mk-convert

28 May 2014

Contents

1 Converting to miniKanren, quasimatically.

```
#lang racket
(require utah/mk)
(require utah/let-pair)
(provide (all-defined-out))
```

Some folk wanted a bit more detail on converting programs from Scheme to miniKanren. So I thought we could walk through an example in some detail, and that way you can play along at home. So here's a real humdinger of an example–list-union. It takes two lists (which represent sets) and returns a list representation of their union.

```
(define (list-union s1 s2)
  (cond
    ((null? s1) s2)
    ((member (car s1) s2) (list-union (cdr s1) s2))
    (else (cons (car s1) (list-union (cdr s1) s2)))))
```

Let's give it a whirl

```
-> (list-union '() '(1 2 3))
(1 2 3)
-> (list-union '(2) '(3 2 1))
(3 2 1)
-> (list-union '(1 2) '(4 5))
(1 2 4 5)
```

That seems to work, ship it.

So, as Will said on Tuesday, my preferred way to do this is to get my Scheme program looking as much like the resultant miniKanren as I can, and then convert to miniKanren from there. Testing the whole way through.

So the plan is to massage this code until it looks like the miniKanren would just fall out trivially, and see if we can't list some rules along the way.

1.1 Variations on a Scheme

Let's start by talking about the *question* of that first cond line, (null? s1). We know, where we're going, that == is our big, giant hammer. Where it makes sense, I'd like to use equal?, the Racket primitive which tests for structural equality, because that's going to map pretty well to ==. So let's rewrite that to a similar question, which maps over nicely.

```
(define (list-union s1 s2)
  (cond
    ((equal? '() s1) s2)
    ((member (car s1) s2) (list-union (cdr s1) s2))
    (else (cons (car s1) (list-union (cdr s1) s2)))))
```

Looking good there. The *answer* of that first cond clause is a value. We know that when we go to miniKanrenize this code, we'll just unify that value with the "out" variable we will introduce.

Now, something to bear in mind is that you can't rely on any Scheme primitives aside from cons. We can cons things together to build structures, but we don't get to pull'em apart any way other than unifying with a pair. In the remaining lines, we're taking cars and cdrs, we'll need some way to represent what we're doing. And here's another miniKanrenizing guideline. When you have multiple cond clauses which require performing the same structural decomposition, only do it once, and then use it. Right quick we're going to do a couple of steps that at first blush look like some pretty fancy dancing. But after doing it a few times, it'll get to be old hat.

```
(define (list-union s1 s2)
  (cond
    ((equal? '() s1) s2)
    ((pair? s1)
     (cond
       ((member (car s1) s2) (list-union (cdr s1) s2))
       (else (cons (car s1) (list-union (cdr s1) s2)))))))
```

In either one of those last two lines, s1 has to be a pair. So that (pair? s1) question we ask there is safe and valid and right and good. And now that we know it's a pair, we have the opportunity to let-bind the car and cdr of s1, and use those names throughout. So here we go.

```
(define (list-union s1 s2)
  (cond
    ((equal? '() s1) s2)
    ((pair? s1)
     (let ((a (car s1))
           (d (cdr s1)))
       (cond
         ((member a s2) (list-union d s2))
         (else (cons a (list-union d s2))))))))
```

That's an absolutely legit maneuver. And if you're happy with doing it that way, power to you. I'm going to use a macro we put together called let-pair to do something similar.

```
(define-syntax let-pair
  (syntax-rules ()
    [(_ ((a . d) call) body)
      (let ((tmp-res call))
        (let ((a (car tmp-res))
              (d (cdr tmp-res)))
          body))]))
```

Let-pair evaluates the expression (which better *be* something with a car and cdr), and then binds the car and cdr to names you give it. It's just a bit of syntax wizardry, but compare the below to the above and you'll get what's going on.

```
(define (list-union s1 s2)
  (cond
    ((equal? '() s1) s2)
    ((pair? s1)
     (let-pair ((a . d) s1)
       (cond
         ((member a s2) (list-union d s2))
         (else (cons a (list-union d s2))))))))
```

I like that slightly better, ‘cause when we go to miniKanren, we just slip an == in there with some backticks and commas and we'll be on our way.

Nooow we get to something pretty interesting. That call to member. We can't rely on Racket's member, because it doesn't know how to "go backwards". We're going to have to write our own function, that way we can miniKanrenize *it*, and make our list-union do what we want when we go to list-uniono.

So, actually, we're at an interesting point here. A small aside with a punchline.

```
-> (member 3 '(4 5 6))
#f
```

Above, the expression evaluates to #f because 3 isn't in that list. Do you know what value the expression below will evaluate to?

```
-> (member 3 '(3 4 5))
```

Give you a hint, it's not #t like you might expect. In fact, it's (3 4 5)

```
-> (member 3 '(3 4 5))
(3 4 5)
```

When we write our version of this function, we've got some decisions to make. We must decide if we want to preserve that behavior, or just return #f or #t–that is, should we write a predicate member? instead. And bear in mind that else in the final line below still has to be dealt with as well.

```
(define (list-union s1 s2)
  (cond
    ((equal? '() s1) s2)
    ((pair? s1)
     (let-pair ((a . d) s1)
       (cond
         ((member a s2) (list-union d s2))
--->     (else (cons a (list-union d s2))))))))
```

Now, here's the deciding factor. When member returns a non-false value, like (3 4 5) above, we don't care what it is. Sometimes, you might. Here we don't. We'll go with the predicate, member?.

```
(define (member? x ls)
  (cond
    ((null? ls) #f)
    ((equal? x (car ls)) #t)
    (else (member? x (cdr ls)))))

(define (list-union s1 s2)
  (cond
    ((equal? '() s1) s2)
    ((pair? s1)
     (let-pair ((a . d) s1)
       (cond
         ((member? a s2) (list-union d s2))
         (else (cons a (list-union d s2))))))))
```

And we can go ahead and test that, make sure it still works. Check. Now, let's do to member? the sorts of things we've done to list-union so far.

```
(define (member? x ls)
  (cond
    ((equal? '() ls) #f)
    ((pair? ls)
     (let-pair ((a . d) ls)
       (cond
         ((equal? x a) #t)
--->     (else (member? x d)))))))

(define (list-union s1 s2)
  (cond
    ((equal? '() s1) s2)
    ((pair? s1)
     (let-pair ((a . d) s1)
        (cond
          ((member? a s2) (list-union d s2))
          (else (cons a (list-union d s2)))))))
```

Will made a point of demonstrating that we need our cond clauses to be non-overlapping to get the behavior we expect and desire when running backwards. Recall that in the rembero example we had to add disequality constraints in order to avoid the odd-looking answers. Same deal here. Now, the else sitting in member? is one we know how to deal with. I know how to write not equal?, and that'll translate right nicely to miniKanren.

```
(define (member? x ls)
  (cond
    ((equal? '() ls) #f)
    ((pair? ls)
     (let-pair ((a . d) ls)
       (cond
         ((equal? x a) #t)
         ((not (equal? x a)) (member? x d)))))))

(define (list-union s1 s2)
  (cond
    ((equal? '() s1) s2)
    ((pair? s1)
     (let-pair ((a . d) s1)
       (cond
         ((member? a s2) (list-union d s2))
         (else (cons a (list-union d s2)))))))
```

Back to list-union. That else in member? was pretty easy to do. The one in list-union, not so much. The fall-through cases of a cond clause with a non-trivial question (that is, where the question of the cond line isn't going to be just a simple == thing) are non-trivial.

We could write something called not-a-member, that returns true in all the cases that member will return false. That is, we construct a predicate that's the complement of member?, and use it where we have the else. Or we can use the boolean value which is the result of the call to member?. This decision impacts how we translate member? to miniKanren. I'm going to choose to make use of the boolean value which we get back. So we'll rewrite that inner cond, and let-bind the result of the call.

```
(define (list-union s1 s2)
  (cond
    ((equal? '() s1) s2)
    ((pair? s1)
     (let-pair ((a . d) s1)
       (let ((b (member? a s2)))
         (cond
           ((equal? b #t) (list-union d s2))
           (else (cons a (list-union d s2)))))))))
```

Now we've got something we can do with that else. If your instict was to jump to disequality constraints, you aren't wrong, but we can do better. If it isn't #t, then it must be #f.

```
(define (list-union s1 s2)
  (cond
    ((equal? '() s1) s2)
    ((pair? s1)
     (let-pair ((a . d) s1)
       (let ((b (member? a s2)))
         (cond
           ((equal? b #t) (list-union d s2))
           ((equal? b #f) (cons a (list-union d s2)))))))))
```

Now I want to look at those two answers of the final two cond clauses. Here there is something interesting. We are computing the same value, (list-union d s2), either way. We can pull that up into a let binding. We can in fact pull that up into the let binding we've already created. This implies that the order in which we evaluate the clauses of that let doesn't matter. In the biz, we use 'res' as a name for the variable in the recursive call, for the RESult of the recursion.

```
(define (list-union s1 s2)
  (cond
    ((equal? '() s1) s2)
    ((pair? s1)
     (let-pair ((a . d) s1)
       (let ((b (member? a s2))
             (res (list-union d s2)))
         (cond
          ((equal? b #t) res)
          ((equal? b #f) (cons a res))))))))
```

Now (cons a res) is a value, which when we go to miniKanren, we'll just unify with "out". I'm going to go ahead and rewrite it with the backtick comma syntax, and then we can marvel at what we've done.

```
(define (member? x ls)
  (cond
    ((equal? '() ls) #f)
    ((pair? ls)
     (let-pair ((a . d) ls)
       (cond
         ((equal? x a) #t)
         ((not (equal? x a)) (member? x d)))))))

(define (list-union s1 s2)
  (cond
    ((equal? '() s1) s2)
    ((pair? s1)
     (let-pair ((a . d) s1)
       (let ((b (member? a s2))
             (res (list-union d s2)))
         (cond
           ((equal? b #t) res)
           ((equal? b #f) `(,a . ,res))))))))
```

We can now reorder our cond lines in whatever order we wish, and we'll still get the same behavior. And we've got something which, to my eye, looks a heck of a lot like miniKanren but still runs in straight Racket.

1.2 Racket to miniKanren, finally

We can now start converting. We can do member? first, and then list-union. With a blind, dogmatic obedience, I dutifully stick an 'o' on the end. I do a find-and-replace, so I never leave a call to the old version. I've been bit by that too many times to count.

```
(define (member?o x ls)
  (cond
    ((equal? '() ls) #f)
    ((pair? ls)
     (let-pair ((a . d) ls)
       (cond
         ((equal? x a) #t)
         ((not (equal? x a)) (member?o x d)))))))
```

We need to add a third argument, the out variable, change cond to conde, and we know how to handle the first clause pretty well.

```
(define (member?o x ls out)
  (conde
    ((== '() ls) (== #f out))
    ...))
```

Now for that second clause, the (pair? ls) question and the let-pair are all satisfied by unification. It has a flavor of both testing and binding—that's why it's the giant hammer. Let's rewrite that part, and then we'll come to the inner cond next. Don't forget to freshen the variables a and d.

```
(define (member?o x ls out)
  (conde
    ((== '() ls) (== #f out))
    ((fresh (a d)
       (== `(,a . ,d) ls)
       ...))))
```

That cond has to be a conde, the first clause of that cond falls pretty easily. (not (equal? ...)) needs to be a disequality constraint, and the recursive call needs the third argument passed along.

```
(define (member?o x ls out)
  (conde
    ((== '() ls) (== #f out))
    ((fresh (a d)
       (== `(,a . ,d) ls)
       (conde
         ((== x a) (== #t out))
         ((=/= x a) (member?o x d out)))))))
```

And we can go ahead and test it.

```
-> (run 1 (q) (member?o 3 '(3 4 5) q))
(#t)
-> (run* (q) (member?o 3 '(3 4 5) q))
(#t)
-> (run* (q) (member?o q '(3 4 5) #t))
(3 4 5)
-> (run 4 (q) (member?o 3 q #t))
((3 . _.0)
 ((_.0 3 . _.1) (=/= ((_.0 3))))
 ((_.0 _.1 3 . _.2) (=/= ((_.0 3)) ((_.1 3))))
 ((_.0 _.1 _.2 3 . _.3) (=/= ((_.0 3)) ((_.1 3)) ((_.2 3)))))
-> (run 4 (q) (fresh (a b c) (== `(,a ,b ,c) q) (member?o a b c)))
((_.0 () #f)
 (_.0 (_.0 . _.1) #t)
 ((_.0 (_.1) #f) (=/= ((_.0 _.1))))
 ((_.0 (_.1 _.0 . _.2) #t) (=/= ((_.0 _.1)))))
```

Lookin' good. Let's bring it home. List-union becomes list-uniono.

```
(define (list-uniono s1 s2 out)
  ...)
```

And let's go ahead and take care of everything up to the let binding.

```
(define (list-uniono s1 s2 out)
  (conde
    ((== '() s1) (== s2 out))
    ((fresh (a d)
       (== `(,a . ,d) s1)
       ...))))
```

The left-hand sides of the clauses of the let reveal that I'm going to need to freshen b and res. And I know that I don't need to freshen them any earlier, either. Because the order of the clauses of the let didn't matter, our transformation didn't give us information regarding which one we want to list first. It won't impact the correctness, but it might have serious performance implications.

```
(define (list-uniono s1 s2 out)
  (conde
    ((== '() s1) (== s2 out))
    ((fresh (a d)
       (== `(,a . ,d) s1)
       (fresh (b res)
         (member?o a s2 b)
         (list-uniono d s2 res)
         ...)))))
```

The variables on the left-hand side of the let binding are what I wanted to pass as the last elements of the relations. Cond becomes conde, we can handle the questions, and the answers similar to the manner above. Let's go ahead and tackle the rest of it.

```
(define (list-uniono s1 s2 out)
  (conde
    ((== '() s1) (== s2 out))
    ((fresh (a d)
       (== `(,a . ,d) s1)
       (fresh (b res)
         (member?o a s2 b)
         (list-uniono d s2 res)
         (conde
           ((== b #t) (== res out))
           ((== b #f) (== `(,a . ,res) out))))))))
```

It's quasi-automatic, reasonably quick, and results in miniKanren.

```
-> (run 1 (q) (list-uniono '(1 2 3 4) '(5 6) q))
((1 2 3 4 5 6))
```

1.3 Back to the beginning

Unfortunately, that miniKanren is going to have bad divergence behavior, and in any case won't be terribly performant running backwards. But we're close.

Let-binding the serious recursive relation invocations above gave us the variable names right where we needed'em, but means that they are above the calls to == in conde. Consider the three clauses of the inner fresh expression in list-uniono. If we run with s1 and s2 *fresh*, but with some information in the variable out, then we will continue to recur, and the information in out will not get propogated through when we do a run*. This may cause running the relation to loop infinitely. We can manually reorder those clauses, and then we've got what we're after.

```
(define (member?o x ls out)
  (conde
    ((== '() ls) (== #f out))
    ((fresh (a d)
       (== `(,a . ,d) ls)
       (conde
         ((== x a) (== #t out))
         ((=/= x a) (member?o x d out)))))))

(define (list-uniono s1 s2 out)
  (conde
    ((== '() s1) (== s2 out))
    ((fresh (a d)
       (== `(,a . ,d) s1)
       (fresh (b res)
         (conde
           ((== b #t) (== res out))
           ((== b #f) (== `(,a . ,res) out)))
         (member?o a s2 b)
         (list-uniono d s2 res))))))
```

And it runs backward too.

```
-> (run 5 (q) (fresh (a b) (== q `(,a ,b)) (list-uniono a b '(1 2))))
((() (1 2)) ((1) (1 2)) ((1 2) ()) ((1) (2)) ((1 1) (1 2)))
```

Now we don't have a prayer of it completing with run*. We assumed our input were lists without duplicates. But we didn't ensure that. See the 5th answer above. Moreover, what we called "sets" are really lists with definite order. As a result running backwards might not give the expected results. If we are okay with this behavior, awesome. If not, well, then we might want to try augmenting our original functional program some, and then redoing the transformation. There's plenty to play with.

1.4 Conclusions and further guidance

Though this seems like a pretty hefty program, in the Scheme of things it's not all that big. As you go on to write more miniKanren, here are some things, big and small, to bear in mind:

- Everything has to be first-order, defunctionalized, and using data-structure representations.

- We let-bound the values of list-union and member?. In general, if you know what the structure of the call is going to be, use match (or pmatch) to evaluate the expression and match the pieces. This helps to avoid freshening unnecessary variables.

- When using match, if you're matching on a variable, and the result of the match is exactly that variable, don't create a new variable name. Conversely, if you are using match and decomposing a structure, don't shadow the variable name by rebinding it in the pattern.

- When you can, put multiple clauses in your let bindings. This indicates when your miniKanren program has clauses for which the order doesn't matter, at least when running in the standard (functional) *mode*.

- If you plan on using the relational arithmetic suite to perform arithmetic operations, recall those are themselves recursive relations, and should be let-bound as appropriate.

- Decide ahead of time if you will need to tag relational numbers (to distinguish them from arbitrary lists), if so, bear the tag in mind when you are thinking about the decomposition. It can help to tag your arabic numbers before going to the miniKanren arithmetic suite, in this case.

- When using match or pmatch, it is often advantageous to reorder your clauses in order to group similar decompositions together.

- When helper functions have been transformed to first-order and data structures, often you can then in-line the data structures, to avoid freshening some variables and unifications.

- Functions with multiple return values are perfectly acceptable. This often avoids unnecessarily constructing and destructing pairs out of a recursion. Use let-values the way we used let above, and feel free to combine let bindings into let-values bindings, if there is no dependence between them.