Predicates and backtracking

Introduction (from Prof. David Liu's notes)

Generating "all possible combinations" of choices is fun and impressive, but not necessarily that useful without some kind of mechanism to filter out unwanted combinations. While we could do this by collecting choices into a stream and applying a traditional `filter` operation, we'll instead take a different approach to combine the *automatic generation* that choice expressions give us with *automatic backtracking*, which is at the core of the technique used by logic programming languages like Prolog.

The **logic programming** paradigm is centred on one simple question: "is this true?" Programs are not a sequence of steps to perform, nor the combination of functions; instead, a logic program consists of the definition of a search space (the universe of values known to the program) and the statement of queries for the computer to answer based on these values.

Based on this description, it is probably not hard to see that logic programs are generally more declarative than either imperative or functional programs, given that they fundamentally specify *what* is being queried, but leave it up to the underlying implementation of the language to determine *how* to answer the query.

(A more familiar instance of this idea is the database query language SQL, in which programmers write queries specifying what data they want, without worrying too much about how this data will be retrieved.)

Now, we already have a mechanism for defining a search space, using combinations of choices. But what about queries? In this section, we'll define two ways of **searching** through a space: first by filtering a stream using a *query operator* `?-`, and then via clever use of continuations.

Searching through a stream using the Query Operator `?-`

The query operator `?-` takes a unary predicate and a thunk that returns a stream, and (lazily) filters the stream so to contain only elements that satisfy the predicate.

```
#|
(define ?- pred stream-thunk) -> a thunk that returns a stream
  pred: a predicate
  stream-thunk: a thunk that returns a stream

  Return a thunk that evaluates to a stream consisting of only
  elements in stream-thunk that satisfies pred
|#
(define (?- pred stream-thunk)
  (let* ([stream (stream-thunk)])        ; evaluate the stream
    (if (s-null? stream)
        (thunk s-null)                   ; the return type is always a stream containing a thunk
        (let* ([elem (s-first stream)]) ; evaluate the first element
          (if (pred elem)                ; if the first element satisfy the predicate
              (thunk (cons (thunk elem) ; then keep it in the stream
                           (?- pred (cdr stream))))
              (?- pred (cdr stream)))))))) ; otherwise, skip the element
```

The definition of `?-` might seem a little more tedious than necessary. We took care to take and return thunks that evaluate to streams, which is the data type that is returned by the `-<` operation. This is to make `?-` compatible with `-<`:

```
> (define g (?- even? (do/-< (-< 1 2 3 4))))
> (next g)
2
> (next g)
4
```

```
> (next g)
'DONE
```

The `?-` function works similarly to the list function `filter`. If a stream value does not satisfy the predicate, then we automatically move on to the *next value*. To use some terminology of artificial intelligence, this behaviour of automatically checking the next value causes **backtracking**, in which the program goes back to a previous point in its execution, and makes a different choice.

Search examples (from Prof. David Liu's notes)

Probably the easiest way to appreciate the power that this combination of choice and querying yields is to see some examples in action. We have already seen the use of this automated backtracking as a lazy `filter` operation, producing values in a list of arguments that satisfy a predicate one at a time, rather than all at once. We can extend this fairly easily to passing in multiple choice points, with the one downside being that our `?-` function only takes unary predicates, and so we'll need to pass all of the choices into a list of arguments.

Our problem will be the following: given a number `n`, determine all triples of numbers `(a, b, c)` whose product is `n`. Here is our definition of the input data; unlike the past examples, we won't walk through the code with you, so we strongly recommend pausing here to study the code carefully!

```
; Generate all triples of numbers, each less than n
(define (triples n)
  (let* ([nums (range 1 n)])
    (do/-< (list (apply -< nums)
                 (apply -< nums)
                 (apply -< nums)))))

(define (product? n)
  (lambda (triple)
    (equal? n (apply * triple))))

> (define g (?- (product? 6) (triples 6)))
> (next! g)
'(1 2 3)
> (next! g)
'(1 3 2)
> (next! g)
'(2 1 3)
> (next! g)
'(2 3 1)
> (next! g)
'(3 1 2)
> (next! g)
'(3 2 1)
> (next! g)
'DONE
```

See that? In just a few lines of code, we expressed a computation for finding factorizations in a purely declarative way: specifying *what* we wanted to find, rather than *how* to find the solutions. This is the essence of pure logic programming: we take a problem and break it down into logical constraints on certain values, give our program these constraints, and let it find the solutions. This should almost feel like cheating—isn't the program doing all of the hard work?

Remember that we are studying not just different programming paradigms, but also trying to understand what it means to design a language. We spent a fair amount of time implementing the language features `-<`, `next`, and `?-`, expressly for the purpose of making it easier for users of our augmented Racket to write code. So here we're putting on our user hats, and take advantage of the designers' work!

Satisfiability (from Prof. David Liu's notes)

One of the most famous problems in computer science is a distilled essence of the above approach: the *satisfiability problem*, in which we are given a propositional boolean formula, and asked whether or not we can make that formula true. Here is an example of such a formula, written in the notation of Racket:[1]

```
(and (or x1 (not x2) x4)
     (or x2 x3 x4)
     (or (not x2) (not x3) (not x4))
     (or (not x1) x3 (not x4))
     (or x1 x2 x3)
     (or x1 (not x2) (not x4)))
```

With our current system, it is easy to find solutions:

```
(define (sat lst)
  (let ([x1 (first lst)]
        [x2 (second lst)]
        [x3 (third lst)]
        [x4 (fourth lst)])
    (and (or x1 (not x2) x4)
         (or x2 x3 x4)
         (or (not x2) (not x3) (not x4))
         (or (not x1) x3 (not x4))
         (or x1 x2 x3)
         (or x1 (not x2) (not x4)))))
```

```
> (define g (?- sat (do/-< (list (-< #t #f) (-< #t #f) (-< #t #f) (-< #t #f)))))
> (next! g)
'(#t #t #t #f)
> (next! g)
'(#t #t #f #f)
> (next! g)
'(#t #f #t #t)
> (next! g)
'(#t #f #t #f)
> (next! g)
'(#f #f #t #t)
> (next! g)
'(#f #f #t #f)
> (next)
'done
```

Backtracking using Continuations

Using `?-` to search through some space is relatively straightforward, but there is one drawback: we need to generate **every item** in a combinatorial search space, and then check if that item satisfy the predicate. For simple problems, like a SAT problem with only four variables, this is not an issue. We only need to check $2 \times 2 \times 2 \times 2 = 16$ possibilities. For larger problems, we may want ways to **restrict** the search space.

We illustrate this strategy using the *triples* problem from above: the problem of finding triples of numbers whose product is `n`. This time, we use a single function that performs both the data generation *and* filtering. Our implementation will consist of three stages:

1. Implementation of a search that does not use backtracking. This solution will be *worse* than when we used `?-`, but is necessary to get to the next stage.

[1] `x1`, `x2`, `x3`, and `x4` are boolean variables

2. Implementation of a search that *does* use backtracking. This solution micmics the behaviour of the query `(?- (product? 6) (triples 6))`.
3. Implementation of a search that uses backtracking, and also restricts the search space.

Implementation 1: No backtracking

First, let's consider how to write a single function that both generates triples of numbers, and checks whether the triples have a product of `n`.

We introduce the main function `(triples-product n)` to generate all triples whose product is `n`. That function calls a helper function `solve-triple` that incrementally builds up a triple of numbers.

```
; Find all triples of numbers (each less than n) whose product is n
(define (triples-product n)
  (do/-< (solve-triple n '())))

#|
(solve-triple n lst)
  n   : target product
  lst : an accumulated list of numbers that are a part of the triple
|#
(define (solve-triple n lst)
  (cond
    ; if lst contains three numbers, then we will check whether their
    ; product is equal to n
    [(equal? (length lst) 3)
     (if (equal? n (apply * lst))
         lst        ; if so, return the lst
         'FAIL)]    ; ...but what about otherwise? let's return 'FAIL for now
    ; if lst contains fewer than three numbers, then we will produce
    ; a set of choices for another number to add to lst, and use
    ; amb -< to search through them all
    [else
     (let* ([next-num (apply -< (range 1 n))])  ; possible next values
       (solve-triple n (cons next-num lst)))]))
```

If we ignore the use of `-<` and assumed that `next-num` referred to a single value, the function `solve-triple` would be fairly easy to understand. That's the beauty of the `-<` operator: we can still write code that operate on a *single* choice of `next-num` (and therefore `lst`)!

Unfortunately, this implementation leaves much to be desired. When a triple of numbers have a product not equal to `n`, we produce a failing value `'FAIL`.

```
> (define g (triples-product 4))
> (next! g)
'FAIL
> (next! g)
'FAIL
> (next! g)
'FAIL
> (next! g)
'FAIL
> (next! g)
'(2 2 1)
> (next! g)
'FAIL
> (next! g)
```

Ideally, we would like the same behaviour as `(?- (product? 6) (triples 6))`, where a failure leads to *automatic backtracking*, or the continued execution of the search until the next successful item is found.

Implementation 2: Backtracking

The solution is actually very simple, but a little mindboggling. Instead of returning `'FAIL` upon failure, we'll instead call a new function `(fail)`. This function calls `(shift k s-null)`, which captures the continuation `k` and *ignores it*, instead returning the empty stream `s-null`.

```
(define (fail)      ; capture the continuation k, and **ignore it**
  (shift k s-null)) ; instead, return the empty stream s-null

(define (triples-product n)
  (do/-< (solve-triple n '())))

(define (solve-triple n lst)
  (cond
    [(equal? (length lst) 3)
     (if (equal? n (apply * lst))
         lst
         (fail))]
    [else
     (let* ([next-num (apply -< (range 1 n))])
       (solve-triple n (cons next-num lst)))]))

> (define g (triples-product 4))
> (next! g)
'(2 2 1)
> (next! g)
'(2 1 2)
> (next! g)
'(1 2 2)
```

Aside (why did that work?)

You might wonder why using `shift` works at all. Why doesn't `shift` capture the *entire* rest of the computation? The reason is that each invocation of a captured continuation has an implicit `reset` around it.

For example, these two expressions are equivalent, and will both evaluate to 3.

```
> (reset (+ (shift k (+ 1 (k 5))) (shift l 2)))
3
> (reset (+ (shift k (+ 1 (reset (k 5)))) (shift l 2)))
3

    (+ (shift k (+ 1 (k 5))) (shift l 2))
<=> (+ (shift k (+ 1 (reset (k 5)))) (shift l 2))
==> (+ 1 (reset (k 5)))     ; where k = (+ _ (shift l 2))
==> (+ 1 (reset (+ _ (shift l 2)) 5))
==> (+ 1 (reset (+ 5 (shift l 2))))
==> (+ 1 2)                 ; where l = (+ 5 _) and NOT (+ 1 (+ 5 _ ))
==> 3
```

So, in our `solve-triple` example, the `(shift k s-null)` makes one of the calls to `(k x)` in `s-append` evaluate to the empty stream `s-null`.

Implementation 3: Restricting the Search Space

With backtracking in place, we can consider other ways of restricting the search space. That is, can we find ways to reduce the choices of `next-num` that we need to search?

One possibility is to consider the numbers already in `lst`, and use it to further cap what the remaining numbers could be. More specifically, if we want the product of the numbers in `lst` to be exactly `n`, then additional numbers should be at most `(/ n (apply * lst))`.

```
(define (triples-product n)
  (do/-< (solve-triple n '())))

(define (solve-triple n lst)
  (cond
    [(equal? (length lst) 3)
     (if (equal? n (apply * lst)) lst (fail))]
    [else
     (let* ([next-num (apply -< (range 1 (min n (+ (/ n (apply * lst)) 1))))])
       (solve-triple n (cons next-num lst)))]))

> (define g (triples-product 4))
> (next! g)
'(2 2 1)
> (next! g)
'(2 1 2)
> (next! g)
'(1 2 2)
```

This implement of `solve-triple` should produce the same result as the second version, but will be faster for large values of `n`.

Acknowledgements

Portions of these notes are derived from the CSC324 Notes by Prof. David Liu. The portions that are directly duplicated are indicated. Other portions are still borrowed heavily from Prof. Liu's work. https://www.cs.toronto.edu/~david/

These notes are developed with the help of Gregory Rosenblatt http://gregrosenblatt.com