Outcome-Oriented Programming (5/12/2004)

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 paper is the first in a series of presentations on *logic programming*. Logic programming is one of the major programming paradigms in current use, and differs significantly from the *functional*, *procedural*, and *object-oriented* paradigms you may have encountered in the past.[1] Throughout our exposition we will use the logic programming language *miniKanren*[2], which is a language built "on top" of Scheme through extensive use of Scheme's macro facility. Although we will use the miniKanren language as our notation, the ideas we will explore are general, and can be applied to other logic programming languages such as Prolog.

Two of the most important concepts in logic programming are *outcomes* and *relations*. In this paper we will learn about outcomes, and their place within the "mini-paradigm" of *Outcome-Oriented Programming*. A following presentation will introduce relations and *Relation-Oriented Programming*. Only after we have mastered the notions of outcomes and relations will we be ready to tackle our third leg, entitled *Logic-Oriented Programming*.

An *outcome* is either successful or unsuccessful. We say that we have *succeeded* when an outcome is successful and we say that we have *failed* when an outcome is unsuccessful. Outcomes loosely correspond to Boolean values, although it is *inaccurate* to say that if something succeeds, it is true and that if something fails, it is false. Characterizing an outcome as a Boolean value is misleading, and obscures the intent, power, and limitations of logic programming. One should therefore avoid conflating Boolean values with outcomes.

Here are the fundamental types we use:

```
            Fk = () -> Ans
            Sk = Fk -> Substitution -> Ans
    Antecedent = Sk -> Sk
```

where `->` (arrow) notation indicates the input and output types for a function. An

[1] Smalltalk, Eiffel, Java, and C# are examples of object-oriented languages, while C, Pascal, and Fortran are all procedural languages. Language theorists categorize Scheme and ML as "impure" functional languages, since they both allow side-effects, unlike the "pure" functional language Haskell.

[2] You can learn more about miniKanren by visiting the Kanren Sourceforge site at `http://kanren.sourceforge.net/`.

`Fk` is a *failure continuation*, which is a thunk that returns an answer of type `Ans`. `Substitution`s are like environments in that variables are associated with values, although we shall discover that `Substitution`s differ from environments in several important ways. An *answer* `Ans` is any type. An `Sk` is a *success continuation*, and an `Antecedent` is a success continuation transformer. Thus, an `Antecedent` is a function of the form:

```
(lambda (sk)
  (lambda (fk)
    (lambda (subst)
      ...)))
```

In the grammar below, $x*$ denotes a sequence of zero or more occurrences of x. We describe antecedents `A`. `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 ::   (project (Id*) A A*)
     | (exists (Id*) A A*)
     ! (eigen (Id*) A A*)
     | (predicate Scheme-Expression)
     | (== Term Term)
     | (ef A A A)
     | (ef/only A A A)
     | (ef/forget A A A)
     | (ef/only/forget A A A)
     | (all A*)
     | (any A*)
     | (fail)
     | (succeed)
     | (fails A)
     | (only A)
     | (forget A)
     | (only/forget A)
     | (all!! A*)
     | (all! A*)
     | (trace-vars Title (Id*))
```

There is one expression, the `run` expression, that is used to interface with the logic system. Here is its syntax. `(run (Id*) A fk subst SE FE)` where the arbitrary Scheme expressions `SE` and `FE` are evaluated upon success or failure of `A`, respectively.

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 `exists`. There is an

operation, `project`[3], that treats a logically-scoped variable as if it were lexically scoped.

```
(project (a b c) A)
```

is the same as

```
(lambda (sk)
  (lambda (fk)
    (lambda (subst)
      (let ([a (subst-in a subst)]
            [b (subst-in b subst)]
            [c (subst-in c subst)])
        (((A sk) fk) subst)))))
```

What this says is that if a logic variable is in the substitution (`subst-in` is sort of like an environment lookup, except it can also be passed something that is not a logic variable.) 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 `run` form runs the antecedent in the scope of logic variables `Id*` and if it succeeds binds a failure continuation and a substitution. In addition, `run` evaluates the specified success-expression `SE` if the antecedent succeeds, and otherwise evaluates the failure-expression `FE`. 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))))
```

[3] The "o" in `project` is pronounced "oh".

```
(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)))))))))))

> (run () test fk subst (fk) #f)
x =1= x.0
v =1= v.0
i =1= i.0
w =1= w.0

15
x =2= 10
v =2= 5
i =2= w.0
w =2= w.0

16
x =3= 10
v =3= 5
i =3= 1
w =3= 1

17
#f
```

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 `all!!`, just as we can place the same Scheme expression in an `and` expression.

If we make a change in the code above by replacing the comment `;__________` by `(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: `(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 `(fails ...)` around the antecedent, just as we wrap `(not ...)` 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 `(ef/only ...)` that appears to act like an `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, `(fail)`[4] causes it to take the fail branch.

```
(define test-ef/only
  (exists (x v i w)
    (all!!
      (trace-vars "=1=" (x v i w))
      (== x 10)
      (== v 5)
      (== i w)
      (ef/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)))))))))
```

```
> (run () test-ef/only fk subst (fk) #f)
x =1= x.0
v =1= v.0
i =1= i.0
w =1= w.0

x =4= 10
v =4= 5
```

[4] The `(fail)` antecedent can be defined as `(any)`.

```
i =4= w.0
w =4= w.0

2015
#f
```

Now, let's replace the first argument in the `ef/only` expression by `(== i 1)`.

```
> (run () test-ef/only fk subst (fk) #f)
x =1= x.0
v =1= v.0
i =1= i.0
w =1= w.0

15
x =2= 10
v =2= 5
i =2= 1
w =2= 1

16
x =3= 10
v =3= 5
i =3= 1
w =3= 1

17
#f
```

Here we see that part of the semantics of `ef/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 `ef/only` by `(fails (== i 1))`.

```
> (run () test-ef/only fk subst (fk) #f)
x =1= x.0
v =1= v.0
i =1= i.0
w =1= w.0

x =4= 10
v =4= 5
i =4= w.0
w =4= w.0

2015
#f
```

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. Interestingly, if we replace `ef/only` by `ef/only/forget`, then if the test succeeds, it *also* forgets the new material it added to the substitution as part of determining the outcome of the test. Thus, it is as if we simply used `(succeed)`[5], which always succeeds, as the test. Of course, we have the benefit of finding out (like an acceptor) if the test indeed succeeds. Later we discuss `ef` and `ef/forget`.

Now that we have a better grasp of `ef/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. `(any A1 A2)` is the same as if we had written `(ef A1 (succeed) A2)`, where `ef` is like `ef/only`, but allows its test to be retried. So, very little needs to be said about it. Again, it is important that we remember that running `A1` succeeds or fails. If it succeeds, then `A1`'s resultant substitution is not forgotten. Here, in order for `all!!` to succeed, its two antecedents, `(trace-vars "=1=" (x w))` and `(== x w)` must succeed. The `(trace-vars ...)` always succeeds. And having `x` share with `w` always succeeds if either `x` or `w` is uninstantiated. Since both are uninstantiated, and we only need one, clearly the first *disjunct* succeeds, which is why we don't see a `=2=` in the first output of the example below.

```
> (define answer1
    (run (x w)
      (any
        (all!!
          (trace-vars "=1=" (x w))
          (== x w))
        (all!!
          (trace-vars "=2=" (x w))
          (== x 10)))
      fk subst fk #f))
x =1= x.0
w =1= w.0

> (answer1)
x =2= x.0
w =2= w.0

#<procedure>
```

Now, consider this slight variation.

[5] The `(succeed)` antecedent can be defined as `(all)`.

```
> (define answer1
    (run (x w)
      (any
        (all!!
          (trace-vars "=1=" (x w))
          (== w 1000)
          (trace-vars "=2=" (x w))
          (== x 100)
          (trace-vars "=3=" (x w))
          (== x w))
        (all!!
          (trace-vars "=4=" (x w))
          (== x 10)
          (trace-vars "=5=" (x w))))
      fk subst fk #f))

x =1= x.0
w =1= w.0

x =2= x.0
w =2= 1000

x =3= 100
w =3= 1000

x =4= x.0
w =4= w.0

x =5= 10
w =5= w.0
```

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)
    (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 (run () test/all fk subst fk #f))
w =1= w.0
x =1= x.0
y =1= y.0

w =2= w.0
x =2= x.0
y =2= 1000

w =3= w.0
x =3= 100
y =3= 1000

w =7= w.0
x =7= 100
y =7= 1000

w =8= w.0
x =8= 100
y =8= 1000

w =9= 2000
x =9= 100
y =9= 1000
> 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 move to `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 `answer1` to try to find some more answers. This is pretty subtle. We are going to lie by invoking the procedure `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 (answer1))
w =19= w.0
x =19= 100
y =19= 1000

w =20= 3000
x =20= 100
y =20= 1000
```

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 `#f`, which is the value returned by evaluating the last argument of `run`.

```
> (define answer3 (answer2))
w =4= w.0
x =4= x.0
y =4= y.0

w =5= w.0
x =5= x.0
y =5= 100

w =6= w.0
x =6= 10
y =6= 100

w =7= w.0
x =7= 10
y =7= 100

> answer3
#f
```

An alternative is to return a list of pairs to a Scheme function. For example, instead of using the run that we used, we could try this:

```
> ((lambda (x) x)
    (run () test/all fk subst
      (cons (concretize subst) (fk)) '()))

...
...

(((w.0 . 2000) (x.0 . 100) (y.0 . 1000))
 ((w.0 . 3000) (x.0 . 100) (y.0 . 1000)))
```

Of course, any Scheme function could replace `(lambda (x) x)`. The `concretize` function takes any Scheme term (may be or contain logic variables) and renders its logic variables in an eye-pleasing way. The `w.0`, `x.0`, and `y.0` are *not* logic variables. Instead, they are symbols and have no meaning in our logic system. If the actual logic variables are needed, we should return the unconcretized substitution. The last invocation of `fk` returns an empty list, so we yield the *list* of answers. In *Logic-Oriented Programming* we show why using explicit invocations of `fk`, as we have done here, can lead to an infinite loop.

There is one last conjoiner, `all!`. Each of the conjoiners, `all`, `all!`, and `all!!` are similar, but they are not the same. There are two questions whose answers distinguish them. The first question is *Does it allow backtracking within its arguments?*. The second question is *Does it treat itself as `(fail)` when failure backs into it?* For the first question, backtracking is only allowed within `all` and `all!`. For the second question, `all!` and `all!!` treat the expression as if it had been `(fail)` when failure backs into it. Thus in a context, it would push failure into the antecedent that preceded it. But `all` is quite different, since it allows backtracking into it.

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!`: `(all!! a b)` = `(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))
```

By placing context around these antecedents, we can observe different behaviors. The first antecedent is equivalent to `(all a b c (fail))`. When `(fail)` fails, `c` is then retried. For the second and third, however, when `(fail)` fails, the inner antecedent is skipped over and `a` is retried. We can also describe `all!` in terms of `all` and `all!!`. `(all! a b)` = `(all!! (all a b))`. The `all` allows backtracking within the `a` and `b`, but when failure backs into the `all!!` antecedent, it forces failure to the previous antecedent if there is one.

In conclusion, `all` and `all!` are similar in that within them, backtracking occurs, but `all!!` and `all!` are similar in that when failure backs into them, they fail. Of course, they are all similiar, since their arguments must succeed for them to succeed.

The form `ef`, unlike `ef/only`, allows for the test to be backtracked into. `ef/forget` forces the success continuation to forget about the substitution generated by the test (it uses the original substitution) and `ef/only/forget` both forces the success continuation to forget about the substitution and prohibits the test from being backtracked into. The third argument is tried in the original success continuation, original failure continuation, and original substitution. The antecedent constructors, `only`, `forget`, and `only/forget` are simple. If its argument fails, the expression fails, but the `only` form uses the original failure continuation, so it cannot be backtracked into. The `forget` form uses the original substitution, thus it forgets anything generated by its argument. This last feature allows for the use of antecedents as acceptors, rather than as generators of extended substitutions.

There are times when we want to lexically bind variables to generated symbols. The purpose of these lexical bindings is to give us a ready supply of arbitrarily chosen names. Thus in logic, we can say things like to prove a universally quantified predicate `p`, we merely need to show the predicate holds for an arbitrarily chosen value. Thus, we write `(eigen (x) (p x))`, and if this antecedent succeeds, we know that it holds for all `x`. We will see its first use in *Type Inference and Type Habitation*.

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