

A Logic System with First-Class Relations

(Draft: November 4, 2003)

Daniel P. Friedman

*Computer Science Department, Indiana University
Bloomington, IN 47405, USA*

Oleg Kiselyov

*Fleet Numerical Meteorology and Oceanography Center,
Monterey, CA 93943*

1 Introduction

We present an implementation of an embedding in R⁵ Scheme of a logic system with first-class relations that we believe interacts smoothly with its host. How can we have logic programming and first-class relations and lose nothing of Scheme? We make the control structure of logic programming explicit, so that Scheme programs work with it. This requires just a small handful of operations that comprises its interface.

In this introduction we skim over the topics that we cover. We ask for the reader's patience when some term or idea is mentioned without a full explanation. Keep in mind that we are not only explaining how to write logic programs, but also how to embed them in Scheme. All aspects of logic programming mentioned in this introduction are described later in the paper. We do assume, however, a reading knowledge of Scheme and some familiarity with macros.

Logic systems such as Prolog have been around for quite a while and much is understood about them. By the way that we have implemented our logic system, we have been able to include first-class relations. Furthermore, we make them extensible, which means that we give ourselves the power of sharing subrelations.

In order to make this approach work, four properties that are not normally associated with the discussion of logic programming had to become apparent. The first property is that most of the time we needn't think about the relations as a monolithic global relation. In fact, in most circumstances, the programmer knows exactly which relation to home in on and we are going to require that knowledge of the programmer.¹ For example, the relation that infers a type for a programming language differs significantly from the relation that determines how to concatenate two lists, and both of these differ considerably from the relation that determines whether two people are related by the fact that one of them is the father of someone and the other is the child of that same someone. Being able to separate these three relations makes possible a smooth integration with Scheme.

¹ This restriction also appears in Silvija Seres and Michael Spivey, "Embedding Prolog in Haskell". In *Haskell Workshop*, Paris, France, September, 1999.

The second property is that there should be no distinction between a Scheme function and a logic relation. This has allowed for the removal of a dispatch, leading to an implementation with *substitution composition* being the only place where recursion appears. This is not quite accurate for three reasons. First, if only one answer is produced, then no recursion is needed, but some mechanism must be in place to handle no answer or ask for more answers (We use streams.). Second, in order to display answers that contain the same logic variable (henceforth called variable) more than once, we need a recursive copier that keeps track of the variables. But, this is for passing results out to arbitrary Scheme functions. Finally, in the last section we introduce pairs (lists) as valid terms, requiring the revision of three functions, each of which becomes recursive.

The third property is that gratuitous lists should never be built. This matters because we treat relations as finite-arity functions and thus we can consider the actual arguments separately instead of joined together as a list. The implications of this decision are far reaching, since it allows us to avoid many applications of substitutions. Where there would be two lists to unify, we now have two sequences of arguments, which means that we can avoid substituting in each `cdr` in each list. (See the redefinition of `unify*` in the last section.) Also, if our raw data does not contain lists, then our unifier does not need to know about them.

The fourth property is that since relations are first-class, they should support a relation extension operation. This operation takes an arbitrary number of relations, all with the same signature. We start the development in a more conventional way treating a relation as a single rule. Then we present relation extension. With it, we get the ability to share relations. We use relation extension throughout, but we show some of the real power of relation extension later when we interactively build a type inference system with it.

This tutorial has been written so that most definitions come before their usage. This has the disadvantage that occasionally the code for the implementation appears before it is explained. It has the advantage, however, that most lines of code can be accounted for while sitting at a terminal. This does not necessarily make learning the material easier, but it does allow the reader to know what parts belong in what order. Skimming the tutorial, however, is mostly discouraged, since some functions and relations are defined multiple times and their order of definition matters!

There are six additional sections. First, we include preliminaries, much of which should be familiar to most readers. We introduce some useful syntactic abstractions as well as some standard functions for dealing with variables, substitutions, and unification. In section 3 we implement and demonstrate the unsullied logic system. In section 4 we explore sullied operators, such as the cut operator, and their implementation. Next, we utilize some recursive relations and include a comparison with an embedding in Haskell. We follow that by a section discussing three famous logic programming problems: `append`, type inference, and a generalization of Prolog's name. We conclude with some perspective on why this approach works.

2 Preliminaries

There are three categories of operations we need. The first category contains operators for creating lexically-scoped variables (`exists`), currying (`lambda@` and `@`), and a lexical binder for multiple values (`let-values`). The second category concerns itself with building (`empty-subst`, `compose-subst`) and applying (`subst-in`) substitutions. The final category has operations for unifying two terms: `unify` and `unify*`.

2.1 Creating variables, Currying, and a macro for multiple values

A data structure that may be a variable or contain variables is called a *term*. We create a variable using the procedure `var`, which takes a symbol as its argument. Here is an example of its use.

```
> (let ([x (var 'x)] [xx (var 'x)])
      (list (var-id x) (var-id xx) (var? x) (eqv? x xx)))

(x x #t #f)
```

There is a procedure `var-id` that retrieves the name that a variable has been built from, there is a predicate `var?` that distinguishes variables from other data structures, and each invocation of `var` makes a new one. (We leave the details of defining these three procedures as an exercise. Certainly, a uniquely tagged cons pair with its name in the `cdr` could be used, but most Scheme systems have their own record facility. For purists, instead of relying on the ability to distinguish two cons cells with `eqv?`, one could associate a unique timestamp with each variable.)

We introduce a macro, `exists`, which allows us to both create variables and have them known within a lexical scope. Since we are generating a `let` expression, it is necessary to include different names in the first argument to `exists`.

```
(define-syntax exists
  (syntax-rules ()
    [(_ () body) body]
    [(_ () body0 body1 ...) (begin body0 body1 ...)]
    [(_ (id ...) body0 body1 ...)
     (let ([id (var 'id)] ...) body0 body1 ...)]))
```

For example,

```
(exists (x y z)
  (list x 1 y 2 z 3))
```

expands to (is the same as writing)

```
(let ([x (var 'x)] [y (var 'y)] [z (var 'z)])
  (list x 1 y 2 z 3))
```

Here is how we can verify these ideas using `expand` and `expand-only`.

```

> (expand
  '(exists (x y z)
    (list x 1 y 2 z 3)))

((lambda (x y z) (list x 1 y 2 z 3))
 (var 'x)
 (var 'y)
 (var 'z))

> (expand-only '(exists)
  '(exists (x y z)
    (list x 1 y 2 z 3)))

(let ([x (var 'x)] [y (var 'y)] [z (var 'z)])
  (list x 1 y 2 z 3))

```

There are other uses of `expand-only` below. The main thing to observe, however, is that we do not have to expand `let` expressions, if we don't want to. The deeper the nesting levels of macro calls, the fewer items we might want to place in the first argument to `expand-only`. This one only has two levels: `exists` expands to `let`, and `let` expands to an application of a `lambda`-expression.

One other item that we should get familiar with is that using `list` works, but misses the structural relationship that we get back using `quasiquote` and `unquote`. The last example can also be written like this,

```

> (expand-only '(exists)
  '(exists (x y z)
    '(,x 1 ,y 2 ,z 3)))

(let ([x (var 'x)] [y (var 'y)] [z (var 'z)])
  '(,x 1 ,y 2 ,z 3))

```

This way we get Scheme's `quasiquote` “`“`” and `unquote` “`,`” to indicate where the variables can be found in a list structure. But, as we discover there is still much we can do with logic programming that does not involve lists. In fact, they don't appear until the last section.

2.1.1 Currying macros

We also define a curried `lambda`, `lambda@`,

```

(define-syntax lambda@
  (syntax-rules ()
    [(_ (formal) body0 body1 ...) (lambda (formal) body0 body1 ...)]
    [(_ (formal0 formal1 formal2 ...) body0 body1 ...)
     (lambda (formal0)
       (lambda@ (formal1 formal2 ...) body0 body1 ...))]))

```

To go along with `lambda@`, we have `@`, a curried application macro.

```
(define-syntax @
  (syntax-rules ()
    [(_ rator rand) (rator rand)]
    [(_ rator rand0 rand1 rand2 ...) (@ (rator rand0) rand1 rand2 ...)]))
> (@ (lambda@ (x y z) (+ x (+ y z))) 1 2 3)
```

6

```
> (expand '(@ (lambda@ (x y z) (+ x (+ y z))) 1 2 3))
```

```
((((lambda (x)
  (lambda (y)
    (lambda (z)
      (+ x (+ y z))))))
  1) 2) 3)
```

Curried application `@` and curried `lambda@` give us the ability to build and invoke our functions so that the arguments arrive one at a time.

2.1.2 A macro for binding multiple return values

There are times when it is better to be explicit about returning multiples values. Here is a simple macro that should encourage such behavior.

```
(define-syntax let-values
  (syntax-rules ()
    [(_ (x ...) vs body0 body1 ...)
     (call-with-values
      (lambda () vs)
      (lambda (x ...) body0 body1 ...))]))
```

```
> (let-values (x y z) (values 1 2 3))
(+ x (* y z))
```

7

Since `values` is a global function it can be called from anywhere,

```
> (= (let ([f (lambda () values)])
      (let-values (x y z) ((f) 1 2 3)
        (+ x (* y z))))
     (let ([f (lambda (w) (values w (+ w 1) (+ w 2)))]
           (let-values (x y z) (f 1)
             (+ x (* y z)))))
```

#t

2.2 Substitutions

A substitution, `s`, is a list of real commitments, represented using an association list. A *commitment* is a pairing (`commitment`) of a variable (`commitment->var`) to a term (`commitment->term`) and it is *real* if its variable is not the same as its term. We say a variable is *committed* if it has an association in a substitution. We will have occasion to naively add a commitment to a substitution, so we must be certain that the commitment is real.

```
(define cons-if-real-commitment
  (lambda (var term subst)
    (cond
      [(eqv? term var) subst]
      [else (cons (commitment var term) subst)])))
```

The committed variables (i.e., `(map commitment->var s)`) of a substitution must form a set. We introduce two kinds of substitutions. The first is the empty substitution: `empty-subst`, which we represent with the empty list. The second, built with `compose-subst`, is formed by refining one substitution with another to form a new substitution.²

```
(define empty-subst '())

(define compose-subst
  (lambda (base refining)
    (let refine ([base base] [survivors refining])
      (cond
        [(null? base) survivors]
        [else (cons-if-real-commitment
                  (commitment->var (car base))
                  (subst-in (commitment->term (car base)) refining)
                  (refine (cdr base)
                          (cond
                            [(assv (commitment->var (car base)) survivors]
                             => (lambda (c) (remv c survivors))]
                            [else survivors])))))])))
```

When does `compose-subst` behave like `append`? There are two aspects of the code to consider. First, the `assv` test must always fail. For if it doesn't, `survivors`, which starts out as `refining` shrinks. Second, the `subst-in` expression must not yield a new term the same as `(commitment->var (car base))`. One way to do this would be to make sure that the variables of `(commitment->term (car base))` do not overlap with `(map commitment->var refining)`.

But, of course, `compose-subst` rarely behaves exactly like `append`. A commitment in the original `refining` substitution may not show up in the resultant substitution.

² This material on substitutions is derived from definitions and examples on pages 18 and 19 of J. W. Lloyd's *Foundations of Logic Programming*.

This happens when a variable is committed in `base` also is committed in `refining`. This should be obvious, since no variable can be committed to more than one term in the resultant substitution. The one committed in `base` is the one that matters. So composing two substitutions is like taking the union of the commitments relative to the variables and utilizing the content of the refining substitution to affect the term in `base`'s substitution. Any resultant unreal commitments are excluded from the resultant substitution.

The procedure `subst-in` below translates everything to itself except variables. If the variable is committed, its associated term is returned, otherwise the variable itself is returned, since it is virtually associated with itself. Initially, we restrict terms to be variables or values that can be trivially compared. In the last section we *enlarge the set of terms to include pairs (lists), which may contain variables*.

```
(define subst-in
  (lambda (t subst)
    (cond
      [(var? t)
       (cond
         [(assv t subst) => commitment->term]
         [else t])]
      [else t])))
```

Exercise 1: Rewrite `compose-subst` and `subst-in` to check in advance to see if any of its substitution arguments are empty. \diamond

In the definition of `unify*` below we create a substitution of exactly one real commitment. We make this explicit with the definition of `unit-subst` below. The outer `list` in the definition is there because every substitution is a *list* of commitments.

```
(define unit-subst
  (lambda (var t)
    (list (commitment var t))))
```

Consider the use of substitutions in everyday experiences. Before you buy your first motorized vehicle, you have made no commitments to yourself about its purchase. You have the empty substitution. Then you decide to buy a four-wheeled motorized vehicle. Now you have enlarged your empty substitution to include a single commitment that whatever you buy should have four wheels and an engine. Then, you decide to purchase a car. You have refined your earlier commitment to buy a truck, a car, or some other four-wheeled vehicle to buying a car. You still have only one commitment in your substitution, though you have composed two nonempty substitutions: the one that stated that you would buy a four-wheeled motorized vehicle and the one that stated that you would buy a car. You can refine your current substitution by stating that the car would not be over three years old. You still have only one commitment. At some point, you can *also* commit to purchasing a television. Now, you have two commitments. Each time you refine these two commitments you get a substitution with two commitments. For example, you might choose to get a sedan and a color television. You still don't know what you

are going to get, but as your current substitution is refined, you are getting closer and closer to making your decision. Let's consider some examples:

```
> (exists (x y)
  (equal?
    (compose-subst (unit-subst x y) (unit-subst y 52))
    '(,(commitment x 52) ,(commitment y 52))))
#t

> (exists (w x y)
  (equal?
    (let ([s (compose-subst (unit-subst y w) (unit-subst w 52))])
      (compose-subst (unit-subst x y) s))
    '(,(commitment x 52) ,(commitment y 52) ,(commitment w 52))))
#t

> (exists (w x y)
  (equal?
    (let ([s (compose-subst (unit-subst w 52) (unit-subst y w))])
      (compose-subst (unit-subst x y) s))
    '(,(commitment x w) ,(commitment w 52) ,(commitment y w))))
#t

> (exists (w x y)
  (equal?
    (let ([s (compose-subst (unit-subst y w) (unit-subst x y))]
          [r (compose-subst
              (compose-subst (unit-subst x 'a) (unit-subst y 'b))
              (unit-subst w y))])
      (compose-subst s r))
    '(,(commitment x 'b) ,(commitment w y))))
#t
```

In the first example, the base substitution commits x to y . Then, the refining substitution commits y to 52, refining x to 52. In the second example, the base substitution of s commits y to w , refining y to 52. Then s is used as the refining substitution, so the resultant substitution commits x to 52. In the third example, the refining substitution of s does not influence its base. Thus s contains the two commitments. Then the base substitution of the resultant substitution is refined by the y -commitment committing x to w . In the fourth example, we construct two substitutions manually. The first contains two commitments and the second contains three commitments. The second refines the first to yield a substitution with only two commitments, since we attempt to create a commitment of y to y . **Exercise 2:** Hand trace the fourth example, above, and thus prove that composing a substitution with two commitments and a substitution with three commitments can yield a substitution with just two commitments. \diamond

2.3 Unification

We introduce the unification interface: `unify` and `unify*`. The interface `unify` below tries to find a substitution that will treat the two substituted for terms as equal if the resultant substitution were applied (using `subst-in`) to them. If successful, it returns a composed substitution. If it cannot unify the two terms, then `false` is returned. Since we have a limited definition of term at this time, we can easily write the auxiliary procedure `unify*` below. Two terms unify if they are the same variable, the same constant term, or one of them is the *anonymous* variable. Then, `unify*` returns the empty substitution. That is why we do not choose `false` to represent the empty substitution. Returning the empty substitution means that the two terms unified. Of course, that would mean that the two terms contained no variables or virtually contained no variables. Otherwise, if either term is a variable, it treats the other as a term and returns a substitution of a singleton commitment. In all other cases, the unifier returns `false`. (We again stress that these definitions work until we get to the last section, where we add pairs, which can contain variables. When we make this change, we also must change `subst-in` to support pairs.

```
(define _ (exists (_ _))

(define unify
  (lambda (t u subst)
    (cond
      [(unify* (subst-in t subst) (subst-in u subst))
       => (lambda (refining-subst)
            (compose-subst subst refining-subst))]
      [else #f])))

(define unify*
  (lambda (t u)
    (cond
      [(trivially-equal? t u) empty-subst]
      [(var? t) (unit-subst t u)]
      [(var? u) (unit-subst u t)]
      [else #f])))

(define trivially-equal?
  (lambda (t u)
    (or (eqv? t u)
        (eqv? t _)
        (eqv? u _)
        (and (string? t) (string? u) (string=? t u))))))
```

Here are a few examples.

```
> (exists (x y)
  (and
    (equal? (unify x 3 empty-subst) '(,(commitment x 3)))
    (equal? (unify 4 y empty-subst) '(,(commitment y 4)))
    (equal? (unify x y empty-subst) '(,(commitment x y)))
    (equal? (unify 'x 'x empty-subst) empty-subst)
    (equal? (unify x x empty-subst) empty-subst)
    (not (unify 4 'y empty-subst))
    (not (unify 'x 3 empty-subst))
    (not (unify 3 4 empty-subst))))
```

#t

Only the fifth example is interesting. It returns the empty substitution, since the two terms are the same variable. The sixth and seventh don't unify because a symbol (not a variable) can never be equal to a number.

3 The unsullied logic system

The logic system has just two primary operators: `relation` (fact is derived from `relation`), which expands into a lambda expression, and `extend-relation`, which gives us the ability to form a new relation from other relations. In addition, we have three macros for handling sequences of antecedents: `all`, its deterministic counterpart `all!`, and `all`'s dual `any` (derived from `relation` and `extend-relation`). To enter the system, we have a procedure query, and two interfacing operations: `solve` and `solution` (derived from `solve`).

Here are the types we use in this system:

```
Fk = () -> Ans
Ans = Nil + [[Subst, Cutk], Fk]
Cutk = Fk
Sk = Fk -> Subst -> Cutk -> Ans
Antecedent = Sk -> Sk
Relation = Term* -> Antecedent
```

`Ans` is a stream of `[Subst, Cutk]` pairs, since `Fk` is a function of zero arguments. An antecedent is an `Sk` transformer.

3.1 It's a small world

Let's define a single fact. (Warning: This is not the best time to try to figure out these definitions.)

```
(define succeed (lambda (sk) sk))

(define-syntax all
  (syntax-rules ()
    [(_) succeed]
    [(_ ant) ant]
    [(_ ant0 ant1 ...)
     (lambda (sk)
       (ant0 (splice-in-ants/all sk ant1 ...)))]))

(define-syntax splice-in-ants/all
  (syntax-rules ()
    [(_ sk ant) (ant sk)]
    [(_ sk ant0 ant1 ...)
     (ant0 (splice-in-ants/all sk ant1 ...))]))
```

The `all` expression has two things in common with Scheme's `and` expression. The second rule is superfluous in both. Also, the variant with no arguments is a constant. With `and`, it is true and with `all`, it is `succeed`. Later we discuss the difference between succeeding and being true.

Let's take a look at how it expands, without worrying about the actual arguments to `all`.

```
> (expand-only '(all splice-in-ants/all lambda@)
  '(all a b c d e))
(lambda (sk) (a (b (c (d (e sk))))))
```

It looks like the first thing that happens is that `e` is applied, but although that is the first thing that happens, it is `a` that actually does the first bit of work. How is that possible. Let's look at a simpler example.

```
> (expand-only '(all splice-in-ants/all)
  '(all a b))
(lambda (sk) (a (b sk)))
```

This is just function composition. We know that `(b sk)` must return an `sk`, since that is what `a` is expecting. But suppose that, `a` is also expecting another argument, say `fk`. So, we can see this as `(lambda@ (sk fk) ((a (b sk)) fk))`. Now, eventually, if `a` decides to invoke the `(b sk)`, which was passed in, it will do it on a different `fk`, but it should be clear that `a` gets to run first. These two lambda expressions are η -convertible, however, so we do not need to worry about the `fks`. Moreover, there are two more arguments, `subst`, and `cutk`, that must arrive before `a` runs. This is tricky, but what is important is that we must understand that the

order of the arguments in `all!` determines when the antecedents are run, which is what we would expect.

```
(define-syntax all!
  (syntax-rules ()
    [(_) succeed]
    [(_ ant) ant]
    [(_ ant0 ant1 ...)
     (lambda@ (sk fk)
       (@ ant0 (splice-in-ants/all! sk fk ant1 ...) fk))]])
```

```
(define-syntax splice-in-ants/all!
  (syntax-rules ()
    [(_ sk fk ant)
     (lambda (ign-fk)
       (@ ant sk fk))]
    [(_ sk fk ant0 ant1 ...)
     (lambda (ign-fk)
       (@ ant0 (splice-in-ants/all! sk fk ant1 ...) fk))]])
```

And again, we get a feel for what it does by looking at its expansion.

```
> (expand-only '(all! splice-in-ants/all!) '(all! a b c))
(lambda@ (sk fk)
  (@ a
    (lambda (ign-fk)
      (@ b
        (lambda (ign-fk)
          (@ c sk fk))
        fk))
    fk))
```

The macro `all!` generates a procedure that expects a success and a failure continuation (and two more, which are η -reduced away). Each antecedent is run in a new success continuation, but each is run with the same failure continuation. Thus, this is a deterministic `all`. One slip up, and you fail, whereas, with `all`, you get to backup through the antecedents and retry them until success, but we are getting a bit ahead of our story.

Finally, we come to `relation` below, which is like a special `lambda`. There are five rules to consider. The first and the last conspire to get to the middle three rules. They do that by using the exact match of the `to-show` keyword, which then sets up a loop accumulating fresh variables the length of the arguments to the `to-show` keyword. Once the accumulation has been accomplished in the `(g ...)`, we have three remaining possibilities. In each case we create the `lambda (g ...)`. In the first case (second rule), since there are no antecedents, we simply have the `all!` of a bunch of unifications. In the next two cases (third and fourth rules) we deal with whether or not we intend to use the *cut*. In the first of these rules, where we use _

to indicate no interest in the *cut*, we include an *all* expression. It runs in the state returned from the *all!*, but specifically, it runs in the substitution generated by the sequence of unifications. In the second of these rules, we bind (*!! cutk*) below to *cut-id* before entering the *all* expression. The antecedent bound to *cut-id* always succeeds. When failure backs into it, however, it invokes the *cutk* which had been passed to *!!*. The reason we need the (*lambda@ (sk fk subst cutk) ...*) is to give us access to *cutk* in order to build the *cut* antecedent. It is only the free variable *cutk* that forbids us from doing an η -reduction in this case.

Of course, the special cases of no antecedents and using *_* are not really part of the definition, so it is sort of okay to think of this macro as composed only of the first, fourth, and last rule. But, because the underscore is used as a variable that matches everything, we don't wish to bind it, which gives rise to the need for the exact match in the third rule. Thus, the mere choice of underscore as the indicator that no cut will occur forces us to include the third rule. On the positive side, however, when the declaration is made, the code generated is easier to follow. Thus, provided that underscore is *not* used, then the first, fourth, and last rules of relation form a well-defined macro.

```
(define-syntax relation
  (syntax-rules (to-show _)
    [(_ cut-id (ex-id ...) (to-show x ...) ant ...)
      (relation cut-id (ex-id ...) () (x ...) (x ...) ant ...)]
    [(_ cut-id (ex-id ...) (g ...) () (x ...))
      (lambda (g ...)
        (exists (ex-id ...)
          (all! (== g x) ...)))]
    [(_ _ (ex-id ...) (g ...) () (x ...) ant ...)
      (lambda (g ...)
        (exists (ex-id ...)
          (all! (== g x) ... (all ant ...)))]
    [(_ cut-id (ex-id ...) (g ...) () (x ...) ant ...)
      (lambda (g ...)
        (exists (ex-id ...)
          (all! (== g x) ...
            (lambda@ (sk fk subst cutk)
              (let ([cut-id (!! cutk)])
                (@ (all ant ...) sk fk subst cutk))))))]
    [(_ cut-id ex-ids (var ...) (x0 x1 ...) xs ant ...)
      (relation cut-id ex-ids (var ... g) (x1 ...) xs ant ...)])

(define !!
  (lambda (exiting-fk)
    (lambda@ (sk fk)
      (@ sk exiting-fk))))
```

```
(define ==
  (lambda (t u)
    (lambda@ (sk fk subst cutk)
      (cond
        [(unify t u subst)
         => (lambda (subst)
              (@ sk fk subst cutk))]
        [else (fk)]))))))
```

We now have enough tools to define a relation with a single fact. Our relation says that “Pete is the father of Sal.”

```
(define father
  (relation _ ()
    (to-show 'pete 'sal)))
```

Let’s focus on the single fact that says that “Pete is the father of Sal.” But, what it really says is that there is a relation, that joins “Pete to Sal,” which we are calling “The father relation.” And when do we know that Pete is the father of Sal? When we have shown that all the expressions below the `to-show` keyword expression hold. But, since there are none, it holds vacuously. Therefore, Pete is always the father of Sal. But, if we have another relation, *older than*, we can have the fact that “Pete is older than Sal.” The only thing that would change is that we would replace the variable `father` by `older-than`. This is a lot deeper than it may first appear. Our relations are first class. If we wanted to do the same thing, we could also just `(define older-than father)`, which would say that “Pete is older than Sal.” Of course, that should not come as a surprise, since Pete is Sal’s father, but these two relations are completely disjoint! For example, we may be referring to four different people: two Petes and two Sals.

Sometimes the list of variable identifiers that follows `relation` is nonempty, but don’t be confused: *This list does not tell how many arguments the relation takes. That is determined by the number of terms following the keyword to-show.*

Consider the partial expansion of `father`

```
> (expand-only '(relation)
  '(define father
    (relation _ ()
      (to-show 'pete 'sal))))

(define father
  (lambda (g1 g2)
    (exists ()
      (all! (== g1 'pete) (== g2 'sal)))))
```

The relation `father` is represented as a procedure of two arguments, `g1` and `g2`.³ It can figure that out from the number of operands in the `to-show` keyword

³ The actual expansion produces `g` and `g`. But, they are different variables.

expression. Also, the “(exists ())” is useless. If we can unify the value of the variable `g1` with the value of `'pete` and the value of the variable `g2` with the value of `'sal`, then the resultant substitution is passed to a procedure that addresses any remaining antecedents. In our simple example, there are none.

Whenever `father` is invoked, a `dad (g1)` and a `child (g2)` get bound to a term. Then an antecedent is returned. This antecedent is waiting for a success continuation, a failure continuation, a substitution, and another failure continuation. The failure continuation is invoked if the arguments fail to unify. This effects a return from the call to `father`.

Next, we define a relation `child-of-male` and compare its expansion to the previous relation.

```
(define child-of-male
  (relation _ (child dad)
    (to-show child dad)
    (father dad child)))

> (expand-only '(relation)
  '(define child-of-male
    (relation _ (child dad)
      (to-show child dad)
      (father dad child))))
```

```
(define child-of-male
  (lambda (g1 g2)
    (exists (child dad)
      (all!
        (== g1 child)
        (== g2 dad)
        (all (father dad child)))))))
```

We can see that to show that some person (`child`) is the child of a male (`dad`), all we have to show is that `dad` is the father of `child`. This is very naive but logical reasoning.⁴

Exercise 3: How does this expanded definition differ if we also expand this code with respect to `all`? ◇

3.2 Testing father and child-of-male

We are now ready to test `father`. For example, we might wish to determine “If Pete is the father of Sal.”

⁴ Use the partially-expanded variant of `child-of-male` instead of the unexpanded `relation` to install a `let` expression around the `exists` expression. For example, one might want to read from a file, print what was read, and then unify against it instead of against `child`.

```

(define initial-sk
  (lambda@ (fk subst cutk)
    (cons (cons subst cutk) fk)))
(define initial-fk (lambda () '()))
(define initial-cutk initial-fk)

> (let ([ant (father 'pete 'sal)])
  (@ ant initial-sk initial-fk empty-subst initial-cutk))
(() . #<initial-fk>)

```

If the goal succeeds by invoking `initial-sk` on a substitution and two failure continuations, then once it gets the second failure continuation, it returns a pair of (a pair of the substitution and the `initial-cutk`) and a failure continuation. If the goal fails, the empty list, the result of invoking `initial-fk`, is returned.

Since our test has no variables, we know that we do not refine the original substitution, but unify when the raw values in each term are the same. What is the purpose of the empty substitution, `()`? At this point, we can (using `subst-in`) apply the empty substitution to each term in the original term and produce what we started with: `(pete sal)`, but with the assurance that `pete` has been shown to be the father of `sal`.

We abstract the previous test using the function, query below.

```

(define query
  (lambda (ant)
    (@ ant initial-sk initial-fk empty-subst initial-cutk)))

```

We next consider the role of nonempty substitutions. Instead of asking a specific question about Pete's relationship to Sal, let's determine a child of Pete. To do this, we introduce a variable.

```

> (exists (x)
  (let ([result (query (father 'pete x))])
    (and
      (equal? (caar result) '(,(commitment x 'sal)))
      (null? ((cdr result))))))
#t

> (exists (x)
  (let ([result (query (father 'pete x))])
    (let ([subst (caar result)])
      (list (subst-in 'pete subst) (subst-in x subst)))))
(pete sal)

```

We see that the resultant substitution binds `x` to `sal`. When we use this substitution, `pete` becomes `pete`, but `x` becomes `sal`.

We can test `child-of-male` the same way, but this time we choose to give it no raw data, just variables, and hope that a nonempty substitution, the `car` of the query, is returned.

```
> (exists (x y)
    (let ([result (query (child-of-male x y))])
        (let ([subst (caar result)])
            (list (subst-in x subst) (subst-in y subst))))))

(sal pete)
```

And, we discover that Sal is the child of Pete.

3.3 Generating more than one answer

The difference between a relation and a function is that with a function, only one answer is associated with an input, but with a relation the same input can lead to many answers. Now, we demonstrate in what ways functions such as the definition of `father` below can be treated as relations.

Suppose that Pete is also the father of Pat, what changes could be made to get both answers? First, we introduce `binary-extend-relation` to allow for an additional fact.

```
(define-syntax binary-extend-relation
  (syntax-rules ()
    [(_ (id ...) rel-exp1 rel-exp2)
     (let ([rel1 rel-exp1] [rel2 rel-exp2])
         (lambda (id ...)
           (lambda@ (sk fk subst cutk)
             (@ (rel1 id ...)
                sk (lambda () (@ (rel2 id ...) sk fk subst cutk)) subst cutk)))))]))

(define pete-pat
  (relation _ ()
    (to-show 'pete 'pat)))
```

Now, we have two relations. What we want to do is combine the two relations into a single one.

```
(define father (binary-extend-relation (a1 a2) father pete-pat))
```

The operator `binary-extend-relation` takes two (thus the *binary*) relations and a list of lambda variables, the length of which is the arity of the relation. In the convention we use, the *i* in the last *a_i* in the list is the arity (In `father`, it is 2.) of the relations. It tries to find a result in `rel1`. If it succeeds, it returns a substitution along with a failure continuation, which is to try to find a result in `rel2`. If it fails, then it invokes the just described failure continuation. The early evaluation of the relation expressions are critical in order to guarantee that we can write expressions such as `(define x (extend-relation (a1 a2) x y))`. Without this early evaluation, the original relation `x` would not be part of the new definition of `x`, which would lead to an infinite loop. This is an argument for making `binary-extend-relation` a procedure, but then it would be necessary

to use `(lambda args ...)` and `apply`, which we believe should be avoided, when possible. Later, we present two variants of `binary-extend-relation`.

A convenient abbreviation of a relation where there is no antecedent is described by the macro `fact`.

```
(define-syntax fact
  (syntax-rules ()
    [(_ (ex-id ...) x ...) (relation _ (ex-id ...) (to-show x ...))]))
```

Then we could write the `father` relation like this.

```
(define father
  (binary-extend-relation (a1 a2)
    (fact () 'pete 'sal)
    (fact () 'pete 'pat)))

> (exists (x)
  (let ([result (query (father 'pete x))])
    (let ([subst (caar result)]
          [fk (cdr result)])
      (cons
        (list (subst-in 'pete subst) (subst-in x subst))
        (let ([result (fk)])
          (let ([subst (caar result)]
                [fk (cdr result)])
            (cons
              (list (subst-in 'pete subst) (subst-in x subst))
              (let ([result (fk)])
                (if (null? result)
                    '()
                    (error 'father "Pete only has two children"))))))))))))

((pete sal) (pete pat))
```

First, we add that Pete is also the father of Pat. Then, when we invoke `fk` the first time, we get another result. When we invoke it a second time, we are out of results, so we get back the empty list, the result of invoking the initial failure continuation.

3.4 Streams (infinite lists) provide a natural interface

We would like to abstract the previous program in a more coherent way. Later, we see an example where there is no limit on the number of answers, but if we want to process the answers as a list, we must place some bound on the size of the list.

The `(car result)` is a substitution/cut-continuation pair, and the `(cdr result)` is a thunk that if invoked returns a value whose `car` is a substitution/cut-continuation pair, and whose `cdr` is a thunk, etc. This is a stream (possibly infinite list) of such pairs. We write the function, `stream-prefix`, which given a bound `n`, and a stream,

`strm`, yields a list with at most one element, if `n` is 0; at most two elements, if `n` is 1; etc. But, we must be careful that going for the `n+1`st element of a stream does not lead to an infinite loop, even though we only want the first `n` elements! This causes `stream-prefix` to be written in a slightly awkward fashion.

```
(define stream-prefix
  (lambda (n strm)
    (if (null? strm) '()
        (cons (car strm)
              (if (zero? n) '()
                  (stream-prefix (- n 1) ((cdr strm))))))))
```

Before we go any further, we have some details about displaying answers that needs addressing. This might seem a bit bizarre at first reading, but there is only one reason for this code: *Variables are not eye-pleasing*. So, we want to devise a good way to display them. Normally, a logic system uses a `gensym`, but then the results are very difficult to follow. So rather than use generated identifiers, we construct our own artificial ones.

```
(define artificial-id
  (lambda (var-c)
    (string->symbol
     (string-append
      (symbol->string (var-id (car var-c))) ". " (number->string (cdr var-c))))))
```

```
> (exists (who) (artificial-id '(,who . 5)))
```

```
who.5
```

We associate a unique artificial identifier with each variable. Each variable, however, only stores a symbol within it. Two different variables could easily have the same symbol within it, so there needs to be a mechanism to have two different artificial identifiers. The way we solve this problem is to use an environment that is both an input and an output result. Each item in the environment binds a variable to a counter. If the variable is found in the environment, then the associated artificial identifier is returned. If it is not found in the environment, then a second search using `assv/var-id` below is attempted. If a variable built from the same identifier is in the environment, then a new item is added to the environment, but its counter is one larger than the previous one. If these two attempts fail, then the new item added to the environment starts the counter at zero. This is the job of `concretize-var` below.

```

(define concretize-var
  (lambda (var env)
    (cond
      [(assv var env)
       => (lambda (var-c) (values (artificial-id var-c) env))]
      [else (let ([var-c '(,var . ,(cond
                                   [(assv/var-id (var-id var) env)
                                    => (lambda (var-c) (+ (cdr var-c) 1))]
                                   [else 0])])]
                (values (artificial-id var-c) (cons var-c env))))))]

(define assv/var-id
  (lambda (id env)
    (cond
      [(null? env) #f]
      [(eqv? (var-id (caar env)) id) (car env)]
      [else (assv/var-id id (cdr env))])))

```

The only thing to notice about this code is that two items are returned using values as the result of any call to `concretize-var`.

Because we are in a logic system, we don't know if we are concretizing a variable or a trivial constant. Thus, we use a driver `concretize-term`, which dispatches appropriately. *When we add pairs as a kind of term, then we need to extend this definition.*

```

(define concretize-term
  (lambda (t env)
    (cond
      [(var? t) (concretize-var t env)]
      [else (values t env)])))

```

In keeping with our philosophy of avoiding gratuitous lists, we have the macro `concretize-sequence`, which builds a list of concretized terms from a bunch of terms.

```

(define-syntax concretize-sequence
  (syntax-rules ()
    [(_ t0 ...) (concretize-sequence-aux '() t0 ...)]))

(define-syntax concretize-sequence-aux
  (syntax-rules ()
    [(_ env) '()]
    [(_ env t0 t1 ...)
     (let-values ((ct env) (concretize-term t0 env))
       (cons ct (concretize-sequence-aux env t1 ...)))]))

```

Let us return to our discussion about how we are going to use `stream-prefix`. Once we have a finite list of substitutions, we are free to use them however we wish.

For example, we define a macro `solve` that takes a positive integer upper bound and a relation call. It returns a list like the one above for `pete`'s children. Each variable is replaced by the associated term of its commitment.

```
(define-syntax solve
  (syntax-rules ()
    [(_ n (rel t ...))
     (map (lambda (subst/cutk)
           (concretize-sequence (subst-in t (car subst/cutk)) ...))
          (stream-prefix (- n 1) (query (rel t ...)))))]))
> (exists (x) (solve 5 (father 'pete x)))

((pete sal) (pete pat))
```

Of course, the resultant list contains only two answers, but asking for five does no harm, since it quits early when the `null?` test in `stream-prefix` holds.

Let's see what this call partially expands to.

```
> (expand-only '(solve)
  '(exists (x) (solve 5 (father 'pete x))))

(exists (x)
  (map (lambda (subst/cutk)
        (concretize-sequence
          (subst-in 'pete (car subst/cutk))
          (subst-in x (car subst/cutk))))
        (stream-prefix (- 5 1) (query (father 'pete x)))))
```

If we further expand `concretize-sequence`, we see exactly what this does.

```
(exists (x)
  (map (lambda (subst/cutk)
        (let-values (ct new-env)
          (concretize-term (subst-in 'pete (car subst/cutk)) ())
          (cons ct
                (let-values (ct new-env)
                  (concretize-term (subst-in x (car subst/cutk)) new-env)
                  (cons ct ()))))))
        (stream-prefix (- 5 1) (query (father 'pete x)))))
```

Exercise 4: Redefine `solve` to set up an interactive loop to force more answers. Use 0 to indicate no more answers and use + to indicate more answers. \diamond

We can simplify things a bit when we want at most one result with `solution`,

```
(define-syntax solution
  (syntax-rules ()
    [(_ x)
     (let ([ls (solve 1 x)])
       (if (null? ls) #f (car ls)))]))
```

```
> (exists (x) (solution (father 'pete x)))

(pete sal)
```

Exercise 5: Rewrite `solution` by redefining `initial-sk` and `initial-fk`. \diamond

Using `stream-prefix` meets our needs for this tutorial, but in general, it may be too naive. We may want all the answers until some predicate holds about one or more of the answers. For example, the result might be a stream of integers that terminates after the third odd integer appears. In this case, it is impossible to know what integer bound to pass to `stream-prefix`. In those circumstances, it is best to process the stream of substitutions using a hand-crafted procedure.

3.5 Sequences of antecedents

In the definition of `child-of-male`, we have exactly one antecedent. We are going to look at relations with additional antecedents.

Let's first redefine `father`.

```
(define father
  (binary-extend-relation (a1 a2)
    (fact () 'john 'sam)
    (binary-extend-relation (a1 a2)
      (fact () 'sam 'pete)
      (binary-extend-relation (a1 a2)
        (fact () 'pete 'sal)
        (fact () 'pete 'pat))))))
```

Before proceeding, it is time to introduce `extend-relation`, which takes one or more relations.

```
(define-syntax extend-relation
  (syntax-rules ()
    [(_ (id ...) rel-exp) rel-exp]
    [(_ (id ...) rel-exp0 rel-exp1 ...)
     (binary-extend-relation (id ...) rel-exp0
       (extend-relation (id ...) rel-exp1 ...))]))
```

Then we can redefine `father`,

```
(define father
  (extend-relation (a1 a2)
    (fact () 'john 'sam)
    (fact () 'sam 'pete)
    (fact () 'pete 'sal)
    (fact () 'pete 'pat)))
```

So, we see that Sam is the grandfather of Sal and Pat, and John is the grandfather of Pete. We might ask, "Is Sam the grandfather of anyone in our closed five-person world?"

```

(define grandpa-sam
  (relation _ (grandchild)
    (to-show grandchild)
    (exists (parent)
      (all (father 'sam parent) (father parent grandchild))))))
> (exists (y) (solve 6 (grandpa-sam y)))

((sal) (pat))

```

It is correct, because Sam is the father of Pete, and Pete is the father of Sal and Pat. Here is how the substitutions that led to this answer were built. First `y` became some `grandchild`. At this point, `y` is not instantiated, only *shared* with the variable `grandchild`. Then `parent` is instantiated to Pete. Next, `grandchild` is instantiated to Sal, but we know that `y` is shared with `grandchild`, so `y` is also instantiated to Sal. Then we fail, thus re-instantiating `grandchild` to Pat. Finally, the failures back all the way out, leading to the empty list (the result of invoking `initial-fk`).

The operator `any` is the dual of `all`. It invokes a success continuation whenever one of its antecedents succeeds, whereas `all` invokes a failure continuation whenever one of its antecedents fails. Each antecedent is transformed into a zero argument relation. Then the result of creating the extended relation, which we know to be a zero argument relation, is *invoked*.

```

(define-syntax any
  (syntax-rules ()
    [(_ ant ...) ((extend-relation () (relation _ () (to-show) ant) ...))]))

```

The macro call `(any ant)` can, therefore be rewritten as `(relation _ () (to-show) ant)`. Thus, it is possible to turn any antecedent into a zero-argument relation,

```
(define ant->relation
  (lambda (ant)
    (relation _ () (to-show) ant)))
```

Watch how the macros expand down to almost nothing.

```
> (expand-only '(relation exists) '(relation _ () (to-show) ant))
(lambda () (all! (all ant)))
> (expand-only '(relation all) '(relation _ () (to-show) ant))
(lambda () (all! ant))
> (expand-only '(relation all all!) '(relation _ () (to-show) ant))
(lambda () ant)
```

Exercise 6: Show that the output of

```
> (expand-only '(extend-relation binary-extend-relation relation all all! exists)
  '((extend-relation ()
     (relation _ () (to-show) ant1)
     (relation _ () (to-show) ant2))))
```

can easily be translated to this antecedent, which is a two-antecedent any.

```
(lambda@ (sk fk subst cutk)
  (@ ant1 sk (lambda () (@ ant2 sk fk subst cutk)) subst cutk))
```

A one-argument any is

```
(lambda@ (sk fk subst cutk)
  (@ ant sk fk subst cutk))
```

which, of course, η -reduces to ant. But, what is a zero-argument any?

```
(define fail
  (lambda@ (sk fk subst cutk)
    (fk)))
```

Now, armed with this information, redefine the macro any for zero or more arguments. \diamond

The implementation of the basic unsullied logic system is now complete. Most of what we may want to do with a logic system can be programmed with this basic system comprised of `relation` (and `fact`), `extend-relation`, `all`, `all!`, and `solve`, with the two basic constants, `fail` and `succeed`.

Let's return (using `all`) to the most recent definition of `grandpa-sam`. Obviously this is not as general as we might expect. Consider an attempt to take us beyond concerns for Sam.

```
(define grandpa-maker
  (lambda (grandad)
    (relation _ (grandchild)
      (to-show grandchild)
      (exists (parent)
        (all (father grandad parent) (father parent grandchild))))))
```

```
> (exists (x) (solve 6 ((grandpa-maker 'sam) x)))
```

This just uses lexical scope to reconstruct `grandpa-sam`. But, it still requires that we know who we want to find out about. We do something a bit more abstract below. We postpone the determination of the function guide by making it, too, a variable. Here it is lexical, but it can be a logic variable, instead, because procedures are treated as raw values. Thus, we can pass the function `father`, which then gets invoked. This allows for the potential creation of a more abstract relation. For example, if we had a `mother` definition, like our `father` definition, then `grandpa-maker` would still work, as long as `mother` is passed to `grandpa-maker` instead of `father`. Then we would have a matrilineal grandparent instead of a patrilineal one. Of course, then “`grandpa-maker`” and “`grandad`” would be poorly chosen names.

```
(define grandpa-maker
  (lambda (guide grandad)
    (relation _ (grandchild)
      (to-show grandchild)
      (exists (parent)
        (all (guide grandad parent) (guide parent grandchild))))))
```

```
> (exists (x) (solve 4 ((grandpa-maker father 'sam) x)))
((sal) (pat))
```

Next we consider a more concrete problem, which is to use our logic system to define `grandpa`. Now, we can replace the lexical variable `grandad` with the logic variable `grandad`, which leaves the decision completely open.

Here is our first (somewhat naive) definition of `grandpa`.

```
(define grandpa
  (relation _ (grandad grandchild)
    (to-show grandad grandchild)
    (exists (parent)
      (all (father grandad parent) (father parent grandchild))))))
```

```
> (exists (x) (solve 4 (grandpa 'sam x)))
((sam sal) (sam pat))
```

We make Pete an uncle of Betty and David, the children of his sister, Polly. First, we include Polly in the `father` relation. Then, we define a `mother` relation,

including some facts like we did for the father relation. Finally, we add a relation to the grandpa relation, so that Pete's sister's children can claim Sam as their grandfather.

```
(define sam-polly (fact () 'sam 'polly))
(define father (extend-relation (a1 a2) father sam-polly))

(define polly-betty (fact () 'polly 'betty))
(define polly-david (fact () 'polly 'david))
(define mother (extend-relation (a1 a2) polly-betty polly-david))

(define grandpa
  (extend-relation (a1 a2) grandpa
    (relation _ (grandad grandchild)
      (to-show grandad grandchild)
      (exists (parent)
        (all (father grandad parent) (mother parent grandchild))))))
> (exists (y) (solve 10 (grandpa 'sam y)))

((sam sal) (sam pat) (sam betty) (sam david))
```

And we discover that Sam is, indeed, the grandfather of Betty and David.

3.6 Lexically-shared variables

In the previous definition of `grandpa`, both relations use the same variables, so why should we write them twice? Instead, we use `exists` like this,

```
(define grandpa
  (exists (parent grandad grandchild)
    (extend-relation
      (relation _ ()
        (to-show grandad grandchild)
        (all (father grandad parent) (father parent grandchild)))
      (relation _ ()
        (to-show grandad grandchild)
        (all (father grandad parent) (mother parent grandchild))))))
```

And since the `to-show`'s are the same, we can use any like this.

```
(define grandpa
  (relation _ (grandad grandchild)
    (to-show grandad grandchild)
    (exists (parent)
      (any
        (all (father grandad parent) (father parent grandchild))
        (all (father grandad parent) (mother parent grandchild))))))
```

And we can distribute `any`.

```
(define grandpa
  (relation _ (grandad grandchild)
    (to-show grandad grandchild)
    (exists (parent)
      (all (father grandad parent)
        (any (father parent grandchild)
          (mother parent grandchild)))))))
```

Exercise 7: Determine if

```
(define grandpa
  (exists (parent)
    (relation _ (grandad grandchild)
      (to-show grandad grandchild)
      (any
        (all (father grandad parent) (father parent grandchild))
        (all (father grandad parent) (mother parent grandchild))))))
```

works and compare this variant with the previous ones. Which one is better? There are relations for which this last style backfires. Characterize when this placement of `exists` might backfire by defining a more complicated relation. (Although this problem can be solved, here, it might be best to have understood the definition of ancestor described later before trying it.) \diamond

Exercise 8: Suppose that the keyword `to-show` were restricted to take a single argument. Here is an example to study where `grandpa-sam` has the right structure for thinking about such arcane problems.

```
(define grandpa-sam
  (let ([r (relation _ (child)
    (to-show child)
    (exists (parent)
      (all (father 'sam parent) (father parent child))))])
    (relation _ (child)
      (to-show child)
      (r child))))
```

Rewrite some variant of `grandpa` with that restriction. \diamond

We return to the issue of extending relations. We mentioned earlier that we would perhaps want to combine two relations in a different way. Under what circumstances might this come up? Since we are free to think about each relation as returning a stream of results, we can envisage a scenario where the first stream is unbounded. Then no results will arrive from the second stream. We can avoid this problem by combining two relations using `binary-extend-relation-interleave`.

```

(define-syntax binary-extend-relation-interleave
  (syntax-rules ()
    [(_ (id ...) rel-exp1 rel-exp2)
     (let ([rel1 rel-exp1] [rel2 rel-exp2])
       (lambda (id ...)
         (lambda@ (sk fk subst cutk)
           (let ([ant1 (rel1 id ...)] [ant2 (rel2 id ...)])
             ((interleave sk fk (finish-interleave sk fk))
              (@ ant1 initial-sk initial-fk subst cutk)
              (@ ant2 initial-sk initial-fk subst cutk))))))))))

(define interleave
  (lambda (sk fk finish)
    (letrec
      ([interleave
       (lambda (q1 q2)
         (cond
          [(null? q1) (if (null? q2) (fk) (finish q2))]
          [else (let ([fk (cdr q1)] [subst (caar q1)] [cutk (cdar q1)])
                   (@ sk (lambda () (interleave q2 (fk))) subst cutk))]]))
       interleave)))

(define finish-interleave
  (lambda (sk fk)
    (letrec
      ([finish
       (lambda (q)
         (cond
          [(null? q) (fk)]
          [else (let ([fk (cdr q)] [subst (caar q)] [cutk (cdar q)])
                   (@ sk (lambda () (finish (fk))) subst cutk))]]))
       finish])))

```

The interesting part of this code takes place in `interleave`. There, it takes turns taking a result from one stream and then the other. Once it has exhausted one of the streams, it finishes the job on the other one. The success and failure continuations that are passed into `interleave` and `finish-interleave` from the start are the ones that came in as part of the invocation of an antecedent. But, when the actual antecedent gets passed arguments, it uses the initial success and failure continuations.

Exercise 9: Implement `extend-relation-interleave`, which takes an arbitrary number of relations. \diamond

What other variant is possible?

```

(define-syntax binary-extend-relation-interleave-non-overlap
  (syntax-rules ()
    [(_ (id ...) rel-exp1 rel-exp2)
     (let ([rel1 rel-exp1] [rel2 rel-exp2])
       (lambda (id ...)
         (lambda@ (sk fk subst cutk)
           (let ([ant1 (rel1 id ...)] [ant2 (rel2 id ...)])
             ((interleave-non-overlap sk fk (finish-interleave sk fk))
              (@ ant1 initial-sk initial-fk subst cutk)
              (@ ant2 initial-sk initial-fk subst cutk)
              fail
              ant2)))))))]))

(define interleave-non-overlap
  (lambda (sk fk finish)
    (letrec
      ([interleave
       (lambda (q1 q2 ant1 ant2)
         (cond
          [(null? q1) (if (null? q2) (fk) (finish q2))]
          [else (let ([fk (cdr q1)] [subst (caar q1)] [cutk (cdar q1)])
                  (if (satisfied? ant2 subst)
                      (interleave q2 (fk) ant2 ant1)
                      (@ sk (lambda () (interleave q2 (fk) ant2 ant1)) subst cutk)))))]])
       interleave)))

(define satisfied?
  (lambda (ant subst)
    (not (null? (@ ant initial-sk initial-fk subst initial-cutk)))))

```

Here, again, *finish* takes whatever is left to do after one of the streams is exhausted. But, this time if it can be shown that either antecedent can be satisfied with the same substitution, we drop one of the results.

In the remainder of the paper, we present some sullied operators, consider the role of recursion in our logic system, and study some famous examples that display some of the power of logic programming.

4 Sullied antecedents

In this section we introduce the sullied operators, which reveal some parts of the underlying structure. In each instance, we are expanding the kinds of antecedents that are available. Among these are the two features that often work together: *cut* and *fail*; four operators for interfacing with Scheme: *pred-call*, *fun-call*, *pred-nocheck*, and *fun-nocheck*; and three useful, miscellaneous operators: *fails*, *instantiated*, and *view-subst*.

The grammar for antecedents `<A>` completes the language.

```
<A> =
  <relation call>
  | (all <A>*)
  | (all! <A>*)
  | (any <A>*)
  | <sullied antecedent>
  | <or any Scheme expression that evaluates to these,
    since these are all values.>
```

```
<sullied antecedent> =
  | cut
  | fail
  | (pred-call <t> <t>*)
  | (fun-call <t> <t> <t>*)
  | (pred-call/no-check <t> <t>*)
  | (fun-call/no-check <t> <t> <t>*)
  | (fails <A>)
  | (instantiated <var>)
  | (view-subst <t>)
```

where `<t>` is a term, and `<var>` is a variable.

4.1 Short cuts and fail

In this section we introduce the conventional *cut* operator. The relation operator takes a lexical variable as its first argument, and it finds itself bound to `(!! cutk)` inside the big lambda expression. What `!!` does is take the current `cutk` failure continuation and build an antecedent that always succeeds, but as part of succeeding it installs that `cutk` as the failure continuation. Thus it exits the relation call if failure backs into it.

We present three variants of `grandpa`, each with only one use of `cut` below. The first one places it as the last antecedent of `grandpa/father`. If we look back at the definition of `relation`, we can see that `cut` becomes a lexically-scoped variable.

```
(define grandpa/father
  (relation cut (grandad grandchild)
    (to-show grandad grandchild)
    (exists (parent)
      (all (father grandad parent) (father parent grandchild) cut))))
```

expands to

```
(define grandpa/father
  (lambda (g1 g2)
    (exists (grandad grandchild)
      (all!
        (== g1 grandad)
        (== g2 grandchild)
        (lambda@ (sk fk subst cutk)
          (let ([cut (!! cutk)])
            (@ (all (exists (parent)
                      (all (father grandad parent) (father parent grandchild) cut)))
              sk fk subst cutk))))))))))
```

The procedure `!!` takes a failure continuation `cutk` and returns an antecedent. If the antecedent is invoked, it *succeeds* and replaces the current failure continuation with `cutk`. Thus, if failure backs into it, the attempt to find a result for this relation call is abandoned. To see this in action, we redefine `grandpa` below.

```
(define grandpa/mother
  (relation _ (grandad grandchild)
    (to-show grandad grandchild)
    (exists (parent)
      (all (father grandad parent) (mother parent grandchild))))))
```

```
(define grandpa
  (extend-relation (a1 a2) grandpa/father grandpa/mother))
> (exists (x y) (solve 10 (grandpa x y)))
```

```
((john pete))
```

For `!!`, `exiting-fk` is the `cutk` of the call to `grandpa`. The effect of using `cut` at the end is that once the first answer is found and failure forced, the `exiting-fk` that has been passed as the failure continuation is invoked, so it takes us completely out of the call to `grandpa`.

Next, consider the following revision of `grandpa`, where we swap the last two antecedents of `grandpa/father`.

```
(define grandpa/father
  (relation cut (grandad grandchild)
    (to-show grandad grandchild)
    (exists (parent)
      (all (father grandad parent) cut (father parent grandchild))))))
```

```
(define grandpa
  (extend-relation (a1 a2) grandpa/father grandpa/mother))
> (exists (x y) (solve 10 (grandpa x y)))
```

```
((john pete) (john polly))
```

The variable `grandad` cannot be re-instantiated because of the invocation of the `exiting-fk`, but `parent` and `grandchild` can. As failure works its way back into the same call to `father`, new instantiations for `parent` and `grandchild` are found. If we run out of data to match `parent` and `grandchild`, then instead of looking for the next match of `grandad` and `parent`, we invoke the `cutk` that was passed to `!!`. So, we have merely two answers. One way to describe this is to state that we only want the *first* father's grandchildren. So, if we moved the facts about Pete and his children to the top of `father`, we would get no answers, because then Pete would be the first father and he has no grandchildren.

Finally, we have the last variation of `grandpa`, where the `cut` is the first antecedent of `grandpa/father`.

```
(define grandpa/father
  (relation cut (grandad grandchild)
    (to-show grandad grandchild)
    (exists (parent)
      (all cut (father grandad parent) (father parent grandchild))))))

(define grandpa
  (extend-relation (a1 a2) grandpa/father grandpa/mother))

> (exists (x y) (solve 10 (grandpa x y)))

((john pete) (john polly) (sam sal) (sam pat))
```

This variation yields only paternal grandfathers, because when it runs out of fathers as potential grandfathers, `cutk` is invoked. This is the same as completely ignoring the second relation. If we reorganize the facts in `father` as we did at the end of the previous example, we get the same four answers. If Pete were the father of Betty, too, then she would show up as Sam's grandchild. She would have appeared only once, however, because `cutk`'s invocation has precluded us from using the `grandpa/mother` relation in all three examples.

It is okay to use `cut` more than once within a single rule. Be careful, however, to fully appreciate its consequences each time it is used. Such a powerful control mechanism must be handled with great care. In the type inferencer we have several antecedents where there is more than one occurrence of `cut`.

An idiom is to `cut` and then fail immediately. Doing the `cut` always succeeds and the `exiting` doesn't happen until failure works its way back into the `cut`. One way to force that failure is with the `fail` antecedent. For example, we might say that if someone is a *mother* she cannot be a *grandpa* under any circumstances.

```
(define fail
  (lambda@ (sk fk subst cutk)
    (fk)))
```

```

(define no-grandma
  (relation cut (grandad grandchild)
    (to-show grandad grandchild)
    (exists (parent)
      (all (mother grandad parent) cut fail))))

(define no-grandma-grandpa
  (extend-relation (a1 a2) no-grandma grandpa))

> (exists (x) (solution (no-grandma-grandpa 'polly x)))
#f

```

4.2 Interfacing Scheme functions

Suppose we want to restrict the answers of `grandpa` so that someone isn't a grandfather unless his child's name starts with the letter, "p." John's child's name is Sam, so John is no longer considered a grandfather. But, Sam's child's name is Pete, so Pete's children are still someone's grandchildren.

```

(define grandpa
  (relation _ (grandad grandchild)
    (to-show grandad grandchild)
    (exists (parent)
      (all
        (father grandad parent)
        (starts-with-p?-rel parent)
        (father parent grandchild)))))

(define starts-with-p?
  (lambda (x)
    (and
      (symbol? x)
      (string=? (string (string-ref (symbol->string x) 0)) "p"))))

(define starts-with-p?-rel
  (lambda (x)
    (lambda@ (sk fk subst cutk)
      (if (starts-with-p? (nonvar! (subst-in x subst)))
          (@ sk fk subst cutk)
          (fk)))))

(define nonvar!
  (lambda (t)
    (if (var? t)
        (error 'nonvar! "Variable found in call: ~s" (concretize-var t))
        t)))

```

```
> (exists (x y) (solve 10 (grandpa x y)))
```

```
((sam sal) (sam pat))
```

Let's look closely at the Scheme function `starts-with-p?-rel`. Like other relations, it returns a function that expects the same arguments: a success continuation, a substitution, and two failure continuations. If `x` (after it has been substituted for) is needed for the actual test, then it should not still be a variable, which is the job of `nonvar!`, above. It is certainly possible for an argument to get substituted by a variable, but in this case it would not be meaningful. If `starts-with-p?` test is true, then we take the success path, which is *always* `(@ sk fk subst cutk)`, otherwise we take the failure path, which is *always* `(fk)`.

The predicate `starts-with-p?` is arbitrary. Any Scheme predicate of any number of arguments can be used. The Scheme predicate acting as a relation can do anything that any Scheme predicate can do, including capturing continuations, setting variables, writing to files, etc. The full panoply of options is available to the user. The only requirement is that these relations take the same four arguments, and that they exit the same way: `(@ sk fk subst cutk)` for success and `(fk)` for failure. We abstract this with a new operator, `pred-call` below.

```
(define-syntax pred-call
  (syntax-rules ()
    [(_ p t ...)
     (lambda@ (sk fk subst cutk)
       (if ((nonvar! (subst-in p subst)) (nonvar! (subst-in t subst)) ...)
           (@ sk fk subst cutk)
           (fk)))]))
```

Also, we can call *any* Scheme function and unify the results of the call with something else. For example, we use `(fun-call * 15 3 5)` as an antecedent. We know that all but the *second* argument (15 here) should be a value after substitution, so we are safe to apply the Scheme function to those values. We then unify the answer with the second argument. If that second argument is an uninstantiated variable or contains an uninstantiated variable, then this causes the exiting substitution to be larger than the entering one.

```
(define-syntax fun-call
  (syntax-rules ()
    [(_ f t u ...)
     (lambda@ (sk fk subst cutk)
       (cond
        [(unify t ((nonvar! (subst-in f subst)) (nonvar! (subst-in u subst)) ...) subst)
         => (lambda (subst)
              (@ sk fk subst cutk))]
        [else (fk)])))]))
```

```
> (exists (q) (solution (fun-call * q 3 5)))
(#<procedure *> 15 3 5)
```

The Scheme interface operators `pred-call` and `fun-call` catch errors when calling in the presence of variables. There are times, however, when you want to pass a variable to some Scheme procedure. For those times, we use `pred-call/no-check` and `fun-call/no-check` below. The `pred-calls` allow the user to treat Scheme predicates as logic predicates, whereas the `fun-calls` allow the user to treat Scheme functions as logic functions (A logic function is a logic relation that has at most one value for each input.) Since functions return a value, `fun-call` and `fun-call/no-check` cause that value to get unified with the (possibly uninstantiated) second argument. The “/no-check” versions below are for conveniently writing extensions to the logic system that *can* handle logic variables.

```
(define-syntax pred-call/no-check
  (syntax-rules ()
    [(_ p t ...)
     (lambda@ (sk fk subst cutk)
       (if ((subst-in p subst) (subst-in t subst) ...)
           (@ sk fk subst cutk)
           (fk)))]))

(define-syntax fun-call/no-check
  (syntax-rules ()
    [(_ f t u ...)
     (lambda@ (sk fk subst cutk)
       (cond
        [(unify t ((subst-in f subst) (subst-in u subst) ...) subst)
         => (lambda (subst)
              (@ sk fk subst cutk))]
        [else (fk)])))]))
```

4.2.1 *fails, instantiated, and view-subst*

Below is the function `fails` that fails when its goal succeeds and succeeds when its goal fails. Basically, we hand build both the success and failure continuations.

```
(define fails
  (lambda (ant)
    (lambda@ (sk fk subst cutk)
      (@ ant
         (lambda@ (ign-fk ign-subst ign-cutk) (fk))
         (lambda () (@ sk fk subst cutk))
         subst
         cutk)))
```

And here is an example of fails.

```
(define grandpa
  (relation _ (grandad grandchild)
    (to-show grandad grandchild)
    (exists (parent)
      (all
        (father grandad parent)
        (fails (starts-with-p?-rel parent))
        (father parent grandchild))))))
> (exists (x y) (solve 10 (grandpa x y)))
```

```
((john pete) (john polly))
```

To determine if a variable, *t*, is instantiated, use `(instantiated t)` as an antecedent and this definition.

```
(define instantiated
  (lambda (t)
    (pred-call/no-check (lambda (x) (not (var? x))) t)))
```

At this point, *t* is either some variable or some value.

To view a substitution, use `(view-subst t)` with this definition. The definition of `concretize-subst` should be obvious, since it simply concretizes every variable in a substitution.

```
(define view-subst
  (lambda (t)
    (lambda@ (sk fk subst cutk)
      (pretty-print (subst-in t subst))
      (pretty-print (concretize-subst subst))
      (@ sk fk subst cutk))))

(define concretize-subst
  (letrec
    ([cs (lambda (subst env)
      (cond
        [(null? subst) '()]
        [else
         (let ([comm (car subst)])
           (let-values (cv env) (concretize-var (commitment->var comm) env)
             (let-values (ct env) (concretize-term (commitment->term comm) env)
               (cons (commitment cv ct) (cs (cdr subst) env)))))]))]])
    (lambda (subst)
      (cs subst '()))))
```

Here is a new definition of `grandpa`, like an earlier one, but views the substitution.

```
(define grandpa
  (relation _ (grandad grandchild)
    (to-show grandad grandchild)
    (exists (parent)
      (all
        (father grandad parent)
        (father parent grandchild)
        (view-subst grandchild))))))

> (exists (x y) (solve 10 (grandpa x y)))

pete
((grandad.0 x.0)
 (grandchild.0 y.0)
 (x.0 john)
 (parent.0 sam)
 (y.0 pete))

polly
((grandad.0 x.0)
 (grandchild.0 y.0)
 (x.0 john)
 (parent.0 sam)
 (y.0 polly))

sal
((grandad.0 x.0)
 (grandchild.0 y.0)
 (x.0 sam)
 (parent.0 pete)
 (y.0 sal))

pat
((grandad.0 x.0)
 (grandchild.0 y.0)
 (x.0 sam)
 (parent.0 pete)
 (y.0 pat))

((john pete) (john polly) (sam sal) (sam pat))
```

This completes the discussion of the features and implementation of our logic system. We have accomplished this using merely eight people. The population of our world is about to grow.

5 Recursive definitions

In this section, we introduce two interesting problems. The first finds the youngest common ancestor of two people in our world. The second is the well-known “Towers of Hanoi” problem. The second is interesting because it uses recursion, `pred-call`, and `fun-call` in one relation. We start with the youngest common ancestor problem.

5.1 Youngest common ancestor

Suppose that we want to know if someone is a (patrilineal) ancestor. We know that if someone old is the father of someone young, then the old person is a patrilineal ancestor. But, also, if the old person is the father of someone not so old, and the not so old person is the ancestor of the young person, then we know that we have an ancestor. This way of describing ancestor is defined directly in our logic system. We add a few more facts to `father` to make the outcomes a bit more interesting. Specifically, we add that “John is the father of Harry,” “Harry is the father of Carl,” and “Sam is the father of Ed.”

```
(define father
  (extend-relation (a1 a2) father
    (fact () 'john 'harry)
    (fact () 'harry 'carl)
    (fact () 'sam 'ed)))
```

We are now ready to solve the problem that we stated earlier. To do this, we first define and test the ancestor relation.

```
(define ancestor
  (extend-relation
    (relation _ (old young)
      (to-show old young)
      (father old young))
    (relation _ (old young)
      (to-show old young)
      (exists (not-so-old)
        (all (father old not-so-old) (ancestor not-so-old young))))))
```

Exercise 10: Redefine the previous definition of `ancestor` as one relation using any. \diamond

Here is a simple test.

```
> (exists (x y) (solve 100 (ancestor x y)))
```

```
((john sam)
 (sam pete)
 (pete sal)
 (pete pat)
 (sam polly)
 (john harry)
 (harry carl)
 (sam ed)
 (john pete)
 (john polly)
 (john ed)
 (john sal)
 (john pat)
 (sam sal)
 (sam pat)
 (john carl))
```

Once we have the concept of ancestor, it is easy to think in terms of a common ancestor. Two people share a common ancestor if somewhere along their respective ancestor chains, their paths cross. Here is how we can write that in our logic system.

```
(define common-ancestor
  (relation _ (young-a young-b old)
    (to-show young-a young-b old)
    (all (ancestor old young-a) (ancestor old young-b))))
```

```
> (exists (x) (solve 4 (common-ancestor 'pat 'ed x)))
```

```
((pat ed john) (pat ed sam))
```

This says that John and Sam are both common ancestors of Pat and Ed.

If two people share two common ancestors, then we can determine if one of the common ancestors is younger than the other common ancestor.

```
(define younger-common-ancestor
  (relation _ (young-a young-b old not-so-old)
    (to-show young-a young-b old not-so-old)
    (all
      (common-ancestor young-a young-b not-so-old)
      (common-ancestor young-a young-b old)
      (ancestor old not-so-old))))
```

According to relation, the “(all” and matching “)” are unnecessary, but we use them for clarification.

```
> (exists (x y) (solve 4 (younger-common-ancestor 'pat 'ed x y)))

((pat ed john sam))
```

We finally come to the problem that we are most interested in, which is how do we determine the youngest common ancestor. We already know that Pat and Ed share Sam and John, but we want the youngest among the common ancestors. Since John must be older than Sam, the answer should be Sam.

```
(define youngest-common-ancestor
  (relation _ (young-a young-b not-so-old)
    (to-show young-a young-b not-so-old)
    (all
      (common-ancestor young-a young-b not-so-old)
      (exists (y)
        (fails (younger-common-ancestor young-a young-b not-so-old y))))))

> (exists (x) (solve 4 (youngest-common-ancestor 'pat 'ed x)))

((pat ed sam))
```

The tricky part of this is that if we find a younger ancestor, then the one we have chosen is not the youngest common ancestor. So, the ancestor is the youngest common ancestor provided each attempt to find a younger common ancestor fails.

What is interesting about this approach is that it is recursive: we define ancestor in terms of ancestor. Although we don't use recursion to implement our logic system, our model relies heavily on the idea that users are facile with recursion and we rely heavily on the fact that Scheme supports recursion. For instance, the recursion supported by `define` in these four definitions could just as easily have been supported using `letrec`. Furthermore, we can bind `father` lexically, so that the expression takes any binary relation as an argument.

```
(define youngest-common-ancestor
  (lambda (father)
    (letrec ([ancestor ...]
              [common-ancestor ...]
              [younger-common-ancestor ...]
              [youngest-common-ancestor ...])
      youngest-common-ancestor)))
```

This would also solve a problem related to an organizational chart. If we pass a relation, say `supervisor`, instead of `father`, then `youngest-common-ancestor` would find the least common supervisor. In general, this procedure produces the *least upper bound*, provided one exists, of a *finite* binary relation.

5.2 Comparison with Seres and Spivey

The similarities with Seres and Spivey are striking. In fact, a subset of our model maps directly onto theirs, although our goal has been a full implementation of logic programming, including many of the sullied antecedents as well as some meta operations.

Using `==` our Scheme embedding resembles their Haskell embedding. Where we differ is that we would be limited to using only `any`, `all`, and `==`. We would no longer have `relation` and `extend-relation`, and all the sullied operators. This means that we could use `lambda` to define relations. For example, we can define `father` and `ancestor` like this,

```
(define father
  (lambda (dad child)
    (any
      (all (== dad 'john) (== child 'sam))
      (all (== dad 'sam) (== child 'pete))
      (all (== dad 'sam) (== child 'polly))
      (all (== dad 'pete) (== child 'sal))
      (all (== dad 'pete) (== child 'pat))
      (all (== dad 'john) (== child 'harry))
      (all (== dad 'harry) (== child 'carl))
      (all (== dad 'sam) (== child 'ed))))))

(define ancestor
  (lambda (old young)
    (any
      (father old young)
      (exists (not-so-old)
        (all (father old not-so-old) (ancestor not-so-old young))))))
```

And this can be tested similarly.

```
> (exists (x) (solve 20 (ancestor 'john x)))

((john sam)
 (john harry)
 (john pete)
 (john polly)
 (john ed)
 (john sal)
 (john pat)
 (john carl))
```

Does the Haskell embedding differ in any significant way from our embedding in Scheme? Yes, besides avoiding sullied operators, there is a fundamental difference. Everything that works in the Haskell embedding works in the Scheme embedding, but not vice versa. Because the Haskell embedding does unification piecewise within

an all, any time that the unification fails to match, there is automatic backtracking. That is not the case with `relation` and `fact`. In our embedding, if unification fails, it invokes the failure continuation, immediately. It is as though each relation has a lock on it and a term either opens it or a new one is tried. In order to accomplish this in the Haskell embedding, they would need `all!` or something similar.

In sum, the Haskell embedding is good as far as it goes, but it leaves many of the more interesting aspects of logic programming unresolved. The approach of treating every antecedent as a stream of substitutions is not far from our approach. Since we have substitutions `subst` and failure continuations `fk` in every antecedent, we could use a different representation of the body of the antecedent, where all but `fk` would comprise a data structure and the `fk` would be a stream (`thunk`). Then each antecedent closure could be modelled as the consing of the data structure to a stream. Because this is a relatively small program, making this last step in order to use the stream monad seems like overkill, especially since this approach does not appear to extend naturally to the all important sullied antecedents.

5.3 “Towers of Hanoi”

Three poles (a *left*, a *middle*, and a *right*) can hold disks of various sizes provided that a larger one is never on top of a smaller one. The initial state of the problem has a set of n disks (all different sizes) sitting on the left pole. The goal is to place the entire set of disks on the middle pole. Only the top disk of any pole can be moved to a different pole and then that disk becomes the top disk of the chosen pole.⁵

```
(define towers-of-hanoi
  (letrec
    ([move
      (extend-relation (a1 a2 a3 a4)
        (relation cut () (to-show 0 _ _ _) cut)
        (relation _ (n a b c)
          (to-show n a b c)
          (exists (m)
            (all
              (pred-call positive? n)
              (fun-call - m n 1)
              (move m a c b)
              (pred-call printf "Move a disk from ~s to ~s~n" a b)
              (move m c b a))))))]
      (relation _ (n)
        (to-show n)
        (move n 'left 'middle 'right))))
```

⁵ This solution has been derived from page 141 of *Programming in Prolog Fourth Edition* by W. F. Clocksin and C. S. Mellish.

```
> (begin (solution (towers-of-hanoi 3)) (void))
```

```
Move a disk from left to middle
Move a disk from left to right
Move a disk from middle to right
Move a disk from left to middle
Move a disk from right to left
Move a disk from right to middle
Move a disk from left to middle
```

The algorithm is straightforward. First, figure out how to solve the problem for one fewer disks and then move that stack of disks as a virtual disk. The use of `pred-call` allows us to have a procedure that writes one move. Since we are displaying information, we know that the value of `printf` is not false. Also, the move subtracts one from the number of disks using `fun-call` and its first relation uses the anonymous variable three times. (Recall that the anonymous variable unifies with everything, but it adds nothing to the substitution; the underscore after `relation` has been chosen arbitrarily. If the scope uses underscore, choose a variable that is not live in the scope. It is always okay to use `cut`, however, limiting its use to when the program relies on it seems more prudent.)

5.4 The difference between success and truth

We revisit any to make a point about the difference between *success* and *truth*. Consider the following three relations (`test1`, `test2`, and `test3`), which rely only on `lambda` as in the Haskell embedding.

```
> (define test1
  (lambda (x)
    (any (pred-call < 4 5) (fun-call < x 6 7))))

> (exists (x) (solution (test1 x)))
(x.0)

> (define test2
  (lambda (x)
    (any (pred-call < 5 4) (fun-call < x 6 7))))

> (exists (x) (solution (test2 x)))
(#t)

> (define test3
  (lambda (x y)
    (any (fun-call < x 5 4) (fun-call < y 6 7))))

> (exists (x y) (solution (test3 x y)))
(#f y.0)
```

The first two should not be much of a surprise. In the first example, 4 is less than 5, so `x` remains uninstantiated, and 6 is less than 7, so `x` is instantiated in the second example. In the last example, however, we discover that `y` remains uninstantiated even though 5 is *not* less than 4. Why? A `fun-call` always succeeds if its second argument is an uninstantiated variable. So, succeeding is not the same as being true in a logic system.

6 Three famous problems

Three famous problems that we discuss are the so-called `append` problem, the type-inference problem of a typed variant of the lambda calculus with constants, conditional expressions, primitives, and polymorphic `let`, and a generalization of Prolog's name predicate. Before we can begin this walk down memory lane, we must enlarge the set of possible terms and consequently change the behavior of three functions: `subst-in`, `concretize-term`, and `unify*`.

6.1 Enlarging the set of terms

Presently, a term is a variable or something that can be compared trivially. Now, we include in the set of terms those values that cannot be compared trivially, such as pairs and vectors. We restrict our concerns to pairs.

```
(define subst-in
  (lambda (t subst)
    (cond
      [(var? t)
       (cond
         [(assv t subst) => commitment->term]
         [else t])]
      [(pair? t) (cons (subst-in (car t) subst) (subst-in (cdr t) subst))]
      [else t])))

(define concretize-term
  (lambda (t env)
    (cond
      [(var? t) (concretize-var t env)]
      [(pair? t)
       (let-values ((carct env) (concretize-term (car t) env))
         (let-values ((cdrct env) (concretize-term (cdr t) env))
           (values (cons carct cdrct) env)))]
      [else (values t env)])))
```

These are not significant changes. They each say walk recursively through the term and replace all variables by their association in a substitution (or in a threaded environment in the case of `concretize-term`).

Here is an interesting example,

```

> (exists (x y z)
  (let ([term '(p ,x ,y (g ,z))])
    (let ([s (compose-subst (unit-subst y z) (unit-subst x '(f ,y)))]
          [r (compose-subst (unit-subst x 'a) (unit-subst z 'b))])
      (let ([new-term (subst-in term s)])
        (printf "~s~n" (concretize new-term))
        (printf "~s~n" (concretize (subst-in new-term r)))
        (let ([sr (compose-subst s r)]
              (printf "~s~n" (concretize-subst sr))
              (concretize (subst-in term sr)))))))
  (p (f y.0) z.0 (g z.0))
  (p (f y.0) b (g b))
  ((y.0 b) (x.0 (f y.0)) (z.0 b))
  (p (f y.0) b (g b))

```

In the example, we demonstrate a fact about substitutions: it does not matter if we apply the substitutions to a term one at a time or apply the composed substitution to the same term.

Next, we must change `unify*` (Recall that `unify*` is called from `unify`.) to accommodate this new kind of term. We can ask if two data structures are `equal?`, which does a recursive tree walk, comparing subparts. If they are equal, `equal?` responds with `true`, otherwise, it responds with `false`. The redefinition of `unify*` below shares that attribute with `equal?`. That is, when two data structures are equal, `unify*` returns the empty substitution, otherwise, it returns `false`. But, unification is more than just equality. The function `unify*` takes two terms, and returns a substitution that allows for the two terms to be perceived as *equal* if the substitution were applied to the two terms. To accomplish this, each variable in the two terms is added to the substitution by associating some term with it. In the recursive tree walk, two leaves that are aligned must be `equal?`. If one argument has a variable and the other one contains something other than a variable, then that pair is added to the substitution. If the same variable is aligned in both arguments, then the substitution remains unchanged. If both are variables, then one of them is treated as the term. Each time something is added to the substitution, it is a commitment. So, as the recursive tree walk continues, it refines previous commitments.

The version of `unify*` below takes any two terms and returns a substitution if the terms unify and returns `false`, otherwise.

```

(define unify*
  (lambda (t u)
    (cond
      [(trivially-equal? t u) empty-subst]
      [(var? t) (if (occurs? t u) #f (unit-subst t u))]
      [(var? u) (if (occurs? u t) #f (unit-subst u t))]
      [(and (pair? t) (pair? u))
       (cond
         [(unify* (car t) (car u))
          => (lambda (s-car)
              (cond
                [(unify* (subst-in (cdr t) s-car) (subst-in (cdr u) s-car))
                 => (lambda (s-cdr)
                     (compose-subst s-car s-cdr))]
                [else #f])]]
          [else #f])]]
      [else #f]))))

```

There are only three kinds of terms for us to consider: trivial values, variables, and pairs. The first case is straightforward. Two terms (including two variables) unify and return the empty substitution if they are trivially equal. If at least one of them is a variable, then the other becomes the term associated with it, provided that that term does not contain an occurrence of the variable. Two pairs might unify if their cars unify. But, before we determine if the cdrs unify, we must take the substitution returned from successfully unifying the cars, its commitments, and apply it to the cdrs. This brings the cdrs up-to-date with those commitments. Then all the commitments from unifying the cars, *s-car*, and all the commitments from unifying the substituted for cdrs, *s-cdr*, are composed.

Do not underestimate just how difficult it is to understand the pair case. As it is written, it looks rather straightforward, but when we process a deeply nested term, we make lots of commitments and they must be repeatedly refined by future substitutions. It is a testament to the power of recursion that it can look this simple.

The check for an occurrence of a variable in a term below is a straightforward recursive tree walk, however, for reasons of efficiency, most logic systems have chosen not to include it. Later, we show a variant of unification that simply avoids it even though there might be an occurrence.

```

(define occurs?
  (lambda (var t)
    (cond
      [(var? t) (eqv? t var)]
      [(pair? t) (or (occurs? var (car t)) (occurs? var (cdr t)))]
      [else #f]))))

```

Exercise 11: Each of the functions, *subst-in*, *concretize-term*, *unify**, and *occurs?* treats pairs. Include vectors as terms and allow the values in a vector to be any term. ◇

Exercise 12: All calls to `unify` can be improved by passing it success and failure continuations, making sure to thread them through the auxiliary calls. Revise the entire system to make this improvement. \diamond

6.2 Unification examples

There are many ways for two terms to fail to unify. For example, it is possible for the same variable to appear twice in one structure but be associated with different integers. Another easy way for two terms to fail to unify is to have the same variable appear once in each term but be associated with different integers. These two ways are demonstrated below. We start with some handy (global) variables,

```
> (define u (exists (u) u))
> (define w (exists (w) w))
> (define x (exists (x) x))
> (define y (exists (y) y))
> (define z (exists (z) z))
```

```
> (unify '(,x ,4) '(3 ,x) empty-subst)
```

```
#f
```

```
> (unify '(,x ,x) '(3 4) empty-subst)
```

```
#f
```

In the preceding examples, `x` commits to 3, but then looking at the second elements of each term, there is an attempt to commit `x` to 4, which violates an earlier commitment. The following two examples do unify, however.

```
> (concretize-subst (unify '(,x ,y) '(3 4) empty-subst))
```

```
((x.0 3) (y.0 4))
```

```
> (concretize-subst (unify '(,x 4) '(3 ,y) empty-subst))
```

```
((x.0 3) (y.0 4))
```

First, `x` commits to 3 and then `y` commits to 4. Let's try a slightly more difficult example.

```
> (concretize-subst (unify '(,x 4 3 ,w) '(3 ,y ,x ,z) empty-subst))
```

```
((x.0 3) (y.0 4) (w.0 z.0))
```

We can see that the last two items in each term do not violate earlier commitments. Thus, the two terms unify. If any other integer appears where the 3 in the first argument appears, then these fail to unify. If `y` appears where the `x` appears in

the second argument, then these also fail to unify. The variable `w` shares with the variable `z`, which means that when one of them commits to some non-variable term, so does the other one. Consider these two examples.

```
> (concretize-subst (unify '(,x 4) '(,y ,y) empty-subst))
```

```
((x.0 4) (y.0 4))
```

```
> (unify '(,x 4 3) '(,y ,y ,x) empty-subst)
```

```
#f
```

In the first example, `x` commits to `y` and agrees to be associated with the same term. Then `y` commits to `4`. In the second example, once `y` has committed to `4`, since `y` and `x` are shared, `x` is committed to `4`, but `x` tries to commit to `3`, which violates an earlier commitment.

6.2.1 More unification examples

We present five unification examples below.

```
> (concretize-subst (unify '(,w ,(x ,(y ,z) 8)) '(,w ,(u (abc ,u) ,z)) empty-subst))
```

```
((x.0 8) (y.0 abc) (z.0 8) (u.0 8))
```

```
> (concretize-subst (unify '(p (f a) (g ,x)) '(p ,x ,y) empty-subst))
```

```
((x.0 (f a)) (y.0 (g (f a))))
```

```
> (concretize-subst (unify '(p (g ,x) (f a)) '(p ,y ,x) empty-subst))
```

```
((y.0 (g (f a))) (x.0 (f a)))
```

```
> (concretize-subst (unify '(p a ,x (h (g ,z))) '(p ,z (h ,y) (h ,y)) empty-subst))
```

```
((z.0 a) (x.0 (h (g a))) (y.0 (g a)))
```

```
> (unify '(p ,x ,x) '(p ,y (f ,y)) empty-subst)
```

```
#f
```

The first example commits `w` to itself, which is no commitment at all. Then `x` shares with `u`, `y` commits to the symbol `abc`, `z` shares with `u`, and `z` commits to `8`. In the second example, the `p` symbols match, leading to no commitments. Then, `x` commits to `(f a)`. Finally, `y` commits to `(g (f a))`, since the `x` in `'(g ,x)` gets replaced by what `x` is committed to, `(f a)`. The third example produces the same values for the variables, but in a different way. Again, `p` symbols lead to no commitments. Then `y` commits to `'(g ,x)`. Finally, `x` commits to `(f a)`. The fourth example first commits `z` to the symbol `a`. Then, `x` commits to `'(h ,y)`. Next the `h` symbols

contribute nothing. Finally, y commits to (g, z) . But, since z has committed to a , we know that y must become $(g a)$, which is why x 's value is $(h (g a))$. The fifth example fails to unify, but only when we reach the last item in each term. The p symbols unify. Then, x is shared with y . When we try to unify x with (f, y) , we discover that since y has been replaced by x , we are really unifying x with $(f x)$. It should be clear that this is a violation of the *occurs check* in the second `cond` clause of `unify`.

6.2.2 A lazy unify

A different approach to writing `unify` is to postpone when we apply a substitution. This postponing makes it possible to avoid applying substitutions in two separate circumstances. First, if somewhere deep in the two terms we have not seen any variable, but we have found a place where the two terms fail to unify, then we return false without having applied any substitutions. Also, if the two terms unify, but contain no variables, we also avoid applying a substitution. This differs significantly from the preceding `unify*`, where we applied a substitution to every `cdr`.

In this definition of `unify`⁶ below the excitement happens only when a variable is found in at least one of the terms. Otherwise, we walk recursively through the two terms making sure that they are equal until we find a variable in either term. Then we pass the variable, the substituted for other term, and the substitution to `unify/var` below.

```
(define unify
  (lambda (t u subst)
    (cond
      [(trivially-equal? t u) subst]
      [(var? t) (unify/var t (subst-in u subst) subst)]
      [(var? u) (unify/var u (subst-in t subst) subst)]
      [(and (pair? t) (pair? u))
       (cond
         [(unify (car t) (car u) subst)
          => (lambda (subst)
              (unify (cdr t) (cdr u) subst))]
         [else #f])]
      [else #f])))

(define unify/var
  (lambda (t-var u subst)
    (cond
      [(assv t-var subst) (unify (subst-in t-var subst) u subst)]
      [(occurs? t-var u) #f]
      [else (compose-subst subst (unit-subst t-var u))])))
```

⁶ This version is similar to the algorithm on pages 219–222 in Dybvig's *The Scheme Programming Language ANSI Scheme, second edition*. There, however, the substitutions are represented as functions from terms to terms.

In `unify/var`, we must compose the substitution with a commitment to be built with the variable `t-var` and the term `u`. As part of the call, we have brought `u` up-to-date. If `t-var` is committed in the substitution, we start the whole process over again after bringing `t-var` up-to-date, too. Otherwise, we try to commit `u` to `t-var`.

6.3 Unification without the occurs check

It is also possible to completely avoid the occurs check even if there is an occurs-check violation. In most Prologs, there is a conscious decision to avoid the occurs check, but not too many of them have a way to avoid it even if there is a violation. In this section we present such a unifier.⁷

Before we can discuss the unifier, we must redefine `subst-in` to account for decisions that will be made within the unifier. We choose to use `subst-in` sparingly, so when we do use it, we have more work to do. Before, `subst-in` merely replaced each variable by its associated term. Now, however, it starts over with that term. If it is not committed, it stops.

```
(define subst-in
  (lambda (t subst)
    (cond
      [(var? t)
       (cond
         [(assv t subst)
          => (lambda (c)
              (subst-in (commitment->term c) subst))]
         [else t]]])
      [(pair? t)
       (cons (subst-in (car t) subst) (subst-in (cdr t) subst))]
      [else t])))
```

The unifier below avoids `subst-in` until it is ready to do a `compose-subst`. The two auxiliary procedures each take a variable, but one takes another variable and the other takes something that is not a variable.

⁷ The trick used here also appears in SICTus Prolog.

```
(define unify
  (lambda (t u subst)
    (cond
      [(trivially-equal? t u) subst]
      [(var? t) (if (var? u) (unify-var/var t u subst) (unify-var/value t u subst))]
      [(var? u) (unify-var/value u t subst)]
      [(and (pair? t) (pair? u))
       (cond
         [(unify (car t) (car u) subst)
          => (lambda (car-subst)
              (unify (cdr t) (cdr u) car-subst))]
         [else #f]]]
      [else #f])))
```

The procedure `unify-var/var`, unifies the term associated with the first variable against the term associated with the second one, if they are committed. If one of them is committed and the other is not, then we set up a loop in `uncommitted/committed` below that follows the committed one until either the same variable comes up, or it ends up with a value. If both variables are uncommitted, and we already know that they are not the same from the test in `unify`, then we naively extend the substitution. It's probably easier to skip to the committed `u-var` case first, before studying the committed `t-var` case.

```
(define unify-var/var
  (lambda (t-var u-var s)
    (cond
      [(assv t-var s)
       => (lambda (ct)
          (cond
            [(assv u-var s)
             => (lambda (cu)
                  (let ([u-term (commitment->term cu)]
                        [t-term (commitment->term ct)])
                    (unify t-term u-term s)))]
            [else ((uncommitted/committed u-var s) ct)])]]
      [(assv u-var s) => (uncommitted/committed t-var s)]
      [else (extend-subst t-var u-var s)])))
```

Formally, we could define `extend-subst` like this,

```
(define extend-subst
  (lambda (var t refining)
    (compose-subst (unit-subst var t) refining)))
```

but, consider the simplicity of `extend-subst` below. We have a variable and a term that we would like to naively include in a substitution. If the variable has no association in the substitution, the variables in the term are not in `(map commitment->var`

refining), and if the term is not the same as the variable, then we can use this naive definition,

```
(define extend-subst
  (lambda (var t refining)
    (cons (commitment var t) refining)))
```

But, we must convince ourselves that it is the right definition. Clearly, we must be certain that its first two arguments are not the same, for if they were, the result would be the original substitution. Also, if the first argument has an association in the substitution, then we know that its associated term would be removed from the original substitution. Finally, if the second argument contains any variables with associations in the substitution, their associated terms would affect the term that would be committed in the extended substitution.

```
(define uncommitted/committed
  (lambda (t-var s)
    (lambda (cu)
      (let loop ([cm cu])
        (let ([u-term (commitment->term cm)])
          (cond
            [(eqv? u-term t-var) s]
            [(var? u-term)
             (cond
               [(assv u-term s) => loop]
               [else (unify-var/value t-var u-term s)]]])
            [else (extend-subst t-var u-term s)]))))))
```

The procedure `unify-var/value` is the interesting part of this algorithm. Now we know that the second argument is a non-variable (i.e., a value). First, we check to see if the variable is committed and if so, unify against its associated term.. Now, if the variable is *uncommitted* and the value of the second argument is a *pair*, then we have the situation like before when we used the `occurs` check. But, this time, we will do something else. Since we know that we have a pair, we construct a pair with two new logic variables. Then we place the pair in the substitution. We use the new substitution to unify the first new logic variable with the car of the term and when that returns, we use the resultant substitution to unify the second new logic variable with the cdr of the term. If it not a pair, we do the same as above.

```

(define unify-var/value
  (lambda (t-var u-value s)
    (cond
      [(assv t-var s)
       => (lambda (ct)
            (let ([t-term (commitment->term ct)])
              (unify t-term u-value s)))]
      [(pair? u-value)
       (let ([car-var (var ':a)]
             [cdr-var (var ':d)])
         (cond
           [(unify car-var (car u-value)
                    (extend-subst t-var (cons car-var cdr-var) s))
            => (lambda (s)
                 (unify cdr-var (cdr u-value) s))]
           [else #f])])]
      [else (extend-subst t-var u-value s)])))

```

This allows us to produce a legitimate substitution in the example that failed the occurs check. In order to actually view the substitution, we need to use an auxiliary procedure `subst-vars-recursively`, which is nearly the same as `subst-in`, but when it follows the associated term of a variable, it makes sure that the associated commitment is not found again by removing it from the substitution.

```

(define subst-vars-recursively
  (lambda (t subst)
    (cond
      [(var? t)
       (cond
         [(assv t subst) =>
          (lambda (c)
            (subst-vars-recursively
              (commitment->term c) (remq c subst)))]
         [else t]]]
      [(pair? t)
       (cons
         (subst-vars-recursively (car t) subst)
         (subst-vars-recursively (cdr t) subst))]
      [else t]))))

> (concretize-subst
  (let ([s (unify '(p ,x ,x) '(p ,y (f ,y)) empty-subst)])
    (let ([vars (map commitment->var s)])
      (map commitment vars (subst-vars-recursively vars s)))))

((:d.0 ())

```

```
(:a.0 (f :a.0))
(:d.1 ((f . :d.1)))
(:a.1 f)
(y.0 (f (f . :d.1)))
(x.0 (f (f . :d.1)))
```

If we look closely at the output, we can see the circularity in the associated term of `:a.0` and `:d.1`. Therefore, any term that contains them is circular, which includes the terms associated with `x.0`, and `y.0`.

6.3.1 “Towers of Hanoi” revisited

Before, we look at the three famous problems, let’s take another look at the “Towers of Hanoi” problem. It is less than satisfying that the solution is not a value returned but just some displaying of information. Instead, we can replace those effects by other effects and build the answer in a table. Then, that path can be the result. Thus, we are free to write functions that process the path. For example, we can now determine how many steps it takes for any `n`. Before, we were limited by our willingness to read and process screens or files.

```
(define towers-of-hanoi-path
  (let ([steps '()])
    (let ([push-step (lambda (x y) (set! steps (cons '(,x ,y) steps)))]
      (letrec
        ([move
          (extend-relation (a1 a2 a3 a4)
            (relation cut () (to-show 0 _ _ _) cut)
            (relation _ (n a b c)
              (to-show n a b c)
              (exists (m)
                (all
                  (pred-call positive? n)
                  (fun-call - m n 1)
                  (move m a c b)
                  (pred-call push-step a b)
                  (move m c b a))))))]
          (relation _ (n path)
            (to-show n path)
            (begin
              (set! steps '())
              (any
                (fails (move n 'l 'm 'r))
                (== path (reverse steps))))))))))

> (exists (path) (solution (towers-of-hanoi-path 3 path)))
(3 ((l m) (l r) (m r) (l m) (r l) (r m) (l m)))
```

The primary difference between this version and the earlier version is that in this version there is a lexical variable `steps` that holds each step, where before we printed each step. Then, by forcing failure with `fails`, we are guaranteed to process the second rule. It always succeeds, since `path` is guaranteed to be uninstantiated. We reverse the steps so that it looks like our earlier output. Everything else is the same.

Exercise 13: Use this definition of `towers-of-hanoi-path` to produce a table of the number of disks with the number of steps it takes to move that number of disks. \diamond

Our unifier now handles pairs (lists), so we can continue the discussion of the three famous problems.

6.4 The Append Problem

We can often mimic value-returning functions with relations that take an *additional* argument. For example, we can write a function that concatenates *two* lists.

```
(define concat
  (lambda (xs ys)
    (cond
      [(null? xs) ys]
      [else (cons (car xs) (concat (cdr xs) ys))])))
```

```
> (concat '(a b c) '(u v))
```

```
(a b c u v)
```

or the equivalent

```
> (exists (q) (cdr (solution (fun-call concat q '(a b c) '(u v)))))
```

```
((a b c u v) (a b c) (u v))
```

And we can write the corresponding relation over *three* lists,

```
(define concat
  (extend-relation
    (fact (xs) '() xs xs)
    (relation _ (x xs ys zs)
      (to-show '(,x . ,xs) ys '(,x . ,zs))
      (concat xs ys zs))))
```

```
> (exists (q) (solve 6 (concat '(a b c) '(u v) q)))
```

```
((a b c) (u v) (a b c u v))
```

which determines that there is only one answer and which shows if we concatenate `(a b c)` to `(u v)`, we get `(a b c u v)`. But, we can move `q` to another position.

```
> (exists (q) (solve 6 (concat '(a b c) q '(a b c u v))))
```

```
((a b c) (u v) (a b c u v))
```

This time we determine that q should be $(u\ v)$, which is *not* possible with `concat` as a function. Similarly, we can determine that q is $(a\ b\ c)$.

```
> (exists (q) (solve 6 (concat q '(u v) '(a b c u v))))
```

```
((a b c) (u v) (a b c u v))
```

But what if we include another variable?

```
> (exists (q r) (solve 6 (concat q r '(a b c u v))))
```

```
((() (a b c u v) (a b c u v))
 ((a) (b c u v) (a b c u v))
 ((a b) (c u v) (a b c u v))
 ((a b c) (u v) (a b c u v))
 ((a b c u) (v) (a b c u v))
 ((a b c u v) () (a b c u v)))
```

We get all the ways that we might concatenate two lists to form $(a\ b\ c\ u\ v)$. Now, what if we include yet another variable?

```
> (exists (q r s) (solve 6 (concat q r s)))
```

```
((() xs.0 xs.0)
 ((x.0) xs.0 (x.0 . xs.0))
 ((x.0 x.1) xs.0 (x.0 x.1 . xs.0))
 ((x.0 x.1 x.2) xs.0 (x.0 x.1 x.2 . xs.0))
 ((x.0 x.1 x.2 x.3) xs.0 (x.0 x.1 x.2 x.3 . xs.0))
 ((x.0 x.1 x.2 x.3 x.4) xs.0 (x.0 x.1 x.2 x.3 x.4 . xs.0)))
```

Here we see that the empty list and any list yield that list. Then we get all sorts of constructed lists with the first N elements of the list chosen as variables of that length. There is no bound on the number of answers.

We can also get an unbounded number of answers with only two variables.

```
> (exists (q r) (solve 6 (concat q '(u v) '(a b c . ,r))))
```

```
((a b c) (u v) (a b c u v))
 ((a b c x.0) (u v) (a b c x.0 u v))
 ((a b c x.0 x.1) (u v) (a b c x.0 x.1 u v))
 ((a b c x.0 x.1 x.2) (u v) (a b c x.0 x.1 x.2 u v))
 ((a b c x.0 x.1 x.2 x.3) (u v) (a b c x.0 x.1 x.2 x.3 u v))
 ((a b c x.0 x.1 x.2 x.3 x.4) (u v) (a b c x.0 x.1 x.2 x.3 x.4 u v)))
```

The first answer is the one we expect, where q is instantiated to $(a\ b\ c)$ and r is instantiated to $(u\ v)$. But, then we discover that q could be a bit longer.

And here is an unbounded number of answers with a single variable.

```
> (exists (q) (solve 6 (concat q '() q)))

((( () ()))
 ((x.0) () (x.0))
 ((x.0 x.1) () (x.0 x.1))
 ((x.0 x.1 x.2) () (x.0 x.1 x.2))
 ((x.0 x.1 x.2 x.3) () (x.0 x.1 x.2 x.3))
 ((x.0 x.1 x.2 x.3 x.4) () (x.0 x.1 x.2 x.3 x.4)))
```

Again, the first answer is the one that we expect, but the others make sense, too, since no matter what we replace the variables $x.i$ with, we create a legitimate equation.

A program like `concat` is what excited the logic programming world. It was called `append` because of the use of that name in Lisp, but since `concat` is defined globally, it would be wise to avoid overriding the built-in Scheme function, `append`.

6.5 The Type-Inference Problem

The second famous problem is the type-inference problem. We start by considering integers and booleans. Next, we include some familiar primitives. When we are comfortable with those features we include conditionals, followed by lexical variables, then `lambda`, application, and `fix` expressions. Finally, we include polymorphic `let`. Type inference allows for the system to determine a unique type if the expression has one.

6.5.1 Building a type inferencer with small relations

The language for which we infer a type is basically a lambda-calculus variant of Scheme. We have chosen, however, to parse this variant into a language where every expression has a tag as in `parse` below. We have also included an `unparse` below to get back the original Scheme variant.

```

(define parse
  (lambda (e)
    (cond
      [(symbol? e) '(var ,e)]
      [(number? e) '(intc ,e)]
      [(boolean? e) '(boolc ,e)]
      [else
       (case (car e)
         [(zero?) '(zero? ,(parse (cadr e)))]
         [(sub1) '(sub1 ,(parse (cadr e)))]
         [(+) '(+ ,(parse (cadr e)) ,(parse (caddr e)))]
         [(if) '(if ,(parse (cadr e)) ,(parse (caddr e)) ,(parse (caddr e)))]
         [(fix) '(fix ,(parse (cadr e)))]
         [(lambda) '(lambda ,(cadr e) ,(parse (caddr e)))]
         [(let) '(let ([,(car (car (cadr e))]) ,(parse (cadr (car (cadr e)))))]
                  ,(parse (caddr e)))]
         [else '(app ,(parse (car e)) ,(parse (cadr e)))]))]))

(define unparse
  (lambda (e)
    (case (car e)
      [(var) (cadr e)]
      [(intc) (cadr e)]
      [(boolc) (cadr e)]
      [(zero?) '(zero? ,(unparse (cadr e)))]
      [(sub1) '(sub1 ,(unparse (cadr e)))]
      [(+) '(+ ,(unparse (cadr e)) ,(unparse (caddr e)))]
      [(if) '(if ,(unparse (cadr e)) ,(unparse (caddr e)) ,(unparse (caddr e)))]
      [(fix) '(fix ,(unparse (cadr e)))]
      [(lambda) '(lambda (,(car (cadr e))) ,(unparse (caddr e)))]
      [(let)
       '(let ([,(car (car (cadr e))])
              ,(unparse (cadr (car (cadr e)))))]
          ,(unparse (caddr e)))]
      [(app) '(,(unparse (cadr e)) ,(unparse (caddr e)))]))]))

```

While you are reading the code for the type system, `!-`, it is important to keep in mind that although we are presenting a type inferencing algorithm, it is just a relatively simple logic program.

`!-` corresponds to the mathematical symbol \vdash (turnstile) and reads “we can infer.” That is, from looking at the relation `int-rel` and the definition of `!-` below, we can read it as, “From *g we can infer* that *x* is of type `int` provided that *x* is an `intc`.” For now, we leave *g* unspecified.

In the expressions below, we use `int`, `bool`, and `-->` for our type constructors. We define `int` and `bool` to avoid using lots of quotes.

```
(define int 'int)
(define bool 'bool)

(define int-rel
  (fact (g x) g '(intc ,x) int))

(define !- int-rel)

(define g (exists (g) g))

> (let ([result (solution (!- g (parse 17) int))])
    '(!- ,(car result) ,(unparse (cadr result)) ,(caddr result)))
(!- g.0 17 int)

(define ? (exists (?) ?))

> (let ([result (solution (!- g (parse 17) ?))])
    '(!- ,(car result) ,(unparse (cadr result)) ,(caddr result)))
(!- g.0 17 int)
```

In the first example, we verify that 17 is of type `int`. In the second example, the type is unknown, but whatever is instantiated to the variable `?` is the type. In this case, `?` is instantiated to `int`. The existence of `g.0` in the answers indicates that `g` is uninstantiated, so these work for all possible `gs`.

As a way to abstract the behavior of our testing technology, we use `infer-type` below, which abbreviates some of the repetition of the first two examples and supports the idea that the expression might not have a type.

```
(define-syntax infer-type
  (syntax-rules ()
    [(_ g term type)
     (cond
      [(solution (!- g (parse term) type))
       => (lambda (result)
            '(!- ,(car result) ,(unparse (cadr result)) ,(caddr result)))]
      [else #f])]))
```

Next, we include `bool-rel` below in `!-`.

```
(define bool-rel
  (fact (g x) g '(boolc ,x) bool))

(define !- (extend-relation (a1 a2 a3) !- bool-rel))
```

Exercise 14: Test `infer-type` over true and false. ◇

Before we include relations for the arithmetic primitives, we observe that we need to use `all!`. This means that whenever there is failure, all the antecedents fail. We do this because our type inferencer has this deterministic behavior. There cannot be any backtracking. That is, once a decision is made, it cannot be reconsidered!

Now we can extend `!-` below with the relations `zero?-rel`, `sub1-rel`, and `+rel`, which correspond to the primitives `zero?`, `sub1`, and `+`, respectively.

```
(define zero?-rel
  (relation (g x)
    (to-show g '(zero? ,x) bool)
    (all! (!- g x int))))

(define sub1-rel
  (relation (g x)
    (to-show g '(sub1 ,x) int)
    (all! (!- g x int))))

(define +-rel
  (relation (g x y)
    (to-show g '(+ ,x ,y) int)
    (all! (!- g x int) (!- g y int))))

(define !- (extend-relation (a1 a2 a3) !- zero?-rel sub1-rel +-rel))

(define ? (exists (?) ?))
> (infer-type g '(zero? 24) ?)

(!- g.0 (zero? 24) bool)
> (infer-type g '(zero? (+ 24 50)) ?)

(!- g.0 (zero? (+ 24 50)) bool)
```

The type system can infer that `(zero? 24)` is of type `bool`, because it can infer that `24` is of type `int`. It can infer that `(+ 24 50)` is of type `int`, so the answer in the second example must be of type `bool`. We can, of course, make more complicated examples using `zero?`, `sub1`, and `+`, but if they have a type, it is `int` or `bool`. For example,

```
> (infer-type g '(zero? (sub1 (+ 18 (+ 24 50)))) ?)

(!- g.0 (zero? (sub1 (+ 18 (+ 24 50)))) bool)
```

Although our parser (and unparser) expects a larger language, at each stage of defining `!-`, we are writing a type inferencer for a larger and larger language. When we have defined `!-` for `let`, then we have a type inferencer for the full language. To reiterate, the language starts out very small! It only contains integers. Then, as we progress, it gets bigger. But, the beauty of type inferencing, is that these little

relations grow naturally. Of course, we cannot write the relation for `zero?` until we have a relation for numbers and booleans, so there is a natural ordering to some extent.

In this type system, we must preserve the property that every well-typed expression has *one* type. So, what do we do about conditionals? Easy. We require that not only must the test be of type `bool`, but the true branch and the false branch must have the same type. In a language without variables, lambda expressions, applications, `fix` expressions, and `let` expressions, that means that they must both be of type `int` or they must both be of type `bool`. By extending `!-`, we can now handle `if` expressions.

```
(define if-rel
  (relation (g t test conseq alt)
    (to-show g '(if ,test ,conseq ,alt) t)
    (all! (!- g test bool) (!- g conseq t) (!- g alt t))))

(define !- (extend-relation (a1 a2 a3) !- if-rel))

> (infer-type g '(if (zero? 24) 3 4) ?)
```

```
(!- g.0 (if (zero? 24) 3 4) int)
```

Not surprisingly, we discover that the type of the test is `bool` and the type of the entire expression is `int`.

Next, we include lexical variables, which are represented using symbols. What is the type of `(zero? a)`? If the type of `a` is `int`, then we know that the type of the entire expression is `bool`, but if the type of `a` is `bool`, then the expression does not have a type. How do we determine the type of `a`? We look in `g`, which is a type environment that associates lexical (both generic and non-generic) variables with types. So far we have ignored `g`, but now we consider its content using non-generic (`a tag`) variables. We extend `!-` to include a relation for variables.

```
(define var-rel
  (relation (g v t)
    (to-show g '(var ,v) t)
    (all! (env g v t))))

(define !- (extend-relation (a1 a2 a3) !- var-rel))

(define non-generic-match-env
  (fact (g v t) '(non-generic ,v ,t ,g) v t))

(define non-generic-recursive-env
  (relation (g v t w type-w)
    (to-show '(non-generic ,w ,type-w ,g) v t)
    (all! (instantiated g) (env g v t))))
```

```

(define env (extend-relation (a1 a2 a3)
  non-generic-match-env
  non-generic-recursive-env)))
> (solution (env '(non-generic b int (non-generic a bool ,g)) 'a ?))

((non-generic b int (non-generic a bool g.0)) a bool)
> (infer-type '(non-generic a int ,g) '(zero? a) ?)

(!- (non-generic a int g.0) (zero? a) bool)
> (infer-type '(non-generic b bool (non-generic a int ,g)) '(zero? a) ?)

(!- (non-generic b bool (non-generic a int g.0)) (zero? a) bool)

```

The first example tests `env`. The environment starts out with `int` bound to `b` and `bool` bound to `a`. The `non-generic-recursive-env` relation succeeds, since we are looking up `a`, and then the `non-generic-match-env` relation succeeds, since we find `a`. In the second answer, we have one item in the type environment and the `non-generic-match-env` relation is followed. In the third example, we have two items in the type environment, so we take the `non-generic-recursive-env` relation, then the `non-generic-match-env` relation succeeds, since we have stripped off `b` and `bool`, leaving `a` and its associated type.

Now that we can deal with lexical (non-generic) variables, we can consider the relation for lambda expressions by extending `!-`.

```

(define lambda-rel
  (relation (g v t body type-v)
    (to-show g '(lambda (,v) ,body) '(--> ,type-v ,t))
    (all! (!- '(non-generic ,v ,type-v ,g) body t))))

(define !- (extend-relation (a1 a2 a3) !- lambda-rel))

```

```

> (infer-type
  '(non-generic b bool (non-generic a int ,g))
  '(lambda (x) (+ x 5))
  ?)

(!- (non-generic b bool (non-generic a int g.0))
    (lambda (x) (+ x 5))
    (--> int int))

> (infer-type
  '(non-generic b bool (non-generic a int ,g))
  '(lambda (x) (+ x a))
  ?)

(!- (non-generic b bool (non-generic a int g.0))
    (lambda (x) (+ x a))
    (--> int int))

> (infer-type g '(lambda (a) (lambda (x) (+ x a)))) ?)

(!- g.0 (lambda (a) (lambda (x) (+ x a)))
    (--> int (--> int int)))

```

In the first answer, we see that we have an arrow (`-->`) type. The left argument of the arrow type is the type of argument coming into the function, and the right argument of the arrow type is the type of the result going out of the function. So, the inferred type is a function whose argument is an integer and whose result is an integer. The second answer states that the argument is an integer, but consults the type environment to make sure that the argument going out is an integer. In the third example, we forget about the first environment, because there are no free variables in the expression. We see that the argument coming in is an integer, but the result is an arrow type, which takes in an integer and returns an integer. Close inspection of the type (directly aligned below the item it is typing) shows that for each lambda there is an arrow and for each formal parameter there is a type. Also, there is a type for the body of each lambda. If we think about the type from the inside out, we see that `(+ x a)` is an integer only if `x` and `a` are integers. That determines the type of the inner lambda and then the type of the outer lambda. It is important that we *can* infer the type of lambda expressions, even though we do not yet have application in our language. This should be a bit of a surprise.

We come next to application. In determining the type of an application, we know that the operator in an application should be some arrow type. Furthermore, once we know that type, we know that the type going out of that type is the same as the type of the entire application and we know that the operand of the application must be the type going into that type.

```

(define app-rel
  (relation (g t rand rator)
    (to-show g '(app ,rator ,rand) t)
    (exists (t-rand)
      (all! (!- g rator '(--> ,t-rand ,t)) (!- g rand t-rand))))))

(define !- (extend-relation (a1 a2 a3) !- app-rel))

> (infer-type g '(lambda (f) (lambda (x) ((f x) x))) ?)

(!- g.0
  (lambda (f) (lambda (x) ((f x) x)))
  (--> (--> type-v.0
          (--> type-v.0 t.0))
  (--> type-v.0 t.0)))

```

Here, the type of f is $(\rightarrow \text{type-v.0 } (\rightarrow \text{type-v.0 } t.0))$, so the type of x must be type-v.0 , and the type of $(\lambda (x) ((f x) x))$ must be $(\rightarrow \text{type-v.0 } t.0)$. As should be evident, once we add a relation for application, things start to get a bit tricky. We can no longer rely on aligning the lambdas with the arrows. Here we have two lambdas and four arrows. Yet another surprise. Our inferencer is starting to be clever.

We may be tempted to use our language to write (and test) recursive functions. To test the expression, we use the call-by-value `fix` primitive:

```

(define fix-rel
  (relation (g rand t)
    (to-show g '(fix ,rand) t)
    (all! (!- g rand '(--> ,t ,t)))))

(define !- (extend-relation (a1 a2 a3) !- fix-rel))

```

In Scheme, we define `fix` below. Although `fix` can be defined using simple lambda terms as in the Y combinator, *this* type system cannot determine a type for it. Thus, `fix` must be primitive and the associated primitive call's type can be inferred as above.

```

(define fix
  (lambda (e)
    (e (lambda (z) ((fix e) z)))))

```

```
> (infer-type
  g
  '((fix (lambda (sum)
          (lambda (n)
            (if (zero? n)
                0
                (+ n (sum (sub1 n)))))))
    10)
  ?)
```

```
(!- g.0
  ((fix (lambda (sum)
          (lambda (n)
            (if (zero? n)
                0
                (+ n (sum (sub1 n)))))))
    10)
  int)
```

Let's consider the following expression

```
> ((fix (lambda (sum)
          (lambda (n)
            (+ n (sum (sub1 n))))))
    10)
```

It fails to terminate. But, can we infer its type? Yes, *an expression may have a type, even if evaluating it would lead to nontermination*. This is a confusing aspect of type inference. We know from the “Halting Problem” that we cannot tell in advance whether evaluating an arbitrary expression will terminate, but *this* type inferencing system is guaranteed to terminate. Thus, we can infer the type before we run it. As a result, information that the run-time system can learn from the type (or the process of inferring the type) can be put to good use. Here is its type.

```
> (infer-type
  g
  '((fix (lambda (sum)
          (lambda (n)
            (+ n (sum (sub1 n))))))
    10)
  ?)

(!- g.0
  ((fix (lambda (sum) (lambda (n) (+ n (sum (sub1 n)))))) 10)
  int)
```

6.6 Polymorphic let

The let-expression is a bit more subtle. Let's take a look at an expression that should type check, but won't in the absence of let.

```
> (infer-type
  g
  '((lambda (f)
     (if (f (zero? 5))
         (+ (f 4) 8)
         (+ (f 3) 7)))
    (lambda (x) x))
  ?)
#f
```

Because `f` becomes the non-generic identity, once a type for `f` is determined, it must stay the same. Obviously, we would expect that the evaluation of “((lambda (f) ...) ...)” to be 10, but it has no type. If, however, we change the expression to use `let`

```
(let ([f (lambda (x) x)])
  (if (f (zero? 5))
      (+ (f 4) 8)
      (+ (f 3) 7)))
```

and think about β -substituting for `f` throughout, then we can see that this expression should have a type, `int`. Instead of doing the substitution, we mark certain variables *generic* as they are placed in the environment.

This is the *polymorphic* let, since the variable is tagged with `generic` in the environment. If the variable were tagged with `non-generic`, then this would be the familiar `let`. We have only to determine what happens in environment lookup when a variable with a `generic` tag is an arrow type. Those who wish to include a more general `let` expression, one whose binding variable is bound to a non-generic, feel free to do so, but for our purposes, we assume that all right-hand sides of `let` expressions are lambda expressions.

```
(define polylet-rel
  (relation (g v rand body t)
    (to-show g '(let ([v ,rand] ,body) t)
      (exists (t-rand)
        (all!
          (!- g rand t-rand)
          (!- '(generic ,v ,t-rand ,g) body t))))))
```

```
(define !- (extend-relation (a1 a2 a3) !- polylet-rel))
```

In order to implement these generics, we introduce a relation, `instantiate`, which uses `let*-inject/no-check`, whose purpose is to associate the type `(--> targ tresult)` with the type `t`.

```

(define instantiate
  (letrec
    ([instantiate-term
      (lambda (t env)
        (cond
          [(var? t)
            (cond
              [(assv t env)
                => (lambda (pr)
                    (values (cdr pr) env))]
              [else (let ([new-var (var (var-id t))]
                          (values new-var (cons '(,t . ,new-var) env)))]))]
          [(pair? t)
            (let-values ((a-t env) (instantiate-term (car t) env)
                          (d-t env) (instantiate-term (cdr t) env)
                          (values (cons a-t d-t) env)))]
          [else (values t env)]))])
    (lambda (t)
      (let-values ((ct env) (instantiate-term t '()
                                                ct))))))

(define generic-base-env
  (relation (g v targ tresult t)
    (to-show '(generic ,v (--> ,targ ,tresult) ,g) v t)
    (let*-inject/no-check ([t^ (targ tresult) (instantiate '(--> ,targ ,tresult))]
                             (== t t^))))

(define generic-recursive-env
  (relation/cut cut (g v w type-w t)
    (to-show '(generic ,w ,type-w ,g) v t)
    (all! cut (env g v t))))

(define generic-env
  (extend-relation (a1 a2 a3)
    generic-base-env
    generic-recursive-env))

(define env
  (extend-relation (a1 a2 a3)
    env
    generic-env))

```

Now that we have extended our environments to handle generic as well as non-generic variables, we can infer the right type.

```
> (infer-type
   g
   '(let ([f (lambda (x) x)])
        (if (f (zero? 5))
            (+ (f 4) 8)
            (+ (f 3) 7))))
   ?)

(!- g.0
 (let ([f (lambda (x) x)])
   (if (f (zero? 5))
       (+ (f 4) 8)
       (+ (f 3) 7)))
 int)
```

6.6.1 Long cuts

Each call returns whenever the computation backs into the cut, but that means that recursive calls just return by following the control stack of procedure calls. Instead, we would prefer to pick the point where we want to be whenever failure backs into a cut. This is possible and we demonstrate it by passing a specific cut into the function `!-generator` below. Thus no matter how deeply nested we are in an expression that we wish to type check, as soon as an invalid expression occurs or we have a type violation, we jump *way* out.

```

(define !-generator
  (lambda (long-cut)
    (letrec
      ([!- (extend-relation (a1 a2 a3)
        (relation (g v t)
          (to-show g '(var ,v) t)
          (all long-cut (env g v t)))
        (fact (g x) g '(intc ,x) int)
        (fact (g x) g '(boolc ,x) bool)
        (relation (g x)
          (to-show g '(zero? ,x) bool)
          (all long-cut (!- g x int)))
        (relation (g x)
          (to-show g '(sub1 ,x) int)
          (all long-cut (!- g x int)))
        (relation (g x y)
          (to-show g '(+ ,x ,y) int)
          (all long-cut (all! (!- g x int) (!- g y int))))
        (relation (g t test conseq alt)
          (to-show g '(if ,test ,conseq ,alt) t)
          (all long-cut
            (all! (!- g test bool) (!- g conseq t) (!- g alt t))))
        (relation (g v t body type-v)
          (to-show g '(lambda (,v) ,body) '(--> ,type-v ,t))
          (all long-cut (!- '(non-generic ,v ,type-v ,g) body t)))
        (relation (g t rand rator)
          (to-show g '(app ,rator ,rand) t)
          (exists (t-rand)
            (all long-cut
              (all!
                (!- g rator '(--> ,t-rand ,t))
                (!- g rand t-rand))))))
        (relation (g rand t)
          (to-show g '(fix ,rand) t)
          (all long-cut (!- g rand '(--> ,t ,t))))
        (relation (g v rand body t)
          (to-show g '(let ([,v ,rand]) ,body) t)
          (exists (t-rand)
            (all long-cut
              (all!
                (!- g rand t-rand)
                (!- '(generic ,v ,t-rand ,g) body t)))))))]))
    !-)))

```

```
(define !-
  (relation/cut cut (g exp t)
    (to-show g exp t)
    ((!-generator cut) g exp t)))
```

And, now we can test it with `infer-type` as before.

6.6.2 Type inhabitation

We are going to do an experiment and in order to get the results we want, we need to respecify the order of the relations. Thus we redefine `!-`.

```
(define !-
  (extend-relation (a1 a2 a3)
    var-rel int-rel bool-rel zero?-rel sub1-rel +-rel
    if-rel lambda-rel app-rel fix-rel polylet-rel))
```

Here are four, perhaps unexpected, examples.

```
> (let ([res (exists (g ?)
  (solution (!- g ? '(--> int int)))]])
  '(!- ,(car res) ,(unparse (cadr res)) ,(caddr res)))

(!- (non-generic v.0 (--> int int) g.0)
  v.0
  (--> int int))

> (let ([res (exists (g la f b)
  (solution
    (!- g '(,la (,f) ,b) '(--> int int)))]])
  '(!- ,(car res) ,(unparse (cadr res)) ,(caddr res)))

(!- g.0 (lambda (v.0) v.0) (--> int int))

> (let ([res (exists (g h r q z y t)
  (solution
    (!- g '(,h ,r (,q ,z ,y) t)))]])
  '(!- ,(car res) ,(unparse (cadr res)) ,(caddr res)))

(!- (non-generic v.0 int g.0) (+ v.0 (+ v.0 v.0)) int)

> (let ([res (exists (g h r q z y t u v)
  (solution
    (!- g '(,h ,r (,q ,z ,y) '(,t ,u ,v)))]])
  '(!- ,(car res) ,(unparse (cadr res)) ,(caddr res)))

(!- g.0 (lambda (v.0) (+ v.0 v.0)) (--> int int))
```

The first example attempts to find an expression whose type is `(--> int int)`, but instead finds a type environment that binds that type to the variable `v.0`, and then the expression is trivially `v.0`. The second example produces an expression given the type. This is answering the question, “What expression *inhabits* that type?” In our case, the identity function inhabits that type. But, to make these first two examples work, we had to place `var-rel` first in the definition of `!-`, above. In the third example, it infers that `t` must be of `int` type. Then since there is only one binary operation that returns an `int` (i.e., `+`), it determines `q` and `h`. Next, we can infer that `r`, `z`, and `y` must be of `int` type, and what is easier than making them all the same variable and placing it in the initial type environment. The last example only differs in the shape of the resultant type. Here it assumes that since the type contains three parts, it must be an arrow type. That means that `h` must be the symbol `lambda`. Once again the only binary operator is `+` making `z` and `y` be of `int` type.

6.7 Prolog’s name as a relation

Consider treating invertible binary operators as three-place relations. The function `invertible-binary-function->ternary-relation` below expects that at most one of the three arguments is a variable and solves the problem by determining which of the three variables is uninstantiated.

```
(define invertible-binary-function->ternary-relation
  (lambda (op inverted-op)
    (extend-relation (a1 a2 a3)
      (relation _ (x y z)
        (to-show x y z)
        (all (fails (instantiated z)) (fun-call op z x y)))
      (relation _ (x y z)
        (to-show x y z)
        (all (fails (instantiated y)) (fun-call inverted-op y z x)))
      (relation _ (x y z)
        (to-show x y z)
        (all (fails (instantiated x)) (fun-call inverted-op x z y)))
      (relation _ (x y z)
        (to-show x y z)
        (fun-call op z x y))))))

(define ++ (invertible-binary-function->ternary-relation + -))
(define -- (invertible-binary-function->ternary-relation - +))
(define ** (invertible-binary-function->ternary-relation * /))
(define // (invertible-binary-function->ternary-relation / *))

> (exists (x) (solution (++ x 16.0 8)))
(-8.0 16.0 8)
```

```
> (exists (x) (solution (** 10 x 50)))
```

```
(10 5 50)
```

```
> (exists (x) (solution (-- 10 7 x)))
```

```
(10 7 3)
```

And we can do something similar with invertible unary functions.

```
(define invertible-unary-function->binary-relation
  (lambda (op inverted-op)
    (extend-relation (a1 a2)
      (relation _ (x y)
        (to-show x y)
        (all (fails (instantiated y)) (fun-call op y x)))
      (relation _ (x y)
        (to-show x y)
        (all (fails (instantiated x)) (fun-call inverted-op x y)))
      (trace-relation third _ (x y)
        (to-show x y)
        (begin (pretty-print "Third rule") (fun-call op y x))))))

(define symbol->lnum
  (lambda (sym)
    (map char->integer (string->list (symbol->string sym)))))

(define lnum->symbol
  (lambda (lnums)
    (string->symbol (list->string (map integer->char lnnums)))))

(define name
  (invertible-unary-function->binary-relation symbol->lnum lnum->symbol))

> (exists (x) (solution (name 'sleep x)))

(sleep (115 108 101 101 112))

> (exists (x) (solution (name x '(115 108 101 101 112))))

(sleep (115 108 101 101 112))
```

In the first example, we return the `char->integer` of each character in `sleep`. In the second, given a list of integers, presumably derived from `char->integer`, it returns the symbol made from those integers. Thus, we have the Prolog relation

name. For our purposes, which is an embedding in Scheme, it is probably unnecessary, but it is interesting, nonetheless, that we can define these two relation-generating Scheme functions. There are more unary functions that can be so treated, like `symbol->string` and `string->symbol`, but we leave their inclusion to the programmer who might need them.

7 Final thoughts

In a relation call such as `(exists (x) (foo x 'a y))` we have three different kinds of values. The symbol `'a` is a raw value, the lexical variables, `foo` and `y`, become values as part of the call, and the logic variable `x` becomes a value (or is shared with another logic variable) when its argument unifies with another value. Having the call allow for their intermingling clarifies why we need to manage their scopes. Because the call is awaiting other arguments, we do not need to think about this in terms of a finished call that returns a value until these additional arguments are absorbed by the result of the relation call.

We have given up two main features of logic programming? First, we do not have meta-level operations that allow for construction of relations from existing relations. This may be retrieved by building the relations as data structures and using `eval` to construct the actual relations when they are needed. This solution leaves much to be desired and the way Prolog handles this is cleaner, especially since `eval` should only be used in the rarest of circumstances. It may be possible, however, with higher-order capabilities to get around most of these needs. Second, and more importantly, we have abandoned the database capability associated with logic programming. By that, we mean the ability for relations to be treated as a global monolithic relation. To circumvent this shortcoming, one can write a driver that knows some subset (possibly all) of the relations and their arities and whenever a relation of a particular arity is invoked, it searches through all the relations of that arity. This would be easy to set up with `extend-relation` and an association list that associates an arity with an extended relation of that arity. We can further partition this association list by replacing the arity with a type signature. But, this approach would have to re-address the cut. Since, this browsing capability is not required by the problem domains we have in mind, probably it is best that it be developed on an as needed basis. From the start we have been developing a tool that would allow for easy implementation of language-related programs such as type inferencers, interpreters, and compilers. These clearly do not need a global database.

One disadvantage of this implementation is that there is no obvious place to change the depth-first search strategy to one that supports breadth-first search. In its place we have made relations lexically-scoped, first-class, and extensible, which seems to be a fair tradeoff. One approach that might work for getting back other search strategies works like this. The formals to `lambda@` could be re-ordered to have the last two always be `cutk` and `fk`. This would mean that there could be just one argument instead of four. That one argument would be a stream of say, *packages*, where a package contains everything but `fk`. Then, each package, which

now contains `sk subst`, and `cutk` could, in turn, be treated as a stream of *packs*, where a pack contains everything but `cutk`. A pack could be implemented with `cons`. Now that these four variables can be treated like a stream of streams, we can think about different ways of combining streams to perhaps yield different and interesting search strategies. But, this is left as an exercise for the reader. Be wary that not every use of `@` physically takes four arguments, so there is a little bookkeeping to get this to work. The main thing to do is change the `lambda@` into `lambda` first. Once that is running, changing the argument order and turning the data into streams is simple.

Our goal has been to present the ideas of logic programming without using a lot of special features of Scheme and without losing the feel of programming in Scheme. Now, we can say that the basic unsullied logic system interface contains five operators: `relation`, `extend-relation`, `all`, `all!`, and `solve`.

8 Acknowledgments

This paper would not have been possible without the earlier work on implementing logic systems with Anurag Mendhekar. The work with Anurag led to Jon Rossie's use of our logic system in the development of his dissertation's results. His utilization of the tool helped us understand how we had to weave things together. But, it wasn't until work with Mitch Wand and Chris Haynes in *Essentials of Programming Languages, Second Edition* on the material on unification, substitutions, and logic programming that it seemed there might be a chance for a "Poor Man's" solution. Steve Ganz's dissertation also needed an inference system. Over the years, Steve began to see how we could make the seamlessness of Scheme possible by showing how to partition the monolithic relations into functions. Such a function then represents a relation that can be invoked as a function. Next we noticed that the sullied operators could also be written as functions with the same interface, thus removing a dispatch. Once the dispatch was removed, it was easy to remove the lone remaining search down the antecedents. Discovering that a non-recursive unifier was possible came much later. The first implementation and discussion of polymorphic `let` is the work of Jeremiah Willcock. Treating the consequent of a relation as an antecedent has been inspired by Seres and Spivey's paper. Several fruitful discussions with Venkatesh Choppella led to more thought about the types in the code of this logic system. We are grateful to Oscar Waddell for implementing `expand-only`, the partial macro expander. Regretably, it cannot be made fully general, however, it does meet our needs.