Kanren: A Logic System with First-Class Relations (Draft: May 10, 2004)

Daniel P. Friedman
Computer Science Department, Indiana University
Bloomington, IN 47405, USA

Oleg Kiselyov
Fleet Numerical Meteorology and Oceanography Center,
Monterey, CA 93943, USA

1 Introduction

We present an implementation of an embedding in R^5 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. 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 several 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. 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 relation contraction and extension operations. These operations take 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.

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 those that view the substitution at any point in a program, 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 generalizaton of Prolog's `name`. Then, we show how to prove a theorem about a famous program, `mirror`. We conclude with some perspective on why this approach works.

2 Preliminaries

There are four categories of operations we need. The first category contains operators for creating lexically-scoped variables (`let-lv`), currying (`lambda@` and `@`), a special `let*` called `let*-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 third section introduces some useful functions and macros that return substitutions that distinguish one variable from another even though they may have been built with the same name. The final category has an operation for unifying two terms: `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 `logical-variable`, which takes a symbol as its argument. Here is an example of its use.

```
> (let ([x (logical-variable 'x)] [xx (logical-variable 'x)])
    (list (logical-variable-id x)
          (logical-variable-id xx)
          (var? x)
          (eq? x xx)))
(x x #t #f)
```

There is a procedure `logical-variable-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 `logical-variable` makes a new one. Although the constructor, accessor, and predcate should be defined using records, a 2-elememt vector suffices, since we do not support vectors as kinds of terms. For purists, instead of relying on the ability to distinguish two `vector` with `eq?`, one could associate a unique timestamp with each variable.)

We introduce a macro, `let-lv`, 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 `let-lv`.

```
(define-syntax exists
  (syntax-rules ()
    [(_ () body) body]
    [(_ () body0 body1 ...) (begin body0 body1 ...)]
    [(_ (id ...) body)
     (let ([id (logical-variable 'id)] ...)
       body ...)
    [(_ (id ...) body0 body1 ...)
     (let ([id (logical-variable '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 (logical-variable 'x)] [y (logical-variable 'y)] [z (logical-variable '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))
 (logical-variable 'x)
 (logical-variable 'y)
 (logical-variable 'z))

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

(let ([x (logical-variable 'x)]
      [y (logical-variable 'y)]
      [z (logical-variable 'z)])
  (list x 1 y 2 z 3))
```

The macro `exists` above has four clauses, but only the last one matters. The others are there to improve the readability of the output when `expand-only` is used. This may seem, on the surface, rather foolish, since we know that the full `expand` will be invoked, but we do this to improve the readabilty of the programs expanded by `expand-only`. When there is doubt about how a program works, it is often advantageous to expand a relation using `expand-only`. There are places where we do not use this approach, primarily because the macros, themselves, get too complicated. We restrict this approach to the inclusion of some simple superfluous clauses as in `exists`.

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 (logical-variable 'x)]
      [y (logical-variable 'y)]
      [z (logical-variable '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]

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] This is not quite true, since we concretize substitutions, which are lists, so we need our `subst-in` to be aware of pairs, but this is a very special case, which might vanish during the writing of this draft.

2.1.2 A special macro for lexical scope

When we want to bind lexically like `let*`, but only if the value being bound is non-false, then we use a macro `let*-and`. If at any time the value being bound is false, then an expression is invoked whose value is the value of the entire expression.

```
(define-syntax let*-and
  (syntax-rules ()
    [(_ false-exp () body0 body1 ...) (begin body0 body1 ...)]
    [(_ false-exp ([var0 exp0] [var1 exp1] ...) body0 body1 ...)
     (let ([var0 exp0])
       (if var0
         (let*-and false-exp ([var1 exp1] ...) body0 body1 ...)
         false-exp))]))
```

Here is a simple but contrived example, which relies on the fact that `memv` returns a list whose `car` is the matched item. Otherwise, `memv` returns false.

```
> (let ([ls '(c a b a b b b a b c)])
    (let*-and 'too-short
      ([ls (memv 'a (cdr ls))]
       [ls (memv 'a (cdr ls))]
       [ls (memv 'a (cdr ls))])
      ls))
(a b c)
```

In this example, if one of the `a`s are missing from the original list, then the value of the expression is the symbol `too-short`.

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 need to naively add a commitment to a substitution, so we must be certain that the commitment is real. This definition is naive, because it assumes that some other function has verified that `var` is not a variable of any commitment in `subst`.

```
(define cons-if-real-commitment
  (lambda (var term subst)
    (cond
      [(eq? 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. [3]

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

(define compose-subst/own-survivors
  (lambda (base refining survivors)
    (let refine ([b* base])
      (if (null? b*) survivors
          (cons-if-real-commitment
            (commitment->var (car b*))
            (subst-in (commitment->term (car b*)) refining)
            (refine (cdr b*)))))))

(define compose-subst
  (lambda (base refining)
    (compose-subst/own-survivors base refining
      (let survive ([r* refining])
        (cond
          [(null? r*) '()]
          [(assq (commitment->var (car r*)) base) (survive (cdr r*))]
          [else (cons (car r*) (survive (cdr r*)))])))))
```

When does `compose-subst` behave like `append`? There are two aspects of the code to consider. First, the `assq` 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 b*))`. One way to do this would be to make sure that the variables of `(commitment->term (car b*))` 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. 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.

`Exercise 1:` The definition of `compose-subst` searches the base substitution multiple times. It is possible to rewrite the code so that each base commitment is looked at once, but then the refining substitution is searched multiple times. Implement that variant. ◇

The procedure `subst-in` below translates everything to itself except variables. If the variable is committed, its associated term is returned, otherwise the variable

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

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
         [(assq 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. ◇

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. ◇

2.3 Concretizing terms

We introduce a simple function that returns an eye-pleasing term. In each term we have variables and non-variables. We need to make sure that even if two variables in the term were constructed from the same name, they must appear as different symbols. Thus, we introduce the procedure `artificial-id`. 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 (t-id num)
    (string->symbol
      (string-append
        (symbol->string t-id) "." (number->string num)))))

> (exists (who) (artificial-id (logical-variable-id who) 5))
who.5
```

We associate a unique artificial identifier with each variable. We find the variables in a term using `vars-of`, below.

```
(define vars-of
  (lambda (term)
    (let loop ([term term] [fv '()])
      (cond
        [(var? term) (if (memq term fv) fv (cons term fv))]
        [(pair? term) (loop (cdr term) (loop (car term) fv))]
        [else fv]))))

> (exists (x y z) (vars-of `(,x 1 ,y ,x 2 ,z 3 ,x)))
(#[logical-variable z]
 #[logical-variable y]
 #[logical-variable x])
```

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 in `concretize` below is to use an environment that binds each symbol to the number of the most recently constructed artificial identifier. Then consult the environment when we find a new variable. We initially reverse the list of variables so that the order of their appearance in the term is as we would expect it.

```
(define concretize
  (lambda (t)
    (subst-in t
      (let loop ([fv (reverse (vars-of t))] [env '()])
(cond
  [(null? fv) empty-subst]
  [else (let ([id (logical-variable-id (car fv))])
  (let ([num (let*-and 0 ([pr (assq id env)]) (+ (cdr pr) 1))])
    (cons (commitment (car fv) (artificial-id id num))
      (loop (cdr fv) (cons (cons id num) env)))))]))))))
```

Here is a somewhat surprising example.

```
> (concretize
    (compose-subst
```

```
      (exists (x) (unit-subst x 3))
      (exists (x y) (unit-subst x y))))
([x.0 . 3] [x.1 . y.0])
```

If the two `x`s were the same, then we would have only the first commitment. But, since each `x` is a different variable, we have two commitments in the concretized substitution.

We reiterate what we stated in the beginning of this section. This operator is used strictly for building values that work as Scheme values. Although it is true that logic variables are Scheme values, when they are displayed, we cannot tell one logic variable that was built with the name `x` from another logic variable whose name is also built from `'x`. Thus when we see answers, we must concretize the variables.

2.4 Unification

We introduce the unifier `unify`, which given two terms and a substitution, 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 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 `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 substitution that it started with. That is why we do not choose false to represent the empty substitution. Returning a 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 composed with the 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)
    (let ([t (subst-in t subst)] [u (subst-in u subst)])
      (cond
        [(eqv? t u) subst]
        [(and (string? t) (string? u) (string=? t u)) subst]
        [(eq? t _) subst]
        [(eq? u _) subst]
        [(var? t) (compose-subst subst (unit-subst t u))]
        [(var? u) (compose-subst subst (unit-subst u t))]
        [else #f]))))
```

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.

2.5 Distinguishing Boolean values from success and failure

Consider the simplied[4] definition of `binary-or`, below.

```
(define-syntax binary-or
  (syntax-rules ()
    [(or expr1 expre2) (if expr1 #t (if expr2 #t #f))]))
```

If the first argument *is non-false*, the result *is true*, otherwise if the second argument *is non-false*, the result *is true*. Otherwise the result *is false*.

We define `binary-any` to make a point about the difference between *success* and *failure* and Boolean values. If the first argument *succeeds*, the result *succeeds*, otherwise, if the second argument *succeeds*, the result *succeeds*. Otherwise, the result is *fails*.

```
(define-syntax binary-any
  (syntax-rules ()
    [(_ ant1 ant2)
     (lambda@ (sk fk subst)
       (@ ant1 sk
         (lambda ()
           (@ ant2 sk fk subst))
         subst))]))
```

What distinguishes `binary-any` from `or` is that it awaits three arguments: a success continuation, a failure continuation, and a substitution. Functions that take these three arguments are called *antecedents*. So, `binary-any` returns an antecedent, that *when* it gets its arguments, invokes its first argument, which is an antecedent, in

[4] Of course, this is not exactly how `or` works, since it actually returns the first non-false value. Furthermore, it takes any number of arguments, but we have chosen this simplified version to point out some of the similiarities.

a failure continuation, which when invoked, invokes its second argument, which is also an antecedent, in the original failure continuation.

Below are a couple of antecedent generators, that when they get their arguments, they return an antecedent. In the case of `predicate`, it transforms a boolean expression into an antecedent, that when it gets its arguments (the state) treats true as success and false as failure. The second one, `==`, succeeds with the result of unifying its two arguments, otherwise it fails.

```
(define-syntax predicate
  (syntax-rules ()
    [(_ scheme-expression) (if scheme-expression succeed fail)]))

(define succeed
  (lambda (sk)
    sk))

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

(define-syntax ==
  (syntax-rules ()
    [(_ t u)
     (lambda@ (sk fk subst)
       (let*-and (fk) ([subst (unify t u subst)])
         (@ sk fk subst)))]))
```

The macro `solve-it` simply looks up the association of each variable in the substitution, `subst`, provided that it is not false. In the invocation of `(query (test1 x))`, below, the pair returned is `(x.0 x.0)`. The first `x.0` is the concretized name of the variable that we are looking up in the substitution and the second one indicates that there was no association in the substitution, since `(subst-in x empty-subst)` is always `x` and through concretization yields `x.0`.

```
(define-syntax query
  (syntax-rules ()
    [(_ (redo-k subst id ...) ant scheme-expression0 scheme-expression1 ...)
     (let-lv (id ...)
       (@ ant
         (lambda@ (redo-k subst) scheme-expression0 scheme-expression1 ...)
         (lambda () '())
         empty-subst))]))
```

Consider the following three relations (`test1`, `test2`, and `test3`).

```
> (define test1
    (lambda (x)
      (binary-any
```

```
        (predicate (< 4 5))
        (== x (< 6 7)))))

> (query (redo-k subst x y) (test1 x) (concretize subst))
()

> (define test2
    (lambda (x)
      (binary-any
        (predicate (< 5 4))
        (== x (< 6 7)))))

> (query (redo-k subst x) (test2 x) (concretize subst))
((x.0 . #t))

> (define test3
    (lambda (x y)
      (binary-any
        (== x (< 5 4))
        (== y (< 6 7)))))

> (query (redo-k subst x y) (test3 x y) (concretize subst))
((x.0 . #f))
```

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? `==`s like those above always succeed when its argument is uninstantiated. In this case, `x` becomes false. That is a form of success! That is why `y` remains uncommitted.

The definition of `binary-any` is generalized later. `binary-any` becomes `any`, a macro that takes any number of arguments.

3 The unsullied logic system

The logic system has a handful operators: `relation` (`fact` is derived from `relation`.), which expands into a `lambda` expression. We have several macros for handling sequences of antecedents: `all`, its dual `any`, and its deterministic counterpart `all!`, and its deterministic and unforgiving counterpart `all!!`. In `relation`, we rely on `if-only` and `if-all!`. `extend-relation`, which gives us the ability to form a new relation from other relations, is derived from `any`. To enter the system, we use `query`, which is defined above to define `solve`. Also there is`intersect-relation`, which takes a sequence of relations and yields their intersection.

Here are the types we use in this system:

```
Fk = () -> Ans
Ans = Nil + [Subst, Fk]
```

```
          Sk = Fk -> Subst -> Ans
  Antecedent = Sk -> Sk
    Relation = Term* -> Antecedent
```

`Ans` is a stream of `Subst`s, 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-syntax all
  (syntax-rules ()
    [(_) (lambda@ (sk) sk)]
    [(_ ant) ant]
    [(_ ant0 ant1 ...)
     (lambda@ (sk)
       (splice-in-ants/all sk ant0 ant1 ...))]))

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

(define succeed (all))
```

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` or `(lambda (sk) sk)` as described above). 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 ant1 ant2 ant3 ant4 ant5))

(lambda (sk) (ant1 (ant2 (ant3 (ant4 (ant5 sk))))))
```

It looks like the first thing that happens is that `ant5` is applied, but although that is the first thing that happens, it is `ant1` 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 ant1 ant2))

(lambda (sk) (ant1 (ant2 sk)))
```

This is just function composition. We know that `(ant2 sk)` must return an `sk`, since that is what `ant1` is expecting. But suppose that, `ant1` is also expecting another argument, say `fk`. So, we can see this as `(lambda@ (sk fk) ((ant1 (ant2 sk)) fk))`. Now, eventually, if `ant1` decides to invoke the `(ant2 sk)`, which was passed in, it will do it on a different `fk`, but it should be clear that `ant1` gets to run first. These two `lambda` expressions are η-convertible, however, so we do not need to worry about the `fk`s. Moreover, there is an additional argument, `subst`, that must arrive before `ant1` 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.

Next, we consider `all!`, the more deterministic variant of `all`.

```
(define-syntax promise-one-answer
  (syntax-rules ()
    ((_ ant) ant)))

(define-syntax all!
  (syntax-rules (promise-one-answer)
    [(_) (all)]
    [(_ (promise-one-answer ant)) (promise-one-answer ant)]
    [(_ ant0 ant1 ...)
     (promise-one-answer
       (lambda@ (sk fk)
         (@
           (splice-in-ants/all (lambda@ (fk-ign) (@ sk fk)) ant0 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! ant1 ant2 ant3))

(promise-one-answer
  (lambda@ (sk fk)
    (@ (@ ant1
          (@ ant2
            (@ ant3
              (lambda@ (fk-ign)
                (@ sk fk)))))
       fk)))
```

As in the previous version, if all antecedents succeed, then the original success continuation `sk` is invoked. But, this time, however, once `ant3` succeeds, if failure backs into it, it invokes the original failure continuation, rather than retry `ant3`. That is, the `all!` expression fails. This is what is meant by "promise one answer".

Finally, we introduce the fully deterministic, unforgiving `all!!`.

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

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

Again, it should be easier to follow a specific instance of its use.

```
(promise-one-answer
  (lambda@ (sk fk)
    (@ ant1
      (lambda (fk-ign)
        (@ ant2
          (lambda (fk-ign)
            (@ ant3
              (lambda (fk-ign)
                (@ sk fk))
              fk))
          fk))
      fk)))
```

The macro `all!!` generates a procedure that expects a success and a failure continuation (and one more argument, which is 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 fully deterministic, unforgiving `all`. One slip up, and you fail, whereas, with `all` and `all!`, you get to backup through the antecedents and retry them until success, but we are getting a bit ahead of our story.

Even with all this technology to explain the idea of using `all`, `all!`, and `all!!`, it is still possible that some confusion remains. So, we shall look at three simple examples that should clarify how these work together.

The best way to see how this works is to compare three similar expressions. Let us assume that `a`, `b`, and `c` are antecedents. Then consider these three antecedents.

```
1.  (all a b fail)
2.  (all! a b fail)
3.  (all!! a b fail)
```

In the absence of context, the first two antecedents have identical behavior. If `a` and `b` both succeed, the `fail` antecedent fails and causes `b` to be tried again. If `b` fails, it retries `a`. In the third case, however, if `a` and `b` succeed, the failure antecedent fails and the *entire* expression fails. No further backtracking is attempted. Thus, `all!!` is different from the other two in that if any of the antecedents fail at any time, the whole expression fails. In fact, `all!!` can be rewritten in terms of `all!`: `(all!! a b)` = `(all! (all! a) (all! b))`.

Now consider these three new antecedents:

```
1.  (all a (all b c) fail)
2.  (all a (all! b c) fail)
3.  (all a (all!! b c) fail)
```

When context is placed around these antecedents, different behaviors can be observed. The first antecedent is equivalent to `(all a b c fail)`. When the `fail` antecedent fails, `c` is then retried. For the second and third, however, when `fail` fails, the inner antecedent is skipped over and `a` is retried. We can also describe `all!` in terms of `all` and `all!!`. `(all! a b)` = `(all!! (all a b))`. The `all` allows backtracking within the `a` and `b`, but when failure backs into the `all!!` antecedent, it forces failure to the previous antecedent if there is one.

In conclusion, `all` and `all!` are similar in that within them, backtracking occurs, but `all!!` and `all!` are similar in that when failure backs into them, they fail. Of course, they are all similiar, since their arguments must succeed for them to succeed.

We have one more unusal control operator, `if-only`.

```
(define-syntax if-only
  (syntax-rules ()
    [(_ condition then)
     (lambda@ (sk fk)
       (@ condition
         (lambda@ (fk-ign) (@ then sk fk))
         fk))]
    [(_ condition then else)
     (lambda@ (sk fk subst)
       (@ condition
         (lambda@ (fk-ign) (@ then sk fk))
         (lambda () (@ else sk fk subst))
         subst))]))
```

Again, the complexity of this macro should encourage us to see how some simple examples expand.

```
> (expand-only '(if-only) '(if-only ant then))

(lambda@ (sk fk)
  (@ ant (lambda@ (fk-ign) (@ then sk fk)) fk))
```

Here we see that `ant` is run in a success continuation that runs the antecedent `then` (with the substitution generated by `ant`, but if `then` fails, `ant` is not retried. If `ant` fails, the entire expression fails.

```
> (expand-only '(if-only) '(if-only ant then else))

(lambda@ (sk fk subst)
  (@ ant
    (lambda@ (fk-ign) (@ then sk fk))
    (lambda () (@ else sk fk subst))
    subst))
```

The antecedent `ant` is run in the same success continuation as before, but now, if `ant` fails, its tries `else` using the original substitution.

Finally, we come to `relation` below, which is like a special `lambda`. There are many clauses to consider, so let's not spend any time worrying about the macro, itself, but let's spend time on what it does in particular interesting cases.

```
(define-syntax relation
  (syntax-rules (to-show head-let once _)
    [(_ (head-let head-term ...) ant)
     (relation-head-let (head-term ...) ant)]
    [(_ (head-let head-term ...)) ; not particularly useful without body
     (relation-head-let (head-term ...))]
    [(_ () (to-show term ...) ant) ; pattern with no vars _is_ linear
     (relation-head-let (`,term ...) ant)]
    [(_ () (to-show term ...)) ; the same without body: not too useful
     (relation-head-let (`,term ...))]
    [(_ (ex-id ...) (to-show term ...) ant)  ; body present
     (relation "a" () () (ex-id ...) (term ...) ant)]
    [(_ (ex-id ...) (to-show term ...))      ; no body
     (relation "a" () () (ex-id ...) (term ...))]
    ; process the list of variables and handle annotations
    [(_ "a" vars once-vars ((once id) . ids) terms . ant)
     (relation "a" vars (id . once-vars) ids terms . ant)]
    [(_ "a" vars once-vars (id . ids) terms . ant)
     (relation "a" (id . vars) once-vars ids terms . ant)]
    [(_ "a" vars once-vars () terms . ant)
     (relation "g" vars once-vars () () () (subst) terms . ant)]
    ; generating temp names for each term in the head
    ; don't generate if the term is a variable that occurs in
    ; once-vars
    ; For _ variables in the pattern, generate unique names for the lambda
    ; parameters, and forget them
    ; also, note and keep track of the first occurrence of a term
    ; that is just a var (bare-var)
    [(_ "g" vars once-vars (gs ...) gunis bvars bvar-cl (_ . terms) . ant)
     (relation "g" vars once-vars (gs ... anon) gunis
       bvars bvar-cl terms . ant)]
    [(_ "g" vars once-vars (gs ...) gunis bvars (subst . cls)
           (term . terms) . ant)
     (id-memv?? term once-vars
       ; success continuation: term is a once-var
       (relation "g" vars once-vars (gs ... term) gunis bvars (subst . cls)
terms . ant)
       ; failure continuation: term is not a once-var
       (id-memv?? term vars
; term is a bare var
(id-memv?? term bvars
```

```
   ; term is a bare var, but we have seen it already: general case
   (relation "g" vars once-vars  (gs ... g) ((g . term) . gunis)
     bvars (subst . cls) terms . ant)
   ; term is a bare var, and we have not seen it
   (relation "g" vars once-vars (gs ... g) gunis
     (term . bvars)
     (subst
       (subst (unify-free/any term g subst))
       (fast-path? (and (pair? subst)
      (eq? term (commitment->var (car subst)))))
       (term (if fast-path? (commitment->term (car subst)) term))
       (subst (if fast-path? (cdr subst) subst))
       . cls)
     terms . ant))
 ; term is not a bare var
 (relation "g" vars once-vars  (gs ... g) ((g . term) . gunis)
   bvars (subst . cls) terms . ant)))]
    [(_ "g" vars once-vars gs gunis bvars bvar-cl () . ant)
     (relation "f" vars once-vars gs gunis bvar-cl . ant)]

    ; Final: writing the code
    [(_ "f" vars () () () (subst) ant)   ; no arguments (no head-tests)
      (lambda ()
(exists vars ant))]
                                     ; no tests but pure binding
    [(_ "f" (ex-id ...) once-vars (g ...) () (subst) ant)
     (lambda (g ...)
       (exists (ex-id ...) ant))]
    ; the most general
    [(_ "f" (ex-id ...) once-vars (g ...) ((gv . term) ...)
       (subst let*-clause ...) ant ...)
     (lambda (g ...)
       (exists (ex-id ...)
 (lambda@ (sk fk subst)
   (let* (let*-clause ...)
     (let*-and (fk) ((subst (unify gv term subst)) ...)
       (@ ant ... sk fk subst))))))]))
```

```
(define-syntax id-memv??
  (syntax-rules ()
    [(id-memv?? form (id ...) kt kf)
     (let-syntax ([test
                   (syntax-rules (id ...)
                     ((test id _kt _kf) _kt) ...
                     ((test otherwise _kt _kf) _kf))])
       (test form kt kf))]))


(define-syntax relation-head-let
  (syntax-rules ()
    [(_ (head-term ...) . ants)
     (relation-head-let "g" () (head-term ...) (head-term ...) . ants)]
    ; generate names of formal parameters
    [(_ "g" (genvar ...) ((head-term . tail-term) . ht-rest)
       head-terms . ants)
     (relation-head-let "g" (genvar ... g) ht-rest head-terms . ants)]
    [(_ "g" (genvar ...) (head-term . ht-rest) head-terms . ants)
     (relation-head-let "g" (genvar ... head-term) ht-rest head-terms . ants)]
    [(_ "g" genvars  () head-terms . ants)
     (relation-head-let "d" () () genvars head-terms genvars . ants)]
    ; partition head-terms into vars and others
    [(_ "d" vars others (gv . gv-rest) ((hth . htt) . ht-rest) gvs . ants)
     (relation-head-let "d" vars ((gv (hth . htt)) . others)
       gv-rest ht-rest gvs . ants)]
    [(_ "d" vars others (gv . gv-rest) (htv . ht-rest) gvs . ants)
     (relation-head-let "d" (htv . vars) others
       gv-rest ht-rest gvs . ants)]
    [(_ "d" vars others () () gvs . ants)
     (relation-head-let "f" vars others gvs . ants)]

    ; final generation
    [(_ "f" vars ((gv term) ...) gvs) ; no body
     (lambda gvs                                    ; don't bother bind vars
       (lambda@ (sk fk subst)
 (let*-and (fk) ([subst (unify gv term subst)] ...)
   (@ sk fk subst))))]

    [(_ "f" (var0 ...) ((gvo term) ...) gvs ant)
     (lambda gvs
       (lambda@ (sk fk subst) ; first unify the constants
 (let*-and (fk) ([subst (unify gvo term subst)] ...)
           (let ([var0 (if (eq? var0 _) (logical-variable '?) var0)] ...)
             (@ ant sk fk subst)))))]))
```

So, the first thing is to realize that in most cases the only rule that matters is the very last clause and how to get to it naturally. It should be painfully obvious that `"f"` appears as the first item to `relation`. The user, however, will never type it. It is used to guide the flow through the macro. How might we get to the last clause? Just one of the `"g"` clauses generates an `"f"` clause, so it too is necessary. The other `"g"` clause is necessary, but only to accumulate as many genererated variables as there are arguments in the `to-show`. Then, there are two clauses that rely on `to-show`. Everything else, is related to optimizing special cases.

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

```
(define father
  (relation ()
    (to-show 'rob 'sal)))
```

Let's focus on the single fact that says that "Rob is the father of Sal." But, what it really says is that there is a relation, that connects "Rob to Sal," which we are calling "The father relation." And when do we know that Rob is the father of Sal? When we have shown that all the antecedent expressions below the `to-show` keyword expression hold. But, since there are none, it holds vacuously. Therefore, Rob is always the father of Sal. But, if we have another relation, *older_than*, we can have the fact that "Rob 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 want to do the same thing, we could also just `(define older-than father)`, which would say that "Rob is older than Sal." Of course, that should not come as a surprise, since Rob is Sal's father, but these two relations are completely disjoint! For example, we may be referring to four different people: two Robs 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 relation-head-let)
    '(define father
       (relation ()
         (to-show 'rob 'sal))))

(define father
  (lambda (g1 g2)
    (lambda@ (sk fk subst)
      (let*-and (fk) ([subst (unify g2 `,'sal subst)]
                      [subst (unify g1 `,'rob subst)])
        (|@| sk fk subst)))))
```

The relation `father` is represented as a procedure of two arguments, `g1` and `g2`.[5] It can figure that out from the number of operands in the `to-show` keyword expression. If we can unify the value of the variable `g2` with the value of `'sal` and the value of the variable `g1` with the value of `'rob`, then the resultant substitution is passed to a procedure that addresses the remaining antecedent, of which there is none, so it invokes the success continuation `sk` using the new substitution.

Whenever `father` is invoked, a dad (`g1`) and a child (`g2`) each get bound to a term. Then an antecedent is returned. This antecedent is waiting for a success continuation, a failure continuation, and a substitution. The failure continuation is invoked if the arguments fail to unify. This causes a return from the call to `father`, since if any unification fails, the entire invocation of `father` fails.

There is a relation between `relation` and antecedents. It is possible to turn any antededent into a zero-argument relation using `ant->relation` below.

```
(define-syntax ant->relation
  (syntax-rules ()
    [(_ ant) (relation () (to-show) ant)]))
```

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)))
```

Such a relation is often written using a horizontal line, $\dfrac{\texttt{(father dad child)}}{\texttt{(child-of-male child dad)}}$

```
(define expand-only*
  (lambda (names expression)
    (expand-only '(id-memv??)
      (expand-only names expression))))

> (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 (dad child)
      (lambda@ (sk fk subst)
        (let*  ([subst (unify-free/any dad g2 subst)]
```

[5] The actual expansion produces `g` and `g`. But, they are different variables.

```
            [fast-path?
              (and (pair? subst)
                   (eq? dad (commitment->var (car subst))))]
            [dad (if fast-path? (commitment->term (car subst)) dad)]
            [subst (if fast-path? (cdr subst) subst)]
            [subst (unify-free/any child g1 subst)]
            [fast-path?
              (and (pair? subst)
                   (eq? child (commitment->var (car subst))))]
            [child (if fast-path? (commitment->term (car subst)) child)]
            [subst (if fast-path? (cdr subst) subst)])
      (let*-and (fk) () (@ (father dad child) sk fk subst)))))))
```

This says that since we know that `dad` is free (not yet bound in the substitution), we can check to see if unify placed it as the first commitment. If so, we can bind `dad` lexically and shrink the substitution. The same thing holds for `child`. Thus when we run `(father dad child)` it merely looks up the variable for dad and child lexically and furthermore in this example, `subst` has not changed. That's a lot of improvement.

But, we can actually do better. We can see that in the `to-show`, `child` (and `dad`) occurs at most once. Furthermore, in the antecedent, `child` (and dad) occurs at most once. We say that those variables are *linear*. Therefore, we can annotate these occurrences in the list following the keyword `relation` like this

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

(define child-of-male
  (lambda (child dad)
    (father dad child)))
```

Thus our intuition is borne out most directly. To find out if a child is the offspring of a dad, see if the dad is a father of the child. Could that be simpler? No.

Thus, we can see, because of the special properties of `child-of-male` that the definition is as trivial as me might have expected. Because `id-memv??` uses a local macro, we need the additional call of `expand-only`. Later we will discover additional ways to utilize the optimizations that are built into `relation`, but for now, we should be content to know that things are not nearly as expensive they at first appear to be.

3.2 Testing `father` and `child-of-male`

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

```
> (query (redo-k subst) (father 'rob 'sal) (concretize subst))
()
```

If the goal succeeds, it returns the concretized substitution. If the goal fails, it returns the empty list. How are we going to deal with this bit of confusion? Instead of returning the substitution, we place the concretized substitution in a pair of parentheses.

```
> (query (redo-k subst) (father 'rob 'sal) (cons (concretize subst) '()))
(())
```

Now, we can distinguish them. Later, we see much more sophisticated ways of dealing with the Scheme expression (last) arguments to `query`.

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 argument in the original antecedent call and produce what we started with: `rob` and `sal`), but with the assurance that `rob` has been shown to be the father of `sal`.

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

```
> (query (redo-k subst x) (father 'rob x) (concretize subst))
((x.0 . sal))
```

Now we know that if the result is nonempy, we have succeeded and if not, we have failed, so we can go back to not worrying about whether the answer is in a list or not.

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 result of the query, is returned.

```
> (query (redo-k subst y x) (child-of-male y x) (concretize subst))
((x.0 . rob) (y.0 . sal))
```

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

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 Rob is also the father of Pat, what changes could be made to get both answers? First, we introduce `extend-relation` to allow for additional facts.

The operator `any` is the dual of `all`.

```
(define-syntax any
  (syntax-rules ()
    [(_) (lambda@ (sk fk subst) (fk))]
    [(_ ant) ant]
    [(_ ant ...)
     (lambda@ (sk fk subst)
       (splice-in-ants/any sk fk subst ant ...))]))

(define-syntax splice-in-ants/any
  (syntax-rules ()
    [(_ sk fk subst ant1) (@ ant1 sk fk subst)]
    [(_ sk fk subst ant1 ant2 ...)
     (@ ant1 sk (lambda ()
                  (splice-in-ants/any sk fk subst ant2 ...))
       subst)]))

(define fail (any))
```

The first two clauses are simple. The first, is the dual of `all`, since when there are no antecedents for `all`, we `succeed`, whereas with `any` when we have no antecedents, we `fail`. As before, just one antecedent, expands to that one antecedent. So, let's see what happens when we have more than one antecedent.

```
> (expand-only '(any splice-in-ants/any)
    '(any ant1 ant2))

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

and

```
> (expand-only '(any splice-in-ants/any)
    '(any ant1 ant2 ant3))

(lambda@ (sk fk subst)
  (@ ant1
     sk
     (lambda ()
       (@ ant2
          sk
          (lambda ()
            (@ ant3 sk fk subst))
          subst))
```

```
    subst))
```

Each failure continuation starts with the same success continuation and the same substitution.

Now, we are able to build a single relation from a sequence of relations.

```
(define-syntax define-rel-lifted-comb
  (syntax-rules ()
    [(_ rel-syntax-name ant-proc-or-syntax)
     (define-syntax rel-syntax-name
       (syntax-rules ()
         [(_ ids . rel-exps)
          (lift-ant-to-rel-aux ant-proc-or-syntax ids () . rel-exps)]))]))

(define-syntax lift-ant-to-rel-aux
  (syntax-rules ()
    [(_ ant-handler (id ...) ([g rel-var] ...))
     (let ([g rel-var] ...)
       (lambda (id ...)
         (ant-handler (g id ...) ...)))]
    [(_ ant-handler ids (let-pair ...) rel-exp0 rel-exp1 ...)
     (lift-ant-to-rel-aux ant-handler ids
       (let-pair ... [g rel-exp0]) rel-exp1 ...)]))

(define-rel-lifted-comb extend-relation any)
```

For each relation expression, we generate a `let` expression pair that holds the value of the relation expression. Then, a hand-bulit relation of the right number of arguments is built, with `any` wrapped around all the relation calls.

```
(define rob-pat
  (relation ()
    (to-show 'rob 'pat)))
```

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

```
(define father (extend-relation (a1 a2) father rob-pat))
```

The operator `extend-relation` takes a list of `lambda` variables, the length of which is the arity of the relation, and a sequence of relations. In the convention we use, the i in the last ai in the list is the arity (In `father`, it is 2.) of the relations. It tries to find a result in the first relation. If it succeeds, it returns a substitution along with a failure continuation, which is to try to find a result in the remaining relations. If it fails, then it invokes the just described failure continuation. The early evaluation of the relation expressions is 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.[6] Later, we present other variants of `extend-relation`.

Let's see how this new definition of `father` expands.

```
> (expand-only '(lift-ant-to-rel-aux extend-relation)
    '(extend-relation (a1 a2) father rob-pat))

(let ([g1 father] [g2 rob-pat])
  (lambda (a1 a2) (any (g1 a1 a2) (g2 a1 a2))))
```

What is critical here is that we get a kind of call-by-value behavior, since the relation arguments, `father` and `rob-rel`, to `extend-relation`, are evaluated (in this case just dereferenced) before the extended relation is returned. Also the arguments to `any` involve applications of the generated (`g`i) variables passing along the artificial (`a`i) variables.

A convenient abbreviation of a relation where there are no antecedents is decribed by the macro `fact`.

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

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

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

and test it like this.

```
> (query (redo-k subst x) (father 'rob x) (cons (subst-in x subst) (redo-k)))
(sal pat)
```

First, we add that Rob is also the father of Pat. Then, when we invoke `redo-k` (i.e., invoke the thunk bound to `redo-k` 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 used in `query`. Now, we see how easy it is to return a list of terms.

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.

```
> (define str (query (redo-k subst x) (father 'rob x) (cons (subst-in x subst) redo-k)))
```

[6] This is an argument for making `extend-relation` a procedure, but then it would be necessary to use `(lambda args ...)` and `apply`, which we believe should be avoided, when possible.

Now, because we are returning `redo-k` *instead* of invoking it, we are in charge of how many values will be in the final list, including a possible infinite number of values. The result of invoking query here can then be passed to a function `stream-prefix` which returns a list no larger than the specified prefix size.

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

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, a list of variable names, and an antecedent (here a relation call). It returns a list like the one above for Rob's children. Each variable is associated with the substituted for term of its commitment.

```
(define-syntax solve
  (syntax-rules ()
    [(_ n (var0 ...) ant)
     (if (<= n 0) '()
       (stream-prefix (- n 1)
         (query (redo-k subst var0 ...)
           ant
           (cons (concretize (list (cons var0 (subst-in var0 subst)) ...)) redo-k))))]))
```

```
> (solve 5 (x) (father 'rob x))
(([x.0 sal])
 ([x.0 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.

`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. ◇

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

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

```
> (solution (x) (father 'rob x))
([x.0 sal])
```

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 pass the appropriate Scheme expression to `query`.

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
  (extend-relation (a1 a2)
    (fact () 'jon 'sam)
    (fact () 'sam 'rob)
    (fact () 'rob 'sal)
    (fact () 'rob 'pat)))
```

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

```
> (expand-only* '(relation relation-head-let exists)
    '(define grandpa-sam
       (relation ((once grandchild))
         (to-show grandchild)
         (exists (parent)
           (all (father 'sam parent) (father parent grandchild))))))

(define grandpa-sam
  (lambda (grandchild)
    (exists (parent)
      (all (father 'sam parent) (father parent grandchild)))))

> (solve 6 (y) (grandpa-sam y))
(([y.0 sal])
 ([y.0 pat]))
```

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

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 `relation` (and `fact`), `any` (and `extend-relation`), `all`, `all!`, `all!!`, `exists`, and `solve`.

3.6 New observations

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 ((once grandchild))
      (to-show grandchild)
      (exists (parent)
        (all (father grandad parent) (father parent grandchild))))))

> (solve 6 (x) ((grandpa-maker 'sam) x))
(([x.0 sal])
 ([x.0 pat]))
```

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 ((once grandchild))
      (to-show grandchild)
      (exists (parent)
        (all (guide grandad parent) (guide parent grandchild))))))

> (solve 4 (x) ((grandpa-maker father 'sam) x))
(([x.0 sal])
 ([x.0 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`.

```
> (expand-only* '(relation relation-head-let)
    '(define grandpa
       (relation ((once grandad) (once grandchild))
```

```
        (to-show grandad grandchild)
        (exists (parent)
          (all (father grandad parent) (father parent grandchild))))))

(define grandpa
  (lambda (grandad grandchild)
    (exists (parent)
      (all (father grandad parent) (father parent grandchild)))))

> (solve 4 (x) (grandpa 'sam x))
(([x.0 sal])
 ([x.0 pat]))
```

We were literally able to make the `to-show` clause disappear. This works, because the point of `==` is to place a meaning of `grandad` and `grandchild` into the substitution, so it can be accessed, but this works as well, and there is no growth to the substitution until the `all` is processed. Of course, we must take responsibility of knowing when we meet the linearity constraint to do this, but it shows up in a surprising number of examples. If a variable appears more than once in either the `to-show` or the antecedent, you are obligated to remove the `once` annotation. Furthermore, the logic system semantics are undefined if you accidentally annotate a variable that violates the constraint.

This would be a good time to study a bit of `relation` and observe how these linear variable are added to the gs (formal parameter list) instead of a new unique variable. Also, you should observe that when a *once* variable is added to the gs, it is not placed in an `==`. So, if we had a non-linear variable, like this very contrived example.

```
(define grandpa
  (relation ((once grandad) x (once grandchild))
    (to-show grandad x grandchild x)
    (exists (parent)
      (all (father* grandad parent x) (father* parent x grandchild)))))
```

it would expand like this

```
(define grandpa
  (lambda (grandad g1 grandchild g2)
    (exists (x)
      (if-only (all!! (== g2 x) (== g1 x))
        (exists (parent)
          (all (father* grandad x parent) (father* parent x grandchild)))))))
```

Each occurrence of a non-linear variable got its own `==`.

Let's return to our growing world population. We make Rob an uncle of Sue and Sid, the children of his sister, Roz. First, we include Roz 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 Rob's sister's children can claim Sam as their grandfather.

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

(define roz-sue (fact () 'roz 'sue))
(define roz-sid (fact () 'roz 'sid))
(define mother (extend-relation (a1 a2) roz-sue roz-sid))

(define grandpa
  (extend-relation (a1 a2) grandpa
    (relation ((once grandad) (once grandchild))
      (to-show grandad grandchild)
      (exists (parent)
        (all (father grandad parent) (mother parent grandchild))))))

> (solve 10 (y) (grandpa 'sam y))
(([y.0 sal])
 ([y.0 pat])
 ([y.0 sue])
 ([y.0 sid]))
```

And we discover that Sam is, indeed, the grandfather of Sue and Sid.

3.7 Sharing

In the previous definition of `grandpa`, both relations use the same `to-show`, so why should we write them twice?

```
(define grandpa
  (extend-relation (a1 a2)
    (relation ((once grandad) (once grandchild))
      (to-show grandad grandchild)
      (exists (parent)
        (all (father grandad parent) (father parent grandchild))))
    (relation ((once grandad) (once grandchild))
      (to-show grandad grandchild)
      (exists (parent)
        (all (father grandad parent) (mother parent grandchild))))))
```

Instead, we can write it with `any` like this.

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

which expands to

```
(define grandps
  (lambda (grandad grandchild)
    (exists (parent)
      (any
        (all (father grandad parent) (father parent grandchild))
        (all (father grandad parent) (mother parent grandchild))))))
```

And we can distribute `any`, since in both cases we are invoking `(father grandad parent)`

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

which now expands to

```
(define grandpa
  (lambda (grandad grandchild)
    (exists (parent)
      (all (father grandad parent)
        (any (father parent grandchild) (mother parent grandchild))))))
```

Thus, we can see that our linearity constraint needs a little tweaking. A variable is linear in an antecedent if no part of the computation relies on its value more than once. So, even though `grandchild` appears twice in the `any` antecedent, once the first one fails, the asociation of `grandchild` vanishes, and then a second try occurs to bind it with a call to `mother`. So, once again, calling `grandpa` requires no unifications.

`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.

```
> (expand-only* '(relation relation-head-let)
    '(define grandpa-sam
       (let ([r (relation ((once child))
                  (to-show child)
                  (exists (parent)
                    (all (father 'sam parent) (father parent child))))])
         (relation ((once child))
           (to-show child)
           (r child)))))
```

```
(define grandpa-sam
  (let ([r (lambda (child)
             (exists (parent)
               (all (father 'sam parent) (father parent child))))])
    (lambda (child)
      (r child))))
```

Rewrite some variant of `grandpa` with that restriction. ◇

Finally, we can intersect a sequence of relations.

```
(define-rel-lifted-comb intersect-relation all)
```

Here is an example of its use.

```
(define parents-of-scouts
  (extend-relation (a1 a2)
    (fact () 'sam 'rob)
    (fact () 'roz 'sue)
    (fact () 'rob 'sal)))

(define parents-of-athletes
  (extend-relation (a1 a2)
    (fact () 'sam 'roz)
    (fact () 'roz 'sue)
    (fact () 'rob 'sal)))

(define busy-parents
  (intersect-relation (a1 a2)
    parents-of-scouts
    parents-of-athletes))

(define conscientious-parents
  (extend-relation (a1 a2)
    parents-of-scouts
    parents-of-athletes))

> (solve 5 (x) (exists (y) (busy-parents x y)))
(([x.0 roz])
 ([x.0 rob]))

> (solve 7 (x) (exists (y) (conscientious-parents x y)))
(([x.0 sam])
 ([x.0 roz])
 ([x.0 rob])
 ([x.0 sam])
 ([x.0 roz])
 ([x.0 rob]))
```

In these calls to `solve`, we created an antecedent with `exists` in order to avoid seeing the children.

`Exercise d` The second example introduces a new problem. Since we believe that both occurrences of Roz, Sam, and Rob refer to the same people, we probably would be as happy seeing their names only once. Revise the program so that only the *first* occurrences of these names appear. ◇

4 Antecededents that work as functions and predicates

```
(define-syntax project
  (syntax-rules ()
    [(_ (id ...) ant)
     (lambda@ (sk fk subst)
       (let ([id (nonvar! (subst-in id subst))] ...)
         (@ ant sk fk subst)))]))

(define nonvar!
  (lambda (t)
    (if (var? t)
      (error 'nonvar! "Logic variable ~s found after substituting." (concretize t))
      t)))

(define-syntax project/no-check
  (syntax-rules ()
    [(_ (id ...) ant)
     (lambda@ (sk fk subst)
       (let ([id (subst-in id subst)] ...)
         (@ ant sk fk subst)))]))
```

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, "r." Jon's child's name is Sam, so Jon is no longer considered a grandfather. But, Sam's child's name is Rob, so Rob's children are still someone's grandchildren.

```
> (expand-only* '(relation project predicate)
    '(define grandpa
       (relation ((once grandad) (once grandchild))
         (to-show grandad grandchild)
           (exists (parent)
             (all (father grandad parent)
               (project (parent)
                 (all
                   (predicate (starts-with-r? parent))
                   (father parent grandchild))))))))

(define grandpa
```

```
  (lambda (grandad grandchild)
    (exists (parent)
      (all
        (father grandad parent)
        (lambda@ (sk fk subst)
          (let ([parent (nonvar! (subst-in parent subst))])
            (@ (all
                 (if (starts-with-r? parent) succeed fail)
                 (father parent grandchild))
               sk
               fk
               subst)))))))

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

```
> (solve 10 (x y) (grandpa x y))
(([x.0 sam] [y.0 sal])
 ([x.0 sam] [y.0 pat]))
```

The expanded program binds the lexical variable `parent` to the value of the logical variable `parent`. Furthermore, before it binds it, it verifies that it has found something other than a logical variable. The assumption is that Scheme functions don't have any real use of logical variables. (In the case where they might, we have `project/no-check`, which doesn't check.). Once the lexical scope is determined, then running the expression `(starts-with-r? parent)` is the same as its Scheme counterpart, but it must succeed or fail instead of return true or false.

The Scheme expression is arbitrary. The Scheme expression acts like an antecedent, but can do anything that any Scheme expression can do, including capturing continuations, setting variables, writing to files, etc. The full panoply of options is available to the user.

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

5 Additional antecedents

In this section we introduce some additional operators, which rely on the underlying structure. In each instance, we are expanding the kinds of antecedents that are available. Among these are an operator `instantiated`, which determines if a variables has a non-variable as its binding, and three useful, miscellaneous operators: `fails`, `instantiated`, and `trace-vars`.

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

```
<A> =
     <relation call>
   | (exists (id ...) <A>)
   | fail
   | succeed
   | (all <A>*)
   | (all! <A>*)
   | (all!! <A>*)
   | (any <A>*)
   | (project (<var*>) <A>)
   | (predicate <Scheme Expression>)
   | <additional antecedent>
   | <or any Scheme Expression that evaluates to these,
       since these are all values.>

<additional antecedent> =
   | (fails <A>)
   | (succeeds <A>)
   | (instantiated <var>)
   | (trace-vars <var>*)
```

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

5.1 Interfacing Scheme functions

5.2 `fails`, `succeeds`, `instantiated`, and `view-subst`

Below are the functions `fails`, which fails when its goal succeeds and succeeds when its goal fails, and `succeeds`, which succeeds when its goal succeeds and fails when it fails. Basically, we hand build both the success and failure continuations.

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

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

And here is an example of `fails`.

```
(define grandpa
  (relation ((once grandad) (once grandchild))
    (to-show grandad grandchild)
    (exists (parent)
      (all
        (father grandad parent)
        (project (parent)
          (all
            (fails (predicate (starts-with-r? parent)))
            (father parent grandchild)))))))
```

```
> (solve 10 (x y) (grandpa x y))
(([x.0 jon] [y.0 rob])
 ([x.0 jon] [y.0 roz]))
```

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

```
(define instantiated
  (lambda (t)
    (project/no-check (t)
      (predicate (not (var? t))))))
```

At this point, `t` is either some variable or some other term.
We can use `once` on `grandchild`, even though it appears a second time. This is because we are not unifying against it.

```
(define-syntax trace-vars
  (syntax-rules ()
    [(_ title (var0 ...))
     (promise-one-answer
       (project/no-check (var0 ...)
         (predicate
   (for-each (lambda (name val)
               (cout title " " name ": " val nl))
             '(var0 ...) (concretize `(,var0 ...))))))]))

(define grandpa
  (relation ((once grandad) (once grandchild))
    (to-show grandad grandchild)
    (exists (parent)
      (all
        (trace-vars "Variables and values:" (grandad parent grandchild))
        (father grandad parent)
        (trace-vars "Variables and values:" (grandad parent grandchild))
        (father parent grandchild)
```

```
        (trace-vars "Variables and values:" (grandad parent grandchild))))))
```

```
> (solve 5 (x y) (grandpa x y))
```

There are times when we would like to watch as the variables are getting instantiated. Using `trace-vars` as an antecedent, and the definition of `grandpa` below, we can watch as the variables become instantiated.

`Exercise 5.2`: Using the current version of our population, what is displayed in the invocation above. ◇

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.

6 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, `project`, and `predicate` in one relation. We start with the youngest common ancestor problem.

6.1 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 "Jon is the father of Hal," "Hal is the father of Ted," and "Sam is the father of Jay."

```
(define father
  (extend-relation (a1 a2) father
    (fact () 'jon 'hal)
    (fact () 'hal 'ted)
    (fact () 'sam 'jay)))
```

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 ((once old) (once young))
      (to-show old young)
      (father old young))
    (relation ((once old) (once 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`. ◇

Here is a simple test.

```
> (solve 100 (x y) (ancestor x y))
(([x.0 jon] [y.0 sam])
 ([x.0 rob] [y.0 sal])
 ([x.0 rob] [y.0 pat])
 ([x.0 sam] [y.0 roz])
 ([x.0 jon] [y.0 hal])
 ([x.0 hal] [y.0 ted])
 ([x.0 sam] [y.0 jay])
 ([x.0 jon] [y.0 rob])
 ([x.0 jon] [y.0 roz])
 ([x.0 jon] [y.0 jay])
 ([x.0 jon] [y.0 sal])
 ([x.0 jon] [y.0 pat])
 ([x.0 sam] [y.0 sal])
 ([x.0 sam] [y.0 pat])
 ([x.0 jon] [y.0 ted]))
```

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 ((once young-a) (once young-b) old)
    (to-show young-a young-b old)
    (all (ancestor old young-a) (ancestor old young-b))))

> (solve 4 (x) (common-ancestor 'pat 'jay x))
(([x.0 jon] [y.0 sam]))
```

This says that Jon and Sam are both common ancestors of Pat and Jay.

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 (once old) (once 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))))
```

```
> (solve 4 (x y) (younger-common-ancestor 'pat 'jay x y))
(([x.0 jon] [y.0 sam]))
```

Thus Sam is the younger of the two common ancestors.

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 Jay share Sam and Jon, but we want the youngest among the common ancestors. Since Jon 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))))))

> (solve 4 (x) (youngest-common-ancestor 'pat 'jay x))
(([x.0 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.

6.2 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 'jon) (== child 'sam))
      (all (== dad 'sam) (== child 'rob))
      (all (== dad 'sam) (== child 'roz))
      (all (== dad 'rob) (== child 'sal))
      (all (== dad 'rob) (== child 'pat))
      (all (== dad 'jon) (== child 'hal))
      (all (== dad 'hal) (== child 'ted))
      (all (== dad 'sam) (== child 'jay)))))

(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.

```
> (solve 20 (x) (ancestor 'jon x))
(([x.0 jon] [y.0 sam])
 ([x.0 jon] [y.0 hal])
 ([x.0 jon] [y.0 rob])
 ([x.0 jon] [y.0 roz])
 ([x.0 jon] [y.0 jay])
 ([x.0 jon] [y.0 sal])
 ([x.0 jon] [y.0 pat])
 ([x.0 jon] [y.0 ted]))
```

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 require that an unsuccessful unification invoke a a non-backtracking failure continuation.

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 `cons`ing 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.

6.3 "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.[7]

```
(define towers-of-hanoi
  (letrec
      ([move
         (extend-relation (a1 a2 a3 a4)
           (fact () 0 _ _ _)
           (relation (n a b c)
             (to-show n a b c)
             (project (n)
               (if (positive? n)
                 (let ([m (- n 1)])
                   (all
                     (move m a c b)
                     (project (a b)
                       (begin
                         (printf "Move a disk from ~s to ~s~n" a b)
                         (move m c b a)))))
                 fail))))])
    (relation (n)
      (to-show n)
      (move n 'left 'middle 'right))))
```

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

```
> (query (redo-k subst) (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. Because each arm of the `if`-expression is an antecedent, we know that using simple `if` works. Also, Recall that the anonymous variable unifies with everything, but it adds nothing to the substitution.

7 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` pedicate. Before we can begin this walk down memory lane, we must enlarge the set of possible terms consequently change the behavior of the unifier.

7.1 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.

```
> (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 sr))
            (concretize (subst-in term sr)))))))
(p (f z.0) z.0 (g z.0))
(p (f b) b (g b))
([y.0 . b] [x.0 . (f y.0)] [z.0 . b])
(p (f b) 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 substi-

tution to the same term. Although we do not run this program, it gives rise to a different way of representing a substitution.

Next, we must change `unify` to accomodate 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 a substitution, 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.

This characterization is only approximate because it does not take into account certain properties that we wish to maintain. (See Properties paper on Source Forge.) Now we get specific.

7.2 Unification with proper (and improper) lists

In most Prologs, there is a conscious decision to avoid concerns for circularity. (This is called 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 one such a unifier.[8]

In this section, we present the full unifier of arbitrarily complex terms. Now, terms include pairs. The premiss is still the same, which is to return a substitution so that if we were to apply `subst-in` to two terms using the same substitution, we would have `equal?` terms. One way would be to produce a new unifier and use the same `subst-in`. Another way would be to produce a new unifier and a new *subst-in*. So, we have chosen the latter approach. We introduce a new `subst-in` that not only handles pairs in the obvious way, but also, treats variables a bit differently. Earlier, we just looked up the associated term of a variable in a substitution, but now, we take that term and using the same substitution, recur on the term. If the variable is not in the substitution, then we return the variable, as before. The critical two lines are the first two. Although we know that _ cannot be a term, it can be stored in a term. Therefore, we make sure that we don't waste resources looking it up on either arm of the recursion.

```
(define subst-in
  (lambda (t subst)
```

[8] The trick used here also appears in SICTus Prolog.

```
(cond
  [(eq? t _) _]
  [(var? t)
   (let*-and t ([c (assq t subst)])
     (subst-in (commitment->term c) subst))]
  [(pair? t)
   (cons (subst-in (car t) subst) (subst-in (cdr t) subst))]
  [else t]))
```

The only way that we add to a substitution during unification is with `extend-subst` below. Thus, we must test for `u` being unequal to `t` and `u` being unequal to `_` as part of the definition of `unify`.

```
(define extend-subst
  (lambda (t u subst)
    (cons (cmmitment t u) subst)))
```

Now that we include pairs in terms, we have many possibilites. To handle all of them, we need to introduce some nomenclature. We refer to a variable that is not in `(map commitment->var subst)` as *free* in the substitution. Otherwise, it is *bound* in the substitution. Although, `_` is a variable, it is never bound in a substitution, and no commitment in a substitution has `_` as its associated term. However, a commitment's term may contain occurrences of `_` within it. This characterization is only possible because we now include pairs. A term without variables is *ground*. When we use `bound` in the name of a procedure, it means that we have passed a commmitment, which will include a bound variable as well as its associated term.

In the `unify` driver below, we first check to see if either argument is `_` or if we have two identical terms, possibly both variables. In each case, we return the substitution that was passed in. If not, we next worry about whether either of the terms are variables. We start by looking at the first argument, `t`. If `t` is a variable, but it is *not* bound, then we have a free variable, so we invoke `unify-free/any`, which will worry about `u`. If `t` is bound, then we get its associated commitment. Then we check to see if `u` is a variable, but not bound. In this case, we invoke `unify-free/bound`. The free variable being passed, however, is `u`, not `t`, since it is bound. If both `t` and `u` are bound, we unify their associated terms. If `u` is not a variable, then we unify it with the term associated with `t`. Next, if `t` is not a variable, We ask similar, but fewer, questions of `u`. If `u` is a variable but not bound, then we can use `unify-free/any`. If `u` is bound, then we unify its associated term with `t`, which we now know cannot be a variable. If both terms are pairs, then we take the substitution returned from a call to `unify` on the *car*s of the two terms and use it as the substitution to unify the `cdr`s of the two terms. If at any time, the substitution returns false, we return false, instead of a substitution. Finally, if we have two terms that can be compared with `equal?` and the two terms are equal, then we return the substitution that we started with.

```
(define unify
  (lambda (t u subst)
```

```
    (cond
      [(eq? t u) subst]
      [(eq? t _) subst]
      [(eq? u _) subst]
      [(var? t)
       (let*-and (unify-free/any t u subst) ([ct (assq t subst)])
         (unify-bound/any ct u subst))]
      [(var? u)
       (let*-and
         (cond
           [(pair? t) (unify-free/list u t subst)]
           [else (extend-subst u t subst)])
         ([cu (assq u subst)])
         (unify (commitment->term cu) t subst))]
      [(and (pair? t) (pair? u))
       (let*-and #f ([subst (unify (car t) (car u) subst)])
         (unify (cdr t) (cdr u) subst))]
      [else (and (equal? t u) subst)])))
```

When one of the variables is bound, we consider the possibilites of the other in `unify-bound/any`. Again for an underscore, we return the substitution. If the other argument is a free variable, we pass the task onto `unify-free/bound`. If not, we pass the task to `unify-bound/bound`.

```
(define unify-bound/any
  (lambda (ct u subst)
    (cond
      [(eq? u _) subst]
      [(var? u)
       (let*-and (unify-free/bound u ct subst) ([cu (assq u subst)])
         (unify-bound/bound ct cu subst))]
      [else
        (unify (commitment->term ct) u subst)])))
```

In `unify-bound/bound`, we utilize the fact that underscore is never the term associated with a commitment. The code is easy to follow if that information is kept in mind, since it handles cases similar to `unify`.

```
(define unify-bound/bound
  (lambda (ct cu subst)
    (let unify-internal ([t (commitment->term ct)]
                         [u (commitment->term cu)]
                         [subst subst])
      (cond
        [(eq? t u) subst]
        [(var? t)
         (let*-and (cond
```

```
                  [(var? u)
                   (let*-and (extend-subst t u subst) ([cu (assq u subst)])
                     (unify-free/bound t cu subst))]
                  [else
                    (extend-subst t u subst)])
          ([ct (assq t subst)])
          (cond
            [(var? u)
             (let*-and (unify-free/bound u ct subst) ([cu (assq u subst)])
               (unify-bound/bound ct cu subst))]
            [else
              (unify-internal (commitment->term ct) u subst)]))]
        [(var? u)
         (let*-and (extend-subst u t subst) ([cu (assq u subst)])
           (unify-internal (commitment->term cu) t subst))]
        [(and (pair? t) (pair? u))
         (let*-and #f ([subst (unify-internal (car t) (car u) subst)])
           (unify-internal (cdr t) (cdr u) subst))]
        [else (and (equal? t u) subst)])))
```

In `unify-free/any` below, we assume that we know nothing about `u`, so we have a test for each possibility: an `_`, a variable, a pair, or anything else. If `u` is an underscore, we just return the substitution. If `u` is a variable, and not bound, we extend the substitution. We can do this because we know that `t-var` and `u` are different. Any calls to `unify-free/any` must verify that property in advance. If `u` is bound, then we invoke `unify-free/bound`, with `u`'s commitment.

```
(define unify-free/any
  (lambda (t-var u subst)
    (cond
      [(eq? u _) subst]
      [(var? u)
       (let*-and (extend-subst t-var u subst) ([cu (assq u subst)])
         (unify-free/bound t-var cu subst))]
      [(pair? u) (unify-free/list t-var u subst)]
      [else (extend-subst t-var u subst)])))
```

Next, we come to `unify-free/bound`. As long as `cu`'s associated term is not `t-var`, but is a variable, we loop on that variable if it is bound. If the variable is free, we just extend the substitution. We can extend the substitution since we know that `t-var` and `u-term` are different. Finally, if `u-term` is not a variable, we can also extend the substitution.

```
(define unify-free/bound
  (lambda (t-var cu s)
    (let loop ([cm cu])
      (let ([u-term (commitment->term cm)])
```

```
      (cond
        [(eq? u-term t-var) s]
        [(var? u-term)
          (cond
            [(assq u-term s) => loop]
            [else (extend-subst t-var u-term s)])]
        [else (extend-subst t-var u-term s)])))))
```

Only `unify-free/list` remains to be described. How should we unify a free variable with a list (or improper list)? One way would be with this definition.

```
(define unify-free/list
  (lambda (t-var u subst)
    (extend-subst t-var u subst)
```

But, this definition does not handle the case where `t-var` occurs in `u`, which would cause a circularity. (This *occurs* means `u` and any terms associated with variables within `u`, recursively.) Instead, we offer the definition of `unify-free/pair` below, which handles these kinds of circularites. If we know that these circularities will not be a problem, then we are free to use the definition above.

Our plan is to take each top-level pair (and final `cdr`) that is not ground in its `car` (or final `cdr`), and build a new flat structure by replacing each such top-level term by a fresh (free) logical variable. That is the result of invoking `ufl-rebuild-with-vars` (See its definition below.). To do this rebuilding, we keep an ordered association list of these fresh variables and their associated top-level terms. This is accomplished with `ufl-analyze-pair` below. But, in order to rebuild the pair, we must find each top-level term and associated fresh variable by the equivalent of `(assq term (map (lambda (x) (cons (cdr x) (car x))) <a-list>))`. Of course, since the terms are ordered, we can do them one at a time and *in one pass* until we find all the pairs in the association list. If there is only one item in the association list and it is not a pair (something that has a `cdr`), then we know that the flat list has a final `cdr`. That is why in `ufl-rebuild-with-vars`, we can replace it with the fresh variable without matching for its associated term using `eq?`. This algorithm has the property that as long as `car`s are not ground, that term is not copied, which improves efficiency.

```
(define unify-free/list
  (lambda (t-var u-value subst)
    (let ([to-unify (ufl-analyze-list u-value)])
      (cond
        [(null? to-unify)
         (extend-subst t-var u-value subst)]
        [else
          (let loop ([subst
                      (unify-free/any (caar to-unify) (cdar to-unify)
                        (extend-subst t-var
                          (ufl-rebuild-with-vars to-unify u-value)
```

```
                          subst))]
                  [to-unify (cdr to-unify)])
            (cond
              [(null? to-unify) subst]
               [else
                 (loop (unify-free/any (caar to-unify) (cdar to-unify) subst)
                   (cdr to-unify))])])))))
```

So, we'll assume that we have built `to-unify`, the association list of fresh variables and their terms. If this list is empty, then we know that the term is ground, so we simply extend the substitution. If not, we rebuild the term to be flat using this association list. Next, we set up a loop. Each time through the loop we construct a new substitution and `cdr` down the association list until we reach the end. So, we are basically left with the question of how do we construct the new substitution. We are going to invoke `unify-free/any` using each pair of the association list to get the final substitution. We can do this because we know that each fresh variable is not `eq?` to its associated term. But, with what substitution are we going to get it started? We know that we need to associate something with the free `t-var`, so we take the flattened (possibly improper) rebuilt list and add it to the substitution before we get started. Once we have started, we unify the variables to the terms in the association list, potentially increasing the size of the substitution. Why only potentially? Because in `unify-free/any`, it is again possible that the term is `_`, and it does not lead to a new commitment.

```
(define ufl-analyze-list
  (lambda (lst-src)
    (cond
      [(pair? lst-src)
       (cond
         [(ground? (car lst-src)) (ufl-analyze-list (cdr lst-src))]
         [else
           (let ([fresh-var (logical-variable 'a*)])
             (cons (cons fresh-var (car lst-src))
               (ufl-analyze-list (cdr lst-src))))])]
      [(or (null? lst-src) (ground? lst-src)) '()]
      [else (list (cons (logical-variable 'd*) lst-src))])))

(define ground?
  (lambda (t)
    (cond
      [(var? t) #f]
      [(pair? t) (and (ground? (car t)) (ground? (cdr t)))]
      [else #t])))

(define ufl-rebuild-with-vars
  (lambda (term-assoc lst)
    (cond
```

```
      [(null? term-assoc) lst]
      [(pair? lst)
        (if (eq? (cdar term-assoc) (car lst))
          (cons (caar term-assoc)
            (ufl-rebuild-with-vars (cdr term-assoc) (cdr lst)))
          (cons (car lst)
            (ufl-rebuild-with-vars term-assoc (cdr lst))))]
      [else (caar term-assoc)])))
```

We have now completed the unifier, which has bascially four functions: `unify`, `unify-free/any`, `unify-free/bound`, and `unify-free/pair`. The function `unify-free/bound` is a leaf function, since we treat `extend-subst` as primitive. The function `unify-free/any` is called from `unify` and from `unify-free/pair`, and it calls `unify-free/bound`. The functions `unify`, `unify-free/any`, and `unify-free/bound` are called from `unify`. The calling structure is a bit complicated, but there are only six functions and three of them assume that the first argument is a free variable, two of them assume that the first argument is a bound variable, one of them is a leaf function.

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)
       (let*-and t ([cmt (assq t subst)])
         (subst-vars-recursively
           (commitment->term c) (remq c subst)))]
      [(pair? t)
       (cons
         (subst-vars-recursively (car t) subst)
         (subst-vars-recursively (cdr t) subst))]
      [else t])))

> (exists (x y)
    (concretize
      (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))))))
([y.0 . (f y.0)]
 [x.0 . (f y.0)])
```

The output shows the circularity in the associated term of `y.0` and `(f y.0)`. Therefore, any term that contains them is circular, which includes the terms associated with `x.0` and `y.0`.

7.2.1 "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)
               (fact () 0 _ _ _)
               (relation (n a b c)
                 (to-show n a b c)
                 (project (n)
                   (if (positive? n)
                     (let ([m (- n 1)])
                       (all
                         (move m a c b)
                         (project (a b)
                           (begin
                             (push-step a b)
                             (move m c b a)))))
                     fail)))])
        (relation (n path)
          (to-show n path)
          (begin
            (set! steps '())
            (any
              (fails (move n 'l 'm 'r))
              (== path (reverse steps)))))))))
```

```
> (query (redo-k subst path) (towers-of-hanoi-path 3 path) (subst-in path subst))
((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. ◇

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

7.3 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

```
> (query (fk subst q) (== q (concat '(a b c) '(u v))) (subst-in q subst))
(a b c u v)
```

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

```
(define concat
  (extend-relation (a1 a2 a3)
    (fact (xs) '() xs xs)
    (relation (x xs (once ys) zs)
      (to-show `(,x . ,xs) ys `(,x . ,zs))
      (concat xs ys zs))))
```

```
> (expand-only* '(relation project predicate)
    '(define concat
       (extend-relation (a1 a2 a3)
         (fact (xs) '() xs xs)
         (relation (x xs (once ys) zs)
           (to-show `(,x . ,xs) ys `(,x . ,zs))
           (concat xs ys zs)))))
```

```
(define concat
  (extend-relation (a1 a2 a3)
    (fact (xs) '() xs xs)
    (lambda (g1 ys g2)
      (exists (zs xs x)
        (lambda@ (sk fk subst)
```

```
          (let*-and (fk)
            ((subst (unify g2 `(,x . ,zs) subst))
             (subst (unify g1 `(,x . ,xs) subst)))
            (@ (concat xs ys zs) sk fk subst)))))))
```

```
> (query (fk subst q) (concat '(a b c) '(u v) q) (subst-in q subst))
(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.

```
> (query (fk subst q) (concat '(a b c) q '(a b c u v)) (subst-in q subst))
(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)`.

```
> (query (fk subst q) (concat q '(u v) '(a b c u v)) (subst-in q subst))
(a b c)
```

But what if we include another variable?

```
> (query (fk subst q r) (concat q r '(a b c u v)) (cons (subst-in (list q r) subst) (fk)))
((() (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?

```
> (solve 6 (q r s) (concat q r s))
> (solve 4 (q r s) (concat q r s))
(([q.0 ()] [r.0 r.0] [s.0 r.0])
 ([q.0 (x.0)] [r.0 r.0] [s.0 (x.0 . r.0)])
 ([q.0 (x.0 x.1)] [r.0 r.0] [s.0 (x.0 x.1 . r.0)])
 ([q.0 (x.0 x.1 x.2)] [r.0 r.0] [s.0 (x.0 x.1 x.2 . r.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.

```
> (solve 4 (q r) (concat q '(u v) `(a b c . ,r)))
(([q.0 (a b c)] [r.0 (u v)])
 ([q.0 (a b c x.0)] [r.0 (x.0 u v)])
 ([q.0 (a b c x.0 x.1)] [r.0 (x.0 x.1 u v)])
 ([q.0 (a b c x.0 x.1 x.2)] [r.0 (x.0 x.1 x.2 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.

```
> (solve 4 (q) (concat q '() q))
(([q.0 ()])
 ([q.0 (x.0)])
 ([q.0 (x.0 x.1)])
 ([q.0 (x.0 x.1 x.2)]))
```

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`.

7.4 The Type-Inference Problem (rest of document is under repair)

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.

7.4.1 Buiding 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 (cadddr 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 (cadddr 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)

> (let ([result (solution (g) (!- g (parse 17) int))])
    `(!- ,(cadar result) 17 int))
(!- g.1 17 int)

> (let ([result (solution (g ?) (!- g (parse 17) ?))])
    `(!- ,(cadar result) ,(unparse (parse 17)) ,(caadr result)))
(!- g.1 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 type) (!- g (parse term) type))
        => (lambda (result)
             `(!- ,(cadr (car result)) ,term ,(cadr (cadr 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 primitves `zero?`, `sub1`, and `+`, respectively.

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

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

(define +-rel
  (relation ((once 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))
```

```
> (infer-type g '(zero? 24) ?)
(!- g.1 (zero? 24) bool)

> (infer-type g '(zero? (+ 24 50)) ?)
(!- g.1 (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.1 (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 ((once g) v (once 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)
    (to-show `(non-generic ,_ ,_ ,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 (g ?) (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 ((once 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 (once 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 `(--> type-v.0 (--> 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 `(--> 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 `lambda`s with the arrows. Here we have two `lambda`s 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 ((once g) rand (once 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)
```

7.5 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 `lamdda` expressions.

```
(define polylet-rel
  (relation (g v rand body (once 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`, 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
                [(assq t env)
                 => (lambda (pr)
                      (values (cdr pr) env))]
                [else (let ([new-var (logical-variable (logical-variable-id t))])
                        (values new-var (cons `(,t . ,new-var) env)))])]
             [(pair? t)
              (let-values (a-t env) (instantiate-term (car t) env)
                (let-values (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 (once t))
    (to-show `(generic ,v (--> ,targ ,tresult) ,g) v t)
    (project/no-check (targ tresult)
      (== t (instantiate `(--> ,targ ,tresult))))))

(define generic-recursive-env
  (relation (g (once v) (once t))
    (to-show `(generic ,_ ,_ ,g) v t)
    (all!! (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)
```

7.5.1 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.

```
> (solution (g ?) (!- g ? '(--> int int)))
([g.0 (non-generic v.0 (--> int int) g.0)] [?.0 (var v.0)])


> (solution (g la f b) (!- g `(,la (,f) ,b) '(--> int int)))
([g.0 g.0]
 [la.0 lambda]
 [f.0 f.0]
 [b.0 (var f.0)])

> (solution (g h r q z y t) (!- g `(,h ,r (,q ,z ,y)) t))
([g.0 (non-generic v.0 int g.0)]
 [h.0 +]
 [r.0 (var v.0)]
 [q.0 +]
 [z.0 (var v.0)]
 [y.0 (var v.0)]
 [t.0 int])

> (solution (g h r q z y t u v) (!- g `(,h ,r (,q ,z ,y)) `(,t ,u ,v)))
([g.0 g.0]
 [h.0 lambda]
 [r.0 (v.0)]
 [q.0 +]
 [z.0 (var v.0)]
 [y.0 (var v.0)]
 [t.0 -->]
 [u.0 int]
 [v.0 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.

`Exercise d`: Take the results of these four test programs and reconstruct what the terms are by substituting for each variable. ◇

7.6 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))
          (project (x y)
            (== z (op x y)))))
      (relation (x y z)
        (to-show x y z)
        (all
          (fails (instantiated y))
          (project (z x)
            (== y (inverted-op z x)))))
      (relation (x y z)
        (to-show x y z)
        (all
          (fails (instantiated x))
          (project (z y)
            (== x (inverted-op z y)))))
      (relation (x y z)
        (to-show x y z)
        (project (x y)
          (== z (op 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 / *))

> (solution (x) (++ x 16.0 8))
([x.0 -8.0])

> (solution (x) (** 10 x 50))
([x.0 5])

> (solution (x) (-- 10 7 x))
([x.0 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))
          (project (x)
            (== y (op x)))))
      (relation (x y)
        (to-show x y)
        (all
          (fails (instantiated x))
          (project (y)
            (== x (inverted-op y)))))
      (relation (x y)
        (to-show x y)
        (begin
          (pretty-print "Third rule")
          (project (x)
            (== y (op 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 lnums)))))

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

```
> (solution (x) (name 'sleep x))
([x.0 (115 108 101 101 112)])

> (solution (x) (name x '(115 108 101 101 112)))
([x.0 sleep])
```

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 treat-

ed, like `symbol->string` and `string->symbol`, but we leave their inclusion to the programmer who might need them.

7.7 Proving a nontrivial theorem

In this section, we present a simple program and then we prove a simple fact about the program. The program is the function `mirror`, which takes an arbitrary S-expression of pairs and atomic values and if its argument is a pair, it swaps the `car` with the `cdr`, recursively. Otherwise, the atomic value is left unchanged.

Here is the definition of `mirror`.

```
(define mirror
  (lambda (x)
    (cond
      [(not (pair? x)) x]
      [else (cons (mirror (cdr x)) (mirror (car x)))])))
```

and here is the theorem about this definition that we shall prove:

$$\forall x.\ x = (\text{mirror}(\text{mirror}\, x))$$

First of all, we have no notion of $\forall$. Second, we have no notion of what $=$ means. We address these issues below. On some level, we know that $\forall x$ means that the predicate must be true for every element of the domain of the discourse–that is, all Scheme values. That is perhaps too strong a statement. We want the formula $x = (\text{mirror}(\text{mirror}\, x))$ to be true for every *suitable* value x, that is, for every x that makes sense to pass to the procedure `mirror`.

$$\forall x.\ (\text{suitable}\, x) \wedge x = (\text{mirror}(\text{mirror}\, x))$$

The predicate $(\text{suitable}\, x)$ restricts the values x to be in the domain of *mirror*.

Let's test the theorem on a few examples:

```
> (mirror (mirror '(a b c)))
(a b c)
> (mirror (mirror '((a b) c)))
((a b) c)
```

Clearly, it makes sense to see if there is an obvious program that contradicts the *theorem*, but since we present the full proof of the theorem, we know that no such counterexample exists.

Now, what does the $=$ mean? We might be tempted to treat $=$ as a logical equivalence, and try to show that `(== x (mirror (mirror x)))` in our logical system. That would not be right. We must draw a distinction between logical (i.e., tautological) equivalence, which is syntactic, and domain equality (which is semantic). This distinction is akin to the distinction between `eq?` and `equal?` in the pure functional subset of Scheme. The relation $=$ that appears in our theorem is a binary equality relation defined in our domain. We shall use EQ for $=$ to avoid confusing this relation with the logical equivalence or Scheme's numeric equality. Incidentally, we

cannot formulate our theorem as `(== x (mirror (mirror x)))` for another reason: the latter relation is empty, as an attempt to unify `x` with `(mirror (mirror x))` would create an unacceptable circularity.

The relation EQ is an equivalence relation on our domain, so it is supposed to satisfy the usual axioms of reflexivity, symmetry, and transitivity. Thus we need a way to introduce axioms. In mathematics, we introduce a two argument entailment relation $\vdash$.

$$\Gamma \vdash \mathit{conseq}$$

asserts *conseq* subject to the list of assumptions Γ. In other words, *conseq* is derivable from Γ using the existing axioms and inference rules. The list of assumptions can be empty (in which case the notation above is truly an axiom). Thus the three axioms of equivalence relations EQ can be stated as

$$\begin{aligned} & \vdash && (\mathrm{E}Q\ \mathit{val}\ \mathit{val}) \\ (\mathrm{E}Q\ a\ b) & \vdash && (\mathrm{E}Q\ b\ a) \\ (\mathrm{E}Q\ a\ c), (\mathrm{E}Q\ c\ b) & \vdash && (\mathrm{E}Q\ a\ b) \end{aligned}$$

It is still proper to call these axioms *schemas* because they contain free variables. As usual, the free variables that occur on the right-hand side (such as a and b) are universally quantified and the free variables that occur only on the left-hand side (such as c) are existentially quantified.

In Prolog notation, we can write these axioms as

```
proof(eq(Val,Val)).
proof(eq(B,A)) :- proof(eq(A,B)).
proof(eq(A,B)) :- proof(eq(A,C)), proof(eq(C,B)).
```

meaning that there is a proof of `eq(Val,Val)`, a proof exists for `eq(B,A)` if there is a proof of `eq(A,B)`, and that if `eq(A,C))` and `eq(C,B)` are provable, then so is `eq(A,B)`. The latter style makes our prover a backward-chaining prover.

Our system is designed not so much for theorem proving, but for proof verification (see Athena system). It is important to let the user specify exactly which axioms should participate in a proof of each particular formula. Thus, we need a way to represent sets of axioms. We introduce a first-class relation called *kb*. The relation is unary: its domain is the set of formulas. (kb $formula$) is non-empty if $formula$ is either an axiom or can be derived from axioms in finitely many steps. Because our workhorse relations are unary, it helps to add some syntactic sugar.

```
(define-syntax extend-unary-relation
  (syntax-rules ()
    [(_ rel1 ...) (extend-relation (a1) rel1 ...)]))
```

We can write the three axioms of equivalence relations for EQ in our notation as follows:

```
(define EQ-axioms
  (lambda (kb)
    (extend-unary-relation
```

```
      (fact (val) `(EQ ,val ,val))
      (relation (a b)
        (to-show `(EQ ,a ,b))
        (project/no-check (a b)
          (all
            (predicate (printf "symmetry: ~a ~a ~n" (concretize a) (concretize b)))
            (kb `(EQ ,b ,a)))))
      (relation (a b)
        (to-show `(EQ ,a ,b))
        (exists (c)
          (all
            (kb `(EQ ,a ,c))
            (kb `(EQ ,c ,b))))))))
```

The first axiom, *reflexivity*, says that something is equal to itself. To be more precise, identical things are equal: syntactic equality implies semantic equality. The second axiom, *symmetry*, says that (EQ a b) is provable if (EQ b a) is provable. Finally, we have the last axiom, *transitivity*. It says that if we have c so that both (EQ a c) and (EQ c b) are provable, then we may conclude that (EQ a b) is provable.

Recall that we have a fix point operator Y such that we can turn a λ expression into a recursive function.

```
(define Y
  (lambda (f)
    ((lambda (u) (u (lambda (x) (lambda (n) ((f (u x)) n)))))
     (lambda (x) (x x)))))

> ((Y (lambda (!)
       (lambda (n)
         (if (zero? n)
           1
           (* n (! (- n 1)))))))) 5)
120
```

A `LEAF` wrapped with `BTREE` is a `BTREE` and a `ROOT` wrapped with `BTREE` is a `BTREE` provided that its two sub-terms are `BTREE`s in the thus far generated knowledge base, `kb`.

```
(define is-a-BTREE
  (lambda (kb)
    (extend-unary-relation
      (fact (val) `(BTREE (LEAF ,val)))
      (relation (t1 t2)
        (to-show `(BTREE (ROOT ,t1 ,t2)))
        (project/no-check (t1 t2)
          (all
```

```
        (predicate (printf "BTREE ~s ~s ~n" (concretize t1) (concretize t2)))
        (kb `(BTREE ,t1))
        (kb `(BTREE ,t2))))))))
```

Then using these characterizations, we have an axiom for trees. This axiom says that two `ROOT`s are EQ if using `kb`, their `car`s and `cdr`s are EQ.

We must now write our semantic equality relation EQ on our domain. We have to define which trees we shall consider equal.

```
(define EQ-axioms-trees
  (lambda (kb)
    (extend-unary-relation
      (relation (a b c d)
        (to-show `(EQ (ROOT ,a ,b) (ROOT ,c ,d)))
        (all
          (project/no-check (a b)
            (all
              (predicate
                (printf "trees: ~a ~a ~a ~a ~n"
                  (concretize a) (concretize b) (concretize c) (concretize d)))
              (kb `(EQ ,a ,c))
              (kb `(EQ ,b ,d)))))))))
```

The fact in `EQ-axioms-trees` is redundant, since we can rely on EQ's property of reflexivity. Thus, we simplify the definition.

```
(define EQ-axioms-trees
  (lambda (kb)
    (extend-unary-relation
      (relation (a b c d)
        (to-show `(EQ (ROOT ,a ,b) (ROOT ,c ,d)))
        (all
          (project/no-check (a b)
            (all
              (predicate
                (printf "trees: ~a ~a ~a ~a ~n"
                  (concretize a) (concretize b) (concretize c) (concretize d)))
              (kb `(EQ ,a ,c))
              (kb `(EQ ,b ,d)))))))))
```

In addition, we have three axioms relating to `mirror` and EQ.

```
(define EQ-axioms-MIRROR
  (lambda (kb)
    (extend-unary-relation
      (relation (a b)
        (to-show `(EQ (MIR ,a) ,b))
        (project/no-check (a b)
```

```
          (all
            (predicate (printf "mirror: ~a ~a~n" (concretize a) (concretize b)))
            (exists (c)
              (all
                (kb `(EQ ,b (MIR ,c)))
                (kb `(EQ ,a ,c))))))))))

(define MIRROR-axiom-EQ-1
  (lambda (kb)
    (relation (val)
      (to-show `(EQ (LEAF ,val) (MIR (LEAF ,val))))
      (project/no-check (val)
        (predicate (printf "mirror ax1: ~a~n" (concretize val)))))))

(define MIRROR-axiom-EQ-2
  (lambda (kb)
    (relation (t1 t2)
      (to-show `(EQ (MIR (ROOT ,t1 ,t2)) (ROOT (MIR ,t2) (MIR ,t1))))
      (project/no-check (t1 t2)
        (predicate
          (printf "mirror ax2: ~a~n"
            (concretize `(EQ
                          (ROOT ,t1 ,t2)
                          (ROOT (MIR ,t2) (MIR ,t1))))))))))
```

The first axiom says that `mirror` has no effect on leaves. The second axiom says that `mirror` of a `ROOT` is the `ROOT` with mirror applied to the `cdr` and the `car`. This last axiom states that to show `(EQ (MIR` a`)` b`)`, show that there exists a c such that in the knowledge base `kb (EQ` b `(MIR` c`))` is provable and that `(EQ` a c`)`.

There is something subtle going on here. We are proving a theorem about terms; we are not running a program. Therefore, we have used the all uppercase symbols to remind us that we are not invoking procedures like `EQ`, `MIR`, `ROOT`, or `LEAF`. They are just constants. To be more precise, `EQ`, `MIR`, `ROOT`, and `LEAF` are functional symbols of our term algebra. We build formulas from these symbols, denoting various objects of our domain, and variables. Variables stand for objects of our domain, thus our formulas are first-order. We are interested in proving or verifying theorems: that is, given a formula, a set of assumptions, and a set of axioms, we verify that the resultant formula is provable using modus potens.

Our goal is to prove

$$\forall x.\ \text{suitable}(x) \wedge x = (\text{mirror}(\text{mirror}\, x))$$

where suitable(x) is btree(x). For brevity, we introduce a predicate GOAL such that

$$GOAL(x) === btree(x) \wedge x = (mirror(mirror x))$$

In other words, we introduce two axioms for our new literal GOAL:

$$GOAL(x) \quad \vdash \quad btree(x) \wedge x = (\text{mirror}(\text{mirror}\, x))$$
$$btree(x) \wedge x = (\text{mirror}(\text{mirror}\, x)) \quad \vdash \quad GOAL(x)$$

For abbreviation we shall call the first axiom `Goal-Rev` and the second `Goal-Fwd`. In the notation of our system, `Goal-Fwd` can be written as

```
(define GOAL-fwd
  (lambda (kb)
    (relation (t)
      (to-show `(GOAL ,t))
      (all
        (kb `(BTREE ,t))
        (kb `(EQ (MIR (MIR ,t)) ,t))))))
```

and the act of adding it to the current set of axioms `kb` can be written as

```
(let ([kb (extend-unary-relation (GOAL-fwd kb) kb)]) ...)
```

In logic, we want to prove

$$\vdash \forall x\, Goal(x)$$

We shall use the standard logical inference rule "From $(C \rightarrow A(x))$, where x is a variable that does not occur free in C, conclude $(C \rightarrow \forall x\, A(x))$." where C is true and $A(x)$ is $Goal(x)$. Thus, it's enough to prove

$$\vdash Goal(T)$$

where T is some literal (constant).

We shall be doing a structural induction proof, which consists of the base step and the induction step. The proof follows the inductive structure of our domain, BTREEs. For the base step, we have to prove

$$\vdash Goal(LEAF(Val))$$

Here's the proof by hand. By the axiom,

$$\vdash BTREE(LEAF(Val))$$

Then by axiom,

$$\vdash EQ(LEAF(Val), MIR(LEAF(Val)))$$

With the axiom of symmetry of EQ,

$$\vdash EQ(MIR(LEAF(Val)), LEAF(Val))$$

Finally, using the axiom of EQ with respect to MIR,

$$\vdash EQ(MIR(MIR(LEAF(Val))), LEAF(VAL))$$

Then $Goal(Leaf(Val))$ follows from the GoalFwd axiom. Here's how we ask our system to verify the same proof.

```
(define eigen
  (lambda (symbol)
    (artificial-id `(,(logical-variable symbol) . ,(random 10)))

> (concretize-subst
    (car
      (query
        (let ([eigen-x (eigen 'x)])
          (let ([kb (Y init-kb-coll)])
            (let ([kb (extend-unary-relation (GOAL-fwd kb) kb)])
              (kb `(GOAL (LEAF ,eigen-x)))))))))
```

For the induction step, we have to prove that

$$\vdash \forall t_1, t_2\ Goal(t_1), Goal(t_2) \rightarrow Goal(ROOT(t_1, t_2))$$

Again, by availing ourselves to the inference rule, we merely need to prove that

$$\vdash Goal(t_1), Goal(t_2) \rightarrow Goal(ROOT(t_2, t_2))$$

where t_1 and t_2 are some constants.

We shall now use the deduction theorem

$$A \vdash B \Longleftrightarrow \vdash A \rightarrow B$$

(provided that we do not quantify the free variables that may occur in A).

To be precise, first we rearrange $A, B \rightarrow C$ as $A \rightarrow (B \rightarrow C)$ so

$$\begin{array}{rcl} & \vdash & (A \wedge B) \rightarrow C \\ & \vdash & A \rightarrow (B \rightarrow C) \\ A & \vdash & B \rightarrow C \\ A, B & \vdash & C \end{array}$$

So, we need to prove

$$Goal(t_1), Goal(t_2) \vdash Goal(ROOT(t_1, t_2))$$

In our system,

```
> (concretize-subst
    (car
      (query
        (let ([eigen-x (eigen 'x)]
              [eigen-y (eigen 'y)])
          (let ([kb (Y (lambda (kb)
                         (extend-unary-relation
                           (init-kb-coll kb)
                           (fact () `(GOAL ,eigen-x))
                           (fact () `(GOAL ,eigen-y))
                           (GOAL-rev kb))))])
```

```
            (let ([kb (GOAL-fwd kb)])
              (kb `(GOAL (ROOT ,eigen-x ,eigen-y)))))))))
```

We add to our current set of axioms assumptions $Goal(t_1)$, $Goal(t_2)$, the axiom `Goal-rev`, and the axiom `Goal-fwd`. We then try to prove $Goal(ROOT(t_1, t_2))$. The system says that the relation `(kb `(GOAL (ROOT ,eigen-x ,eigen-y)))` is satisfiable–there is a proof of $Goal(ROOT(t_1, t_2))$.

`Goal-rev` in the above relation is

```
(define GOAL-rev
  (lambda (kb)
    (extend-unary-relation
      (relation (t)
        (to-show `(BTREE ,t))
        (kb `(GOAL ,t)))
      (relation (t)
        (to-show `(EQ (MIR (MIR ,t)) ,t))
        (kb `(GOAL ,t))))))
```

In the above, `init-kb-coll` is the set of basic axioms, which we extend with other axioms to prove one or the other branch of the complex proof. The `with-depth` is a way of making sure that we don't loop while trying the axioms. The number 5, which is one larger than the number of arguments to `any`, manages that.

```
(define init-kb-coll
  (lambda (kb)
    (lambda (t)
      (with-depth 5
        (any
          ((is-a-BTREE kb) t)
          ((EQ-axioms-atoms kb) t)
          ((EQ-axioms-MIRROR kb) t)
          ((EQ-axioms-trees kb) t))))))

(define depth-counter-var (exists (*depth-counter-var*) *depth-counter-var*))

(define with-depth
  (lambda (limit ant)
    (lambda@ (sk fk subst)
      (cond
        [(assq depth-counter-var subst)
         => (lambda (cmt)
              (let ([counter (commitment->term cmt)])
                (if (= counter limit)
                  (fk)
                  (let ([s (extend-subst depth-counter-var (+ counter 1) subst)])
                    (@ ant sk fk s)))))]
```

```
      [else
        (let ([s (extend-subst depth-counter-var 0 subst)])
          (@ ant sk fk s))])))))
```

The procedure `with-depth` takes an upper bound and an antecedent and it returns an antecedent to be run in the same state one more time than the limit for that antecedent. It does this by associating a counter with a substitution, so that a particular antecedent that runs in that state gets run a limited number of times. This, keeps infinite loops, which are natural when proving theorems, from taking place. Alternatively, `with-depth` limits the number of applications of a particular axiom/rule when searching for a proof. By extending the substitution on each invocation of the antecedent, and by searching using `assq`, we are guaranteed to get the most recent value of the counter associated with that substitution.

Now, we are ready to test the program to see if it can prove the theorem. But, before we do, we have one additional facet to explain. A logician writes $\forall x\ P(x)$, but this is really the same as $P(x)$. But, to make the point, the x has to be chosen arbitrarily. Thus we can say that for any x, since it is chosen arbitrarily, $P(x)$ holds. The arbitrary variable is called an *eigenvariable*. We will model an eigenvariable using `artificial-id`, which we recall produces a symbol. In each of the four examples, we return a substitution, which means that the `(GOAL ...)` succeeded. The examples are used for the proof. The first one proves the base case of the induction and the second one proves the inductive case. We leave the details for the reader to work out.

```
> (pretty-print
    (concretize-subst
      (car
        (query
          (let ([eigen-x (eigne 'x)])
            (let ([kb0 (Y (lambda (kb)
                            (extend-unary-relation
                            (MIRROR-axiom-EQ-1 kb)
                            (init-kb-coll kb))))])
              (let ([kb1 (extend-unary-relation (GOAL-fwd kb0) kb0)])
                (kb1 `(GOAL (LEAF ,eigen-x)))))))))))
```

```
sym: (MIR (MIR (LEAF x.8))) (LEAF x.8)
sym: (LEAF x.8) (MIR (MIR (LEAF x.8)))
sym: (MIR (MIR (LEAF x.8))) (LEAF x.8)
sym: (LEAF x.8) (MIR (MIR (LEAF x.8)))
sym: (MIR (MIR (LEAF x.8))) (LEAF x.8)
mirror: (MIR (LEAF x.8)) (LEAF x.8)
mirror ax1: x.8
mirror ax1: x.8
sym: (MIR (LEAF x.8)) (MIR (MIR (LEAF x.8)))
mirror: (LEAF x.8) (MIR (MIR (LEAF x.8)))
sym: (LEAF x.8) b.0
mirror ax1: x.8
sym: (MIR (MIR (LEAF x.8))) (LEAF x.8)
mirror: (MIR (LEAF x.8)) (LEAF x.8)
mirror ax1: x.8
sym: (MIR (MIR (LEAF x.8))) b.0
sym: a.0 (MIR (MIR (LEAF x.8)))
mirror ax1: val.0
mirror: a.0 (MIR (MIR (LEAF x.8)))
sym: (MIR (MIR (LEAF x.8))) b.0
mirror: (MIR (LEAF x.8)) b.0
mirror ax1: val.0
mirror: (MIR (LEAF x.8)) b.0
mirror ax1: val.0
sym: (MIR (LEAF x.8)) (LEAF val.0)
mirror ax1: x.8
mirror: (LEAF x.8) (LEAF val.0)
mirror ax1: val.0
sym: a.0 (MIR c.0)
mirror ax1: val.0
mirror: a.0 (MIR c.0)
mirror: (MIR (LEAF x.8)) (LEAF x.8)
mirror ax1: x.8
sym: (MIR (LEAF x.8)) (LEAF x.8)
mirror ax1: x.8

([*depth-counter-var*.0 . 4]
 [a*.0 . x.8]
 [*depth-counter-var*.0 . 3]
 [*depth-counter-var*.0 . 2]
 [a*.1 . (LEAF x.8)]
 [a*.2 . (MIR a*.1)]
 [*depth-counter-var*.0 . 1]
 [*depth-counter-var*.0 . 0])
```

```
> (concretize-subst
    (car
      (query
        (let ([eigen-x (eigen 'x)]
              [eigen-y (eigen 'y)])
          (let ([kb (Y (lambda (kb)
                         (extend-unary-relation
                           (init-kb-coll kb)
                           (fact () `(GOAL ,eigen-x))
                           (fact () `(GOAL ,eigen-y))
                           (MIRROR-axiom-EQ-2 kb)
                           (GOAL-rev kb))))])
            (let ([kb1 (GOAL-fwd kb)])
              (kb1 `(GOAL (ROOT ,eigen-x ,eigen-y)))))))))
```

```
BTREE x.5 y.4
sym: (MIR (MIR (ROOT x.5 y.4))) (ROOT x.5 y.4)
sym: (ROOT x.5 y.4) (MIR (MIR (ROOT x.5 y.4)))
sym: (MIR (MIR (ROOT x.5 y.4))) (ROOT x.5 y.4)
sym: (ROOT x.5 y.4) (MIR (MIR (ROOT x.5 y.4)))
sym: (MIR (MIR (ROOT x.5 y.4))) (ROOT x.5 y.4)
mirror: (MIR (ROOT x.5 y.4)) (ROOT x.5 y.4)
sym: (ROOT x.5 y.4) b.0
trees: x.5 y.4 c.0 d.0
sym: (MIR (MIR (ROOT x.5 y.4))) (ROOT x.5 y.4)
mirror: (MIR (ROOT x.5 y.4)) (ROOT x.5 y.4)
sym: (MIR (MIR (ROOT x.5 y.4))) b.0
sym: a.0 (MIR (MIR (ROOT x.5 y.4)))
mirror ax2: (EQ (ROOT t1.0 t2.0) (ROOT (MIR t2.0) (MIR t1.0)))
mirror: a.0 (MIR (MIR (ROOT x.5 y.4)))
sym: (MIR (MIR (ROOT x.5 y.4))) b.0
mirror: (MIR (ROOT x.5 y.4)) b.0
mirror: (MIR (ROOT x.5 y.4)) b.0
mirror ax2: (EQ (ROOT x.5 y.4) (ROOT (MIR y.4) (MIR x.5)))
sym: a.0 (MIR c.0)
mirror ax2: (EQ (ROOT t1.0 t2.0) (ROOT (MIR t2.0) (MIR t1.0)))
mirror ax2: (EQ (ROOT x.5 y.4) (ROOT (MIR y.4) (MIR x.5)))
mirror ax2: (EQ (ROOT t1.0 t2.0) (ROOT (MIR t2.0) (MIR t1.0)))
mirror: a.0 (MIR c.0)
mirror: (MIR (ROOT x.5 y.4)) (ROOT x.5 y.4)
sym: (ROOT x.5 y.4) (MIR c.0)
sym: (MIR c.0) (ROOT x.5 y.4)
mirror ax2: (EQ (ROOT t1.0 t2.0) (ROOT (MIR t2.0) (MIR t1.0)))
mirror: a.0 (ROOT x.5 y.4)
sym: (ROOT x.5 y.4) b.0
trees: x.5 y.4 c.0 d.0
sym: (ROOT x.5 y.4) (MIR (MIR (ROOT x.5 y.4)))
sym: (MIR (MIR (ROOT x.5 y.4))) (ROOT x.5 y.4)
mirror: (MIR (ROOT x.5 y.4)) (ROOT x.5 y.4)
sym: (ROOT x.5 y.4) b.0
trees: x.5 y.4 c.0 d.0
sym: (ROOT x.5 y.4) b.0
sym: (ROOT x.5 y.4) (MIR (MIR (ROOT x.5 y.4)))
sym: a.0 (ROOT x.5 y.4)
sym: (ROOT x.5 y.4) b.0
trees: x.5 y.4 c.0 d.0
sym: a.0 b.0
mirror ax2: (EQ (ROOT t1.0 t2.0) (ROOT (MIR t2.0) (MIR t1.0)))
mirror ax2: (EQ (ROOT t1.0 t2.0) (ROOT (MIR t2.0) (MIR t1.0)))
mirror: a.0 b.0
```

```
trees: a.0 b.0 c.0 d.0
mirror ax2: (EQ (ROOT t1.0 t2.0) (ROOT (MIR t2.0) (MIR t1.0)))
mirror ax2: (EQ (ROOT t1.0 t2.0) (ROOT (MIR t2.0) (MIR t1.0)))
mirror ax2: (EQ (ROOT t1.0 t2.0) (ROOT (MIR t2.0) (MIR t1.0)))
mirror ax2: (EQ (ROOT t1.0 t2.0) (ROOT (MIR t2.0) (MIR t1.0)))
mirror ax2: (EQ (ROOT t1.0 t2.0) (ROOT (MIR t2.0) (MIR t1.0)))
sym: (ROOT (MIR t2.0) (MIR t1.0)) (ROOT x.5 y.4)
trees: (MIR t2.0) (MIR t1.0) c.0 d.0
sym: x.5 (ROOT x.5 y.4)
sym: y.4 (ROOT x.5 y.4)
mirror: a.0 (ROOT x.5 y.4)
sym: (ROOT x.5 y.4) (MIR c.0)
trees: a.0 b.0 c.0 d.0
sym: a.0 x.5
mirror ax2: (EQ (ROOT t1.0 t2.0) (ROOT (MIR t2.0) (MIR t1.0)))
mirror: a.0 x.5
sym: a.0 y.4
mirror ax2: (EQ (ROOT t1.0 t2.0) (ROOT (MIR t2.0) (MIR t1.0)))
mirror: a.0 y.4
sym: (ROOT (MIR (MIR x.5)) (MIR (MIR y.4))) (MIR (MIR (ROOT x.5 y.4)))
sym: (ROOT x.5 y.4) b.0
trees: x.5 y.4 c.0 d.0
sym: (ROOT x.5 y.4) b.0
sym: a.0 (ROOT x.5 y.4)
mirror ax2: (EQ (ROOT t1.0 t2.0) (ROOT (MIR t2.0) (MIR t1.0)))
mirror: a.0 (ROOT x.5 y.4)
trees: a.0 b.0 c.0 d.0
sym: (ROOT x.5 y.4) b.0
trees: x.5 y.4 c.0 d.0
trees: x.5 y.4 c.0 d.0
sym: x.5 b.0
trees: x.5 y.4 c.0 d.0
sym: y.4 b.0
sym: x.5 b.0
sym: a.0 x.5
mirror ax2: (EQ (ROOT t1.0 t2.0) (ROOT (MIR t2.0) (MIR t1.0)))
mirror: a.0 x.5
sym: y.4 b.0
sym: x.5 b.0
sym: (MIR (MIR (ROOT x.5 y.4))) (ROOT x.5 y.4)
sym: (ROOT x.5 y.4) (MIR (MIR (ROOT x.5 y.4)))
sym: (MIR (MIR (ROOT x.5 y.4))) (ROOT x.5 y.4)
mirror: (MIR (ROOT x.5 y.4)) (ROOT x.5 y.4)
sym: (ROOT x.5 y.4) b.0
trees: x.5 y.4 c.0 d.0
```

```
sym: (MIR (MIR (ROOT x.5 y.4))) (ROOT x.5 y.4)
mirror: (MIR (ROOT x.5 y.4)) (ROOT x.5 y.4)
sym: (MIR (MIR (ROOT x.5 y.4))) b.0
sym: a.0 (MIR (MIR (ROOT x.5 y.4)))
mirror ax2: (EQ (ROOT t1.0 t2.0) (ROOT (MIR t2.0) (MIR t1.0)))
mirror: a.0 (MIR (MIR (ROOT x.5 y.4)))
sym: (MIR (MIR (ROOT x.5 y.4))) b.0
mirror: (MIR (ROOT x.5 y.4)) b.0
mirror: (MIR (ROOT x.5 y.4)) b.0
mirror ax2: (EQ (ROOT x.5 y.4) (ROOT (MIR y.4) (MIR x.5)))
sym: a.0 (MIR c.0)
mirror ax2: (EQ (ROOT t1.0 t2.0) (ROOT (MIR t2.0) (MIR t1.0)))
mirror ax2: (EQ (ROOT x.5 y.4) (ROOT (MIR y.4) (MIR x.5)))
mirror ax2: (EQ (ROOT t1.0 t2.0) (ROOT (MIR t2.0) (MIR t1.0)))
mirror: a.0 (MIR c.0)
mirror: (MIR (ROOT x.5 y.4)) (ROOT x.5 y.4)
sym: (ROOT x.5 y.4) (MIR c.0)
sym: (MIR c.0) (ROOT x.5 y.4)
mirror ax2: (EQ (ROOT t1.0 t2.0) (ROOT (MIR t2.0) (MIR t1.0)))
mirror: a.0 (ROOT x.5 y.4)
sym: (ROOT x.5 y.4) b.0
trees: x.5 y.4 c.0 d.0
sym: (MIR (MIR (ROOT x.5 y.4))) b.0
sym: (MIR (MIR (ROOT x.5 y.4))) (ROOT x.5 y.4)
sym: (ROOT x.5 y.4) (MIR (MIR (ROOT x.5 y.4)))
sym: (MIR (MIR (ROOT x.5 y.4))) b.0
mirror: (MIR (ROOT x.5 y.4)) b.0
mirror: (MIR (ROOT x.5 y.4)) (ROOT x.5 y.4)
sym: (ROOT x.5 y.4) (MIR c.0)
sym: a.0 (MIR (MIR (ROOT x.5 y.4)))
sym: (MIR (MIR (ROOT x.5 y.4))) (ROOT x.5 y.4)
mirror: (MIR (ROOT x.5 y.4)) (ROOT x.5 y.4)
sym: (MIR (MIR (ROOT x.5 y.4))) b.0
sym: a.0 (MIR (MIR (ROOT x.5 y.4)))
mirror ax2: (EQ (ROOT t1.0 t2.0) (ROOT (MIR t2.0) (MIR t1.0)))
mirror: a.0 (MIR (MIR (ROOT x.5 y.4)))
sym: (MIR (MIR (ROOT x.5 y.4))) b.0
mirror: (MIR (ROOT x.5 y.4)) b.0
mirror: (MIR (ROOT x.5 y.4)) b.0
mirror ax2: (EQ (ROOT x.5 y.4) (ROOT (MIR y.4) (MIR x.5)))
sym: a.0 (MIR c.0)
mirror ax2: (EQ (ROOT t1.0 t2.0) (ROOT (MIR t2.0) (MIR t1.0)))
mirror ax2: (EQ (ROOT x.5 y.4) (ROOT (MIR y.4) (MIR x.5)))
mirror ax2: (EQ (ROOT t1.0 t2.0) (ROOT (MIR t2.0) (MIR t1.0)))
mirror: a.0 (MIR c.0)
```

```
sym: val.0 (MIR (MIR (ROOT x.5 y.4)))
mirror ax2: (EQ (ROOT t1.0 t2.0) (ROOT (MIR t2.0) (MIR t1.0)))
mirror: a.0 (MIR (MIR (ROOT x.5 y.4)))
sym: a.0 b.0
sym: a.0 b.0
mirror ax2: (EQ (ROOT t1.0 t2.0) (ROOT (MIR t2.0) (MIR t1.0)))
mirror ax2: (EQ (ROOT t1.0 t2.0) (ROOT (MIR t2.0) (MIR t1.0)))
mirror: a.0 b.0
trees: a.0 b.0 c.0 d.0
mirror ax2: (EQ (ROOT t1.0 t2.0) (ROOT (MIR t2.0) (MIR t1.0)))
mirror ax2: (EQ (ROOT t1.0 t2.0) (ROOT (MIR t2.0) (MIR t1.0)))
mirror ax2: (EQ (ROOT t1.0 t2.0) (ROOT (MIR t2.0) (MIR t1.0)))
mirror ax2: (EQ (ROOT t1.0 t2.0) (ROOT (MIR t2.0) (MIR t1.0)))
mirror ax2: (EQ (ROOT t1.0 t2.0) (ROOT (MIR t2.0) (MIR t1.0)))
sym: (MIR (ROOT t1.0 t2.0)) (MIR (MIR (ROOT x.5 y.4)))
mirror ax2: (EQ (ROOT t1.0 t2.0) (ROOT (MIR t2.0) (MIR t1.0)))
mirror: (ROOT t1.0 t2.0) (MIR (MIR (ROOT x.5 y.4)))
sym: (MIR (MIR x.5)) (MIR (MIR (ROOT x.5 y.4)))
mirror: (MIR x.5) (MIR (MIR (ROOT x.5 y.4)))
sym: (MIR (MIR y.4)) (MIR (MIR (ROOT x.5 y.4)))
mirror: (MIR y.4) (MIR (MIR (ROOT x.5 y.4)))
mirror ax2: (EQ (ROOT t1.0 t2.0) (ROOT (MIR t2.0) (MIR t1.0)))
sym: a.0 b.0
mirror ax2: (EQ (ROOT t1.0 t2.0) (ROOT (MIR t2.0) (MIR t1.0)))
mirror ax2: (EQ (ROOT t1.0 t2.0) (ROOT (MIR t2.0) (MIR t1.0)))
mirror ax2: (EQ (ROOT t1.0 t2.0) (ROOT (MIR t2.0) (MIR t1.0)))
mirror: a.0 b.0
trees: a.0 b.0 c.0 d.0
mirror ax2: (EQ (ROOT t1.0 t2.0) (ROOT (MIR t2.0) (MIR t1.0)))
mirror ax2: (EQ (ROOT t1.0 t2.0) (ROOT (MIR t2.0) (MIR t1.0)))
mirror ax2: (EQ (ROOT t1.0 t2.0) (ROOT (MIR t2.0) (MIR t1.0)))
mirror ax2: (EQ (ROOT t1.0 t2.0) (ROOT (MIR t2.0) (MIR t1.0)))
mirror ax2: (EQ (ROOT t1.0 t2.0) (ROOT (MIR t2.0) (MIR t1.0)))
sym: (ROOT (MIR t2.0) (MIR t1.0)) b.0
mirror ax2: (EQ (ROOT t1.0 t2.0) (ROOT (MIR t2.0) (MIR t1.0)))
trees: (MIR t2.0) (MIR t1.0) c.0 d.0
mirror ax2: (EQ (ROOT t1.0 t2.0) (ROOT (MIR t2.0) (MIR t1.0)))
mirror ax2: (EQ (ROOT t1.0 t2.0) (ROOT (MIR t2.0) (MIR t1.0)))
mirror ax2: (EQ (ROOT t1.0 t2.0) (ROOT (MIR t2.0) (MIR t1.0)))
mirror ax2: (EQ (ROOT t1.0 t2.0) (ROOT (MIR t2.0) (MIR t1.0)))
sym: x.5 b.0
sym: y.4 b.0
mirror: a.0 b.0
mirror ax2: (EQ (ROOT t1.0 t2.0) (ROOT (MIR t2.0) (MIR t1.0)))
sym: a.0 (MIR c.0)
```

```
mirror ax2: (EQ (ROOT t1.0 t2.0) (ROOT (MIR t2.0) (MIR t1.0)))
mirror ax2: (EQ (ROOT t1.0 t2.0) (ROOT (MIR t2.0) (MIR t1.0)))
mirror ax2: (EQ (ROOT t1.0 t2.0) (ROOT (MIR t2.0) (MIR t1.0)))
mirror: a.0 (MIR c.0)
trees: a.0 b.0 c.0 d.0
mirror ax2: (EQ (ROOT t1.0 t2.0) (ROOT (MIR t2.0) (MIR t1.0)))
sym: a.0 b.0
mirror ax2: (EQ (ROOT t1.0 t2.0) (ROOT (MIR t2.0) (MIR t1.0)))
mirror ax2: (EQ (ROOT t1.0 t2.0) (ROOT (MIR t2.0) (MIR t1.0)))
mirror ax2: (EQ (ROOT t1.0 t2.0) (ROOT (MIR t2.0) (MIR t1.0)))
mirror ax2: (EQ (ROOT t1.0 t2.0) (ROOT (MIR t2.0) (MIR t1.0)))
mirror ax2: (EQ (ROOT t1.0 t2.0) (ROOT (MIR t2.0) (MIR t1.0)))
mirror: a.0 b.0
trees: a.0 b.0 c.0 d.0
mirror ax2: (EQ (ROOT t1.0 t2.0) (ROOT (MIR t2.0) (MIR t1.0)))
mirror ax2: (EQ (ROOT t1.0 t2.0) (ROOT (MIR t2.0) (MIR t1.0)))
mirror ax2: (EQ (ROOT t1.0 t2.0) (ROOT (MIR t2.0) (MIR t1.0)))
mirror ax2: (EQ (ROOT t1.0 t2.0) (ROOT (MIR t2.0) (MIR t1.0)))
mirror ax2: (EQ (ROOT t1.0 t2.0) (ROOT (MIR t2.0) (MIR t1.0)))
mirror ax2: (EQ (ROOT t1.0 t2.0) (ROOT (MIR t2.0) (MIR t1.0)))
mirror ax2: (EQ (ROOT t1.0 t2.0) (ROOT (MIR t2.0) (MIR t1.0)))
mirror ax2: (EQ (ROOT t1.0 t2.0) (ROOT (MIR t2.0) (MIR t1.0)))
mirror ax2: (EQ (ROOT t1.0 t2.0) (ROOT (MIR t2.0) (MIR t1.0)))
mirror ax2: (EQ (ROOT t1.0 t2.0) (ROOT (MIR t2.0) (MIR t1.0)))
mirror ax2: (EQ (ROOT t1.0 t2.0) (ROOT (MIR t2.0) (MIR t1.0)))
mirror ax2: (EQ (ROOT t1.0 t2.0) (ROOT (MIR t2.0) (MIR t1.0)))
mirror ax2: (EQ (ROOT t1.0 t2.0) (ROOT (MIR t2.0) (MIR t1.0)))
mirror ax2: (EQ (ROOT t1.0 t2.0) (ROOT (MIR t2.0) (MIR t1.0)))
sym: a.0 b.0
mirror ax2: (EQ (ROOT t1.0 t2.0) (ROOT (MIR t2.0) (MIR t1.0)))
mirror ax2: (EQ (ROOT t1.0 t2.0) (ROOT (MIR t2.0) (MIR t1.0)))
mirror: a.0 b.0
trees: a.0 b.0 c.0 d.0
mirror ax2: (EQ (ROOT t1.0 t2.0) (ROOT (MIR t2.0) (MIR t1.0)))
mirror ax2: (EQ (ROOT t1.0 t2.0) (ROOT (MIR t2.0) (MIR t1.0)))
mirror ax2: (EQ (ROOT t1.0 t2.0) (ROOT (MIR t2.0) (MIR t1.0)))
mirror ax2: (EQ (ROOT t1.0 t2.0) (ROOT (MIR t2.0) (MIR t1.0)))
mirror ax2: (EQ (ROOT t1.0 t2.0) (ROOT (MIR t2.0) (MIR t1.0)))
sym: (ROOT (ROOT (MIR t2.0) (MIR t1.0))
           (ROOT (MIR t2.1) (MIR t1.1)))
     (MIR (MIR (ROOT x.5 y.4)))
sym: (ROOT (ROOT (MIR t2.0) (MIR t1.0)) x.5) (MIR (MIR (ROOT x.5 y.4)))
sym: (ROOT (ROOT (MIR t2.0) (MIR t1.0)) y.4) (MIR (MIR (ROOT x.5 y.4)))
sym: a.0 b.0
mirror ax2: (EQ (ROOT t1.0 t2.0) (ROOT (MIR t2.0) (MIR t1.0)))
```

```
mirror ax2: (EQ (ROOT t1.0 t2.0) (ROOT (MIR t2.0) (MIR t1.0)))
mirror: a.0 b.0
trees: a.0 b.0 c.0 d.0
mirror ax2: (EQ (ROOT t1.0 t2.0) (ROOT (MIR t2.0) (MIR t1.0)))
mirror ax2: (EQ (ROOT t1.0 t2.0) (ROOT (MIR t2.0) (MIR t1.0)))
mirror ax2: (EQ (ROOT t1.0 t2.0) (ROOT (MIR t2.0) (MIR t1.0)))
mirror ax2: (EQ (ROOT t1.0 t2.0) (ROOT (MIR t2.0) (MIR t1.0)))
mirror ax2: (EQ (ROOT t1.0 t2.0) (ROOT (MIR t2.0) (MIR t1.0)))
sym: (ROOT x.5 (ROOT (MIR t2.0) (MIR t1.0))) (MIR (MIR (ROOT x.5 y.4)))
sym: (ROOT x.5 x.5) (MIR (MIR (ROOT x.5 y.4)))
sym: (ROOT x.5 y.4) (MIR (MIR (ROOT x.5 y.4)))
sym: a.0 b.0
mirror ax2: (EQ (ROOT t1.0 t2.0) (ROOT (MIR t2.0) (MIR t1.0)))
mirror ax2: (EQ (ROOT t1.0 t2.0) (ROOT (MIR t2.0) (MIR t1.0)))
mirror: a.0 b.0
trees: a.0 b.0 c.0 d.0
mirror ax2: (EQ (ROOT t1.0 t2.0) (ROOT (MIR t2.0) (MIR t1.0)))
mirror ax2: (EQ (ROOT t1.0 t2.0) (ROOT (MIR t2.0) (MIR t1.0)))
mirror ax2: (EQ (ROOT t1.0 t2.0) (ROOT (MIR t2.0) (MIR t1.0)))
mirror ax2: (EQ (ROOT t1.0 t2.0) (ROOT (MIR t2.0) (MIR t1.0)))
mirror ax2: (EQ (ROOT t1.0 t2.0) (ROOT (MIR t2.0) (MIR t1.0)))
sym: (ROOT y.4 (ROOT (MIR t2.0) (MIR t1.0))) (MIR (MIR (ROOT x.5 y.4)))
sym: (ROOT y.4 x.5) (MIR (MIR (ROOT x.5 y.4)))
sym: (ROOT y.4 y.4) (MIR (MIR (ROOT x.5 y.4)))
mirror ax2: (EQ (ROOT t1.0 t2.0) (ROOT (MIR t2.0) (MIR t1.0)))
sym: (ROOT (MIR t2.0) (MIR t1.0)) (MIR (MIR (ROOT x.5 y.4)))
sym: (MIR (MIR (ROOT x.5 y.4))) (ROOT (MIR t2.0) (MIR t1.0))
mirror: (MIR (ROOT x.5 y.4)) (ROOT (MIR t2.0) (MIR t1.0))
sym: (ROOT (MIR t2.0) (MIR t1.0)) b.0
mirror ax2: (EQ (ROOT t1.0 t2.0) (ROOT (MIR t2.0) (MIR t1.0)))
trees: (MIR t2.0) (MIR t1.0) c.0 d.0
mirror ax2: (EQ (ROOT t1.0 t2.0) (ROOT (MIR t2.0) (MIR t1.0)))
mirror ax2: (EQ (ROOT t1.0 t2.0) (ROOT (MIR t2.0) (MIR t1.0)))
mirror ax2: (EQ (ROOT t1.0 t2.0) (ROOT (MIR t2.0) (MIR t1.0)))
mirror ax2: (EQ (ROOT t1.0 t2.0) (ROOT (MIR t2.0) (MIR t1.0)))
sym: x.5 (MIR (MIR (ROOT x.5 y.4)))
sym: (MIR (MIR (ROOT x.5 y.4))) x.5
mirror: (MIR (ROOT x.5 y.4)) x.5
sym: x.5 b.0
sym: y.4 (MIR (MIR (ROOT x.5 y.4)))
sym: (MIR (MIR (ROOT x.5 y.4))) y.4
mirror: (MIR (ROOT x.5 y.4)) y.4
sym: y.4 b.0
mirror: a.0 (MIR (MIR (ROOT x.5 y.4)))
sym: a.0 (MIR (ROOT x.5 y.4))
```

```
mirror ax2: (EQ (ROOT x.5 y.4) (ROOT (MIR y.4) (MIR x.5)))
mirror ax2: (EQ (ROOT t1.0 t2.0) (ROOT (MIR t2.0) (MIR t1.0)))
mirror: a.0 (MIR (ROOT x.5 y.4))
sym: (MIR (MIR (ROOT x.5 y.4))) (MIR c.0)
sym: (MIR c.0) (MIR (MIR (ROOT x.5 y.4)))
mirror ax2: (EQ (ROOT t1.0 t2.0) (ROOT (MIR t2.0) (MIR t1.0)))
mirror: a.0 (MIR (MIR (ROOT x.5 y.4)))
sym: (MIR (MIR (ROOT x.5 y.4))) b.0
mirror: (MIR (ROOT x.5 y.4)) b.0
mirror: (MIR (ROOT x.5 y.4)) (MIR c.0)
mirror ax2: (EQ (ROOT x.5 y.4) (ROOT (MIR y.4) (MIR x.5)))
mirror ax2: (EQ (ROOT x.5 y.4) (ROOT (MIR y.4) (MIR x.5)))
sym: (MIR c.0) (MIR c.0)
mirror ax2: (EQ (ROOT t1.0 t2.0) (ROOT (MIR t2.0) (MIR t1.0)))
mirror: a.0 (MIR c.0)
sym: (MIR (MIR (ROOT x.5 y.4))) (ROOT x.5 y.4)
mirror: (MIR (ROOT x.5 y.4)) (ROOT x.5 y.4)
sym: (MIR (MIR (ROOT x.5 y.4))) b.0
sym: a.0 (MIR (MIR (ROOT x.5 y.4)))
mirror ax2: (EQ (ROOT t1.0 t2.0) (ROOT (MIR t2.0) (MIR t1.0)))
mirror: a.0 (MIR (MIR (ROOT x.5 y.4)))
sym: (MIR (MIR (ROOT x.5 y.4))) b.0
mirror: (MIR (ROOT x.5 y.4)) b.0
mirror: (MIR (ROOT x.5 y.4)) b.0
mirror ax2: (EQ (ROOT x.5 y.4) (ROOT (MIR y.4) (MIR x.5)))
sym: a.0 (MIR c.0)
mirror ax2: (EQ (ROOT t1.0 t2.0) (ROOT (MIR t2.0) (MIR t1.0)))
mirror ax2: (EQ (ROOT x.5 y.4) (ROOT (MIR y.4) (MIR x.5)))
mirror ax2: (EQ (ROOT t1.0 t2.0) (ROOT (MIR t2.0) (MIR t1.0)))
mirror: a.0 (MIR c.0)
sym: (MIR (MIR (ROOT x.5 y.4))) b.0
sym: (MIR (MIR (ROOT x.5 y.4))) b.0
mirror: (MIR (ROOT x.5 y.4)) b.0
sym: a.0 (MIR (MIR (ROOT x.5 y.4)))
sym: (MIR (MIR (ROOT x.5 y.4))) b.0
mirror: (MIR (ROOT x.5 y.4)) b.0
sym: a.0 b.0
mirror ax2: (EQ (ROOT t1.0 t2.0) (ROOT (MIR t2.0) (MIR t1.0)))
mirror ax2: (EQ (ROOT t1.0 t2.0) (ROOT (MIR t2.0) (MIR t1.0)))
mirror: a.0 b.0
trees: a.0 b.0 c.0 d.0
mirror ax2: (EQ (ROOT t1.0 t2.0) (ROOT (MIR t2.0) (MIR t1.0)))
mirror ax2: (EQ (ROOT t1.0 t2.0) (ROOT (MIR t2.0) (MIR t1.0)))
mirror ax2: (EQ (ROOT t1.0 t2.0) (ROOT (MIR t2.0) (MIR t1.0)))
mirror ax2: (EQ (ROOT t1.0 t2.0) (ROOT (MIR t2.0) (MIR t1.0)))
```

```
mirror ax2: (EQ (ROOT t1.0 t2.0) (ROOT (MIR t2.0) (MIR t1.0)))
sym: (ROOT (MIR t2.0) (MIR t1.0)) (MIR (MIR (ROOT x.5 y.4)))
sym: x.5 (MIR (MIR (ROOT x.5 y.4)))
sym: y.4 (MIR (MIR (ROOT x.5 y.4)))
mirror: a.0 (MIR (MIR (ROOT x.5 y.4)))
sym: (MIR (MIR (ROOT x.5 y.4))) (MIR c.0)
mirror: (MIR (ROOT x.5 y.4)) (MIR c.0)
sym: (MIR (MIR (ROOT x.5 y.4))) b.0
mirror: (MIR (ROOT x.5 y.4)) b.0
sym: (MIR (MIR (ROOT x.5 y.4))) b.0
sym: a.0 (MIR (MIR (ROOT x.5 y.4)))
mirror ax2: (EQ (ROOT t1.0 t2.0) (ROOT (MIR t2.0) (MIR t1.0)))
mirror: a.0 (MIR (MIR (ROOT x.5 y.4)))
sym: (MIR (MIR (ROOT x.5 y.4))) b.0
mirror: (MIR (ROOT x.5 y.4)) b.0
mirror: (MIR (ROOT x.5 y.4)) b.0
mirror ax2: (EQ (ROOT x.5 y.4) (ROOT (MIR y.4) (MIR x.5)))
mirror ax2: (EQ (ROOT (MIR y.4) (MIR x.5)) (ROOT (MIR (MIR x.5)) (MIR (MIR y.4))))
sym: a.0 (MIR c.0)
mirror ax2: (EQ (ROOT t1.0 t2.0) (ROOT (MIR t2.0) (MIR t1.0)))
mirror ax2: (EQ (ROOT x.5 y.4) (ROOT (MIR y.4) (MIR x.5)))
mirror ax2: (EQ (ROOT t1.0 t2.0) (ROOT (MIR t2.0) (MIR t1.0)))
mirror: a.0 (MIR c.0)
mirror: (MIR (ROOT x.5 y.4)) b.0
sym: (MIR (ROOT x.5 y.4)) b.0
mirror ax2: (EQ (ROOT x.5 y.4) (ROOT (MIR y.4) (MIR x.5)))
mirror: (ROOT x.5 y.4) b.0
mirror ax2: (EQ (ROOT x.5 y.4) (ROOT (MIR y.4) (MIR x.5)))
sym: (MIR (ROOT (MIR y.4) (MIR x.5))) b.0
mirror ax2: (EQ (ROOT (MIR y.4) (MIR x.5)) (ROOT (MIR (MIR x.5)) (MIR (MIR y.4))))
mirror: (ROOT (MIR y.4) (MIR x.5)) b.0
mirror ax2: (EQ (ROOT (MIR y.4) (MIR x.5)) (ROOT (MIR (MIR x.5)) (MIR (MIR y.4))))
sym: (ROOT (MIR (MIR x.5)) (MIR (MIR y.4))) (ROOT x.5 y.4)
trees: (MIR (MIR x.5)) (MIR (MIR y.4)) c.0 d.0

([*depth-counter-var*.0 . 5]
 [a*.0 . (MIR a*.1)]
 [a*.2 . (MIR a*.0)]
 [a*.3 . (MIR a*.4)]
 [a*.5 . (MIR a*.3)]
 [a*.4 . x.5]
 [a*.6 . (MIR a*.4)]
 [a*.1 . y.4]
 [a*.7 . (MIR a*.1)]
 [a*.8 . (ROOT a*.7 a*.6)]
```

```
[*depth-counter-var*.0 . 4]
[*depth-counter-var*.0 . 3]
[*depth-counter-var*.0 . 2]
[a*.9 . (ROOT x.5 y.4)]
[a*.10 . (MIR a*.9)]
[*depth-counter-var*.0 . 1]
[*depth-counter-var*.0 . 0])
```

8 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 of 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. 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 always be `fk`. This would mean that there could be just one argument instead of three. That one argument would be a stream of say, *package*s, where a

package contains everything but `fk`. Then, each package, which now contains `sk` and `subst` could, in turn, be treated as a stream of *pack*s. A pack could be implemented with `cons`. Now that these three 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 three 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!`, `all!!`, and `solve`.

9 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 more direct 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.