Relational Synthesis of Programs

(Author names omitted for submission)

Abstract

Recent work has shown how to use constraint logic programming to relationally define an interpreter that runs "backwards," mapping an output value (such as 6) onto a potentially infinite stream of input expressions (such as $((\lambda\ (n)\ (*\ n\ 2))\ 3)$ and $(+\ 5\ 1)$). Surprisingly, a trivial one-line query causes the interpreter to efficiently generate programs that evaluate to themselves ("quines"). Naturally, one wonders whether this approach to program synthesis can do more than generate quines.

We demonstrate a simpler, more flexible approach to relational program synthesis, using a relational reducer for combinatory logic. Combinatory logic has no notion of variable binding, and can be implemented in any logic programming language that supports complete search and universal quantification. The relational reducer is extremely terse: 11 lines of code, each of which corresponds to a mathematical rule from combinatory logic. The reducer can synthesize combinators directly from their logical definitions, providing program synthesis "for free"; for example, the query $\exists F.\forall X.FX = X(FX)$ generates a fixpoint combinator F. We show that sharing between input and output arguments vastly reduces the search space, making combinator synthesis practical.

Categories and Subject Descriptors I.2.2 [*Automatic Programming*]: Program synthesis; D.1.6 [*Programming Techniques*]: Logic Programming; D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming

General Terms Languages

Keywords combinatory logic, miniKanren, relational programming

1. Introduction

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*. In other words, a quine is a program that evaluates to itself (Hofstadter 1979). A classic non-trivial quine in Scheme (Thompson II) is:

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

There are mathematical techniques for constructing quines, based on Kleene's Second Recursion Theorem (Rogers 1967). Recently Byrd et al. (2012) proposed an alternate approach: let the interpreter solve the "difficult mathematical type exercise" for us! Their approach was to use a logic programming language, miniKanren (Byrd 2009; Byrd and Friedman 2006; Friedman et al. 2005), extended with various constraints, to define an interpreter as a relation. The resulting evaluation relation, *eval-exp^o*, could then be used to generate quines, by associating both the "input" expression and the "output" value with the query variable q:

```
(caar (run¹ (q) (eval-expᵒ q '() q))) ⇒
((λ (_₀) (list _₀ (list 'quote _₀)))
 '(λ (_₀) (list _₀ (list 'quote _₀))))
```

This quine, generated in approximately one second on a laptop computer, is equivalent to *quinec*, above.

This approach to quine generation demonstrates the promise of program synthesis using very high-level relational interpreters and term reducers. Unfortunately, the *eval-exp^o* has several drawbacks. The implementation of the interpreter is subtle, and requires that miniKanren be extended with various constraints. Most importantly, *eval-exp^o* is implemented as a *big-step* interpreter, which reduces an expression all the way to a value. This makes it impossible to synthesize certain interesting classes of programs, such as fixpoint combinators (Barendregt 1984).

In this paper we explore a simpler approach to relational program synthesis, based on combinatory logic (Bimbó 2012; Curry and Feys 1958; Schönfinkel 1924) rather than Scheme. 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.

Our paper is structured as follows. Section 2 provides an overview of the miniKanren language, and describes the two language extensions we will require. Section 3 gives a brief overview of combinatory logic. Section 4 explains our translation of combinatory logic into miniKanren, and gives examples of combinator synthesis. Section 5 discusses performance. Section 6 contains our concluding thoughts, including a discussion of how our approach differs from that of Byrd et al. (2012).

We begin by introducing the extended miniKanren language that we will use to write the relational term reducer.

2. The Extended miniKanren Language

In this section we briefly review the core miniKanren language (section 2.1), then introduce **eigen**, a new operator for expressing universal quantification using eigenvariables (section 2.2). We conclude with a description of the **defmatch**^{e} pattern-matching syntax (section 2.3), used to make the relational reducer more succinct.

Readers already familiar with miniKanren can safely skip to section 2.2 to learn about **eigen**, while those wishing to learn more about miniKanren should see Byrd (2009), Byrd and Friedman (2006) (from which this subsection has been adapted), and Friedman et al. (2005).

2.1 miniKanren Refresher

Our code uses the following typographic conventions. Lexical variables are in *italic*, forms are in **boldface**, and quoted symbols are in sans serif. By our convention, names of relations end with a superscript *o*—for example *any*^{o}, which is entered as `anyo`. Some relational operators do not follow this convention: ≡ (entered as `==`), **cond**^{e} (entered as `conde`), and **fresh**. Similarly, (**run**^{5} (*q*) *body*) and (**run*** (*q*) *body*) are entered as `(run 5 (q) body)` and `(run* (q) body)`.[1]

Core miniKanren extends Scheme with three operators: ≡, **fresh**, and **cond**^{e}. (One additional operator, **eigen**, is introduced in section 2.2.) There is also **run**, which serves as an interface between Scheme and miniKanren, and whose value is a list.

≡ unifies two terms. **fresh**, which syntactically looks like **lambda**, introduces lexically-scoped Scheme variables that are bound to new logic variables; **fresh** also performs conjunction of the relations within its body. Thus

```
(fresh (x y z) (≡ x z) (≡ 3 y))
```

would introduce logic variables x, y, and z, then associate x with z and y with 3. This, however, is not a legal miniKanren program—we must wrap a **run** around the entire expression.

```
(run^1 (q) (fresh (x y z) (≡ x z) (≡ 3 y))) ⇒ (_0)
```

The value returned is a list containing the single value $\_{0}$; we say that $\_{0}$ is the *reified value* of the unbound query variable q and thus represents any value. q also remains unbound in

```
(run^1 (q) (fresh (x y) (≡ x q) (≡ 3 y))) ⇒ (_0)
```

We can get back more interesting values by unifying the query variable with another term.

```
(run^1 (y)        (run^1 (q)        (run^1 (y)
  (fresh (x z)      (fresh (x z)      (fresh (x y)
    (≡ x z)           (≡ x z)           (≡ 4 x)
    (≡ 3 y)))         (≡ 3 z)           (≡ x y))
                      (≡ q x)))       (≡ 3 y))
```

Each of these examples returns (3); in the rightmost example, the y introduced by **fresh** is different from the y introduced by **run**.

A **run** expression can return the empty list, indicating that the body of the expression is logically inconsistent.

```
(run^1 (x) (≡ 4 3)) ⇒ ()
```

[1] It is conventional in Scheme for the names of predicates to end with the '?' character. We have therefore chosen to end the names of miniKanren goals with a superscript *o*, which is meant to resemble the top of a '?'. The superscript *e* in **cond**^{e} stands for 'every,' since every **cond**^{e} clause may contribute answers.

```
(run^1 (x) (≡ 5 x) (≡ 6 x)) ⇒ ()
```

We say that a logically inconsistent relation *fails*, while a logically consistent relation, such as (≡ 3 3), *succeeds.*

cond^e, which resembles **cond** syntactically, is used to produce multiple answers. Logically, **cond**^{e} can be thought of as disjunctive normal form: each clause represents a disjunct, and is independent of the other clauses, with the relations within a clause acting as the conjuncts. For example, this expression produces two answers.

```
(run^2 (q)
  (fresh (w x y)
    (cond^e
      ((≡ `(,x ,w ,x) q)
       (≡ y w))
      ((≡ `(,w ,x ,w) q)
       (≡ y w))))) ⇒ ((_0 _1 _0) (_0 _1 _0))
```

Although the two **cond**^{e} lines are different, the values returned are identical. This is because distinct reified unbound variables are assigned distinct subscripts, increasing from left to right—the numbering starts over again from zero within each answer, which is why the reified value of x is $\_{0}$ in the first answer but $\_{1}$ in the second. The superscript 2 in **run** denotes the maximum length of the resultant list. If the superscript $*$ is used, then there is no maximum imposed. This can easily lead to infinite loops.

```
(run* (q)
  (let loop ()
    (cond^e
      ((≡ #f q))
      ((≡ #t q))
      ((loop))))) ⇒ ⊥
```

If we replace $*$ by a natural number n, then an n-element list of alternating #f's and #t's is returned. The first answer is produced by the first **cond**^{e} clause, which associates q with #f. To produce the second answer, the second **cond**^{e} clause is tried. Since **cond**^{e} clauses are independent, the association between q and #f made in the first clause is forgotten—we say that q has been *refreshed.* In the third **cond**^{e} clause, q is refreshed again.

We now look at several interesting examples that rely on *any*^{o}, which tries g an unbounded number of times.

```
(define any^o
  (λ (g)
    (cond^e
      (g)
      ((any^o g)))))
```

Consider the first example,

```
(run* (q)
  (cond^e
    ((any^o (≡ #f q)))
    ((≡ #t q))))
```

which does not terminate because the call to *any*^{o} succeeds an unbounded number of times. If $*$ were replaced by 5, then we would get (#t #f #f #f #f). (The user should not be concerned with the order of the answers produced.)

Now consider

```
(run^10 (q)
  (any^o
  (cond^e
    ((≡ 1 q))
    ((≡ 2 q))
    ((≡ 3 q))))) ⇒ (1 2 3 1 2 3 1 2 3 1)
```

Here the values 1, 2, and 3 are interleaved; our use of any^o ensures that this sequence is repeated indefinitely.

Even if a relation within a $\mathbf{cond}^e$ clause loops indefinitely (or *diverges*), other $\mathbf{cond}^e$ clauses can contribute to the answers returned by a **run** expression. For example,

```
(run^3 (q)
  (let ((never^o (any^o (≡ #f #t))))
    (cond^e
      ((≡ 1 q))
      (never^o)
      ((cond^e
         ((≡ 2 q))
         (never^o)
         ((≡ 3 q)))))))
```

returns (1 2 3). Replacing $\mathbf{run}^{\mathbf{3}}$ with $\mathbf{run}^{\mathbf{4}}$ would cause divergence, since $never^o$ would loop indefinitely looking for the non-existent fourth answer.

2.2 Eigenvariables and Universal Quantification

We extend core miniKanren with a new operator used to express universal quantification, **eigen**. Syntactically, **eigen** is similar to **fresh**, but introduces eigenvariables rather than fresh logic variables.[2] An *eigenvariable* can be thought of as a lexically-scoped gensym (generated symbol): an eigenvariable e unifies only with itself, or with a fresh logic variable introduced *inside* the scope of e.

We can represent the logical formula $\forall X.X = X$ as the miniKanren goal (**eigen** (X) $(\equiv X\ X)$), which succeeds as expected, since any eigenvariable can unify with itself.

We can represent $\forall X.\exists Y.X = Y$ as the goal

(**eigen** (X) (**fresh** (Y) $(\equiv X\ Y)$)),[3]

which succeeds as expected, since the eigenvariable X can unify with any fresh logic variable introduced inside its scope (such as Y).

We can represent $\exists X.\forall Y.X = Y$ as the goal

(**fresh** (X) (**eigen** (Y) $(\equiv X\ Y)$)).

This goal fails, as expected, since the eigenvariable Y cannot unify with a fresh logic variable introduced outside of its scope (such as X).[4]

Compound terms containing eigenvariables can be unified with other terms as usual, provided that the eigenvariables follow the rules described above. In addition, a fresh logic variable x must never be associated with a term containing an eigenvariable introduced *inside* the scope of x. For example,

(**eigen** (A) (**fresh** (X) $(\equiv$ `(1 2 3 ,A 4) $X)$)),

succeeds, since the eigenvariable A is introduced outside the scope of X, but

(**fresh** (X) (**eigen** (A) $(\equiv$ `(1 2 3 ,A 4) $X)$))

fails, since A is introduced inside the scope of X.

Eigenvariables can never escape their scope, and therefore can never appear in the reified output of a **run** expression.

2.3 Pattern-matching Syntax

Syntactically, miniKanren's $\mathbf{cond}^e$, **fresh**, and $\equiv$ operators are designed to resemble Scheme's **cond**, **let**, and =. miniKanren programs can often be written more succinctly using a pattern-matching syntax that can implicitly introduce fresh logic variables, and implicitly performs disjunction and unification.[5]

In this paper we use $\mathbf{defmatch}^e$, a variant of the pattern matching syntax presented in Keep et al. (2009):

```
(defmatch^e (<relation-name> <id> ...)
  (<pattern> <goal> ...)
  <clause>
  ...
  )
```

$\mathbf{defmatch}^e$ defines a new top-level relation, with zero or more formal parameters <*id*> .... The body of the relation comprises one or more <*clause*>s, each of which is of the form (<*pattern*> <*goal*> ...). The <*pattern*> of a clause is any legal miniKanren term. The pattern is implicitly quasiquoted; any unquoted identifiers in the pattern are treated as fresh logic variables, whose scope is limited to the pattern and goals of that clause. The right-hand-side of each clause comprises zero or more miniKanren <*goal*>s. The formal parameters <*id*> ... are visible within the right-hand-side goals of all clauses, but not within the patterns.

Semantically, the body of a $\mathbf{defmatch}^e$ represents a disjunction of clauses, with each clause representing a conjunction of its pattern and right-hand-side goals. A clause succeeds if its pattern successfully unifies with the current values of the formal parameters, and all right-hand-side goals in the clause succeed (in the context of any new associations for the logic variables in the pattern caused by the pattern's unification).

As an example, we use $\mathbf{defmatch}^e$ to succinctly define $append^o$, a relation on three lists l_1, l_2, and l_3 that succeeds if l_1 ++ $l_2 = l_3$.

```
(run* (q) (append^o '(w v x) q '(w v x y z))) ⇒ ((y z))

(defmatch^e (append^o l1 l2 l3)
  ((() ,l ,l))
  (((,a . ,d) ,s (,a . ,res)) (append^o d s res)))
```

The first clause of $append^o$ associates l_2 with l_3 if l_1 unifies with the empty list. The pattern of the second clause matches if l_1 unifies with the pair of fresh logic variables,

[2] More on eigenvariables, in the context of λProlog, can be found in Miller and Nadathur (2012).

[3] The **fresh** operator was originally named **exist**, but the name was changed to avoid confusion with the R^6RS Scheme standard's *exists* function (also known as *ormap*). For a similar reason **eigen** is not named **forall**.

[4] The interaction between eigenvariables and the disequality ($\not\equiv$) and $absent^o$ constraints described in Byrd et al. (2012) is complex, and is not handled by the current implementation of **eigen**. Fortunately, the combinatory logic reducer can be implemented using only unification, without requiring the $\not\equiv$, $absent^o$, $symbol^o$, or $number^o$ constraints used in the relational interpreter of Byrd et al. (2012)—indeed, this is one of the advantages of our approach. We will need **eigen** only for queries to the reducer, not for the definition of the reducer.

[5] Our pattern-matching syntax is similar to that of Prolog. The code for the combinatory logic reducer can easily be translated to any Prolog implementation that includes universal quantification and an efficient implementation of complete search, both of which are needed for combinator synthesis.

‘(,a . ,d), and l_3 unifies with the pair of fresh logic variables, ‘(,a . ,res). These unifications associate the first element of l_1 with the first element of l_3. $append^o$ is then called recursively on the remaining elements of l_1, the entire list l_2, and the remaining elements of l_3.

The **defmatch**^{e} definition of $append^o$ is semantically equivalent to this more verbose definition (although **defmatch**^{e} may produce more efficient code):

```
(define appendᵒ
  (λ (l₁ l₂ l₃)
    (condᵉ
      ((fresh (l)
         (≡ '() l₁)
         (≡ l l₂)
         (≡ l l₃)))
      ((fresh (a d s res)
         (≡ `(,a . ,d) l₁)
         (≡ s l₂)
         (≡ `(,a . ,res) l₃)
         (appendᵒ d s res))))))
```

As we will see in section 4, the addition of universal quantification (**eigen**) and pattern-matching syntax (**defmatch**^{e}) essentially turns miniKanren into a Domain-Specific Language for implementing the rules of combinatory logic in a relational fashion. Any logic programming language with universal quantification, pattern-matching, and an efficient implementation of complete search would also be suitable for this purpose.

Next we present a brief overview of combinatory logic, and then show how combinatory logic can be implemented in miniKanren.

3. Combinatory Logic

Introduced by Schönfinkel in the early 1920's, combinatory logic is perhaps the oldest formalism for expressing computable functions, preceding both the λ-calculus and Turing Machines, but equivalent in power to both (Schönfinkel 1924). Standard texts on combinatory logic include Curry and Feys (1958), Curry et al. (1972), Stenlund (1972), Revesz (1988), Hindley and Seldin (2008), and Bimbó (2012).

In the λ-calculus, a *combinator* is function with no free variables—for example, the identity function $\lambda x.x$. Unlike in the λ-calculus, combinatory logic has no notion of variable binding; instead, the identity combinator is represented as the constant **I**, with an associated *contraction axiom*, $\mathsf{I}x \triangleright x$. Two other standard combinators, **K** and **S**, have the contraction axioms $\mathsf{K}xy \triangleright x$ and $\mathsf{S}xyz \triangleright xz(yz)$, respectively.

A term in combinatory logic is either a combinator constant (such as **I**), or an *application* MN of two terms M and N. Parentheses are used to express grouping of pairs of terms. By convention, application is left-associative: the term **KSK** is equivalent to (**KS**)**K**.

A set of combinator constants forms a *basis*. The basis {**S K I**} is *complete*, in that any computable function can be represented as a term containing only the constants **S**, **K**, and **I**. The basis {**S K**} is also complete—that is, **I** can be represented as a term containing only the constants **S** and **K** (as we shall see below). The basis {**K I**} is *not* complete, however, as **S** cannot be represented as a term containing only the constants **K** and **I**.

The contraction axioms can be used to rewrite terms. For example, the axiom for the **S** combinator is $\mathsf{S}xyz \triangleright xz(yz)$. By this axiom, the term **SK(IK)S** can be contracted, with x matching **K**, y matching **IK**, and z matching **S**.[6] The resulting contracted term is $xz(yz)$—in this case, **KS(IKS)**.[7] A contraction axiom can only be applied when the combinator is supplied with sufficiently many "argument" terms. For example, the term **SSK** cannot be contracted using the axiom for **S**, since **S** must be supplied with three terms.

In addition to the contraction axioms, we will use the standard one-step reduction and weak reduction relations from combinatory logic. The one-step reduction relation $\triangleright_{1w}$ performs a single contraction, anywhere in a term. The weak reduction relation $\triangleright_w$ is the reflexive, transitive closure of $\triangleright_{1w}$. Figures 1, 2, and 3 show the contraction axioms for the {**S K I**} basis, the $\triangleright_{1w}$ relation, and the $\triangleright_w$ relation, all written as inference rules.

3.1 Combinatory Logic and the λ-Calculus

Combinatory logic has strong connections to the λ-calculus. Any combinatory logic term can be translated to an equivalent λ-calculus term, and the other way around. Our approach to combinator synthesis is performed entirely within combinatory logic; however, it can be useful to convert synthesized terms into equivalent λ-calculus terms. Translating from λ-calculus to combinatory logic does not appear as useful; we therefore do not include the standard "bracket" abstraction rules for translating in this direction.

Translating a term from combinatory logic to call-by-name λ-calculus is obvious: replace each combinator constant in the term with a λ-calculus term equivalent to that combinator's contraction algorithm. For example, the axiom for the **S** combinator is $\mathsf{S}xyz \triangleright xz(yz)$; therefore, each occurrence of **S** is replaced with $\lambda xyz.xz(yz)$. The combinatory logic term **S**(**KK**) would therefore become $(\lambda xyz.xz(yz))\ (\lambda xy.x\ \lambda xy.x)$.

Translating a term from combinatory logic to call-by-value λ-calculus is slightly trickier, since not all terms with normal forms in call-by-name λ-calculus have normal forms in call-by-value λ-calculus. For example, in the call-by-name λ-calculus, Curry's fixpoint combinator **Y** (Barendregt 1984) is defined as $\lambda f.(\lambda x.f(xx))\ (\lambda x.f(xx))$. However, applying **Y** to any term in the call-by-value λ-calculus results in a term with no normal form, which causes divergence when evaluated in a call-by-value language such as Scheme. This divergence is caused by the self-applications (xx) present in **Y**. To avoid this problem, we can perform an η-*expansion*[8] around each of the self-applications in **Y**, resulting in the call-by-value fixpoint combinator **Z**: $\lambda f.(\lambda x.f(\lambda v.(xx)v))\ (\lambda x.f(\lambda v.(xx)v))$.

We observe that, for the {**S K I**} basis, only contractions of **S** can introduce self-application. Therefore, we can avoid premature divergence in the call-by-value λ-calculus by performing η-expansions within the body of the λ-term cor-

[6] Given the left-associativity of application, the term **SK(IK)S** is equivalent to **((SK)(IK))S**.

[7] The name "contraction" is somewhat misleading, in that the resulting term may be smaller, larger, or the same size as the original term.

[8] The η-equivalence rule in the λ-calculus states that $M =_\eta \lambda x.Mx$, provided that x does not occur free in M (Barendregt 1984). For example, η-expanding the unary function *add1* in Scheme yields (λ (n) (*add1* n)). If M has no normal form in the call-by-value λ-calculus, η-expansion ensures the resulting term has a normal form (and therefore does not diverge in a call-by-value language like Scheme).

responding to **S**: $\lambda xyzw.(\lambda v.xzv\ \lambda v.yzv)w$. The conversion rules for **K** and **I** remain unchanged.

Figures 4 and 5 show the inference rules for converting combinatory logic terms to the call-by-name and call-by-value λ-calculus, respectively.

4. Combinatory Logic in miniKanren

We are now ready to implement the combinatory logic relations from section 3 in miniKanren. We begin with the three critical relations from combinatory logic: $\rhd$, $\rhd_{1w}$, and $\rhd_w$.

The binary relation $\rhd$ in figure 1 represents the contraction axioms for the {**S K I**} basis; these inference rules can be directly translated into the binary relation $\rhd^o$ in figure 6.[9] The $\rhd^o$ relation succeeds if its first argument T unifies with one of the terms ‘(I ,x), ‘((K ,x) ,y), or ‘(((S ,x) ,y) ,z), (equivalent to the left-hand-sides of $\rhd$: **I**x, **K**xy, and **S**xyz), and its second argument $\hat{T}$ unifies with the corresponding term x, x, or ‘((,x ,z) (,y ,z)) (equivalent to the right-hand-sides of $\rhd$: x, x, and $xz(yz)$). miniKanren uses Scheme s-expressions to represent terms, which accounts for the fully-parenthesized syntax. Also, recall from section 2.3 that **defmatch**^{e} patterns are implicitly quasiquoted.

Similarly, the three inference rules of the binary relation $\rhd_{1w}$ in figure 2 correspond to the three **defmatch**^{e} clauses of the binary relation $\rhd_{1w}^o$ in figure 7.[10] The pattern portion of a **defmatch**^{e} clause corresponds to the part of the inference rule below the horizontal line (the *consequent*), while recursive calls (or calls to other miniKanren relations) correspond to the part above the horizontal line (the *antecedent*).

The two inference rules of the binary relation $\rhd_w$ in figure 3 correspond to the three **defmatch**^{e} clauses of the binary relation $\rhd_w^o$ in figure 8.[11] The second **defmatch**^{e} clause requires a use of **fresh** to introduce a local logic variable N used in the transitive closure of $\rhd_{1w}^o$.

The 11 lines of miniKanren defining the relations $\rhd$, $\rhd_{1w}$, and $\rhd_w$ allow us to perform program synthesis, as we will see in section 4.1. We can also define miniKanren relations for translating from combinatory logic to λ-calculus. The L^o miniKanren relation in figure 9 corresponds to the call-by-name L rules in figure 4, while the L_η^o relation in figure 10 corresponds to the call-by-value L_η rules in figure 5.[12]

4.1 (Some) Homework for Free

We are finally ready to synthesize programs in combinatory logic. We begin with an exercise from a standard textbook on combinatory logic (Hindley and Seldin 2008). This exercise[13], which is marked “Tricky” in the text, asks the reader to construct combinatory logic terms **B’** and **W** that satisfy **B’**$xyz \rhd_w y(xz)$ and **W**$xy \rhd_w xyy$.

We can find a term that satisfies the definition of **W** using the query:

(**run**$^\mathbf{1}$ (W) (**eigen** $(x\ y)$ ($\rhd_w^o$ ‘((,W ,x) ,y) ‘((,x ,y) ,y)))) $\Rightarrow$ (((S S) (S K)))

This query returns in 3 milliseconds.[14]

The query for **B’** looks similar:

(**run**$^\mathbf{1}$ $(\hat{B})$ (**eigen** $(x\ y\ z)$ ($\rhd_w^o$ ‘(((,$\hat{B}$,x) ,y) ,z) ‘(,y (,x ,z)))))

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.

4.2 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:

(**run**$^\mathbf{1}$ (F) (**eigen** (X) ($\rhd_w^o$ ‘(,F ,X) ‘(,X (,F ,X))))) $\Rightarrow$ (((S I) (((S (S (K (S I)))) I) ((S (S (K (S I)))) I))))

This query takes roughly eight minutes, using non-pattern-matching versions of $\rhd$, $\rhd_{1w}$, and $\rhd_w$ (which perform fewer unifications than the versions using **defmatch**^{e}, which is not as efficient as it could be).

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)
      (▷ᵒw ‘(,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.

Combining our original fixpoint query with our L_η^o relation, we can 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)
                            (▷ᵒw ‘(,F ,x) ‘(,x (,F ,x)))
                            (Lᵒη F F_V)))))
                 (environment ’(rnrs)))))
  ((F_V (λ (f)
          (λ (n)
            (if (= n 0)
                1
                (∗ n (f (− n 1)))))))
   5)) ⇒ 120
```

[9] The name $\rhd^o$ is entered as `contracto`.

[10] The name $\rhd_{1w}^o$ is entered as `->1wo`.

[11] The name $\rhd_w^o$ is entered as `->wo`.

[12] The names L^o and L_η^o are entered as `Lo` and `L-etao`.

[13] Exercise 2.17b on p. 26.

[14] Running under 64-bit build of Vicare Scheme 0.1d2+ (using (*optimize-level* 2)), on a MacBook Pro with a 2.8 GHz Intel Core i7 and 16 GB RAM, running OS X 10.9.

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

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

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

Figure 1. Contraction

$$\frac{M \rhd M'}{M \rhd_{1w} M'} \qquad (\rhd_{1w}\text{-CONTRACTION})$$

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

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

Figure 2. One-step Reduction

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

$$\frac{M \rhd_{1w} N \quad N \rhd_w P}{M \rhd_w P} \qquad (\rhd_w\text{-TRANSITIVE})$$

Figure 3. Weak Reduction

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

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

$$\mathsf{S}\ L\ \lambda xyz.xz(yz) \qquad (L\text{-}\mathsf{S})$$

$$\frac{M\ L\ M' \quad N\ L\ N'}{MN\ L\ M'N'} \qquad (L\text{-COMPOUND})$$

Figure 4. Conversion to Call-by-Name λ-Calculus

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

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

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

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

Figure 5. Conversion 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 6. 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 7. 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 8. Weak Reduction (miniKanren)

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

Figure 9. Conversion to Call-by-Name λ-Calculus (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 10. Conversion to Call-by-Value λ-Calculus (miniKanren)

5. Performance

Our approach to combinator synthesis is limited by the fact that the search tree grows exponentially in the size of the combinators being synthesized. Even worse, a query whose specification does not match any combinator will diverge. There is no way around these limitations; they reflect the structure of combinatory logic, and the undecidability of universal computation.

Given these limitations, it may seem that combinator synthesis is a hopeless problem. However, there are several mitigating factors that can help us generate combinators faster than might otherwise seem reasonable.

First, queries sharing structure between the "input" term and the "output" term can *fail fast*, resulting in aggressive pruning of the search tree. This is evident in the fixpoint combinator query,

$(\mathbf{run^1}\ (F)\ (\mathbf{eigen}\ (X)\ (\triangleright_w^o\ `(,F\ ,X)\ `(,X\ (,F\ ,X))))),$

Here, the sub-term $`(,F\ ,X)$ appears on both sides of the call to the $\triangleright_w^o$ relation. This structure allows the query to return much faster, and with much less memory usage, than if miniKanren had performed naïve "guess and check"-style search. This sharing of structure is also evident in the quine-generating query from Byrd et al. (2012):

$(\mathbf{run^1}\ (q)\ (\textit{eval-exp}^o\ q\ '()\ q))$

This query generates a quine in approximately one second, which is orders of magnitude faster than for naïve "guess and check"-style search. The self-application hint $F = UU$ for generating a fixpoint combinator (section 4.2) is another example of exploiting the structure of the query.

Secondly, the pure logic programming approach we use is implicitly parallel. In theory, at least, we could try generating a combinator using many different combinator bases in parallel; larger bases have a larger branching factor for the search space, but may result in a smaller term (and therefore a shallower search).

6. Conclusion

The definitions of the $\triangleright^o$, $\triangleright_{1w}^o$, and $\triangleright_w^o$ relations are remarkable for their brevity, and for their one-to-one correspondence with the inference rules of combinatory logic. The succinctness of these definitions belies the complexity of the underlying miniKanren implementation required to perform program synthesis—for example, to perform miniKanren's complete, interleaving search. In Vicare Scheme[15], the 4-line definition of $\triangleright^o$ expands to 143 lines of Scheme code, the 4-line definition of $\triangleright_{1w}^o$ expands to 158 lines, the 3-line definition of $\triangleright_w^o$ expands to 132 lines, and the one-line query for synthesizing fixpoint combinators expands into 48 lines. These 12 lines of miniKanren code expand into 481 lines of Scheme code, not counting the functions that implement miniKanren's constraint-solving, stream-handling, and reification algorithms. Each $\mathbf{defmatch}^e$/query line is equivalent to approximately 40 lines of Scheme code, illustrating the point that miniKanren, extended with **eigen** and $\mathbf{defmatch}^e$, is essentially a domain-specific language for implementing combinatory logic in a relational manner.

This observation should be understood in the context of two other points. First, the relational interpreters presented in Byrd et al. (2012) are far more complicated than the $\triangleright^o$, $\triangleright_{1w}^o$, and $\triangleright_w^o$ relations. These relational interpreters require careful handling of environments and term representation (and the subtle interactions between the two). Implementing small-step versions of the interpreters—required for synthesizing fixpoint combinators, for example—is more complicated still. A substitution-based small-step interpreter requires nominal logic, higher-order abstract syntax, De Bruijn indices, or an equivalent technique in order to implement capture-avoiding substitution. These approaches either complicate the implementation of miniKanren (as in αKanren, for nominal logic programming (Byrd and Friedman 2007)), the implementation of the interpreter, or both, and generally complicate interpretation of answers, especially those containing fresh logic variables. Alternatively, the small-step interpreter could be written in the style of a CEK machine (Felleisen 1987), using an environment and an explicit continuation. This also complicates the interpreter; furthermore, the explicit continuation can break the strong connection between expression argument and value argument that allows the search tree to be pruned aggressively when performing program synthesis.

The second point is that the code in this paper can be easily translated to any logic programming language that supports both universal quantification and complete search. Prolog uses an incomplete depth-first strategy that can easily diverge, even when there are answers to be found. However, many Prolog systems implement some form of complete search in addition to depth-first search. Implementation of universal quantification seems less consistent between Prolog implementations; λProlog (Miller and Nadathur 2012) is one system that implements eigenvariables as described in section 2.2. Any sufficiently featureful Prolog system can therefore be consider a domain-specific language for implementing combinatory logic relationally.

Even logic systems that do not implement complete search can be used for combinator synthesis, by implementing depth-limited search (or incrementally-deepening depth-first search) on top of standard depth-first search. Nada Amin has recently used this technique to perform simple combinator synthesis in the Twelf (Pfenning and Schürmann 1999) theorem prover, using the $\triangleright^o$, $\triangleright_{1w}^o$, and $\triangleright_w^o$ relations from this paper, and using Twelf's support for higher-order abstract syntax to express universal quantification. Alas, Twelf's search seems too slow for fixpoint combinator synthesis to be practical.

Several other researchers have used related approaches to program synthesis. Kiselyov (2013) has written a Haskell program to enumerate fixpoint combinators in the λ-calculus; unlike in our approach, his code is designed specifically for generating combinators. This means that the generator is more efficient, but loses the direct connection between the inference rules and implementation that we feel is important.

Closer in spirit to our work, a group of European researchers has recently published a number of papers on combinator synthesis, with applications to program synthesis for software engineering (Cégielski and Durand 2012; Düdder et al. 2012, 2013a,b; Graham-Lengrand and Paolini 2013; Rehof and Urzyczyn 2011, 2012). Their work is similar to ours, in that they implement a logic program (in this case, in Prolog) that synthesizes combinators in combinatory logic from high-level specifications. Their approach differs from ours in that they use *type habitation* of intersection types to perform their combinator synthesis. In other words, their

[15] Vicare Scheme version 0.1d2+, 64-bit (revision 08bd828acfa9382324150b41f4e86c540c10a886, build 2013-08-27

queries take the form of type specifications, and their synthesis tool performs type inference to find a combinatory logic term with the matching type. It may be possible to combine the type habitation approach to synthesis with our combinator reduction-based approach to improve performance of synthesis.

In conclusion, our work greatly simplifies the approach to relational program synthesis demonstrated in Byrd et al. (2012). Our approach is also much more flexible. For example, the quine-generating query

$$(\mathbf{run^1}\ (q)\ (\textit{eval-exp}^o\ q\ \text{'}()\ q))$$

from Byrd et al. (2012) is completely symmetrical—indeed, it is the simplest meaningful query using *eval-exp*^{o}, in terms of the number of distinct tokens. Unfortunately, since *eval-exp*^{o} is a big-step interpreter, it is incapable of synthesizing many programs that require more complex queries, such as fixpoint combinators. In contrast, our approach handles richer queries, such as

$$(\mathbf{run^1}\ (F)\ (\mathbf{eigen}\ (X)\ (\triangleright^o_w\ \text{`}(,F\ ,X)\ \text{`}(,X\ (,F\ ,X))))),$$

which directly mirrors the fixpoint combinator definition $\exists F.\forall X.FX = X(FX)$.

Our work, along with the work presented in Byrd et al. (2012), shows that it is possible to synthesize interesting programs using reducers and interpreters written in a relational style. Much further work remains to be done, including developing techniques for writing small-step interpreters for the λ-calculus and Scheme, and improving the performance of program synthesis through optimizations and parallelization.

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 and Daniel P. Friedman. From variadic functions to variadic relations: A miniKanren perspective. In Robby Findler, editor, *Proceedings of the 2006 Scheme and Functional Programming Workshop*, University of Chicago Technical Report TR-2006-06, pages 105–117, 2006.

William E. Byrd and Daniel P. Friedman. αKanren: A fresh name in nominal logic programming. In *Proceedings of the 2007 Workshop on Scheme and Functional Programming, Universite Laval Technical Report DIUL-RT-0701*, pages 79–90 (*see also* `http://www.cs.indiana.edu/~webyrd` *for improvements*), 2007.

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.

Patrick Cégielski and Arnaud Durand, editors. *Computer Science Logic (CSL'12) - 26th International Workshop/21st Annual Conference of the EACSL, CSL 2012, September 3-6, 2012, Fontainebleau, France*, volume 16 of *LIPIcs*, 2012. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik. ISBN 978-3-939897-42-2.

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

H. B. Curry, J. R. Hindley, and J. P. Seldin. *Combinatory Logic, Volume II.* North-Holland, 1972.

Boris Düdder, Moritz Martens, Jakob Rehof, and Pawel Urzyczyn. Bounded combinatory logic. In Cégielski and Durand (2012), pages 243–258. ISBN 978-3-939897-42-2.

Boris Düdder, Oliver Garbe, Moritz Martens, Jakob Rehof, and Pawel Urzyczyn. Using inhabitation in bounded combinatory logic with intersection types for composition synthesis. In Graham-Lengrand and Paolini (2013), pages 18–34.

Boris Düdder, Moritz Martens, and Jakob Rehof. Intersection type matching with subtyping. In Masahito Hasegawa, editor, *TLCA*, volume 7941 of *Lecture Notes in Computer Science*, pages 125–139. Springer, 2013b. ISBN 978-3-642-38945-0, 978-3-642-38946-7.

Matthias Felleisen. *The calculi of λ_v-CS Conversion: A syntactic theory of control and state in imperative higher-order programming languages.* PhD thesis, Indianapolis, IN, USA, 1987.

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

Stéphane Graham-Lengrand and Luca Paolini, editors. *Proceedings Sixth Workshop on Intersection Types and Related Systems, ITRS 2013, Dubrovnik, Croatia, 29th June 2012*, volume 121 of *EPTCS*, 2013.

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

Douglas R. Hofstadter. *Gödel, Escher, Bach : an eternal golden braid.* Basic, 1979.

Andrew W. Keep, Michael D. Adams, Lindsey Kuper, William E. Byrd, and Daniel P. Friedman. A pattern-matcher for miniKanren -or- How to get into trouble with CPS macros. In *In Proceedings of the 2009 Workshop on Scheme and Functional Programming, Cal Poly Technical Report CPSLO-CSC-09-03*, pages 37–45, 2009.

Oleg Kiselyov. Many faces of the fixed-point combinator: Fix-point combinators are infinitely many and recursively-enumerable. `http://okmij.org/ftp/Computation/fixed-point-combinators.html#many-fixes`, 2013.

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

Dale Miller and Gopalan Nadathur. *Programming with Higher-Order Logic.* Cambridge University Press, 2012.

Frank Pfenning and Carsten Schürmann. System description: Twelf a meta-logical framework for deductive systems. In *Automated Deduction CADE-16*, volume 1632 of *Lecture Notes in Computer Science*, pages 202–206. Springer Berlin Heidelberg, 1999.

Jakob Rehof and Pawel Urzyczyn. Finite combinatory logic with intersection types. In C.-H. Luke Ong, editor, *TLCA*, volume 6690 of *Lecture Notes in Computer Science*, pages 169–183. Springer, 2011. ISBN 978-3-642-21690-9.

Jakob Rehof and Pawel Urzyczyn. The complexity of inhabitation with explicit intersection. In Robert L. Constable and Alexandra Silva, editors, *Logic and Program Semantics*, volume 7230 of *Lecture Notes in Computer Science*, pages 256–270. Springer, 2012. ISBN 978-3-642-29484-6.

G. Revesz. *Lambda-calculus, Combinators, and Functional Programming.* Cambridge University Press, Cambridge, UK, 1988.

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.

Sören Stenlund. *Combinators, λ-Terms and Proof Theory*. D. Reidel, 1972.

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