Metaprogramming: Relational Programming — Nada Amin

Lecture 2 - *June 27, 2025*

1 Introduction

Relational programming is basically: programming with relations instead of functions. No distinction between input and output; we can get multiple behaviors out of the same relation. The language shown in the lecture is called *miniKanren*, which is a pure constraint logic programming language.[1]

2 Basic language features

Here's a "normal" functional `append` in Scheme:

```
(define (my-append l s)
  (if (null? l )
      s
      (cons (car l) (my-append (cdr l) s))))
```

And now, here's a relational version of `append` (note that the convention here for naming relations is to add an "o" to the end of the functional analog):

```
(defrel (my-appendo l s ls)
  ;; conde is similar to cond from before, but relationally takes
  ;; nondeterministic branch
```

[1] "The Reasoned Schemer" is a book that goes into far more detail about miniKanren.

```
  (conde
   ((== l '()) (== s ls))
   ((fresh (a d ds)
           (== (cons a d) l)
           (== (cons a ds) ls)
           (my-appendo d s ds)))))
```

Note that textttdefrel defines a generic relation, which is a constraint on the search space of our generator (space of terms that can be generated). Now, here is what happens when we run our relational `my-appendo`:

```
(run 1 (q)
     (my-appendo '(a b c '(d e) q)))
;; we get ((a b c d e))

(run 1 (q)
     (my-appendo q '(d e) '(a b c d e)))
;; we get ((a b c))

(run* (x y)
      (my-appendo x y '(a b c d e)))
;; returns all possible pairs of lists x, y that can concatenate to (a b c d e)
```

The primitives here are:

1. `conde`: similar to standard Scheme conditional operator `cond`, but takes nondeterministic branch
2. `==`: unification
3. recursive calls to relation
4. `fresh` to introduce new logical variables

2.1 Unification

```
(run* (q)
      (== 1 1))
;; gives us (_.0), i.e. a fresh logical variable (unification succeeds!)
```

```
(run* (q)
      (== 1 2))
;; gives us (), an empty list ()
(run* (q)
      (== 1 q))
;; gives us (1)
```

2.2 conde

```
(run* (q)
      (conde
       ((== q 1))
       ((== q 2))
       ((== 1 1))))
;; gives us (1 2 _.0)
```

2.3 lookupo

Writing a relation to look up a variable in the environment.

```
(defrel (lookupo x env t)
  (fresh (rest y v)
         (== '((,y . ,v) . ,rest) env)
         (conde
          ((==y x) (== v t))
          ((=/= y x) (lookupo x rest t)))))

;; now, using lookupo...
(run 1 (q)
     (lookupo 'y '((x . 1) (y . 2)) q))
;; gives us 2, as expected
```

An aside: search in miniKanren is an interleaving search that is complete.

2.4 `absento`

We don't want our system to allow for quoting of the internal representation, and `absento` makes sure that we can't quote closures. In other words, it answers the following question: *"Does this subterm occur anywhere in the term either as a term or another subterm? If so, fail."*.

```
(run 1 (q) (absento 'foo '(foo)))
;; this fails and gives ()

(run 1 (q) (absento 'foo '(foobar)))
;; succeeds and gives (_.0)
```

3 Recursive synthesis of `append`

There is an automated way to get from a function to a relation, i.e. from `append` to `appendo`. Specifically, we start by defining `append` (i.e. functional append), then wrap it to get some relational behavior:

```
(test
 (run* (x y)
       (evalo
        `(letrec ((append (lambda (xs ys)
                            (if (null? xs) ys
                                (cons (car xs) (append (cdr xs) ys))))))
           (append ',x ',y))
        '(a b c d e)))
 '(
   (() (a b c d e))
   ((a) (b c d e))
   ((a b) (c d e))
   ((a b c) (d e))
   ((a b c d) (e))
   ((a b c d e) ())
   ))
```

Note how the wrapper around `append` gives us the output we expect from relational `appendo`.

There is an alternative approach to turning a function into a relation, which involves trying to synthesize the relational version by example (using the original function). We can start synthesizing the relational `appendo` from `append` as follows (this is Racket code):

```
1  (define-term-syntax-rule (append-sketch-and-calls hole)
2    `(letrec ([append
3               (lambda (xs ys)
4                 (if (null? xs) ys
5                     (cons ,hole (append (cdr xs) ys))))])
6       (list
7        (append '() '())
8        (append '(a) '(b))
9        (append '(c d) '(e f)))))
10
11 ;; unstaged version (slower)
12 (time
13  (run 1 (e)
14    (evalo-unstaged
15     (append-sketch-and-calls e)
16     '(() (a b) (c d e f)))))
17
18 ;; staged version (faster)
19 (time
20  (run 1 (e)
21    (time-staged
22     (evalo-staged
23      (append-sketch-and-calls e)
24      '(() (a b) (c d e f))))))
25 (generated-code)
```

The second, staged version is a bit faster, but running both result in the correct synthesis of the term represented by `,hole` (in line 5): `'((car xs))`.

4 Multi-stage Relational Programming

Talks about staged miniKanren as presented in "Multi-stage Relational Programming" from PLDI 2025.[2]. Staging:

[2] `https://dl.acm.org/doi/10.1145/3729314`

- First stage performs various deterministic optimizations
- Subsequent stages work with the result of the first stage to improve the generation.

5 Proof by Reflection

"Change theorem proving in the theory into evaluation in the metatheory. Reflection through metaprogramming is a rough tool that can give one a glimpse of what is possible computationally."[3]

A key philosophical shift from reflection as a meta-theoretical add-on to computation as a first-class citizen, recognizing that:

- Computation and deduction are complementary.
- Many proofs are just computation in disguise.
- The type system itself can encode the soundness of reflection.

Lean's dependent type theory unifies the object and meta levels of FOL. Both Prop and Bool are first-class types in Lean.

5.1 FOL vs. Lean

FOL requires manual attachment of computational semantics, while Lean's type theory has computation built into its core. The function isEven automatically computes during type checking.

FOL manually constructs the reflection infrastructure.

- Explicit representation of terms, formulas, and predicates.
- Manual axioms connecting syntactic and semantic levels.
- Complex axiom to enable reflection.

In contrast, Lean provides a systematic decidability framework. FOL reflection produces theorems as side effects. Lean's decide tactic produces actual proof terms that can be type-checked independently. The computation is the proof. The type system enforces the soundness of reflection.

[3] `https://io.livecode.ch/learn/namin/GETFOL`

5.2 Summary

To summarize, there is a discussion between theorem proving in Lean and using metaprogramming (with FOL). The idea is that proofs through metaprogramming is done by encoding our proofs as relations that are then unified through evaluation. While this works, it requires explicitly moving between meta-levels to reason about the proofs, while Lean encodes all of this directly through its type system, making all of this more "seamless" to the user.