6.905 Final Project:
Generic Abstract Mathematics

Amyas Chew Lynn Chua Yongquan Lu

May 13, 2015

1 Introduction

- Motivation - Role of computers in math - Computer Algebra Systems (CAS) take some computation load off user - Examples of proofs using computers: 4-color theorem, etc etc - Significance of project - Scheme allows emphasis on ideas rather than programming syntax - Want a flexible system allowing users to make own constructs - Free software and extensible so users can extend easily - can keep in pace with real world progress in math community

- Potential use cases - Convenience tool for keeping track of abstract mathematical data - Simple scripting language for a CAS backend - Possible basis for proof-assistants, proof-checkers, etc - ??

2 Overview

- Overall structure - written using mit-scheme - Different levels of implementation: - Core - Math implementation - Extensions to base math

3 Underlying structure

3.1 Adapted infrastructure

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. Each of these are discussed briefly below.

3.1.1 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)
```

3.1.2 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
```

3.1.3 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))
```

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

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.

3.2 Math-object Datatype

- explanation of choice of structure

4 Math Implementation

4.1 Sets

- sets stuff

4.2 Group-like objects

- group-like stuff

4.3 Ring-like objects

- ring-like stuff

4.4 Hypergraph-like

- hypergraph-like stuff

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 $\{0 \ldots n-1\}$ 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 $x \in \{0 \ldots n-1\}$ 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 3.1.1 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 3.1.3.

```
(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 $10! = 3628800$ to a much more reasonable $1! \cdot 4! \cdot 1! \cdot 4! = 576$ 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

The latest version of the code can be obtained from `https://github.com/amyascwk/math-scm`.