TUTORIAL

A Poorman's 'Roll Your Own' Logic System

Daniel P. Friedman †

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

1 Introduction

We present a complete implementation of a rudimentary logic system that interacts seamlessly with Scheme. The issue is how to have logic programming and lose nothing of Scheme. The solution is to make the control structure of logic programming explicit, so that Scheme programs work seamlessly with it. This, of course, is a matter of taste, and there have been others who have presented such a system, but this version is virtually seamless, requiring a small handful of functions that comprises its interface.

I believe that implementing a logic system should be a relatively easy task. In order to make this not only useful, but comprehensible, I had to become aware of two properties that are not normally associated with the discussion of logic programming. The first of these properties is that there is absolutely no need to think about the rules as a monolithic global list of rules. In fact, in virtually every circumstance, the programmer knows exactly which set of rules to home in on and we are going to require that knowledge of the programmer. 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. Unbeknownst to me at the time was that removing this search could lead to a much cleaner implementation of a logic system, thus making the seamlessness of the integration with Scheme possible. Naturally, there are still searches roaming around in the system, but they are the responsibility of the user who indirectly makes calls in a set of mutually recursive functions, something that Scheme already supports.

The second of these properties is that there should be no distinction between a Scheme function and a logic relation. This discovery has allowed for the removal of a dispatch, thus, leading to a virtually conditional-free implementation. (Of course,

† I know in my heart what a "poorman's" solution is, but I have not yet articulated it.

we need to unify and prepare terms, which requires tests and we need to apply substitutions, the results of these unifications, to terms, which also require tests, but we can legitimately think of these tests as part of a different system having to do with the notions of unification, substitutions, and to the kinds of terms we have chosen.)

In section 2 we implement and demonstrate the unsullied logic system. In section 3 we explore sullied operators, such as `!`, the cut operator, and their implementation. Next we utilize some recursive rules and we follow that by a section discussing two famous logic programming problems: `append` and type inference. We conclude with some perspective on why this approach works and introduce a final function that replaces four of the seven functions needed to build the unsullied logic system.

2 The unsullied logic system

The logic system interface has seven functions: `if*`, `to-show`, `axiom`, `query`, `show`, `empty`, and `also`. First, we introduce `if*` (and its associated special form `cond*`). Next, we define `to-show`, the interface to the unifier, and `axiom`. Then we introduce `query`, which starts the inferencing process. At this point, we have a system that works provided that everything is an axiom. Next, we introduce `solve` for producing more than a single result, but it is a mere convenience. Then we introduce `show` and `empty` for processing a clause with exactly one antecedent. Finally, we introduce `also` for processing a clause with additional antecedents.

2.1 Taking control with `if*`

We introduce `if*`, which works like a functional `if` except that it allows for a true path to be abandoned after it has started and then it follows the false path as if the test had been false. Here is the definition of `if*`.

```
(define if*
  (lambda (test true-k false-k)
    (if test (true-k false-k) (false-k))))
```

Consider a simple expression that uses `if*`.

```
> (define test-if*
    (lambda (n)
      (if* (zero? (- n 2))
        (lambda (fk) (write n) (fk))
        (lambda () (write (* n n))))))
> (begin (test-if* 3) (test-if* 2))
924
```

Through the function call, we bind `false-k`, which is a failure continuation, and we bind `true-k` to a function that accepts that same failure continuation. Then if `test` is false, we invoke the failure continuation. So far, we have the expected semantics of `if`. But, if `test` is true, we pass to `true-k` the ability to invoke

`false-k`, but leave that to the code of `true-k`. Thus, we build in a mechanism for allowing the writer of `true-k` to forget that its path has been taken: simply tail call the passed failure continuation from the code of `true-k`.

We can envision a variant of `test-if*` that uses `cond*`, a natural generalization of `if*`.

```
(define test-if*/using-cond*
  (lambda (n)
    (cond*
      [(zero? (- n 2))
       (lambda (fk) (write n) (fk))]
      [else (write (* n n))])))
```

Just as `cond` has other facets, like `=>`, so also does `cond*`. Here is a typical program that uses `=>`.

```
(define test-cond*
  (lambda (n)
    (cond*
      [(zero? (- n 1))
       (lambda (fk)
         (write n)
         (fk))]
      [(+ n 1)
       => (lambda (num)
            (write num)
            (lambda (fk)
              (write (* num num))
              (fk)))]
      [else (write 5) (newline)])))
```

Each `cond*` clause gets the ability to forget that its test has succeeded. Let's walk through `(test-cond* 1)`, which prints `1245`. How? Each clause prints at most two digits, so if these were `cond` clauses, we would not expect to see four digits. First, each right-hand-side of a `cond*` clause is a function. In the first case, the function is `(lambda (fk) ... (fk))`. In the second clause, because of the `=>` the `(lambda (fk) ...)` is below what binds the result. The argument to `test-cond*` is `1`, so the first test is true, thus we write `1`, then we invoke `fk`, which moves us to the next clause. The next test returns 2, which is not false, thus binding 2 to `num`. We write 2. Then we write `4`. Again, we forget by invoking `fk` and move to the next clause, which writes `5`.

Just as we know that `cond` expands to a sequence of nested `if`s, we discover that `cond*` expands to a sequence of nested `if*`s. We present the definition of `cond*` in Appendix A.

Of course, we can rewrite `test-cond*` without `cond*` by expanding into `if*`.

```
(define test-cond*/using-if*
  (lambda (n)
    (if* (zero? (- n 1))
      (lambda (fk)
        ((lambda (fk) (write n) (fk)) fk))
      (lambda ()
        ((lambda (temp)
           (if* temp
             (lambda (fk)
               (((lambda (num)
                   (write num)
                   (lambda (fk)
                     (write (* num num))
                     (fk)))
                 temp)
                fk))
             (lambda ()
               (write 5)
               (newline))))
         (+ n 1))))))
```

This ability to forget past decisions is called *backtracking*. We do not have much more to say about backtracking, except that when we need it, we use `cond*`, instead of `cond`. So, since our logic programs assume backtracking, which is standard practice, we *always* use `cond*`.

2.2 It's a small world

Let's define some facts.

```
(define father
  (lambda (goal b cut)
    (cond*
      [(to-show '(john sam) goal b)
       => axiom]
      [(to-show '(sam pete) goal b)
       => axiom]
      [(to-show '(pete sal) goal b)
       => axiom]
      [else (cut)])))
```

The function `father` takes as arguments a goal that we would like to show holds, a bundle, `b`, (described below), and a failure continuation that we call `cut`. The bundle is built using a record constructor, `bundle`. Each bundled record `b` contains three items: a success continuation (`bundle->sk b`), a counter (`bundle->counter b`), and a substitution (`bundle->subst b`). (We leave the implementation of these four trivial operations unspecified, since records are not in the Scheme standard, but virtually all Schemes have some built-in mechanism for handling records.) Does the order of the `cond*` clauses in `father` matter? Yes, but not yet, since we are merely interested in determining if a particular goal matches any of the three `cond*` clauses. (See Appendix B for a definition of `father` that does not use `cond*`.)

We have two functions that we have not yet described: `to-show` and `axiom`. We start with `to-show`.

```
(define to-show
  (lambda (term goal b)
    (to-show/prepared (prepare term (bundle->counter b)) goal b)))

(define to-show/prepared
  (lambda (prepared-term goal b)
    (cond
      [(unify prepared-term goal (bundle->subst b))
       => (let ([sk (bundle->sk b)]
                [c (bundle->counter b)])
            (lambda (exit-subst)
              (bundle sk c exit-subst)))]
      [else #f])))
```

The argument to `to-show`, `term`, is unprepared (below). The bundle contains the appropriate counter, one that has never been used. When `unify` returns with a substitution, we return a new bundle with the probably enlarged substitution in place of the one that was passed into `unify`. We have split `to-show` into two functions, because we have a use for the second one, `to-show/prepared`, a bit later.

The function `prepare`, below, takes a term, `t`, and an integer, `c`, and returns a term identical to `t` except that each variable is replaced by a variable with the same integer suffix, `c`. For example, if `c` were 3, the term `(a X Y d X f)` would become `(a X.3 Y.3 d X.3 f)`. (Any symbol whose first character is upper case or an underscore is considered a variable. Everything else is a constant. Here we choose to break with the Scheme standard, where upper-case symbols are indistinguishable from lower-case symbols. If you want to stay within the standard, one way would be to simply require some special character (such as an underscore or question mark) as the first character of every variable and change the definition of `uc-symbol?`.) Because we are using symbols, we know that two symbols made at different times, like the first and second `X.3`, are the same symbol. (If `format` is unavailable, it is easy to write this use with `string-append`, `symbol->string`, and `number->string`. If a period cannot be used within symbols, use a colon.) Somewhat surprisingly, everything in Scheme that is not a variable (`uc-symbol?`) or a list is treated as a

constant, which includes procedures and vectors, as well as the expected empty list, characters, numbers, booleans, strings, and lower case symbols.

```
(define prepare
  (lambda (t c)
    (cond
      [(uc-symbol? t) (string->symbol (format "~s.~s" t c))]
      [(constant? t) t]
      [else (cons (prepare (car t) c) (prepare (cdr t) c))])))
```

Since the variables in a freshly `prepared` term have never been used, they are not yet in any substitution. Therefore, there is no reason to apply the substitution to the first argument passed to `to-show/prepared`, although no harm would be done. By the time that `to-show` returns, these freshly prepared variables are in the bundle's substitution, either instantiated or uninstantiated.

The function `axiom` accepts a bundle, `b`, and passes its counter and its substitution to its success continuation.

```
(define axiom
  (lambda (b)
    ((bundle->sk b) (bundle->counter b) (bundle->subst b))))
```

This function invokes the bundle. An *axiom* is a `cond*` clause that has no antecedents. A clause with or without antecedents is called a *rule*.

2.3 Testing `father`

We have now defined the two functions, `to-show` and `axiom`. We can ask if `pete` is the `father` of `sal`.

```
> (father '(pete sal) initial-bundle initial-cut)
```

where we define `initial-bundle` and `initial-cut` as follows.

```
(define initial-bundle
  (let ((initial-sk (lambda (global-c subst)
                      (lambda (fk)
                        (cons subst fk)))))
    (bundle initial-sk 0 empty-subst)))

(define initial-cut
  (lambda ()
    '()))
```

If the goal succeeds by invoking the initial bundle's success continuation, it returns a function that awaits a failure continuation. Once it gets the failure continuation, it returns a pair of a substitution and that continuation. If the goal fails, the empty list, the result of invoking `initial-cut`, is returned.

Since our test has no variables, we know that we do not extend the original substitution, but unify when the constants in each term are the same. What is the

purpose of `empty-subst`, the empty substitution? At this point, we can apply the empty substitution to the original goal and produce what we started with, the value of `'(pete sal)`, but with the assurance that `pete` has been shown to be the father of `sal`.

What is `((axiom initial-bundle) initial-cut)`? If you do the β steps, you should discover that it returns a dotted pair with `empty-subst` in the `car` and `initial-cut` in the `cdr`.

We next consider the role of nonempty substitutions. Instead of asking a specific question about `pete`'s relationship to `sal`, let's determine `pete`'s children.

```
> (father '(pete X) initial-bundle initial-cut)
((unit-subst X sal) . #<procedure false-k>)
```

Then we can consider:

```
> (let ([goal '(pete X)])
    (let ([result (father goal initial-bundle initial-cut)])
      (subst-in goal (car result))))
```

We can see in `(unit-subst X sal)` that the variable, `X`, has been instantiated to the constant, `sal`, and when we use this substitution, the `X` in `'(pete X)` gets replaced by `sal`, which leads to `(pete sal)`.

Alternatively, we can write this using a `guide/goal` like `` `(,father pete X) ``, but introduce a function `query`, which packages the initial bundle and the initial cut before invoking the guide, `father`, on the goal, `(pete X)`.

```
> (let ([result (query `(,father pete X))])
    (subst-in '(pete X) (car result)))
```

```
(define query
  (lambda (guide/goal)
    (call guide/goal initial-bundle initial-cut)))
```

```
(define call
  (lambda (guide/goal bundle cut)
    (if (not (procedure? (car guide/goal)))
      (error 'call "~s is not a procedure." (car guide/goal)))
    ((car guide/goal) (cdr guide/goal) bundle cut)))
```

We use `call` in a definition later, otherwise there would be no reason to split it away from the definition of `query`.

2.4 Generating more than one result

Suppose that pete is also the father of pat, what changes could be made to get both results?

```
(define father
  (lambda (goal b cut)
    (cond*
      [(to-show '(john sam) goal b)
       => axiom]
      [(to-show '(sam pete) goal b)
       => axiom]
      [(to-show '(pete sal) goal b)
       => axiom]
      [(to-show '(pete pat) goal b)
       => axiom]
      [else (cut)])))

> (let ([result1 (query `(,father pete X))])
    (list
      (subst-in '(pete X) (car result1))
      (let ([result2 ((cdr result1))])
        (subst-in '(pete X) (car result2)))))

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

First, we add that pete is also the father of pat. Then, we invoke the cdr of result1, which continues searching for another result.

Next, consider the following:

```
> (let ([result1 (query `(,father pete X))])
    (cons
      (subst-in '(pete X) (car result1))
      (let ([result2 ((cdr result1))])
        (cons
          (subst-in '(pete X) (car result2))
          (let ([result3 ((cdr result2))])
            (if (null? result3)
              '()
              (cons (subst-in '(pete X) (car result3)) '())))))))
```

This expression clarifies that we do not try to get more than three results, but we might stop at two, if that turns out to be all we can find. In other words, after finding two results, we check to see what happens if we try for a third result. If no result exists, we quit, otherwise, we include that third result and then simply return the three-element list. In this case, we quit with just two values.

2.5 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 results, but if we want to process the results as a list, we must place some bound on the number that we actually want.

The `(car result1)` is a substitution, and the `(cdr result1)` is a function that if invoked returns a value whose `car` is a substitution, and whose `cdr` is a function, etc. Thus, we can see this result as 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 1 element, if n is 0; 2 elements, if n is 1; etc. But, we must be careful that going for the third element of a stream does not lead to an infinite loop, even though we only want the first two elements! (Consider what would happen in the code, above, if `((cdr result2))` fails to terminate.) In other words, 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 function that takes a positive integer upper bound and a `guide/goal`, (like `` `(,father pete X) ``) and returns a list like the one above for `pete`'s children. We call this function `solve`.

```
(define solve
  (lambda (n guide/goal)
    (map (lambda (subst) (subst-in (cdr guide/goal) subst))
      (stream-prefix (- n 1) (query guide/goal)))))
```

```
> (solve 5 `(,father pete X))
```

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

`Exercise:`

Rewrite `solve` to set up an interactive loop to force more results. Use `0` to indicate no more results and use `+` to indicate more results.

`End of exercise`

Let's extend the most recent definition of `father`, by adding a `cond*` clause with `=>` just above the `else` clause so that it enters an infinite loop if the first argument to `solve` is greater than 2.

```
(define father
  (lambda (goal b cut)
    (cond*
      [(to-show '(john sam) goal b)
       => axiom]
      [(to-show '(sam pete) goal b)
       => axiom]
      [(to-show '(pete sal) goal b)
       => axiom]
      [(to-show '(pete pat) goal b)
       => axiom]
      [(to-show '(pete mephistopheles) goal b)
       => (lambda (b)
            (lambda (fk)
              (let infinite-loop () (infinite-loop))))]
      [else (cut)])))
```

Would it have changed the result if we used the variable, `Mephistopheles`, instead of the constant, `mephistopheles`? We would have gotten into the same infinite loop, of course, but our substitution would have included the knowledge that the variable `X` is treated the same as the variable `Mephistopheles.0`, so if we determine a value for one of the variables, that value would also be *shared* by the other variable.

So far our substitutions have associated two different kinds of information with a single variable. The first, and most familiar, is a constant, like `sal`, and the second and most unusual, is a variable, like `Mephistopheles.0`. We can also instantiate a variable with a term that includes constants and/or variables, but we ignore such examples for now.

2.6 Goals with more than one variable

What is the value of `(solve 6 '(,father X Y))`?

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

Going back to an earlier definition of `father`, `X` gets instantiated to `john` and `Y` gets instantiated to `sam`; then we fail and re-instantiate `X` to `sam` and `Y` to `pete`; then we fail and re-instantiate `X` to `pete` and `Y` to `sal`; then we fail and re-instantiate `X` to `pete` and `Y` to `pat`; and we fail one more time invoking `initial-cut`, which returns the empty list. Although this should have probably been obvious, it demonstrates some of the power of using more than a single variable in a goal.

2.7 Clauses with nonempty antecedents

We might ask if john is the grandfather of anyone in our closed five-person world.

```
(define grandpa-john
  (lambda (goal b cut)
    (cond*
      [(to-show '(YOUNG) goal b)
       => (show `(,father john PARENT)
            (also `(,father PARENT YOUNG) empty))]
      [else (cut)])))

(define show
  (lambda (guide/goal-to-be also*)
    (lambda (b)
      (((also guide/goal-to-be also*) (bundle->counter b) (bundle->sk b))
       (bundle->counter b) (bundle->subst b)))))

(define also
  (lambda (guide/goal-to-be also*)
    (lambda (local-c sk)
      (lambda (global-c subst)
        (lambda (fk)
          (call (cook guide/goal-to-be local-c subst)
            (bundle (also* local-c sk) (+ global-c 1) subst)
            fk))))))

(define empty
  (lambda (counter sk) sk))

(define cook
  (lambda (guide/goal-to-be local-c subst)
    (subst-in (prepare guide/goal-to-be local-c) subst)))
```

```
> (solve 6 `(,grandpa-john X))
((pete))
```

It is correct, because john is the father of sam and sam is the father of pete. Here is how the substitution that led to this result was built. First X became someone YOUNG.0. At this point, X is not instantiated, only shared with the variable YOUNG.0. Then PARENT.0 is instantiated to sam, then YOUNG.0 is instantiated to pete, but we know that X is shared with YOUNG.0, so X is also instantiated to pete and is substituted for X in the original goal.

Don't worry about understanding the definition of also, just yet, but observe that in the call, the first argument is *cooked* (i.e., prepared with the counter local-c and then substituted into) and that in the new bundle being passed to the call, the counter is different from the global-c coming into the call while the

arguments to `also*` are the same as the formal parameters to the second `lambda` expression in `also`.

The functions `show`, `also`, and `empty` are the last pieces of our logic system puzzle. Other than sullied operators, which are only used in antecedents, the entire system has been presented.

We can describe a template that works for all such functions by first defining `succeed`.

```
(define succeed
  (lambda (no-args b cut)
    ((axiom b) cut)))
```

Then we can characterize the template like `grandpa-john`. Thus, logic programming becomes a matter of filling in the `thing-`*i*`s` and the `ant-`*ij*`s` (antecedents). All functions representing relations have the following shape.

```
(define r
  (lambda (goal b cut)
    (cond*
      [(to-show thing-1 goal b)
       => (show ant-11 (also ant-12 (also ...)))]
      ...
      [(to-show thing-n goal b)
       => (show ant-n1 (also ant-n2 (also ...)))]
      [else (cut)])))
```

where `n` is nonnegative and `(show '(,succeed) empty)` is another way of writing `axiom`. Of course, we would suggest using `axiom` instead of the `'(,succeed)` antecedent. When `n` is 0, `r` has the name `fail`.

```
(define fail
  (lambda (no-args b cut)
    (cut)))
```

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

```
(define grandpa-maker
  (lambda (old)
    (lambda (goal b cut)
      (cond*
        [(to-show '(YOUNG) goal b)
         => (show '(,father ,old PARENT)
              (also '(,father PARENT YOUNG) empty))]
        [else (cut)]))))

> (solve 6 '(,(grandpa-maker 'john) X))
```

This just uses lexical scope to re-make `grandpa-john`. But, it still requires that we know who we want to find out about. We can do something a bit more abstract. We can postpone the determination of the guide by making it, too, a variable. Thus, we can pass the function `father` as a constant, which then gets invoked. (I choose the name `FATHER`, but a better name might be `GUIDE`.) 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` (below) would still work, as long as `mother` was passed to `grandpa-maker` instead of `father`. Then we would have a matrilineal grandparent instead of a patrilineal one. Of course, `grandpa-maker` would be a poorly chosen function name, then.

```
(define grandpa-maker
  (lambda (old)
    (lambda (goal b cut)
      (cond*
        [(to-show '(FATHER YOUNG) goal b)
         => (show `(FATHER ,old PARENT)
              (also `(FATHER PARENT YOUNG) empty))]
        [else (cut)]))))
> (solve 4 `(,(grandpa-maker 'john) ,father X))
((#<procedure father> pete))
```

We return to our more concrete problem, which is to use our logic system to define `grandpa`. Now, we can replace the lexical variable `old` with a variable `OLD` that leaves the decision completely open. Here is our first (somewhat naive) definition of `grandpa`. (Alternate syntaxes are considered in the conclusion and Appendix C.)

```
(define grandpa
  (lambda (goal b cut)
    (cond*
      [(to-show '(OLD YOUNG) goal b)
       => (show `(,father OLD PARENT)
            (also `(,father PARENT YOUNG) empty))]
      [else (cut)])))

> (solve 6 `(,grandpa john X))
((john pete))
```

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

```
(define father
  (lambda (goal b cut)
    (cond*
      [(to-show '(john sam) goal b)
       => axiom]
      [(to-show '(sam pete) goal b)
       => axiom]
      [(to-show '(sam polly) goal b)
       => axiom]
      [(to-show '(pete sal) goal b)
       => axiom]
      [(to-show '(pete pat) goal b)
       => axiom]
      [else (cut)])))

(define mother
  (lambda (goal b cut)
    (cond*
      [(to-show '(polly betty) goal b)
       => axiom]
      [(to-show '(polly david) goal b)
       => axiom]
      [else (cut)])))

(define grandpa
  (lambda (goal b cut)
    (cond*
      [(to-show '(OLD YOUNG) goal b)
       => (show `(,father OLD PARENT)
            (also `(,father PARENT YOUNG) empty))]
      [(to-show '(OLD YOUNG) goal b)
       => (show `(,father OLD PARENT)
            (also `(,mother PARENT YOUNG) empty))]
      [else (cut)])))

> (solve 10 `(,grandpa X Y))
((john pete) (john polly) (sam sal) (sam pat) (sam betty) (sam david))
```

And we discover that `sam` is the grandfather of `betty` and `david`.

We can redefine `grandpa` by introducing two new functions: `grandpa-through-son` and `grandpa-through-daughter`, since `show` (and `also`) applied to arguments evaluates to a function.

```
(define grandpa
  (lambda (goal b cut)
    (cond*
      [(to-show '(OLD YOUNG) goal b)
       => grandpa-through-son]
      [(to-show '(OLD YOUNG) goal b)
       => grandpa-through-daughter]
      [else (cut)])))

(define grandpa-through-son
  (show `(,father OLD PARENT)
    (also `(,father PARENT YOUNG) empty)))

(define grandpa-through-daughter
  (show `(,father OLD PARENT)
    (also `(,mother PARENT YOUNG) empty)))
```

This works because we are not relying on any lexical values. Of course, it is more difficult to understand when the antecedents have been separated from the driver like this. But, it is important to realize that we can do this. It also clarifies that the variables are not lexical, since `OLD` in the first clause is the same as `OLD` in the first antecedent of `grandpa-through-son`.

2.8 How does `also` *work?*

Let's look at the code of `also`, closely. Just as the definition of `show` binds `also*`, so does `also`. The arguments to `also` are `guide/goal-to-be` and `also*`. The function `also` immediately returns a function `(lambda (local-c sk) ...)`. The `local-c` is the same counter that was passed to `to-show`, so that all variables in every `guide/goal-to-be` in the antecedents of *this* call will have that suffix. Thus, even though YOUNG appears twice, it becomes YOUNG.0 in both places. Once `local-c` and `sk` are bound, a success continuation, `(lambda (global-c subst) (lambda (fk) ...))` is returned. Eventually, this success continuation accepts a failure continuation. When it does, we take the current substitution and apply it to a freshly `prepared` (with the `local-c` suffix) `guide/goal-to-be`. This cooking of the `guide/goal-to-be` is like evaluating the arguments in a call-by-value function call. Each time we try to show an antecedent, we make a new call, but if the same goal is attempted more than once at the same `local-c`, the arguments are not recoooked. Each call requires, in addition to the cooked arguments, a bundle and the current failure continuation. The way that `query` differs from `also` is that its `guide/goal` is treated as if it were precooked.

We construct the bundle in three steps. First, we include a success continuation, which we get by passing the same local counter and success continuation to `also*`. Second, we include a fresh counter. The counter is determined by incrementing the globally-threaded counter, which starts out in `show` the same as the local counter. Finally, we use the most recent substitution (as long as we succeed, it keeps accumulating commitments).

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

3 Using sullied operators in antecedents

In this section we present sullied operators: `!`, the cut operator, `!fail`, `fails`, and `instantiated`. Along the way, we present `pred` and `fun`, which allow for the seamless integration of Scheme predicates and functions. We also include a facility for viewing substitutions.

3.1 ! and !fail

We present three variants of `grandpa`, each with only one use of `` `(,! ,cut) ``. The first example places it as the last antecedent of the first clause.

```
(define grandpa
  (lambda (goal b cut)
    (cond*
      [(to-show '(OLD YOUNG) goal b)
       => (show `(,father OLD PARENT)
            (also `(,father PARENT YOUNG)
              (also `(,! ,cut) empty)))]
      [(to-show '(OLD YOUNG) goal b)
       => (show `(,father OLD PARENT)
            (also `(,mother PARENT YOUNG) empty))]
      [else (cut)])))

(define !
  (lambda (list-of-cut b cut)
    ((axiom b) (car list-of-cut))))
```

Only one value is returned for this call,

```
> (solve 10 `(,grandpa X Y))
((john pete))
```

For `!`, `(car list-of-cut)` is the `cut` of the call to `grandpa`. The effect of using `` `(,! ,cut) `` at the end is that once the first result is found and failure forced, the `cut` 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 reached `grandpa`'s `else` clause.

Next, consider the following revision of grandpa, where we swap the last two goals.

```
(define grandpa
  (lambda (goal b cut)
    (cond*
      [(to-show '(OLD YOUNG) goal b)
       => (show `(,father OLD PARENT)
            (also `(,! ,cut)
              (also `(,father PARENT YOUNG) empty)))]
      [(to-show '(OLD YOUNG) goal b)
       => (show `(,father OLD PARENT)
            (also `(,mother PARENT YOUNG) empty))]
      [else (cut)])))
```

This yields

```
> (solve 10 `(,grandpa X Y))
((john pete) (john polly))
```

OLD.0 cannot be re-instantiated because of the cut, but PARENT.0 and YOUNG.0 can. As failure works its way back into the same call to father, new instantiations for PARENT.0 and YOUNG.0 are found. If we run out of data to match (PARENT.0 YOUNG.0), then instead of looking for the next match of (OLD.0 PARENT.0), we invoke the cut that was passed to ! and whose behavior is the same as if it had reached grandpa's else clause. So, we have merely two results. One way to describe this program 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 results, because then pete would be the first father and he has no grandchildren.

Finally, we have the last variation, where the `(,! ,cut) is the first thing to show.

```
(define grandpa
  (lambda (goal b cut)
    (cond*
      [(to-show '(OLD YOUNG) goal b)
       => (show `(,! ,cut)
            (also `(,father OLD PARENT)
              (also `(,father PARENT YOUNG) empty)))]
      [(to-show '(OLD YOUNG) goal b)
       => (show `(,father OLD PARENT)
            (also `(,mother PARENT YOUNG) empty))]
      [else (cut)])))
```

```
> (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, cut is invoked. This is the same as ignoring the second

clause, altogether. If we reorganize the clauses in `father` as we did at the end of the previous example, we get the same four results. If `pete` were the father of betty, too, then `betty` would show up as `sam`'s grandchild. She appears only once, however, because the cut precluded us from using the second clause in all three examples.

It is okay to use `` `(,! ,cut) `` more than once within a single clause. Be careful, however, to fully appreciate its consequences each time you use it. Such a powerful control mechanism must be handled with great care.

We can redefine `grandpa` by using a function, like `grandpa-through-son`, but this time we must account for the free lexical variable, `cut`.

```
(define grandpa
  (lambda (goal b cut)
    (cond*
      [(to-show '(OLD YOUNG) goal b)
       => (grandpa-through-son/cut cut)]
      [(to-show '(OLD YOUNG) goal b)
       => (show `(,father OLD PARENT)
            (also `(,mother PARENT YOUNG) empty))]
      [else (cut)])))

(define grandpa-through-son/cut
  (lambda (cut)
    (show `(,! ,cut)
      (also `(,father OLD PARENT)
        (also `(,father PARENT YOUNG) empty)))))
```

An idiom is to cut and then fail right away, so that we might see something like this: `(show ... (also ... (also `(,! ,cut) (also `(,fail) empty))))`. What this idiom says is that we want to exit this call 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 grand*pa* under any circumstances.

```
(define grandpa
  (lambda (goal b cut)
    (cond*
      [(to-show '(OLD YOUNG) goal b)
       => (show `(,mother OLD CHILD)
            (also `(,! ,cut)
              (also `(,fail) empty)))]
      [(to-show `(OLD YOUNG) goal b)
       => (show `(,father OLD PARENT)
            (also `(,father PARENT YOUNG) empty))]
      [(to-show '(OLD YOUNG) goal b)
       => (show `(,father OLD PARENT)
            (also `(,mother PARENT YOUNG) empty))]
      [else (cut)])))
```

We can define an antecedent, `‘(,!fail ,cut)`, to support this idiom.

```
(define !fail
  (lambda (list-of-cut b cut)
    ((car list-of-cut))))
```

3.2 Interfacing Scheme Functions

Suppose we want to restrict the results 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
  (lambda (goal b cut)
    (cond*
      [(to-show '(OLD YOUNG) goal b)
       => (show `(,father OLD PARENT)
            (also `(,starts-with-p? PARENT)
              (also `(,father PARENT YOUNG) empty)))]
      [else (cut)])))

(define starts-with-p?
  (lambda (args b cut)
    (if (uc-symbol? (car args))
        (error 'starts-with-p? "Variable found: ~s." (car args)))
    (if (not (symbol? (car args)))
        (error 'starts-with-p? "Non-symbol found: ~s." (car args)))
    (let ([char (string-ref (symbol->string (car args)) 0)])
      (if (string=? (string char) "p")
          ((axiom b) cut)
          (cut)))))

> (solve 10 `(,grandpa X Y))
((sam sal) (sam pat))
```

Let's look closely at the Scheme function `starts-with-p?`. Like other user functions, it takes the same arguments: a list of cooked operands (Here we pass only one cooked operand.), a bundle, and a cut. If an argument in `args` is needed for the actual test, then it cannot be a variable, that is, it cannot be an uppercase symbol (`uc-symbol?`). It is certainly possible for some of the arguments to evaluate to variables, but in this case it would not be meaningful. If the test, `string=?` is true, then we take the success path, which is *always* `((axiom b) cut)`, otherwise we take the failure path, which is *always* `(cut)`.

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 three arguments, and that they exit the same way: `((axiom b) cut)` for success and `(cut)` for failure.

We abstract this with a new operator, `pred`. (We hesitate to include this operator, since we are trying to minimize the tools that we use. Here we are using `apply`, but as we have shown, its use is not strictly necessary. Furthermore, we use `apply` in the definition of `fun`, below, but again, it is not strictly necessary.)

```
(define pred
  (lambda (p)
    (lambda (args b cut)
      (approve-apply 'pred p args)
      (if (apply p args) ((axiom b) cut) (cut)))))
```

```
(define approve-apply
  (lambda (name proc args)
    (if (not (procedure? proc))
      (error name "Non-procedure found: ~s" proc))
    (if (ormap uc-symbol? args)
      (error name "Variable found: ~s" args))))
```

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 could call *any* Scheme function and `unify` the results of the call with something else. For example, we could use `` `(,(fun +) A B 5) `` as an antecedent. Since the arguments are already cooked by this time, we know that all but the last argument should be a value, so we are safe to apply the Scheme function to those values. We then `unify` the result with the (cooked) last argument. If that last argument is an uninstantiated variable or contains an uninstantiated variable, then this causes the exiting bundle's substitution to be larger than the entering bundle's.

The functions `pred` (and `fun`) allow the user to treat Scheme `predicates` (and `functions`) as logic predicates (and logic functions). Since functions return a value, that value gets unified with the (possibly uninstantiated) additional argument. Unlike with relations, all other arguments to `predicates` and `functions` must be instantiated.

```
(define fun
  (lambda (f)
    (lambda (args b cut)
      (let ((args (rotate-right args)))
        (approve-apply 'fun f (cdr args))
        (cond
          [(to-show/prepared (apply f (cdr args)) (car args) b)
           => (lambda (exit-b) ((axiom exit-b) cut))]
          [else (cut)])))))

(define rotate-right
  (lambda (ls)
    (cond
      [(null? (cdr ls)) (cons (car ls) '())]
      [else (let ([rcdr (rotate-right (cdr ls))])
              (cons (car rcdr) (cons (car ls) (cdr rcdr))))])))
```

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 (list-of-one-goal b cut)
    (call (car list-of-one-goal)
      (bundle
        (lambda (global-c subst)
          (lambda (fk) (cut)))
        (bundle->counter b)
        (bundle->subst b))
      (lambda () ((axiom b) cut)))))
```

And here is a simple example of `fails`.

```
(define grandpa
  (lambda (goal b cut)
    (cond*
      [(to-show '(OLD YOUNG) goal b)
       => (show `(,father OLD PARENT)
            (also `(,fails (,starts-with-p? PARENT))
              (also `(,father PARENT YOUNG) empty)))]
      [else (cut)])))
```

```
> (solve 6 `(,grandpa X Y))
((john pete))
```

To determine if a variable has been instantiated, use this argument to `show`: `` `(,instantiated X) `` with this definition.

```
(define instantiated
  (lambda (list-of-one-arg b cut)
    (if (not (uc-symbol? (car list-of-one-arg)))
      ((axiom b) cut)
      (cut))))
```

By the time `instantiated` has been entered, `list-of-one-arg` has been cooked. So, at this point, `(car list-of-one-arg)` is either some `X.`i or the value of `X.`i.

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

```
(define view-subst
  (lambda (list-of-one-arg b cut)
    (write (car list-of-one-arg))
    (newline)
    (pretty-print (flatten-subst (bundle->subst b)))
    (newline)
    ((axiom b) cut))))
```

We have used the function `flatten-subst` for readability.

Here is a new definition of `grandpa`, like an earlier one, but with viewing enabled.

```
(define grandpa
  (lambda (goal b cut)
    (cond*
      [(to-show '(OLD YOUNG) goal b)
       => (show `(,father OLD PARENT)
            (also `(,father PARENT YOUNG)
              (also `(,view-subst YOUNG) empty)))]
      [else (cut)])))
```

```
> (solve 10 `(,grandpa X Y))
pete
((OLD.0 == X)
 (YOUNG.0 == Y)
 (X == john)
 (PARENT.0 == sam)
 (Y == pete))

sal
((OLD.0 == X)
 (YOUNG.0 == Y)
 (X == sam)
 (PARENT.0 == pete)
 (Y == sal))
```

```
pat
((OLD.0 == X)
 (YOUNG.0 == Y)
 (X == sam)
 (PARENT.0 == pete)
 (Y == pat))

((john pete) (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. The only constraint is that this description assumes that the reader understands the interplay of unification with substitutions.

4 Recursive definitions

Suppose that we wanted 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 `leon`.

```
(define ancestor
  (lambda (goal b cut)
    (cond*
      [(to-show '(OLD YOUNG) goal b)
       => (show `(,father OLD YOUNG) empty)]
      [(to-show '(OLD YOUNG) goal b)
       => (show `(,father OLD NOT-SO-OLD)
            (also `(,ancestor NOT-SO-OLD YOUNG) empty))]
      [else (cut)])))

> (solve 9 `(,ancestor john X))
((john sam)
 (john harry)
 (john pete)
 (john polly)
 (john leon)
 (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
  (lambda (goal b cut)
    (cond*
      [(to-show '(YOUNG-A YOUNG-B OLD) goal b)
       => (show `(,ancestor OLD YOUNG-A)
            (also `(,ancestor OLD YOUNG-B) empty))]
      [else (cut)])))

> (solve 4 `(,common-ancestor pat leon X))
((pat leon john) (pat leon sam))
```

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

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-ancestor
  (lambda (goal b cut)
    (cond*
      [(to-show '(YOUNG-A YOUNG-B OLD NOT-SO-OLD) goal b)
       => (show `(,common-ancestor YOUNG-A YOUNG-B NOT-SO-OLD)
            (also `(,ancestor OLD NOT-SO-OLD) empty))]
      [else (cut)])))

> (solve 4 `(,younger-ancestor pat leon john X))
((pat leon john sam))
```

We finally come to the problem that we are most interested in, which is how do we determine the youngest common ancestor. We already know that `pete` and `leon` 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
  (lambda (goal b cut)
    (cond*
      [(to-show '(YOUNG-A YOUNG-B OLD) goal b)
       => (show `(,common-ancestor YOUNG-A YOUNG-B OLD)
            (also
              `(,fails
                 (,younger-ancestor YOUNG-A YOUNG-B OLD NOT-SO-OLD))
              empty))]
      [else (cut)])))

> (solve 4 `(,youngest-common-ancestor pat leon X))
((pat leon sam))
```

The tricky part of this program is that if we find a younger ancestor, then the one we chose 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`, etc. Although we don't use recursion to implement the seven functions, our model relies heavily on the idea that users will be 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 developed using `letrec`.

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

In this example we choose to export `youngest-common-ancestor`, although we could export the others, as well.

5 Two famous problems

Two famous problems that we discuss are the so-called `append` problem and the type-inference problem of a typed variant of the lambda calculus with constants and primitives. We start with the `append` problem.

5.1 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 (ls1 ls2)
    (cond
      [(null? ls1) ls2]
      [else (cons (car ls1) (concat (cdr ls1) ls2))])))
```

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

or the equivalent

```
> (solve 1 `(,(fun concat) (a b c) (u v) Q))
(((a b c) (u v) (a b c u v)))
```

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

```
(define concat
  (lambda (goal b cut)
    (cond*
      [(to-show '(() X X) goal b)
       => axiom]
      [(to-show '((X . XS) YS (X . ZS)) goal b)
       => (show `(,concat XS YS ZS) empty)]
      [else (cut)])))
```

```
> (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 result 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.

```
> (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)`.

```
> (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?

```
> (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?

```
> (solve 6 `(,concat Q R S))
((() S S)
 ((X.0) Zs.0 (X.0 . Zs.0))
 ((X.0 X.1) Zs.1 (X.0 X.1 . Zs.1))
 ((X.0 X.1 X.2) Zs.2 (X.0 X.1 X.2 . Zs.2))
 ((X.0 X.1 X.2 X.3) Zs.3 (X.0 X.1 X.2 X.3 . Zs.3))
 ((X.0 X.1 X.2 X.3 X.4) Zs.4 (X.0 X.1 X.2 X.3 X.4 . Zs.4)))
```

Here we see that the empty list and any list yields that list. Then we get all sorts of constructed lists with the first N elements of the list chosen as variables of that length. This might be a good time to run this with a `(,view-subst arg)` strategically placed to watch the action. There is no bound on the number of results.

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

```
> (solve 6 '(,concat Q (u v) (a b c . R)))
(((a b c) (u v) (a b c u v))
 ((a b c X.3) (u v) (a b c X.3 u v))
 ((a b c X.3 X.4) (u v) (a b c X.3 X.4 u v))
 ((a b c X.3 X.4 X.5) (u v) (a b c X.3 X.4 X.5 u v))
 ((a b c X.3 X.4 X.5 X.6) (u v) (a b c X.3 X.4 X.5 X.6 u v))
 ((a b c X.3 X.4 X.5 X.6 X.7) (u v) (a b c X.3 X.4 X.5 X.6 X.7 u v)))
```

The first result 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.

We can even get an unbounded number of results with a single variable.

```
> (solve 6 '(,concat Q () Q))
((() () ())
 ((X.0) () (X.0))
 ((X.0 X.1) () (X.0 X.1))
 ((X.0 X.1 X.2) () (X.0 X.1 X.2))
 ((X.0 X.1 X.2 X.3) () (X.0 X.1 X.2 X.3))
 ((X.0 X.1 X.2 X.3 X.4) () (X.0 X.1 X.2 X.3 X.4)))
```

Again, the first result is the one that we expect, but the others make sense, too, since no matter what we replace the variables `X.`i with, we will create a legitimate result.

A simple 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 I defined `concat` globally, I decided it would be wise to avoid overriding the built-in Scheme function, `append`.

5.2 The Type Inferencing Problem

The second famous problem is the type-inferencing problem. We start by considering the rules that include integers and booleans. Next, we include some familiar primitives. When we are comfortable with those features we include conditionals, followed by variables, then `lambda` expressions, and finally applications and `fix` expressions. Type inferencing allows for the system to determine a type if the expression has one. This particular system has unique types. Because of this property, we are free to always pass `1` to `solve`, which we abstract with `solution`. (As an exercise, try to write `solution` passing in the appropriate `initial-bundle` and `initial-cut` to `query`, instead.)

```
(define solution
  (lambda (guide/goal)
    (let ((ls (solve 1 guide/goal)))
      (if (null? ls) #f (car ls)))))
```

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

In the program below, we have used different kinds of constants in different roles. Strings are used to indicate types: `"bool"`, `"int"`, and `"->"`. Lower case symbols (`lc-symbol?`) denote lexical variables of the language, and the integer and boolean constants represent themselves. Later, we show another less error-prone representation. The name of the function is `!-`. It corresponds to the mathematical symbol $\vdash$ (turnstyle) and reads "we can infer". That is, from looking at the first rule, we can read it as, "From `G` and `N` *we can infer* the type `"int"` provided that `N` is an integer." For now, we leave `G` unspecified.

```
(define !-
  (lambda (goal b cut)
    (cond*
      [(to-show '(G N "int") goal b)
       => (show `(,(pred integer?) N)
            (also `(,! ,cut) empty))]
      [(to-show '(G B "bool") goal b)
       => (show `(,(pred boolean?) B)
            (also `(,! ,cut) empty))]
      [else (!-1 goal b cut)])))

(define !-1 fail)
```

```
> (solution `(,!- G 17 "int"))
(G 17 "int")

> (solution `(,!- G 17 X))
(G 17 "int")
```

In the first example, we verify that `17` is of type `"int"`. In the second example, we don't know the type, but whatever is instantiated to `X` will be the type. In this case, `X` is instantiated to `"int"`.

By defining `!-1` to be `fail`, the last line has the same semantics that it had before (i.e., `(cut)`), but later we can redefine `!-1` giving us a kind of weak dynamic inheritance as long as all recursive calls refer to `!-` and not to `!-1`. It would not be difficult to move to a complete hierarchical structure, rather than this linear one, but it would take us beyond where we need to go. The point is that we can sequentially *construct* and *test* the type inferencer.

Next, we add the primitives: `zero?`, `sub1`, and `+`.

```
(define !-1
  (lambda (goal b cut)
    (cond*
      [(to-show '(G (zero? N) "bool") goal b)
       => (show `(,! ,cut)
            (also `(,!- G N "int") empty))]
      [(to-show '(G (sub1 N) "int") goal b)
       => (show `(,! ,cut)
            (also `(,!- G N "int") empty))]
      [(to-show '(G (+ N M) "int") goal b)
       => (show `(,! ,cut)
            (also `(,!- G N "int")
              (also `(,!- G M "int") empty)))]
      [else (!-2 goal b cut)])))

(define !-2 fail)
```

```
> (solution `(,!- G (zero? 24) X))
(G (zero? 24) "bool")
```

```
> (solution `(,!- G (zero? (+ 24 50)) X))
(G (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 result in the second example must be of type `"bool"`. We can, of course, make more complicated examples using `zero?`, `sub1`, and `+`, but they will ultimately have `"int"` or `"bool"` type. For example,

```
> (solution `(,!- G (zero? (sub1 (+ 18 (+ 24 50)))) X))
(G (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 the 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 redefining `!-2`, `!-` can now handle `if` expressions.

```
(define !-2
  (lambda (goal b cut)
    (cond*
      [(to-show '(G (if Test Conseq Alt) T) goal b)
       => (show `(,! ,cut)
            (also `(,!- G Test "bool")
              (also `(,!- G Conseq T)
                (also `(,!- G Alt T) empty))))]
      [else (!-3 goal b cut)])))

(define !-3 fail)
```

```
> (solution `(,!- G (if (zero? 24) 3 4) X))
(G (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"`.

Let's raise the ante just a bit by worrying about lexical variables, which are represented using lower-case symbols (`lc-symbol?`).

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`. `G` is a type environment that associates lexical variables with types. So far we have ignored `G`, but now we consider its content. We redefine `!-3` to include a clause for variables.

```
(define !-3
  (lambda (goal b cut)
    (cond*
      [(to-show '(G V T) goal b)
       => (show `(,(pred lc-symbol?) V)
            (also `(,! ,cut)
              (also `(,env G V T) empty)))]
      [else (!-4 goal b cut)])))

(define !-4 fail)
```

```
(define env
  (lambda (goal b cut)
    (cond*
      [(to-show '((V TV G) V TV) goal b)
       => (show `(,! ,cut) empty)]
      [(to-show '((W TW G) V TV) goal b)
       => (show `(,! ,cut)
            (also `(,unequal? W V)
              (also `(,env G V TV) empty)))]
      [else (cut)])))

(define unequal?
  (lambda (goal b cut)
    (cond*
      [(to-show '(X X) goal b)
       => (show `(,fail) empty)]
      [(to-show '(X Y) goal b)
       => (show `(,succeed) empty)]
      [else (cut)])))

> (solution `(,env (b "int" (a "bool" G)) a X))
((b "int" (a "bool" G)) a "bool")

> (solution `(,!- (a "int" G) (zero? a) X))
((a "int" G) (zero? a) "bool")

> (solution `(,!- (b "bool" (a "int" G)) (zero? a) X)))
((b "bool" (a "int" G)) (zero? a) "bool")
```

The first example tests `env`. The environment starts out with `"int"` bound to `b` and `"bool"` bound to `a`. The second clause succeeds, since we are looking up `a`, and then the first clause succeeds, since we find `a`. In the second result, we have one item in the type environment and the first `env` clause is followed. In the third example, we have two items in the type environment, so we take the second clause, 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 variables, we can consider the rule for `lambda` expressions by redefining `!-4`.

```
(define !-4
  (lambda (goal b cut)
    (cond*
      [(to-show '(G (lambda (V) Body) ("->" TV TBody)) goal b)
       => (show `(,! ,cut)
            (also `(,!- (V TV G) Body TBody) empty))]
      [else (!-5 goal b cut)])))

(define !-5 fail)
```

```
> (solution `(,!- (b "bool" (a "int" G)) (lambda (x) (+ x 5)) X))
((b "bool" (a "int" G)) (lambda (x) (+ x 5)) ("->" "int" "int"))

> (solution `(,!- (b "bool" (a "int" G)) (lambda (x) (+ x a)) X))
((b "bool" (a "int" G)) (lambda (x) (+ x a)) ("->" "int" "int"))

> (solution `(,!- G (lambda (a) (lambda (x) (+ x a))) X))
(G (lambda (a) (lambda (x) (+ x a))) ("->" "int" ("->" "int" "int")))
```

In the first result, we see that we have an arrow (`"->"`) type. The left argument of the arrow type is the type of argument coming into the function, the right argument of the arrow type is the type of the answer 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 result states that the argument is an integer, but consults the type environment to make sure that the argument going out is an integer. In the third example, we forget about the first environment, because there are no free variables in the expression. We see that the argument coming in is an integer, but the result is an arrow type, which takes in an integer and returns an integer.

We come next to application. In determining the type of an application, we can guess that the operator in an application is some function type. Furthermore, once we know that type, we know that the result of the function type is the same as the result of the entire application and we know that the operand of the application must be the type of the input to the function. We redefine `!-5` and include clauses for both `fix` and application. They are defined at the same time, because the application clause must be the *last* one.

```
(define !-5
  (lambda (goal b cut)
    (cond*
      [(to-show '(G (fix E) T) goal b)
       => (show `(,! ,cut)
            (also `(,!- G E ("->" T T)) empty))]
      [(to-show '(G (Rator Rand) TBody) goal b)
       => (show `(,! ,cut)
            (also `(,!- G Rator ("->" TV TBody))
              (also `(,!- G Rand TV) empty)))]
      [else (cut)])))
```

```
> (solution `(,!- G (lambda (f) (lambda (x) ((f x) x))) X))
(G (lambda (f) (lambda (x) ((f x) x)))
   ("->" ("->" TV.4 ("->" TV.4 TBody.2)) ("->" TV.4 TBody.2)))
```

Here, the type of `f` is `("->" TV.4 ("->" TV.4 TBody.2))`, so the type of `x` must be `TV.4`, and the type of `(lambda (x) ((f x) x))` must be `("->" TV.4 TBody.2)`. As should be evident, once we add a rule for application, things start to get a bit tricky.

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

```
> (solution `(,!- G
                ((fix (lambda (sum)
                        (lambda (n)
                          (if (zero? n)
                            0
                            (+ n (sum (sub1 n)))))))
                 10)
                X))
(G ((fix (lambda (sum)
           (lambda (n)
             (if (zero? n)
               0
               (+ n (sum (sub1 n)))))))
    10)
  "int")
```

Application is the *last* rule, since it is a catchall case. A more robust solution would be to preprocess the expression to tag the individual expressions, in the style of `if`-espressions. We choose the following tags: `app`, `var`, `intc`, and `boolc`.

```
(define expr2tagged
  (lambda (x)
    (cond
      [(lc-symbol? x) `(var ,x)]
      [(integer? x) `(intc ,x)]
      [(boolean? x) `(boolc ,x)]
      [(zero?-exp? x) `(zero? ,(expr2tagged (cadr x)))]
      [(sub1-exp? x) `(sub1 ,(expr2tagged (cadr x)))]
      [(+-exp? x) `(+ ,(expr2tagged (cadr x)) ,(expr2tagged (caddr x)))]
      [(if-exp? x)
       `(if ,(expr2tagged (cadr x))
          ,(expr2tagged (caddr x))
          ,(expr2tagged (cadddr x)))]
      [(lambda-exp? x) `(lambda ,(cadr x) ,(expr2tagged (caddr x)))]
      [(fix-exp? x) `(fix ,(expr2tagged  (cadr x)))]
      [else `(app ,(expr2tagged (car x)) ,(expr2tagged (cadr x)))])))
```

```
(define !-
  (lambda (goal b cut)
    (cond*
      [(to-show '(G (var V) T) goal b)
       => (show `(,! ,cut)
            (also `(,env G V T) empty))]
      [(to-show '(G (intc N) "int") goal b)
       => (show `(,! ,cut) empty)]
      [(to-show '(G (boolc B) "bool") goal b)
       => (show `(,! ,cut) empty)]
      [(to-show '(G (zero? N) "bool") goal b)
       => (show `(,! ,cut)
            (also `(,!- G N "int") empty))]
      [(to-show '(G (sub1 N) "int") goal b)
       => (show `(,! ,cut)
            (also `(,!- G N "int") empty))]
      [(to-show '(G (+ N M) "int") goal b)
       => (show `(,! ,cut)
            (also `(,!- G N "int")
              (also `(,!- G M "int") empty)))]
      [(to-show '(G (if Test Conseq Alt) T) goal b)
       => (show `(,! ,cut)
            (also `(,!- G Test "bool")
              (also `(,!- G Conseq T)
                (also `(,!- G Alt T) empty))))]
      [(to-show '(G (lambda (V) Body) ("->" TV TBody)) goal b)
       => (show `(,! ,cut)
            (also `(,!- (V TV G) Body TBody) empty))]
      [(to-show '(G (app Rator Rand) TBody) goal b)
       => (show `(,! ,cut)
            (also `(,!- G Rator ("->" TV TBody))
              (also `(,!- G Rand TV) empty)))]
      [(to-show '(G (fix E) T) goal b)
       => (show `(,! ,cut)
            (also `(,!- G E ("->" T T)) empty))]
      [else (cut)])))
```

We make two 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 clauses to use `pred`. Therefore, we can include the cut at the top of each clause. 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 functions).

Here are four, perhaps unexpected, examples.

```
> (solution `(,!- G X ("->" "int" "int")))
((Id.0 ("->" "int" "int") G.2) (var Id.0) ("->" "int" "int"))

> (solution `(,!- G (L (F) B) ("->" "int" "int")))
(G (lambda (Id.2) (var Id.2)) ("->" "int" "int"))

> (solution `(,!- G (H R (Q Z Y)) T))
((Id.12 "int" G.4) (+ (var Id.12) (+ (var Id.12) (var Id.12))) "int")

> (solution `(,!- G (H R (Q Z Y)) (T U V)))
(G (lambda (Id.8) (+ (var Id.8) (var Id.8))) ("->" "int" "int"))
```

The first example attempts to find a program whose type is `("->" "int" "int")`, but instead finds a type environment that binds that type to the variable `Id.0`, and then the program is trivially `(var Id.0)`. The second example produces a program, given the type. This is answering the question, "What program *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` clause 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., `+`), that determines `Q` and `H`. Next it infers 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 Final thoughts

In an expression such as `` `(,foo X a ,y) `` we have three different kinds of values. The symbol `a` is already a value, the lexical variables, `foo` and `y` become values as part of the construction of the list, and the variable `X` becomes a value (or is shared with another variable) when the list unifies with another value. Having one expression that allows for their intermingling clarifies why we need to manage their scopes. Second, by treating that expression as a function call waiting for some additional arguments, we integrate the calling mechanism of Scheme with the calling mechanism of our logic system. In Scheme, we could write `` `(,foo X a ,y) `` and treat this as if we had written `(let ([q `(,foo X a ,y)]) ((car q) (cdr q)))`, since the `foo` and `y` are evaluated in both cases. This gives us a way to think about evaluation without using `eval`, but more importantly, it gives us the ability to *evaluate* the pieces of the application in each way. That is what makes this approach possible.

We introduce `show-all`, which improves the syntax, but requires the use of recursion in its locally-defined function, `also-all`.

```
(define show-all
  (letrec ([also-all
            (lambda (ants)
              (if (null? ants)
                empty
                (also (car ants) (also-all (cdr ants)))))])
    (lambda (ants)
      (if (null? ants)
        axiom
        (show (car ants) (also-all (cdr ants)))))))
```

The improvement in syntax is that the set of (unquasiquoted) antecedents is wrapped in a quasiquoted list.

```
(define grandpa
  (lambda (goal b cut)
    (cond*
      [(to-show '(OLD YOUNG) goal b)
       => (show-all
            `((,father OLD PARENT)
              (,father PARENT YOUNG)))]
      [else (cut)])))
```

Now, we can say that the unsullied logic system relies on four functions: `if*`, `to-show`, `query`, and `show-all`. If we include the sullied antecedents, however, we need the definition of `axiom`.

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. Of course, fancy macros 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.

7 Acknowledgments

This tutorial would not have been possible without the earlier work on implementing logic systems with Anurag Mendhekar. The work with Anurag led to Jon Rossie using a logic system in the development of his results in his dissertation. His utilization of the tool helped us understand how we had to weave things together. But, it wasn't until my work with Mitch Wand and Chris Haynes in EOPL2 on the material on unification, substitutions, and logic programming that I began to see that there was a chance for a poorman's solution. Steve Ganz's dissertation also needed an inferencing system. Over the years, Steve began to see how we could make the seamlessness of Scheme possible by showing how to break up the monolithic set of rules into functions. Each function then represents a relation, which could be

invoked as a function. Then we noticed that each of 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, which led to a poorman's solution.

8 APPENDIX A

The macro, `cond*` (below), is identical to `cond` as it appears in the R^5 Scheme Standard, except that there are three places where `if` has been replaced by `if*`, the last argument to `if*` has been wrapped in a `(lambda () ...)`, and the second argument has been left alone, except that it has been passed the lexical variable `fk` and this call has been wrapped in a `(lambda (fk) ...)`, where necessary. In other words, for the second argument, we have applied η^{-1} in two of the three places to avoid potential bad paths.

```
(define-syntax cond*
  (syntax-rules (else =>)
    [(_ (else result1 result2 ...))
     (begin result1 result2 ...)]
    [(_ (test => result))
     (let ([temp test])
       (if temp (result temp)))]
    [(_ (test => result) clause1 clause2 ...)
     (let ([temp test])
       (if* temp
         (lambda (fk) ((result temp) fk))
         (lambda () (cond* clause1 clause2 ...))))]
    [(_ (test)) test]
    [(_ (test) clause1 clause2 ...)
     (let ([temp test])
       (if* temp
         temp
         (lambda () (cond* clause1 clause2 ...))))]
    [(_ (test result1 result2 ...))
     (if test (begin result1 result2 ...))]
    [(_ (test result1 result2 ...) clause1 clause2 ...)
     (if* test
       (lambda (fk) ((begin result1 result2 ...) fk))
       (lambda () (cond* clause1 clause2 ...)))]))
```

9 APPENDIX B

Of course, `cond*` is not strictly necessary. We can rewrite `father` and all the other definitions that use `cond*` by replacing occurrences of `cond*` with appropriate nestings of the function `if*`. Because we find this quite a bit more difficult to comprehend, we have opted for using `cond*`. but it should be kept in mind that the

decision to use `cond*` is merely a matter of convenience, having nothing whatsoever to do with the semantics of our definitions. Here is what the definition of `father` looks like in the absense of `cond*`.

```
(define father
  (lambda (goal b cut)
    (let ([possb (to-show '(john sam) goal b)])
      (if* possb
        (lambda (fk)
          ((axiom possb) fk))
        (lambda ()
          (let ([possb (to-show '(sam pete) goal b)])
            (if* possb
              (lambda (fk)
                ((axiom possb) fk))
              (lambda ()
                (let ([possb (to-show '(pete sal) goal b)])
                  (if* possb
                    (lambda (fk)
                      ((axiom possb) fk))
                    cut)))))))))))
```

10 APPENDIX C

For those who find the nested structure of the antecedents to be clumsy, we propose the macro `show-all`, which takes a nonegative number antecedents.

```
(define-syntax show-all
  (syntax-rules ()
    [(_) axiom]
    [(_ ant0 ant1 ...) (show ant0 (also-all* ant1 ...))]))

(define-syntax also-all*
  (syntax-rules ()
    [(_) empty]
    [(_ ant0 ant1 ...) (also ant0 (also-all* ant1 ...))]))

(define grandpa
  (lambda (goal b cut)
    (cond*
      [(to-show '(OLD YOUNG) goal b)
       => (show-all
            `(,father OLD PARENT)
            `(,father PARENT YOUNG))]
      [else (cut)])))
```

11 Appendix D

This is the definition of `fix`.

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