Understanding Unification (5/11/2004)

Daniel P. Friedman, William E. Byrd, David W. Mack
Computer Science Department, Indiana University
Bloomington, IN 47405, USA

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

Here we continue the discussion of logic programming we began with *Outcome-Oriented Programming*, *Relation-Oriented Programming*, and *Logic-Oriented Programming*. In this installment we present commitments, substitutions, and the unification of two terms with respect to a substitution.

We use the following naming conventions:

`w, x, y, z` represent logic variables,
`t, u, v` represent terms,
s, s', s'' represent substitutions.

A *commitment* is a binding pair composed of a logic variable and a term. In the commitment `(x . t)`, `x` is the logic variable and `t` is the term. The Kanren functions `commitment->var` and `commitment->term` return the logic variable and term of a commitment, respectively. We say that a commitment *binds* a variable to a term. If the term is known to be a variable, we say that the two variables within the commitment *co-refer* or *share*.

A commitment must satisfy the following constraints.

1. The logic variable and the term must differ. Therefore, `(x . x)` is not a valid commitment.
2. The logic variable may not be the anonymous variable, `_`, nor may the term contain the anonymous variable. Neither `(_ . t)` nor `(x . (_ . t))` are valid commitments.

A *substitution* is a collection of commitments. In the Kanren system, a substitution is represented as a list of pairs, also known as an *association list*.

```
((w . (3 x u)) (x . y) (z . 5))
```

A variable may appear in the term of a commitment, even if that variable does not appear in the variable position of any commitment in the substitution. In the substitution above, for example, `y` appears only as a term. A variable may not appear in the logic variable position of more than one commitment within a substitution, thus guaranteeing that these variables form a set[1]. Thus, `((x . y) (x . z))` is

[1] In other words, for any substitution `subst`, the list returned by `(map commitment->var subst)` must not contain duplicate values.

not a valid substitution, since x appears in the variable position of more than one commitment. A *free* variable is not committed (has no binding pair) in the substitution, but it may appear as a term (or in a term) associated with some other variables. A *bound* variable is one that is committed, but perhaps to another variable. When we pass a variable that has been determined to be bound, we *always* pass the commitment, which includes the term that it binds.

As logic programmers, often we must determine if and how two terms can be made equal to each other, in the sense of Scheme's `equal?` operator. For example, we may wish to know the values of `x` and `y` for which the terms `(x . 3)` and `(7 . y)` are `equal?`. For our example, we assume that both `x` and `y` are unbound logic variables. Obviously, these terms are `equal?` only when `y` is bound to the value 3 and `x` is bound to the value `7`. The commitments of `y` to 3 and `x` to `7` are naturally recorded in the substitution `((y . 3) (x . 7))`.

In technical jargon, we have just *unified* the terms `(x . 3)` and `(7 . y)` in the context of the *empty substitution*, producing a new substitution `((y . 3) (x . 7))`. We can use Kanren's `unify` function to verify that our answer is correct.

```
> (exists (x y) (unify `(,x . 3) `(7 . ,y) empty-subst))
```

We wrap the call to `unify` within the `exists` form in order to introduce `x` and `y` as fresh logic variables. Both lists are quasiquoted, so that we can unquote the logic variables. Finally, we specify the empty substitution explicitly. Kanren responds by returning the expected substitution.

```
((#[logical-variable y] . 3) (#[logical-variable x] . 7))
```

We know that unifying `(x . 3)` and `(7 . y)` in the context of the empty substitution succeeds, and returns a new substitution. Using Kanren's `subst-in` function, we can *apply this new substitution* to our original terms to yield the ground[2] term `(7 . 3)`.

```
> (exists (x y)
    (let ([t `(,x . 3)] [u `(7 . ,y)])
      (let ([subst (unify t u empty-subst)])
        (let ([t2 (subst-in t subst)] [u2 (subst-in u subst)])
          (printf "t2: ~s  u2: ~s  t2 = u2: ~s\n" t2 u2 (equal? t2 u2))))))
t2: (7 . 3)  u2: (7 . 3)  t2 = u2: #t
```

Not all pairs of terms can be successfully unified. The pair `(x . 5)` cannot be made equal to the pair `(3 . x)`, regardless of the value we substitute for `x`. The Kanren `unify` function returns `#f` instead of a valid substitution whenever unification of two terms fails.

```
> (exists (x) (unify `(,x . 5) `(3 . ,x) empty-subst))
#f
```

[2] A *ground* term is a term that does not contain logic variables.

These last two examples illustrate the fundamental idea behind unification, which can be expressed in Kanren notation as follows.

Unification Principle Given two terms t and u, and a substitution s, if `(unify` t u s`)` succeeds by returning a valid substitution[3] s', then,

`(equal? (subst-in` t s'`) (subst-in` u s'`))` $\Rightarrow$ `#t`.

If `(unify` t u s`)` fails by returning `#f`, then for *every* valid substitution s'',

`(equal? (subst-in` t s''`) (subst-in` u s''`))` $\Rightarrow$ `#f`.

So far we have performed unification in the context of the empty substitution. We can also unify two terms in the context of a substitution returned from a previous unification. For example, what happens if first we unify `y` with 3, and then we attempt to unify `x` with `y`?

```
> (exists (x y) (unify x y (unify y 3 empty-subst)))
((#[logical-variable x] . 3) (#[logical-variable y] . 3))
```

Not surprisingly, unification succeeds, with both `x` and `y` bound to 3 in the resulting substitution. Since `x` and `y` unified, we know that we can apply the resulting substitution to produce `equal?` terms.

```
> (exists (x y)
    (let ([subst (unify x y (unify y 3 empty-subst))])
      (equal? (subst-in x subst) (subst-in y subst))))
#t
```

What happens if we unify `y` with 3, and then use the resulting substitution to try to unify `y` with 4?

```
> (exists (x y) (unify y 4 (unify y 3 empty-subst)))
#f
```

Unification fails, of course, since we cannot substitute 4 for `y` after `y` has been committed to the value 3.

The goal of unifying terms t and u is to return a substitution s' that satisfies the unification principle. When studying the unifier, therefore, it is important to keep in mind the definition of `subst-in`, below.

Given a term, `subst-in` replaces as many logic variables in the term as is possible. Because each logic variable may be bound to a term that contains other logic variables, `subst-in` is recursive. The key to understanding this definition of `subst-in` is to realize that of the three `cond` cases, the second and third simply perform a recursive copy. The `(var? t)` case that handles logic variables is therefore the only interesting case. If the logic variable is bound in the substitution, then the term associated with the variable is the new term in the recursion. If the logic variable is unbound, then the variable itself is returned.

[3] More specifically, `unify` returns the *most general substitution*, also referred to as the *most general unifier* (*mgu*). A unifier is most general if every other unifier can be characterized as an instantiation of it. See, for example, "Unification Theory" by Frank Baader and Wayne Snyder in *The Handbook of Automated Reasoning*, Elsevier Science Publishers B.V., 1999.

```
(define subst-in
  (lambda (t subst)
    (cond
      [(var? t)
       (let*-and[4] t ([ct (assq t subst)])
         (subst-in (commitment->term ct) subst))]
      [(pair? t) (cons (subst-in (car t) subst) (subst-in (cdr t) subst))]
      [else t])))
```

Now that we are familiar with `subst-in`, we are ready to examine the unifier itself. To understand the structure of the unifier, it is important to appreciate that generally a logic variable goes from being free to being bound, and once it is bound, it stays bound.

We begin by examining `unify` below, which assumes that the first two arguments are arbirary terms, including possibly the anonymous variable.

```
(define unify
  (lambda (t u subst)
    (cond
      [(or (eq? t u) (eq? t _) (eq? u _)) subst]
      [(and (var? t) (var? u)) (unify-var/var t u subst)]
      [(var? t) (unify-var/nonvar t u subst)]
      [(var? u) (unify-var/nonvar u t subst)]
      [else (unify-nonvar/nonvar t u subst)])))
```

Structurally, there are five cases to consider. The first case captures any two items that are identical, or where either one is the anonymous variable. The first case does not increase the size of the substitution. The last case calls `unify-nonvar/nonvar`. This case handles anything that is not a variable and requires more generality than the `eq?` predicate, which includes pairs and vectors. This leaves us with three cases. In the third case, we know that `t` is a variable and `u` is a non-variable, so we simply call the `unify-var/nonvar` helper function. The fourth case is symmetric to the third case, with the roles of `t` and `u` reversed. In the second case, which is the only interesting one, we know that both `t` and `u` are variables. We call the help function `unify-var/var`. If `t` and `u` are both bound, we invoke `unify`, passing in the term associated with each binding pair. If `u` and `t` are free variables, we extend the substitution, choosing `t` as the binding's variable and `u` as the binding's term. If `t` is bound and `u` is free we call the `unify-free/bound` helper, passing in the binding pair of `t`. If `u` is bound and `t` is free we call the same function, passing in `u`'s commitment.

[4] The `subst-in` function uses the `let*-and` helper macro, which expands as follows.

```
(let*-and false-exp ([var exp]) body) ⇒ (let ([var exp]) (if var body false-exp))
```

The `let*-and` macro, as with all Scheme binding forms, wraps an implicit `begin` around the body. Although `let*-and` supports multiple `[var exp]` bindings, all occurrences of `let*-and` here use only a single `[var exp]` binding pair.

The first helper function used by `unify` is `unify-var/var`.

```
(define unify-var/var
  (lambda (t u subst)
    (let ([ct (assq t subst)]
          [cu (assq u subst)])
      (cond
        [(and ct cu)
         (unify-i/i (commitment->term ct) (commitment->term cu) subst)]
        [ct (unify-free/bound u ct subst)]
        [cu (unify-free/bound t cu subst)]
        [else (extend-subst t u subst)]))))

(define unify-i/i
  (lambda (t u subst)
    (unify t u subst)))
```

The first clause of `unify-var/var` passes to `unify-i/i` the terms bound to `t` and `u` in the substitution. The name `unify-i/i` is chosen, since both terms passed to `unify-i/i` are *internal* to the substitution. An internal term neither contains nor is the anonymous variable. Although `unify-i/i` naively calls `unify`, later we redefine it to capitalize on knowing that its first two arguments are internal.

The definition of `unify-var/var` is completely symmetric with respect to `t` and `u`, with the exception of the `else` clause. In this asymmetric case, we give "preference" to `t` as the new commitment's logic variable. Even this asymmetry can be removed by replacing the clause using *Chez Scheme's* `random` function, making the choice of the new binding's logic variable arbitrary.

```
(if (zero? (random 2))
  (extend-subst t u subst)
  (extend-subst u t subst))]
```

We can use the same solution in `unify-free/bound` below, if `u` becomes a free variable.

Here are definitions for several more helper functions.

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

(define unify-var/nonvar
  (lambda (t u subst)
    (let ([ct (assq t subst)])
      (if ct
        (unify-nonvar/bound u ct subst)
        (if (pair? u)
          (unify-free/list t u subst)
          (extend-subst t u subst))))))
```

```
(define unify-nonvar/nonvar
  (lambda (t u subst)
    (if (and (pair? t) (pair? u))
      (let*-and #f ([subst (unify (car t) (car u) subst)])
        (unify (cdr t) (cdr u) subst))
      (if (equal? t u) subst #f))))

(define extend-subst
  (lambda (var term subst)
    (cons (commitment var term) subst)))
```

The `unify-var/nonvar` function requires the following simple definitions.

```
(define unify-free/list
  (lambda (t u subst)
    (extend-subst t (or (rebuild-without-anons u) u) subst)))

(define rebuild-without-anons
  (lambda (lst)
    (cond
      [(eq? lst _) (logical-variable '*anon)]
      [(not (pair? lst)) #f]
      [(null? (cdr lst))
       (let ([new-car (rebuild-without-anons (car lst))])
         (and new-car (cons new-car '())))]
      [else
        (let ([new-car (rebuild-without-anons (car lst))]
              [new-cdr (rebuild-without-anons (cdr lst))])
          (if new-car
            (cons new-car (or new-cdr (cdr lst)))
            (and new-cdr (cons (car lst) new-cdr))))])))
```

The `unify-free/list` function simply extends the substitution with a new commitment binding a free variable to a list, after assuring that the list has no occurrences of the anonymous variable.

The local helper function `rebuild-without-anons` busily rebuilds its argument, replacing all occurrences of the anonymous variable with a fresh variable. If either the `car` (or the `cdr`) does not contain an anonymous variable, it uses the original `car` (or the original `cdr`) instead of the copy.

Next we consider `unify-free/bound`, the one non-trivial helper function used by `unify-var/var`.

```
(define unify-free/bound
  (lambda (t cu subst)
    (let loop ([cu cu])
      (let ([u (commitment->term cu)])
        (cond
          [(eq? u t) subst]
          [(var? u)
           (cond
             [(assq u subst) => loop]
             [else (extend-subst t u subst)])]
          [else (extend-subst t u subst)])))))
```

Because `cu` is a commitment, we know that `u` is always internal. Thus, as long as `u` is a bound variable, which has been shown to be unequal to the free variable `t`, we loop on `u`. If the variable `u` is free, we extend the substitution by a new commitment binding `t` to `u`. We can safely extend the substitution since we know that `t` and `u` are different and that `u` is internal. Finally, if `u` is not a variable, we know that `t` and `u` differ, and we know that `u` is internal, so once again we can extend the substitution by a new commitment binding `t` to `u`.

Finally, we come to `unify-nonvar/bound`, which is called from `unify-var/nonvar`.

```
(define unify-nonvar/bound
  (lambda (t cu subst)
    (let loop ([cu cu])
      (let ([u (commitment->term cu)])
        (cond
          [(var? u)
           (cond
             [(assq u subst) => loop]
             [(pair? t) (unify-free/list u t subst)]
             [else (extend-subst u t subst)])]
          [else (unify-nonvar/nonvar u t subst)])))))
```

As long as `cu`'s associated term is a bound variable, we loop on `u`. Otherwise, we know that `t` is not a variable, so if it is a pair, we again purge it of occurrences of the anonymous variable before extending the substitution. This is possible because we know that `u` is now free. If `t` is not a pair, we can extend the substitution, since it must be internal. Finally, if `u` is not a variable, we pass the responsibility onto `unify-nonvar/nonvar`.

Now that we have seen the implementation of the naive unifier, we can explore Kanren's `==` function.

```
(def-syntax (== t u)
  (lambda@ (sk fk subst)
    (let ([subst (unify t u subst)]) (if subst (@ sk fk subst) (fk)))))
```

This may look complicated, but as with other antecedents that we have seen, if it succeeds, it does so using the same success continuation `sk` and the same failure continuation `fk`, but with a potentially extended substitution. If the two terms fail to unify with the current substitution, it fails by invoking the failure continuation.

Let's use `==` to test our naive unifier on some very simple examples. We need the `answer` macro from *Logic-Oriented Programming*.

We start by unifying variables, numbers, and symbols.

```
> (answer (x) (== x 3))
x :: 3
> (answer (y) (== 4 y))
y :: 4
> (answer (x y) (== x y))
x :: y.0
y :: y.0
> (answer () (== 'x 'x))
> (answer (x) (== x x))
x :: x.0
> (answer () (== 4 'y))
#f
> (answer () (== 'x 3))
#f
> (answer () (== 3 4))
#f
```

Only the fifth example is interesting. It contains no commitments, and thus returns the empty substitution, since the two terms are the same variable. The sixth and seventh don't unify because a symbol (not a variable) can never be equal to a number.

We have barely scratched the surface of just how complicated the test programs can become. Here are some more.[5]

[5] Since the list lengths are the same and *both* use `quasiquote`, we could rewrite these and the remaining examples by including several `==`'s in an `all!!` like this:

```
> (answer (x) (all!! (== x 3) (== 4 x)))
#f
```

Of course, neither of these conditions might hold. For example, we might have:

```
> (answer (x) (== `(,x 4) ((lambda (a b) `(3 ,a)) x 'ignore)))
#f
```

Or, we might have something like this:

```
> (answer (x) (== `(,x ,x) '(3 4)))
#f
```

```
> (answer (x) (== `(,x ,4) `(3 ,x)))
#f
```

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

```
> (answer (x y) (== `(,x ,y) '(3 4)))
x :: 3
y :: 4
> (answer (x y) (== `(,x 4) `(3 ,y)))
x :: 3
y :: 4
```

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

```
> (answer (w x y z) (== `(,x 4 3 ,w) `(3 ,y ,x ,z)))
w :: z.0
x :: 3
y :: 4
z :: z.0
```

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

```
> (answer (x y) (== `(,x 4) `(,y ,y)))
x :: 4
y :: 4
> (answer (x y) (== `(,x 4 3) `(,y ,y ,x)))
#f
```

In the first example, `x` co-refers to `y` and agrees to be associated with the same term. Then `y` binds to 4. In the second example, once `y` has been bound to 4, since `y` and `x` co-refer, `x` binds to 4, but `x` tries to bind to 3, which violates its commitment.

Here are four more unification examples.

```
> (answer (x) (== `(a b . ,x) `(a b c d e))) x :: (c d e)
```

where the *lengths*, such as they are, are not the same. Of course, if we are careful, we could rewrite this variant like this.

```
> (answer (x) (all!! (== 'a 'a) (== 'b 'b) (== x '(c d e))))
> (answer (u w x y z) (== `(,w (,x (,y ,z) 8)) `(,w (,u (abc ,u) ,z))))
u :: 8
w :: w.0
x :: 8
y :: abc
z :: 8
> (answer (x y) (== `(p (f a) (g ,x)) `(p ,x ,y)))
x :: (f a)
y :: (g (f a))
> (answer (x y) (== `(p (g ,x) (f a)) `(p ,y ,x)))
x :: (f a)
y :: (g (f a))
> (answer (x y z) (== `(p a ,x (h (g ,z))) `(p ,z (h ,y) (h ,y))))
x :: (h (g a))
y :: (g a)
z :: a
```

The first example binds `w` to itself, which makes no new commitments. Then `x` co-refers with `u`, `y` binds to the symbol `abc`, `z` co-refers with `u`, and `z` binds to 8. In the second example, the `p` symbols match, leading to no new commitments. Then, `x` binds to `(f a)`. Finally, `y` binds to `(g (f a))`, since the `x` in `` `(g ,x)`` gets replaced by what `x` is bound to, `(f a)`. The third example produces the same values for the variables, but in a different way. Again, `p` symbols lead to no new commitments. Then `y` binds to `` `(g ,x)``. Finally, `x` binds to `(f a)`. The fourth example first binds `z` to the symbol `a`. Then, `x` binds to `` `(h ,y)``. Next the `h` symbols contribute nothing. Finally, `y` binds to `` `(g ,z)``. But, since `z` binds to `a`, we know that `y` must become `(g a)`, which is why `x`'s value is `(h (g a))`.

We have one more example, and this one is quite interesting.

```
(def-syntax (run-raw (id ...) ant fk subst SE FE)
  (exists (id ...)
    (@ ant
      (lambda@ (fk subst) SE)
      (lambda () FE)
      empty-subst)))

> (for-each (lambda (cmt)
            (write (commitment->var cmt))
            (display " :: ")
            (write (commitment->term cmt))
            (newline))
  (run-raw (x y) (== `(p ,x ,x) `(p ,y (f ,y))) fk subst (concretize subst) #f))
y.0 :: (f y.0)
x.0 :: y.0
```

This example unifies. The `p` symbols unify. Then, `x` co-refers with `y`. When we try to unify `x` with `‘(f ,y)`, we discover that since `y` and `x` co-refer, we are really unifying `x` with `(f x)`. It should be clear that this is a violation of the so called *occurs check*, and this circularity should be cause for concern. But, this unifier manages this circularity, which can be seen by the displayed output.[6] It would loop, however, if we tested this using `run` or `answer` instead or `run-raw`, since the `subst-in` would never terminate.

We now improve the efficiency of this unifier. By redefining `unify-i/i` below, we start a cascade of decisions that guarantees calls to `unify-free/list` occur only when absolutely necessary: when we are certain that the list does not come from a substitution. For if the list has come from a substitution we know that it does not contain any occurrences of the anonymous variable. We want to avoid the invocation of `unify-free/list`, since it searches the list for an occurrence of the anonymous variable. If `unify-free/list` finds the anonymous variable, then the function must copy the entire list, replacing each occurrence of the anonymous variable with a logic variable. These two operations are expensive!

The function `unify-i/i` uses the knowledge that both terms passed in are internal. All but `unify-i/i`'s second clause can take advantantage of this property. The `unify-i/i` function calls `unify-var/nonvar&i` and `unify-nonvar&i/nonvar&i`, which are “smarter twins” of the `unify-var/nonvar` and `unify-nonvar/nonvar` functions. Both of these `nonvar&i` functions are more concise and efficient than their “slower” twins, since they know that one or more of their arguments are internal.

```
(define unify-i/i
  (lambda (t u subst)
    (cond
      [(eq? t u) subst]
      [(and (var? t) (var? u)) (unify-var/var t u subst)]
      [(var? t) (unify-var/nonvar&i t u subst)]
      [(var? u) (unify-var/nonvar&i u t subst)]
      [else (unify-nonvar&i/nonvar&i t u subst)])))
```

The function `unify-var/nonvar&i` calls `unify-nonvar&i/bound`, and avoids the `pair?` test and possible call to `unify-free/list` performed by its slower twin: `unify-var/nonvar`.

```
(define unify-var/nonvar&i
  (lambda (t u subst)
    (let ([ct (assq t subst)])
      (if ct
        (unify-nonvar&i/bound u ct subst)
        (extend-subst t u subst)))))
```

[6] The concretized variable name of the logic variable for both commitments are displayed.

The function `unify-nonvar&i/nonvar&i` calls `unify-i/i` on the `car` and `cdr`, thus avoiding any risk of invoking `unify-free/list` further down the list.

```
(define unify-nonvar&i/nonvar&i
  (lambda (t u subst)
    (if (and (pair? t) (pair? u))
      (let*-and #f ([subst (unify-i/i (car t) (car u) subst)])
        (unify-i/i (cdr t) (cdr u) subst))
      (if (equal? t u) subst #f))))
```

Finally, the call to `unify-nonvar&i/bound` carries with it that the first argument is internal. This also avoids both the test for pair and the subsequent call to `unify-free/list` that exists in its slower twin: `unify-nonvar/bound`.

```
(define unify-nonvar&i/bound
  (lambda (t cu subst)
    (let loop ([cu cu])
      (let ([u (commitment->term cu)])
        (cond
          [(var? u)
           (cond
             [(assq u subst) => loop]
             [else (extend-subst u t subst)])]
          [else (unify-nonvar&i/nonvar&i u t subst)])))))
```

Since we do not know in advance that the terms being unified are free of occurrences of the anonymous variable, we continue to use `unify` as the driver.

There were a couple of functions used that were not defined: `commitment->var` and `commitment->term`. These functions are `car` and `cdr`, since commitments are built with `cons`. The definition of `commitment->var` follows from the way that we use `assq`. The latter is arbitrary, chosen for the purpose of saving one cons-cell for each commitment.

This concludes the description of mini-Kanren's unifier. The code in Kanren is written with fewer functions and with slighly different names, but the algorithm is the same. It can take a long time to internalize all the facets of this unifier. Most importantly, it is necessary to be able to figure out what `==` applied to two terms yields given an existing substitution. Once you are comfortable with the behavior of `==`, delve into the code of this unifier.