Programming with Scheme Values and Algebraic Laws

Norman Ramsey

Spring 2019

1 Introduction

This handout goes deeper into programming with algebraic laws, in three parts:

- You get the bare bones of programming with μScheme values, including lists and S-expressions, using proof systems and algebraic laws. You see the same techniques as before, but with new forms of data.
- You see a wider world of algebraic laws, distinguishing *algorithmic* laws from non-algorithmic *properties*. When coding from scratch, you must learn to make your laws algorithmic.
- You see some issues that you may have when working more broadly with algebraic laws.

2 Proof systems and laws for μScheme data

The first section of this handout revisits the ideas in the natural-number case study, but for some common forms of μScheme data.

Proof systems for μScheme data

As noted in Figure 2.1 on page 95 of *Programming Languages: Build, Prove, and Compare*, a μScheme value is either an atom, a function, or a `cons` cell. A "fully general S-expression" is any of these except a function. We could write a proof system like this:

$$\frac{\vdash v \texttt{ symbol}}{\vdash v \texttt{ gsx}} \qquad \frac{\vdash n \texttt{ number}}{\vdash n \texttt{ gsx}} \qquad \frac{}{\vdash \texttt{\#t gsx}}$$

$$\frac{}{\vdash \texttt{\#f gsx}} \qquad \frac{}{\vdash \texttt{'() gsx}} \qquad \frac{\vdash v_1 \texttt{ gsx} \qquad \vdash v_2 \texttt{ gsx}}{\vdash \texttt{(cons } v_1 \ v_2\texttt{) gsx}}$$

We could define "list of A" using the proof system from section 2.6, which starts on page 116 of the textbook:

$$\textsc{EmptyList}\ \frac{}{\texttt{'()} \in \mathit{LIST}(A)}$$

$$\textsc{ConsList}\ \frac{a \in A \qquad \mathit{as} \in \mathit{LIST}(A)}{\texttt{(cons } a \ \mathit{as}\texttt{)} \in \mathit{LIST}(A)}$$

On your homework I'll ask you to define "*nonempty* list of A's."

We could define "ordinary" S-expression using just ideas 1 and 2 from Figure 2.1 on page 95. The notation of that last rule gets a little dodgy:

$$\frac{\vdash n \texttt{ number}}{\vdash n \texttt{ osx}} \qquad \frac{}{\vdash \texttt{\#t osx}} \qquad \frac{}{\vdash \texttt{\#f osx}}$$

$$\frac{\mathit{vs} \in \mathit{LIST}(\texttt{osx})}{\vdash \mathit{vs} \texttt{ osx}}$$

Writing $\mathit{LIST}(\texttt{osx})$ is flagrant abuse of notation. There's a better way.

An informal alternative

Proof systems are great for describing the structure of natural numbers, as well as more complex structures like computations. But for describing simpler data structures, we don't need the expressive power of proof systems, and it's often difficult to come up with good judgment forms—that's where we got into trouble above. As an alternative, we can write an inductive definition informally. We name the set we're trying to define, and we list all the ways that data in the set could be formed. Examples follows.

A *fully general S-expression* is one of the following:

- A symbol
- A number
- A Boolean
- The empty list `'()`
- `(cons` v_1 v_2`)`, where v_1 and v_2 are fully general S-expressions

Data	*Left-hand side*	
Fully general	(*f a*)	= ···, where *a* is an atom
S-expression	(*f* (cons *y z*))	= ···
List of *A*	(*f* '())	= ···
	(*f* (cons *y ys*))	= ···
Ordinary	(*f a*)	= ···, where *a* is an atom
S-expression	(*f* (cons *y ys*))	= ···
Also	(*f* '())	= ···
Ordinary	(*f a*)	= ···, where *a* is an atom but not '()
S-expression	(*f* (cons *y ys*))	= ···
Nonempty list of *A*	(homework)	
	···	

Table 1: Forms of laws for a one-argument function *f*

A *list of A's* is one of the following:

- The empty list '()
- (cons *a as*), where *a* is an *A* and *as* is a list of *A*'s

An *ordinary S-expression* is one of the following:

- A symbol
- A number
- A Boolean
- A list of ordinary S-expressions

It is frequently useful to expand that last bullet. It is equally true that an ordinary S-expression is one of the following:

- A symbol
- A number
- A Boolean
- The empty list '()
- (cons *v vs*), where *v* is an ordinary S-expression and *vs* is a list of ordinary S-expressions

Writing algebraic laws and code

As with natural numbers, the forms of data determine the left-hand sides of algebraic laws, which determine the case analysis that goes into your code. Table 1 shows what laws will look like for a one-argument function *f*. Table 2 shows how to identify forms of data and how to extract the parts from which data is formed.

3 More uses of algebraic laws

The first assignment introduces you to algebraic laws purely as a tool for designing functions that you code from scratch. The tool works even better for lists and S-expressions than it works for natural numbers. For example, here are laws that define a function for asking how many elements there are in a list:

```
(length '())          == 0
(length (cons x xs)) == (+ 1 (length xs))
```

A set of laws like this is called *algorithmic*: the laws specify the algorithm for `length`, and they are very close to an implementation.

More generally, we can write algebraic laws for any property that we believe is true. For example, if we append two lists, the length of the result is the sum of the lengths of the arguments:

```
(length (append xs ys)) ==
         (+ (length xs) (length ys))
```

This law is *not* algorithmic—a law like this is called a *property*. Let's go deeper into the distinction.

Understanding and using algorithmic laws

You can recognize an algorithmic set of laws by these hallmarks:

- Each left-hand side is a function to be defined, applied to one or more arguments, where each argument is either a *variable* or a *form of data*. In the `length` example, both '() and (cons x xs) are forms of data.
- In an algorithmic set of laws, each law is *mutually exclusive* with the others. That is, given any particular input, at most one law applies. Mutual exclusion is accomplished either using mutually exclusive forms of data, like '() and

Data	*Form of argument x or xs*	*Test for form*	*Parts argument is formed from*

Fully general S-expression	a
	`(cons` y z`)`
		`(atom?` x`)`
		`(not (atom?` x`))`
		or `(pair?` x`)`
			$a = x$
			$y =$ `(car` x`)` $z =$ `(cdr` x`)`

List of A	`'()` `(cons` y ys`)`	`(null?` xs`)` `(not (null?` xs`))`	$y =$ `(car` xs`)` $ys =$ `(cdr` xs`)`

Ordinary S-expression	a
	`(cons` y ys`)`
		`(atom?` x`)`
		`(not (atom?` x`))`
		or `(pair?` x`)`
			$a = x$
			$y =$ `(car` x`)` $ys =$ `(cdr` x`)`

Also Ordinary S-expression	`'()`
	a
	`(cons` y ys`)`
		`(null?` x`)`
		`(atom?` x`)`
		`(not (atom?` x`))`
			$a = x$
			$y =$ `(car` x`)` $ys =$ `(cdr` x`)`

Nonempty list of A	$\cdots$ (homework) $\cdots$		

Table 2: Identifying forms and extracting parts

`(cons x xs)`, or by using mutually exclusive side conditions.

There are rare cases in which algorithmic laws allow some *overlap*: inputs for which more than one law could apply. In cases of overlap, *all applicable right-hand sides must produce the same result*. These cases are sufficiently rare that I don't have an example.

- Collectively, an algorithmic set of laws *accounts for every input that is permitted by a function's contract*. If an input is permissible, there must be a law that applies.
- In every recursive call on every right-hand side, *some input is getting smaller*.

Algorithmic laws are used for these purposes:

- Algorithmic laws are used primarily *to design and implement functions*.
- Algorithmic laws can also be used *to test functions*.

Understanding and using properties

Technically, every law in an algorithmic law is also a property. But you can but sure to recognize a non-algorithmic property by these hallmarks:

- On a left-hand side, *a function is applied to the result of another function*. For example, in the `length` property, `length` is applied to the result of `append`.
- Properties might not be mutually exclusive, and they needn't account for every permissible input.

Properties have many more uses than algebraic laws, including these purposes:

- Properties are used for *testing*. Substitute a permissible value for each variable in the property, and check that equality holds. For example, here's a property we use to test arithmetic in Smalltalk:

  ```
  (* 2 n) == (+ n n)
  ```

  We can test this property with any natural number `n`.

  Here's a property about lists that is useful only for testing:

  ```
  (permutation? (cons x (cons y zs))
                (cons y (cons x zs))) == #t
  ```

  Good tooling for programming languages frequently includes *random, automated, property-based testing* based on substituting randomly generated values for variables in properties.

- Properties are used for *refactoring*, which means rewriting code to improve its structure, without changing its semantics. A good example is code simplification. Many of the properties found in section 2.4 of the textbook, like this `append-cons` law, can be used to simplify code:

  ```
  (append (cons x '()) xs) == (cons x xs)
  ```

- Properties are used for *code improvement*, which means rewriting code to improve its performance,

without changing its semantics. (Code improvement is often called "optimization.") Some of the properties found in section 2.4 of the textbook, like this `append-append` law, can be used to improve performance:

```
(append (append xs ys) zs) ==
               (append xs (append ys zs))
```

- Properties are used for *specification*, especially of abstract data types. Programmers may use properties to say how an abstraction behaves without saying how it is implemented. Here's a typical property from an abstraction of sets:

  ```
  (member? x (add-element x xs)) == #t
  ```

  The property says that if we add an element `x` to any set, then `x` is a member of the resulting set.

4 Issues with algebraic laws on the first homework

In addition to the faults listed in the general rubric, we found three other common faults in algebraic laws from the first homework. The faults reflect confusion about what a variable in a law stands for: an actual parameter or a *part* of an actual parameter? When a variable stands for a part of a parameter, trouble sometimes follows.

In the first fault, the right-hand side of a law uses a variable that doesn't appear on the left, as in

```
(log10 (10 * m + d)) == (+ (log10 (/ n 10)) 1)
```

The `n` on the right-hand side is not specified—it could be anything. I can see what's going on: the left-hand side specifies the *parts* of the actual parameter, but the right-hand side uses `n` to name the parameter itself. What's meant by `(/ n 10)` is actually `m`.

To avoid this fault, remember this rule: *The right-hand side of an algebraic law may use any variable that appears on the left-hand side, and only those variables.*

In the second fault, a right-hand side misuses a variable from the left as if it were the argument, rather than a part of the argument. Here's an example:

```
(population-count (* 2 m)) ==
          (population-count (/ m 2))
```

The variable `m` is *already* meant to be half the argument: `m` = `n`/2. The right-hand side of the law incorrectly applies to `m` the operation that is meant to be applied to `n`.

In the third fault, a name like `m` is used in the algebraic laws to stand for a part of an argument, but in the code to stand for the entire argument. Here's an example:

```
;;   (log10 d)              == ...
;;   (log10 (+ (* 10 m) d)) == ...

(define log10 (m)
    ...)  ;; case 2: m == (+ (* 10 m) d)???
```

Each part is technically correct by itself, but mixing the two is just too confusing: the argument can't be *both* m and $10 \times m + d$. To avoid this fault, make sure each name stands consistently either for an argument or for a part of an argument, but not both.

5 Common issues using algebraic laws with Scheme

Below are some other issues you might run into when writing algebraic laws for Scheme functions.

Correct use of variables

A common mistake is to write laws thinking that variables are mutually exclusive with other forms of data. They aren't. When you write a variable, you are saying implicitly, "this could be *any* form of data, and I don't care which." In other words, when you write a variable in an argument position, you are promising not to look and see how the argument was formed. In particular, when you write a variable, you are promising never to apply `null?`, `car`, or `cdr` to that variable.

Here's an example of this common mistake:

```
(sublist? xs '()) == #f   ;; WRONG
(sublist? '() ys) == #t
... more cases below ...
```

The student who wrote these laws meant for `xs` and `ys` meant to be nonempty. But a variable could be any list, including the empty list. In this example, if both `xs` and `ys` are empty, the laws give inconsistent results. That's how we're certain that something is wrong. Here's a correct version, in which every argument is either explicitly empty or explicitly nonempty.

```
(sublist? (cons w ws) '())         == #f ;; RIGHT
(sublist? '()         (cons z zs)) == #t
... more cases below ...
```

These left-hand sides can't possibly be confused.

This version can be refined by observing that in the original set, the problem lies with the first law. The law `(sublist? '() ys) == #t` is actually good: the empty list is a sublist of any list `ys`, whether `ys` is empty or not. So we could write the laws this way:

```
(sublist? (cons w ws) '()) == #f ;; RIGHT
(sublist? '()         ys)  == #t ;; SPLENDID
... more cases below ...
```

The advantage of this final specification is that we might then have to consider fewer alternatives in the "more cases below."

Breaking S-expression inputs down by cases

Quite often it's useful to define an ordinary S-expression as one of the following:

- The empty list
- `(cons z zs)`, where `z` is an S-expression and `zs` is a list of S-expressions
- `a`, where `a` is an atom but not the empty list

A *common mistake* here is to forget the side condition on `a`. Here are some mistaken laws for counting the number of atoms in an ordinary S-expression:

```
(atom-count '())         = 0
(atom-count (cons z zs)) =
            (+ (atom-count z) (atom-count zs))
(atom-count a)           = 1     ;; WRONG
```

The last law needs a side condition:

```
(atom-count a)           = 1,
     where a is a non-null atom  ;; RIGHT
```

You can't break a function down by cases

Some of the problems on the homework, like `takewhile`, `dropwhile`, and `arg-max`, take functions as inputs. You can't break a function down by cases, because there's no way to ask a function how it was formed. All you can do with a function is apply it. How, then, should a function appear in an algebraic law? As a variable. Here's an example for `takewhile`, which takes two arguments, a predicate `p?` and a list of values. A function has one case and a list has two, and multiplied together there are two in total:

```
(takewhile p? '())         = ...
(takewhile p? (cons x xs)) = ...
```

That's not the end of the story, however: once we have both `p?` and an `x` that we could apply `p?` to, we could have extra cases depending on whether `(p? x)` is true or false. Those cases would be written as side conditions.

One final example: function `arg-max` takes a function and a *nonempty* list of values. The laws for `arg-max` will have one case for the function input (just a variable), and other cases for the nonempty list. (Finding the forms of a nonempty list is a homework problem.)