The Weyl Computer Algebra Substrate Manual

SimLab Group[1]
Cornell University
Ithaca, NY 14853

April 9, 1995

[1]This research was supported in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research Contract N00014–92–J–1989, the National Science Foundation through grant IRI–9006137, the Office of Naval Research through contract N00014–92–J–1839 and in part by the U.S. Army Research Office through the Mathematical Science Institute of Cornell University.

This page intentionally left blank.

Contents

Chapter 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 a practitioners through a number of algebraic manipulation systems. Among the most widely distributed systems are Macsyma [9], Reduce [3], Maple [5] and Mathematica [10]. These systems are designed to be self-contained and are not intended to be incorporated into larger, more specialized systems. This is at odds with the experience with numerical computation where libraries of carefully coded routines like Linpack [2] and Eispack [7] have been of the greatest value because they could be incorporated in larger systems (like fluid dynamics simulators or circuit analyzers). Linear algebra systems like Matlab [6], though of significant value, where developed much later and tend to be used more for research in linear algebra than as part of large computations.

Another limitation of current symbolic mathematics systems is that they generally deal with a relatively limited and fixed set of algebraic types, which the user is not expected to extend significantly. Thus it may be difficult for a user to experiment with algorithms for dealing with Poisson series if the algebra system does not already have a data type that matches the behavior of Poisson series. This situation is exacerbated by the unavailability of the code that implements the algebraic algorithms of the symbolic manipulation system. Scratchpad [4] 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 by Scratchpad are designed with the extension mechanisms just mentioned in mind.

Weyl is an extensible algebraic manipulation substrate that has been designed to represent all types of algebraic objects. It deals not only with the basic symbolic objects like polynomials, algebraic functions and differential forms, but can also deal with 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 [8] using the Common Lisp Object Standard [1] so that all of Common Lisp's facilities and development tools can be used in concert with Weyl's symbolic tools.

It should be noted that the initial implementation of Weyl is intended to be as clean and semantically correct as possible. The primary goal of Weyl is provide the tools needed to express algebraic algorithms naturally and succinctly. Nonetheless, we believe that algebraic algorithms can be efficiently implemented within the Weyl framework even though that was not a goal of the initial implementation.

1.1 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 the function $1/(x^3-2)$:

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

Notice that, although the original problem involved rational functions with integer coefficients ($\mathbb{Z}$), 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. 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 often new algebraic quantities must be added.

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 symbol $\sqrt{5}$ be interpreted, *i.e.*, how does one know that $(\sqrt{5})^2=5$, and is $\sqrt{5}$ equal to 2.2361 or -2.2361? Furthermore, after solve has been asked to solve x^2-5, what is returned as the solution of 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 in Weyl by using *domains*. Domains are collections of objects, usually with operations among the 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, $\mathbb{C}$. On the other hand, we rarely think of a set of numbers, *e.g.*, $\{1492, 1776, 1917\}$, as having any mathematical structure and being a domain.

In the example above, solve is first called on x^2-5, which is an element of $\mathbb{Q}[x]$. For the duration of this section, we will use the notation $(x^2-5)_{\mathbb{Q}[x]}$ when it is necessary to indicate 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 could 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 both the simplification rules that $\sqrt{5}$ must obey and the embedding in $\mathbb{C}$. Notice that this information is associated with the domains, rather than attached 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 $(x^2-20)_{\mathbb{Q}[\sqrt{5}][x]}$, however, the coefficient field does not 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 $\sqrt{20}$.

Weyl provides mechanisms to support multiple distinct but isomorphic domains in the same computation. Why would one want to have two copies of $\mathbb{Z}$, say, present during a computation? Algebraic extensions provide one 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. Since each has a different embedding in the complex numbers, they are, in fact, different algebraic extensions of $\mathbb{Q}$. However, algebraically each extension is a simple cubic extension 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 elements of these fields can be represented as univariate polynomials modulo an irreducible univariate polynomial. Algebraically these fields are isomorphic. The only difference between these three fields is their embedding in $\mathbb{C}$, *i.e.*,

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

Thus we need to be able to represent distinct fields that are algebraically isomorphic. Only when their embedding into the complex numbers is specified is it possible to distinguish them.

In addition, 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 subdomains, k for the coefficients and $\mathbb{Z}$ for the exponents.

The value of this approach is apparent when examining polynomial differentiation. In that algorithm an exponent must be multiplied with coefficient. 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.

Chapter 2

Getting Started

Weyl is implemented as a large body of functions that extend Lisp's functionality to include non-numeric algebraic computations. These functions can be loaded into an existing Lisp image using the tools described in Section 2.1. All of Weyl's extensions are attached to symbols in the weyl or weyli packages. The philosophy behind these packages and how they are to be used is discussed in Section 2.2. Some issues with developing programs using Weyl are discussed in Section 2.3. In Section 2.4 we give a simple of example what is involved when using Weyl.

2.1 Creating a Weyl World

Common Lisp does not include a *defsystem* facility, which is often used to manage the compilation dependencies between different files in a Lisp system.[1] Although many Lisp vendors provide a defsystem, there is no standard, and each vendor's version differs slightly from the others. As a consequence, we have chosen not to use any of them and instead include with Weyl a copy of Mark Kantrowitz's version.

The first thing one must do to get Weyl started is to load the file sysdef.lisp. This file creates the weyl package, where all of Weyl's symbols are kept, describes which files comprise the Weyl systems, and provides a number utility functions to help manage the Weyl sources. These functions are all defined in the user package, which is the package in which Lisp's usually start up. Thus you will probably not need to include the user:: prefix that we have included in this section.

The function user::compile-weyl will recompile any Weyl sources that need recompilation. To load the the recompiled files use user::load-weyl.

user::compile-weyl [*Function*]

Compile those Weyl source files that need recompilation. Increment Weyl's minor version number.

user::load-weyl [*Function*]

Load all binary files for Weyl and initialize Weyl's data structures. If Weyl has already been loaded, this function loads any binary files that are more recent than the ones already loaded. This is commonly used as a stripped down "load-patches."

[1] The defsystem facility plays the same roles as `make` files do in Unix systems-building.

user::dump-weyl [*Function*]

Loads the current binary files for Weyl and dumps a world.

2.2 Packages

In Common Lisp, values and variables are attached to symbols. Collections of symbols are organized into *packages* to control access to the values and functions. The symbols used by Weyl are contained in two Common Lisp packages, weyl and weyli. Those symbols that are intended for users are contained in the weyl package and exported. The internal routines of Weyl are contained in the weyli package so that they will not interfere with the user's symbols.

Common Lisp places all of its own symbols in the lisp package. Weyl extends the functionality of certain function in Common Lisp by providing its own version of certain symbols from the lisp package. Weyl's version of these symbols *shadow* the Common Lisp symbols. Among these symbols are the arithmetic functions, ($+$, $-$, $*$, ...), most of the mathematical functions (sqrt, sin, ...) and some of the sequence functions (map, delete, member, ...). The variable *weyli-shadowed-symbols* contains a list of all shadowed symbols. In all cases, when applied to arguments admissible to the Common Lisp version of the function, the Weyl versions of these functions perform in exactly the same fashion as the Common Lisp version. The only difference is that the Weyl versions have been extended to work with the mathematical objects defined in Weyl.

The only symbols in the weyl package are those intended for users. The internal functions and values that are maintained by the system itself and that are needed only by those extending Weyl are contained in the weyli package.

In some cases there are symbols with the same name in both the weyl and the weyli packages. This is usually done when a version with more error checking is needed by users, but where such error checking would compromise the performance of Weyl if used internally where such error checking is not necessary.

An example of this is the make-element function. When constructing vectors and matrices, weyl:make-element makes sure that each element of the vector or matrix to be created belongs to the coefficient domain, and if not tries to coerce the element into the coefficient domain. The function weyli::make-element does not, but it is only used internally in the vector and matrix routines.

Most projects that use Weyl will create their own package for their routines. Because of the shadowing done by the weyl package, if one simply included weyl in the user's new package one would get many conflicts. Instead one should use the function use-weyl-package, which will perform the necessary shadowing.

use-weyl-package *package* [*Function*]

Does a shadowing import of the shadowed symbols in the weyl package and then makes *package* use the package weyl.

Thus a common idiom for creating a package for an new application that uses Weyl is

```
(unless (find-package 'Application-Package)
  (make-package "Application-Package" :use '(lisp clos))
  (use-weyl-package (find-package 'Application-Package)))
```

2.3 Programming Style

In many cases the symbols in the weyl package shadow normal Common Lisp symbols. When this is done, the functions in the weyl package are upward compatible extensions of the Common Lisp ones. For instance, the Common Lisp function lisp:expt can only be applied to Lisp numbers, while the function weyl:expt can be applied to polynomials, rational functions and other mathematical objects as well as numbers. In all cases, the weyl versions of these shadowed functions are upwardly compatible with the Common Lisp versions.

We have adopted the convention that the name of all boolean predicates end with a ?. This is different from the convention adopted by Common Lisp, where most predicates end with a p. Thus the Weyl function which determines if a number is positive is plus?, where the Common Lisp function is plusp.

2.4 F and G Series Computation

In this section we show how to perform a simple algebraic calculation using Weyl. Throughout this section we will use terms, such as *domains*, that are not yet well defined. These terms will be defined in later sections.

First one needs to start a Lisp image that contains Weyl. Using Lucid's lisp on a Decstation we get a prompt like:

```
;;; Lucid Common Lisp/DECsystem
;;; Development Environment Version 4.0, 12 November 1990
;;; Copyright (C) 1985, 1986, 1987, 1988, 1989, 1990 by Lucid, Inc.
;;; All Rights Reserved
;;;
;;; This software product contains confidential and trade secret information
;;; belonging to Lucid, Inc.  It may not be copied for any reason other than
;;; for archival and backup purposes.
;;;
;;; Lucid Common Lisp is a trademark of Lucid, Inc.
;;; DECsystem is a trademark of Digital Equipment Corp

;;; Weyl Version 4.55, saved 14:38 Thursday, October 3, 1991

>
```

The first thing that should be done at this point is to type (user::load-weyl) which will cause any bug fixes and improvements that have been created since the Lisp image was dumped was loaded, as illustrated below:

```
> (user::load-weyl)
;;; Loading file #P"/amd/nori/a/weyl/develop/lisp-support.mbin"
;;; Done loading system USER::WEYL
;;; Weyl 4.56 loaded.

> (in-package 'weyl)
#<Package "WEYL" 1028943E>
```

We have also switched the weyl package so that can easily access all the functions in Weyl. We are now ready to begin.

Computations using Weyl are usually involve first creating a set of domains in which to perform a computation, creating a few elements of these domains to initiate the computation, and then performing the computation itself. This is illustrated in the following routines that computes the coefficients of the "F and G series." These coefficients are defined by the following recurrences.

$$f_0 = 0$$

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

$$g_0 = 1$$

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

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 $\mathbb{Z}[\mu, \epsilon, \sigma]$. Thus our first task in computing them is to generate the domain $\mathbb{Z}[\mu, \epsilon, \sigma]$. This can be done by the following line:

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

It is useful to have our hands on the variables μ, ϵ and σ. This is done by coercing the symbols m, e and s into 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, we'll store them in some global variables

```
> (setq x1 (* (- sigma) (+ mu (* 2 eps))))
- s m + -2 s e

> (setq x2 (+ eps (* -2 (expt sigma 2))))
e + -2 s^2

> (setq x3 (* -3 mu sigma))
-3 s m

> (setq x4 (- mu))
- m
```

Notice that we used the usual Lisp functions for manipulating the elements of R. Second, there is an automatic coercion from the integers into almost every domain we are working with, thus we could write (* 2 eps) above, rather than the more explicit (* (coerce 2 R) eps).

Finally, we can write out the recursion formulae.

```
(defun f (n)
  (if (= n 0) (coerce 0 R)
    (let ((fn-1 (f (1- n))))
```

```
      (+ (* x4 (g (1- n)))
         (* x1 (partial-deriv fn-1 eps))
         (* x2 (partial-deriv fn-1 sigma))
         (* x3 (partial-deriv fn-1 mu))))))

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

And compute f_5 and g_5

```
> (f 5)
- m^3 + (-24 e + 210 s^2) m^2 + (-45 e^2 + 630 s^2 e + -945 s^4) m

> (g 5)
-30 s m^2 + (-180 s e + 420 s^3) m
```

The explicit coerce's in the definitions of f and g are included to ensure that f and g always return elements of the same ring.

Chapter 3

Generic Tools

Some facilities needed by symbolic computing routines are more generally useful. For instance, it is occasionally necessary to perform some function for every permutation of the elements of a set. To capture this idiom, Weyl provides a new control structure, called permute. This control structure, and other combinatorial control structures, are described in Section 3.1. Other data and control structures that are not directly tied to mathematical computation are given in the other sections.

3.1 Combinatorial Tools

Weyl provides control abstractions that are often useful for computations that involve combinatorial enumeration and searching. These control structures are permute and choose, which enumerate all permutations of a sequence and all subsets of a fixed size of a sequence.

permute *sequence (var . options)* &body *body* [*Special Form*]

> *sequence* is a sequence of elements.[1] The variable *var* is repeatedly bound to the different permutations of *sequence*, and *body* is evaluated each time. The options are provided to specify even and odd permutations, but are not at this point implemented.

For example, the following code will print all permutations of the list (a b c) and count their number:

```
> (let ((count 0))
    (permute ' (a b c) (p)
      (print p)
      (incf count))
    (format t "~%~D permutations total. ~%" count))

(C B A)
(B C A)
(C A B)
(A C B)
(B A C)
(A B C)
6 permutations total.
NIL
```

[1] At the moment this control structure is only implemented for lists.

choose *set (var n . options)* &body *body* [*Special Form*]

While the first argument to permute must be ordered, the first argument to choose need only be a set (but again, only lists are currently implemented). The variable *var* is bound to each subset of *set* that has precisely *n* elements and *body* is evaluated in each case. At the moment no options are permitted.

A *partition* of a positive integer *n* is a representation of *n* as a sum of positive integers. The following control structure is used to enumerate partitions.

partition *(var n . options)* &body *body* [*Special Form*]

The argument *n* is assumed to be an integer. This control structure repeatedly binds *var* to additive partitions of *n* and then evaluates the *body*. The options allow one to control which partitions of *n* are produced. The options are

`:number-of-parts`	The number of parts each partition is allowed to contain.
`:minimum-part`	The minimum value each for each of the components of the partition.
`:maximum-part`	The maximum value each for each of the components of the partition.
`:distinct?`	If specified as `T` then each of the components of the partition must be distinct.

Partition returns no values.

The following examples illustrate the use of the partition control structure. First, compute all of the partitions of 6:

```
> (partition (l 6) (print l))
(1 1 1 1 1 1)
(2 1 1 1 1)
(3 1 1 1)
(2 2 1 1)
(4 1 1)
(3 2 1)
(5 1)
(2 2 2)
(4 2)
(3 3)
(6)
```

Now restrict the partitions to those that do not contain 1's or those that consist of precisely 3 components:

```
> (partition (l 6 :minimum-part 2)
    (print l))
(2 2 2)
(4 2)
(3 3)
(6)

> (partition (l 6 :number-of-parts 3)
```

```
    (print l))
(4 1 1)
(3 2 1)
(2 2 2)
```

We can further restrict the partitions to only include components that contain components no larger than 3 and to those partitions that consist of distinct components.

```
> (partition (l 6 :number-of-parts 3 :maximum-part 3)
    (print l))
(3 2 1)
(2 2 2)

> (partition (l 6 :number-of-parts 3 :maximum-part 3 :distinct? t)
    (print l))
(3 2 1)

> (partition (l 6 :distinct? t)
    (print l))
(3 2 1)
(5 1)
(4 2)
(6)
```

It is well known, and easy to prove, that the number of partitions of an integer n into m parts is equal to the number of partitions of n where the largest component of the partition is m. We can check this numerically with the following functions. The first function counts the number of partitions of n with exactly m parts. For this the :number-of-parts option suffices.

```
(defun partition-count-fixed-parts (n m)
  (let ((cnt 0))
    (partition (part n :number-of-parts m)
      (declare (ignore part))
      (incf cnt))
      cnt))
```

The function partition-count-exact-max computes the number of partitions where the maximum component is exactly m. In this case, the :maximum-part option helps filter the partition, but then an additional test needs to be applied to ensure that each partition actually has an element of size m.

```
(defun partition-count-exact-max (n m)
  (let ((cnt 0))
    (partition (part n :maximum-part m)
      (when (= m (apply #'cl::max part))
        (incf cnt)))
    cnt))
```

Finally we provide a routine for testing the functions given above.

```
(defun partition-test-1 (n m)
  (let ((part-count1 (partition-count-fixed-parts n m))
        (part-count2 (partition-count-exact-max n m)))
    (list (= part-count1 part-count2)
```

```
            part-count1 part-count2)))

> (partition-test-1 10 3)
(T 8 8)

> (partition-test-1 15 3)
(T 19 19)

> (partition-test-1 15 4)
(T 27 27)
```

3.2 Memoization

It often occurs that the straightforward way of expressing an algorithm is quite inefficient because it performs a great deal of recomputation of values that were previously computed. The simplest example along these lines is the Fibonacci function

```
(defun fib (n)
  (if (< n 2) 1
      (+ (fib (- n 1)) (fib (- n 2)))))
```

Due to the double recursion, this implementation can take time exponential in n. By caching values of (fib n) to avoid recomputation, it is possible to reduce this to a computation that is linear in n. This is easily done using the control abstraction memoize. For instance, we have the following efficient version of the Fibonacci function:

```
(defun fib-memo (n)
  (memoize `(fib ,n)
    (if (< n 2) 1
        (+ (fib-memo (- n 1)) (fib-memo (- n 2))))))
```

The memoize form should be understood as follows. The first argument is an expression that identifies a particular computation. The above example, the form `(fib ,n) represents the computation of the n-th Fibonacci number. The body of the memoize form contains the code that actually performs the computation. The memoize form checks a cache to see if the computation indicated by the first argument has been previously performed. If so, the value returned by the earlier computation is returned immediately. If not the body is executed and the value cached for future use.

The cache used by the memoize control structure is handled by the class weyli::has-memoization. The domain of general expressions (discussed in Chapter 6) is always a subclass of weyli::has-memoization and is the domain with which memoization is usually associated. The internal function weyli::%memoize allows one to associate a memoization with any subclass of weyli::has-memoization.

weyli::%memoize *domain expression* &body *body* [*Method*]

> Each time this form is executed, it checks to see if *expression* is cached in *domain*'s memoization cache. If so, the value in the cache is returned without executing the body. If *expression* is not found in the cache, then the forms in body are evaluated, and the value of the last form is both returned and saved in *domain*'s memoization cache.

It is usually much more convenient to use the memoize control structure:

memoize *expression* &body *body* [*Special Form*]

Performs the same functions as weyli::%memoize except that the domain used is *general*.

One should note that the expression which is used as the index of the cache cannot be a constant. It should contain all of the information that can influence the value returned by the body. Also, at this writing, only the first value returned by the expression being memoized is actually cached. The other values are ignored.

3.3 Tuples

Instances of the class weyli::tuple are simply one-dimensional vectors. Generally, one doesn't create instances of bare tuples, but rather includes the weyli::tuple class with some other classes to create more interesting objects. For instance, Weyl vectors are instances of a class that than includes tuple and domain-element. When creating an instance of a weyli::tuple one needs to initialize a slot that contains the values of the tuple. This can be done by using the :values initialization keyword. For instance,

```
> (setq tup (make-instance 'weyli::tuple :values '(1 2 3)))
<1, 2, 3>
```

The initialization value can be either a list or a Lisp vector.

Since tuples are instances of a class, various methods can be overloaded to work with them. The simplest such function is length, which computes the number elements in the tuple. Another useful function is ref, which accesses different elements of the tuple by their index. The generic function ref is used in place of the Common Lisp functions aref or svref.

ref *sequence* &rest *indices* [*Macro*]

Accesses the indicated elements of sequence. Tuples are one-dimensional arrays so only one index is allowed. The indexing scheme is zero based, so the first element of the tuple has index 0, the second 1 and so on.

For example,

```
> (list (ref tup 0) (ref tup 2) (ref tup 1))
(1 3 2)
```

It is sometimes useful to be able to convert a tuple into a list of its elements. This can be done with the following function:

list-of-elements *tuple* [*Method*]

Returns a list of the elements of *tuple*. For example,

```
> (list-of-elements tup)
(1 2 3)
```

The following functions extend the Common Lisp mapping functions to work with tuples as well as the usual sequences of Common Lisp.

map *type function tuple* &rest *sequences* [*Method*]

The number of arguments of *function* is expected to be one more than the number of elements in *sequences*. *function* is applied to each element of *tuple* and the corresponding elements of each of the elements of *sequences*. For instance,

```
> (map 'tuple #'cons tup tup)
<(1 . 1), (2 . 2), (3 . 3)>
```

Algebraic objects in Weyl have a slot that indicates the algebraic domain of which the object is an element. When creating an instance of such an object it is necessary to indicate this domain. If the sequence returned by map requires this information then the domain will be extracted from the tuple. If it is necessary to explicitly specify the domain of the resulting tuple, the following function may be used.

map-with-domain *type domain function tuple* &rest *sequences* [*Method*]

Similar to map but the domain of the resulting tuple will be *domain*.

3.4 Arithmetic with Lists

In Lisp, lists are extremely convenient data structures, especially for temporary results in computations. To make them even more useful, Weyl provides a couple of functions for performing arithmetic operations with lists. These routines use the Weyl arithmetic operations, and thus can be used both for arrays and lists that contain Lisp numbers and those those that contain Weyl's mathematical objects [*This section needs to be fleshed out. I believe that there are many similar operations that should be added. Perhaps we should just overload the standard arithmetic operations? –RZ*] [*Or we could do something like the* `Listable` *attribute in Mathematica. –TW*]

list-inner-product *list1 list2* [*Method*]

Computes the sum of the pairwise product the elements of *list1* and *list2*.

list-expt *base-list expt-list* [*Method*]

Returns a list of consisting of the elements of *base-list* raised to the power of the corresponding element of *expt-list*.

array-times *array1 array1* [*Method*]

Checks to make sure the arguments of the of the right dimensions and then computes a matrix product storing the result in a new, appropriately sized matrix.

3.5 AVL Trees

Chapter 4

Basics

This chapter describes the basic tools used by Weyl to model the algebraic structures of mathematics. The fundamental difference between Weyl and most other systems is that in addition to providing a representation for domain elements like polynomials, Weyl also provides representations for the algebraic domains of which the polynomials are elements. In Section 4.2 we discuss *domains* and *domain elements*. Section 4.3 discusses morphisms, which are used to define the relationships between the elements of two domains. Section 4.4 illustrates how to make use of these relationships. Section 4.5 presents the hierarchy of domains being used in Weyl. Weyl uses the Common Lisp Object System quite heavily. We begin this chapter with a brief introduction to CLOS, focusing on those points that are relevant to Weyl.

4.1 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[1] and most commercial Common Lisp vendors provide efficient implementations. This introduction hits only the highlights needed for Weyl. Many other features are present and 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 derived from a *class* of similar objects. For example, we might want to construct the class to represent complex numbers of the form $a + bi$, where i is $\sqrt{-1}$. Each instance of this class contains two slots, which we indicate by the names a and b, in which the real and imaginary parts of the complex number are stored. The class of such instances might be called complex-number and defined as:

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

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

```
(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:

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

The keywords provided with the :initarg options in the class definition are used to indicate the initializing values for each slot. Their relationship to the slots is given in the defclass form that defines complex-number.

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.[2] 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 (complex-number)
  ((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 a 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 instances' set of slots contains the union of the slots of the superclasses. This introduces an potential problem when the same slot name is used in more than one superclass. How this dealt is with is discussed in detail in the CLOS specification [1]. 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. This could be done using the following definition:

```
(defclass dimensioned-floating-number ()
  ((float :initarg :value :accessor float-value)
   (units :initarg :units :accessor units)))
```

However, the definition of the units slots is now repeated in two different class definitions. Thus modifications to one slot may require modifications at several places in the source code.

To simplify our definitions and to ensure that 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

[2] Recall that setf is the extensible assignment operator used by Common Lisp. Its first argument is an accessor and its second argument is the new value that is stored in the location referred to by the accessor. It can be used to update the value of variables and modify lists, arrays and structures. Since arbitrary functions can be used to update these structures, nontrivial computations can be caused by the use of setf.

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

Similarly, the definition of dimensioned-float should be

```
(defclass floating-number (dimensioned-number)
  ((float :initarg :value :accessor float-value)))

(defclass dimensioned-floating-number
    (floating-number dimensioned-number)
  ())
```

Notice that we split off a floating-number class so that dimensioned-floating-number is the combination of two classes that provide orthogonal functionality, in accordance with good software-engineering practice.

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

The basic building block of Lisp is function 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 plus to add two numbers. In particular, we want plus to work for two complex numbers. This can be accomplished by defining the following *method*:

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

The + function used above is the addition function of Common Lisp. Similarly, we should create a plus method for combining two floating-numbers.

```
(defmethod plus ((z1 floating-number) (z2 floating-number))
  (make-instance 'floating-number
                 :value (+ (float-value z1) (float-value z2))))
```

Behind the scenes, CLOS creates a *generic function*, which it places in the function cell of the symbol plus. When invoked, the generic function chooses the most specific method for the classes of its arguments and invokes it. The code for choosing the most appropriate plus method is generated at run time, after all the plus methods are known. In addition, the process of invoking the selected method is usually done using a mechanism that is more efficient than a standard function call. This allows the code to be quite efficient.

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 a precise mechanism for eliminating this ambiguity.

Notice that methods are not associated with particular classes like complex-number or floating-number as is done in languages like Smalltalk, but with generic functions. An example of where this is of value is defining a method for adding a complex-number and floating-number:

```
(defmethod plus ((z1 complex-number) (z2 floating-number))
  (make-instance 'complex-number
                 :real (+ (real-part z1) (float-value z2))
                 :imag (imag-part z1)))
```

This "generic function oriented" object-oriented programming style is the major difference between CLOS and most other objected-oriented programming environments. It allows CLOS to treat the arguments to generic functions symmetrically, without bias towards one or the other.

If plus is applied to two dimensioned-complex-numbers, the value returned will be a complex-number, not a dimensioned-complex-number. Furthermore, nothing will check to see if the units of the two complex numbers match. Thus we really could add apples and oranges. This problem can be addressed by adding the following method:

```
(defmethod plus ((z1 dimensioned-complex-number) (z2 dimensioned-complex-number))
  (if (eql (units z1) (units z2))
      (make-instance 'dimensioned-complex-number
                     :real (+ (real-part z1) (real-part z2))
                     :imag (+ (imag-part z1) (imag-part z2))
                     :units (units z1))
      (error "Combining dimensioned numbers with different units.")))
```

Notice that when the dimensions of two complex-numbers differ this method will signal an error. A more sophisticated method might perform a unit conversion.

In order to distinguish the different plus methods (there are now three), we will qualify the names of the methods with the classes of their arguments. So this last plus method is the (plus dimensioned-complex-number dimensioned-complex-number) method.

The way in which we coded this last plus method has a problem. Each time a new subclass of dimensioned-objects is defined, we will have to define a new, specialized version of the plus method. Furthermore, the central code of the complex-number plus method is repeated. This problem can be avoided by using a special type of method, called an :around method, which "wraps itself around all other methods." When :around methods are provided, each of the applicable "around" methods are collected into a list with the regular or primary methods 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.

Using this mechanism, we can replace the (plus dimensioned-complex-number dimensioned-complex-number) method by the following :around method for dimensioned-object.

```
(defmethod plus :around ((z1 dimensioned-number) (z2 dimensioned-number))
  (if (eql (units z1) (units z2))
      (let ((sum (call-next-method z1 z2)))
        (setf (units sum) (units z1))
        sum)
      (error "Combining dimensioned numbers with different units.")))
```

This method checks to see if its two arguments have the same dimensions. If so, it has the remaining applicable methods perform the addition. It then updates the units slot of the value to be returned.

Unfortunately, this is not enough. The basic plus methods, those for adding two complex-numbers or two floating-numbers, will not return objects of the right class. In particular, they will not create instances of classes that include the dimensioned-object class. However, this is easily fixed with a slight change to the basic methods. For instance, the complex-number plus method should be modified to:

```
(defmethod plus ((z1 complex-number) (z2 complex-number))
  (let ((z1-class (class-of z1))
        (new-class (if (eql z1-class (class-of z2)) z1-class
                       'complex-number)))}
    (make-instance new-class
                   :real (+ (real-part z1) (float-value z2))
                   :imag (imag-part z1)))
```

With these techniques, any class that admits addition can be extended to a dimensioned class that also admits addition, and where addition is properly coded. This use of "around" methods allows us to divide the code into more orthogonal components than is possible with more conventional functional programming approaches.

The process of building up methods from pieces, as was done in this example, is called *method combination* and is 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 touched on. However, we hope that this quick introduction will make Weyl's internal organization easier to digest.

4.2 Domains

The objects that are usually manipulated in algebraic computation systems—integers, polynomials, algebraic functions, etc.—are called "domain elements" in Weyl. They are elements of algebraic objects called *domains*. Examples of domains are the ring of (rational) integers $\mathbb{Z}$, the ring of polynomials in x, y and z with integer coefficients, $\mathbb{Z}[x, y, z]$, and the field of Gaussian numbers, $\mathbb{Q}[i]$.

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 when constructing domains includes classes corresponding to groups, rings, fields and many other familiar types of algebraic domains.

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.[3] 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 used to implement the operations. As an example in a less mathematical context, consider the difference between binary trees and balanced binary trees.

In addition we provide information about the permissible operations involving elements of the domain. 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 obeys a predicate.

We attach information about the permissible operations on the elements of a domain to the domain itself. This ensures that when an instance of a domain is instantiated, the appropriate oper-

[3] An integral domain R is ring that does not have any zero divisors. That is, for all $a, b \in R$, if $ab = 0$ then either $a = 0$ or $b = 0$.

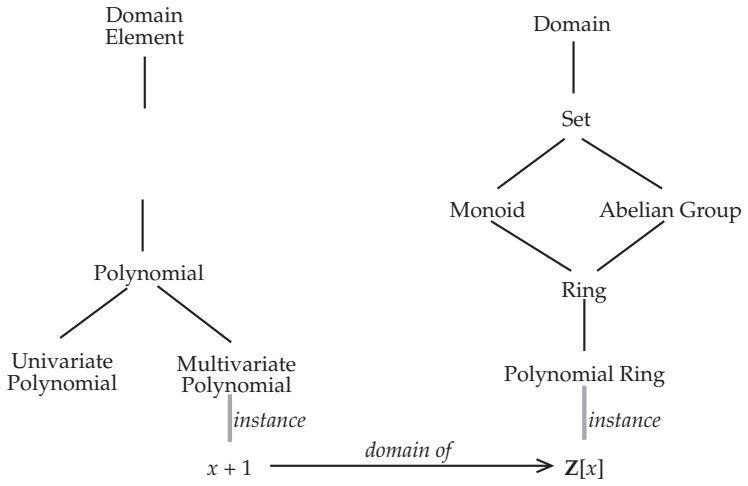

Figure 4.1: Domain Example

ations have been provided. This is especially valuable in an incremental development environment like those common for Lisp.

Domain elements are also implemented as instances of CLOS classes, but in addition they are associated with a particular domain. This association is implemented by a reference in the domain element to the containing domain. The CLOS class structure provides the "structural type" referred to earlier.

Figure 4.1 illustrates the relationship between domains, domain elements and CLOS classes for a simple example. The polynomial $x + 1_{\mathbb{Z}[x]}$ is an instance of the CLOS class polynomial. The domain $\mathbb{Z}[x]$ is an instance of the class polynomial-ring, which inherits from Ring, monoid and abelian-group. The polynomial $x + 1_{\mathbb{Z}[x]}$ is an element of the domain $\mathbb{Z}[x]$. Notice that there is a complete type hierarchy sitting above the class polynomial-ring.

There are always two functions for obtaining a domain. The one that begins "weyli::make-" always creates a new domain. The one beginning with "get-" tries to find an isomorphic domain that already exists. Thus (get-rational-integers) always returns the same rational integer domain, while (make-rational-integers) always creates a new instance of the rational integers.

4.2.1 Subdomains and Superdomains

In some circumstances it is useful to be able to create a domain that is a subdomain of a larger domain. For instance, the positive integers are viewed as a subdomain of the rational integers. The elements of an ideal of ring form an additive group that is a subdomain of the original ring.

super-domains-of *domain* [*Generic*]

Returns the superdomains that contain *domain*.

At the moment, nothing uses this mechanism. The superdomain concept is intended for dealing with deductive questions like: Are there any zero divisors in the set of positive integers? Although the set of positive integers is not an integral domain, it is a subsemigroup of an integral domain and thus none of the positive integers is a zero divisor. Similarly, it is intended that an ideal of a ring R will be implemented as an R-module that is also a subdomain of R.

4.3 Morphisms

Morphisms are maps between domains. They 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 are created using the function make-morphism, which takes three arguments.

make-morphism *domain mapping range* &key *(replace?* T*)* [*Generic*]

This function creates a morphism from *domain* to *range*. *Mapping* is a function of one argument that takes an element of *domain* and returns an element of *range*. If *replace?* is true, then any existing morphisms between *domain* and *range* are deleted before the new morphism is created.

All morphisms created are remembered in the *morphisms* data structure. (Initially, this is just a list.) To find any existing homomorphisms we can use the function get-morphisms.

get-morphisms &key *domain range* [*Function*]

When given no arguments this function returns a list of all the morphisms Weyl currently knows about in this context. When the *domain* is provided, it returns all morphisms from *domain* to anywhere. Similarly, when *range* is provided, only those morphisms that map into *range* are returned. When both *range* and *domain* are given, then only those morphisms from *domain* to *range* are returned.

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, an injection, a surjection, an isomorphism and an automorphism. At the moment nothing takes advantage of this information.

4.4 Coercions

Explicit coercions are performed by the function coerce:

coerce *element domain* [*Generic*]

Coerce finds an element of *domain* that corresponds with *element*. This is done using one of two methods. First, there may be a "canonical" coercion, which is one that is defined via explicit "coerce" methods. These methods take care of mapping Lisp expressions, like numbers and atoms, into Weyl domains. If there are no canonical coercion methods then coerce checks to see if there is a unique morphism between *element*'s domain and *domain*. If so, this morphism is used to map *element* to *domain*. If there is more than one morphism then an error is signaled.

If the the switch *allow-coercions* is set to false (nil) then the canonical maps are the only ones that will be used by coerce. However, if the user sets *allow-coercions* to T then by creating a *homomorphism* between two domains, the set of canonical maps between domains can be extended. If coerce cannot find any other predefined mapping between the domain of *element* and *domain* it then searches the set of all defined homomorphisms. If there exists a *canonical homomorphism* between the two domains then it is used to map *element* into *domain*. If there does not exist a canonical homomorphism but there is exactly one homomorphism between the two domains, then it is used.

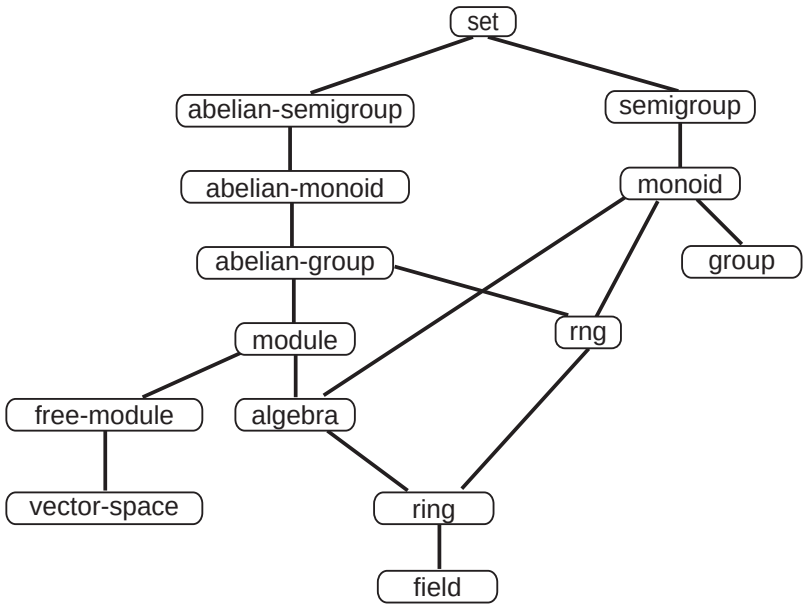

Figure 4.2: Algebraic Domains

As a general rule, Weyl does not provide for implicit coercions of arguments to functions. Thus we assume that in the expression (+ a b) the domains of a and b are the same. If this is not the case, the user must explicitly insert a coercion. The reason for this is to deal with problems such as would arise if, for example, a were $1/2$ and b were x. The domain of a is $\mathbb{Q}$, while the domain of b is $\mathbb{Z}[x]$. The sum cannot be represented in either domain, and in fact we have the choice of embedding it in either $\mathbb{Q}[x]$ or $\mathbb{Z}(x)$. The wrong choice could be lead to very inefficient algorithms. However, we do make one exception. There is assumed to be a canonical homomorphism of the rational integers into every domain. If only one of the arguments to one of the four basic arithmetic operations (+, -, * and /) is an element of $\mathbb{Z}$, then it is automatically coerced into the domain of the other argument.

4.5 Hierarchy of Domains

The root of the domain hierarchy is the class domain. It provides a number of utility routines for managing the set of operations and axioms associated with a domain. However, all algebraic domains are built on the higher level class set.

The basic algebraic domains are given in Figure 4.2. The elements of a domain of class set can be compared using the binary operator weyli::binary=. However, users will find it more convenient to use the macro =, which converts several arguments into a sequence of calls to weyli::binary=. That is

(= a b c) ⟹ (and (weyli::binary= a b)(weyli::binary= b c)).

weyli::binary= x y [*Generic*]

Test to see if x and y are equal. Equality is meant in the mathematical sense, so two vectors are = if their components are =.

4.5.1 Semigroups, Monoids and Groups

This section discusses domains that have one operation. That is, they consist of a set S and a binary operation $\oplus$ such that for two elements a and b in S, $a \oplus b$ is an element of S. The simplest interesting class of such domains is a *semigroup*, where $\oplus$ is assumed to be *associative*. That is, for a, b and c elements of S,

$$(a \oplus b) \oplus c = a \oplus (b \oplus c).$$

In Weyl, domains that are semigroups include the class semigroup. In addition, the operation of a semigroup is assumed to be weyli::times.[4] As with weyli::binary=, there is a macro * that simplifies its use. For instance,

(* a b c) = (weyli::times (weyli::times a b) c).

In addition, repeated multiplication can be indicated by expt. That is,

(expt a 3) = (weyli::times (weyli::times a a) a),

although it may be computed more efficiently for elements of some domains.

* &rest *args*

weyli::times *x y* [*Generic*]

Returns the product of two elements of a semigroup.

If one wants to compute the product of all the elements of a lists, a convenient idiom is to "apply" the "*" function to the list. This is not possible with the Weyl "*" operator since it is a macro, not a function. As a work around for this problem, Weyl also provides a function for multiplication caled "%times." Semantically, this function is the same as the "*" macro, but the code that would be generated if "%times" were used in place of "*" is not as efficient.[5]

%times &rest *args*

Monoid's are semigroups that contain a multiplicative identity. This element is called one and may be accessed by applying the function one to the domain.

one *semigroup* [*Generic*]

Returns the multiplicative identity of monoid.

1? *elt* [*Generic*]

Returns true if its argument is the multiplicative identity and false otherwise.

In addition, the expt function is extended so if its second argument is 0, it returns the multiplicative identity.

The elements of a group have multiplicative inverses which can be determined using recip

recip *elt* [*Generic*]

Returns the multiplicative inverse of *elt*.

[4] This is a limitation of the current implementation of Weyl and will be fixed shortly.

[5] The need for this function can be eliminated by compiler macros, but this solution cannot currently be implemented in a portable manner.

For efficiency reasons it often valuable to have a binary operation that multiplies an element by the multiplicative inverse of the second argument. This operation is called weyli::quotient. The macro / can be used to simplify its use:

$$(/\ \mathsf{a}) = (\mathsf{recip\ a})$$
$$(/\ \mathsf{a\ b}) = (\mathsf{weyli::quotient\ a\ b})$$
$$(/\ \mathsf{a\ b\ c}) = (\mathsf{weyli::quotient\ (weyli::quotient\ a\ b)\ c})$$

In a group, the expt operation is extended to apply to negative second arguments also. In this case

$$(\mathsf{expt\ x}\ -n) = (\mathsf{expt\ (recip\ x)}\ n)$$

expt *a n* [*Generic*]

> Multiplies a by itself n times. If the domain of a just a semigroup, then n must be a positive Lisp integer. When the domain of a is a monoid, then n can be equal to zero. If the domain of a is a group then n can be negative.

A binary operation $\oplus$ on a semigroup S is *commutative* if for every pair of elements in a and b in S, $a \oplus b = b \oplus a$. The domains abelian-semigroup, abelian-monoid and abelian-group are similar to the domains semigroup, monoid and group except the binary operation is plus instead of times. This is the biggest distinction between the domain structure of Weyl and the domains of modern algebra.[6]

+ &rest *args*
%plus &rest *args*

weyli::plus *x y* [*Function*]

> Returns the sum of x and y, which are elements of an abelian-semigroup. weyli::plus is a commutative binary operation.

The multiple argument version of weyli::plus is +, which is what should be used in all applications.

zero *abelian-semigroup* [*Function*]

> Returns the additive identity of an abelian semigroup.

zero? *elt*

0? *elt* [*Function*]

> Returns true if its argument is the additive identity and false otherwise.

The additive inverse of an element of an abelian-group can be obtained using weyli::minus. As in the multiplicative case, there is a binary operation, weyli::difference and a macro - that makes these operations easier to use.

$$(\mathsf{-\ a}) = (\mathsf{weyli::minus\ a})$$
$$(\mathsf{-\ a\ b}) = (\mathsf{weyli::difference\ a\ b})$$
$$(\mathsf{-\ a\ b\ c}) = (\mathsf{weyli::difference\ (weyli::difference\ a\ b)\ c})$$

[6] It would be better if any operation could be declared to be commutative.

$\mathbb{Z}, \mathbb{Q}, \mathbb{R}, \mathbb{C}$	The ring of rational integers, the field of rational numbers, the field of real numbers and the algebraically closed field of complex numbers.
$R[x, y, z]$	The ring of polynomials in x, y and z whose coefficients lie in R.
$R(x, y, z)$	The ring of rational functions (quotients of two polynomials) in x, y and z whose coefficients lie in the ring R.
$R[[t]]$	The ring of univariate power series with non-negative integral exponents in t whose coefficients lie in the ring R.
$R((t))$	The ring of univariate power series with integral exponents in t whose coefficients lie in the ring R.

Figure 4.3: Different Ring Notations

The function "%difference" is provided for use when a multi-argument version of the "-" should be passed to another function. The same caveats about efficiency apply here.

Notice that algebraically we consider all of these domains to possess only one operation weyli::times, or weyli::plus in the abelian case. The other operations are either abbreviations, *e.g.*, expt, or are constructive versions of existential axioms. For instance, a monoid M obeys the axiom: There exists an element e in M such that all for elements a of M, $e \cdot a = a \cdot e = a$. The function one just returns the element e.

4.5.2 Rings

A ring is a triple $(R, +, \times)$ such that $(R, +)$ is an abelian grop and $(R, \times)$ is a monoid. In addition, $\times$ distributes over $+$. That is, for all a, b and c in R

$$a \times (b + c) = (a \times b) + (a \times c) \quad \text{and} \quad (a + b) \times c = (a \times c) + (b \times c).$$

A *commutative ring* is a ring where the operation $\times$ is commutative. In many parts of algebra and geometry the only rings that occur are commutative rings, and more often than not, introductory books contain the phrase "throughout, by ring we mean a commutative ring with identity." In fact, this is what Weyl means by the class ring. Instances of the class rng satisfy only the stricter (broader) definition of a ring.

Examples of rings occur throughout algebra. The table in Figure 4.3 gives some examples of rings and fixes the notation that is commonly used. Additional ring structures are discussed in Chapter 7 and Section 9.4. More baroque structures often arise in commutative algebra. For instance, if S is a subring of the ring R, then the set of polynomials whose constant term lies in S is a ring, $R[S]$, called the ring of *half open polynomials*.

For some rings, when the multiplicative identity element is added to itself a number of times we are left with zero. That is,

$$\overbrace{1 + 1 + \cdots + 1}^{p \text{ times}} = 0.$$

In this case the smallest positive p which has this property is called the *characteristic* of the ring. If the ring is an integral domain, then it is not hard to prove that p must be a prime number. If

there is no such p, then we say the characteristic of the ring is zero. The function characteristic returns the characteristic of a ring.

If all of the non-zero elements of a ring have multiplicative inverses, then the ring is a field.

4.5.3 Modules

A module is an abelian group whose elements can be acted upon by elements of a ring. That is, if M is an A-module then M is an abelian group, and if x is an element of M and a is an element of A, then $a \cdot x$ is also an element of M. Furthermore, for all x and y in M and a and b in A we have

$$a \cdot (x + y) = a \cdot x + a \cdot y, \qquad a \cdot x + b \cdot x = (a + b) \cdot x,$$
$$a \cdot (b \cdot x) = (ab) \cdot x, \qquad 1 \cdot x = x.$$

If this is the case, M is said to be a (left) A-module. Similarly, we can define right A-modules. A ring that is both a left and a right A-module is simply called an A-module. If G is an abelian group then it is trivially a $\mathbb{Z}$-module by the action

$$n \cdot a \mapsto \overbrace{a + a + \cdots + a}^{n \text{ times}},$$

and $(-n) \cdot a \to n \cdot (-a)$ is used to define the action for negative n. The integer 0 sends all elements of M to the additive identity of M.

There are numerous examples of modules in mathematics. A vector space with with real coefficients is a $\mathbb{R}$-module. The elements of an ideal of a ring R form an R-module. In Weyl, all domains that are instances of the class *module* represent modules.[7] If M is an A module, then we call A the *coefficient domain* of M.

coefficient-domain-of *module* [*Function*]

Returns the coefficient domain of *module*.

Note that classes are also defined for free-module and vector-space.[8] These classes and the operations available for manipulating their elements are discussed in more detail in Section 7.2 and Section 8.1.

4.5.4 Properties

There are a variety of different properties that algebraic objects can possess. Many of these properties are important in deciding which algorithms should be used in a computation. However, these properties can sometimes be rather difficult to determine. For instance, an algebraic number ring may or may not be a unique factorization domain. Determining whether it is is quite difficult and we would like to have the option of delaying this determination until it is absolutely necessary.

Although we could have indicated these properties by the class of the domains, and once did, we currently use a different approach. These properties are attached to domains (via their property lists), so that it is not necessary to determine the state of the properties when the domain is created.[9]

The properties that are currently defined are given in Figure 4.4. We have given the predicates that can be used to determine if the property holds of a particular domain.

[7] At this time we do not distinguish between left and right modules.

[8] A *vector space* is a free module over a field.

[9] In fact all the mathematical properties should probably be managed in this way. Taking this approach would compromise some of the modularity in creating some domains, so we haven not taken this step yet.

complete-set?	Does the limit of all Cauchy sequences exist in this domain? In practice this means that floating point numbers can be coerced into this domain.
ordered-domain?	Can $>$ be applied to the elements of this domain
integral-domain?	If true, this domain does not contain any zero divisors.
euclidean-domain?	If true, greatest common divisors can be defined in this domain.
gcd-domain?	If true, not only is this domain euclidean, but a gcd function is actually defined for its elements.
unique-factorization-domain?	If true, the elements of this domain can be factored uniquely.

Figure 4.4: Mathematical Properties

Chapter 5

Scalar Domains

Almost all domains that occur in mathematics can be constructed from the rational integers, $\mathbb{Z}$. For instance, the the rational numbers ($\mathbb{Q}$) are the quotient field of $\mathbb{Z}$, the real numbers are completion of $\mathbb{Q}$ using the valuations at infinity and so on. For performance reasons provides direct implementations of four basic scalar domains, the rational integers ($\mathbb{Z}$), the rational numbers ($\mathbb{Q}$), the real numbers ($\mathbb{R}$) and the complex numbers ($\mathbb{C}$). In addition, a similar direct implementation of finite fields is also provided. These scalar domains are called *numeric domains*, and the corresponding domain hierarchy is given in Figure 5.1. Instances of the GFp class are finite fields of p elements. (These are parameterized domains.) The GFq class is used to implement fields of $q = p^r$ elements. The class GFm is discussed in more detail in Section 5.6.

Instances of the characteristic zero domains may be created using the functions get-rational-integers, get-rational-numbers, get-real-numbers and get-complex-numbers. To create finite fields one can use the function get-finite-field. Depending on its argument, which must be a power of a single prime, this function will return a domain of type GFp or GFq.

Elements of these domains are implemented using the hierarchy of structure types shown in Figure 5.2. The elements of a rational integer domain are all of structure type rational-integer. However, some numeric domains may contain elements of different structure types. For instance, objects of structure type rational-integer as well as rational-number can be elements of the rational number domains ($\mathbb{Q}$). This approach allows us to represent exactly integers and rational numbers in $\mathbb{R}$ and $\mathbb{C}$. In the future, a similar approach will allow us to represent algebraic and transcendental numbers exactly.

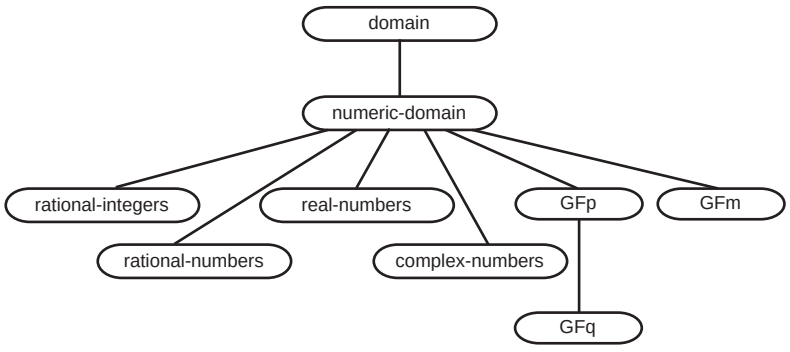

Figure 5.1: Mathematical Scalar Domains

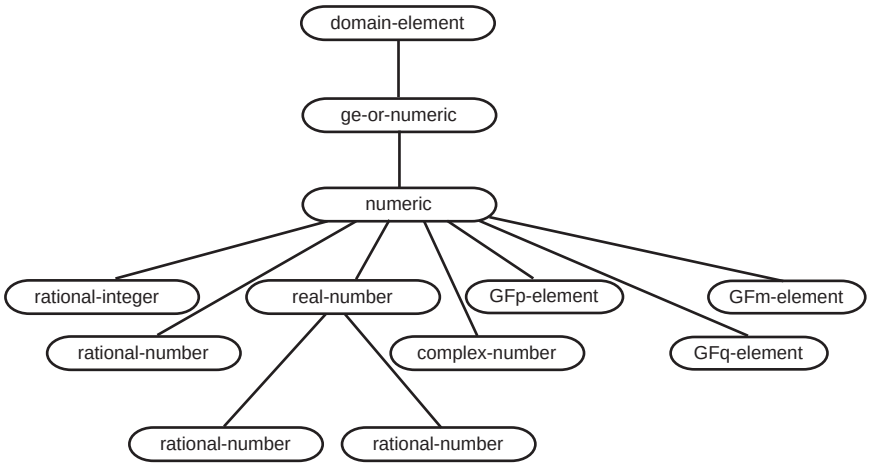

Figure 5.2: Mathematical Scalars

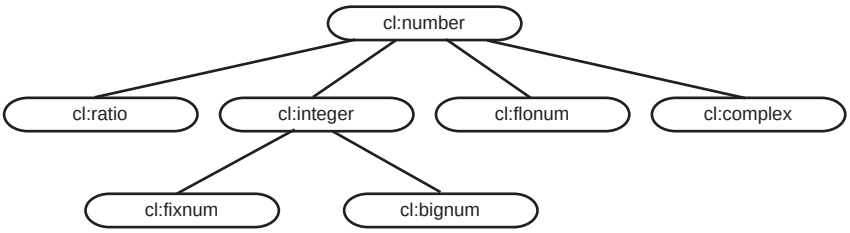

Figure 5.3: Lisp Numbers

One of the complications Weyl must deal with is that Lisp has its own model of the numbers, which must co-exist with Weyl's model. The type structure used by Lisp is given in Figure 5.3. To simplify use of Weyl, we allow users to create Weyl numbers from Lisp numbers, and to incorporate Lisp numbers into their code by providing some automatic coercions. Thus adding a Lisp number to an element of $\mathbb{R}$ causes the Lisp number to be coerced to an element of $\mathbb{R}$ of structure rational-integer.

There are several different ways to determine if an object is a scalar.

(typep *obj* 'cl:number)	a Lisp number,
(typep *obj* 'numeric)	a Weyl number,
(number? *obj*)	either a Lisp number or a Weyl number.

The following sections are organized by the different structure types. Sections 5.1 through 5.6 deal with rational integers, rational numbers, real numbers, complex numbers and elements of finite fields respectively. In each section we discuss the different domains these structure elements can be used in followed by a discussion of the operations that can be performed with each structure type.

5.1 Rational Integers

The rational integers are the integers of elementary arithmetic:

$$\mathbb{Z} = \{\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots\}.$$

Other than limitations on the memory of the host computer, there is no limitation on the size of the elements of $\mathbb{Z}$. The term *rational* integer is used to distinguish this domain from other domains of *algebraic* integers, *e.g.*, $\mathbb{Z}[(1+\sqrt{5})/2]$.

A domain of rational integers can be created using the following function.

get-rational-integers [*Function*]

Returns a domain that is isomorphic to the rational integers, $\mathbb{Z}$. When called repeatedly, it always returns the same value until reset-domains is called.

Most of the time, there only needs to be one rational integer domain. The domain of rational integers is a euclidean domain.

Elements of structure type rational-integer can be elements of domains of type rational-numbers, real-numbers and complex-numbers. Equivalently, if *domain* is a domain that admits elements of structure type rational-integer, then one invoke make-element with *domain* and a Lisp integer.

Rational integers are most easily created by coercing a Lisp integer to a rational integer domain using the function coerce. Furthermore, the usual arithmetic routines (+, -, *, / and expt) work with rational integers that are elements of the same domain. If the domain is a field then / may return a rational-number.

For instance, the following routine could be used to compute factorials.

```
(setq *Z* (get-rational-integers))

(defun factorial (n)
  (if (< n 2) (one *Z*)
      (* (coerce n *Z*) (factorial (- n 1)))))
```

Notice that the unit element of *Z* was created by using the function one, rather than (coerce 1 *Z*). In general, this is more efficient.

One of the more commonly used control structures is that used to construct exponentiation from multiplication by repeated squaring. This control structure is captured by the internal function weyli::repeated-squaring.

weyli::repeated-squaring *mult one* [*Function*]

Returns a function of two arguments that is effectively

```
(lambda (base exp)
  (declare (integer exp))
  (expt base exp))
```

except that the body does the exponentiating by repeated squaring using the operation *mult*. If exp is 1, then *one* is returned.

Using this function, one could have defined exponentiation as

```
(defun expt (x n)
  (funcall (weyli::repeated-squaring #'times (coerce 1 (domain-of x)))
           x n))
```

However, this routine can be used for operations other than exponentiation. For instance, if one wanted a routine that replicates a sequence n times, one could use the following:

```
(defun replicate-sequence (x n)
  (funcall (weyli::repeated-squaring #'append ())
           x n))
```

isqrt n [*Function*]

Returns the integer part of the square root of n.

integer-nth-root m n [*Function*]

Computes the largest integer not greater than the n-th root of m.

power-of? *number* &optional *base* [*Function*]

Returns *base* and k if *number* $=$ *base*^{k}, otherwise it returns nil. If *base* is not provided returns the smallest integer of which *number* is a perfect power.

factor n [*Function*]

Factors n into irreducible factors. The value returned is a list of dotted pairs. The first component of the dotted pair is the divisor and the second is the number of times the divisor divides n. The type of factorization method used, can be controlled by setting the variable *factor-method*. The allowable values are simple-integer-factor and fermat-integer-factor.

prime? n [*Function*]

Returns true if n is a prime number. (For other domains, if n has no factors that are not units.)

totient n [*Function*]

Returns the Euler totient function of n, the number of positive integers less than n that are relatively prime to n, *i.e.*:

$$\mathsf{totient}(n) = n \prod_p \left(1 - \frac{1}{p}\right),$$

where product is over all prime divisors of n.

factorial n [*Function*]

Computes $n!$

pochhammer n k [*Function*]

Computes the Pochhammer function of n and k, which is closely related to the factorial:

$$\mathsf{pochhammer}(n,k) = (n)_k = n \cdot (n+1) \cdot (n+2) \cdots (n+k-1).$$

combinations n m [*Function*]

Computes the number of combinations of n things taken m at a time.

$$\mathsf{combinations}(n, m) = \binom{n}{m} = \frac{n!}{m!\,(n-m)!}.$$

newprime n [*Function*]

Returns the largest prime less than its argument.

(Need to point out that the elements of the second rational integer domain created are totally different from that those that are elements of the first instance of the rational integers.)

5.2 Rational Numbers

The domain rational numbers, $\mathbb{Q}$, is the quotient field of the ring of rational integers. The elements of a rational number domain can have structure type either rational-integer or rational-number. Elements

As in Common Lisp there is a set of four functions for truncating numbers and ratios to integers. If the second argument is not provided then it defaults to 1. If only the first argument is provided and it is a rational integer, then all four functions return the same values.

A domain of rational integers is created by the following function.

get-rational-numbers [*Function*]

Returns a domain that is isomorphic to the rational numbers, $\mathbb{Q}$. When called repeatedly, it always returns the same value until reset-domains is called.

`floor` *number* &optional *divisor*
`ceiling` *number* &optional *divisor*
`truncate` *number* &optional *divisor*
`round` *number* &optional *divisor*

5.3 Real Numbers

The entire real number situation is somewhat confused. In particular, the relationship between floating point numbers and real numbers is jumbled. These issues will be fixed at a later date.

get-real-numbers &optional *precision* [*Function*]

This returns a domain whose elements are floating point numbers. If *precision* is not specified, then the machines default double precision floating point numbers will be used. If *precision* is specified, then a special arbitrary precision floating point package will be used. Operations with these numbers will be somewhat slower (and will cause more garbage collection) than when using the machine's floating point data types.

floor *number* &optional *divisor* [*Function*]

Computes the floor of *number*.

ceiling *number* &optional *divisor* [*Function*]

Computes the ceiling of *number*.

truncate *number* &optional *divisor* [*Function*]

Computes the truncate of *number*.

round *number* &optional *divisor* [*Function*]

Computes the round of *number*.

sqrt n [*Function*]

For positive n returns positive $\sqrt{n}$ with the same precision as n.

The following standard trigonometric and hyperbolic routines are provided

sin n	asin n	sinh n	asinh n
cos n	acos n	cosh n	acosh n
tan n	atan n	tanh n	atanh n

exp n [*Function*]

Returns e^n.

log n &optional b [*Function*]

For positive n returns the principal part of $\log_b n$. If b is not supplied then e, the base of natural logarithms, is used for b.

5.4 Complex Numbers

Not yet implemented.

realpart z [*Function*]

If $z = x + iy$ returns x.

imagpart z [*Function*]

If $z = x + iy$ returns y.

conjugate z [*Function*]

If $z = x + iy$ returns $x - iy$.

abs z [*Function*]

If $z = x + iy$ returns $|z| = \sqrt{x^2 + y^2}$.

phase z [*Function*]

If $z = re^{i\theta}$ returns θ. r is the absolute value of z.

5.5 Quaternions

Quaternions are a non-commutative algebra over a field, usually the reals, that are often used to represent three dimensional rotations. Weyl can construct a quaternion algebra over any field, F. This algebra is a four dimensional vector space over F with the following relations. The element $\langle 1,0,0,0\rangle$ is the multiplicative identity. If we denote $i = \langle 0,1,0,0\rangle$, $j = \langle 0,0,1,0\rangle$ and $k = \langle 0,0,0,1\rangle$, then

$$i^2 = j^2 = k^2 = -1, \quad ij = -ji, \quad jk = -kj \quad \text{and} \quad ik = -ki.$$

get-quaternion-domain *field* [*Function*]

Gets a quaternion algebra over *field*, which must be a field.

Quaternions can be created using make-element.

make-element *quaternion-algebra* [*Function*]

v_1 v_2 v_3 v_4 Creates an element of quaternion-algebra from its arguments. The value returned will be $v_1 + i \cdot v_2 + j \cdot v_3 + k \cdot v_4$. As with other versions of make-element, the function weyli::make-element assumes the arguments are all elements of the coefficient domain and is intended only for internal use.

As an algebraic extension of the real numbers, the quaternions are a little strange. The subfield of quaternions generated by 1 and i, is isomorphic to the complex numbers. Adding j and k makes the algebra non-commutative and causes it to violate some basic intuitions. For instance, -1 has at least three square roots!

We illustrate some of these issues computationally. First we create a quaternion algebra in which to work.

```
> (setq q (get-quaternion-domain (get-real-numbers)))
Quat(R)
```

Next, we can create some elements of the quaternions and do some simple calculations with them.

```
> (setq a (make-element q 1 1 1 1))
<1, 1, 1, 1>

> (setq b (/ a 2))
<1/2, 1/2, 1/2, 1/2>

> (* b b b)
<-1, 0, 0, 0>
```

As expected, one can multiply quaternions by other quaternions and by elements of the coefficient field (or objects that can be coerced into the coefficient field).

conjugate *quaternion* [*Function*]

This is an extension of the concept of complex conjugation. It negates the coefficients of i, j and k. This is illustrated by the following example.

```
> (setq c (make-element q 1 2 3 4))
<1, 2, 3, 4>

> (conjugate c)
<1, -2, -3, -4>

> (* c (conjugate c)
<30, 0, 0, 0>
```

Notice that the components of the product of a quaternion with its conjugate are all zero except for the very first component. This matches what happens when one multiplies a complex number with its complex conjugate.

5.6 Finite Fields

The usual finite fields are provided in Weyl, $\mathbb{F}_p$ and algebraic extensions of $\mathbb{F}_q$. Such domains are called *GFp* domains. Since all finite fields with the same number of elements are isomorphic, fields are created by specifying the elements in the field.

get-finite-field *size* [*Function*]

Size is expected to be a a power of a prime number. This function returns a finite field with the indicated number of elements. If *size* is nil then a *GFm* field is returned.

number-of-elements *finite-field* [*Function*]

Returns the number of elements in *finite-field*.

At the moment Weyl can only deal with the fields $\mathbb{F}_{2^k}$ and F_p. For instance,

```
> (setq F256 (get-finite-field 256))
GF(2^8)

> (characteristic F256)
2

> (number-of-elements F256)
256
```

Elements of a GFp are created by coercing a rational integer into a GFp domain. For finite fields with characteristic greater than 2, coercing an integer into F_p maps n into $n \pmod p$. For F_{2^k}, the image of an integer is a bit more complicated. Let the binary representation of n be

$$n = n_\ell n_{\ell-1} \cdots n_2 n_1 n_{0_2},$$

and let α be the primitive element of F_{2^k} over F_2. Then

$$n \longrightarrow n_{k-1}\alpha^{k-1} + n_{n-2}\alpha^{k-2} + \cdots + n_1\alpha + n_0.$$

This mapping is particularly appropriate for problems in coding theory.

In addition, elements of finite fields can be created using make-element.

make-element *finite-field integer* &optional *rest* [*Function*]

Creates an element of *finite-field* from *integer*. This is the only way to create elements of $\mathbb{F}_{p^k}$. (As with all make-element methods, the argument list includes &rest arguments, but for finite fields any additional arguments are ignored.)

As an example of the use of finite fields, consider the following function, which determines the order of an element of a finite field (the hard way).

```
(defun element-order (n)
  (let* ((domain (domain-of n))
         (one (coerce 1 domain)))
    (loop for i upfrom 1 below (number-of-elements domain)
          for power = n then (* n power)
        do (when (= power one)
             (return i)))))
```

A more efficient routine is provided by Weyl as multiplicative-order.

multiplicative-order *elt* [*Function*]

Elt must be an element of a finite field. This routine computes multiplicative order of *elt*. This routine requires factoring the size of the multiplicative group of the finite field and thus is appropriate for very large finite fields.

The following illustrates use of these routines.

```
> (element-order (coerce 5 (get-finite-field 41)))
20

> (multiplicative-order (coerce 5 (get-finite-field 41)))
```

Consider what is involved when implementing an algorithm using the Chinese remainder theorem. The computation is done in a number of domains like $\mathbb{Z}/(p_1)$, $\mathbb{Z}/(p_2)$ and $\mathbb{Z}/(p_3)$. The results are then combined to produce results in the domains $\mathbb{Z}/(p_1p_2)$ and $\mathbb{Z}/(p_1p_2p_3)$. Rather than working in several different domains and explicitly coercing the elements from one to another, it is easier to assume we are working in a single domain that is the union of $\mathbb{Z}/(m)$ for all integers m and marking the elements of this domain with their moduli. We call this domain a GFm.

GFm domains are also created using the get-finite-field but by providing `nil` as the number of elements in the field.

Elements of GFm are printed by indicating their modulus in a subscript surrounded by parentheses. Thus $2_{(5)}$ means 2 modulo 5. Combining two elements $a_{(m)}$ and $b_{(m)}$ that have the same moduli is the same as if they were both elements of $\mathbb{Z}/(m)$. To combine elements of two different rings, we find a ring that contains both as subrings and perform the calculation there. Thus combining $a_{(m)}$ and $b_{(n)}$ we combine the images of a and b as elements of $\mathbb{Z}/(\gcd(m,n))$.

FIXTHIS: Need works something out for dealing with completions of the integers at primes, and how we are going to compute with elements.

Chapter 6

General Expressions

Most sophisticated computations using Weyl take place within one of the more specialized domains discussed in the later chapters. Generally, these domains deal with algebraic structures like groups, rings and fields. Unfortunately, none of these structures can deal with all types of mathematical objects. For instance, while vectors can be elements of vector spaces, there is no algebraic domain that contains both vectors and polynomial. Similarly, there is no algebraic domain that can deal with special functions, summations, products etc. What is required is a domain that can represent mathematical objects *syntactically*. This is dealt with in Weyl by the general-expression domain.

General expressions are intended to be flexible enough to represent any mathematical expression that one might write on a piece of paper. On paper one can write the two distinct expressions $a(b+c)$ and $ab+ac$, even though as polynomials they are equivalent. Within the polynomial domain discussed in Chapter 9, these two expressions are indistinguishable. However, as general expressions they are distinct objects that can be differentiated.

While, general expressions are extremely flexible, they are not necessarily very efficient. As a consequence, for large scale computations it is usually preferable to use one of the special purpose domains described in the later sections. While some computations are purely algebraic and need never reference general expressions, most engineering and scientific computations need a place to store information about the dimensions of variables, defining relationships and so on. These purposes are also served well by the general expression structures.

6.1 Class Structure

The general expression domain is an instance of the class general-expression. An instance of this class is always bound to the variable *general*. The class structure of the general expression class is shown in Figure 6.1.

The class non-strict-domain is used as a special indicator that any operation that can syntactically be applied to arguments of this domain should be allowed to proceed. This matches the view

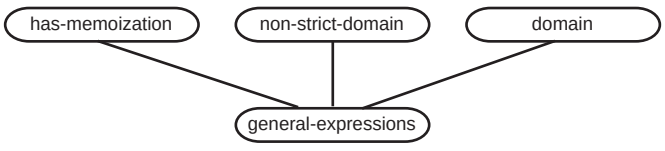

Figure 6.1: General Expressions

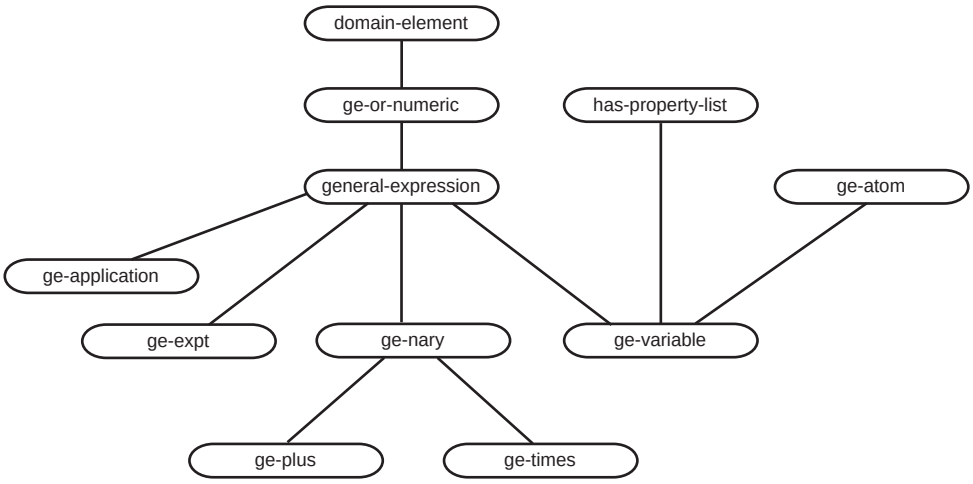

Figure 6.2: Classes of General Expression Elements

that general expressions are syntactic objects, and do not adhere to any strict algebraic structure. The has-memoization class provides the data structures required to support memoizing code (see the definition of memoize in Section 3.2).

The general expression domain is initialized by the function reset-domains, which creates a new general expression domain and deletes all references to other domains in Weyl.

The class structure of elements of the general expression domains are shown in Figure 6.2.

6.2 Variables

Variables are have the following form

ge-variable? *x* [*Function*]

Returns T if *x* is a general expression that is a variable.

add-subscripts *var* &rest *subscripts* [*Function*]

Creates a new variable, which has the subscripts indicated. If the variable already has subscripts, then the new subscripts are appended to the ones already present.

All variables have property lists which can be used to store important information about them. This information is not attached to the variables, but to the domain from which the variables arise.

get-variable-property *domain var key* [*Function*]

Returns the *key* property of *var*. A return value of nil indicates that the was no *key* property associated with *var*. New properties may be established by using setf on this form.

Dependencies of one variable on another can be declared using declare-dependencies:

declare-dependencies *kernel* &rest *vars* [*Function*]

This indicates that *kernel* depends upon each of the variables in vars.

depends-on? *exp* &rest *vars* [*Function*]

This predicate can be applied to any expression, not just to variables. It returns t if the *exp* depends on *all* of the variables in *vars*, otherwise it returns nil. The expression can also be a list, in which case nil is returned only if every element of *exp* is free of *vars*.

different-kernels *exp list-of-kernels* [*Function*]

Returns a list of the kernels in *exp* that are different from those in *list-of-kernels*.

6.3 Numbers

Should say something intelligent here. In particular need to deal with arbitrary precision floating point numbers as well as the IEEE floating point standards. At the moment, we just accept floating point numbers however they are printed out.

6.4 Operators

The basic boolean operations are supported:

$a+b+c$	`(+ a b c)`
$a-b$	`(+ a b)`
$-a$	`(- a)`
ab	`(* a b)`
a/b	`(/ a b)`
a^b	`(expt a b)`

In addition the standard special functions are supported log sin cos tan cot sec csc sinh cosh tanh coth sech csch

6.5 Tools for General Expressions

Weyl provides a number of programming tools that make writing symbolic programs simpler. This section discusses some of these tools.

6.5.1 Display tools

print-object *exp stream* [*Function*]

This method is provided for all CLOS instances. It is used whenever an object is printed using princ or a related function. In Weyl, a print-object method is provided for classes of objects to make the objects more readable when debugging or when doing simple computations. The printed form produced by print-object cannot be read to produce the object again (as can be done with lists and some other Lisp expressions.

display *expr* &optional *(stream* *standard-output**) [*Function*]

Prints the expression *expr* onto *stream*. If stream a graphics stream then a two dimension display will be used (not yet implemented), otherwise some textual display will be used.

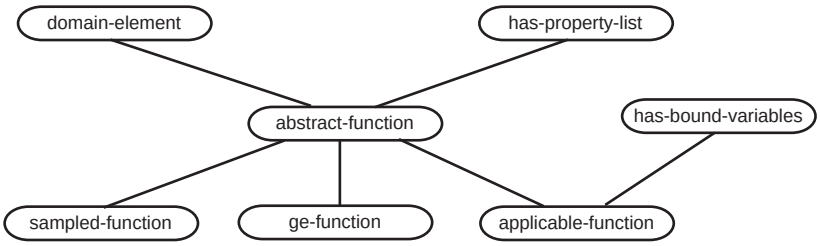

Figure 6.3: Function Classes

6.5.2 Simplification Tools

simplify *expr* [*Function*]

Performs simple simplifications of *expr*, $0 + x \rightarrow x$ and so on.

expand *exp* [*Function*]

Replaces all products of sums in *exp* by sums of products.

ge-equal *x y* [*Function*]

Returns T if *x* and *y* are syntactically identical general expressions.

ge-great *x y* [*Function*]

To speed up operations like simplification of expressions, an order is placed on all expressions in the general representation. This ordering is provided by the function ge-great.

def-ge-operator *operator* &rest *keyword-expr-pairs* [*Function*]

When a new operator is introduced this definer should be used. It allows one to define the simplifier, display and equality tester functions.

deriv *exp var* &rest *vars* [*Function*]

Computes the derivative of *exp* with respect to *var* and simplifies the results. This is done for *var* and each element of *var*. Thus the second derivative of *exp* with respect to T could be computed by: (deriv *exp* 't 't).

6.6 Functions

There are three different types of functions as illustrated in Figure 6.3, sampled-functions, ge-functions and applicable-functions. Each of these represent functions about which different aspects are known. To represent "well-known" functions, like sine, cosine and f (in the expressions $f(x)$), we use instances of the class ge-function. We are given only these functions names. Sometimes we know more about the functions, like their derivative or their expansion as a power series. This additional information is placed on their property lists.

applicable-functions are functions for which we have a program for computing their values. They are essentially λ-expressions. Finally, sampled-functions are functions about which we know only their graph. That is, we are given the values of the function at certain points and mechanisms for interpolating those values. Each of these types of functions are described in more detail in the following sections.

Each class that inherits from abstract-function includes a slot that indicates the number of arguments an instance of this class (*i.e.*, functions) accepts. This information can be accessed using the method nargs-of.

nargs-of *abstract-function* [*Function*]

Returns the number of arguments accepted by abstract-function.

Each of these types of functions can be applied to arguments to get the value of the function at that point. This can be done by either of the following two functions, which are extensions of the usual Lisp ones (and continue to work with Lisp arguments).

funcall *fun* arg_1 arg_2 ... arg_n [*Function*]

Apply fun to the specified arguments. If the number of arguments provided does not match the number of arguments of the function, then an error is signaled.

apply *fun* arg_1 arg_2 ... arg_n *list* [*Function*]

Apply *fun* to the k arguments specified and the elements of *list*. If the number of arguments of the function differ from k plus the length of *list* then an error is signaled.

6.6.1 GE Functions and Applications

The most commonly used type of function in Weyl is ge-function. These are functions with "well-known " names. The easiest way to generate examples of ge-functions is to invoke the Lisp function with the same name. For instance,

```
> (setq appl (sin 'x))
sin(x)
```

This expression $\sin x$ is not a ge-function, but a ge-application. It consists of two pieces, a function and an argument list. These pieces can be extracted using the functions funct-of and args-of.

funct-of *application* [*Function*]

Returns the ge-function in the functional position of *application*.

args-of *application* [*Function*]

Returns a list of the arguments of *application*.

For instance, using appl created above, we have

```
> (funct-of appl)
sin

> (args-of appl)
(x)
```

The ge-function can be used with funcall and apply to create new applications.

```
> (funcall (funct-of appl) (expt 'y 2))
sin(y^2)

> (apply (funct-of appl) (list 'y))
sin(y)
```

Like all classes that inherit from ge-function, instances of abstract-function handle the nargs-of method.

```
> (nargs-of (funct-of appl))
1
```

In addition to information about the number of arguments accepted by a function, we can also indicate the function's derivative. This is done using the following function:

declare-derivative *func args* &body *body* [*Function*]

This form is used to define the derivative of *func*. The body of this form can do any computation it wants on the arguments of *func*. For instance, the derivative of sin and cosine are defined as follows:

```
(declare-derivative sin (x) var
  (* (deriv x var) (cos x)))

(declare-derivative cos (x) var
  (* (- (deriv x var)) (sin x)))
```

With these definitions, which are already in Weyl, derivatives of general expressions involving sine and cosine can be performed:

```
> (deriv (* (sin 'x) (cos 'x)) 'x)
(cos(x))^2 - (sin(x))^2
```

6.6.2 Applicable functions

Applicable functions are a mechanism to enable manipulation of anonymous functions. In languages like Scheme and Common Lisp they correspond to λ-expressions. One of the useful features of Weyl is that one can perform arithmetic operations on functions. This often forces us to produce applicable functions. For instance,

```
> (+ (funct-of (sin 'x)) (funct-of (cos 'x)))
(lambda (v.1) sin(v.1) + cos(v.1))
```

To explicitly create an applicable function, the function make-applicable-function is used.

make-applicable-function *args body* [*Function*]

Create an applicable function. The variables used in *args* are replaced by newly generated variables, and the uses of the variables in *body* are similarly replaced.

Like other functions, one can differentiate applicable functions.

deriv *function integer1 integer2* ... [*Function*]

Returns the function obtained by taking the positional derivative of function with respect to the given positions.

For example,

```
> (setq f1 (make-applicable-function '(x y) (+ (* 'x 'y) (* 'y 'y))))
(lambda (v.1 v.2) v.1 v.2 + v.2^2)

> (deriv f1 1)
(lambda (v.1 v.2) v.2)

> (deriv f1 2)
(lambda (v.1 v.2) v.1 + 2 v.2)

> (deriv f1 2 1)
(lambda (v.1 v.2) 1)
```

A more significant example of the use of the applicable functions arises in the specification of the Lagrangian in mechanics. Recall that the Lagrangian of a mechanical system is defined as the difference of the kinetic and potential energies of the system. So for a point mass constrained to one dimension, we have a Lagrangian of

$$L = KE + PE = \frac{m\dot{x}^2}{2} - gx,$$

where x is the position of the point, m is the mass of the point and g is the acceleration due to gravity.

The Lagrangian can be viewed as a map from phase space to action, where each point of phase space specifies a position x and a velocity. Alternatively, it can be viewed as a map from the space of all possible paths of the particle to functions from time to action. This second interpretation is the most appropriate one for the calculus of variations and the Lagrangian formulation of mechanics. In Weyl, a single function will suffice:

```
> (defun point-lagrangian (x &optional (xdot (deriv x 1)))
    (- (* 1/2 'm xdot xdot) (* 'g x)))
```

If both the position and velocity are specified, the result is a simple computation of the action of the particle:

```
> (point-lagrangian 1 2)
2m - g
```

However, one can also pass the function the path that the particle takes:

```
> (setq action (point-lagrangian
                 (make-applicable-function '(t) (+ 't (sin 't)))))
(lambda (v.1) -1 v.1 g - ((sin(v.1)) g) + 1/2 (cos(v.1))^2 m
   + 1/2 (cos(v.1)) m + 1/2 m (cos(v.1)) + 1/2 m)

> (funcall a 0)
2m

> (funcall a 1)
1/2 m (cos(1)) + 1/2 m - g - ((sin(1)) g) + 1/2 (cos(1))^2 m + 1/2 (cos(1) m
```

6.6.3 Sampled Functions

Chapter 7

Sums, Products and Quotients of Domains

Let $\mathfrak{A}$ be category of domains, *e.g.*, groups, rings, fields, etc. Given two elements of $\mathfrak{A}$, say A and B, there are often ways of combining them to produce another element of $\mathfrak{A}$. This chapter discusses several of those techniques.

7.1 Direct Sums

Given two domains A and B, their direct sum can be viewed as the set of all pairs of elements in A and B, where arithmetic operations are performed component-wise. We denote the direct sum of A and B by $A \oplus B$. If A, B and C are domains, then

$$A \oplus (B \oplus C) = (A \oplus B) \oplus C = A \oplus B \oplus C,$$

so multiple direct sums are well defined. Direct sums are created using the function get-direct-sum:

get-direct-sum domain$_1$ ...domain$_n$ [*Function*]

> Creates a domain that is the direct sum of $domain_1, \ldots, domain_n$. If any of the $domain_i$ are themselves direct sums, then their components are included as if they had been explicitly specified.

The algebraic structure of the object returned by get-direct-sum depends upon the type of the objects of which it is composed. The following table illustrates the options.

A	B	$A \oplus B$
Group	Group	Group
Ring	Ring	Ring
Field	Field	Ring
Group	Ring	Group
Abelian	Abelian	Abelian

The dimension of a direct sum is the number of its components:

dimension *direct-sum* [*Function*]

> Returns the number of components of the direct sum.

To illustrate this the direct sum we can compute the direct sum of the real numbers, the rational numbers and the rational integers as follows:

```
> (setq direct-sum (get-direct-sum (get-real-numbers)
                                   (get-rational-numbers)
                                   (get-rational-integers))
R (+) Q (+) Z
```

The symbol used to indicate a direct sum when only ASCII characters are available is (+). This domain has dimension 3 and it is a ring as one might expect:

```
> (dimension direct-sum)
3

> (typep direct-sum 'ring)
T

> (typep direct-sum 'field)
NIL
```

Even though some of the components of the direct sum domain are fields, the direct sum itself is a ring. Individual elements of the direct sum can be extracted using the routine ref:

```
> (ref direct-sum 0)
R

> (ref direct-sum 2)
Z
```

ref *direct-sum i* [*Function*]

Returns the *i*-th element (zero based) of *direct-sum*. *direct-sum* can either be a direct sum domain or an element of a direct sum domain.

Elements of a direct sum domains are created using the following function:

make-element *direct-sum* element$_1$ ... element$_n$ [*Function*]

Creates an element of *direct-sum*. The number of elements provided must match the dimension of *direct-sum*. Each of the elements is coerced into the domain of their associated component of the *direct-sum* before the element is created.

The following illustrates the use of elements of a direct sum.

```
> (setq x (make-element direct-sum 1 2 3))
1 (+) 2 (+) 3

> (+ x x)
2 (+) 4 (+) 6

> (* 3 x)
3 (+) 6 (+) 9
```

As with direct sum domains, the dimension of an element of a direct sum domain can be computed using dimension, and individual components can be determined using ref.

```
> (dimension x)
3

> (loop for i below (dimension x)
        do (format t "~%Component ~D: ~S, domain: ~S"
                   i (ref x i) (domain-of (ref x i))))
Component 0: 1, domain R
Component 1: 2, domain Q
Component 2: 3, domain Z
```

7.2 Free Modules

M is a free R-module if M is both a free abelian group and an R-module. In Weyl, elements of free modules are represented as n-tuples of elements of the coefficient domain R. Thus, we are only able to deal with finite dimensional free modules. If n is the rank of M as a free abelian group, then as a free R-module, M is isomorphic to the direct sum of n copies of R. Closely related to the concept of a free module is that of a vector space. A vector space is a free module whose coefficient domain is a field. Additional information about operations on elements of vector spaces can be found in Section 8.1.

The basic routine for creating a free module is get-free-module.

get-free-module *domain rank* [*Function*]

Creates a free module of dimension *rank* where the elements' components are all elements of *domain*. *domain* must be a ring. If *domain* is a field the domain returned will be a vector space.

If one expects the coefficient domain to be a field, and thus the affine module will actually be a vector space, then the routine get-vector-space should be used instead of get-free-module. This routine explicitly checks that the coefficient domain is a field and signals an error if it is not a field.

Once a free R-module has been created, it is often useful to refer to the domain R itself. This can be done using the routine coefficient-domain-of. The dimension of the free module can be obtained using dimension.

coefficient-domain-of *domain* [*Function*]

Returns the domain of he coefficients of *domain*.

dimension *domain* [*Function*]

This method is defined for free modules but the value returned is not specified. (Actually it should be infinity.)

Elements of a free module can be created using the function keylispmake-element. The routine make-point calls dimensions, so that code that uses free modules as vector spaces can be written more euphoniously.

make-element *domain value* &rest *values* [*Function*]

Make an element of the module *domain*, whose first component is *value*, etc. If value is a Lisp vector or one dimensional array, then elements of that array are used as the components of the free module element.

Use of these routines is illustrated below [Add something here –RZ]

ref *vector i* [*Function*]

Returns the i-th element (zero based) of *vector*.

inner-product u v [*Function*]

Computes the inner (dot product) of the two vectors u and v. If $u = \langle u_1, \ldots u_k \rangle$ and $v = \langle v_1, \ldots v_k \rangle$ then

$$(\text{dot-product } u\ v) = u_1 v_1 + u_2 v_2 + \cdots + u_k v_k.$$

7.3 Tensor Products

7.4 Rings of Fractions

Given an arbitrary ring R, we can construct a new ring, called the *quotient ring* of R, whose elements are pairs of elements in R subject to the following equivalence relation. If (a, b) and (c, d) are elements of a quotient ring, then they are equal if and only if $ad = bc$. The sum and product of two elements of the quotient ring are defined as follows

$$(a, b) + (c, d) = (ad + bc, bd),$$
$$(a, b) \times (c, d) = (ac, bd).$$

If the ring R is an integral domain then the quotient ring is actually a field.

More generally, let S be a multiplicatively closed subset of R. The localization of R with respect to a multiplicative subset of R, S, written $S^{-1}R$, is the ring of pairs (a, s), where a is an element of R and s is an element of S.

make-ring-of-fractions *domain* &optional *(multiplicative-set domain)* [*Function*]

Constructs a quotient ring from *domain*. If a second argument is provided, then this make-quotient-ring returns a ring representing the localization of domain with respect to *multiplicative-set*. Otherwise, the quotient ring of domain is returned. If *domain* is an integral domain, then the ring returned will be a field.

make-quotient-field *domain* [*Function*]

This generic function constructs the quotient field of *domain*. The domain returned by this operation will be a field. If *domain* is itself a field, then it will be returned without any modification. If *domain* is a gcd domain then operations with the elements of the resulting quotient field will reduce their answers to lowest terms by dividing out the common gcd of the resulting numerator and divisor.

Two special cases are handled specially by make-quotient. If the argument domain is either the rational integers or a polynomial ring then special domains of the rational numbers or rational functions are used. This is for efficiency reasons. Hopefully, a more general solution can be found in the near future.

Two operations can be used to create elements of a quotient field: make-quotient-element and quotient-reduce. The operation make-quotient-element creates a quotient element from the numerator and denominator domains. Quotient-reduce does the same thing but removes the common GCD from the numerator and denominator first.

(Ed: Need to think about things like localizations here.)

get-quotient-field *ring* [*Function*]

Returns a field which is the quotient field of *ring*. In some cases, this is special cased to return return something more efficient than the general quotient field objects.

weyli::qf-ring *qf* [*Function*]

Returns the ring from which the quotient field *qf* was built.

numerator *q* [*Function*]

Returns the numerator of *q*.

denominator *q* [*Function*]

Returns the denominator of *q*.

quotient-reduce *domain numerator* &optional *denominator* [*Function*]

Numerator and *denominator* are assumed to be elements of base ring of *domain*. quotient-reduce creates and quotient element in *domain* from *numerator* and *denominator*. If *denominator* is not provided, the multiplicative unit of *domain* is used.

with-numerator-and-denominator *(num den) q* &body *body* [*Function*]

Creates a new lexical environment where the variables *num* and *den* are bound to the numerator and denominator of *q*. Each

7.5 Factor Domains

Let B be a subgroup of A. One can divide the elements of A into equivalence classes as follows: Two elements of A are in the same equivalence class if their quotient is an element of B. The equivalence classes of A with respect to B form a group, called the *factor group* of A by B. Using this construction one can form factor modules of one module by a submodule. If A is a ring and B is an ideal of A, then the factor module of A by B is a ring, called the *factor ring* of A by B.

Weyl provides four classes for dealing with factor domains, as shown in Figure 7.1. The factor domain (group, module or ring) of A and B is written A/B. The components of a factor domain may be accessed using the following routines.

factor-numer-of *factor-domain* [*Function*]

Returns the "numerator" of a factor domain.

factor-denom-of *factor-domain* [*Function*]

Returns the "denominator of a factor domain.

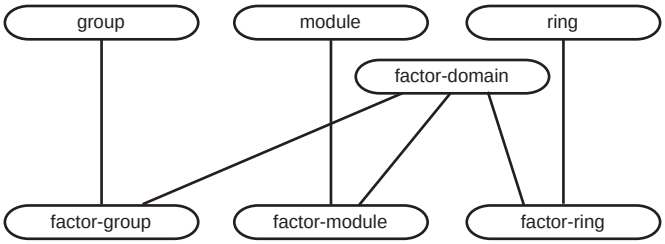

Figure 7.1: Factor Domains

Chapter 8

Linear Spaces

The general approach in Weyl is to use functors to create more complex domains from simple domains. The simplest such functors are those that takes a ring or field R and produces a vector space whose elements lie in R. This chapter discusses these domains and the operations that apply to their elements. The simplest of these domains are free modules and vector spaces. Both are modules (you can add their elements and multiply them by elements of R). The difference is that elements of a vector space can also be divided by elements of R, *i.e.*, R is a field. Free modules and vector spaces are discussed in Section 7.2 and Section 8.1 respectively.

8.1 Vector Spaces

Vector spaces are free modules where the coefficient domain is a field.

get-vector-space *domain dim* [*Function*]

Create a vector space of dimension *dim* where the vectors' components are all elements of *domain*. *Domain* must be a field.

8.2 General Matrices

get-matrix-space *ring* [*Function*]

Returns a general matrix domain, where the entries in the matrices are elements of *ring*.

make-element *domain value* &rest *values* [*Function*]

Make an element of the module *domain*, where the first component is *value*, etc.

The following two functions return square matrices whose off diagonal elements are all zero. Its diagonal elements are 0 for zero-matrix and 1 for one-matrix.

`zero-matrix` *matrix-domain* &optional *rank*
`one-matrix` *matrix-domain* &optional *rank*

matrix-dimensions *matrix* [*Function*]

Returns the dimensions of this particular matrix.

ref *vector i j* [*Function*]

Returns the (i, j) element (zero based) of *vector*. If i or j is :*, then a row or column vector is returned.

with-matrix-dimensions *(dim1 dim2* &optional *array) matrix* &body *body* [*Function*]

This control abstraction binds *dim1* and *dim2* to the dimensions of *matrix*. If *array* is provided, it is bound to an array that whose elements are the elements of the *matrix*. This control structure can lead to substantially more efficient manipulation of dense matrices.

Ultimately, *array* will turn into an efficient accessor for the elements of *matrix* so that this technique will work with other than dense matrices.

transpose *matrix* [*Function*]

Transpose *matrix*.

jacobian *function-list var-list* [*Function*]

Returns the Jacobian of *function-list* with respect to *vector-list*. These should really be vectors, but list will work also.

direct-sum &rest *matrices* [*Function*]

Assumes the matrices all have the same number of rows, n. It returns a matrix of n rows, where each row is the concatenation of the rows of each of its arguments.

determinant *matrix* [*Function*]

Returns the determinant of the *matrix*.

sparse-determinant *matrix* [*Function*]

Returns the determinant of the *matrix*.

subdeterminant *matrix* [*Function*]

Returns the determinant of a nonsingular submatrix of the *matrix* of maximum rank.

hermite *matrix* [*Function*]

Returns the *Hermite normal* form of the *matrix*. At the moment, the method is implemented only for matrices over the ring of integers.

smith *matrix* [*Function*]

Returns the vector of diagonal elements of the *Smith normal form* of the *matrix*. At the moment, the method is implemented only for matrices over the ring of integers.

8.3 Matrix Groups

Chapter 9

Polynomial Rings

Polynomial rings are domains which consist of polynomials in some number of variables over a coefficient domain, e.g. $\mathbb{Z}[x, y]$, $\mathbb{R}[x]$, $\mathbb{Q}((t))[x]$. In these three cases the coefficient domains are the rational integers $\mathbb{Z}$, the real numbers $\mathbb{R}$ and powerseries' in t. Polynomial rings are created using the function get-polynomial-ring.

get-polynomial-ring *coefficient-domain list-of-variables* [*Function*]

> *Coefficient-domain* is must be a ring. *List-of-variables* is a list of arbitrary Lisp objects, each of which represents a generator of the polynomial ring.

transcendence-degree *domain1 domain2* [*Function*]

> This function checks to see that *domain2* is a subring *domain1*, and if so returns the transcendence degree of *domain1* over *domain2*.

9.1 Information About Variables

The variables of a polynomial ring have three different representations: as a polynomial, as symbol, or as a *variable index*. The simplest representation is as a symbol. These are the symbols passed to get-polynomial-ring and are the only mechanism the user has to influence the external printed representation of the variable. Internal to the polynomial package, there are (Lisp) integers associated with variables. These integers are used as indices into tables and to indicate the ordering of variables in multivariate polynomials. Finally, a variable can be represented as a polynomial.

Regardless of the internal representation used for polynomials, there is an integer associated with each variable of the polynomial. These numbers are used as an index into all data structures that hold additional information about the variable. The index corresponding to a variable is obtained by the generic function variable-index. The variable corresponding to an order number is variable-symbol.

variable-index *domain variable* [*Function*]

> Returns the variable index corresponding to *variable*. *Variable* can be either a symbol or a polynomial.

variable-symbol *domain variable* [*Function*]

Returns the variable symbol corresponding to *variable*. *Variable* can be either an integer or a polynomial.

There is a property list associated with each variable in a polynomial ring. This property list is "ring" specific and not global. The ring property list is accessed using the generic function get-variable-property. Properties can modified using setf, as with normal property lists.

get-variable-property *domain variable property* [*Function*]

Returns a *property* property of *variable*.

In contrast with normal mathematical usage, the polynomial rings of Weyl can be modified in a limited fashion after they have been created. In particular, it is possible to add variables to a polynomial ring, but removing variables is not allowed. The variables are added to the end of the ring's list of variables so that they will be less "main" than any of the existing variables, and thus only minimal changes to existing polynomial representations will be needed.

add-new-variable *ring var* [*Function*]

Var is a variable in the general representation. This function modifies *ring* to have an addition variable.

The behavior of this routine is illustrated by the following examples. First, we create a polynomial ring with two variables, and then add a third variable to it.

```
> (setq r (get-polynomial-ring (get-rational-numbers) '(x y)))
Q[x, y]

> (add-new-variable r 'z)
Q[x, y, z]
```

Now try adding a more complex expression to the ring. In this case, add-new-variable determines that w and sin(z+x) are the only new kernels and adds them.

```
> (add-new-variable r (expt (+ 'x 'w (sin (+ 'x 'z))) 3))
Q[x, y, z, w, sin(z + x)]
```

9.2 Polynomial Arithmetic

The simplest way to create a polynomial is to coerce a general expression into a polynomial ring. For instance, if r is the ring $\mathbb{Q}[x, y]$, as defined above, we could proceed as follows. First, we create a polynomial in x and y as general expressions.

```
> (expt (+ 'x 'y) 3)
(y + x)^3
```

Notice that the expressions is not expanded. Next, we coerce the expression into the polynomial ring r.

```
> (coerce (expt (+ 'x 'y) 3) r)
x^3 + 3 y x^2 + 3 y^2 x + y^3
```

This time the expression is expanded. When represented as an element of a polynomial ring, as opposed to being general expressions, polynomials are represented using a canonical representation (as described in Section 9.5). This canonical representation expresses polynomials as a sum of terms. This is easily seen in the following example.

```
> (setq p (+ (* (- 'x 'y) (+ 'x 'y)) (expt 'y 2)))
(-1 y + x) (y + x) + y^2
```

General expressions can represent polynomials as products of sums, and thus do not have a canonical representation for the polynomial. When the expression is coerced into a polynomial ring, the products are expanded to produce a simple form for the expression.

```
> (coerce p r)
x^2
```

9.3 Polynomial Operators

scalar? *polynomial* [*Function*]

The argument of this function is expected to be an element of a polynomial domain. If the argument is an element of the coefficient field this function returns T otherwise it returns nil

degree *polynomial var* [*Function*]

Returns the degree of *polynomial* in the variable *var* as a lisp integer.

The usual arithmetic operations including plus, minus, difference, times, quotient, recip and expt, can be applied to polynomials. For polynomial rings, expt is restricted to integer exponents.

remainder x y [*Function*]

Computes the polynomial pseudo-remainder of x and y.

gcd x y [*Function*]

Computes the polynomial greatest common divisor of x and y. The variable weyli::poly-gcd-algorithm is bound to the particular GCD algorithm to be used.

partial-deriv *polynomial var* [*Function*]

This function takes the partial derivative of *polynomial* with respect to *var*. *var* is actually a polynomial, not the lisp object which is the variable

deriv *polynomial var* &rest *vars* [*Function*]

Computes the derivative of *polynomial* with respect to *var*. This is done for *var* and each element of *var*. The variables can either be elements of *general* or of a polynomial domain.

coefficient *polynomial var* &optional *degree* [*Function*]

Compute the coefficient of the monomial in *var* of order *degree* in *polynomial*. *degree* defaults to 1.

substitute *value var polynomial* [*Function*]

Substitutes *value* for each occurrence of *var* in *polynomial*. If *value* is a list, it is interpreted as a set of values to be substituted in parallel for the variables in *var*. The values being substituted must be either elements of the domain of *polynomial* or its coefficient domain.

list-of-variables *polynomial* &optional *vars* [*Function*]

Returns a list of the variables that actually appear in *polynomial*. If provided *vars* must be a list. The variables that appear in *polynomial* are added to *vars*.

interpolate *vars points values* &optional *degrees* [*Function*]

Uses the information provided to produce a polynomial whose values at each of the specified points is the corresponding value. The variables of the polynomials are indicated by *vars*. For a univariate interpolation, *vars* should be a single variable and *points* and *values* should be simple lists. In this case the degree of the interpolated polynomial will be less than the length of *points*. For multivariate polynomials, *points* and *values* are lists of lists, where each of the sub-lists has the same length as the number of *variables*. The degree bounds for the multivariate polynomial can be specified by the *degrees* argument. It can be either a list of the maximum degrees in each of the variables, or the symbols :total or :maximum. In either of these two cases, interpolate uses the maximum degree bound that yields a polynomial with fewer terms than the number of points supplied. When :maximum is specified it is assumed that each variable can attain the maximum degree independently. So for two variables with a maximum of 10 terms, the possible terms of the returned polynomial are:

$$1, x, y, xy, x^2, x^2y, xy^2.$$

When :total is specified, with the same parameters, the possible terms are:

$$1, x, y, x^2, xy, y^2, x^3, x^2y, xy^2, y^3.$$

interpolate *domain Bp degree-bounds* [*Function*]

The *domain* should be multivariate-polynomial-ring. The *Bp* is the name of the function representing the black box, that is, the function should return the value of the polynomial at the requested point. The *degree-bounds* is the list of degree bounds for the multivariate polynomial in each of the ring variables of the domain. The function will return the multivariate polynomial computed using the probabilistic sparse multivariate interpolation algorithm given in the book by Zippel.

9.4 Differential Rings

Differential rings are polynomial rings that have *derivations*. A derivation of a differential ring $R\langle x\rangle$ is a map δ from $R\langle x\rangle$ to $R\langle x\rangle$ which has the following properties:

$$\delta(ap + bq) = a\,\delta p + b\,\delta q$$

$$\delta a = 0$$

$$\delta(pq) = p\,\delta q + q\,\delta p$$

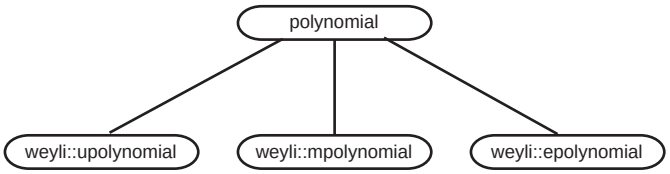

Figure 9.1: Polynomial Class Structure

where $a, b \in R$ and p and q are not elements of R.

get-differential-ring *domain variables* [*Function*]

Return a differential ring whose coefficients lie in *domain* and which contains the variables *variables*. For example:

```
> (setq R (get-differential-ring (get-rational-integers) '(x y)))
Z<x, y>
```

derivation p [*Function*]

Returns the derivative of the differential polynomial p. For example:

```
> (setq p (expt (+ (coerce 'x r) (coerce 'y r)) 2))
x^2 + 2 y x + y^2

> (deriv p)
(2 D{x, 1} + 2 D{y, 1}) x + ((2 D{x, 1} + 2 D{y, 1}) y)
```

variable-derivation *domain variable* [*Function*]

Returns the derivative of *variable*. The derivative can be either an element of *domain*, or *:generate* which indicates that the derivative will be a new variable which is yet to be created.

The derivative of a variable can be set using *setf*.

9.5 Structure Types for Polynomials

This section discusses the internal structures used represent polynomials. This material is only of value for those problems that require especially efficient access to the low level polynomial primitives in Weyl.

Polynomials are represented using one of three different structure types. The class structure of these types is given in Figure 9.1. The simplest structure is only used by elements of univariate polynomial rings and is called weyli::upolynomials. Two different representations are provided for multivariate polynomials. The weyli::mpolynomial structure type uses a recursive structure, so polynomials can be views as univariate polynomials with polynomial coefficients. This is the classical representation used by systems like Macsyma. The weyli::epolynomial structure is represents polynomials as a set of pairs of exponent vectors and coefficients.

In all three cases, instances of the the polynomial class include a slot called form, in which the data representing the polynomial is kept. These lower level data structures are what is actually

passed between low level polynomial routines. For univariate polynomials the form slot contains a vector of the polynomial's coefficients. Instances of weyli::mpolynomial contain a recursive list structure while instances of weyli::epolynomial contain a sorted list of the polynomial's monomials with non-zero coefficients. For efficiency the internal code used by the more complex algorithms uses these internal structures, not instances of the polynomial classes. The algorithms themselves however, accept their arguments and return values that are instances of the polynomial classes. This approach couples maximum system flexibility with efficient representations when needed.

Throughout the remainder of this section we will refer to the classes of these polynomials without the "weyli::" prefix for succinctness.

9.5.1 Multivariate Polynomials

There are two basic representations of polynomials, "recursive" and "expanded." The difference between these two is illustrated by the polynomial

$$x^3 + x^2(y + 3z^3) + xy^3 + y^4,$$

which is given in recursive form, and by

$$x^3 + x^2y + 3x^2y^3 + xy^3 + y^4,$$

in expanded form. Both of these forms express the polynomial as a sum of terms. The expanded polynomial is a somewhat simpler representation since it consists of a sum of monomials in all of the variables. Section 9.5.2 is devoted to this representation.

The recursive representation uses terms that are products of x to some power times a polynomial in the remaining variables. This coefficient is then represented recursively as a sum of terms in y with coefficients in the remaining variables. The details of the representation also depend on the variable order chosen. In the example given above, we have chosen the variable ordering x, y, z. If we had chosen z, y, x, the polynomial would have the following form:

$$z^3(3x^2) + (y^4 + y^3(x) + y(x^2) + x^3).$$

Multivariate polynomials are implemented using three different levels of structure. First, there is the mpolynomial structure type.

9.5.2 Expanded Polynomials

For some important algorithms, especial those of commutative algebra that are based on the Gröbner basis algorithm, it is convenient to represent a polynomial as a sorted list of monomials.

As in the previous section we assume that we are representing polynomials in the ring $k[x_1, \ldots, x_n]$, k is assumed to be a ring.

The monomials of an epolynomial are represented by a (simple) vector of $n + 1$ elements. The first component of the vector contains the coefficient of the monomial. The remaining components contain the exponents of the variables, represented as Lisp fixnums.

Associated with each epolynomial is a function that orders the monomials. These functions can be provided by the user, but it is usually preferable to let Weyl create them since Weyl's version is usually the most efficient. A selection of ordering function can be produced by the following function

weyli::make-comparison-fun *n list-of-vars* &key *(total? nil) (new? nil) (reverse? nil)* [*Function*]

Returns a function that can be used to order the monomial structures used by epolynomials. n is the number of variables in the ring, or equivalently, one more then the length of the vectors used to represent the monomials. *list-of-vars* is a list of the indices from 1 to n in the order in which the corresponding exponents should be examined. If *total?* is specified, then the total degree of the monomial will be tested before the individual exponents. weyli::make-comparison-fun tries to reuse functions that were previously created. If *new?* is specified, this is not done and a new function is generated. This is usually only necessary for debugging purposes.

When special orderings of the variables are not needed and one of the standard variable orderings is used, the following function is often more convenient.

weyli::get-comparison-fun n *type* [*Function*]

Returns a function that can be used as to order the monomial structures used by epolynomials. n is the number of variables ion the ring, or equivalently, one more than the length of the vectors used to represent the monomials. *type* is one of the keywords given the table below.

`:lexical`	Lexical monomial ordering
`:revlex`	Reverse lexical monomial ordering
`:total-lexical`	Monomials are ordered by their total degree. If they have the same total degree then the lexical ordering is used to break ties.
`:total-revlex`	Monomials are ordered by their total degree. If they have the same total degree then the reverse lexical ordering is used to break ties.

Expanded polynomials are created using the following function. Notice that its argument pattern is slightly different from that used to create univariate and recursive multivariate polynomials. It is also necessary to provide a term ordering function, such as one returned by weyli::get-comparison-fun.

weyli::make-epolynomial *domain greater-fun poly-form* [*Function*]

Generates an instance of an epolynomial in *domain*. The terms of the resulting polynomials are sorted using *greater-fun*. The argument *poly-form* can be any object that can be coerced into the ring *domain*.

9.5.3 Univariate Polynomials

Univariate polynomials are the simplest representation of polynomials used by Weyl and are only intended for special, performance intensive reasons. Their existence should not be visible to the user. Nonetheless, for certain algorithms significant performance improvements can be achieved through their use.

Chapter 10

Algebraic Structures

This chapter begins with a discussion of Weyl's representation of ideals and spectrums of rings. Taking the quotient of these structures and their rings gives leads to domains that can be used to represent algebraic numbers and algebraic functions.

10.1 Ideals

Recall that an ideal of a commutative ring R is an additive subgroup of R such that the product of any element of the ideal by an element of R is an element of the ideal. Thus the ideal is an R module.

Ideals are a slightly unusual objects in Weyl since they are treated like domain elements (arithmetic operations can be applied to ideals), but are in fact domains. One might think that they should be both domains and domains elements, and perhaps they will be in the future, but doing so now would be problematic. If they were domain elements, they would have to be elements of some domain and there is no natural domain that would suffice. (The spectrum of the ring only includes the prime ideals, and additional possesses a topology.)

The set of classes used to represent ideals is given in Figure 10.1. All ideals are subclasses of the class ideal. Every ideal is represented by a (not necessarily minimal) list of elements that generate the ideal. Currently, Weyl is only capable of representing ideals that are finitely generated. Ideals can be created using the following function:

make-ideal *domain* &rest *elements* [*Function*]

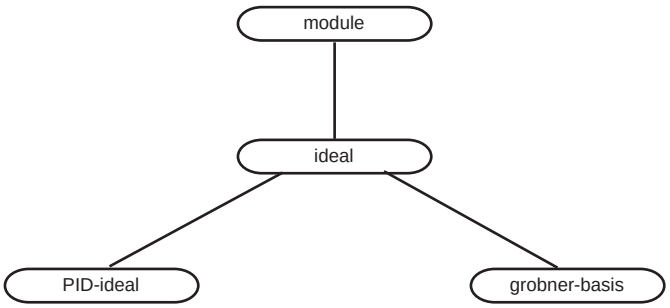

Figure 10.1: Ideal Class Structure

Creates the ideal of the ring domain, which is generated by the polynomials *polys*. The polynomials must all come from the same ring. Initially, we have only implemented the Grö bner basis algorithm for polynomials over fields, the domain of the polynomials must be a polynomial ring over a field.

There are two subclasses of the class ideal: PID-ideal and grobner-basis. If the ring of the ideal is a principal ideal domain, then make-ideal will create an instance of PID-ideal. The ring of the ideal is a polynomial ring then a grobner-basis ideal will be created.[1]

Two basic arithmetic operations are provided for manipulating ideals: addition and multiplication. Given the ideal ideals $A = (a_1, \ldots, a_m)$ and $B = (b_1, \ldots, b_n)$, their sum is defined to be

$$A + B = (a_1, \ldots, a_m, b_1, \ldots, b_n)$$

and their product is defined to be

$$A \cdot B = (a_1 b_1, a_1, b_2, \ldots, a_1 b_n, a_2 b_1, \ldots, a_m b_{n-1}, a_m b_n).$$

The standard + and * operators in Weyl are thus overloaded to work with ideals also. This is illustrated in the following simple examples:

```
> (setq a (make-ideal (get-rational-integers) 12))
#Id(12)

> (setq b (make-ideal (get-rational-integers) 15))
#Id(15)
```

An ideal is printed as a parenthesized list of its generators. Since we know that $\mathbb{Z}$ is a principal ideal domain, the list of generators will only have one element.[2]

```
> (+ a b)
#Id(3)

> (* a b)
#Id(180)
```

For a more sophisticated example consider:

```
> (setq r (get-polynomial-ring (get-rational-numbers) '(x y)))
Q[x, y]

> (setq a (make-ideal r (+ (* 'x 'x 'x) (* 'y 'y)) (* 'x 'y)))
#Id( x y,  x^3 +  y^2)

> (setq b (+ a (* a a)))
#Id(x^2 y^2, x^4 y + x y^3, x^4 y +  x y^3, x^6 + 2 x^3 y^2 + y^4, x^3 + y^2, x y)
```

Although the last two results do exhibit bases for a ideals, they are not the minimal bases. A reduced basis can be obtained by applying the following routine.

[1] The name of the class grobner-basis is very poor and should be changed so that it doesn't refer to a particular algorithm.

[2] There are a lot of very delicate issues here. The test for whether or not the ring is a PID is generally quite difficult. In the current implementation, Weyl explicitly checks for $\mathbb{Z}$. Even if one knows the ring is a PID, it can be difficult to determine the ideal's generator. The current implementation assumes that in addition to being a PID there is an explicit GCD algorithm available.

reduce-basis *ideal* [*Function*]

Reduces the basis provided for the ideal using the techniques available. For polynomials ideals this means computing the Gröbner basis assuming a monomial ordering has been provided.

When this routine is applied to the previous example we have:

```
> (reduce-basis a)
#Id( x y,  x^3 +  y^2,  y^3)

> (reduce-basis b)}
#Id( x y,  x^3 +  y^2,  y^3)
```

So we see that a is equal to (+ a (* a a)). Further notice that the reduce basis has more elements that the original basis.

10.2 The Spectrum of a Ring

This section needs to discuss the topological structure of the spectrum as well.

spectrum *Ring* [*Function*]

This function returns a new domain which is the spectrum of *ring*. The elements of this domain are the ideals of *ring*. (Need to figure out how the topology of Spec R fits into all this.)

10.3 Algebraic Extensions

algebraic-degree *domain1 domain2* [*Function*]

This function checks to ensure that *domain2* is a subring of *domain1* and, if so, returns the algebraic degree of *domain2* over *domain1*.

10.4 Algebraic Closures

Need to special case the algebraic closure of the real numbers. This is the only algebraic closure that is of finite degree.

Chapter 11

Truncated Power Series

Formal power series are infinite degree polynomials, where the arithmetic operations are defined formally using the same rules as a polynomial arithmetic. Thus

$$f(x) = \sum_{0 \le i} a_i x^i = a_0 + a_1 x + a_2 x^2 + \cdots,$$

is a power series, and we have

$$\left(\sum_{0 \le i} a_i x^i \right) \times \left(\sum_{0 \le i} b_i x^i \right) = \sum_{0 \le i} \left(\sum_{0 \le j \le i} a_j b_{i-j} \right) x^i.$$

Some operations are possible with formal power series that are not possible with polynomials. For instance, if the leading coefficient of a formal power series is invertible then the formal power series has a reciprocal.

Rings of formal power series are denoted using a similar notation to that used for polynomials except that the brackets are doubled. Thus, $\mathbb{Z}[[x]]$ and $\mathbb{Q}[t][[x]]$ represent rings of formal power series in x whose coefficients are rational integers and polynomials with rational number coefficients. The quotient field of a ring of formal power series consists of formal power series whose leading term can have negative order. For example, we have

$$\left(x + x^2 + \cdots \right)^{-1} = \frac{1}{x} - 1 + \cdots.$$

The field of formal power series is denoted using doubled parenthesis—similar adaptation to the notation for the field of rational functions. Thus, $R((x))$ is the field of formal power series with real number coefficients.

In mathematics one also deals with "regular" power series, which are formal power series that are convergent in some sense. Thus, while

$$1 + 2x + 3!x^2 + 4!x^3 + \cdots,$$

is a perfectly reasonable formal power series, but it is not convergent for any value of x. When one represents physical quantities, convergent power series are usually more interesting, since they can actually be evaluated. However, arithmetic operations with power series may shrink their radius of convergence, and keeping track of the radius of convergence throughout the computation can be quite difficult. Instead, it is often more convenient to deal with formal power series and leave the convergence issues to later.

While Weyl does not have a representation for the power series (either convergent or formal), it does provide tools for manipulating truncated power series. A truncated power series only contains those monomials of a formal power series that have degree less than some bound. Thus, while the function $\sin x$ has the infinite power series expansion

$$\sin x = x - \frac{x^3}{6} + \frac{x^5}{120} - \frac{x^7}{5040} + \cdots,$$

its truncated power series expansion is

```
> (setq sinx (taylor (sin qx) domain 5))
x - (1/6)x^3 + (1/120)x^5 - (1/5040)x^7 + o(x^5)
```

The small "o" notation indicates that all terms of order greater than 5 have been ignored. The argument domain is some domain of truncated power series.

The following sections describe the tools provided by Weyl for computing with formal power series.

11.1 Creating Truncated Power Series Domains

Truncate power series rings and fields can be created using the function get-tpower-series.

get-tpower-series-domain *coefficient-domain variable* [*Function*]

> *Coefficient-domain* must be a ring. If the *coefficient-domain* is also a field, the constructed truncated power series domain will be a field. *variable* should be coercible into a general expression variable. (Notice that this is somewhat more restricted than what is permissible for polynomials. At some point in the future this restriction will be lifted.) The variable indicated will be the generator of the truncated power series.

The following form creates the ring of power series with rational integer coefficients

```
> (weyli::get-tpower-series-domain (get-rational-integers) 'x)
Z[[x]]
```

Notice that this domain is actually a ring and not a field is indicated by the square brackets. When a power series domain is created over a field, round brackets are used:

```
> (setq domain (get-tpower-series-domain (get-rational-numbers) 'x))
Q((x))
```

11.2 Truncated Power Series Operators

The simplest way to actually create a truncated power series is to use the Taylor series expansion function:

taylor *expr tps-domain order* [*Function*]

> Returns the power expansion of the general expression *expr* as a truncated power series of order order in the domain *tps-domain*. The variable of expansion will be the power series variable of *tps-domain*.

For instance, to compute the power series expansion of $\sin x$ one could type

```
> (setq sinx (taylor (sin 'x) domain 8))
x - (1/6)x^3 + (1/120)x^5 - (1/5040)x^7 + (1/362880)x^9 + O(x^10)
```

The taylor function also works with more complex functions.

```
> (setq temp1 (taylor (cos (sin 'x)) d 5))
1 - (1/2)x^2 + (5/24)x^4 + o(x^5)

> (setq temp2 (taylor (sin (sin 'x)) d 5))
x - (1/3)x^3 + (1/10)x^5 + o(x^5)
```

Arithmetic operations can also be performed with power series expressions:

```
> (+ (* temp1 temp1) (expt temp2 2))
1 + o(x^5)
```

Although taylor returns a truncated power series approximation to the power series expansion of the expression, if the coefficient arithmetic is exact, then every coefficient returned will be exact. So, in this case up to order 5, the only non-zero term in the power series expansion is 1.

Truncated power series can be manipulated with the usual arithmetic operations. For binary operations, the order of the result may be reduced to ensure that only the known coefficients are advertised as known. For instance, although temp2 has order 5 its sum with a truncated power series of order 3 is only of order 3.

```
> (+ temp2 (taylor (sin 'x) d 3))
2x - (1/2)x^3 + o(x^3)
```

Notice also, the even unary operations can yield results with different orders, *viz.*,

```
> (* temp2 temp2)
x^2 - (2/3)x^4 + (14/45)x^6 + o(x^6)

> (expt temp2 -3)
x^(-3) + x^(-1) + (11/30)x + o(x)
```

Nonetheless, when the taylor function is requested to compute the power series expansion to a given order, it computes the internal subexpressions to a sufficiently high order, to ensure that the correct coefficients are computed.

```
> (expt (- temp1 1) -3)
(-8)x^(-6) - 10x^(-4) + o(x^(-3))

> (taylor (expt (- (cos (sin 'x)) 1) -3) d 5)
(-8)x^(-6) - 10x^(-4) - (88/15)x^(-2) - 16783/7560 - (32267/50400)x^2
 - (1045753/6652800)x^4 + o(x^5)
```

The power series package can also deal with power series with fractional exponents. For instance,

```
> (expt (taylor (sin qx) d 5) -1/3)
x^(1/3) - (1/18)x^(7/3) - (1/3240)x^(13/3) + o(x^(13/3))
```

One power series can be substituted into another using the substitute function. In addition, derivatives of power series can also be computed using deriv.

substitute *value var tps* [*Function*]

Returns the truncated power series computed by substituting the truncated power series value for the ring variable in *tps*. The general expression variable var should be equal to the ring variable of the domain of *tps*.

deriv *tps* &rest *vars* [*Function*]

The derivative of the *tps* with respect to each variable in *vars* is computed successively and the final truncated power series is returned.

For instance, another way to compute the first few terms of the power series of $\sin\sin x$ is to substitute the power series of $\sin x$ into itself.

```
> (substitute (taylor (sin 'x) d 5) qx (taylor (sin 'x) d 5))
x - (1/3)x^3 + (1/10)x^5 + o(x^5)
```

As mentioned, before the order of the resulting power series expansion may be less than the order of arguments.

```
> (deriv * 'x)
1 - x^2 + (1/2)x^4 + o(x^4)
```

One of the operations that can be performed with power series that cannot be performed with polynomials is the reversion. That is, given a power series for $f(x)$, we can compute the power series for $f^{-1}(x)$. This computation is performed by the function revert

revert *tps* [*Function*]

Returns the truncated power series that is the reversion of *tps*.

For instance, given the power series expansion for $\sin x$

```
> (setq sinx (taylor (sin 'x) domain 10))
x - (1/6)x^3 + (1/120)x^5 - (1/5040)x^7 + (1/362880)x^9 + O(x^10)
```

we can compute the reversion of $\sin x$ as follows:

```
> (reversion sinx)
x + (1/6)x^3 + (3/40)x^5 + (5/112)x^7 + (35/1152)x^9 + O(x^10)
```

If we substitute the reversion of sinx obtained above for x in sinx, we should get x, as expected.:

```
> (substitute (reversion sinx) 'x sinx)
x + O(x^10)
```

solve-diff-eqn *diff-eqn coef-ring var order init-list* [*Function*]

Solves the differential equation *diff-eqn* given in the form of a general expression. The solution is given as a truncated power series of order *order* with coefficients in the *coef-ring* with *var* taken as the ring variable. The *init-list* is the list of initial values of length equal to the order of the *diff-eqn*.

The following examples illustrate how to obtain power series solutions to ordinary differential equations in one variable:

```
> (setq y (funct y 'x))
y(x)

> (setq y0 (deriv y 'x) y00 (deriv y 'x 'x))
y_{0}(x)

> (setq diff-eqn1 (+ y y0 (* -1 qx qx))
y(x) + y_{0}(x) - x^2

> (solve-diff-eqn diff-eqn1 (get-rational-numbers) 'x 6 '(1))
1 - x + (1/2)x^2 + (1/6)x^3 - (1/24)x^4 + (1/120)x^5 - (1/720)x^6 + o(x^6)

> (setq diff-eqn2 (- y00 y))

-1 (y(x)) + y_{00}(x)

> (solve-diff-eqn diff-eqn2 (get-rational-numbers) 'x 5 '(c0 c1))

c0 + c1x + ((c0)/(2))x^2 + ((c1)/(6))x^3 + ((c0)/(24))x^4 + ((c1)/(120))x^5 + o(x^5)
```

11.3 Truncated Power Series Internals

This section describes the internal representation used to implement truncated power series and is only of interest to those who want to extend the package.

Truncated power series are currently represented by a single Lisp class, weyli::tpower-series, which is a representation in which all of the coefficients are enumerated in an array. Generative representations are not currently available.

11.3.1 Enumerated Truncate Power Series

This class, currently the only representation for power series, maintains an array of known coefficients and is therefore useful only for power series with a finite and typically small number of nonzero coefficients. The power series may have an infinite order (in which case the behavior is nearly identical to that of polynomials) but in such cases the largest nonzero term must have a finite exponent. This class will perform best when used with dense power series (*i.e.*, power series in which the sequences of exponent numerators of nonzero terms do not have large skips).

Truncated power series in Weyl are a little more complex than the proceeding discussion might indicate. In particular, we permit truncated power series to have negative and fractional exponents in order to allow for certain algebraic operations on power series. For instance, we would like to represent

$$\sqrt{x+x^2} = x^{1/2} - \frac{x^{3/2}}{2} + \frac{x^{5/2}}{8} + \cdots,$$

and the reciprocal of the function

$$\frac{1}{\sqrt{x+x^2}} = x^{-1/2} - \frac{x^{1/2}}{2} + \frac{3}{8}x^{3/2} - \frac{5}{16}x^{5/2} + \cdots.$$

To deal with these issues the truncated power series representation in Weyl has four components. The most obvious is the vector with coefficients of the power series. The *branch order* of the power

series is the least common multiple of the denominators of the exponents that arise in the power series. Thus in the pervious two examples the branch order is 2.

The *valence* of the power series is the degree of the term with nonzero coefficient with least degree when multiplied by the branch order. In the previous two examples, the valences are 1 and -1 respectively.

Finally, the product of the branch order and the degree of the highest term retained in the truncated power series is called the *order* of the power series. Notice that these definitions have been chosen so that all of the parameters are integers.

The following routines can be used to access the parameters of a truncated power series.

branch-order *tps* [*Function*]

Returns the branching order of *tps* as a lisp integer. This number is the greatest common divisor of all of the denominators of the exponents in the truncated power series.

order *tps* [*Function*]

Returns the order of *tps* as a Lisp integer. This number is the product of the greatest exponent in the truncated power series and the branch order of the power series.

valence *tps* [*Function*]

Returns the valence of *tps* as a Lisp integer. This number is the product of the least exponent in the truncated power series and the branch order of the power series.

The following examples illustrate the use of these routines:

```
> (setq temp (taylor (expt (+ 'x (* 'x 'x)) -1/2) d 2))
x^(-1/2) - (1/2)x^(1/2) + (3/8)x^(3/2) - (5/16)x^(5/2) + o(x^(5/2))
```

temp is the power series of the expression given above. It has both negative and fraction exponents.

```
> (describe temp)
x^(-1/2) - (1/2)x^(1/2) + (3/8)x^(3/2) - (5/16)x^(5/2) + o(x^(5/2))
  is an instance of the class WEYLI::TPOWER-SERIES:
The following slots have allocation :INSTANCE:
DOMAIN          Q((x))
VALENCE         -1
BRANCH-ORDER    2
ORDER           5
COEFFS          #(1 0 -1/2 0 3/8 0 -5/16)
```

Notice that the branch order, valence and order are all rational integers.

The branch order of the power series can be modified by using setf and the branch-order accessor. This operation increases the size of the coefficient array of the power series and adjusts the order and valence appropriately. This operation is used internally to before combining two power series with different branching orders.

```
> (setf (weyli::branch-order temp) 4)
x^(-1/2) - (1/2)x^(1/2) + (3/8)x^(3/2) - (5/16)x^(5/2) + o(x^(5/2))
```

Notice that the outward appearance of the power series has not changed, even though the branching order has been increased. However, the internal form has changed.

```
> (describe temp)
x^(-1/2) - (1/2)x^(1/2) + (3/8)x^(3/2) - (5/16)x^(5/2) + o(x^(5/2))
  is an instance of the class WEYLI::TPOWER-SERIES:
The following slots have allocation :INSTANCE:
DOMAIN          Q((x))
VALENCE         -2
BRANCH-ORDER    4
ORDER           10
COEFFS          #(1 0 0 0 -1/2 0 0 0 3/8 0 ...)
```

When writing low level code using truncated power series, it is more convenient not to use the accessors for the slots in a truncated power series, but instead to have variables bound to the various pieces. This is done by the following special form.

weyli::with-tpower-series *((var$_1$ tps$_1$) ... (var$_1$ tps$_1$))* &body *body* [*Function*]

For each truncated power series provided, this form binds four new variables, with names based on *var$_i$*, one for each of the slots in *tps$_i$*. The names of these slots are *var$_i$*-bo for the branching order, *var$_i$*-val for the valence, *var$_i$*-order for the order and *var$_i$*-coeffs for the coefficient vector.

Similarly, the following function need not be here but should be a setf method on order

weyli::truncate-order *tps integer* [*Function*]

Truncates the order of *tps* so that the largest exponent is no greater than *integer*. If *integer* is greater than or equal to the current truncation order, then *tps* is returned unaltered.

Chapter 12

Spaces and Topology

Spaces are sets of mathematical objects (*domains* in Weyl's terminology) that we endow with some topological structure. Examples of spaces include n-dimensional Euclidean space, the space of square integrable functions and the spectrum of a ring. In addition, we often provide a metric to provide the geometric structure of a space. This chapter describes the tools Weyl provides for creating and manipulating these spaces.

The basic domains Weyl provides for spaces are shown in Figure 12.1. All spaces are subclasses of abstract-space. It would be very useful to be able to indicate that a space is Hausdorff, connected, compact or has some similar topological property. This is not generally possible at the moment, but will be added at some future date.

One property that can be attached to a space is its *dimension*. Intuitively, the dimension of a space can be thought of as the number of orthogonal directions that can be taken in the space. However, this does not apply to spaces that are not embedded in some Euclidean space. Mathematically, the dimension of a topological space is either a non-negative integer or infinity.[1] For Weyl's purposes, it is useful to, in addition, have spaces without a specific dimension. A dimensional-space is a space that has a specific, possibly infinite, dimension.

A very useful, specific type of dimensional space is a *Euclidean space*. A Euclidean space is $\mathbb{R}^n$ endowed with the Euclidean metric for measuring distances between points. The open sets of a Euclidean space can be constructed via set union from the open balls $B_d(p)$, the set of points of

[1] By using the Hausdorff dimension for the dimension of a space, we can construct spaces that have non-integral dimension.

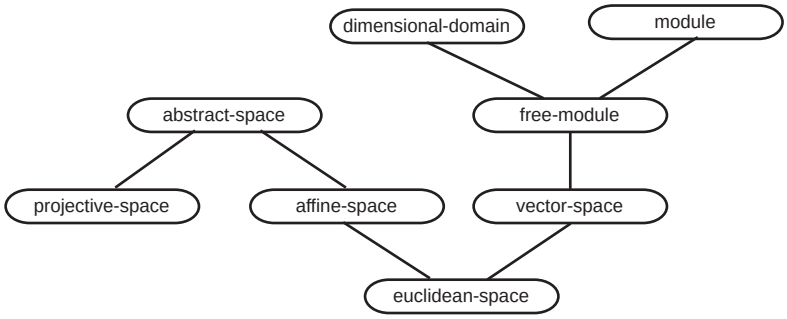

Figure 12.1: Basic Domains for Spaces

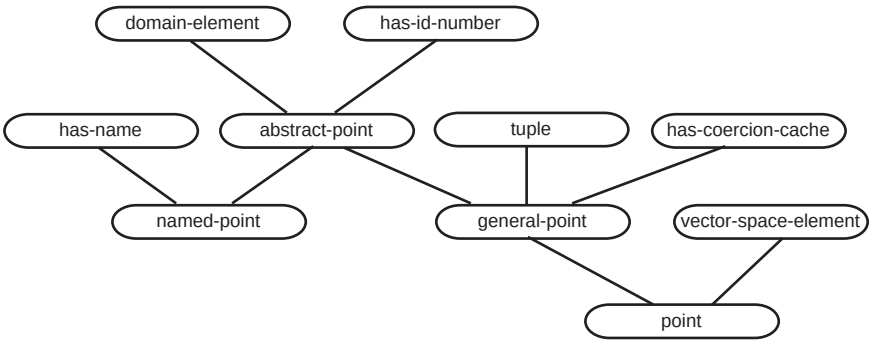

Figure 12.2: Basic Classes for Points

distance less than d from the point p. The open sets are closed under union and finite intersection. The closed sets are the complements of the open sets.

Closely related to Euclidean spaces are general vector spaces and modules. Recall from Section 4.5.3, that M is an R-module if M is an abelian group and if multiplication by elements of R is closed in M. A *free module* is a module that is freely generated. As can be seen from Figure 12.1, Weyl only permits free modules that have specified dimensions. A *vector space* is a free R-module where R is a field. From Figure 12.1 we see that Weyl treats a Euclidean space as a vector space, where the coefficient domain is the real numbers, R. The routines for dealing with free modules are described in Section 7.2.

Finally, Weyl provides a mechanism for dealing with projective spaces. A projective space is an $n+1$ dimensional space together with an equivalence relation that equates points that are nonzero scalar multiples of each other, i.e., $u \cong v$ if $u = \alpha v$, for $\alpha \neq 0$. Due to the equivalence relation, the projective space has dimension n.

The actual dimensions of a space can be determined by the function dimensions.

dimensions *space* [*Function*]

> Returns a list of the dimensions of the space. For free-modules, vector spaces, projective spaces, and so on there is only one element in this list. However for spaces like $\mathbb{R}^3 \times \mathbb{Z}^5$ there may be more elements.

12.1 Point Set Topology

Objects in topologies are subsets of *spaces*. The root class of all spaces is abstract-space. Abstract spaces do not have a dimension and consist only of points. To create an abstract space one must explicitly call make-instance on the class abstract-space. The most commonly used spaces, the Euclidean spaces, can be created using the function get-euclidean-space:

get-euclidean-space *dimension* &optional *(domain* *general**) [*Function*]

> Creates, if needed, an instance of the Euclidean space with the indicated dimension. Elements of the Euclidean space will be represented as n-tuples. The components of these n-tuples will be elements of *domain*.

To create points in a space we use the function make-point:

make-point *space coord1* &rest *coords* [*Function*]

Creates a point in *space*. If *space* is an abstract space, then only one argument is expected and will be treated as a unique identifier for the point in the space. If the first coordinate is nil then a new *anonymous* point will be created. For vector spaces, the values are expected to be the coordinates of the point.

All points in a space are identified by a unique integer identifier. This is currently implemented (using the class weyli::has-id-number) by including a slot in each point containing an integer unique to that point. When anonymous points are generated in an abstract space, the printed representation is distinguished by this number. For some spaces, *e.g.*, $\mathbb{R}$, it may be more appropriate to use a different ordering.

```
> (setq abs-space (make-instance 'abstract-space))
#<Domain: ABSTRACT-SPACE>

> (progn
     (setq a (make-point abs-space nil)
           b (make-point abs-space nil)
           c (make-point abs-space 'c))
     (list a b c))
(<1> <2> <C>)
```

Notice that the third point was created with a name and, unlike the other two points, its printed form includes that name.

Similarly, we can create points in Euclidean domains. In this case, the printed representation of a point includes the point's coordinates.

```
> (setq ee (get-euclidean-space 3))
E^3

> (make-point ee 1 2 3)
<1, 2, 3>
```

12.2 Affine Spaces

Affine spaces are created using the function make-affine-space:

make-affine-space *field dimension* [*Function*]

Create an affine space of dimensions *dimension* where the components are elements of the field *field*. (Ed: I think this should be weakened to work for arbitrary rings.)

```
>(setq R (get-real-numbers))
R
>(setq space (make-affine-space R 2)
```

Elements of spaces, both affine and projective, are created using the generic function make-point:

make-element *space* &rest *elements* [*Function*]

Creates an element of *space*. *Elements* is a list of n elements which can be coerced into the coefficient domain of space.

There is also an internal function weyli::make-element that does not do any checking of its arguments and can lead to rather subtle problems if used incorrectly. On the other hand it is noticeably faster.

cross-product u v [*Function*]

This function is only defined for elements of three dimensional vector spaces. If $u = \langle u_1, u_2, u_3 \rangle$ and $v = \langle v_1, v_2, v_3 \rangle$ then

$$(\text{cross-product} uv) = (v_2 u_3 - u_3 v_2, u_3 u_1 - u_1 v_3, u_1 v_2 - u_2 v_1).$$

12.3 Projective Spaces

make-projective-space *field dimension* [*Function*]

Create a projective space of dimensions *dimension* where the components are elements of the field *field*.

As in the case of regular affine spaces, elements of a projective space are created using the generic function make-point.

make-point *space* &rest *elements* [*Function*]

Creates a point which an element of *space*. For projective spaces, *elements* can be a list of either n or $n + 1$ elements of the coefficient domain of space. If $n + 1$ elements are provided then these are the full set of elements of the point. If only n elements are provided, the final missing element is filled out by a 1 from the coefficient domain of space.

```
(setq p (make-projective-space r 2))
(make-point p 1 1 1)
```

Affine spaces can be embedded in projective spaces. For projective spaces of dimension n, there are $n + 1$ canonical embeddings. The function make-affine-projection is passed a projective space and creates an affine space with an attached homomorphism into the projective space.

make-affine-projection *space* &optional *dimension* [*Function*]

This function returns an affine space that is the projection of *space* where we hold the component *dimension* fixed. For instance, let $p = (u, v, w)$ is an element of a two dimension projective space P^2. The image of p in the affine projection of P^2 with dimension 0 held fixed is $(v/u, w/u)$. With dimension 1 is held fixed then p maps to $(u/v, w/v)$. Finally, if dimension 2 is held fixed, p maps to $(u/w, v/w)$. If *dimension* is not provided then we produce an affine space where the last dimension is held fixed.

12.4 Algebraic Topology

12.4.1 Cells and Simplices

A *k-cell* is a region (subset) of a space that is homeomorphic to B^n, the unit ball in R^n: $B^n = \{p \in R^n | ||p|| <= 1\}$. That is, a k-cell is a set that is topologically equivalent to an n-dimensional ball – it is connected, without any holes, etc.

A *k-simplex* is a k-cell defined by the convex hull of $k+1$ points, called vertices. For example, a 1-simplex is a line segment defined by two vertices, while a 2-simplex is a triangle, defined by three. An *oriented* k-simplex is defined by an ordered list of vertices, and the lists are partitioned into two *orientations* – those that are even permutations of some reference ordering, and those that are odd permutations.

Unlike most objects in Weyl, (oriented) simplices are not domain elements. They should be viewed as sorted lists of sets of points.

make-simplex $vertex_0$ ... $vertex_k$ [*Function*]

Creates a k simplex whose components are $\mathsf{vertex}_0, \ldots, \mathsf{vertex}_k$. This routine ensures that all of the points are from the same space. Simplices are immutable, *i.e.*, once created they can not be modified.

Assuming that the variables a, b and c are points in some abstract space, a triangle could be created as follows:

```
> (setq triangle (make-simplex a b c))
[<A>, <B>, <C>]
```

Two simplices are equal if they have the same vertex set, with the same orientation.

```
> (= triangle (make-simplex a b c))
t
```

vertices-of *simplex* [*Function*]

Returns a list of the vertices of *simplex*. For instance,

```
> (vertices-of triangle)
(<A> <B> <C>)
```

Notice that there aren't any commas between the vertices. This is a LISP list as opposed to a simplex.

face? *cell1 cell2* [*Function*]

A predicate that returns T if *cell1* is a face of *cell2*. Cells are defined to be faces of themselves.

same-cell? *cell1 cell2* [*Function*]

A predicate that returns T if *cell1* and *cell2* are the same cell, independent of orientation.

dimension-of *cell* [*Function*]

Returns a Lisp integer that is the dimension of *cell*. Weyl provides a dimension function only for simplices.

12.4.2 Complexes

A *simplicial complex* K is a set of simplices of the same space with the property that if $s \in K \rightarrow \text{faces}(s) \subset K$, that is if a simplex s is in the complex, the all of the faces of s must also be in K. It follows that there is a set of maximal simplices (those that are not the face of any other simplex in

K) that provide a unique minimal representation for K. Although the maximal cells in a complex often have the same dimension, this is not required.

make-simplicial-complex *simplices* [*Function*]

Creates a simplicial complex containing each of the simplices in *simplices* together with their faces .

```
> (setq complex (make-simplicial-complex (list triangle)))
#<COMPLEX>
```

map-over-cells *(var* &optional *n complex) struct* &body *body* [*Function*]

The forms in *body* are evaluated with *var* bound to each n-dimensional face of *struct*. If n is nil then the body is evaluated for all faces, regardless of dimension. *struct* may be either a simplex or a simplicial complex. If *complex* is nil (the default), then map-over-cells maps over lists of vertices (i.e., *var* will be set to a list of the vertices of the given cell. Otherwise map-over-cells maps over the canonical cell structures in *complex*.

Here is an example of mapping over the faces of a 2-simplex `[<A>, <B>, <C>]`. Note that since dimension is not specified, simplices of all dimension are printed. Also, since no *complex* was specified, each simplex is represented by the list of its vertices.

```
> (map-over-cells  (simp) triangle
    (print simp))
(<A>)
(<B> <A>)
(<C> <B> <A>)
(<C> <A>)
(<B>)
(<C> <B>)
(<C>)
```

A second example shows iteration over the 1-cells of `triangle`, first with, then without specifying the complex from which the simplex structures should be extracted. Note the brackets used in the Weyl representation of a simplex in the second example.

```
(map-over-cells  (simp 1)  triangle
    (print simp))
(<B> <A>)
(<C> <A>)
(<C> <B>)

(map-over-cells  (simp 1 complex)  triangle
    (print simp))
[<A>, <B>]
[<A>, <C>]
[<B>, <C>]
```

get-canonical-cell *cell-complex cell* [*Function*]

If *cell* is contained in *cell-complex* then get-canonical-cell returns two values: the canonical cell with the vertices of *cell* that lies in *cell-complex*, and the sign of the relative orientation between the cell returned and *cell*. If *cell* is not contained in *cell-complex* then nil is returned.

get-canonical-cell *cell-complex* &rest *points* [*Function*]

If the cell whose vertices are *points* is contained in *cell-complex* then get-canonical-cell returns two values: the canonical cell with vertices *points* that lies in *cell-complex*, and the sign of the relative orientation between the ordering of the cell returned and the ordering of the points provided. If *cell* is not contained in *cell-complex* then nil is returned.

vertex-set *complex* [*Function*]

Returns a list of the vertices of each of the maximal cells in *complex*.

Cell complexes are, in general, immutable. New cell complexes can be created from old cell complexes using boolean operations like union and intersection.

union &rest *complexes* [*Function*]

Returns a new complex, whose maximal cells include the maximal cells of each of the elements of *complexes*.

intersection &rest *complexes* [*Function*]

Returns a new complex, which is the intersection of the elements of *complexes*.

The following operations violate the immutability of cell complexes and thus only intended to be used in situations that require additional performance, or to implement higher level operations which do preserve the immutability of their arguments.

New simplices can be added and deleted from cell-complexes using insert and delete.

insert *cell cell-complex* [*Function*]

Destructively modifies cell-complex by adding *cell* and each of its sub-cells to cell-complex if each is not not already an element. This operation is provided for use by internal routines.

delete *cell cell-complex* &key *(subsimplices?* T*)* [*Function*]

Destructively modifies cell-complex by deleting *cell* from cell-complex. Any subcells of a cell that are not contained in the remaining maximal cell of cell-complex are also deleted. This operation is provided for use by internal routines.

12.4.3 Chains

If K is an oriented simplicial complex, and G an abelian group, a p-chain $c_p \in C_p(K, G)$ is a map $c_p : p - \text{simplices}(K) \to G$ that assigns an element of G to each p-simplex in K. Equivalently, we may say that a p-chain is a *formal sum* of the p-simplices of K with coefficients in G. The group operation in G is then used to define the $+$ operator for $C_p(K, G)$, yielding the abelian group of p-chains over K and G.

get-chain-module *complex n* &optional *group* [*Function*]

Creates the group of n-chains of *complex* with coefficients in *group*. If *group* is not provided then $\mathbb{Z}$ is used instead. If provided, *group* must be an abelian group.

Individual chains may be created by coercing a simplex into the chain module.

```
> (setq 1-chains (get-chain-module complex 1))
C[1](#<COMPLEX>)

> (coerce (make-simplex a b) 1-chains)
[<A>, <B>]

> (+ (coerce (make-simplex a b) 1-chains)
     (* 2 (coerce (make-simplex b c) 1-chains)))
[<A>, <B>] + 2[<B>, <C>]
```

If $\sigma = [v_0, \ldots, v_p]$ is a p-simplex, then its boundary is the following $p-1$ chain:

$$\partial\sigma = \partial[v_0, \ldots, v_p] = \sum_{i=0}^{p} (-1)^i [v_0, \ldots, \hat{v_i}, \ldots, v_p],$$

where $\hat{v_i}$ indicates that the v_i vertex is missing. The boundary operation can be extended to chains by linearity.

boundary *chain* [*Function*]

chain can be either chain or a simplex. It returns the chain representing the boundary of *chain*.

This is illustrated by computing the boundary of triangle.

```
> (setq tri-bound (boundary triangle))
[<A>, <B>] + [<B>, <C>] - [<A>, <C>]

> (boundary tri-boundary)
0
```

It is generally true that $\partial^2\sigma = 0$ for all simplices σ.

boundary-domain *chain* [*Function*]

Returns the chain-module that is the domain for the boundary of *chain*.

boundary-domain *simplex* [*Function*]

Returns a chain whose coefficients are the derivatives of the given chain coefficients. It is assumed that the new chain is in the same chain module as the old chain.

deriv *chain* &rest *params* [*Function*]

Returns a chain whose coefficients are the derivatives of the given chain coefficients. It is assumed that the new chain is in the same chain module as the old chain.

Chains may be added and multiplied by elements of their coefficient domain.

make-chain *chain-module simplex-coefficient-pairs* [*Function*]

Creates a chain

Chapter 13

Meshing

Weyl provides for the creation and manipulation of unstructured (triangular) meshes. Currently, these facilities are limited to 2-dimensional (flat) meshes, but we plan to add facilities for meshing curved surfaces (embedded in 3-dimensions) in the near future. A desirable feature of the meshing technique used by Weyl is that the resulting meshes are mathematically guaranteed to exhibit the following properties:

- Boundaries, both external and internal, are respected.
- Element shapes are guaranteed. All elements are triangles with angles between 30 and 120 degrees (with the exception of badly shaped elements that may be required by the specified boundary).
- Element density can be controlled, producing small elements in "interesting" areas and large elements elsewhere.

A mesh is implemented as a simplicial-complex with some additional information attached. This additional information includes a subsidiary simplicial-complex representing the boundary. Both external boundaries and internal boundaries are supported. Internal boundaries are used, for instance, when analyzing an object made out of more than one material or when analyzing cracks.

The desired shapes for the triangles of the mesh are specified by giving angle-bounds for the minimum and maximum angles that are allowed to appear in the mesh.

The mesh density is controlled by specifying a size-list. Basically, this is a simple (flattened) table associating names with sizes where a size can be either a simple number (used as an upper bound on the size) or a function of two numerical arguments that returns a number. Both boundaries and subregions can have names that appear in a size-list.

13.1 Creating and Storing Meshes

Meshes are created by using make-mesh, by using make-mesh-from-file, or by using individual meshing functions as described below in Section 13.3. Meshes can be stored in a file by using write-mesh. The format of the resulting file is described in Section 14.2.

The two simplest ways to create a mesh are to use either make-mesh or make-mesh-from-file. Make-mesh provides syntactic sugar to make it easy to specify a mesh from within a program. Make-mesh-from-file controls mesh creation via a data file; the file's format is specified in Section 14.2. Make-mesh is somewhat more versatile since it allows size bounds that are functions while the

data-file format allows only size bounds that are numeric constants. Note though that size bounds in the data file can be overridden by size bounds specified in the call to make-mesh-from-file; thus, size bounds that are functions can overrride the constant bounds in the file.

make-mesh (*space* &key *angle-bounds size-list*) . *boundary-and-region-specs* [*Macro*]

> Use the boundary and region specifcations to create a mesh in which the elements satisfy the the requirements given by *angle-bounds* and *size-list*.

The *angle-bounds* represent the minimum and maximum angles allowed in the triangles of the mesh (note though that smaller angles can occur in the final mesh if the specified boundary includes small angles). If *angle-bounds* are unspecified then the default bounds are used; the default bounds lead to meshes in which all angles are between 30° and 120°. If given, the *angle-bounds* can be either a 2, 1, or 0 element list. Each of the elements in *angle-bounds* is an integer specifying an angle in degrees. If 2 elements are specified then the first is the lower bound and the second is the upper bound. If just one element is given then this is the lower bound (note that this implies an upper bound of 180° minus twice the lower bound). If an empty list is given then triangles are not checked for shape. Looser bounds produce meshes with fewer triangles. If tighter bounds are used (tighter than the default bounds) then there are 3 possible outcomes depending on the problem:

1. the result can be a high-quality mesh satisfying the given bounds,
2. the result can be a mesh satisfying the given bounds, but it includes some small elements in arbitrary places, or
3. the meshing algorithm can fail to halt.

An upper bound below about 100° is likely to lead to a failure to halt.

The *size-list*, if provided, is an alternating list of names and *size-bounds*. Each name should correspond to the name of either a boundary-edge or a region. A *size-bound* is either a number or a function of two numeric arguments that will produce a number. The size-bound following a name determines the maximum edge length that will be allowed either along the boundary with that name or within the region of that name. In other words, a triangle (or a boundary-edge) is replaced with smaller triangles (boundary-edges) if its size is greater than the corresponding size-bound. When the size-bound is a function it is evaluated at the mean of the triangle's (boundary-edge's) vertices.

The boundary and region specs are a list of specialized elements each of which is a list starting with an identifier. Currently, there are three types of elements that can appear in the boundary and region specs; the corresponding identifiers for these elements are: point, region, and boundary. The formats for these elements are given below. They have the following meanings only within the scope of a make-mesh form.

point *name* $coord_1$ ... $coord_k$ [*Special Form*]

> Creates a named point that can be used (by name) in later boundary elements. The number of coordinates must match the dimension of the embedding *space* specified as the first argument to make-mesh.

region *name* $coord_1$ ... $coord_k$ [*Special Form*]

> Specify a region to be meshed and give it a name. The boundaries (given by boundary elements) divide the embedding *space* (i.e., the first argument to make-mesh) into one or more regions. A region-to-be-meshed is specified (and named) here by giving a point

within that region. During the meshing process, all triangles within this region have this same name.

boundary *name* (&key (*closed?* nil) (*split* 0)) *type* . *description* [*Special Form*]

Insert a boundary into the mesh. Each segment of the boundary is given the specified name. At the moment, there are just two types that can appear: LINE and ARC. The description formats for these two types are given below. If :closed? is non-null then, in effect, the description is made to end at the same place it started. If :split is an integer greater than zero then each segment of the boundary is recursively split that number of times before it is inserted in the mesh. This can be used, for instance, to make the initial region have a shape closer to the actual region.

Care must be taken when an object (a hole for instance) is near a curved boundary. This is because all segments, even curved ones, are represented as straight lines during the meshing process. (Note though that when a curved segment is split, the splitting point is correctly placed on the curved path of the segment.) Curved segments that are not split into enough pieces can misbehave in the sense that a nearby object can appear to intersect the segment or even to be on the wrong side of the segment.

For LINE, an arbitrary number of points can follow LINE where each point is specified as either a name (that appeared earlier in a point element) or a list of the form (PT coord_1 ... coord_k). The boundary is formed by straight lines between adjacent points.

As an example, meshing of a simple square can be accomplished by the following use of make-mesh. It produces a mesh with small triangles in the interior of the square with slightly smaller triangles along the boundary.

```
(make-mesh ((get-euclidean-space 2) :size-list '(inside .2 edges .1))
                                    :angle-bounds '(10 120))
  (point origin 0 0)
  (region inside 0.5 0.5)
  (boundary edges (:closed? T) line
    origin (pt 1 0) (pt 1 1) (pt 0 1)))
```

For ARC, the description is an alternating list of points and arc options. The points are specified as described for LINE. The arc options are lists of keyword arguments. The valid keywords are :center, :thru, :radius, :clockwise (or :cw), and :counterclockwise (or :ccw). Only one of :center, :thru, or :radius should be specified. Also, at most one of :cw or :ccw should be given. The arc options and the two adjacent points (call them a and b) determine a circle where a and b are both on the circle. For each such circle there are two arcs between a and b. The one chosen is the one that has the specified direction (either clockwise or counterclockwise) going from a to b; the default direction is counterclockwise. The different kinds of specifications are described below. In each instance that calls for a point, the point is either a name or a list of the form (PT ...) as described for LINE

- (). If an empty list is used for the arc options this results in a straight line between the two adjacent points.

- (:center point). The circle chosen is the one through the two adjacent points that has the given center. Note that it is possible, although very undesirable, to specify an invalid center — one that is not equidistant from a and b.

- (:thru point). The circle chosen is the one through the two adjacent points (a and b) and through the specified thru-point.
- (:radius number). The circle chosen is one of the two circles through a and b that have the given radius. If number is positive then the circle chosen is the one to the left of the vector from a to b. A negatvie number indicates the other circle. If the radius is too small (e.g., zero, for instance) then the circle with a and b on the diameter is used. In this case, a warning message is printed except when number is exactly zero.

As an example, a circle can be meshed by using the following code.

```
(make-mesh ((get-euclidean-space 2) :size-list '(inside .4 outside .2))
  (point origin 0 0)
  (region inside 0 0)
  (boundary outside (:closed? t :split 3) arc
    (pt 1 0) (:center origin) (pt -1 0) (:center origin)))
```

make-mesh-from-file *stream* &key *angle-bounds size-list* [*Function*]

Create a mesh using data from a Lisp input stream. The file's format is specified in Section 14.2. The angle-bounds and size-list can be specified in the file, but the file versions are overridden if the keywords are used. See the text following the description of make-mesh for how to use the *angle-bounds* and *size-list* keywords.

write-mesh *mesh stream* [*Function*]

Write the given mesh to a Lisp output stream. The format of the resulting file is described in Section 14.2.

Assuming that the file meshRequest contains data in the proper format, the following instructions would create a mesh and write it out to the the file newMesh.

```
(let ((mesh nil))
  (with-open-file (in-file "meshRequest" :direction :input)
    (setf mesh (make-mesh-from-file in-file)))
  (with-open-file (out-file "newMesh" :direction :output)
    (write-mesh mesh out-file)))
```

13.2 Access to Portions of a Mesh

The following functions are useful for examining an existing mesh.

boundary-complex-of *mesh* [*Function*]

Return the simplicial-complex that holds the boundaries (internal and external) of the mesh.

home-of *mesh* [*Function*]

Return the embedding *space* of the mesh (*i.e.*, the first argument to make-mesh).

all-names *simplicial-complex* [*Function*]

Return a list of the different names that appear in the given simplicial complex. (Boundaries and regions of a mesh can have names; the names are used in size-lists to allow control of mesh density.)

In the example below, the first use of all-names reports all the names for boundaries of the mesh. The second use reports all the names for subregions of the mesh.

```
(all-names (boundary-complex-of mesh))
(all-names mesh)
```

Two other routines that are often useful when working with meshes are map-over-maximal-cells and map-over-cells. One common use of map-over-maximal-cells is to map over all triangles of a mesh or over all boundaries of a mesh.

```
(map-over-maximal-cells (triangle) mesh
   (print triangle))

(map-over-maximal-cells (b-segment) (boundary-complex-of mesh)
   (print b-segment))
```

Map-over-cells can be used to map over all edges of a mesh or to map over all vertices of a mesh. Note that (first vertex) is used in the second example since a 0-dimensional cell appears here as a 1-element list.

```
(map-over-cells (edge 1) mesh
   (print edge))

(map-over-cells (vertex 0) mesh
   (print (first vertex)))
```

13.2.1 Individual Components of a Mesh

The following functions are useful for examining a single component of a mesh (*e.g.*, a single triangle or a single boundary-edge). Note that several of these functions apply to simplices; thus they can be used on both triangles (2-simplices) and boundary edges (1-simplices).

locate *point mesh* [*Function*]

Return a triangle of the given mesh that has the given point in its interior or on its boundary. The point must be a point in the embedding space of the mesh.

As an example, the following code will find the triangle of mesh my-mesh that contains the origin.

```
(locate (make-point (home-of my-mesh) 0 0) my-mesh)
```

name *simplex simplicial-complex* [*Function*]

Return the name of the given simplex. Used to determine names of triangles and boundary-edges.

vertices-of *simplex* [*Method*]

Return a list of the vertices of the given simplex. Each vertex is a point in the embedding space of the mesh. Used to find the vertices of a triangle or to find the endpoints of a boundary-edge.

angles *triangle* [*Function*]

Return a list of the given triangle's angles in degrees.

simplex-size *simplex* [*Function*]

Return the length of the longest side of the given simplex. For example, this gives the length of the longest side of a mesh-triangle or the length of a boundary-edge.

13.3 Individual Meshing Functions

This section describes the direct programming interface to the mesher. In particular, it describes the functions read-mesh and refine-mesh that allow a user to refine an existing mesh. It is unlikely that a casual user will need the remaining functions unless the user plans to control the meshing process from within a program. (One might wish to do this for use with adaptive remeshing, for instance.)

read-mesh *stream* [*Function*]

Read an existing mesh from a Lisp input stream. The file format is described in Section 14.2. This function can read a mesh that was written to a file by write-mesh.

The following function can refine an existing mesh. See the text following the description of make-mesh for how to use the keywords angle-bounds and size-list.

refine-mesh *mesh* &key *angle-bounds size-list* [*Function*]

Refine an existing mesh using the specified *angle-bounds* and *size-list*.

As an example, the following code will read an existing mesh from a file, refine it (presumabley the size requested is smaller than when the initial mesh was created), and write the mesh to another file.

```
(let ((mesh nil))
  (with-open-file (in-file "oldMesh" :direction :input)
    (setf mesh (read-mesh in-file)))
  (refine-mesh mesh :size-list '(inside .1))
  (with-open-file (out-file "newMesh" :direction :output)
    (write-mesh mesh out-file)))
```

An empty mesh can be created using the following function.

create-mesh *space* [*Function*]

Create an empty mesh in the given space. Typically, the space is Euclidean 2-space.

insert-boundary *simplex mesh* &key *name* [*Method*]

Insert a 0-dimensional or a 1-dimensional simplex into a mesh as a boundary. A 0-dimensional simplex corresponds to a vertex that cannot be removed from the mesh. A 1-dimensional simplex (a boundary segment) also cannot be removed, but can be split during the meshing process. The name, if specified, can be used within a size-table in a keyword argument to refine-mesh.

insert-boundary *complex mesh* &key *name* [*Method*]

Here, complex is a simplicial-complex defined in the same space as the mesh. Each simplex in the complex is inserted into the mesh with the given name. The simplices should be dimension 1 or 0.

name-region *name point mesh* [*Function*]

Give a name to the enclosed (by boundaries) region of the mesh that contains the given point. Regions that are not named are eliminated from the mesh (in refine-mesh) before mesh refinement begins.

Note that the initial triangulation of the mesh is accomplished during the first call to name-region. Thus, this is when crossed boundary-edges are detected.

The function make-curved-segment can be used to create a curved (parametric) boundary that can be inserted into a mesh using insert-boundary. Note that straight-line boundary segments are created using make-simplex (see Section 12.4).

make-curved-segment *space param-a end-a param-b end-b generator* [*Function*]

Return 1-simplex that is a parametric curve. The *generator* must be a function that takes a number (a parameter value) and returns an element of *space*. *End-a* and *end-b* are the endpoints of the segment; *param-a* and *param-b* represent the corresponding parameter values. Knowing the parameter values for the endpoints and knowing the generator, we can, of course, generate endpoints, but these would be new endpoints and would thus not link properly with adjacent curves. Requiring the specifiation of *end-a* and *end-b* alleviates this problem. The user is reponsibe for ensuring that the parameter values and the endpoints agree.

Chapter 14

File Formats

It is often useful to store Weyl objects in disk files—as intermediate back ups in long computations, to transport mathematical objects between sites, or as a communication mechanisms with external systems that want to use objects created by Weyl. This chapter discusses the file formats used by Weyl for a variety of different mathematical objects.

14.1 Topology and Geometry

Curently, there are two file formats related to topology and geometry, one for a vertex set and one for a simplicial complex.

The `<VertexSet>` format is

```
VertexSet <SpaceDimension> <NumVerts>
<vertex_0>
<vertex_1>
...
<vertex_{NumVerts - 1}>
```

where `<SpaceDimension>` is the dimension of the embedding space and `<NumVerts>` is the number of vertices. Each `<vertex>` is a list of floating point numbers; the length of the list must match the `<SpaceDimension>`.

In effect, the VertexSet creates a set of vertices numbered (in the order given) from 0. These vertex numbers are used in the file formats for other structures such as *SimplicialComplex*.

The `<SimplicialComplex>` format is

```
SimplicialComplex <Dim> <NumSimps>
[name]
<simplex_0>
[name]
<simplex_1>
...
[name]
<simplex_{NumSimps - 1}>
```

where `<Dim>` is the maximum dimension of the simplicial complex and `<NumSimps>` is the number of maximum dimensional simplices. Note that this format is somewhat limited in that all simplices given here must have the same dimension (e.g., they must all be triangles or must all be segments).

A simplex is just a list of vertex numbers (the length of each such list is one more than <Dim>); the vertex numbers are those defined by a preceding *VertexSet*. Names are optional. If a name is given then each of the simplices between this name and the next one is assigned that name. Currently, the names are useful only when the SimplicialComplex is part of a Mesh. For a Mesh, the names are used in the specification of MeshQuality.

14.2 Meshes

There are two file formats, one to request the meshing of a region and the other to read/write meshes from/to files.

14.2.1 MeshRequest File Format

The *MeshRequest* file format is:

```
MeshRequest
<VertexSet>
<SimplicialComplex>
<Regions>
<MeshQuality>
```

<VertexSet> and <SimplicialComplex> describe the mesh boundaries and use file formats as described above. The <SimplicialComplex> describes a simplicial-complex that makes use of the vertices given in the preceding VertexSet. The <Regions> section determines (and names) the subregion(s) to be meshed. The <MeshQuality> section gives the angle bounds and the element sizes.

To mesh a planar region, the VertexSet would have <SpaceDimension> = 2 and the SimplicialComplex would have <Dim> = 1 (since each boundary is a segment—a simplex of dimension 1).

The boundary given by the SimplicialComplex divides the embedding space into one or more subregions. The <Regions> section is used to determine which of those subregions are to be meshed. A subregion is meshed if and only if there is a point given in the Regions section that lies witin that subregion.

The format for the <Regions> section is

```
Regions <NumPoints>
<name_0> <region_pt_0>
<name_1> <region_pt_1>
...
<name_{NumPoints - 1}> <region_pt_{NumPoints - 1}>
```

where <NumPoints> is the number of region points to be defined. Each <region_pt> is a list of floating point numbers, where the length of the list matches the dimension of the embedding space (the <SpaceDimension>) given in the preceding <VertexSet>. The names are used to specify the desired element sizes for the mesh in the <MeshQuality> section. During meshing, each simplex of a region has a name based on the name assigned here to the region.

The format for the <MeshQuality> section is

```
[AngleBounds <real> <real>]
[SizeTable <NumEntries>
```

```
<name_0> <real>
<name_1> <real>
...
<name_{NumEntries - 1}> <real>]
```

If AngleBounds are given then the first number is a lower bound on element angles and the second number is an upper bound. If no angle bounds are given then the resulting elements angles are between 30 and 120 degrees. The SizeTable controls the density of the mesh by specifying the maximum size (the length of the longest edge) allowed for each named region or boundary. Note that sizes in this file format are simply numbers. This is more restrictive than for meshes created from within Weyl where it is possible to specify sizes that are functions.

As an example, a request to mesh a square could look like:

```
MeshRequest
VertexSet 2 4
0.0 0.0
1.0 0.0
1.0 1.0
0.0 1.0
SimplicialComplex 1 4
name
0 1
1 2
otherName
2 3
3 0
Regions 1
interior
0.5 0.5
AngleBounds 10 150
SizeTable 3
name 0.2
otherName 0.1
interior 0.4
```

The function make-mesh-from-file is used to read a MeshRequest and to create the corresponding mesh. See Section 13.1.

14.2.2 Mesh File Format

The Mesh file format is

```
Mesh
<VertexSet>
<SimplicialComplex>
<SimplicialComplex>
```

where the first SimplicialComplex represents the boundaries of the mesh, and the second represents the elements of the mesh. Names can appear within each SimplicialComplex.

For example, a very simple mesh might be written to a file as

```
Mesh
VertexSet 2 4
```

```
0.0 0.0
1.0 0.0
1.0 1.0
0.0 1.0
SimplicialComplex 1 4
boundary
0 1
1 2
2 3
3 0
SimplicialComplex 2 2
interior
0 1 2
0 2 3
```

The functions write-mesh and read-mesh are used for writing and reading meshes. Function write-mesh is described in Section 13.1. Function read-mesh is described in Section 13.3.

14.2.3 Regions with Curved Boundaries

The current release contains only limited facilities for representing curved boundaries in files. We plan to extend this facility for the next release. Note that there are more powerful techniques available for representing curved boundaries when working from within Weyl (see make-curved-segment in Section 13.3). Within a file, curved boundaries are currently limited to circular arcs. File representations of arcs correspond roughly to the available arc representations in make-mesh (see Section 13.1).

At present, arcs can only be represented in MeshRequest files. When the resulting mesh is written out to a Mesh file, the curve information is lost. (Note though that boundaries retain their names, so it is often possible to reconstruct curve information without severe difficulty.)

Within a MeshRequest file, boundary arcs are indicated via the inclusion of some additional information in the SimplicialComplex. Recall that a line-segment boundary is represented in a file by a pair of vertex numbers. In contrast, an arc is represented by a vertex number, a parenthesized list of arc arguments, and another vertex number. The two vertex numbers determine the endpoints of the arc, e_1 and e_2. The possible arc arguments are

- (c <number>) The counterclockwise arc from e_1 to e_2 that is part of the circle through e_1 and e_2 and centered at the vertex with vertex number <number>. It is possible (but not a good a idea) to specify a center that is not equidistant from e_1 and e_2.
- (t <number>) The counterclockwise arc from e_1 to e_2 that is part of the circle through e_1, e_2, and the vertex with vertex number <number>.
- (r <number>) The counterclockwise arc from e_1 to e_2 that is part of the circle with radius <number> through e_1 and e_2 and centered to the left of the ray from e_1 to e_2. If the radius is negative then the circle is centered to the right of the ray from e_1 to e_2. A warning is issued if the absolute value of the radius is less than half the distance from e_1 to e_2; the arc that results in this case is a half-circle.

It is also possible to specify clockwise arcs by using c-, t-, or r- in place of c, t, or r, respectively (although equivalent effects can be achieved by reversing the order of the two endpoints).

As an example, the following MeshRequest file describes a circle of radius one centered at the origin. Each of the three possible arc arguments have been used for part of the circle.

```
MeshRequest
VertexSet 2 5
0 0
1 0
0 1
-1 0
0 -1
SimplicialComplex 1 4
boundary
1 (c 0) 2
2 (r 1) 3
3 (t 2) 4
1 (c- 0) 4
Regions 1
interior 0 0
AngleBounds 10 120
SizeTable 2
interior 0.4
boundary 0.2
```

Bibliography

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

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

[4] R. D. Jenks, *11 keys to new Scratchpad*, in EUROSAM '84, J. Fitch, ed., vol. 174 of Lecture Notes In Computer Science, Berlin-Heidelberg-New York, 1984, Springer-Verlag, pp. 123–147.

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

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

[7] 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.

[8] G. L. Steele Jr., *Common Lisp, The Language*, Digital Press, Burlington, MA, second ed., 1990.

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

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