

# TUTORIAL

## *A “Poor Man’s” Logic System with First-Class Relations (Draft: January 25, 2003)*

Daniel P. Friedman †

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

### 1 Introduction

We present an implementation of an embedding in R<sup>5</sup> 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 can make them extensible, which means that we give ourselves the power of sharing subrelations, much as object-oriented programming shares behavior. There is, however, the matter of getting *the cut* to work in the extensible relations. It is tricky enough to get the cut to work with a sequence of rules defining a relation. Once multiple relations are included in one relation, then we must wire that knowledge into the extended relation without losing the individuality of the independent relations.

In order to make this approach work, five properties that are not normally associated with the discussion of logic programming had to become apparent. The first of these properties is that sequences of rules or sequences of antecedents can be handled as macros *after* the logic system has been built.

† ...

The second property is that most of the time we needn't think about the rules as a monolithic global list of rules. In fact, in most circumstances, the programmer knows exactly which set of rules to home in on and we are going to require that knowledge of the programmer.<sup>†</sup> For example, the rules that infer a type for a programming language are significantly different from the rules that determine how to concatenate two lists, and both of these are considerably different from the rules that determine 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 sets of rules leads to the removal of a search (possibly with a hash table) of a table of rules. Removing this search leads to a much cleaner implementation of a logic system, thus making the smooth integration with Scheme possible. Naturally, there are still searches roaming around in the system, but they are the responsibility of the user who makes calls in a set of mutually recursive functions, something that Scheme already supports.

The third 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 processing 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 ask for more answers (We use streams.). Second, in order to display answers that contain the same logic variable more than once, we need a recursive copier that keeps track of the logic variables. But, this is about displaying results or passing them to other random Scheme functions. Finally, in the last section we introduce pairs (lists) as valid terms, requiring the revision of two functions, both of which become recursive.

The fourth property is that no gratuitous lists should ever be built. This matters because we treat relations as finite-arity functions and thus we can consider the actual arguments separately instead of joined together as a list. The implications of this decision are far reaching, since it allows us to avoid many applications of substitutions. Where there would be two lists to unify, we now have two sequences of arguments, which means that we can avoid substituting in each `cdr` in each list. (See the redefinition of `unify*` in the last section.) Also, if our raw data does not contain lists, then our unifier does not need to know about them.

The fifth property is that since relations are first-class, they should support a relation extension operation. This operation takes an arbitrary number of relations, all with the same signature. Although we start the development in a more conventional way treating a relation as a sequence of rules, when we present relation extension, we see that we could have treated rules as a sequence of self-contained relations. We could have avoided any discussion of dispatching over the rules, since there would always be just one rule. With this, we get the ability to share relations in the fashion of object-oriented programming. We show some of the power of relation extension later in the paper when we build a type inference system with it.

<sup>†</sup> This restriction also appears in Silvija Seres and Michael Spivey, "Embedding Prolog in Haskell". In *Haskell Workshop*, Paris, France, September, 1999.

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, macros, 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 *logic* variables, substitutions, and unification. In section 3 we implement and demonstrate the unsullied logic system. In section 4 we explore sullied operators, such as the cut operator, and their implementation. Next, we utilize some recursive rules and include a comparison with an embedding in Haskell. We follow that by a section discussing three famous logic programming problems: `append`, type inference, and a generalization of Prolog's name. We conclude with some perspective on why this approach works.

## 2 Preliminaries

### 2.1 Concretization, substitution, and unification

There are three categories of functions that must be understood. The first category is that of concretization of special data structures, called *terms*. The second category relates to substitutions, and the last category is unification. The only operations from here that get exported to the main body of the paper are a term copier `copy-term`, a raw term generator `concretize`, substitution operators `empty-subst`, `compose-subst`, and `subst-in`, a binder of newly-created logic variables `exists`, a curried *lambda* `lambda@`, a curried application `@`, a macro for counting arguments `arity`, and two unifier interfaces `unify` and `unify-incrementally`.

### 2.2 Concretizing data structures

A data structure that may be a *logic* variable or contain such variables is called a *term*. To concretize a term is to replace a term's variables by symbols representing the variables. The primary purpose of concretization is for processing outside of the logic system, such as displaying, etc. Since a term may contain different variables built from the same name, it is necessary to distinguish one from the other. Before we look at an example, we need to know a bit about such variables. We assume that `(lv 'x)` creates a variable that responds like this:

```
> (let ([x (lv 'x)] [xx (lv 'x)])
      (list (lv-name x) (lv-name xx) (lv? x) (eqv? x xx)))
(x x #t #f)
```

There is a procedure `lv-name` that retrieves the name that a variable has been built from, there is a predicate `lv?` that distinguishes variables from other data

structures, and each creation of a variable makes a new one. (We leave the details of defining these three procedures as an exercise. Certainly, a uniquely tagged cons pair with its name in the `cdr` could be used, but most Scheme systems have their own record facility. For purists, instead of relying on the ability to distinguish two cons cells with `eqv?`, one could associate a unique timestamp with each logic variable.)

Let's look at some examples of `concretize` that demonstrate why implementing it is a somewhat interesting, though quite a straightforward problem.

```
> (concretize
   (let ([x (lv 'x)] [xx (lv 'x)] [xyz (lv 'x)])
     (list x xx xyz x xx xyz xx x xyz x)))
(x.1 x.2 x.3 x.1 x.2 x.3 x.2 x.1 x.3 x.1)
```

Each variable gets its own artificial name: `x` becomes `x.1`, `xx` becomes `x.2`, and `xyz` becomes `x.3`. Anything that is not a variable stays the same.

```
> (concretize
   (let ([x (lv 'x)] [xx (lv 'x)] [xyz (lv 'x)])
     (list x 7 xx 9 xyz x xx (list xyz xx x) xyz 'x x)))
(x.1 7 x.2 9 x.3 x.1 x.2 (x.3 x.2 x.1) x.3 x x.1)
```

Finally, we might have variables with different names.

```
> (concretize
   (let ([x (lv 'x)] [y (lv 'y)] [xx (lv 'x)] [yy (lv 'y)])
     (list x 7 'y xx 9 y xx (list yy x) 'x x)))
(x.1 7 y x.2 9 y.1 x.2 (y.2 x.1) x x.1)
```

One easy solution to this problem would be to simply associate a `gensym` with each variable, then use a table to look up the individual variables. This is essentially what we have done, except that we want to have the artificial names created from `lv-name` and to keep the indices as small as possible, so we have a slightly more complicated problem. Here is the code for `concretize`, along with three trivial auxiliary functions, `copy-term`, `artificial-name`, and `assv/lv-name`.<sup>†</sup>

```
(define artificial-name
  (lambda (name c)
    (string->symbol
     (string-append (symbol->string name) "." (number->string c)))))

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

<sup>†</sup> The long underscore appears below code that is part of the full system.

```

(define copy-term
  (lambda (var-fn)
    (letrec
      ([copy-term
        (lambda (t env k)
          (cond
            [(lv? t) (var-fn t env k)]
            [(pair? t)
             (copy-term (car t) env
                        (lambda (a-t env)
                          (copy-term (cdr t) env
                                      (lambda (d-t env)
                                        (k (cons a-t d-t) env))))))]
            [else (k t env)]))]
      (lambda (term)
        (copy-term term '() (lambda (t env) t))))))

(define concretize
  (copy-term
   (lambda (t env k)
     (cond
      [(assv t env)
       => (lambda (pr)
            (k (cadr pr) env))]
      [else (let ([tname (lv-name t)])
              (let ([c (let ([pr (assv/lv-name tname env)])
                          (+ (if (not pr) 0 (caddr pr)) 1))]
                    (let ([name (artificial-name tname c)])
                      (k name (cons (cons t (cons name c)) env))))))]
            ])))))

```

---

A term can be a variable, a pair, which can contain variables, or any other value. Because of this decision, we check for variables, then for pairs before treating the fall through case. This is a bit clumsy, but it frees us from writing a constant? predicate, which is difficult to make robust.

There are basically three cases to consider in `concretize`. First, if we have already seen a particular variable, we simply take the artificial one that we stored in the environment. Second, if we do not find it in the environment, but we find one with the same name, then we use that name to construct the next artificial one, keeping it and an index in the environment's association. Because we know that the last one to enter the environment has the highest index for that name, we are guaranteed that incrementing that index produces an unused artificial name. Finally, we have the case that no artificial name exists for this variable, so we start the index at 1.

Each invocation of `copy-term` requires a procedure that determines what action to take for replacing variables. The pair and fall through cases in `copy-term` are

obvious, but we must remember to thread the environment in the recursive cases. The procedures `artificial-name` and `assv/lv-name` should be clear from their definitions.

The procedure `concretize` is not important, since it only translates one representation into another, but we need it to demonstrate certain kinds of answers in the logic system that would be difficult to express without it. Virtually all of what we need to know about the logic system can be understood without any appreciation of the implementation details of `concretize`.

### 2.2.1 Creating variables and Currying

We quickly tire of using `let` to bind logic variables. In its place, we propose `exists`.

```
(define-syntax exists
  (syntax-rules ()
    [(_ (name ...) body0 body1 ...)
     (let ([name (lv 'name)] ...) body0 body1 ...) ]))
```

---

For example,

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

expands to (is the same as writing)

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

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

```
> (concretize
  (exists (x y z)
    (list x 1 y 2 z 3)))
(x.1 1 y.1 2 z.1 3)

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

> (expand-only '(exists)
  '(exists (x y z)
    (list x 1 y 2 z 3)))
(let ([x (lv 'x)] [y (lv 'y)] [z (lv 'z)])
  (list x 1 y 2 z 3))
```

There will be other uses of `expand-only` as we get further into the paper. 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 (lv 'x)] [y (lv 'y)] [z (lv 'z)])
    '(,x 1 ,y 2 ,z 3))
> (concretize
    (let ([x (lv 'x)] [y (lv 'y)] [z (lv 'z)])
        '(,x 1 ,y 2 ,z 3)))
(x.1 1 y.1 2 z.1 3)
```

The argument to `concretize` is identical to the list that has been passed in the preceding examples, but we get to use “,” to indicate where the logic 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.

Another way to view `exists` is to construct it with the procedure `there-exists`,

```
(define there-exists
  (lambda (name proc)
    (proc (lv name))))
```

and then write the `exists` macro using it.

```
(define-syntax exists
  (syntax-rules ()
    [(_ () body0 body1 ...) (begin body0 body1 ...)]
    [(_ (name0 name1 ...) body0 body1 ...)
     (there-exists 'name0
                    (lambda (name0)
                      (exists (name1 ...) body0 body1 ...)))]))
```

On a formal level, `there-exists` would not require passing a name. The procedure is indeed all that is needed. We find, however, that the name is indispensable when it comes to observing values that contain variables.

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

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

---

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

```
(define-syntax @
  (syntax-rules ()
    [(_ arg0) arg0]
    [(_ arg0 arg1 arg2 ...) (@ (arg0 arg1) arg2 ...)]))
```

---

```
> (@ (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)
```

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

Taking the length of a list is an expensive operation. When it is possible to determine such lengths statically, we opt to do so, avoiding the expensive run-time operation. Because the *Standard's* macros do not allow computing, we derive an expression that most compilers would compile to a number. If we were to go beyond the *Standard*, then it would be possible for the macro to produce the actual number. Here is the macro `arity`.

```
(define-syntax arity
  (syntax-rules ()
    [(_ x0) 1]
    [(_ x0 ...) (+ (arity x0) ...)]))
```

---

```
> (expand '(arity a b c d e))
(+ 1 1 1 1 1)
```

### 2.3 Substitutions

A substitution, *s*, is a list of real commitments, represented using an association list. A *commitment* is a pairing of a variable to a term and it is *real* if its variable is not the same as its term. The variables (i.e., (map car 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. <sup>§</sup>

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

(define compose-subst
  (lambda (base refining)
    (cond
      [(empty-subst? base) refining]
      [(empty-subst? refining) base]
      [else (refine base refining refining)])))
```

---

In `refine`, below, each term of the base substitution is refined by the refining substitution and re-associated with its variable. If the refined term is the same as the variable, it is not included in the resultant substitution, guaranteeing membership to real commitments. If any variable in the base is in `refining-survivors` (which starts out the same as `refining`) substitution, it is removed from the survivors, since it has already been refined.

```
(define refine
  (lambda (base refining refining-survivors)
    (cond
      [(null? base) refining-survivors]
      [else (cons-if-real-commitment
              (car (car base))
              (nonempty-subst-in (cadr (car base)) refining)
              (refine (cdr base) refining
                    (cond
                     [(assv (car (car base)) refining-survivors)
                      => (lambda (p) (remv p refining-survivors))]
                     [else refining-survivors]))]))]))))
```

---

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

```
(define cons-if-real-commitment
  (lambda (lv term refined)
    (cond
      [(eqv? term lv) refined]
      [else (cons (list lv term) refined)])))
```

---

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

```
(define nonempty-subst-in
  (lambda (t subst)
    (cond
      [(lv? t)
       (cond
         [(assv t subst) => cadr]
         [else t])]
      [else t])))
```

---

The procedure `subst-in`, below, is identical to `nonempty-subst-in`, except that it treats the empty substitution as a special case. We know that if the substitution is empty, then we simply return the term, since we have no hope of replacing any part of the term. Thus, although the tests using `empty-subst?` in `compose-subst` and `subst-in` are not strictly necessary, we have included them since they avoid some wasteful copying of terms *when we enlarge the set of terms in the last section*.

```
(define subst-in
  (lambda (t subst)
    (cond
      [(empty-subst? subst) t]
      [else (nonempty-subst-in t subst)])))
```

---

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. The inner list is used to make the commitment. (Implementing commitments with `cons` would halve the space usage, but it would make the substitutions quite unreadable. Then the occurrences of `cadr` in `nonempty-subst-in` (here and where it is redefined in the last section) and `refine` should be replaced by `cdr`.)

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

```
> (concretize
  (exists (x y)
    (let ([s (compose-subst (unit-subst x y) (unit-subst y 52))])
      (subst-in x s))))
```

52

```
> (concretize
  (exists (w x y)
    (let ([s (compose-subst (unit-subst y w) (unit-subst w 52))])
      (let ([r (compose-subst (unit-subst x y) s)])
        (subst-in x r)))))
```

52

```
> (concretize
  (exists (w x y)
    (let ([s (compose-subst (unit-subst w 52) (unit-subst y w))])
      (let ([r (compose-subst (unit-subst x y) s)])
        (subst-in x r)))))
```

w.1

```

> (concretize
  (exists (x y z)
    (let ([s (compose-subst (unit-subst y z) (unit-subst x y))]
          [r (compose-subst
              (compose-subst (unit-subst x 'a) (unit-subst y 'b))
              (unit-subst z y))])
      (write (concretize s))
      (write (concretize r))
      (compose-subst s r))))
((x.1 y.1) (y.1 z.1))
((x.1 a) (y.1 b) (z.1 y.1))
((x.1 b) (z.1 y.1))

```

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  $r$  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  $r$  is refined by the  $y$ -commitment committing  $x$  to  $w$ , which concretizes to  $w.1$ . 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  $y$ . (Trace `refine` to see this happen.)

## 2.4 Unification

Next, we introduce the unification interfaces, `unify-incrementally` and `unify`. The latter is used only to interface Scheme procedures and the former is used to make the logic system work. Its role is central to understanding how the logic system is packaged. But, first we must understand unification.

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

```

(define _ (exists (_) _))

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

(define unify*
  (lambda (t u)
    (cond
      [(or (eqv? t u) (eqv? t _) (eqv? u _)) empty-subst]
      [(lv? t) (unit-subst t u)]
      [(lv? u) (unit-subst u t)]
      [else #f])))

```

---

Here are a few examples that unify,

```

> (concretize
   (exists (y)
    (unify 4 y empty-subst)))
((y.1 4))

> (concretize
   (exists (x y)
    (unify x y empty-subst)))
((x.1 y.1))

> (unify 'x 'x empty-subst)
()

> (concretize
   (exists (x)
    (unify x x empty-subst)))
()

```

and here are some that fail to unify.

```

> (unify 4 'y empty-subst)
#f

> (unify 'x 3 empty-subst)
#f

> (unify 3 4 empty-subst)
#f

```

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

### 2.5 The heart and soul of the logic system

A way that we can think about unifying two sequences of terms is to break each sequence into its constituent parts and then unify the parts one at a time, being careful to bring each next part up-to-date. We rely on this approach, since it allows us to treat the parts somewhat independently.

```
(define-syntax unify-incrementally
  (syntax-rules ()
    [(_ subst) (lambda (yes fk) (yes subst))]
    [(_ subst t-lexvar0 t-lexvar1 ...)
     (lambda (t)
       (cond
        [(unify t t-lexvar0 subst)
         => (lambda (subst)
              (unify-incrementally subst t-lexvar1 ...))]
        [else (lambda@ (t-lexvar1 ...) (lambda (yes fk) (fk)))]))]))
```

---

```
> (concretize
   (exists (x y b z)
    (let ([s (unit-subst y 3)])
      ((@ (unify-incrementally s x y z) 2 b 4)
       (lambda (subst) subst)
       (lambda () #f)))))
((y.1 3) (x.1 2) (b.1 3) (z.1 4))
```

We partially expand the `exists` expression with respect to `unify-incrementally`, below.

```
(exists (x y b z)
  (let ([s (unit-subst y 3)])
    ((@ (lambda (t)
        (cond
          [(unify t x s)
           => (lambda (s)
                (lambda (t)
                  (cond
                    [(unify t y s)
                     => (lambda (s)
                          (lambda (t)
                            (cond
                              [(unify t z s)
                               => (lambda (s) (lambda (yes fk) (yes s)))]
                              [else (lambda@ () (lambda (yes fk) (fk)))]))]
                            [else (lambda@ (z) (lambda (yes fk) (fk)))]))]
                          (lambda (y z) (lambda (yes fk) (fk)))]))
          2 b 4)
        (lambda (subst) subst)
        (lambda () #f))))))
```

Each `else` clause in the expansion returns a function that consumes fewer arguments than the one below it. Once we fail to unify, we can invoke the failure continuation as soon as we absorb the remaining arguments. Thus, we can avoid applying many substitutions until we are ready to unify and only substitute for those terms that are being unified. Without this macro, we would be forced to make a list out of a fixed number of terms and our current unifier does *not* support lists!

### 3 The unsullied logic system

The logic system has just two primary operators. It needs a macro, `relation`, which is like the `lambda` core form and it needs a macro `to-show`. From `to-show` we derive a macro `fact` and a handler of a sequence of rules, `select`. In addition, we have two derived macros for handling sequences of antecedents: `all` and `any`. To enter the system, we have a procedure `query`, and two interfacing operations: `solve` and `solution`, which is just `solve` limited to one solution. Having presented the basic unsullied logic system, we present some additional derived macros: `extend-relation`, which obviates the need for `select`; and `relation@`, a curried `relation`.

Here are the types we use in this system:

```

      Fk = () -> Ans
    Cutk = Fk
      Ans = Nil + [Subst,Fk]
      Sk = Fk -> Subst -> Cutk -> Ans
    Antecedent = Sk -> Sk
      Rule = [Goal-fn,Int] -> Antecedent

```

Ans is a stream of substitutions, since Fk is a function of zero arguments. An Antecedent is an Sk transformer and Rule is a mapping from a goal function, explained below, and its arity to Antecedent.

### 3.1 It's a small world

Let's define a fact.

```

(define-syntax relation
  (syntax-rules ()
    [(_ (t-lexvar0 ...) body0 body1 ...)
      (lambda (t-lexvar0 ...)
        ((begin body0 body1 ...)
          (lambda (entering-subst)
            (unify-incrementally entering-subst t-lexvar0 ...))
          (arity t-lexvar0 ...)))]))

```

---

The type of relation is Term -> Antecedent. The macro relation is unspectacular, except that it creates a unifying function using unify-incrementally. When such a goal function gets a substitution, it is applied to the terms that the goal expects to unify against. The real work of the relation is in (begin body0 body1 ...), whose last body absorbs a goal function and its associated arity. The preceding bodies are to be treated like the multiple bodies of a lambda expression. This, among other features, gives opportunities for displaying values, which may be very useful while debugging.

```

(define-syntax to-show
  (syntax-rules ()
    [(_ t0 ...)
      (lambda (antecedent)
        (lambda (goal-fn lengoal)
          (lambda@ (sk fk entering-subst cutk)
            (possible-arity-mismatch lengoal t0 ...)
            (@ (goal-fn entering-subst) t0 ...)
            (lambda (exiting-subst)
              (@ antecedent sk fk exiting-subst fk))
            fk))))))

```

---

```
(define-syntax possible-arity-mismatch
  (syntax-rules ()
    [(_ lengoal t0 ...)
     (if (not (= (arity t0 ...) lengoal))
         (error 'to-show "arity of ~s is not ~s."
                (list t0 ...) lengoal))]))
```

---

The type of `to-show` is `Term -> Antecedent -> Rule`. The macro `to-show` tries to unify provided that the number of terms `t0 ...` is the same as `lengoal`. (This directly corresponds to *Error: incorrect number of arguments to a procedure*.) If the terms unify with the goal (i.e., `(@ (goal-fn subst) t0 ...)`), the entering substitution leads to a possibly enlarged exiting substitution. (The expression `(lambda (subst) ...)` is not an `Sk`, since it does not take an `Fk` as its first argument. It is, however, a success continuation called from within `unify-incrementally`.) Then, this substitution is passed to `(antecedent sk)`, along with the two failure continuations. If the first of the two failure continuations is invoked, it fails. The second one become the new cutk, which we can ignore until we look at the sullied antecedents.

```
(define-syntax all
  (syntax-rules ()
    [(_) (lambda (sk) sk)]))
```

```
(define-syntax fact
  (syntax-rules ()
    [(_ t0 ...) ((to-show t0 ...) (all))]))
```

---

The macro `fact` builds an empty `to-show`. This is indicated by passing the identity function, to the expansion of `to-show`. Thus, `antecedent` in this case becomes the identity function and `(antecedent sk)` becomes `sk`. (It may be instructive to rewrite `fact` armed with this knowledge and then rewrite `to-show` in terms of `fact`.) The `antecedent` is what must hold for a fact to hold. When we use the identity, there are no antecedents to worry about. Later, when we redefine the macro `all`, we allow for additional antecedents.

We now have enough tools to define a relation with a single fact. Our relation says that `pete` is the father of `sal`.

```
(define father
  (relation (dad child)
            (fact 'pete 'sal)))
```

It is instructive to see what `father` becomes when we expand parts of `relation`.

```
> (expand-only '(relation to-show fact all)
  '(define father
    (relation (dad child)
              (fact 'pete 'sal))))
```

```
(define father
  (lambda (dad child)
    (((lambda (antecedent)
      (lambda (goal-fn lengoal)
        (lambda@ (sk fk entering-subst cutk)
          (possible-arity-mismatch lengoal 'pete 'sal)
          ((@ (goal-fn entering-subst) 'pete 'sal)
            (lambda (exiting-subst)
              (@ antecedent sk fk exiting-subst cutk))
              fk))))
      (lambda (sk) sk))
     (lambda (entering-subst)
       (unify-incrementally entering-subst dad child))
     (arity dad child))))
```

Whenever `father` is invoked, a `dad` and a `child` get bound to a goal term, which is then transformed into a goal function. A failure continuation is invoked if no fact is matched against the goal function. This effects a return from the call to `father`. Therefore, when it fails to find a match, it returns from the call.

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

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

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

(define child-of-male
  (lambda (child dad)
    (((lambda (antecedent)
      (lambda (goal-fn lengoal)
        (lambda@ (sk fk entering-subst cutk)
          (possible-arity-mismatch lengoal child dad)
          ((@ (goal-fn entering-subst) child dad)
            (lambda (exiting-subst)
              (@ antecedent sk fk exiting-subst cutk))
              fk))))
      (father dad child))
     (lambda (entering-subst)
       (unify-incrementally entering-subst child dad))
     (arity child dad))))
```

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

We are now ready to test `father`. For example, we ask if `pete` is the father of `sal`.

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

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

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

All the arguments (i.e., `initial-sk`, `initial-fk`, and `empty-subst`) to `(father 'pete 'sal)` are constants. Therefore, we abstract this with the function `query`, which packages these initial values with the antecedent, the result of the `father` call.

```
(define query
  (let ([initial-fk (lambda () '())]
        [initial-sk (lambda@ (fk subst cutk) (cons subst fk))])
    (lambda (antecedent)
      (@ antecedent initial-sk initial-fk empty-subst initial-fk))))
```

---

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

```
> (concretize
  (exists (x)
    (query (father 'pete x))))
((x.1 sal)) . #<procedure>

> (concretize
  (exists (x)
    (let ([term (list 'pete x)])
      (let ([answer (query (father 'pete x))])
        (map (lambda (t) (subst-in t (car answer))) term))))))
(pete sal)
```

We see in `((x.1 sal))` that the variable bound to `x`, concretized to `x.1`, has been instantiated to `sal`, and when we use this substitution, the `x` in the value of `(list 'pete x)` gets replaced by `sal`, which leads to `(pete sal)`.

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

```
> (concretize
  (exists (x y)
    (let ([term (list x y)])
      (let ([answer (query (child-of-male x y))])
        (map (lambda (t) (subst-in t (car answer))) term))))))
(sal pete)
```

And, we discover that `sal` is the child of `pete`.

### 3.3 Generating more than one answer

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

Suppose that `pete` is also the father of `pat`, what changes could be made to get both answers? First, we introduce `select` to allow for more than one fact.

```
(define-syntax select
  (syntax-rules ()
    [(_)
     (lambda (goal-fn lengoal)
       (lambda@ (sk fk subst cutk)
         (fk)))]
    [(_ rule) rule]
    [(_ rule0 rule1 ...)
     (lambda (goal-fn lengoal)
       (lambda@ (sk fk subst cutk)
         (@ (rule0 goal-fn lengoal)
            sk
            (lambda ()
              (@ ((select rule1 ...) goal-fn lengoal)
                 sk fk subst cutk)))
         subst
         cutk)))]))
```

---

There are two observations to make about `select`: the top two lines of the first and third clause appear in `to-show` and the second clause is unnecessary. In the top clause, when we have run out of rules, we fail. In the third clause, we try each rule *in order* until one succeeds. The `(lambda () ...)` creates a failure continuation that gets invoked at the bottom of `to-show`, thus building in an opportunity for trying additional rules in the same relation.

```
(define father
  (relation (dad child)
    (select
      (fact 'pete 'sal)
      (fact 'pete 'pat))))

> (concretize
  (exists (x)
    (let ([term (list 'pete x)])
      (let ([answer1 (query (father 'pete x))])
        (list
          (map (lambda (t) (subst-in t (car answer1))) term)
          (let ([answer2 ((cdr answer1))])
            (map (lambda (t) (subst-in t (car answer2))) term)))))))
  ((pete sal) (pete pat))
```

First, we add that `pete` is also the father of `pat`. Then, we invoke the `cdr` of `answer1`, a failure continuation, which continues searching for another answer. Next, consider the following rather long expression, below.

```
> (concretize
  (exists (x)
    (let ([term (list 'pete x)])
      (let ([answer1 (query (father 'pete x))])
        (cons
          (map (lambda (t) (subst-in t (car answer1))) term)
          (let ([answer2 ((cdr answer1))])
            (cons
              (map (lambda (t) (subst-in t (car answer2))) term)
              (let ([answer3 ((cdr answer2))])
                (if (null? answer3)
                    '()
                    (cons
                     (map (lambda (t) (subst-in t (car answer3))) term)
                     '()))))))))))))
```

This expression clarifies that we do not try to get more than three answers, but we might stop at two, if that turns out to be all we find. In other words, after finding two answers, we check to see what happens if we try for a third answer. If we do not find another answer, we quit, otherwise, we include that third answer

and then simply return the three-element list. In this case, we quit with just two values.

### 3.4 Streams (infinite lists) provide a natural interface

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

The `(car answer1)` is a substitution, and the `(cdr answer1)` is a thunk that if invoked returns a value whose `car` is a substitution, and whose `cdr` is a thunk, etc. This is a stream (possibly infinite list) of substitutions. We write the function, `stream-prefix`, which given a bound `n`, and a stream, `strm`, yields a list with at most one element, if `n` is 0; at most two elements, if `n` is 1; etc. But, we must be careful that going for the `n+1`st element of a stream does not lead to an infinite loop, even though we only want the first `n` elements! (Consider what would happen in the code, above, if `((cdr answer2))` fails to terminate. We don't have to know that three elements exist if we only want the first two.) This causes `stream-prefix` to be written in a slightly awkward fashion.

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

---

Once we have a finite list of substitutions, we are free to use `map` and other list processing functions. For example, we define a macro `solve` that takes a positive integer upper bound and a relation call. It returns a list like the one above for `pete's children`. If any of the `ti` are variables, then it will be replaced by its value in each substitution.

```
(define-syntax solve
  (syntax-rules ()
    [(_ n (rel t0 ...))
     (concretize
      (map (lambda (subst)
            (list (subst-in t0 subst) ...))
           (stream-prefix (- n 1) (query (rel t0 ...))))))])
```

---

```
> (exists (x) (solve 5 (father 'pete x)))
```

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. (As an exercise, 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 solution with `solution`,

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

---

As an exercise, try to write `solution` by redefining `initial-sk` and `initial-fk`, instead.

```
> (exists (x) (solution (father 'pete x)))
(pete 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 to give to `stream-prefix`. In those circumstances, it is best to process the stream of substitutions using a hand-crafted procedure.

Occasionally, we want the opportunity to bypass the macro `solution` and use a function over a list.

```
> (let ([t* '(pete ,x)])
      (let ([ans (exists (x) (query (apply father t*)))]
            (if (null? (cdr ans))
                #f
                (concretize
                 (map (lambda (t) (subst-in t (car ans))) t*)))))
    (pete sal)
```

This idiom allows some preprocessing of data prior to entering the logic system. Without it, our reliance on the macro `solution` would force the use of `eval`, which, in our opinion, should not be an option, here. This idiom could be packaged into a procedure `query-apply`,

```
(define query-apply
  (lambda (rel t*)
    (let ([ans (query (apply rel t*))])
      (if (null? (cdr ans))
          #f
          (concretize
           (map (lambda (t) (subst-in t (car ans))) t*)))))
```

---

```
> (exists (x) (query-apply father '(pete ,x)))
(pete sal)
```

If `query-apply` is the analogue of `solution`, what is the analogue of `solve`? We leave its implementation as an exercise.

### 3.5 Sequences of antecedents

In the definition of `child-of-male`, we have exactly one antecedent. As with `select`, we would like to have an operator, `all`, which allows for more than one antecedent.

Let's first redefine `father`.

```
(define father
  (relation (dad child)
    (select
      (fact 'john 'sam)
      (fact 'sam 'pete)
      (fact 'pete 'sal)
      (fact 'pete 'pat))))
```

Now, we can see that `sam` is the grandfather of `sal` and `pat` and `john` is the grandfather of `pete`. We might ask if `sam` is the grandfather of anyone in our closed five-person world. To do this, we need two antecedents, which we wire together with function composition.

```
(define grandpa-sam
  (relation (grandchild)
    (exists (parent)
      ((to-show grandchild)
        (lambda (sk)
          ((father 'sam parent)
            ((father parent grandchild) sk)))))))

> (exists (y) (solve 6 (grandpa-sam y)))
((sal) (pat))
```

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

Thus, we make `grandpa-sam` a bit more readable by redefining the macro `all`, which now takes an arbitrary number of antecedents, much like `select` takes an arbitrary number of rules. We have once again chosen the macro over the function version, since we avoid recursion in the implementation.

```
(define-syntax all
  (syntax-rules ()
    [(_) (lambda (sk) sk)]
    [(_ antecedent0) antecedent0]
    [(_ antecedent0 antecedent1 ...)
     (lambda (sk)
       (antecedent0 ((all antecedent1 ...) sk)))]))
```

---

The similarities to `select` are striking. The first and third clauses start out the same: here `(lambda (sk) ...)` and the second clauses of both `select` and `all`, which are unnecessary, treat a singleton as itself, much like in Scheme where `(and e) = (or e) = e`.

Here is `grandpa-sam` written using `all`.

```
(define grandpa-sam
  (relation (grandchild)
    (exists (parent)
      ((to-show grandchild)
       (all (father 'sam parent) (father parent grandchild))))))
```

The macro `any` is the dual of `all`. It succeeds whenever one of its antecedents succeeds, whereas `all` fails whenever one of its antecedents fails. This macro *builds* a relation and then *calls* it. (Consider why the expansion of the macro starts out with two left parentheses `((relation () ...))`.)

```
(define-syntax any
  (syntax-rules ()
    [(_ t0 ...)
     ((relation ()
       (select
         ((to-show) t0)
         ...)))]))
```

---

As an exercise, write the macro `any` without using `relation`, `select`, and `to-show`. Here's a hint. Use `expand-only` on something relatively simple and look for patterns or places where it obviously can be improved.

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

With `all`, we can describe a template that works for many relations. Thus, logic programming becomes a matter of filling in the *term-*ij*s* and the *ant-*ij*s* (antecedents). Therefore, functions representing relations could have the following shape:

```
(define a-relation
  (relation (t1 ...)
    (select
      (exists (lv-1 ...)
        ((to-show term-11 ...)
         (all ant-11 ...)))
      ...
      (exists (lv-1 ...)
        ((to-show term-n1 ...)
         (all ant-n1 ...))))))
```

where  $n$  is nonnegative and the number of terms is the same as the number of  $t_i$ s. When  $n$  is 0, a-relation could have the name `fail = (relation () (select))`. When there is one rule (antecedent) `select (all)` is superfluous. If there are no  $lv-i$ , then the associated `exists` is also unnecessary. The `(t1 ...)` is used for its length to provide coherent error messages and in building a goal function for later unification.

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

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

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

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

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

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

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

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

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

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

We make `pete` an uncle of `betty` and `david`, the children of his sister, `polly`. First, we include `polly` in the `father` relation. Then, we define a `mother` relation, including some facts like we did for the `father` relation. Finally, we add a rule to the `grandpa` relation, so that `pete's` sister's children can claim `sam` as their grandfather.

```
(define father
  (relation (dad child)
    (select
      (fact 'john 'sam)
      (fact 'sam 'pete)
      (fact 'sam 'polly)
      (fact 'pete 'sal)
      (fact 'pete 'pat))))

(define mother
  (relation (mom child)
    (select
      (fact 'polly 'betty)
      (fact 'polly 'david))))
```



```
(define-syntax extend-relation-aux
  (syntax-rules ()
    [(_ (t-lexvar0 ...) sk fk subst cutk) (fk)]
    [(_ (t-lexvar0 ...) sk fk subst cutk rel0 rel1 ...)
     (@ (rel0 t-lexvar0 ...) sk
        (lambda ()
          (extend-relation-aux (t-lexvar0 ...) sk fk subst cutk rel1 ...))
          subst cutk)]))
```

---

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

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

(define grandpa
  (extend-relation (grandad grandchild) grandpa/father grandpa/mother))
```

The implementation of `extend-relation-aux` builds a failure continuation that processes the remaining relations, just as `select` builds a failure continuation that processes the remaining rules. The binding of `cutk` stays the same in each of these failure continuations, so that later when we invoke `cutk`, it will jump out of the extended relation. We have more to say about `cutk` later when we discuss the sullied operators.

Since relations are first-class, the preceding definition of `grandpa` can also be written like this,

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

or like this,

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

The `extend-relation` operator can be written as a procedure that takes an arbitrary number of relations, however, it requires `apply`, multi-arity procedures, and recursion. But, we primarily choose the macro form because of the increased opportunity for descriptive error messages.

With `extend-relation`, we observe that we no longer need `select`: There is a cost, however. When each relation stands on its own, it is not possible to share declarations of variables as we have done in the last definition of `grandpa`.

### 3.8 Curried relations

Just as we were able to curry functions with `lambda@`, we can curry relations with `relation@`.

```
(define-syntax relation@
  (syntax-rules ()
    [(_ (t-lexvar0 ...) body0 body1 ...)
     (let ([r (relation (t-lexvar0 ...) body0 body1 ...)])
       (relation@-aux (t-lexvar0 ...) (r t-lexvar0 ...)))]))

(define-syntax relation@-aux
  (syntax-rules ()
    [(_ (t-lexvar) body0 body1 ...)
     (relation (t-lexvar)
       ((to-show t-lexvar) body0 body1 ...))]
    [(_ (t-lexvar0 t-lexvar1 ...) body0 body1 ...)
     (lambda (t-lexvar0)
       (let ([r (relation (t-lexvar0)
         ((to-show t-lexvar0)
           (lambda@ (sk fk subst cutk)
             (let ([t-lexvar0 (subst-in t-lexvar0 subst)])
               (relation@-aux (t-lexvar1 ...) body0 body1 ...)))]))]
         (query (r t-lexvar0))))))])
```

---

Let's look at a partial expansion of `grandpa-sam`, written using `relation@`.

```
> (expand-only '(relation@ relation@-aux)
  '(define grandpa-sam
    (relation@ (child)
      ((to-show child)
        (exists (parent)
          (all (father 'sam parent) (father parent child)))))))

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

The binding of `r` is nearly identical to the body of the `let`-expression, except that in the body we have simply packaged an artificial call to `r`, whereas in the binding of `r`, we have the original `all` expression.

We define `grandpa` as a curried relation.

```
(define grandpa
  (relation@ (grandad child)
    ((to-show grandad child)
      (exists (parent)
        (all (father grandad parent) (father parent child))))))

> (exists (x y) (solve 8 ((grandpa x) y)))
```

We see another occurrence of `relation@` in the section where `concat` is defined.

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

#### 4 Sullied antecedents

In this section we present three operators for creating a Prolog-like cut operator, four for interfacing Scheme predicates and functions, and three other operators: `fails`, `instantiated`, and `view-subst`.

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

```
<A> =
  <relation call>
  | (all <A>*)
  | (any <A>*)
  | <sullied antecedent>
```

```

<sullied antecedent> =
  (!! <fk>)
  | (fail)
  | (!fail <fk>)
  | (<Scheme interface> <t> ...)
  | (fails <A>)
  | (instantiated <t>)
  | (view-subst <t>)
  | <or any Scheme expression that evaluates to these,
    since these are all values.>

<Scheme interface> =
  (pred <Scheme predicate>)
  | (pred-nocheck <Scheme predicate>)
  | (fun <Scheme function>)
  | (fun-nocheck <Scheme function>)

```

where <t> is a term, and <fk> is a failure continuation that has originally been brought into scope by either `reify-cutk` or its derivative `reify-!`.

#### 4.1 Short cuts

We present three variants of `grandpa`, each with only one use of `(!! cutk)`, below. The first one places it as the last antecedent of the only rule of `grandpa/father`. We have used `reify-cutk` to make the argument `cutk` accessible to `!!`. In fact, we could have been doing this all along, but since we were not concerned about using this failure continuation, we decided to postpone explaining how any of the six values (i.e., `goal-fn`, `lengoal`, `sk`, `fk`, `subst`, or `cutk`) can be made available to the user. Because `cutk` is used in the first argument to `@`, we cannot  $\eta$ -convert the `lambda@` expression.

```

(define reify-cutk
  (lambda (proc)
    (lambda (goal-fn lengoal)
      (lambda@ (sk fk subst cutk)
        (@ ((proc cutk) goal-fn lengoal) sk fk subst cutk))))))

(define !!
  (lambda (exiting-fk)
    (lambda@ (sk fk subst cutk)
      (@ sk exiting-fk subst cutk))))

```

---

```
(define grandpa/father
  (relation (grandad grandchild)
    (reify-cutk
      (lambda (cutk)
        ((to-show grandad grandchild)
          (exists (parent)
            (all
              (father grandad parent) (father parent grandchild) (!! cutk))))))))))

(define grandpa
  (extend-relation (grandad grandchild) grandpa/father grandpa/mother))
```

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

For `!!`, `exiting-fk` is the `cutk` of the call to `grandpa`. The effect of using `(!! cutk)` at the end is that once the first answer is found and failure forced, the `exiting-fk` that has been passed as the failure continuation is invoked, so it takes us completely out of the call to `grandpa`, as if it had been unsuccessful searching for a match within `grandpa`'s rules.

We introduce the procedure `reify-!`, which is used to abbreviate `(!! cutk)`.

```
(define reify-!
  (lambda (proc)
    (lambda (goal-fn lengoal)
      (lambda@ (sk fk subst cutk)
        (@ ((proc (!! cutk)) goal-fn lengoal) sk fk subst cutk))))))
```

---

Next, consider the following revision of `grandpa`, where we swap the last two antecedents of `grandpa/father`. By scoping the lexical variable `!` in the body of the procedure, we get `“!”` as in Prolog.

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

(define grandpa
  (extend-relation (grandad grandchild) grandpa/father grandpa/mother))

> (exists (x y) (solve 10 (grandpa x y)))
((john pete) (john polly))
```

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

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

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

(define grandpa
  (extend-relation (grandad grandchild) grandpa/father grandpa/mother))

> (exists (x y) (solve 10 (grandpa x y)))
((john pete) (john polly) (sam sal) (sam pat))
```

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

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

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

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

---

```
(define no-grandma
  (relation (grandad grandchild)
    (reify-cutk
      (lambda (cutk)
        ((to-show grandad grandchild)
         (exists (parent)
                  (all (mother grandad parent) (!! cutk) (fail))))))))))
```

We define an antecedent, (!fail cutk), to support this idiom.

```
(define !fail
  (lambda (exiting-fk)
    (lambda@ (sk fk subst cutk)
      (exiting-fk))))
```

---

And, revise no-grandma to this:

```
(define no-grandma
  (relation (grandad grandchild)
    (reify-cutk
      (lambda (cutk)
        ((to-show grandad grandchild)
         (exists (parent)
                  (all (mother grandad parent) (!fail cutk))))))))))
```

We include the relation no-grandma with grandpa to build a new relation no-grandma-grandpa to effect this behavior.

```
(define no-grandma-grandpa
  (extend-relation (grandad grandchild) no-grandma grandpa))

> (exists (x) (solve 1 (no-grandma-grandpa 'polly x)))
()
```

#### *4.2 Interfacing Scheme functions*

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

```

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

(define starts-with-p?
  (lambda (x)
    (lambda@ (sk fk subst cutk)
      (let ([x (subst-in x subst)])
        (if (lv? x)
            (error 'starts-with-p? "Variable found: ~s." x)
            (if (and
                  (symbol? x)
                  (string=? (string (string-ref (symbol->string x) 0)) "p"))
                (@ sk fk subst cutk)
                (fk)))))))

> (exists (x y) (solve 10 (grandpa x y)))
((sam sal) (sam pat))

```

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

The function `starts-with-p?` is arbitrary. Any Scheme function of any number of arguments can be used. The Scheme function can do anything that any Scheme function can do, including capturing continuations, setting variables, writing to files, etc. The full panoply of options is available to the user. The only requirement is that these functions take the same arguments, and that they exit the same way: `(@ sk fk subst cutk)` for success and `(fk)` for failure.

We abstract this with new operators, `pred`, and `pred-nocheck`, below. (We hesitate to include such operators, since we are trying to minimize the tools that we use. Here we are using `apply` and multi-arity procedures, but as we have shown with the definition of `starts-with-p?`, their use is not strictly necessary.

```

(define pred
  (lambda (p)
    (pred-nocheck (check 'pred p))))

(define check
  (lambda (name f)
    (lambda term
      (if (not (procedure? f))
          (error name "Non-procedure found: ~s" f)
          (if (ormap lv? term)
              (error name "Variable found: ~s" term)
              (apply f term))))))

(define pred-nocheck
  (lambda (p)
    (lambda term
      (lambda@ (sk fk subst cutk)
        (if (apply p (map (lambda (t) (subst-in t subst)) term))
            (@ sk fk subst cutk)
            (fk))))))

```

---

This allows us to redefine `starts-with-p?`.

```

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

```

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

```

(define fun
  (lambda (f)
    (fun-nocheck (check 'fun f))))

```

```
(define fun-nocheck
  (lambda (f)
    (lambda (t . term)
      (lambda@ (sk fk subst cutk)
        (cond
          [(unify (apply f (map (lambda (x) (subst-in x subst)) term)) t subst)
           => (lambda (subst)
                (@ sk fk subst cutk))]
          [else (fk)]))))))
```

---

```
> (exists (q) (solution ((fun *) q 3 5))
(15 3 5)
```

The Scheme interface functions `pred` and `pred-nocheck` allow the user to treat Scheme predicates as logic predicates, whereas the interface functions `fun` and `fun-nocheck` allow the user to treat Scheme functions as logic functions (A logic function is a logic relation that has at most one value for each input.) Since functions return a value, `fun` and `fun-nocheck` cause that value to get unified with the (possibly uninstantiated) first argument. The `-nocheck` versions are for conveniently writing extensions to the logic system that *can* handle logic variables as parameters, whereas `pred` and `fun` catch errors when calling functions that do not expect their arguments to be logic variables.

#### 4.2.1 *fails, instantiated, and view-subst*

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

```
(define fails
  (lambda (antecedent)
    (lambda@ (sk fk subst cutk)
      (@ antecedent
        (lambda@ (current-fk subst cutk) (fk))
        (lambda () (@ sk fk subst cutk))
        subst
        cutk))))
```

---

And here is an example of `fails`.

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

```
> (exists (x y) (solve 10 (grandpa x y)))
((john pete) (john polly))
```

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

```
(define instantiated
  (pred-nocheck
   (lambda (t)
     (not (lv? t)))))
```

---

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

To view a substitution, use `(view-subst t)` with this definition.

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

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

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

```
> (exists (x y) (solve 10 (grandpa x y)))
pete
((grandad.1 x.1)
 (grandchild.1 y.1)
 (x.1 john)
 (parent.1 sam)
 (y.1 pete))
```

```
polly
((grandad.1 x.1)
 (grandchild.1 y.1)
 (x.1 john)
 (parent.1 sam)
 (y.1 polly))
```

```

sal
((grandad.1 x.1)
 (grandchild.1 y.1)
 (x.1 sam)
 (parent.1 pete)
 (y.1 sal))

pat
((grandad.1 x.1)
 (grandchild.1 y.1)
 (x.1 sam)
 (parent.1 pete)
 (y.1 pat))

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

```

This completes the entire discussion of the implementation of our logic system. We have accomplished this using an eight-person world, while demonstrating the essential facets of logic programming.

## 5 Recursive definitions

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

### 5.1 Youngest common ancestor

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

```

(define father
  (let ([father father])
    (extend-relation (dad child) father
      (relation (dad child)
        (select (fact 'john 'harry) (fact 'harry 'carl) (fact 'sam 'ed))))))

```

The first of the two relations (i.e., `father`) passed to `extend-relation` must be dereferenced, here. If not, then we would create an infinite loop as soon as we called

father. (As an exercise, write an R<sup>5</sup> Scheme macro `extend-relation-deref` that dereferences its relation arguments before calling `extend-relation`. Hint: It is easy to do this for one relation, but you must take advantage of hygiene to generate a set of unused names.)

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

> (exists (x) (solve 21 (ancestor 'john x)))
((john sam)
 (john harry)
 (john pete)
 (john polly)
 (john ed)
 (john sal)
 (john pat)
 (john carl))
```

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

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

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

This says that john and sam are both common ancestors of pat and ed.

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

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

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

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

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

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

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 relation as an argument.

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

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

### 5.2 Comparison with Seres and Spivey

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

With one definition,

```
(define == (fun (lambda (x) x)))
```

---

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`, `to-show`, `fact`, `search`, `extend-relation`, `relation@`, and all the sullied operators. This means that we could use `lambda` to define relations. For example, we can define `father` and `ancestor` like this,

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

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

And this can be tested in the same way.

```
> (exists (x) (solve 20 (ancestor 'john x)))
((john sam)
 (john harry)
 (john pete)
 (john polly)
 (john ed)
 (john sal)
 (john pat)
 (john carl))
```

Does the Haskell embedding differ in any significant way from our embedding in Scheme? Yes, besides avoiding sullied operators, there is a fundamental difference. Everything that works in the Haskell embedding works in the Scheme embedding, but not vice versa. Because the Haskell embedding does unification piecewise within an `all`, any time that the unification fails to match, there is automatic backtracking. That is not the case with `to-show` and `fact`. In our embedding, if unification fails, it moves to the next rule. It is as though each rule has a lock on it and a term either opens the rule or a new rule is selected. In order to accomplish this in

the Haskell embedding, there would need to be a *local* cut that would force failure of one disjunct and require an attempt to find a different disjunct. This is not how a conventional Prolog-like cut would work, but it should be easy enough to implement by introducing another `reify` operation that would reasonably be called `reify-next-disjunct-fk`.

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 (`think`). Then each antecedent closure could be modelled as the consing of the data structure to a stream. Because this is a relatively small program, making this last step in order to use the stream monad seems like overkill, especially since this approach does not appear to extend naturally to the all important sullied antecedents.

### 5.3 any revisited

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

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

> (exists (x) (solution (test1 x)))
(x.1)

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

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

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

> (exists (x y) (solution (test3 x y)))
(#f y.1)
```

The first two should not be much of a surprise. In the first example, 4 is less than 5, so `x` remains uninstantiated, and 6 is less than 7, so `x` is instantiated in the second example. In the last example, however, we discover that `y` remains uninstantiated even though 5 is *not* less than 4. Why? A `fun` call always succeeds if its non-value

argument is an uninstantiated variable. So, succeeding is not the same as being true in a logic system.

#### 5.4 “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.<sup>¶</sup>

```
(define towers-of-hanoi
  (letrec
    ([move
      (relation (n a b c)
        (reify-!
          (lambda (!)
            (select
              ((to-show 0 _ _ _) !)
              (exists (m)
                ((to-show n a b c)
                 (all
                  ((fun -) m n 1)
                  (move m a c b)
                  ((pred (lambda (x y)
                          (printf "Move a disk from ~s to ~s~n" x y))
                   a b)
                  (move m c b a))))))))))
      (relation (n)
        ((to-show n) (move n 'left 'middle 'right))))
  > (begin (solution (towers-of-hanoi 3)) (void))
  Move a disk from left to middle
  Move a disk from left to right
  Move a disk from middle to right
  Move a disk from left to middle
  Move a disk from right to left
  Move a disk from right to middle
  Move a disk from left to middle
```

The algorithm is straightforward. First, figure out how to solve the problem for one fewer disks and then move that stack of disks as a virtual disk. The use of

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

pred allows us to have a procedure that writes one move. Since we are displaying information, we know that the value of `printf` is not false. Also, the move subtracts one from the number of disks using `fun` and its first rule uses the anonymous variable three times. (Recall that the anonymous variable unifies with everything, but it adds nothing to the substitution.)

## 6 Three famous problems

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

### 6.1 Enlarging the set of terms

Presently, a term is a variable or something that can be compared with `eqv?`. Now, we change that set of terms to be those that can be compared with `equal?`. Also we allow for pairs (lists) to contain variables.

```
(define nonempty-subst-in
  (lambda (t subst)
    (cond
      [(lv? t) (cond [(assv t subst) => cadr] [else t])]
      [(pair? t)
       (cons (nonempty-subst-in (car t) subst) (nonempty-subst-in (cdr t) subst))]
      [else t])))
```

---

This is not a significant change. It says that as we recursively walk through the term, we replace all variables by their association in a substitution. This is now more like `concretize`, since it takes any term and builds an identical structure while replacing each variable in the structure with a possibly different term.

Here is an interesting example,

```
> (concretize
  (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)])
          (write (concretize new-term))
          (write (concretize (subst-in new-term r)))
          (let ([sr (compose-subst s r)])
            (write (concretize sr))
            (subst-in term sr))))))))
```

```
(p (f y.1) z.1 (g z.1))
(p (f y.1) b (g b))
((y.1 b) (x.1 (f y.1)) (z.1 b))
(p (f y.1) b (g b))
```

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

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

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

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

---

There are only three kinds of terms for us to consider: variables, pairs, and everything else. The first case is straightforward. Two terms (including two variables) unify and return the empty substitution if they are `eqv?` or one of them is the global *logic* variable denoted by underscore. (This latter case is not part of unification, but is standard in Prolog.) If at least one of them is a variable, then the other becomes the term associated with it, provided that that term does not contain an occurrence of the variable. Two pairs might unify if their `cars` unify. But, before we determine if the `cdrs` unify, we must take the substitution returned from successfully unifying the `cars`, its commitments, and apply it to the `cdrs`. This brings the `cdrs` up-to-date with those commitments. Then all the commitments from unifying the `cars`, `s-car`, and all the commitments from unifying the substituted for `cdrs`, `s-cdr`, are composed. Two non-`eqv?` raw values unify and return the empty substitution if they are `equal?`. The decision to use Scheme's `equal?` for such raw values places significant restrictions on these values. For example, circular structures cannot be `equal?`, even though they might be treated as unifiable in some system.

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

The check for an occurrence of a variable in a term, below, is a straightforward recursive tree walk, however, for reasons of efficiency, most logic systems have chosen not to include it.

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

---

By leaving what a constant is unspecified, we can see that there are opportunities to promote a type of constant from being a raw value to one that contains variables. For example, it would be possible to include vectors, like we do for pairs, in `copy-term`, `nonempty-subst-in`, `unify*`, and `occurs?`. Its semantics should be obvious based on how pairs are handled.

## 6.2 Unification examples

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

```

> (concretize
  (exists (x)
    (unify '(,x ,x) '(3 4) empty-subst)))
#f
> (concretize
  (exists (x)
    (unify '(,x ,4) '(3 ,x) empty-subst)))
#f

```

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

```

> (concretize
  (exists (x y)
    (unify '(,x ,y) '(3 4) empty-subst)))
((x.1 3) (y.1 4))
> (concretize
  (exists (x y)
    (unify '(,x 4) '(3 ,y) empty-subst)))
((x.1 3) (y.1 4))

```

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

```

> (concretize
  (exists (w x y z)
    (unify '(,x 4 3 ,w) '(3 ,y ,x ,z) empty-subst)))
((x.1 3) (y.1 4) (w.1 z.1))

```

We can see that the last two items in each term do not violate earlier commitments. Thus, the two terms unify. If any other integer appears where the 3 in the first argument appears, then these fail to unify. If  $y$  appears where the  $x$  appears in the second argument, then these also fail to unify. The variable  $w$  shares with the variable  $z$ , which means that when one of them commits to some non-variable term, so does the other one. Consider these two examples.

```

> (concretize
  (exists (x y)
    (unify '(,x 4) '(,y ,y) empty-subst)))
((x.1 4) (y.1 4))
> (concretize
  (exists (x y)
    (unify '(,x 4 3) '(,y ,y ,x) empty-subst)))
#f

```

In the first example,  $x$  commits to  $y$  and agrees to be associated with the same term. Then  $y$  commits to 4. In the second example, once  $y$  has committed to 4,

since  $y$  and  $x$  are shared  $x$  is committed to 4, but  $x$  tries to commit to 3, which violates an earlier commitment.

### 6.2.1 More unification examples

We present five unification examples, below.

```
> (concretize
  (exists (r v w x u)
    (unify '(,r (,v (,w ,x) 8)) '(,r (,u (abc ,u) ,x)) empty-subst)))
((v.1 8) (w.1 abc) (x.1 8) (u.1 8))

> (concretize
  (exists (x y)
    (unify '(p (f a) (g ,x)) '(p ,x ,y) empty-subst)))
((x.1 (f a)) (y.1 (g (f a))))

> (concretize
  (exists (x y)
    (unify '(p (g ,x) (f a)) '(p ,y ,x) empty-subst)))
((y.1 (g (f a))) (x.1 (f a)))

> (concretize
  (exists (x y z)
    (unify '(p a ,x (h (g ,z))) '(p ,z (h ,y) (h ,y)) empty-subst)))
((z.1 a) (x.1 (h (g a))) (y.1 (g a)))

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

The first example commits  $r$  to itself, which is no commitment at all. Then  $v$  shares with  $u$ ,  $w$  commits to the symbol `abc`,  $x$  shares with  $u$ , and  $x$  commits to 8. In the second example, the  $p$  symbols match, leading to no commitments. Then,  $x$  commits to `(f a)`. Finally,  $y$  commits to `(g (f a))`, since the  $x$  in `(g ,x)` gets replaced by what  $x$  is committed to, `(f a)`. The third example produces the same values for the variables, but in a different way. Again,  $p$  symbols lead to no commitments. Then  $y$  commits to `(g ,x)`. Finally,  $x$  commits to `(f a)`. The fourth example first commits  $z$  to the symbol `a`. Then,  $x$  commits to `(h ,y)`. Next the  $h$  symbols contribute nothing. Finally,  $y$  commits to `(g ,z)`. But, since  $z$  has committed to `a`, we know that  $y$  must become `(g a)`, which is why  $x$ 's value is `(h (g a))`. The fifth example fails to unify, but only when we reach the last item in each term. The  $p$  symbols unify. Then,  $x$  is shared with  $y$ . When we try to unify  $x$  with `(f ,y)`, we discover that since  $y$  has been replaced by  $x$ , we are really unifying  $x$  with `(f x)`. It should be clear that this is a violation of the *occurs check* in the second condition of `unify`.

## 6.2.2 A lazy unify

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

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

```
(define unify
  (lambda (t u subst)
    (cond
      [(or (eqv? t u) (eqv? t _) (eqv? u _)) subst]
      [(lv? t) (unify-with-lv t (subst-in u subst) subst)]
      [(lv? u) (unify-with-lv u (subst-in t subst) subst)]
      [(and (pair? t) (pair? u))
       (cond
         [(unify (car t) (car u) subst)
          => (lambda (subst)
              (unify (cdr t) (cdr u) subst))]
         [else #f]])]
      [(equal? t u) subst]
      [else #f])))

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

---

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

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

```

(define compose-with-virtual-unit-subst
  (lambda (base t-var u)
    (cons (list t-var u)
          (map
           (lambda (b)
             (list (car b) (subst-in-with-virtual-unit-subst (cadr b) t-var u)))
           base))))

(define subst-in-with-virtual-unit-subst
  (lambda (t t-var u)
    (cond
     [(lv? t) (if (eqv? t t-var) u t)]
     [(pair? t)
      (cons
       (subst-in-with-virtual-unit-subst (car t) t-var u)
       (subst-in-with-virtual-unit-subst (cdr t) t-var u))]
     [else t])))

```

---

The procedure `compose-with-virtual-unit-subst` is `compose-subst` specialized for a refining substitution containing exactly one commitment, represented with two values, `t-var` and `u`. This function replaces `compose-subst` and `unit-subst` in the substitution interface, since everything would then rely on this lazy `unify`. Our reliance on `unify`, however, means that we would be giving up the fully general behavior of `compose-subst`. Furthermore, these improvements place the refining substitution in front of the refined base substitution, instead of behind it, so whenever a substitution is an outcome, its list of commitments are reversed, but this does not affect the code's correctness. Also, because of the `occurs` check that precedes the call to `compose-with-virtual-unit-subst`, we are assured that we are creating only real commitments, and because of the `assv` test that precedes the `occurs` check, there is no overlap in the variables of the two substitutions. This last observation is what allows us to unconditionally add the refining commitment to the refined base commitments. The procedure `subst-in-with-virtual-unit-subst` specializes `non-empty-subst-in`, since it builds in knowledge that there is exactly one commitment, again represented with two values.

### 6.2.3 "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 '()])
    (letrec
      ([move
        (relation (n a b c)
          (reify-!
            (lambda (!)
              (select
                ((to-show 0 _ _ _) !)
                (exists (m)
                  ((to-show n a b c)
                    (all
                     ((fun -) m n 1)
                     (move m a c b)
                     ((pred (lambda (x y)
                               (set! steps (cons (list x y) steps))))
                      a b)
                     (move m c b a))))))))))
        (relation (n path)
          (set! steps '())
          (select
            ((to-show n path) (fails (move n 'l 'm 'r)))
            ((to-show n path) (== path (reverse steps))))))
      > (exists (path) (solution (towers-of-hanoi-path 3 path)))
      (3 ((l m) (l r) (m r) (l m) (r l) (r m) (l m)))

```

The primary difference between this version and the earlier version is that in this version there is a lexical variable `steps` that holds each step, where before we printed each step. Then, by forcing failure with `fails`, we are guaranteed to process the second rule. It always succeeds, since `path` is guaranteed to be uninstantiated. We reverse the steps so that it looks like our earlier output. Everything else is the same. This solution would work with the earlier unifier, since it does not look at the type of the term it is associating with `path`, but a strongly-typed system would complain. (As an exercise, use this definition 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 (lists), so we can continue the discussion of the three famous problems.

### 6.3 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

```
> (exists (q) (solve 1 ((fun concat) q '(a b c) '(u v))))
(((a b c u v) (a b c) (u v)))
```

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

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

Or we can define it using `extend-relation` with independent relations,

```
(define base-concat
  (relation (xs ys zs)
    (exists (xs) (fact '() xs xs))))

(define nonbase-concat
  (relation (xs ys zs)
    (exists (x xs zs) ((to-show '(,x . ,xs) ys '(,x . ,zs)) (concat xs ys zs)))))
```

```
(define concat (extend-relation (xs ys zs) base-concat nonbase-concat))
```

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

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

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

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

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

But what if we include another variable?

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

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

```
> (exists (q r s) (solve 6 (concat q r s)))
((( (s.1 s.1)
  ((x.1) zs.1 (x.1 . zs.1))
  ((x.1 x.2) zs.2 (x.1 x.2 . zs.2))
  ((x.1 x.2 x.3) zs.3 (x.1 x.2 x.3 . zs.3))
  ((x.1 x.2 x.3 x.4) zs.4 (x.1 x.2 x.3 x.4 . zs.4))
  ((x.1 x.2 x.3 x.4 x.5) zs.5 (x.1 x.2 x.3 x.4 x.5 . zs.5))))
```

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

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

```
> (exists (q r) (solve 6 (concat q '(u v) '(a b c . ,r))))
(((a b c) (u v) (a b c u v))
  ((a b c x.1) (u v) (a b c x.1 u v))
  ((a b c x.1 x.2) (u v) (a b c x.1 x.2 u v))
  ((a b c x.1 x.2 x.3) (u v) (a b c x.1 x.2 x.3 u v))
  ((a b c x.1 x.2 x.3 x.4) (u v) (a b c x.1 x.2 x.3 x.4 u v))
  ((a b c x.1 x.2 x.3 x.4 x.5) (u v) (a b c x.1 x.2 x.3 x.4 x.5 u v)))
```

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

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

```
> (exists (q) (solve 6 (concat q '() q)))
((( () ())
 ((x.1) () (x.1))
 ((x.1 x.2) () (x.1 x.2))
 ((x.1 x.2 x.3) () (x.1 x.2 x.3))
 ((x.1 x.2 x.3 x.4) () (x.1 x.2 x.3 x.4))
 ((x.1 x.2 x.3 x.4 x.5) () (x.1 x.2 x.3 x.4 x.5)))
```

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.

We can even define a curried relation `concat`, but we must be careful how we test it, since `solve` is quite ad hoc.

```
(define curried-concat
  (relation@ (xs ys zs)
    (select
      (exists (xs)
        (fact '() xs xs))
      (exists (x xs zs)
        ((to-show '(,x . ,xs) ys '(,x . ,zs))
         (@ curried-concat xs ys zs))))))

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

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

#### 6.4 The Type-Inference Problem

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

##### 6.4.1 Building a type inferencer with small relations

In the expressions below, we have used different kinds of raw values in different roles. Strings are used to indicate type constructors: "bool", "int", and "->". Symbols denote lexical variables of the language, and the integer and boolean constants represent themselves. Later, we show another, but less error-prone, representation. We also assume that no two `lambda` expressions bind the same lexical variable (Use renaming by  $\alpha$ -conversion when necessary.).

While you are reading the code for the type system,  $\vdash$ , it is important to keep in mind that although we are presenting a type inference algorithm, it is just a relatively simple logic program.

$\vdash$  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  $\vdash$ , below, we can read it as, “From  $g$  we can infer that  $x$  is of type “int” provided that  $x$  is an integer.” For now, we leave  $g$  unspecified.

```
(define int-rel
  (relation (g x t)
    (reify-!
      (lambda (!)
        (exists (g x)
          ((to-show g x "int")
            (all ((pred integer?) x) !)))))))

(define bool-rel
  (relation (g x t)
    (reify-!
      (lambda (!)
        (exists (g x)
          ((to-show g x "bool")
            (all ((pred boolean?) x) !)))))))

(define !- (extend-relation (g x t) int-rel bool-rel))

> (exists (g) (solution (!- g 17 "int")))
(g.1 17 "int")

> (exists (g ?) (solution (!- g 17 ?)))
(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 logic variable  $?$  is the type. In this case,  $?$  is instantiated to “int”. The existence of `g.1` in the answers indicates that  $g$  is uninstantiated, so these work for all possible  $g$ s.

We extend  $\vdash$ , below, with the relations `zero?-rel`, `sub1-rel`, and `+-rel`, which correspond to the primitives `zero?`, `sub1`, and `+`, respectively.

```
(define zero?-rel
  (relation (g exp t)
    (reify-!
      (lambda (!)
        (exists (g x)
          ((to-show g '(zero? ,x) "bool")
            (all ! (!- g x "int") !)))))))
```

```

(define sub1-rel
  (relation (g exp t)
    (reify-!
      (lambda (!)
        (exists (g x)
          ((to-show g '(sub1 ,x) "int")
            (all ! (!- g x "int" !))))))))

(define +-rel
  (relation (g exp t)
    (reify-!
      (lambda (!)
        (exists (g x y)
          ((to-show g '(+ ,x ,y) "int")
            (all ! (!- g x "int" !) (!- g y "int" !)))))))

(define !-
  (let ([!- !-])
    (extend-relation (g exp t) !- zero?-rel sub1-rel +-rel)))

> (exists (g ?) (solution (!- g '(zero? 24) ?)))
(g.1 (zero? 24) "bool")

> (exists (g ?) (solution (!- 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,

```

> (exists (g ?)
  (solution
    (!- g '(zero? (sub1 (+ 18 (+ 24 50)))) ?)))
(g.1 (zero? (sub1 (+ 18 (+ 24 50)))) "bool")

```

In this type system, we must preserve the property that every 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 lambda expressions, applications, and fix 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 exp t)
    (reify-!
      (lambda (!)
        (exists (g t test conseq alt)
          ((to-show g '(if ,test ,conseq ,alt) t)
            (all !
              (!- g test "bool") !
              (!- g conseq t) !
              (!- g alt t) !))))))))

(define !-
  (let ([!- !-])
    (extend-relation (g exp t) !- if-rel)))

> (exists (g ?) (solution (!- g '(if (zero? 24) 3 4) ?)))
(g.1 (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 rule for variables.

```

(define lexvar-rel
  (relation (g v t)
    (reify-!
      (lambda (!)
        ((to-show g v t)
          (all ((pred symbol?) v) ! (env g v t) !))))))

(define env
  (relation (g v t)
    (reify-cutk
      (lambda (cutk)
        (exists (v t g w type-w)
          (select
            ((to-show '(non-generic ,v ,t ,g) v t)
              (!! cutk))
            ((to-show '(non-generic ,v ,t ,g) v type-w)
              (all (!! cutk) (unequal? t type-w) (!fail cutk)))
            ((to-show '(non-generic ,w ,type-w ,g) v t)
              (all (!! cutk) (unequal? w v) (env g v t))))))))))

```

```
(define unequal?
  (relation (a b)
    (reify-cutk
      (lambda (cutk)
        (select
          ((to-show a a) (!fail cutk))
          ((to-show a b) (fails (fail))))))))))
```

```
(define !-
  (let ([!- !-])
    (extend-relation (g v t) !- lexvar-rel)))
```

(As an exercise, implement the antecedent, (succeed), which fails to fail. Also, since (fail) and (succeed) are always the same value, define globals, failure and success, that hold those values.)

```
> (exists (g ?)
  (solution
    (env '(non-generic b "int" (non-generic a "bool" ,g)) 'a ?)))
((non-generic b "int" (non-generic a "bool" g.1)) a "bool")

> (exists (g ?)
  (solution
    (!- '(non-generic a "int" ,g) '(zero? a) ?)))
((non-generic a "int" g.1) (zero? a) "bool")

> (exists (g ?)
  (solution
    (!- '(non-generic b "bool" (non-generic a "int" ,g))
      '(zero? a)
      ?)))
((non-generic b "bool" (non-generic a "int" g.1)) (zero? a) "bool")
```

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

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

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

(define !-
  (let ([!- !-])
    (extend-relation (g exp t) !- lambda-rel)))

> (exists (g ?)
  (solution
    (!- '(non-generic b "bool" (non-generic a "int" ,g))
      '(lambda (x) (+ x 5))
      ?)))
((non-generic b "bool" (non-generic a "int" g.1))
 (lambda (x) (+ x 5))
 ("->" "int" "int"))

> (exists (g ?)
  (solution
    (!- '(non-generic b "bool" (non-generic a "int" ,g))
      '(lambda (x) (+ x a))
      ?)))
((non-generic b "bool" (non-generic a "int" g.1))
 (lambda (x) (+ x a))
 ("->" "int" "int"))

> (exists (g ?)
  (solution
    (!- g '(lambda (a) (lambda (x) (+ x a))) ?)))
(g.1
 (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



```

> (exists (g ?)
  (solution
    (!- g '(lambda (f) (lambda (x) ((f x) x))) ?)))
(g.1
 (lambda (f)
  (lambda (x)
   ((f x) x)))
 ("->" ("->" t-rand.1 ("->" t-rand.1 t.1))
 ("->" t-rand.1 t.1)))

```

Here, the type of  $f$  is  $(\text{"->" } t\text{-rand.1 } (\text{"->" } t\text{-rand.1 } t.1))$ , so the type of  $x$  must be  $t\text{-rand.1}$ , and the type of  $(\text{lambda } (x) ((f x) x))$  must be  $(\text{"->" } t\text{-rand.1 } t.1)$ . As should be evident, once we add a relation for application, things start to get a bit tricky.

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
  (lambda (e)
    (e (lambda (z) ((fix e) z)))))
> (exists (g ?)
  (solution
    (!- g
      '((fix (lambda (sum)
              (lambda (n)
                (if (zero? n)
                    0
                    (+ n (sum (sub1 n))))))))
      10)
    ?)))
(g.1
 ((fix (lambda (sum)
        (lambda (n)
          (if (zero? n)
              0
              (+ n (sum (sub1 n))))))))
  10)
"int")

```

The example below works, too, although testing it would fail to terminate.

```
> (exists (g ?)
  (solution
    (!- g
      '((fix (lambda (sum)
              (lambda (n)
                (+ n (sum (sub1 n))))))
        10)
      ?)))
(g.1
 ((fix (lambda (sum)
        (lambda (n)
          (+ n (sum (sub1 n))))))
  10)
 "int")
```

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`.

```
> (exists (g ?)
  (solution
    (!- 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 result of this expression would 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  $\beta$ -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. If you 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 generic-env
  (relation (g v t)
    (reify-!
      (lambda (!)
        (select
          (exists (g targ tresult)
            ((to-show '(generic ,v (">" ,targ ,tresult) ,g) v t)
              (all ((fun-nocheck instantiate) t '((">" ,targ ,tresult)) !))))
          (exists (w type-w)
            ((to-show '(generic ,w ,type-w ,g) v t)
              (all ! (env g v t) !))))))))))

(define instantiate
  (copy-term
    (lambda (t env k)
      (cond
        [(assv t env)
         => (lambda (pr)
              (k (cdr pr) env))]
        [else (let ([new-lv (lv (lv-name t))])
                 (k new-lv (cons '(,t . ,new-lv) env))))]))))

(define env (let ([env env]) (extend-relation (g v t) env generic-env)))
```

In order to implement these generics, we introduce a relation, (fun-nocheck instantiate), whose purpose is to associate the type (">" targ tresult) with the type t. (Recall that copy-term has been presented in the preliminaries.)

Here is an example that tests a polymorphic let expression.

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

(g.1
  (let ([f (lambda (x) x)])
    (if (f (zero? 5))
        (+ (f 4) 8)
        (+ (f 3) 7)))
  "int")
```

#### 6.4.2 Building a robust type inferencer with a large relation

Application is the *last* rule, since it is syntactically a catchall case. A more robust solution would be to preprocess the expressions by tagging them in the style of if-expressions. We choose the following tags: `app` for application, `var` for lexical variables, `intc` for integer constants, and `boolc` for Boolean constants. Thus we can redefine these four relations: `lexvar-rel`, `int-rel`, `bool-rel`, and `app-rel`.

```
(define lexvar-rel
  (relation (g exp t)
    (reify-!
      (lambda (!)
        (exists (v)
          ((to-show g '(var ,v) t)
            (all ! (env g v t) !)))))))

(define int-rel
  (relation (g exp t)
    (reify-!
      (lambda (!)
        (exists (g x)
          ((to-show g '(intc ,x) "int") !))))))

(define bool-rel
  (relation (g exp t)
    (reify-!
      (lambda (!)
        (exists (g x)
          ((to-show g '(boolc ,x) "bool") !))))))

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

(define -!
  (extend-relation (g exp t)
    lexvar-rel int-rel bool-rel zero?-rel sub1-rel +-rel
    if-rel lambda-rel app-rel fix-rel polylet-rel))
```

Alternatively, we can define one big relation with eleven rules.

```
(define !-
  (relation (g exp t)
    (exists (g t x y z tx)
      (reify-!
        (lambda (!)
          (select
            ((to-show g '(var ,x) t)
              (all ! (env g x t) !))
            ((to-show g '(intc ,x) "int") !)
            ((to-show g '(boolc ,x) "bool") !)
            ((to-show g '(zero? ,x) "bool")
              (all ! (!- g x "int") !))
            ((to-show g '(sub1 ,x) "int")
              (all ! (!- g x "int") !))
            ((to-show g '(+ ,x ,y) "int")
              (all ! (!- g x "int") ! (!- g y "int") !))
            ((to-show g '(if ,x ,y ,z) t)
              (all ! (!- g x "bool") ! (!- g y t) ! (!- g z t) !))
            ((to-show g '(lambda (,x) ,y) ('->" ,tx ,t))
              (all ! (!- '(non-generic ,x ,tx ,g) y t) !))
            ((to-show g '(app ,x ,y) t)
              (all ! (!- g x ('->" ,tx ,t)) ! (!- g y tx) !))
            ((to-show g '(fix ,x) t)
              (all ! (!- g x ('->" ,t ,t)) !))
            ((to-show g '(let ([,x ,y]) ,z) t)
              (all
                ! (!- g y tx) ! (!- '(generic ,x ,tx ,g) z t) !)))))))))
```

Here is a test of everything except polymorphic let using the definition of !-, above.

```
> (exists (g ?)
  (solution
    (!- g
      '(app
        (fix
          (lambda (sum)
            (lambda (n)
              (if (if (zero? (var n)) (boolc #t) (boolc #f))
                (intc 0)
                (+ (var n) (app (var sum) (sub1 (var n))))))))))
      (intc 10))
    ?)))

(g.1
  (app (fix (lambda (sum)
            (lambda (n)
              (if (zero? (var n))
                (intc 0)
                (+ (var n) (app (var sum) (sub1 (var n))))))))))
  (intc 10))
"int")
```

### 6.4.3 Long cuts

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

```
(define !-
  (relation (g exp t)
    (exists (g exp t)
      (reify-!
        (lambda (!)
          ((to-show g exp t) ((!-/generator !) g exp t)))))))
```

```

(define !-/generator
  (lambda (!)
    (letrec
      ([!- (relation (g exp t)
                    (exists (g t x y z tx)
                          (select
                            ((to-show g '(var ,x) t)
                              (all ! (env g x t) !))
                            ((to-show g '(intc ,x) "int") !)
                            ((to-show g '(boolc ,x) "bool") !)
                            ((to-show g '(zero? ,x) "bool")
                              (all ! (!- g x "int") !))
                            ((to-show g '(sub1 ,x) "int")
                              (all ! (!- g x "int") !))
                            ((to-show g '(+ ,x ,y) "int")
                              (all ! (!- g x "int") ! (!- g y "int") !))
                            ((to-show g '(if ,x ,y ,z) t)
                              (all ! (!- g x "bool") ! (!- g y t) ! (!- g z t) !))
                            ((to-show g '(lambda (,x) ,y) ('-> ,tx ,t))
                              (all ! (!- '(non-generic ,x ,tx ,g) y t) !))
                            ((to-show g '(app ,x ,y) t)
                              (all ! (!- g x ('-> ,tx ,t)) ! (!- g y tx) !))
                            ((to-show g '(fix ,x) t)
                              (all ! (!- g x ('-> ,t ,t)) !))
                            ((to-show g '(let ([,x ,y]) ,z) t)
                              (all
                                ! (!- g y tx) ! (!- '(generic ,x ,tx ,g) z t) !))))))
      !-)))

```

Here is a test of a polymorphic let-expression using the full system.

```

> (exists (g ?)
  (solution
    (!- g
      '(let ([f (lambda (x) (var x))])
        (if (app (var f) (zero? (intc 5)))
            (+ (app (var f) (intc 4)) (intc 8))
            (+ (app (var f) (intc 3)) (intc 7))))
      ?)))
(g.1 (let ([f (lambda (x) (var x))])
  (if (app (var f) (zero? (intc 5)))
      (+ (app (var f) (intc 4)) (intc 8))
      (+ (app (var f) (intc 3)) (intc 7))))
"int")

```

We make three observations about these implementations of the same type inference system. First, because we have tagged every expression, we no longer require

the integer, boolean, and variable rules to use `pred`. Therefore, we can include the cut as an antecedent. Second, because of the flexibility of our logic system we can write both systems in either style (with or without tags and whole or broken into several relations). Third, because application is a catchall, it acted as a catalyst in developing `extend-relation` in order to build the first inferencer one step at a time.

#### 6.4.4 Type inhabitation

Here are four, perhaps unexpected, examples.

```
> (exists (g ?)
  (solution
    (!- g ? '(">" "int" "int"))))
((non-generic x.1 (">" "int" "int") g.1)
 (var x.1)
 (">" "int" "int"))
> (exists (g la f b)
  (solution
    (!- g '(,la (,f) ,b) '(">" "int" "int"))))
(g.1
 (lambda (x.1) (var x.1))
 (">" "int" "int"))
> (exists (g h r q z y t)
  (solution
    (!- g '(,h ,r (,q ,z ,y) t)))
((non-generic x.1 "int" g.1)
 (+ (var x.1) (+ (var x.1) (var x.1)))
 "int")
> (exists (g h r q z y t u v)
  (solution
    (!- g '(,h ,r (,q ,z ,y) '(,t ,u ,v))))
(g.1
 (lambda (x.1) (+ (var x.1) (var x.1)))
 (">" "int" "int"))
```

The first example attempts to find an expression whose type is (" $\rightarrow$ " "int" "int"), but instead finds a type environment that binds that type to the variable `x.1`, and then the expression is trivially `(var x.1)`. 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 the `var` rule first in the definition of `!-`, above. In the third example, it infers that `t` must be of "int" type. Then since there is only one binary operation that returns an "int" (i.e., `+`),

it determines  $q$  and  $h$ . Next we can infer that  $r$ ,  $z$ , and  $y$  must be of "int" type, and what is easier than making them all the same variable and placing it in the initial type environment. The last example only differs in the shape of the resultant type. Here it assumes that since the type contains three parts, it must be an arrow type. That means that  $h$  must be the symbol `lambda`. Once again the only binary operator is `+` making  $z$  and  $y$  be of "int" type.

### 6.5 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)
    (relation (x y z)
      (select
        ((to-show x y z)
         (all
          (fails (instantiated z))
          ((fun op) z x y)))
        ((to-show x y z)
         (all
          (fails (instantiated y))
          ((fun inverted-op) y z x)))
        ((to-show x y z)
         (all
          (fails (instantiated x))
          ((fun inverted-op) x z y)))
        ((to-show x y z) ((fun op) z x y))))))

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

> (exists (x) (solution (++ x 16.0 8)))
(-8.0 16.0 8)
> (exists (x) (solution (** 10 x 50)))
(10 5 50)
> (exists (x) (solution (-- 10 7 x)))
(10 7 3)
```

And we can do something similar with invertable unary functions.

```
(define invertible-unary-function->binary-relation
  (lambda (op inverted-op)
    (relation (x y)
      (select
        ((to-show x y)
          (all
            (fails (instantiated y))
            ((fun op) y x)))
        ((to-show x y)
          (all
            (fails (instantiated x))
            ((fun inverted-op) x y)))
        ((to-show x y) ((fun op) y x))))))

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

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

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

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

In the first example, we return the `char->integer` of each character in `sleep`. In the second, given a list of integers, presumably derived from `char->integer`, it returns the symbol made from those integers. Thus, we have the Prolog relation `name`. For our purposes, which is an embedding in Scheme, it is probably unnecessary, but it is interesting, nonetheless, that we can define these two relation-generating Scheme functions. There are more unary functions that can be so treated, like `symbol->string` and `string->symbol`, but we leave their inclusion to the programmer who might need them.

## 7 Final thoughts

The need for `cutk` stems from the desire to have `extend-relation`. In a system without `extend-relation`, `cutk` can disappear and everywhere that `cutk` has been passed (not just threaded) as an argument in the implementation can be replaced by `fk`. Deciding to include `extend-relation` has been a burden, since it has made the implementation difficult to understand. In the end, however, ease of writing user programs won out over simplicity of implementation. In any event, threading an extra variable is not difficult in such a short program. As alluded to above, this threading may perhaps be bypassed using monads.

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. This may emerge after serious contemplation. In its place we have made relations lexically-scoped, first-class, and extensible, which seems to be a fair tradeoff.

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 rules from existing rules. This may be retrieved by defining the relations with `quote` 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 the higher-order capabilities to get around most of these needs. Second, and more importantly, we have abandoned the database capability associated with logic programming. By that, we mean the ability for all the rules to be treated as a global monolithic set of rules. To circumvent this shortcoming, one can write a driver that knows some subset (possibly all) of the relations and their arities and whenever a relation of a particular arity is invoked, it searches through all the relations of that arity. This would be easy to set up with `extend-relation` and an association list that associates an arity with an extended relation of that arity. We can further partition this association list by replacing the arity with a type signature. But, this approach would have to re-address the cut. Since, this browsing capability is not required by the problem domains we have in mind, probably it is best that it be developed on an as needed basis. From the start we have been developing a tool that would allow for easy implementation of language-related programs such as type inferencers, interpreters, and compilers. These clearly do not need a global database.

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 seven

macros: `relation`, `to-show` (and `fact`), `all`, `any`, `extend-relation`, and `solve`. Only `relation` of these five must be a macro, the others can be written as functions, but then some of the `lambda` terms would require arbitrary arity which would cause us to abandon all the considerable advantages of `unify-incrementally`. We leave the writing of those as an exercise. Of course, macros written outside of the standard could make the code as user-friendly as you might want, but then what is going on would be concealed from the users. They might want to add features of their own such as keeping track of all the rules that were applied successfully (i.e., a proof tree), but this would be quite difficult unless it is clear exactly how things worked.

## 8 Acknowledgments

This paper would not have been possible without the earlier work on implementing logic systems with Anurag Mendhekar. The work with Anurag led to Jon Rossie's use of our logic system in the development of his dissertation's results. His utilization of the tool helped us understand how we had to weave things together. But, it wasn't until work with Mitch Wand and Chris Haynes in *Essentials of Programming Languages, Second Edition* on the material on unification, substitutions, and logic programming that it seemed there might be a chance for a "Poor Man's" solution. Steve Ganz's dissertation also needed an inference system. Over the years, Steve began to see how we could make the seamlessness of Scheme possible by showing how to partition the monolithic set of rules 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. He also implemented `relation@`. I am grateful for several conversations with Mitch Wand at ICFP 2002 where he indirectly pointed out that `all` and `select` were red herrings when thinking about the architecture of the logic system. 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.