematic editor in Schema to draw a circuit and to indicate its input and output ports. Schema can then be asked to compute the transfer function of the circuit, which it does symbolically, but which it does not present to the designer. The general transfer function of a circuit is often a very large expression, and the designer would not gain any information by looking at it as a ratio of two poylnomials. Schema then asks the designer to specify different numerical values for previously unspecified parameters and presents plots of the transfers functions. This interaction allows the designer to better understand the relationship between different device parameters and the transfer function of the circuit. Though not implemented in Schema, questions like the sensitivity of the circuit to parameter values could easily be answered. Notice that throughout the interaction the designer is oblivious to the underlying symbolic computation.

This, I believe, is typical of how symbolic computation will be used in the future. Users will view symbolic techniques as being as integral to their computational world as numerical techniques and will demand they be as accessible as numerical techniques. We expect symbolic techniques to be organized something like that shown in figure 1. In the bottom layer, there is a algebraic manipulation package along with various numerical and user interface packages. In addition there is a mathematical database that manages information about which functions are analytic, which variables are greater than zero, etc. Above these packages lies the application program that accepts user input and converts it to requests of the algebraic manipulation substrate.

In figure 1, we have shown two different systems. On the right is a conventional system like Macsyma. The algebraic manipulation substrate is exposed to the user and the user is expected to reformulate his or her problems into algebraic terms. In our experience, this is the source of a lot of the difficulty in using symbolic manipulation systems. Either the users have trouble mapping their problems into algebraic terms, or system designer tries to build in some "general domain specific" knowledge which gets in the way of users in different domains. An example of this sort of problem is the confusion that arises in Macsyma between functions of a real variable and functions of complex variable.

Weyl is an attempt to deal with some of these problems. Weyl is an extensible algebraic manipulation substrate that has been implemented in Common Lisp [11] using the Common Lisp Object Standard [1]. This substrate is designed to represent algebraic objects like polynomials, algebraic numbers, matrices and differential forms and higher level objects like groups, rings, ideals and vector spaces. Furthermore, to encourage the use of symbolic techniques within other applications, Weyl is implemented as an extension of Common Lisp so that all of Common Lisp's user interface tools and software development facilities can be used in concert with Weyl's symbolic tools.

At the moment Weyl's structure is appears to be pretty sound. There are tools for dealing with finite fields, polynomials, rational functions, matrices and a few other basic algebraic structure. In addition, algebraic objects like polynomial rings, vector spaces and homomorphisms, which are usually not part of an computer algebra system, have been implemented. However, there are still a large number of computer algebra algorithms that need to be implemented to flesh the system out and make it usable more widely.

2 An Example

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

$$\begin{aligned}
f_0 &= 0 \\
f_n &= -\mu g_{n-1} - \sigma(\mu + 2\epsilon)\frac{\partial f_{n-1}}{\partial \epsilon} + (\epsilon - 2\sigma^2)\frac{\partial f_{n-1}}{\partial \sigma} - 3\mu\sigma\frac{\partial f_{n-1}}{\partial \mu} \\
g_0 &= 1 \\
g_n &= f_{n-1} - \sigma(\mu + 2\epsilon)\frac{\partial g_{n-1}}{\partial \epsilon} + (\epsilon - 2\sigma^2)\frac{\partial g_{n-1}}{\partial \sigma} - 3\mu\sigma\frac{\partial g_{n-1}}{\partial \mu}
\end{aligned}$$

The resulting sequences, f_i and g_i are all polynomials in μ, ϵ and σ with integer coefficients. Thus they lie in the domain $\mathbf{Z}[\mu, \epsilon, \sigma]$.

One of the features that distinguishes Weyl from most other algebra systems is that domains, like $\mathbf{Z}[\mu, \epsilon, \sigma]$, directly enter into the computation. Thus our first task is to generate the *domain* $\mathbf{Z}[\mu, \epsilon, \sigma]$. This done by first getting a copy of the rational integers, $\mathbf{Z}$, using the function `get-rational-integers`. We then construct a polynomial ring in the three variables μ, ϵ and σ over $\mathbf{Z}$. This can be done by the following form:

```
(setf R (get-polynomial-ring (get-rational-integers) '(m e s)))
```

where we have used the symbols `m`, `e` and `s` to represent the greek letters μ, ϵ and σ.

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))
(setf eps (coerce 'e R))
(setf sigma (coerce 's R))
```

The coefficients of the first terms of the both recurrences are the same, so rather than computing them afresh each time, we'll store them in global variables

```
(setq x1 (- mu))
(setq x2 (* (- sigma) (+ mu (* 2 eps))))
(setq x3 (+ eps (* -2 (expt sigma 2))))
(setq x4 (* -3 mu sigma))
```

Notice that we used the usual Lisp functions for manipulating the elements of R. 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.

Finally, we can write out the recursions formulae.

```
(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 given below.

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

We have chosen Lisp as the substrate language for several reasons. First, it is one of the few languages that provides a garbage collector, which greatly simplifies the algebraic algorithms. Second, it is the only language we know of that provides a multiargument dispatching object oriented programming mechanism. Finally, its flexible syntax, macros and package structure makes it much easier to extend with symbolic manipulation facilities than other languages. In principle we could have implemented Weyl in language like C, but this would have increased the implementation difficulty significantly.

3 Weyl's Type Structure

The objects manipulated by Weyl are either *domains*, *domain elements* or *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 ($\mathbf{Z}$), polynomial rings ($\mathbf{Q}[x, y]$, $\mathbf{Z}_p[t]$) and vector spaces ($\mathbf{R}^3$, $P^3(\mathbf{Z})$). The elements of these domains are domain elements.[1] 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 Lisp structure type that is 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 $\mathbf{Z}[x]$. We say that the *structural type* of $x + 1$ is `list` and that its *domain type* is $\mathbf{Z}[x]$. Elements of $\mathbf{Z}[x]$ could be represented as vectors or in other ways, and lists could be used to represent objects in other domains besides $\mathbf{Z}[x]$. Thus the structural type is truly orthogonal to the domain type of an object.

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 2.

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 `abelian-group` must have a `plus` 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 takes an integral domain and produces 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.

---

[1] 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 $\mathbf{Z}$ or $\mathbf{Q}[x, y]$ are domains, and the Lisp object that represents $x^2 + 2y + 1$ is a domain element.

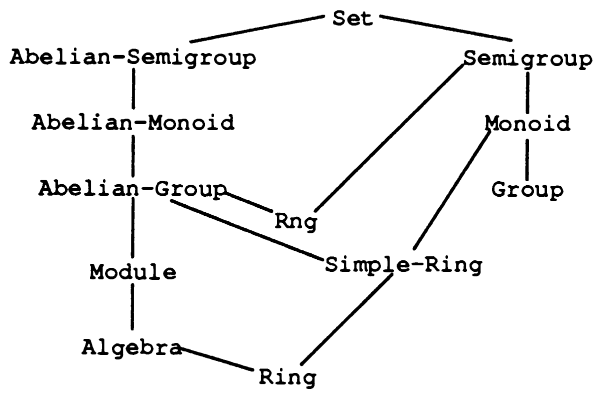

Figure 2: Section of Weyl's Domain Hierarchy

3.1 Domain Hierarchy

The domain hierarchy of Weyl follows that of algebra fairly closely and is modeled on the category hierarchy used in Scratchpad [7]. The root of the hierarchy is the class **set**. A section of the hierarchy is shown in figure 2. As was done in Scratchpad, we have have duplicated the definitions for semigroups, monoids and groups in the definitions for their abelian counterparts. This allows us to be a bit more explicit about the operations they use. There is an implicit assumption that the **plus** operation is always commutative.

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 it 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 **times** operation is closed, and the second that **times** is associative.

```
(define-operations semigroup
  (times (element self) (element self)) -> (element self))

(define-axioms semigroup
  (associative times))
```

This information is used when new domains are created and when choosing certain algorithms. A simple example arises for exponentiation to an integer power. Two algorithms that might be used for exponentiation are repreated 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 Nurpl [3]. 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 techniques are added to the system.

None of the domains in figure 2 can actually be used. 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` domain.

3.2 Morphisms

Morphisms are maps between domains.[2] Though we could just used regular Lisp functions for this purpose, we have decided to encapsulate the mapping functions in an object so that we can add some additional information to the map itself and also to take advantage of the polymorphism and overloading mechanisms provided by CLOS. Morphisms are first class objects in Weyl and can manipulated like domains and domain elements. In particular, two morphisms can be composed using the operation `compose`. Morphisms currently have two user visible components, the domain of the morphism and the range of the morphism.

A few classes 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. In addition, morphisms can injections, surjections or bijections.

Some morphisms are defined by giving just the domain, range and the function that maps the domain to the range. Other morphisms, like those defined by dividing out ideals, have a somewhat richer structure. For some of these richer morphisms it is possible to define some additional operations besides composition. For instance, it is possible to compute the kernel and image of some homomorphisms. This is an example where CLOS's overloading and inheritance mechanisms are useful.

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, if we were to add $1/2 \in \mathbf{Q}$ and $x + 1 \in \mathbf{Z}[x]$ the answer could be either

$$x + \frac{3}{2} \in Q[x] \text{ or } \frac{x+3}{2} \in \mathbf{Z}(x).$$

Because of this type of ambiguity we feel that algebraic algorithms should be coded in a way that makes explicit the coercions take place. Weyl provides two mechanisms for dealing with coercions. First, the programmer can explicitly coerce domain elements from one domain into another using the function `coerce`. `Coerce` only performs *canonical coercions* of its arguments.

[2]This is a slight abuse of the usual mathematical meaning of the term *morphism*.

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

Figure 3: A Sample Computational Context

Thus it will coerce an element of $\mathbf{Q}$ to an element of $\mathbf{Q}[x]$ since $\mathbf{Q}$ is the coefficient field of $\mathbf{Q}[x]$, but it will not coerce an element of $\mathbf{Q}[x]$ into an element of $\mathbf{Q}[x, y]$.

Second, the programmer can create homomorphisms between domains and explicitly indicate which homomorphisms should be in force for a given computation. These homomorphisms will be used by `coerce` whenever appropriate.

The set of domains and homomorphisms which are in force at any point in time is called a *computational context*. Recalling the previous problem of adding $x + 1$ and $1/2$, 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 when application specific systems that use the Weyl substrate are constructed, the particular computational contexts used will be a function of the application.

For convenience sake, we do assume that there is a canonical homomorphism of the rational integers into every domain. If only one of arguments to one of the four basic arithmetic operations is an element of $\mathbf{Z}$, then it is automatically coerced into the domain of the other argument. We will have to see if this particular coercion really is always preferable.

4 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 4.

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 `plus`. 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:

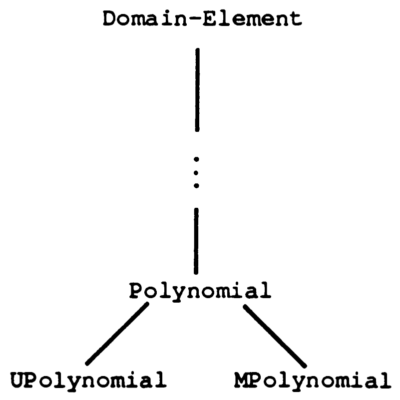

Figure 4: Structural type hierarchy for polynomials

```
(defmethod plus ((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 other three combinations of **mpolynomials** with **upolynomials**.

The reall 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 the one is building a new polynomial algorithm one writes the bulk of the code using functions like `mpoly-plus` and then wrap it within a generic function like `plus`. One retains full generality by using generic functions like `plus` 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)))
                 (spmodular-gcd (poly-form x) (poly-form y)))
                (t (subresultant-gcd (poly-form x) (poly-form y)))))))
          (t (error "Trying to take the GCD of polynomials from ~
                 different domains.")))))
```

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 will need to be rewritten. This is the problem that the Capsules system [16] tries to solve. We have not yet tried to integrate the Capsule ideas into Weyl.

5 Domain Creation

Domains are created in Weyl using special functions called *functors*. Examples of these functors are get-polynomial-ring, which was used in section 2, 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 gcd-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 indicates that the quotient field of a field is just the field itself. The second form is used only if the argument is a GCD domain. It creates an instance of the class quotient-field, which uses the underlying ring's GCD operation to reduce the factions to lowest terms. A different form and class could be used if we wanted to deal with quotient fields over rings that are not GCD domains. In addition, we have decided to cache the values of zero and one in the quotient field to speed later uses.

Algebraic extensions provide an interesting variant of the simple cases given above. In this case, the creator needs to verify that the minimal polynomial of the extension is actually provided, or perhaps to generate one automatically. The first case arises when we try to extend $\mathbf{Q}[\sqrt{2}]$ by the zeroes of $x^4 - 10x^2 + 1$, whose zeroes are $\pm\sqrt{2} \pm \sqrt{3}$. The second case occurs when we ask for a degree 5 extension of $GF(p)$. Here we don't care about the minimal polynomial, just that the system find one for us.

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.

6 Conclusions

In Weyl's current implmentation types all objects at run time and 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 Scratchpad, and some of this is due to the using Xerox's portable implementation of CLOS. This portable implementation of CLOS allows us to run Weyl on a wide variety of platforms, but degrades performance enough that we cannot get an accurate idea of the performance to expect with high performance implementations of CLOS that are integrated into the Lisp compiler. Thus, though we know that some of our design decisions degrade performance somewhat we cannot accurately quantify the degradation.

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 Scratchpad group, in particular Barry Trager and Dick Jenks, whose assistance I gratefully acknowledge.

This report describes research supported by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014–86–K–0591, the National Science Foundation through contract DMC–86–17355 and the Office of Naval Research through contract N00014–86–K-0281.

References

[1] Daniel G. Bobrow, Linda G. DeMichiel, Richard P. Gabriel, Kenneth Kahn, Sonya E. Keene, Gregor Kiczales, Larry Masinter, David A. Moon, Mark Stefik, and Daniel L. Weinreb. Common lisp object system specification. Technical Report 88–002, X3J13, ANSI Common Lisp Standardization Committee, July 1988.

[2] George C. Clark and Richard Eliot Zippel. Schema: An architecture for VLSI CAD. In *Proceedings of the International Conference on Computer Aided Design*, pages 50–52, Santa Clara, CA, October 1985.

[3] 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.

[4] David A. Cyrluk, Richard M. Harris, and Deepak Kapur. GEOMETER: A theorem prover for algebraic geometry. In *Proceedings of the 9th International Conference on Automated Deduction (CADE-9)*, Argonne, IL, May 1988.

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

[6] Anthony C. Hearn. Reduce 3 user's manual. Technical report, The RAND Corp., Santa Monica, CA, 1986.

[7] Richard D. Jenks and Barry Marshal Trager. 11 keys to new Scratchpad. In *Proceedings of EUROSAM '84*, pages 123–147, New York, NY, 1984. Springer-Verlag.

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

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

[10] B. T. Smith, James M. Boyle, Y. Ikebe, Virginia C. Klema, and Cleve B. Moler. *Matrix Eigensystem Routines: EISPACK Guide*. Springer-Verlag, New York, NY, 2 edition, 1976.

[11] Guy Lewis Steele Jr. *Common Lisp, The Language*. Digital Press, Burlington, MA, 1984.

[12] Stanly Steinberg and Patrick J. Roache. Symbolic manipulation and computational fluid dynamics. *Journal of Computational Physics*, 57:251–284, 1985.

[13] Symbolics, Inc., Burlington, MA. *MACSYMA Reference Manual*, 14 edition, 1989.

[14] Paul S. Wang, T. Y. P. Chang, and K. A. van Hulzen. Code generation and optimization for finite element analysis. In John Fitch, editor, *Lecture Notes in Computer Science 174, EUROSAM '84*, pages 237–247, New York, NY, 1984. Springer-Verlag.

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

[16] Richard Eliot Zippel. Capsules. *SIGPLAN Notices*, 18(6):166–169, June 1983. Proceeedings of SIGPLAN '83.