Lens Examples for Locks and Constraints

Sums

This section is motivated by the following example. We are given sales data for a number of stores in a company, with each store identified by its state. The regional managers of the company can view the sum of their own region's sales data and change that number, indicating a wish to increase future sales. If the manager wishes to change the sales figures from n to N, each store's figures will be multiplied by $\frac{N}{n}$ to make the change. Similarly, the CEO can see the total sum of national sales and adjust that number accordingly.

However, the sales data for an individual store or region may be fixed or *locked*, meaning that its sales data may not be changed.

On Notation: the lenses given here all have intuitive semantics in the "get" direction, but in the "put" direction there is some ambiguity. Therefore the examples detail only "put" behavior. The put function of a lens takes (1) some input, (2) the current state of the data, (3) some constraints on the data, and outputs either failure or a new state. This is modeled using arrows:

$$\text{input} \to \text{current state}, \text{constraints} \qquad \text{output}.$$

When the state is a list of length n and the constraint on the data is that certain elements of the list may not change in the output, we represent this *locking constraint* as a boolean list $O_1, \ldots, O_n$ of length n, where $O_i = T$ means the ith element is locked, and $O_i = F$ otherwise.

More general constraints are modeled as sets of feasible output.

Flat Sums

We start with a lens which takes a single list of numbers to its sum. The default case, assuming no locks on the elements of the list, distributes $n - \sum_{i=1}^{m} x_i$ among the list in proportion to each x_i.

$$200 \to [10, 20, 30, 40], \texttt{FFFF} \qquad [20, 40, 60, 80] \tag{1}$$

In the presence of locks on $\{x_j \mid j \in J\}$, we produce a list which is the result of fixing the locked entries and distributing $n - \sum_{i=1}^{m} x_i$ values among the $m - |J|$ locked elements of the list.

$$200 \to [10, 20, 30, 40], \texttt{TFFF} \qquad [10, 42.2, 63.3, 84.4] \tag{2}$$

To keep the first element of the list locked, we must scale the other elements by a factor of $\frac{19}{9}$.

If the constraints do not allow the elements of the list to be mutated to satisfy the sum, there are two possibilities of output. The first option is to fail if the list cannot be mutated in-place.

$$200 \to [10, 20, 30, 40], \texttt{TTTT} \qquad \texttt{fail} \tag{3}$$

The second option is to add elements to the list so that the sum holds.

$$200 \to [10, 20, 30, 40], \texttt{TTTT} \qquad [10, 20, 30, 40, 100] \tag{4}$$

In what context would this notion of sum be appropriate?

Split Sum

Next we consider the lens which maps an ordered pair to a labeled list, with each element of the list labeled as either `L`(left) or `R`(right). The left-labeled elements of the list should sum up to the left element of the ordered pair, and vice verse.

The default case with no locks behaves like an extension of the sum lens above.

$$(50, 50) \to [\texttt{L10}, \texttt{L20}, \texttt{R30}, \texttt{R40}], \texttt{FFFF} \qquad [\texttt{L16.67}, \texttt{L33.33}, \texttt{R21.43}, \texttt{R28.57}] \tag{5}$$

Similarly, with the case for locks...

$$(50, 50) \to [\texttt{L10}, \texttt{L20}, \texttt{R30}, \texttt{R40}], \texttt{TFFT} \qquad [\texttt{L10}, \texttt{L40}, \texttt{R10}, \texttt{R40}] \tag{6}$$

If locks prevent us from mutating either section of the list to fit the constraint, we default to the options presented for plain sums in this situation.

$$(50, 70) \to [\texttt{L10}, \texttt{L20}, \texttt{R30}, \texttt{R40}], \texttt{TTFF} \qquad \texttt{fail} \tag{7}$$

$$(50, 70) \to [\texttt{L10}, \texttt{L20}, \texttt{R30}, \texttt{R40}], \texttt{TTFF} \qquad [\texttt{L10}, \texttt{L20}, \texttt{L20}, \texttt{R30}, \texttt{R40}] \tag{8}$$

Multi-Level Lenses

We can represent the full sales example described in the beginning of this section as a lens with multiple levels. From the CEO's point of view, this lens stretches between the total sales data and a labeled list of store sales data.

When neither intermediary region is completely locked, a change from n to N first scales up the intermediary regions by a factor of N/n. The changes to each individual region must be accounted for, even if some of its sub-elements are locked.

$$200 \to (30, 70), [\texttt{L10}, \texttt{L20}, \texttt{R30}, \texttt{R40}], \texttt{TFFF} \qquad (60, 140), [\texttt{L10}, \texttt{L50}, \texttt{R60}, \texttt{R80}] \tag{9}$$

When an entire region is locked however, we must propagate the locks up to the regional level.

$$200 \to (30, 70), [\texttt{L10}, \texttt{L20}, \texttt{R30}, \texttt{R40}], \texttt{TTFF} \qquad (30, 170), [\texttt{L10}, \texttt{L20}, \texttt{R72.9}, \texttt{R97.1}] \tag{10}$$

Cumulative Average

A student is given some number of grades in a class, and she wants to know what grades she needs to get in the future to obtain a desired average. The constraints on this problem may go beyond locking to illuminate different environments.

Given that all grades fall within a certain range, how many assignments must the student complete to raise her average?

$$95 \to [80, 100], \{\text{lists like } [80, 100, x_0, x_1, x_2, \ldots] \text{ where } 0 \leq x_i \leq 100\} \qquad [80, 100, 100, 100] \tag{11}$$

The constraint might limit the number of future assignments.

$$95 \to [80, 100], \{\text{lists like } [80, 100, x_0, \ldots, x_{n-1}] \text{ where } 0 \leq x_i \leq 100 \text{ and } n \leq 1\} \qquad \texttt{fail} \tag{12}$$

Partition

Constraints on the partition lens can be stated in a very general form, and might be able to guide the lens's alignment/update policy. Let A, B and C be the following lists:

A		B	C
L	Bach	Bach	Asimov
R	Asimov	Beethoven	Austen
L	Beethoven		

For example, we can use the constraint to model the structure of the merge.

$$B, C \to A, \{\text{lists with authors first}\} \qquad \begin{array}{ll} \texttt{R} & \text{Asimov} \\ \texttt{R} & \text{Austen} \\ \texttt{L} & \text{Bach} \\ \texttt{L} & \text{Beethoven} \end{array} \tag{13}$$

$$B, C \to A, \{\text{lists like } A\texttt{++}Z \text{ where elements in } Z \text{ do not appear in } A\} \qquad \begin{array}{ll} \texttt{L} & \text{Bach} \\ \texttt{R} & \text{Asimov} \\ \texttt{L} & \text{Beethoven} \\ \texttt{R} & \text{Austen} \end{array} \tag{14}$$

Database Operations

In certain contexts, it might be valuable to view operations on relational databases as lenses. Though good practice in relational database design specifies that databases should not hold overlapping content, in actuality it might be convenient to have two perminant tables which share some information. These must both be able to be updated independently, and their updates propogated to the other table.

We can represent get functions by SELECT INTO statements and put functions by UPDATE statements, which take the old state of the table into account. WHERE clauses represent constraints we have on the lenses. In the previous examples, the constraints modify the *value* of the update. Here, the constraints modify the *range* of the update. Are these the constraints that we want, or are they just generalized lenses?

Select

Let's start with a simple SELECT statement example. Let JOBS be the following table, describing the jobs of employees at a company.

JOBS	id	name	title
	1	Adam Aardvark	CEO
	2	Brian Badger	VP of Sales
	3	Charlie Camel	VP of Engineering
	4	Diane Duck	Software Engineer

A simple select statement picks out the name of the CEO.

```
SELECT name
INTO CEO
FROM JOBS
WHERE title='CEO'
```

This produces a table CEO as follows:

CEO	name
	Adam Aardvark

Suppose Adam gets married and wants to change his name to a hyphenated version. The following query updates the CEO table:

```
UPDATE CEO
SET name=Adam Aardvark-Albatross
```

The result is the following updated table:

CEO	name
	Adam Aardvark-Albatross

The putback function uses the same constraint to update the JOBS table as it used to select into the CEO table.[1]

```
UPDATE JOBS
SET name = CEO.name
WHERE title = 'CEO'
```

The result is the following:

JOBS	id	name	title
	1	Adam Aardvark-Albatross	CEO
	2	Brian Badger	VP of Sales
	3	Charlie Camel	VP of Engineering
	4	Diane Duck	Software Engineer

Join

Consider the JOIN operation as a function from a pair of tables to a single table. We might also want to move from a single table back to the separate pairs. Let's start with two tables, JOBS as before, and BOSSES which gives the id numbers of employees and their bosses.

JOBS	id	name	title
	1	Adam Aardvark	CEO
	2	Brian Badger	VP of Sales
	3	Charlie Camel	VP of Engineering
	4	Diane Duck	Software Engineer

BOSSES	id	boss_id
	1	1
	2	1
	3	1
	4	3

Using the JOIN operation, we can write the following query which lists the names of employees along with the names of their boss.

```
SELECT J1.name, J2.name
INTO NAMED_BOSSES
FROM JOBS J1 JOIN BOSSES B JOIN JOBS J2
ON J1.id = B.id AND B.boss_id=J2.id
```

The result is the following:

NAMED_BOSSES	name	boss_name
	Adam Aardvark	Adam Aardvark
	Brian Badger	Adam Aardvark
	Charlie Camel	Adam Aardvark
	Diane Duck	Charlie Camel

[1]It is not clear that the constraint should always be the same in the put direction as it is in the get direction. Are there times when the constraint only applies in one way? When we are talking about selecting a view from a larger table, perhaps not, but the story is different for join statements, as we see below.

If we change Diane Duck's boss to Brian Badger, we get

NAMED_BOSSES	name	boss_name
	Adam Aardvark	Adam Aardvark
	Brian Badger	Adam Aardvark
	Charlie Camel	Adam Aardvark
	Diane Duck	Brian Badger

The change needs to be reflected back into the BOSSES table (as, in this case, no change is made in JOBS).

```
UPDATE BOSSES
FROM JOBS J1 JOIN NAMED_BOSSES N JOIN JOBS J2
ON J1.name = N.name AND N.boss_name = J2.name
SET B.boss_id = J2.id
WHERE B.id = J1.id
```

Here, the putback operation's constraints do not match the get operation's constraints; while before we matched on id's, here we must match on names. However, the update results in the expected BOSSES table:

BOSSES	id	boss_id
	1	1
	2	1
	3	1
	4	2