
Daniel P. Friedman  
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 Outcome-Oriented Programming. In order to appreciate Relation-Oriented Programming, we should first understand outcomes. We use the term outcome in the following way. Outcomes are either successful or they are unsuccessful. We say that we have succeeded when an outcome is successful and we say that we have failed when an outcome is unsuccessful. This presentation introduces a subset of the Kanren language. We observe that outcomes loosely correspond to Boolean values. So, we can inaccurately say that if something succeeds, it is true or that if something fails, it is false. This characterization can lead to overestimating the power of Outcome-Oriented Programming and later to overestimating the power of Relation-Oriented Programming. One should avoid conflating Boolean values with outcomes.

Here are the types we use in this system:

\[
\begin{align*}
Fk &= () \rightarrow Ans \\
Ans &= Nil + [\text{Substitution}, Fk] \\
Sk &= Fk \rightarrow \text{Substitution} \rightarrow Ans \\
\text{Antecedent} &= Sk \rightarrow Sk
\end{align*}
\]

Substitutions are like environments in that variables have associations. They are different from environments in several ways that we discuss below. Ans is a stream (possibly the empty stream Nil) of Substitutions, since Fk, a failure continuation, is a function (\rightarrow indicates the type of a function.) of zero arguments. The notation \([x, y]\) indicates a pair of \(x\) and \(y\). An antecedent is an Sk (i.e., a success continuation) transformer. Thus, an antecedent is a function like this:

\[
\begin{align*}
(\lambda \text{sk} \\
(\lambda \text{fk} \\
(\lambda \text{subst} \\
\ldots)))
\end{align*}
\]

In the grammar below, \(x^*\) denotes a sequence of zero or more occurrences of \(x\), and \(\{x\}\) denotes 0 or 1 occurrence of \(x\). We describe antecedents A and answers Ans.Title is an any printable value, Term is any Scheme value, including a logic variable. If Term is a (proper or improper) list, it may also include logic variables.
Id is a lexical variable whose value becomes, is, or was a logic variable. Each of the A-expressions in the grammar, when evaluated, returns an antecedent.

\[
A :: (\text{project (Id*) } A) \\
| (\text{project/no-check (Id*) } A) \\
| (\text{exists (Id*) } A) \\
| (\text{all!! } A*) \\
| (\text{trace-vars Title (Id*)}) \\
| (== \text{Term Term}) \\
| (\text{predicate Scheme-Expression}) \\
| \text{succeed} \\
| (\text{fails } A) \\
| (\text{if-only } A \ A \ {\{A\}}) \\
| \text{fail} \\
| (\text{any } A*) \\
| (\text{all } A*) \\
| (\text{all! } A*)
\]

\[
\text{Ans :: (query } A) \\
\]

An identifier, if it has an association, resides in one of two tables. The first table, the lexical environment holds the values of standard lexically-scoped variables. The second one is the logically-scoped variables, introduced with \textit{exists}. There is an operation, \textit{project} (the “o” is pronounced “oh”) that treats a logically-scoped variable as if it were lexically scoped.

\[
(\text{project (a b c) } A)
\]

is the same as

\[
(\text{lambda (sk)} \\
\quad (\text{lambda (fk)} \\
\quad \quad (\text{lambda (subst)} \\
\quad \quad \quad (\text{let (}}[a \ \text{(nonvar! (subst-in a subst))}] \\
\quad \quad \quad \quad [b \ \text{(nonvar! (subst-in b subst))}] \\
\quad \quad \quad \quad \quad [c \ \text{(nonvar! (subst-in c subst))}] \\
\quad \quad \quad \quad \quad ((A \ sk) \ fk) \ \text{subst))))))
\]

What this says is that if a logic variable is in the substitution (\textit{subst-in} is sort of like an environment lookup, except it can also be passed something that is not a logic variable.) and if it is not instantiated to a logical variable (error tested with \textit{nonvar!}), then we bind its value to the lexically-scoped variable of the same name. (\textit{project/no-check} is the same without the \textit{nonvar!} error tests.) What this means is that henceforth the work of determining the term associated with a logic variable in a substitution has been short-circuited. But, more importantly, in the antecedent expression A, the lookup of these variables is now constant time.

Let’s look at the example below. The \textit{query} form creates a potentially infinite list of substitutions. In our case, the potentially infinite list has \textit{one} value, which is the
empty substitution. Think of all!! as an antecedent-based and that quits when it finds an unsuccessful antecedent. (trace-vars ...) displays a substitution's term associated with each variable, and it always succeeds.) The first time the substitution physically contains zero variables. How is that possible? The values of the variables, x, v, i, and w have associated terms, but not in the substitution. When a variable is uninstantiated in the substitution, it returns a variable, sometimes itself. This is very different from an environment, where such a variable lookup would lead to an error. Next, we add (using ==) three things to the substitution. Now, x is instantiated to the term 10, v is instantiated to the term 5, and i is shared with the term w. In substitutions, it is okay to associate a variable with a variable, another way that substitutions differ from environments.

(define test
  (exists (x v i w)
    (all!!
      (trace-vars "1" (x v i w))
      (== x 10)
      (== v 5)
      (== i w)
      (successful-branch x v i w))))

(define successful-branch
  (lambda (x v i w)
    (project (x v)
      (begin
        (writeln (+ x v))
        (all!!
          (trace-vars "2" (x v i w))
          (== i 1)
          ;-----------
          (project (w)
            (all!!
              (predicate (writeln (+ x v w)))
              (trace-vars "3" (x v i w))
              (project (i)
                (begin
                  (writeln (+ x v w i))
                  (succeed))))))))))
> (define answer1 (query test))
1  (x v i w) (x.0 v.0 i.0 w.0)
15
2  (x v i w) (10 5 w.0 w.0)
16
3  (x v i w) (10 5 1 1)
17
> (cdr answer1)
#<initial-fk>

In successful-branch, the project looks up the logical variables \(x\) and \(v\) in the substitution and associates the same-named lexical variables with their associated terms. The next interesting antecedent associates \(i\) with 1. Of course, we have just associated \(i\) with \(w\), so apparently it also ends up saying that, “Since \(i\) is equal to \(w\) and now \(i\) is 1, then naturally \(w\) is 1.” So, even though we upgraded \(i\) to a constant, the sharing relationship that had been created with \(w\) has led to its associated term also being upgraded. The predicate form creates an antecedent from an arbitrary Scheme expression. If the Scheme expression evaluates to false, then the antecedent fails. Otherwise, it succeeds. That way, the arbitrary expression can be placed in an \texttt{all}!, just as we can place the same Scheme expression in an \texttt{and} expression.

If we make a change in the code above by replacing the comment \;\;\;\;\;\; by \((\text{fails} \;== \;w \;2)\), will we change the outcome of our test? No! This variant has the same behavior as the previous version, but we’ve added one more line to it: \((\text{fails} \;== \;w \;2)\). After having decided that \(i\) and \(w\) share a value, we have decided that instead of \(w\) being 1, we would make it 2. Now, there are two interpretations of this attempt to add new knowledge to the substitution. We could change both \(i\) and \(w\) to have 2 as its association, or we could claim that once it has been upgraded to a constant, it is too late to make this change. Outcome-Oriented Programming chooses the latter. Therefore, \((== \;w \;2)\) does not succeed. But, to make something that fails succeed, we wrap a \((\text{fails} \;\ldots)\) around the antecedent, just as we wrap \((\text{not} \;\ldots)\) around something that is false to make it true. So our test does exactly the same thing that it did before.

In the next example below, we see that we are using an antecedent expression \((\text{if-only} \;\ldots)\) that appears to act like an \texttt{if} expression, but instead of deciding which outcome based on the Boolean values true/false, it decides based on the outcomes succeed/fail. In our case, \texttt{fail} causes it to take the fail branch. (By the way, don’t confuse \texttt{fail} with \texttt{fails}. The former \texttt{is} an antecedent like \texttt{succeed}, but the latter wraps an antecedent, yielding a new antecedent.)
(define test/if-only
  (exists (x v i w)
    (all!!
      (trace-vars "1" (x v i w))
      (== x 10)
      (== v 5)
      (== i w)
      (if-only fail
        (successful-branch x v i w)
        (all!!
          (trace-vars "4" (x v i w))
          (== w 1000)
          (project (x v i w)
            (predicate (writeln (+ i w x v))))))))))

> (define answer1 (query test/if-only))
  1 (x v i w) (x.0 v.0 i.0 w.0)
  4 (x v i w) (10 5 w.0 w.0)
  2015
> (cdr answer1)
  #<initial-fk>

Now, let’s replace the first argument in the if-only expression by (== i 1).

> (define answer1 (query test/if-only))
  1 (x v i w) (x.0 v.0 i.0 w.0)
  4 (x v i w) (10 5 w.0 w.0)
  2015
> (cdr answer1)
  #<initial-fk>

Here we see that part of the semantics of if-only is to remember everything that happened in the test antecedent, if it succeeded. The test is (== i 1), so now, when we enter the successful branch, it now knows that i is instantiated to 1 and since w shares its associated term with i, its associated term is 1, too.

Next, we replace the first argument to if-only by (fails (== i 1)).

> (define answer1 (query test/if-only))
  1 (x v i w) (x.0 v.0 i.0 w.0)
  4 (x v i w) (10 5 w.0 w.0)
  2015
> (cdr answer1)

Not only does the test fail, but it forgets that it has instantiated i to 1! So, even though == appears to add something to the substitution, it only does so if the test succeeds.

Now that we have a better grasp of if-only, we can use it to explain any of two arguments, although like all!!, it works with any number of arguments.
It succeeds if any of its arguments succeeds. \((\text{any } A_1 A_2)\) is the same as if we had written \((\text{if-only } A_1 \text{ succeed } A_2)\). So, very little needs to be said about it. Again, it is important that we remember that running \(A_1\) succeeds or fails. If it succeeds, then \(A_1\)'s resultant substitution survives. Here, in order for \(\text{all!!}\) to succeed, its two antecedents, \((\text{trace-vars } "1" \ (x \ w))\) and \((== x \ w)\) must succeed. The \((\text{trace-vars }\ldots)\) always succeeds. And having \(x\) share with \(w\) always succeeds if either \(x\) or \(w\) is un-instantiated. Since both are un-instantiated, and we only need one, clearly the first disjunct succeeds, which is why we don’t see a 2 below.

Consider this simple example.

\[
> \ (\text{define answer1} \\
> \ (\text{query} \\
> \ \ \ (\text{exists} \ (x \ w) \\
> \ \ \ \ (\text{any} \\
> \ \ \ \ \ (\text{all!!} \\
> \ \ \ \ \ \ (\text{trace-vars } "1" \ (x \ w)) \\
> \ \ \ \ \ \ (== x \ w)) \\
> \ \ \ \ \ (\text{all!!} \\
> \ \ \ \ \ \ (\text{trace-vars } "2" \ (x \ w)) \\
> \ \ \ \ \ \ (== x \ 10))))))
\]

\[
1 \ (x \ w) \ (x.0 \ w.0) \\
> \ (\text{cdr answer1}) \\
#<fk>
\]

Now, consider this slight variation.

\[
> \ (\text{define answer1} \\
> \ (\text{query} \\
> \ \ \ (\text{exists} \ (x \ w) \\
> \ \ \ \ (\text{any} \\
> \ \ \ \ \ (\text{all!!} \\
> \ \ \ \ \ \ (\text{trace-vars } "1" \ (x \ w)) \\
> \ \ \ \ \ \ (== w \ 1000) \\
> \ \ \ \ \ \ (\text{trace-vars } "2" \ (x \ w)) \\
> \ \ \ \ \ \ (== x \ 100) \\
> \ \ \ \ \ \ (\text{trace-vars } "3" \ (x \ w)) \\
> \ \ \ \ \ \ (== x \ w)) \\
> \ \ \ \ \ (\text{all!!} \\
> \ \ \ \ \ \ (\text{trace-vars } "4" \ (x \ w)) \\
> \ \ \ \ \ \ (== x \ 10) \\
> \ \ \ \ \ (\text{trace-vars } "5" \ (x \ w)))))))
\]
The first five antecedents of the first disjunct (i.e., the first all!! expression) succeed, but in the sixth antecedent, we try to have $x$ and $w$ share two different upgraded terms and that always fails. So, we move to the second disjunct, forgetting all the decisions of the first disjunct. The second disjunct succeeds, since here we only associate $x$ with 10.

Next, we consider all, which takes any number of antecedents. When every argument to all succeeds, then the expression succeeds. But, if any argument fails, instead of failing it looks for alternatives. This is very different from all!!, which is like and, but for outcomes instead of Boolean values.

```
(define test/all
  (exists (w x y)
    (all
      (first-conjunct w x y)
      (second-conjunct w x y))))

(define first-conjunct
  (lambda (w x y)
    (any
      (all!!
        (trace-vars "1" (w x y))
        (== y 1000)
        (trace-vars "2" (w x y))
        (== x 100)
        (trace-vars "3" (w x y))
        (all!!
          (trace-vars "4" (w x y))
          (== y 100)
          (trace-vars "5" (w x y))
          (== x 10)
          (trace-vars "6" (w x y)))))))
```
(define second-conjunct
  (lambda (w x y)
    (any
     (all!!
      (trace-vars "7" (w x y))
      (== y 1000)
      (trace-vars "8" (w x y))
      (== w 2000)
      (trace-vars "9" (w x y)))
     (all!!
      (== y 1000)
      (trace-vars "19" (w x y))
      (== w 3000)
      (trace-vars "20" (w x y))))))

> (define answer1 (query test/all))
1 (w x y) (w.0 x.0 y.0)
2 (w x y) (w.0 x.0 1000)
3 (w x y) (w.0 100 1000)
7 (w x y) (w.0 100 1000)
8 (w x y) (2000 100 1000)
9 (w x y) (2000 100 1000)
> (cdr answer1)
#<fk>

The all above has two conjuncts. Each of the conjuncts is a disjunct. Each disjunct has two all!!s. Clearly the first all!! of first-conjunct happens first. That is why the 1, 2, and 3 appear. By the time that 3 appears, we know that w is uninstantiated, but x is 100 and y is 1000. So, we skip the second all!! of first-conjunct, since we have succeeded. Now, we have to move onto second-conjunct. Again, the first all!! succeeds, which accounts for the 7, 8, and 9, and we terminate. But, we terminate with enough information in the cdr of answer1 to try to find some more answers. This is pretty subtle. We are going to lie by invoking the procedure #<fk> residing in the cdr of answer1. We are going to change our minds about the most recent success. This is called backtracking. We pretend that the first all!! of second-conjunct was unsuccessful. So, we try the second all!! of the second-conjunct, and sure enough it succeeds, which is why we see 19 and 20.

> (define answer2 ((cdr answer1)))
19 (w x y) (w.0 100 1000)
20 (w x y) (3000 100 1000)
> (cdr answer2)
#<fk>
And, we get back another answer, which holds enough information to force yet another lie about the most recent success. This time, we reset the variables \( y \) to 100 and \( x \) to 10. This accounts for the appearance of 4, 5, and 6. Next, we look at the first `all!!` of second-conjunct. Its trace is the same except for the title, which is now 7. Then we attempt to associate \( y \) with 1000, but we can see that it fails, since \( y \) has already been upgraded to 100. When this fails, we get back an empty list, which has no `cdr`, of substitutions as the answer, and so we are done.

\[
> (\text{define } \text{answer3} \ ((\text{cdr } \text{answer2})))
\]

\[
\begin{align*}
4 & (w \ x \ y) \ (w.0 \ x.0 \ y.0) \\
5 & (w \ x \ y) \ (w.0 \ x.0 \ 100) \\
6 & (w \ x \ y) \ (w.0 \ 10 \ 100) \\
7 & (w \ x \ y) \ (w.0 \ 10 \ 100)
\end{align*}
\]

\[
> \text{answer3}
\]

\[
()
\]

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

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

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

\[
\text{(all!! a b)} = (\text{all! (all! a) (all! b)}).
\]

Now consider these three new antecedents:

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

When context is placed around these antecedents, different behaviors can be observed. The first antecedent is equivalent to `(all a b c fail)`. When the `fail` antecedent fails, \( c \) is then retried. For the second and third, however, when `fail` fails, the inner antecedent is skipped over and \( a \) is retried.

This concludes the discussion of Outcome-Oriented Programming. The next installment explains Relation-Oriented Programming.