# *Relation-Oriented Programming (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*

This is a presentation of *Relation-Oriented Programming*. In order to appreciate *Logic-Oriented Programming*, we must first understand relations. We use the term *relation* in the following way. A function is a mapping from a value to another value, which can be modelled by a set of pairs where no two pairs have the same first value. Consider the `square` function defined over integral values. We can represent this function as the following infinite set of pairs,

{... `(-3 9) (-2 4) (-1 1) (0 0) (1 1) (2 4) (3 9)` ...}

The left-hand-side of a pair denotes the input to the `square` function, while the right-hand-side of the pair represents the value returned by the function. A relation can be modelled by a set of pairs without this restriction. For example, we can represent the `square-root` relation over the domain of positive integers as follows,

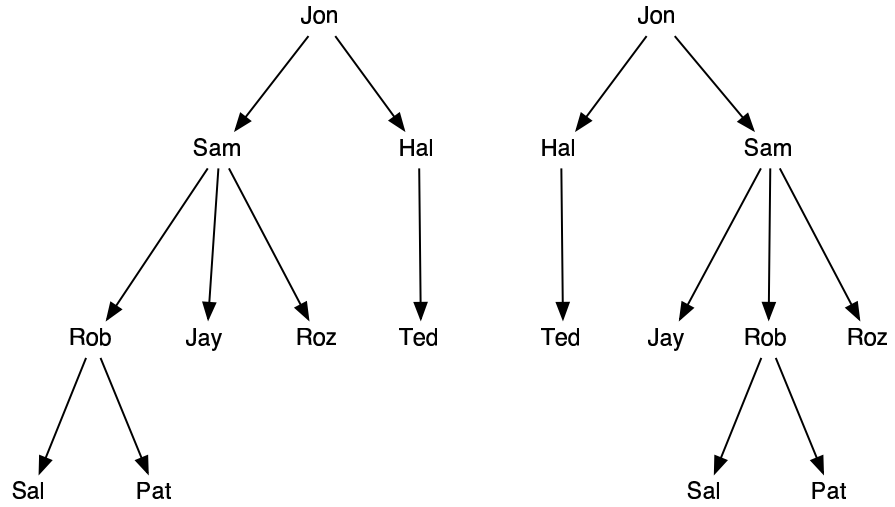
{`(1 1) (1 -1) (2` $\sqrt{2}$`) (2 -`$\sqrt{2}$`)) (3` $\sqrt{3}$`) (3 -`$\sqrt{3}$`) (4 2) (4 -2)` ...}

So `square-root` is a relation but not a function, since each input value (*e.g.*, 1) is associated with more than one output value (1 and -1, in this case there are two output values). We introduce more of the Kanren logic system, which we first encountered while examining *Outcome-Oriented Programming (OOP)*.

We address the notion of relation in two separate ways. First, we show how using `lambda` and outcomes is powerful enough to express Relation-Oriented Programming in its entirety. Then, we show that by a careful crafting of some macros, we can make Relation-Oriented Programming seem more natural.

We will use the following family-oriented relations throughout: `father`, `mother`, `child`, `grandparent`, `ancestor`, `common-ancestor`, `younger-common-ancestor`, and `youngest-common-ancestor`. If we are going to have a program that responds to these relations, we need some real data: Jon is the father of Sam and Hal; Sam is the father of Rob, Jay, and Roz; Hal is the father of Ted; and Rob is the father of Sal and Pat. Figure 1 shows two different representations of this paternal family tree.

Fig. 1. Two different representations of our paternal family tree.




We can represent this paternal relationship with the following `father` procedure.

```
(define father
  (lambda (dad child)
    (any
      (all!! (== dad 'jon) (== child 'sam))
      (all!! (== dad 'jon) (== child 'hal))
      (all!! (== dad 'sam) (== child 'rob))
      (all!! (== dad 'sam) (== child 'jay))
      (all!! (== dad 'sam) (== child 'roz))
      (all!! (== dad 'hal) (== child 'ted))
      (all!! (== dad 'rob) (== child 'sal))
      (all!! (== dad 'rob) (== child 'pat)))))
```

There are eight `(all!! ...)` expressions in the `father` procedure, which we can reorder in any of 8! different ways. Regardless of the order we choose for the `(all!! ...)` expressions, our `father` procedure will impose that same ordering upon the children occupying a given level of the family tree. In the `father` procedure given above, Sam comes "before" Hal, even though both Sam and Hal are children of Jon. This ordering forces us to choose the left-hand tree in Figure 1 as our preferred graphical representation of the family tree, since a breadth-first left-to-right traversal of this tree will yield Jon's descendants in the order specified in the `father` procedure.

If we were to switch the order of the `(all!! ...)` expressions in the `father` procedure, the order of the answers produced by our Relation-Oriented programs might change as well, along with the amount of computational resources consumed by those programs.

Each time `trace-vars` is invoked in the code below, it displays the content of a portion (it may be all of it) of a substitution. After each `trace-vars` completes, it passes the current substitution to `fail`. That causes the computation to back up through the `trace-vars` and then through the `father`. Backing into it will cause the logic variables `f` and `c` to get rebound and then that substitution is sent to `trace-vars`, etc. Since the final outcome is failure, we know that the result will be the `#f`, indicating failure.

```
> (run (f c)
    (all
      (father f c)
      (trace-vars "::" (f c))
      (fail))
    fk subst #t #f)
f :: jon
c :: sam

f :: jon
c :: hal

f :: sam
c :: rob

f :: sam
c :: jay

f :: sam
c :: roz

f :: hal
c :: ted

f :: rob
c :: sal

f :: rob
c :: pat

#f
```

If the number of answers is relatively small, then a simple macro like the one below meets our needs. As is evident, the only things that we abstracted over were the list of variables `(f c)` and the antecedent expression, `(father f c)`. Why must this be a macro? Because the `id ...` in the `run` expression below introduce lexical scope.

```
(def-syntax (answers (id ...) ant)
  (run (id ...) (all ant (trace-vars "::" (id ...))) fk subst (fk) #f))

> (answers (f c) (father f c))
```

We should not always use the `answers` macro, of course, since it is possible to give an antecedent that produces an unbounded number of answers. In *Logic-Oriented-Programming*, we see one way to control that.

We can take projections of this relation with examples like this:

```
> (answers (c) (father 'sam c))
c :: rob

c :: jay

c :: roz

#f

> (answers (f) (father f 'rob))
f :: sam

#f
```

The first example yields three substitutions, each of which gives a child of Sam. The second example yields a single substitution, since Rob has only one father.

An alternative is to build a list of answers like this.

```
> (run (c) (father 'sam c) fk subst (cons c (fk)) '())
(rob roz jay)
```

The `run-list` macro makes creating lists of answers more convenient.

```
(def-syntax (run-list (id ...) ant succeed-item-expr)
  (run (id ...) ant fk subst (cons succeed-item-expr (fk)) '()))

> (run-list (c) (father 'sam c) c)
```

We can define `child` by simply claiming that in our world,

```
(define child
  (lambda (child dad)
    (father dad child)))
```

```
> (run-list (c f) (child c f) (list c f))
((sam jon)
 (hal jon)
 (rob sam)
 (jay sam)
 (roz sam)
 (ted hal)
 (sal rob)
 (pat rob))
```

Thus, to show that $c$ is the child of $f$, we need only show that $f$ is the father of $c$. Of course, this assumes that in our population, no woman has a child. If that were not the case, then we would likely define `child` this way.

```
(define child
  (lambda (child parent)
    (any
      (father parent child)
      (mother parent child))))
```

`Exercise 1:` Create a new relation `mother` and verify that this revised definition of `child` makes sense. What happens when the same child is in both `father` and `mother`? ◇

We next define the grandparent relation, keeping in mind that a parent is the child of a grandparent and a grandchild is the child of the parent.

```
(define grandparent
  (lambda (gr-parent gr-child)
    (exists (parent)
      (child parent gr-parent)
      (child gr-child parent))))

> (run-list (gp gc) (grandparent gp gc) (list gp gc))
((jon rob)
 (jon jay)
 (jon roz)
 (jon ted)
 (sam sal)
 (sam pat))
```

The relation `ancestor` determines if a person is a parent, grandparent, greatgrandparent, greatgreatgrandparent, etc. of someone.

```
(define ancestor
  (lambda (anc desc)
    (any
      (child desc anc)
      (exists (parent)
        (child desc parent)
        (ancestor anc parent)))))
```

```
> (run (a d) (ancestor a d) (list a d))
((jon sam)
 (jon hal)
 (sam rob)
 (sam jay)
 (sam roz)
 (hal ted)
 (rob sal)
 (rob pat)
 (jon rob)
 (jon jay)
 (jon roz)
 (jon ted)
 (sam sal)
 (jon sal)
 (sam pat)
 (jon pat))
```

If we look at the output of this example, we can see that a person might have multiple ancestors. For example, Pat has three ancestors. We might also want a relation that determines if two people share a common ancestor. For example, Jon is the common ancestor of everyone but himself.

```
(define common-ancestor
  (lambda (anc desc1 desc2)
    (all
      (ancestor anc desc1)
      (ancestor anc desc2)
      (project (desc1 desc2)
        (predicate (not (eqv? desc1 desc2)))))))
```

```
> (answers (a) (common-ancestor a 'sal 'jay))
a :: sam

a :: jon

#f
```

`Exercise 2:` How many answers would there be if the test had been `(answers (a d1 d2) (common-ancestor a d1 d2))`? If we removed the last antecedent in the definition of `common-ancestor`, how many answers would we get? ◇

We might want to know of any two common ancestors, which one is younger. For example, Jon is clearly the older of Sam and Jon.

```
(define younger-common-ancestor
  (lambda (young-anc old-anc desc1 desc2)
    (all
      (common-ancestor young-anc desc1 desc2)
      (common-ancestor old-anc desc1 desc2)
      (ancestor old-anc young-anc))))
```

```
> (answers (y o d2) (younger-common-ancestor y o 'jay d2))
y :: sam
o :: jon
d2 :: rob

y :: sam
o :: jon
d2 :: roz

y :: sam
o :: jon
d2 :: sal

y :: sam
o :: jon
d2 :: pat

#f
```

We might want to know given two people, who their youngest common ancestor is. For example, let us determine the youngest common ancestor of Jay and Pat.

```
(define youngest-common-ancestor
  (lambda (young-anc desc1 desc2)
    (all
      (common-ancestor young-anc desc1 desc2)
      (fails
        (exists (y)
          (younger-common-ancestor y young-anc desc1 desc2))))))
```

```
> (answers (y) (youngest-common-ancestor y 'jay 'pat))
y :: sam

#f
```

We discover that Sam, who is Jay's father and Pat's grandfather, is the youngest common ancestor of Jay and Pat.

What we have just demonstrated is that conventional logic programming can be managed at the level of writing Scheme functions and using outcomes. Next, we show how we can change the perspective just a little and yield equivalent programs. To do this, we add the following rule to the grammar introduced in Outcome-Oriented Programming.

```
R :: (relation (Id*) (to-show Term*) A)
   | (fact (Id*) Term*)
   | (extend-relation Arity R*)
   | (intersect-relation Arity R*)

Arity :: (Id*)
```

The length of the `Arity` list must match the number of terms in the relation or relations being extended or intersected. For example, the `father` relation expresses the relationship between two terms, the father and the child. When extending the `father` relation, the `Arity` list must therefore contain exactly two identifiers. Only the length of the `Arity` list is significant, and the actual identifiers appearing in the list may be chosen arbitrarily, but they must be different. The following expressions are therefore equivalent, since each expression's `Arity` list is of length two:

```
(extend-relation (a1 a2) father)
(extend-relation (dad child) father)
```

The `relation` macro is similar to `lambda` in that it introduces lexical scope. Here is a relation, `ancestor-of-both`, that uses the `relation` macro.

```
(define ancestor-of-both
  (relation (a d1 d2)
    (to-show a `(,d1 ,d2))
    (all
      (ancestor a d1)
      (ancestor a d2)
      (project (d1 d2)
        (predicate (not (eq? d1 d2)))))))

> (answers (a) (ancestor-of-both a '(pat jay)))
a :: sam

a :: jon

#f
```

```
> (run-list (d1 d2) (ancestor-of-both 'sam `(,d1 ,d2)) `(,d1 ,d2))
((rob jay)
 (rob roz)
 (rob sal)
 (rob pat)
 (jay rob)
 (jay roz)
 (jay sal)
 (jay pat)
 (roz rob)
 (roz jay)
 (roz sal)
 (roz pat)
 (sal rob)
 (sal jay)
 (sal roz)
 (sal pat)
 (pat rob)
 (pat jay)
 (pat roz)
 (pat sal))
```

Below we show how `ancestor-of-both` would have been written if we did not have `relation`. First, the number of arguments after the `to-show` keyword is two, so we have two `lambda` formals, with arbitrary names: `g1` and `g2`. Second, we have a list of variables wrapped in an `exists` expression. This corresponds to the number of distinct variables in the `to-show`. We can see that there are three variables, `a`, `d1`, and `d2`. Then, each of the introduced variables is placed in an `==` expression, one for each of the arguments of the `to-show`. Finally, the inner `(all ...)` expression is the same as the one antecedent that follows the `to-show` expression.

```
(define ancestor-of-both
  (lambda (g1 g2)
    (exists (a d1 d2)
      (ef/only (all!!
                 (== g1 a)
                 (== g2 `(,d1 ,d2)))
        (all
          (ancestor a d1)
          (ancestor a d2)
          (project (d1 d2)
            (predicate (not (eq? d1 d2)))))
        (fail)))))
```

There is a special case of `relation` where there is no antecedent. For example, we might have the *fact* that Jon is the father of Sam.

```
(define father-sam
  (fact () 'jon 'sam))
```

or

```
(define father-sam
  (relation ()
    (to-show 'jon 'sam)
    (succeed)))
```

The list that follows the symbol `fact` is the list of all the variables used in the fact. Of course, here there are none. Then we could define `father` using a bunch of facts and a new feature: `extend-relation`.

```
(define father
  (extend-relation (f c)
    (fact () 'jon 'sam)
    (fact () 'jon 'hal)
    (fact () 'sam 'rob)
    (fact () 'sam 'jay)
    (fact () 'sam 'roz)
    (fact () 'hal 'ted)
    (fact () 'rob 'sal)
    (fact () 'rob 'pat)))
```

The first list `(f c)` is required and it tells `extend-relation` how many arguments each relation takes. `extend-relation` acts like a call-by-value function, but since the list of variables `(f c)` is not evaluated, it is implemented using a macro. The reason `extend-relation` evaluates all but its first argument is to allow for incremental-relation building.

```
(define father
  (extend-relation (f c)
    father
    (fact () 'jon 'hal)
    (fact () 'sam 'rob)))
```

which would otherwise loop indefinitely when it is called.

`Exercise 3:` Implement `ancestors-of-all`, a variation of `ancestor-of-both`, that works with an arbitrarily long list. That would mean that we might want to use `‘(,d . ,d*)`, where `d` would be the first descendant and `d*` would be all the other descendants. Solve the problem with and without `relation`. ◇

We could have built our `father` relation in another way.

```
(define father
  (let loop ([facts (list
                     (fact () 'jon 'sam)
                     (fact () 'jon 'hal)
                     (fact () 'sam 'rob)
                     (fact () 'sam 'jay)
                     (fact () 'sam 'roz)
                     (fact () 'hal 'ted)
                     (fact () 'rob 'sal)
                     (fact () 'rob 'pat))])
    (cond
      [(null? facts) (relation () (to-show _ _) (fail))]
      [else (extend-relation (f c) (car facts) (loop (cdr facts)))])))
```

This follows, since relations, including facts, are first-class values.

We can redefine `parent` using `extend-relation`, and then we can use it to redefine `child`.

```
(define parent
  (extend-relation (p c)
    father
    mother))

(define child
  (relation (child ther[1])
    (to-show child ther)
    (parent ther child)))
```

Revising `grandparent` to use `relation` yields

```
(define grandparent
  (relation (gr-parent gr-child)
    (to-show gr-parent gr-child)
    (exists (parent)
      (child parent gr-parent)
      (child gr-child parent))))
```

[1] We use the variable `ther` to represent a parent (either a f`ather` or a mo`ther`).

Revising `ancestor` gives us

```
(define ancestor
  (relation (anc desc)
    (to-show anc desc)
    (any
      (child desc anc)
      (exists (parent)
        (child desc parent)
        (ancestor anc parent)))))
```

`Exercise 4:` Implement `youngest-common-ancestor` using `relation`. ◇

We next introduce `intersect-relation`, which acts like `extend-relation`, but instead of using `any`, it uses `all` to build up the relation. Consider the following situation.

```
(define scouts
  (extend-relation (s)
    (fact () 'rob)
    (fact () 'sue)
    (fact () 'sal)))

(define athletes
  (extend-relation (a)
    (fact () 'roz)
    (fact () 'sue)
    (fact () 'sal)))

(define busy-children
  (intersect-relation (c)
    scouts
    athletes))

(define social-children
  (extend-relation (c)
    scouts
    athletes))

> (answers (c) (busy-children c))
c :: sue

c :: sal

#f
```

```
> (answers (c) (social-children c))
c :: rob

c :: sue

c :: sal

c :: roz

c :: sue

c :: sal

#f
```

When we evaluate `(answers (c) (social-children c))`, `sue` and `sal` appear twice. A different implementation of `extend-relation` could filter out these duplicate answers. (See `kanren.ss`.)

The advantages of `(relation Id* (to-show Term*) A)` can hardly be observed using these simple programs. In *Logic-Oriented-Programming*, we shall see how `relation` improves the readability of our definitions.