6.905 Final Project:
Generic Abstract Mathematics

Amyas Chew Lynn Chua Yongquan Lu

May 13, 2015

1 Introduction

The use of computers in mathematical work has increased tremendously from the mechanical calculation aids of the era before the analytical engine, to the current petaFLOPS-scale supercomputers used for the notoriously intensive simulations required in various computational sciences. In these examples, computers serve primarily to take the menial computational load off the users, allowing them to devote time to more productive uses. However, the *role* of computers has also been expanding as well, finding use not just for plain calculations but also in the development of abstract mathematics. There are now computational proof-checkers and even automated theorem provers that attempt to solve sufficiently formalized problems with little human direction. Some of these have been used to generate proofs, for problems like the four-coloring theorem, that are controversially intractible to humans due to their complexity.

In the ideal case, this trend continues until we develop programs that completely take over all the tedious aspects of mathematical work, leaving only the boundary layer where human creativity is still necessary to extend the body of mathematical knowledge. With this overall goal in mind, we wish to implement a system which has a natural structural interpretation, so humans can interact with it in the same way they think about mathematical ideas in real life, and which is also flexible enough to be extended in pace with the introduction of new mathematical constructs.

Our program is an implementation of mathematical constructs written in scheme, which has the advantage of having a simple but flexible syntax. The program consists of constructors and methods with natural analogues to concepts familiar to the mathematical community, built on top of a common infrastructure that is simple enough to be easily used to accommodate new mathematical concepts.

Although the intention for the program was primarily to act as the front end for a user to construct mathematical entities, there are potential use cases that it can potentially have if the appropriate infrastructure is implemented. For example, the program on its own can already serve as a simple convenient tool for keeping track of large or complex mathematical entities. With additional code for converting datatypes, the program can become a frontend for an existing computer algebra system (CAS), calling the latter whenever existing methods are applicable, but still providing the flexibility of building custom mathematical structures that are not implemented by the CAS. In a similar way, the program can also serve as a frontend extension to existing proof assistants or checkers, providing the flexibility whenever needed.

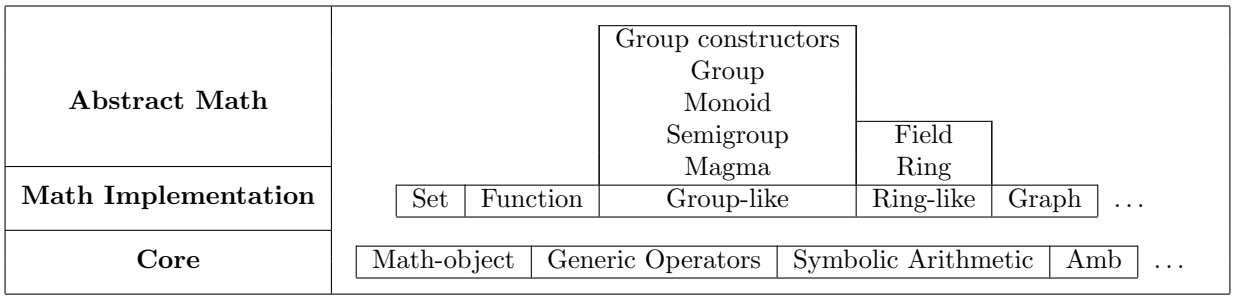
2 Overview of program structure

The program was written as 3 main layers of implementation:

1. The core layer consists of the definition and abstraction for the common record type used to implement generic mathematical entities, as well as the various dependencies of the code.

2. The math implementation layer contains the datatype definitions and methods for various basic mathematical structures stored as versions of the common record type.

3. The abstract math layer consists of specialized constructors in multiple levels of abstraction that wrap the implementation of basic mathematical structures, in order to provide the methods analogous to actual mathematical concepts. There is no clear demarcation separating this from the math implementation layer.



3 Core layer

The core layer contains the abstraction for the "math-object" record type that is used to implement the various mathematical structures in the program. It also contains some dependencies on symbolic infrastructure that is used throughout the program to provide flexibility for further extensions. Finally, the core layer also contains the code for miscellaneous infrastructure used throughout the program, like generic comparators or mathematical entities that only need very simple implementations.

3.1 Math-object

The `math-object` record type is defined using:

```
(define-record-type math-object
  (make-math-object structure data properties)
  math-object?
  (structure math-object-structure)
  (data math-object-data)
  (properties math-object-properties set-math-object-properties!))
```

It consists of three fields: the `data` and `properties` fields contain association lists (alists), while the `structure` field contains a symbol. The difference between the data and `properties` alists is that the content of the `data` alist is supposed to contribute to the identity of the math-object, whereas the `properties` alist stores auxilliary information about the math-object, and is used to memoize mathematical properties about the math-object that were computed the first time they were queried. The value of the `structure` field is used to specify how the content of the data alist should be interpreted, and also to help prevent naming conflicts in the event that two different mathematical structures have similarly named components.

By design choice, the math-object is immutable except for its `properties` alist, since any changes to the data of the math-object can make its memoized properties incorrect, and some properties are too expensive to recompute very often. Therefore, there are both getter and setter methods for the properties of the math-object, but only getter methods for its data. Examples of the use of math-objects will be mentioned in the discussion about the math implementation layer.

3.2 Dependencies

The main dependencies used in the code are discussed as follows.

3.2.1 Generic Operators

Generic operators play an essential part in providing the required flexibility of the program. By dispatching procedures based on the predicates satisfied by the arguments, they allow generic comparators to be used to efficiently sort and store elements of almost any type in a set. They also allow extensions to the methods on mathematical structures, such as allowing a partial extension for infinite sets.

3.2.2 Symbolic Arithmetic

With generic operators, it is easy to implement symbolic arithmetic which is often needed for exact computation in mathematics. This opens up the use of radicals, complex numbers, and polynomials as primitive objects to build mathematical structures with.

3.3 Domain extensions

As we intend for these abstract math objects to be realized over multiple domains, a priority was to build sufficiently many diverse domains to test these constructions on. Of course, each of these could have been implemented as full math-objects themselves, but simple implementations suffice for their use as examples to write tests on. They are discussed briefly below.

3.3.1 Permutations

Permutations are represented as lists of integers of length n; the `permutation?` predicate checks that the elements of the list are distinct and make up the range $1 \dots n$. In this formulation (41320) represents the permutation that sends 0 to 4, 1 to 1, 2 to 3, 3 to 2 and 4 to 0. Note that this is not the same as the more common cycle notation, where this permutation would be represented as (04)(23)(1). Our choice was a conscious decision to simplify implementation of permutation composition, but in practice translating between the two representations for display is easy.

```
(permutation? '(4 1 3 2 0))
;Value: #t

(permutation? '(3 1 a 1))
;Value: #f
```

The `compose-permutation` procedure takes two permutations σ_1, σ_2 and outputs the composition of the two. For each element $i \in \{1 \dots n\}$, it computes $\sigma_2(\sigma_1(i))$ and writes it to the i^{th} position of the output.

```
(compose-permutation '(1 3 2 0 4) '(4 1 2 3 0))
;Value 14: (4 3 2 0 1)
```

3.3.2 Matrices

Matrices are implemented primitively to support group and ring constructions over rings. The matrix constructor `make-matrix` takes a list of list of elements, checks that it is well-formed (each row is the same length) and returns a tagged vector with the number of rows and columns for easy access.

```
(make-matrix '((2 1) (3 4)))
;Value 17: #(matrix 2 2 ((2 1) (3 4)))

(make-matrix '((a b) (c d e)))
;Argument is not a valid matrix.
```

Addition, subtraction and multiplication are implemented generically, so that elements can be in any domain ($\mathbb{Z}_p$, for example).

```
(define a (make-matrix '((1 2) (5 6))))
(define b (make-matrix '((4 1) (2 2))))
(define c (make-matrix '((1 5 2) (3 3 4))))

(+ a b)
;Value: #(matrix 2 2 ((5 3) (7 8)))
```

$$\begin{pmatrix}1 & 2\\5 & 6\end{pmatrix}+\begin{pmatrix}4 & 1\\2 & 2\end{pmatrix}=\begin{pmatrix}5 & 3\\7 & 8\end{pmatrix}$$

```
(+ a c)
;Not matrices of same dimensions

(* a b)
;Value: #(matrix 2 2 ((8 5) (32 17)))
```

$$\begin{pmatrix}1 & 2\\5 & 6\end{pmatrix}\cdot\begin{pmatrix}4 & 1\\2 & 2\end{pmatrix}=\begin{pmatrix}8 & 5\\32 & 17\end{pmatrix}$$

```
(* a c)
;Value: #(matrix 2 3 ((7 11 10) (23 43 34)))
```

$$\begin{pmatrix}1 & 2\\5 & 6\end{pmatrix}\cdot\begin{pmatrix}1 & 5 & 2\\3 & 3 & 4\end{pmatrix}=\begin{pmatrix}7 & 11 & 10\\23 & 43 & 34\end{pmatrix}$$

```
(* c a)
;Not matrices of compatible dimensions
```

3.3.3 Radicals

We would like to perform exact computation with radicals, for example, in order to construct groups as matrices over radicals. Existing procedures within Scheme like `sqrt` and `expt` are inexact, so we built a simple framework that allows linear combinations of integer square roots to be represented as a coefficient list, which we call a *root list*. Each list is a list of cons pairs `(a . b)` representing $a\sqrt{b}$. For example, `((5 . 1) (-1 . 2) (4 . 5))` represents the sum $5-1\sqrt{2}+4\sqrt{5}$. A `simplify` procedure extracts square factors out of each term and then collects and sorts like terms.

```
(simplify '((2 . 49/12)))
;Value: ((7 . 1/3))
```

$$2\sqrt{\tfrac{49}{12}}=7\sqrt{\tfrac{1}{3}}$$

```
(simplify '((2 . 4) (1 . 18) (3. 12)))
;Value: ((4 . 1) (3 . 2) (6 . 3))
```

$$2\sqrt{4}+\sqrt{18}+3\sqrt{12}=4+3\sqrt{2}+6\sqrt{3}$$

Using generic arithmetic, we are also able to support addition, subtraction and multiplication over root lists. Addition and subtraction are performed component wise, while multiplication is implemented by distributing over both root lists, collecting like terms and simplifying. Division is implemented by incrementally rationalizing the denominator.

```
(+ '((1 . 3) (2 . 9))
   '((1 . 5) (7 . 12)))
;Value: ((6 . 1) (15 . 3) (1 . 5))
```

$$(\sqrt{5}+2\sqrt{9})+(\sqrt{5}+7\sqrt{12})=6+15\sqrt{3}+\sqrt{5}$$

```
(* '((1 . 2) (1 . 3))
   '((1 . 1) (2 . 3)))
;Value: ((6 . 1) (1 . 2) (1 . 3) (2 . 6))
```

$$(\sqrt{2}+\sqrt{3})\cdot(1+2\sqrt{3})=6+\sqrt{2}+\sqrt{3}+2\sqrt{6}$$

```
(/ '((1 . 2) (1 . 3))
   '((2 . 1) (1 . 2) (-1 . 3)))
;Value: ((1/23 . 1) (14/23 . 2) (10/23 . 3) (2/23 . 6))
```

$$\tfrac{\sqrt{2}+\sqrt{3}}{2+\sqrt{2}-\sqrt{3}}=\tfrac{1}{23}+\tfrac{14}{23}\sqrt{2}+\tfrac{10}{23}\sqrt{3}+\tfrac{2}{23}\sqrt{6}$$

This system currently works well for square roots, but we can conceive of a system where each tagged list has another parameter to support n^{th} roots as well. A simple extension to `simplify` can use $8 = 2^3$ to reduce $\sqrt[3]{56}$ to $2\sqrt[3]{7}$. We chose, however, not to implement such an abstraction as no straightforward analogue to rationalizing the denominator for division exists.

4 Math Implementation

4.1 Sets

Sets were the one of the most basic mathematical structures that can easily be built using math-objects. In the program, it was easy to implement explicit sets that store information about its elements in a weighted-balanced tree, which already had methods written specifically with sets in mind. In order to allow elements to be almost anything, there was a need for a generic comparator that works for anything that is allowed in a set. This was done using generic operators to define the `expr<?` procedure, which dispatches to various type-specific comparators like `<` for numbers and `symbol<?` for symbols. Comparison between different types was done with a preloaded type priority order.

Generic operators were also used to implement the various set methods like set unions, cardinality calculations, and set membership. This allowed the possibility of a different kind of set implementation to use the same set abstraction. For example, explicit sets were defined partially by the code

```
;;; Primitive constructor of explicit finite sets
(define (make-set . element-list)
  (list->set element-list))

;;; Convert from list of elements
(define (list->set element-list)
  (let* ((inclusion-alist
          (map (lambda (x) (cons x #t))
               element-list))
         (element-wt-tree
          (alist->wt-tree explicit-set-wt-tree-type
                          inclusion-alist)))
    (wt-tree->set element-wt-tree)))

;;; Convert from weight-balanced tree of elements
(define (wt-tree->set element-wt-tree)
  (make-math-object 'set
                    (list (list 'element-wt-tree element-wt-tree))
                    (list (list 'explicit-set? #t)
                          (list 'finite? #t))))
```

which stores the implementation type as an `'explicit-set?` property of the set. Since explicit sets are always finite, it is possible to store the `'finite?` property as well. On the other hand, a partial implementation of implicit sets might be implemented by code like

```
;;; The set of integers
(define set/integers
  (make-math-object 'set
                    (list (list 'membership-predicate-name 'integer?))
                    (list (list 'explicit-set? #f)
                          (list 'finite? #f))))
```

which stores data in a different way and thus requires a different procedure for set methods like checking membership, constructing set unions or calculating the number of elements.

4.2 Group-like objects

We constructed a group-like datatype, to represent mathematical objects built from a set and a binary operation. There are no constraints imposed on the set and the operation:

```
;;; Primitive constructor
;;; operation must be binary operation between elements of the set
(define (make-group-like set operation)
  (if (set? set)
      (make-math-object 'group-like
                        (list (list 'underlying-set set)
                              (list 'operation operation))
                        '())
      (error "Not a set:" set)))
```

Instead, the various interesting properties of group-like objects are written as predicates that check for a memoized value, returning it if there is one, but computing and storing the value otherwise (usually on its first call). For example, the closure property predicate is implemented as

```
;;; Test closure of operation
(define (group-like/closed-operation? group-like)
  ;;Check if closure is satisfied if not already computed
  (if (has-math-property? group-like 'closed-operation?)
      ;;property set, so return its value
      (get-math-property group-like 'closed-operation?)
      ;;property not set, so check its value and return it
      (let* ((set (group-like/underlying-set group-like))
             (operation (group-like/operation group-like))
             (result
              ;;in the future, this should be replaced with a function image check
              (for-all x set
                       (for-all y set
                                (set/member? (operation x y) set)))))
        ;;Set closure property to result
        (set-math-property! group-like 'closed-operation? result)
        result)))
```

The named versions of group-like objects that are interesting in math include groups, semigroups, monoids etc. In our implementation, they can be implemented as a hierachy of abstractions over group-like objects, where a group is a special case of a monoid, a monoid is a type of semigroup and a semigroup is a type of magma. Many operations on groups can simply wrap operations on monoids, which can further wrap operations down the hierarchy. We briefly explain each named group-like type as follows.

4.2.1 Magmas

A magma consists of a set and a closed binary operation. We construct this by creating a group-like object and checking that the binary operation is closed.

4.2.2 Semigroups

A semigroup is a magma where the binary operation is associative. We construct this by creating a group-like object, checking that it is a magma and checking that the operation is associative. We implement operations on semigroups by building on top of the operations for magmas.

4.2.3 Monoids

A monoid is a semigroup with an identity element. We construct this by creating a group-like object, checking that it is a semigroup and checking that it has an identity element. We implement operations on monoids by referencing operations on semigroups.

4.2.4 Groups

A group is a monoid where each element has an inverse. We construct this by creating a group-like object, checking that it is a monoid and checking that each element is invertible. We implemented specialized tools to work with groups; this is elaborated on in Section 5.

4.3 Ring-like objects

We constructed a ring-like datatype, to represent mathematical objects built from a set, an additive operation and a multiplicative operation. This includes rings, semirings, domains and fields.

4.3.1 Rings

To construct a ring, we create a ring-like object and check that the ring axioms are satisfied. Namely, we check that the ring-like object is an abelian group under addition, a monoid under multiplication, and that multiplication is distributive with respect to addition. We also implemented basic operations on rings, including finding the units, idempotents and zero divisors of a ring.

4.3.2 Fields

Fields are nonzero rings where every nonzero element is a unit. We constructed fields by creating a ring and checking that these properties are satisfied. We also implemented basic operations on fields, such as finding the additive and multiplicative inverses of an element in the field. A commonly used example of a field is the finite field of prime order p, which we construct by taking $\{0, 1, ..., p\}$ as the set of elements, together with the operations of addition and multiplication modulo p.

4.4 Graphs

Apart from abstract algebraic objects, we also construct a datatype to represent graphs, by storing the vertices as a set and the edges as a list of sets, where each set consists of elements from the set of vertices. We implemented basic operations on graphs, including checking if the graph is simple, regular or complete, and finding the degree of an element in the graph. Our implementation also allows us to construct hypergraphs, which are a generalization of graphs such that each edge is an arbitrarily-sized subset of the vertices. Graphs are a special case of hypergraphs, where each edge contains exactly two vertices.

5 Specialized Constructors

To demonstrate the utility of this framework, we built further higher-order abstractions to better manipulate and construct groups in different settings. This was done specifically in a group context, but this proof-of-concept demonstrates that these constructions are possible for rings and other settings too.

5.1 Generated groups

Instead of constructing a group (or monoid, or semigroup) out of an explicit set and operation, we can rely on the closure axiom and let Scheme generate the set out of a list of generators.

Within `group-from-generators` is an internal recursive helper function that takes a list of elements so far and pairs it still needs to test. It then `cdr`s down the latter, each time producing an element, testing if

it is a member of its list of elements so far. If so, the helper calls itself with the same list of elements and the remaining pairs to test; if not, it adds the newly found element to the list of elements so far, and appends the $2n$ new pairs to test to the remaining pairs to test before calling itself. `group-from-generators` initializes this with the list of elements and all n^2 initial pairs. This operation, as expected, runs in $O(|G|^2)$ time, where $|G|$ is the order of the initially undetermined group.

```
(define g1 (group-from-generators '(5) (lambda (x y) (modulo (+ x y) 12))))

(group/elements g1)
;Value: (0 1 2 3 4 5 6 7 8 9 10 11)

(define g2 (group-from-generators '(4) (lambda (x y) (modulo (+ x y) 12))))

(group/elements g2)
;Value: (0 4 8)
```

It is the user's responsibility to pass this procedure a valid set of generators and an operation; the procedure will not terminate and overflow if no closure can be found over the given set of generators.

5.2 Parametrized groups

Certain infinite families of groups occur over and over again, so we built special-purpose constructor wrappers to abstract these groups.

5.2.1 Cyclic groups

A cyclic group of order n may be represented as the elements $\{0 \dots n-1\}$ armed with the operation $+ \mod n$.

```
(define c7 (make-cyclic 7))

(group/elements c7)
;Value 15: (0 1 2 3 4 5 6)

((group/operation c7) 3 5)
;Value: 1
```

5.2.2 Dihedral groups

There are many equivalent ways to represent the dihedral group D_n (of size $2n$), but for simplicity our implementation relies on the presentation $< p, q | p^n = q^2 = 1, qpq^{-1} = p^{-1} >$. Elements of D_n are represented as tuples (x, y) where $x \in \{0 \dots n-1\}$ and $y \in \{0, 1\}$, and the operation $\cdot$ is defined as follows:

$$(x_1, y_1) \cdot (x_2, y_2) = (x_1 + x_2 \cdot (-1)^{y_1} \mod n, y_1 + y_2 \mod 2)$$

```
(define d4 (make-dihedral 4))
;Value: d4

(group/elements d4)
;Value: ((0 0) (0 1) (1 0) (1 1) (2 0) (2 1) (3 0) (3 1))

((group/operation d4) '(0 1) '(3 0))
;Value: (1 1)

((group/operation d4) '(3 0) '(0 1))
;Value: (3 1)
```

5.2.3 Permutation groups

Recall that in section 3.3.1 we have already built up the underlying infrastructure for composing and manipulating permutations. Therefore it is easy to construct S_n as the set of all $n!$ permutations on $\{1 \dots n\}$ with the operation `compose-permutation`.

```
(group/elements (make-symmetric 3))
;Value: ((0 1 2) (0 2 1) (1 0 2) (1 2 0) (2 0 1) (2 1 0))
```

The alternating group A_n, defined as the index-2 subgroup of S_n consisting of all even permutations, is a bit harder to explicitly construct. We sacrifice computational runtime for simplicity and define it as the closure of all permutations p_i from $2 \leq i \leq n-1$, where p_i switches 1 and 2 and switches i and $i+1$ (see section 5.1).

```
(group/order (make-alternating 4))
;Value: 12

(group/order (make-alternating 5))
;Value: 60
```

5.3 Cartesian products

We implemented `group/cart-pdt` to take the Cartesian product of two groups; this follows naturally from taking the Cartesian product of the sets and their operations.

```
(define c2xc3 (group/cart-pdt (make-cyclic 2)
                  (make-cyclic 3)))
;Value: c2xc3

(group/order c2xc3)
;Value: 6

((group/operation c2xc3) '(0 2) '(1 1))
;Value: (1 0)
```

5.4 Constructing isomorphisms

We can say that two finite groups G_1 and G_2 are isomorphic iff there exists a bijection ψ between them that preserves the group operation, i.e. $\forall g, g' \in G_1 \ \psi(g)\psi(g') = \psi(gg')$. We construct the procedure `group/isomorphic?` that takes in two groups and constructs an isomorphism between them if possible, and returning false if not.

A naive algorithm would check all $n!$ possible bijections, but our implementation does better by taking into account additional structure within the group. A utility procedure, `group/order-alist` computes the order of each element (the minimum power needed to raise the element to to get back the identity). `invert-alist` flips around the keys and values of the alist and `alist-key-count` collects the number of elements with a particular order, which we can treat as a kind of signature.

```
(group/order-alist c6)
;Value: ((0 1) (1 6) (2 3) (3 2) (4 3) (5 6))

(invert-alist (group/order-alist c6))
;Value: ((1 (0)) (2 (3)) (3 (4 2)) (6 (5 1)))

(alist-key-count (group/order-alist c6))
;Value: ((1 1) (2 1) (3 2) (6 2))
```

For instance, in this case we know that C_6 and D_3 are not isomorphic because they have different number of elements of a given order.

```
(alist-key-count (group/order-alist d3))
;Value 53: ((1 1) (2 3) (3 2))

(group/isomorphic? c6 d3)
;Value: #f
```

We can, however, build a group isomorphic to D_3 out of 2D matrices that correspond to the symmetries of an equilateral triangle. Here we use the root-list abstraction for radicals developed in section 3.3.3.

```
(define g (group-from-generators
          (list (make-matrix '((((-1/2 . 1)) ((-1/2 . 3)))
                 (((1/2 . 3)) ((-1/2 . 1)))))
          (make-matrix '((((1 . 1)) ())
                 (() ((-1 . 1))))))
        *))

(alist-key-count (group/order-alist g))
;Value: ((1 1) (2 3) (3 2))

(group/isomorphic? g (make-dihedral 3))
;Value: (#(matrix 2 2 ((((1 . 1)) ()) (() ((1 . 1))))) (0 0)
         #(matrix 2 2 ((((1 . 1)) ()) (() ((-1 . 1))))) (0 1)
         #(matrix 2 2 ((((-1/2 . 1)) ((1/2 . 3))) (((1/2 . 3)) ((1/2 . 1))))) (1 1)
         #(matrix 2 2 ((((-1/2 . 1)) ((-1/2 . 3))) (((-1/2 . 3)) ((1/2 . 1))))) (2 1)
         #(matrix 2 2 ((((-1/2 . 1)) ((1/2 . 3))) (((-1/2 . 3)) ((-1/2 . 1))))) (2 0)
         #(matrix 2 2 ((((-1/2 . 1)) ((-1/2 . 3))) (((1/2 . 3)) ((-1/2 . 1))))) (1 0))
```

We implement `group/isomorphic?` behind the scenes with amb and require's. For performance, amb is constrained to only generate mappings that map elements of the same order to each other (cutting down the search from $6! = 720$ to a puny $1! \cdot 3! \cdot 2! = 12$ possibilities). Each mapping is then tested for validity over all pairs of elements; if it fails, backtracking happens automatically behind the scenes.

This scheme can easily be extended to enumerate all isomorphisms / automorphisms by triggering a backtrack even when a successful candidate is found, and perhaps to find surjective / injective homomorphisms between a group and a subgroup of another group.

6 Code and Sources

The version of the code at the time of submission can be obtained from the "6.905" branch of the repository at `https://github.com/amyascwk/math-scm`. The "master" branch contains the latest version of the code.

The generic arithmetic architecture used for domain extensions was adapted from problem set 3 of the Spring'15 6.905 class in MIT, while the amb operator and its associated backtracking schemes were adapted from problem set 7.

7 Further Work

One of the major avenues for further work would be to build a logic system that allows the user to introduce mathematical statements as axioms. With an automatic deduction system in place, it is possible to implement mathematical objects that can only be indirectly reasoned about, like infinite sets. This would be a task of great priority, since many interesting mathematical structures are infinite.

The examples shown here were built with special purpose domain extensions (permutations, matrices and radicals). However, extensive work of this nature already exists in the form of scmutils and other libraries, which our platform could integrate with far more generality without reinventing the wheel (complex numbers, vectors, polynomials, ...).

We also hope to develop more extensive higher-level abstractions on top of more math objects like rings and graphs in the style of groups, with specialized constructors and isomorphism searchers.