William E. Byrd—Research Statement

---

1 Wishful Thinking, Lollipops, and Relational Programming

> When someone says "I want a programming language in which I need only say what I wish done," give him a lollipop. —Alan J. Perlis

I want a programming language in which I need only say what I wish done.

I am not the only one. Researchers and industry programmers are increasingly interested in *declarative programming*, in which programmers specify *what* to do, rather than *how* to do it. Perhaps the most common examples of declarative programming languages are database query languages, such as SQL, XQuery, and Datalog. The equational reasoning provided by the lazy functional language Haskell can also be seen as a form of declarative programming.

For decades logic programming has been heralded as the ultimate in declarative programming, with the promise of writing programs as mathematical relations that do not distinguish between "input" and "output" arguments. This is illustrated by the traditional "hello world!" example from logic programming, list concatenation. In the best-known logic programming language—Prolog—the call `append([1,2,3],[4,5],X)` produces the answer `X = [1,2,3,4,5]`. More impressively, the call `append(X,Y,[1,2,3,4,5])` can produce all pairs of lists `X` and `Y` such that `X ++ Y = [1,2,3,4,5]`.

Alas, most interesting Prolog programs are *not* written in a relational style. For reasons of efficiency and expressivity, Prolog programmers tend to use impure "extra-logical" features of the language, such as the cut (`!`), used to prune the search tree, and `assert`/`retract`, used to modify the global database of facts. These features make reasoning about logic programs more difficult, and can even result in unsound behavior. Even when these features are used correctly, they inhibit the ability to treat programs as relations.

In many ways, the current practice of logic programming is reminiscent of how functional programming was practiced before Haskell. Lisp and ML programmers understood the benefits of coding in a pure functional style; in a pinch, however, they could resort to variable mutation or other side-effects. As with Prolog, these impure operators were used for efficiency and expressivity. Haskell's laziness required a more pure approach, however, which in turn led to the adoption of monads for encapsulating effects. By committing to a pure functional style, Haskell advanced the theory and practice of functional programming, even for strict languages. *Constraints spur creativity.*

For the past decade my colleagues and I have been developing *miniKanren* (`minikanren.org/`), a logic programming language specifically designed for programming in a relational style. During that time I have developed relational interpreters, term reducers, type inferencers, and theorem provers, all of which are capable of program synthesis, or of otherwise running "backwards."

Over the past three years miniKanren has gained a significant following in the Clojure community in the form of the popular `core.logic` library, which started as a Clojure port of the code in my dissertation. `core.logic` is now being used in both industry and academia. miniKanren is also growing in popularity in the Racket community, in the form of the `cKanren` library. miniKanren has been ported from Scheme to many other languages, including Python, Ruby, JavaScript, Haskell, and Scala.

2 A Relational Scheme Interpreter

> I finally understood that the half page of code on the bottom of page 13 of the Lisp 1.5 manual was Lisp in itself. These were "Maxwells Equations of Software!" —Alan Kay

To understand the relational style of programming supported by miniKanren, it is best to consider an example. Figure 1 presents the structural operational semantics of an environment-passing interpreter for a small subset of Scheme: the call-by-value λ-calculus, extended with **quote** and *list*. Figure 2 presents the equivalent rules in miniKanren, while figure 3 presents several simple helper relations, mostly concerned with managing the lexical environment, ρ. The miniKanren language is described in Byrd et al. (2012), Byrd (2009), and Friedman et al. (2005). Without getting into the details of the language, miniKanren's key features for supporting relational programming include sound unification, complete search, relational arithmetic, nominal unification, and various constraint logic programming extensions.

$$\frac{\mathsf{quote} \notin dom(\rho)}{\rho \vdash (\mathsf{quote}\ d) \Downarrow d} \quad (\textsc{quote})$$

$$\frac{\begin{array}{c}\rho \vdash e^* \Downarrow v^* \\ \ldots \\ \mathsf{list} \notin dom(\rho)\end{array}}{\rho \vdash (\mathsf{list}\ e^*\ \ldots) \Downarrow v^*\ \ldots} \quad (\textsc{list})$$

$$\frac{(x : v) \in \rho}{\rho \vdash x \Downarrow v} \quad (\textsc{var})$$

$$\frac{\lambda \notin dom(\rho)}{\rho \vdash \lambda x.e \Downarrow <\lambda x.e\ \mathbf{in}\ \rho>} \quad (\textsc{abs})$$

$$\frac{\begin{array}{c}\rho \vdash e_1 \Downarrow <\lambda x.e\ \mathbf{in}\ \rho_1> \\ \rho \vdash e_2 \Downarrow v_2 \\ \rho_1, (x : v_2) \vdash e \Downarrow v\end{array}}{\rho \vdash (e_1\ e_2) \Downarrow v} \quad (\textsc{app})$$

Figure 1: Environment-passing Interpreter (shadowing allowed)

```
(define (eval-exp^o ρ expr v)
  (fresh ()
    (absent^o 'in expr)
    (match^e expr
      ((quote ,datum)
       (not-in-dom^o 'quote ρ) (≡ datum v))
      ((list . ,e*)
       (not-in-dom^o 'list ρ) (eval-list^o ρ e* v))
      (,x (symbol^o x) (lookup^o `(,x : ,v) ρ))
      ((λ (,x) ,e)
       (not-in-dom^o 'λ ρ) (≡ `((λ (,x) ,e) in ,ρ) v))
      ((,e1 ,e2)
       (fresh (x e ρ1 v2)
         (eval-exp^o ρ e1 `((λ (,x) ,e) in ,ρ1))
         (eval-exp^o ρ e2 v2)
         (eval-exp^o `((,x : ,v2) . ,ρ1) e v))))))
```

Figure 2: Environment-passing Interpreter (shadowing allowed, miniKanren)

Even without a knowledge of miniKanren, it should be evident that the code in figure 2 closely matches the inference rules from figure 1. More importantly, *eval-exp*^{o} uses no extra-logical features, and can be treated as a relation.

```
(define (eval^o expr v) (eval-exp^o '() expr v))
(define (eval-list^o ρ expr v)
  (match^e (expr v)
    ((() ()))
    (((,e . ,e*) (,ve . ,ve*))
     (eval-exp^o ρ e ve) (eval-list^o ρ e* ve*))))
```

```
(define (lookup^o binding ρ)
  (match^e (binding ρ)
    (((,x : ,T) ((,x : ,T) . ,_))
    (((,x : ,T) ((,y : ,_) . ,ρ1))
     (≢ x y) (lookup^o binding ρ1))))
(define (not-in-dom^o x ρ)
  (match^e ρ
    (())
    (((,y . ,v) . ,ρ1) (≢ y x) (not-in-dom^o x ρ1))))
```

Figure 3: Interpreter Helper Relations (miniKanren)

2.1 I Love You

What can we do with the relational interpreter? In his blog post, "99 ways to say '(I love you)," [1] my friend and current advisor Matt Might challenges readers to create 99 expressions in Racket or Scheme that evaluate to the list (I love you). Readers of his blog emailed him hand-crafted programs; this is fun but tedious. More fun, and much less tedious, is to use our relational interpreter to generate 99 programs for us:

```
(run^99 (q) (eval^o q '(I love you)))
```

It takes miniKanren about 20 milliseconds to generate 99 programs that evaluate to (I love you). The first answer produced is '(I love you); a more interesting answer is

```
(list 'I 'love ((λ (_.0) (_.0 '_.1)) (λ (_.2) 'you))),
```

where _.0, _.1, and _.2 are miniKanren's way of representing unbound logic variables. Since _.0, _.1, and _.2 are legal symbols in Scheme, this expression can be evaluated in Scheme without modification.

On my laptop is a file containing a million (I love you) Scheme programs generated by miniKanren. This is too easy—let us try something harder.

2.2 A Challenge from McCarthy

McCarthy (1978) posed this problem in his description of *value*, a minimal LISP interpreter:

> Difficult mathematical type exercise: Find a list e such that *value* $e = e$.

Such lists, which are also programs in LISP, Scheme, and Racket, are called *quines*. That is, a quine is a program that evaluates to itself. A classic non-trivial quine in Scheme (Thompson II) is:

```
(define quine_c
  '((λ (x) (list x (list (quote quote) x)))
    (quote (λ (x) (list x (list (quote quote) x))))))
```

[1] http://matt.might.net/articles/i-love-you-in-racket/

There are mathematical techniques for constructing quines, based on Kleene's Second Recursion Theorem (Rogers 1967). However, there is an alternate approach: let the interpreter solve the "difficult mathematical type exercise" for us! We can do this by associating both the "input" expression and the "output" value with the query variable q:

$(caar\ (\mathbf{run^1}\ (q)\ (eval^o\ q\ q))) \Rightarrow$
$((\lambda\ (_{-0})\ (\mathsf{list}\ _{-0}\ (\mathsf{list}\ '\mathsf{quote}\ _{-0})))$
$\ '(\lambda\ (_{-0})\ (\mathsf{list}\ _{-0}\ (\mathsf{list}\ '\mathsf{quote}\ _{-0}))))$

This quine, generated in 20 milliseconds on my laptop, is equivalent to $quine_c$, above. Generating 20 quines takes just over one second.

We can push things further by attempting to generate *thrines*: distinct programs p, q, and r such that $(eval\ p) \Rightarrow q$, $(eval\ q) \Rightarrow r$, and $(eval\ r) \Rightarrow p$.

$(\mathbf{run^1}\ (x)$
$\ (\mathbf{fresh}\ (p\ q\ r)$
$\ \ (\not\equiv\ p\ q)\ (\not\equiv\ q\ r)\ (\not\equiv\ r\ p)$
$\ \ (eval^o\ p\ q)\ (eval^o\ q\ r)\ (eval^o\ r\ p)$
$\ \ (\equiv\ `(,p\ ,q\ ,r)\ x)))$

This thrines-producing query takes a little over two seconds on my laptop.

3 Combinatory Logic in miniKanren

This approach to quine generation demonstrates the promise of program synthesis using very high-level relational interpreters and term reducers. Recently I have been exploring a simpler approach to relational program synthesis, based on combinatory logic (Schönfinkel 1924; Curry and Feys 1958; Bimbó 2012) rather than Scheme. (Figures 4–11.) Combinatory logic does not include a notion of variable binding, which greatly simplifies the implementation of a relational term reducer. The resulting reducer is extremely succinct—just 11 lines of miniKanren code—and can synthesize combinators, including fixpoint combinators, directly from their definitions.

3.1 (Some) Homework for Free

Let us consider an exercise from a standard textbook on combinatory logic (Hindley and Seldin 2008). This exercise[2], which is marked "Tricky" in the text, asks the reader to construct combinatory logic terms **B'** and **W** that satisfy $\mathbf{B'}xyz \triangleright_w y(xz)$ and $\mathbf{W}xy \triangleright_w xyy$.

We can find a term in the standard {**S K I**} combinator basis that satisfies the definition of **W** using the query:

$(\mathbf{run^1}\ (W)\ (\mathbf{eigen}\ (x\ y)\ (\triangleright_w^o\ `((,W\ ,x)\ ,y)\ `((,x\ ,y)\ ,y)))) \Rightarrow (((\mathsf{S}\ \mathsf{S})\ (\mathsf{S}\ \mathsf{K})))$

This query returns in 3 milliseconds.

The query for **B'** looks similar:

$(\mathbf{run^1}\ (\hat{B})\ (\mathbf{eigen}\ (x\ y\ z)\ (\triangleright_w^o\ `(((,\hat{B}\ ,x)\ ,y)\ ,z)\ `(,y\ (,x\ ,z)))))$

[2]Exercise 2.17b on p. 26.

Unlike the previous query, this query does not return, even after running for many minutes. As with other forms of program synthesis, not all queries are equally expensive, even if they look similar syntactically. Still, we can be happy that miniKanren solved half of our homework problem for us, after we spent only a few seconds typing in a query.

Now that we are warmed up, we are ready to generate more interesting programs.

3.2 Synthesizing Fixpoint Combinators

The problem that led us to consider combinatory logic is that of synthesizing fixpoint combinators. Barendregt (1984) gives the following definition for a fixpoint combinator, F:

$$\exists F.\forall X.FX = X(FX)$$

We can express this definition as the query:

```
(run¹ (F) (eigen (X) (⊳ᵒ_w `(,F ,X) `(,X (,F ,X))))) ⇒
(((S I) (((S (S (K (S I)))) I) ((S (S (K (S I)))) I))))
```

This query returns after roughly eight minutes. If we take an educated guess, and assume there may be a fixpoint combinator F that is equal to another combinator U applied to itself (that is, $F = UU$), we end up with the query

```
(run 1 (F)
  (fresh (U)
    (eigen (X)
      (≡ `(,U ,U) F)
      (⊳ᵒ_w `(,F ,X) `(,X (,F ,X))))))
⇒
((((S (S (K (S I)))) I)
  ((S (S (K (S I)))) I)))
```

The self-application hint $F = UU$ reduces the running time from 8 minutes to about 20 seconds.

We can even generate a fixpoint combinator in combinatory logic, translate it to a term in the call-by-value λ-calculus, and then use the resulting combinator to run factorial in Scheme:

```
(let ((F_V (eval (car (run 1 (F_V)
                        (fresh (F)
                          (eigen (X)
                            (⊳ᵒ_w `(,F ,X) `(,X (,F ,X)))
                            (L°_η F F_V)))))
                 (environment '(rnrs)))))
  ((F_V (λ (f)
          (λ (n)
            (if (= n 0)
                1
                (* n (f (- n 1)))))))
   5)) ⇒ 120
```

$$\mathsf{I}x \triangleright x \qquad (\triangleright\text{-}\mathsf{I})$$

$$\mathsf{K}xy \triangleright x \qquad (\triangleright\text{-}\mathsf{K})$$

$$\mathsf{S}xyz \triangleright xz(yz) \qquad (\triangleright\text{-}\mathsf{S})$$

Figure 4: Contraction

$$\frac{M \triangleright M'}{M \triangleright_{1w} M'} \ (\triangleright_{1w}\text{-CONTRACTION})$$

$$\frac{M \triangleright_{1w} M'}{MN \triangleright_{1w} M'N} \qquad (\triangleright_{1w}\text{-LEFT})$$

$$\frac{N \triangleright_{1w} N'}{MN \triangleright_{1w} MN'} \qquad (\triangleright_{1w}\text{-RIGHT})$$

Figure 5: One-step Reduction

$$M \triangleright_w M \qquad (\triangleright_w\text{-REFLEXIVE})$$

$$\frac{M \triangleright_{1w} N \quad N \triangleright_w P}{M \triangleright_w P} \ (\triangleright_w\text{-TRANSITIVE})$$

Figure 6: Weak Reduction

$$\mathsf{I}\ L_\eta\ \lambda x.x \qquad (L_\eta\text{-}\mathsf{I})$$

$$\mathsf{K}\ L_\eta\ \lambda xy.x \qquad (L_\eta\text{-}\mathsf{K})$$

$$\mathsf{S}\ L_\eta\ \lambda xyzw.(\lambda v.xzv\ \lambda v.yzv)w \qquad (L_\eta\text{-}\mathsf{S})$$

$$\frac{M\ L_\eta\ M' \quad N\ L_\eta\ N'}{MN\ L_\eta\ M'N'} \ (L_\eta\text{-COMPOUND})$$

Figure 7: **SKI**-to-Call-by-Value λ-Calculus

```
(defmatch^e (▷^o T T̂)
  (((I ,x) ,x))
  ((((K ,x) ,y) ,x))
  (((((S ,x) ,y) ,z) ((,x ,z) (,y ,z)))))
```

Figure 8: Contraction (miniKanren)

```
(defmatch^e (▷^o_1w T T̂)
  ((,M ,M̂) (▷^o M M̂))
  (((,M ,N) (,M̂ ,N)) (▷^o_1w M M̂))
  (((,M ,N) (,M ,N̂)) (▷^o_1w N N̂)))
```

Figure 9: One-step Reduction (miniKanren)

```
(defmatch^e (▷^o_w T T̂)
  ((,M ,M))
  ((,M ,P) (fresh (N) (▷^o_1w M N) (▷^o_w N P))))
```

Figure 10: Weak Reduction (miniKanren)

```
(defmatch^e (L^o_η T T̂)
  ((I (λ (x) x)))
  ((K (λ (x) (λ (y) x))))
  ((S (λ (x)
        (λ (y)
          (λ (z)
            (λ (w)
              (((λ (v) ((x z) v))
                (λ (v) ((y z) v)))
               w)))))))
  (((,M ,N) (,M̂ ,N̂)) (L^o_η M M̂) (L^o_η N N̂)))
```

Figure 11: **SKI**-to-CBV λ-Calculus (miniKanren)

4 The Long Run

> Often it is the means that justify the ends: Goals advance technique and technique survives even when goal structures crumble. —Alan J. Perlis

The approach to relational programming presented above works for a variety of other interesting programs, including theorem provers and type inferencers. In fact, my colleagues and I wrote the original type inferencer for Harlan—a high-level language for GPU programming—in miniKanren. Unfortunately, we found that miniKanren has its limitation as a tool for writing type inferencers. In particular, miniKanren is not yet expressive enough to implement Harlan's region-based memory management system (which is built into Harlan's type system), and the type errors returned by miniKanren are ... non-existent. These problems could be addressed by adding some form of region constraints to miniKanren, and by using Prolog-style meta-interpreter techniques to generate error messages. Alas, these meta-interpreter techniques themselves are non-relational—writing Prolog style meta-interpreters in a relational style is a problem I am currently working on with Nada Amin. The problem of writing a relational type inferencer for an interesting type system is illustrative of the challenge of writing larger and more complex programs in a relational style.

Relational programs are remarkably flexible. Writing programs in a relational style is remarkably difficult, and requires the support of a language like miniKanren that fully commits to this style of programming. Just as Haskell's commitment to laziness and purity led to advances in the theory and practice of functional programming, I believe miniKanren's commitment to relational programming will lead to advances in the theory and practice of logic programming.

References

Hendrik Pieter Barendregt. *The Lambda Calculus – Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1984.

Katalin Bimbó. *Combinatory Logic: Pure, Applied, and Typed*. Discrete Mathematics and its Applications. CRC Press, Boca Raton, FL, USA, 2012.

William E. Byrd. *Relational Programming in miniKanren: Techniques, Applications, and Implementations*. PhD thesis, Indiana University, 2009.

William E. Byrd, Eric Holk, and Daniel P. Friedman. miniKanren, live and untagged: Quine generation via relational interpreters (programming pearl). In *2012 Workshop on Scheme and Functional Programming*, September 2012.

H. B. Curry and R. Feys. *Combinatory Logic, Volume I*. North-Holland, 1958. Second printing 1968.

Daniel P. Friedman, William E. Byrd, and Oleg Kiselyov. *The Reasoned Schemer*. The MIT Press, Cambridge, MA, 2005.

J. Roger Hindley and Jonathan P. Seldin. *Lambda-Calculus and Combinators: An Introduction*. Cambridge University Press, New York, NY, USA, second edition, 2008.

John McCarthy. A micro-manual for lisp - not the whole truth. *SIGPLAN Not.*, 13(8):215–216, August 1978.

Hartley Rogers, Jr. *Theory of recursive functions and effective computability.* McGraw-Hill, New York, NY, 1967.

Moses Schönfinkel. On the Building Blocks of Mathematical Logic. In Jean van Heijenoort, editor, *From Frege to Gödel: A Source Book in Mathematical Logic, 1879–1931*, pages 355–366. 1924.

Gary P. Thompson II. The quine page (self-reproducing code). `http://www.nyx.org/~gthompso/quine.htm`.