Conversational code

Dylan Holmes

May 9, 2017

Contents

1 The "impossible" puzzle

There is a delightful puzzle known as the *impossible puzzle* or the *sum-product* puzzle, which goes something like this:

> Two mathematicians, S and P, are trying to figure out a certain pair of integers. They have been informed that the numbers are both different, that both numbers are larger than one, and that their total is less than 100. Furthermore, S has just been told what the sum of the two numbers is, and P has been told their product. Both mathematicians are skilled at logic, and both of them know everything in this paragraph.
>
> The two mathematicians have the following conversation:
>
> P: "I can't figure out what the numbers are."
>
> S: "I knew you wouldn't be able to."
>
> P: "Then I know what they are, after all."
>
> S: "Then, I know, too."
>
> What are the two numbers?

Though the puzzle seems to provide inadequate information, you *can* find the unique solution through process of elimination (!). For convenience, you can even write a short computer program to help you translate what the mathematicians are saying, and to shoulder some of the brute-force calculations. It's a lot of fun.

2 A conversational solution

While writing such a program to solve this puzzle, I found that my code corresponded closely with the statement of the problem: each line from the mathematicians' conversation became a line for narrowing down the possible answers.

I began to wonder if I could accentuate this similarity, to write a program that actually **read like a conversation**. Could I architect my code so that a natural description of the problem would produce a solution?

Yes! This article describes the result of that project, which culminated in code like this:

```
(->> possible-answers
     ((know sum (dont know product))) ;; S: I knew you couldn't find the numbers.
     ((know product))                 ;; P: Then I know what they are after all.
     ((know sum)))                    ;; S: Then I do, too.


;; This code returns the solution to the 'impossible' problem.
```

2.1 In Python

Here is the structure I developed in Python:

```
upper_limit = 100 # the Sum is less than this

Product = lambda (x,y) : x * y
Sum = lambda (x,y) : x + y

unique = lambda x : 1 == len(x)
member = lambda coll : lambda x: x in coll


# -----------------------------------
```

```
def group_by(f, xs) :
    ret = {}
    for x in xs :
ret[f(x)] = ret.get(f(x), []) + [x]
    return ret.values()


def know(f, xs, pred=unique) :
    return sum(filter(pred, group_by(f, xs)),[])

def dont(k) :
    return lambda f, xs, pred=unique : k(f, xs, lambda x: not pred(x))


X0 = [(m, n)
      for m in range (1,upper_limit)
      for n in range (upper_limit-m, 0, -1)
      if 1 < m < n and m + n < upper_limit]
```

And here is the code that finds the solution. Note the close correspondence between what the mathematicians say and what the code says.

```
# P: I can't figure out what the numbers are.
# S: I knew you wouldn't be able to.
# P: Then I've found the solution, after all.
# S: Then I have, too.

X1 = dont(know)(Product, X0)
X2 = know(Sum, X0, lambda x: all(map(member(X1), x)))
X3 = know(Product, X2)
X4 = know(Sum, X3)

print X4
```

2.2 In Clojure

Having developed the code in Python, I suddenly remembered Clojure's threading macro `-»` [1]. The central function `know` seemed like a perfect fit

[1] Clojure's built-in thread-last macro (`-»`) behaves as follows: `(-» z (f x y))` becomes `(f x y z)`, `(-» z f)` becomes `(f z)`, and longer threads like `(-» z (f x y) g ... )`

for making a conversational "thread".

After a few days of re-designing the functions—I wanted to replace the abstruse `all(member(...))` line with something more eloquent—I managed to find the following solution. Here is the structure:

```
(ns ai.logical.impossible)

(def upper-limit 100)
(def product (partial apply *))
(def sum (partial apply +))


(def possible-answers
  (for [m (range 1 upper-limit)
n (range 1 upper-limit)
:when (and (< (+ m n) upper-limit) (< 1 m n))]
    [m n]))


;;; -----

(defn uniquely [f]
  (fn [xs]
    (->> (vals (group-by f xs))
 (filter #(= 1 (count %)))
 (reduce concat)
 set)))

(defn know
  ([f selector modifier]
     (fn [xs]
       (->> (vals (group-by f xs))
    (filter (comp modifier (partial every? (selector xs))))
    (reduce concat)
    set)))
  ([f selector]
     (know f selector identity))
  ([f]
```

recursively simplify to `(->» (f x y z) g ...)`.

```
    (know f (uniquely f))))

(defn dont
  ([kn f selector]
     (kn f selector not))
  ([kn f]
     (kn f (uniquely f) not)))
```

And here is the (eloquent!) code for finding the solution [2].

```
(->> possible-answers
     ((know sum (dont know product))) ;; S: I knew you couldn't find the numbers.
     ((know product))                 ;; P: Then I know what they are after all.
     ((know sum)))                    ;; S: Then I do, too.
```

Conversational code like this, besides being beautiful, seems like an expressive way to describe and solve problems. Though the underlying subroutines have somewhat obscure implementations, the resulting API is a kind of blackboxed domain-specific language for solving problems of this kind. Indeed, using the `know` and `dont` operators, you can construct arbitrarily nested statements of meta-knowledge such as this one:

```
(dont know sum (dont know product (know sum)))
;; "S is unaware that P couldn't figure out that S knew the numbers already"
```

You can invent new mathematicians who know their own facts about the numbers:

```
(defn parity [[x y]] (set (map even? [x y])))
;; This mathematician knows whether the numbers are both odd, both even, or one of eacl
```

And so you can solve an infinite variety of problems, using code for which a natural description of the problem yields the solution. [3]

[2] The careful reader will note that the first line of the conversation is missing. In order to work with the threading macro, I had to satisfy one especially strong constraint: each conversational line had to build upon the knowledge accumulated in the previous line. Each line had to be about what the mathematician learned in the previous line. But S's first reply ("I *knew* you couldn't figure out what the numbers are") isn't about what S learned from what P said ("I can't figure out what the numbers are"), but rather about what S knew all along. As a result, S's line had to be the first line in the conversation, and P's line had to be cut.

[3] See also: in one particular section of the programming book SICP, Sussman and Abelson introduce logic programming and constraint satisfaction; they build programs where the description of the problem is interpreted as a method for finding a solution.

2.3 In Scheme

As an addendum, here's a version of the code written in MIT Scheme.

```
(define range
  (lambda (n m)
    (cond
      ((= n m) (list n))
(else (cons n (range ((if (< n m) + -) n 1) m))))))

(define (not-any? pred lst)
  (or (null? lst)
      (and (not (pred (car lst))) (not-any? pred (cdr lst)))))

;; replace the value in the map with (f old-val)
;; or do nothing if the key isn't in the map.
(define (alter m k f)
  (cond ((null? m) m)
((equal? k (caar m))
 (cons (list k (f(cadar m))) (cdr m)))
(#t (cons (car m) (alter (cdr m) k f)))))


(define (group-by feature coll)
  (define (loop m coll)
     (cond ((null? coll) (map cadr m))
   ((false? (assq (feature (first coll)) m))
    (loop (cons (list (feature (first coll))
      (list (first coll))) m)
  (cdr coll)))
    (#t
     (loop (alter m (feature (first coll))
  (lambda (val)
    (cons (first coll) val)))
   (cdr coll)))
     )
     )
  (loop (list) coll))

(define (mathematician op)
  (lambda (xy)
```

```
    (op (first xy) (second xy))))

(define sum (mathematician +))
(define product (mathematician *))


(define (cartesian-product xs ys)
  (cond ((null? xs) xs)
((null? ys) ys)
((append
 (map (lambda (y) (list (first xs) y)) ys)
 (cartesian-product (cdr xs) ys)
 ))
  ))


(define possible-answers
  (filter (lambda (xy)
    ((lambda (x y)
       (and (< x y)
    (> x 1) (> y 1)
    (< (+ x y) 100))
       ) (first xy)(second xy))

    )
  (cartesian-product (range 1 100) (range 1 100))))


;;; ----

(define (uniquely f)
  (lambda (xs)
    (reduce append ()
    (filter (lambda (lst)
      (= 1 (length lst)))
    (group-by f xs)))))


(define (know f #!optional keepgen modifier)
  (cond
```

```
   ((default-object? modifier)
    (know f keepgen (lambda (x) x)))

   ((default-object? keepgen)
    (know f (uniquely f) modifier))

   (#t
    (lambda (xs)
      (let ((qq (keepgen xs)))
      (reduce append ()
      (filter (lambda (group)
(modifier
 (not-any?
  (lambda (x) (not (memv x qq)))
  group)))
      (group-by f xs))))))))

(define (dont k f #!optional keepgen modifier)
  (if (default-object? modifier)
      (k f keepgen not)
      (k f keepgen modifier)))



(define (conversation x #!rest fs)
  (fold-left (lambda (y f) (f y)) x fs))

(define answer
  (conversation
   possible-answers
   (know sum (dont know product))
   (know product)
   (know sum)))
```

2.4 Appendix: How the code works

Because the code is somewhat obscure, here's a short description of what the subroutines are actually doing.

- The named mathematicians `Sum` and `Product` are represented by "fea-

ture" functions that take in a pair of numbers and yield a feature of the pair, i.e. their sum and their product, respectively.

- Each line in the conversation is code that takes in a list of candidate answers, eliminates the answers that are now no longer possible, and yields the list of remaining candidate answers. I use the term `selector` to refer to such candidate-filtering functions.

- When a mathematician (e.g. Sum) can figure out the answer, it means that the true answer is one of the candidates that doesn't share a sum with any other candidate.

- When a mathematician (e.g. Sum) knows some *property* of the answer (e.g. that Product can't figure the answer out at this stage), it means that if you group the remaining candidates by Sum, the true answer must belong to a group where every member of the group has that property.

- In the Python version of the code, the higher-order function `know` takes a feature function ("mathematician") and a predicate as arguments. It returns a selector that groups the candidates by feature value and returns a flattened list of groups that pass the predicate test.

- In the Clojure version of the code, the function `know` takes a feature function, an optional "selector", and an optional modifier function. It returns a set, which we treat as a kind of selector.

- The modifier allows you to affect what gets filtered; it is included so that the `dont` function has a means of altering the internal behavior of the `know` function. By default, the modifier is just the identity function. To invert the effect of the `know` function, the `dont` function simply calls the `know` function with `"not"` as the modifier.

- Clojure's built-in thread-last macro (`->>`) behaves as follows: `(->> z (f x y))` becomes `(f x y z)`, `(->> z f)` becomes `(f z)`, and longer threads like `(->> z (f x y) g ... )` recursively simplify to `(->> (f x y z) g ...)`.