dxo: A System for Relational Algebra and Differentiation

JULIE S. STEELE, Georgetown Day School, USA
WILLIAM E. BYRD, University of Alabama at Birmingham, USA

We present dxo, a relational system for algebra and differentiation, written in miniKanren. dxo operates over math expressions expressed as s-expressions. dxo supports addition, multiplication, exponentiation, variables (represented as tagged symbols), and natural numbers (represented as little-endian binary lists). We show the full code for dxo, and describe in detail the four main relations that compose dxo. We present example problems dxo can solve by combining the main relations. Our differentiation relation, do, can differentiate polynomials, and by running backwards, can also integrate. Similarly, our simplification relation, simpo, can simplify expressions that include addition, multiplication, exponentiation, variables, and natural numbers, and by running backwards, can complicate any expression in simplified form. Our evaluation relation, evalo, takes the same types of expressions as simpo, along with an environment associating variables with natural numbers. evalo can produce a natural number by evaluating the expression with respect to the environment; by running backwards, evalo can generate expressions (or the associated environments) that evaluate to a given value. reordero also takes the same types of expressions as simpo, and relates reordered expressions.

CCS Concepts: • Computing methodologies → Computer algebra systems; • Software and its engineering → Functional languages; Constraint and logic languages.

Additional Key Words and Phrases: relational programming, differentiation, simplification, miniKanren, Racket, Scheme

ACM Reference Format:
https://doi.org/10.1145/nnnnnn.nnnnnnn

1 INTRODUCTION

Consider this calculus problem:

Find two different polynomials, \( f(x) \) and \( g(x) \), and two different natural numbers \( a \) and \( b \), such that 
\[
f'(a) = b, \quad \text{and} \quad g'(b) = a.
\]

Differentiating polynomials is an easy calculus problem, but the problem above is more complicated because of the relationships between the polynomials, their derivatives, and the two natural numbers. We invite the reader to pause, try to find solutions to this problem, and to think about how these types of problems might be solved more generally.

We have developed a relational algebra system, dxo, that uses relational programming to solve problems like the one above. We show the run expression for solving this problem in Section 2. dxo is a collection of four main relations: simpo for simplification, do for differentiation, evalo for evaluation, and reordero for permuting arguments. Implementing dxo relationally makes it flexible. For example, the relation do can differentiate polynomials with respect to some variable. Since do is a relation, it can also integrate polynomials. Also, the expression to be differentiated and the derivative can both contain fresh logic variables. The relations simpo, evalo, and reordero similarly benefit from this flexibility.

We assume the reader is familiar with core miniKanren [Byrd 2009; Byrd and Friedman 2006; Friedman et al. 2018] (\( =\), \( \not=\), fresh, conde, run), extended with disequality (\( =/=\)) and absento constraints [Byrd et al. 2012]. Detailed explanations to the core miniKanren language can be found in Friedman et al. [2018], Byrd [2009], and Byrd and...
Friedman [2006]. Descriptions of disequality and absento constraints can be found in Byrd et al. [2012] and Byrd et al. [2017].

Section 2 gives a high level explanation of \(dxo\), its uses, and its four main relational functions. Section 3 explains the four main relations. Appendix A contains the full implementation of \(dxo\).

## 2 HIGH LEVEL OVERVIEW

\(dxo\) is composed of four main relational functions, \(simpo\), \(do\), \(evalo\), and \(reordero\), that when used in combination can solve interesting differentiation math problems. Here are the four relations and their uses:

- \((simpo \, comp \, simp)\) relates \(comp\) and \(simp\), where \(comp\) can be any arithmetic expression and \(simp\) is an equivalent, fully simplified one;
- \((do \, x \, expr \, deriv)\) relates a polynomial expression in terms of \(x\), \(expr\), with its derivative, \(deriv\), where the derivative is with respect to \(x\);
- \((evalo \, env \, expr \, value)\) relates an expression, \(expr\), and its value where each variable in \(expr\) has a value given by the \(env\) association list;
- \((reordero \, e1 \, e2)\) relates two equivalent expressions, where one is made by changing the order of subexpressions in an addition or multiplication in any level of the expression.

Figure 1 contains the grammar for \(simpo\), \(evalo\), and \(reordero\), and Figure 2 contains the grammar for \(do\), which has the difference of a restriction to be only polynomials (no variable on the exponent). The implementation of \(dxo\) uses the relational arithmetic system created by Oleg Kiselyov, which is presented in [Friedman et al. 2018] and [Kiselyov et al. 2008].
These main functions can be used together in many ways. A built in way is anydo, but the best way to combine them is on your own for a problem.

(anydo expr deriv x), like do, relates expressions and their derivatives with respect to x, except anydo generalizes this to simplified, complicated, or reordered forms of expr and deriv. For example this use of anydo succeeds:

(anydo '(+ (^ (var x) (num (1 1))) (^ (var x) (num ())))) ; x^3 = expr

'(* (num (1)) (^ (var x) (num (0 1))) (num (1 1))) ; 1*x^2*3 = deriv

This succeeds because \( \frac{d}{dx}[x^3] = x^2 \cdot 3 \), and similarly,

(do 'x '(* (var x) (num (1 1))) '(* (^ (var x) (num (0 1))) (num (1 1))))

succeeds. Then anydo calls simpo, complicating the expressions with +x^0 and 1*. Note that calling do with the same arguments fails. anydo is centered around a do relation with arguments similar to expr and deriv, ecomp and dcomp. expr and ecomp are similar in that they simplify to the same value, esimp, making them equivalent. anydo does the same for deriv, except also including a reordering.

(define anydo
  (lambda (expr deriv x)
    (fresh (esimp dsimp ecomp dcomp dcorder)
      (simpo expr esimp)
      (simpo ecomp esimp)
      (do x ecomp dcomp)
      (reordero dcomp dcorder)
      (simpo dcorder dsimp)
      (simpo deriv dsimp))))

anydo is useful because it generalizes the derivative relation and allows you not to worry about the form expr and especially deriv are entered.

Returning back to the problem proposed in the introduction, find two different polynomials, \( f(x) \) and \( g(x) \), and two different natural numbers \( a \) and \( b \), such that \( f'(a) = b \), and \( g'(b) = a \), we can solve this with dxo. We relate \( f \) and \( g \) with their derivatives, \( fd \) and \( gd \), using do. Then we use evalo to evaluate these derivatives at \( a \) and \( b \) respectively (we do this by making one environment where \( x \) is \( a \) and one where \( x \) is \( b \)), and set the evaluation to \( b \) and \( a \) respectively. Last, we make sure \( f \) and \( g \) are different but both simplified and \( a \) and \( b \) are different.

(run 20 (f g envb enva)
  (fresh (b a gd fd)
    (=/= f g)
    (=/= b a)
    (== `((x ,b)) envb)
    (== `((x ,a)) enva)
    (do 'x f fd)
    (simpo f f)
    (do 'x g gd)
    (simpo g g)
    (evalo enva fd b)
    (evalo envb gd a)))

⇒ , Vol. 1, No. 1, Article . Publication date: July 2020.
3 CODE EXPLANATION

In this section we explain in detail the four main relations in dxo.¹

3.1 simpo

(simpo comp simp) relates comp and simp, where comp can be any arithmetic expression and simp is an equivalent, fully simplified one. Simplified means simplifying the following expressions:

- \( v + 0 = v \);
- \( v * 0 = 0 \);
- \( v * 1 = 1 \);
- \( v^0 = 1 \text{ (if } v \neq 0) \);
- \( v^1 = v \);
- \( 0^v = 0 \) (if \( v \) is a nonzero number);
- \( 1^v = 1 \);

where \( v \) is any expression. For example, let's simplify \( 0^5 + (2 * 1) \)

\[
\text{(run* (simp) (simpo `(+ (^ (num ()) (num (1 0 1))) \ (* (num (0 1)) (num (1) ))) simp))}
\]

\[
\Rightarrow ((\text{num (0 1)}) \quad ; 2 = \text{simp})
\]

simpo has base cases of \((= \text{comp simp})\) for just a number or variable. The three non base cases, addition, multiplication, and exponentiation, recursively simplify sub expressions. If comp is \`(^ e1 e2)\), simpo only needs to recur on two parts, but if comp is a multiplication or addition, simpo relates the first terms and also the rest. For \((== \`(* . ,e . ,e*) \text{ comp})\), simpo simplifies e to be s: \((\text{simpo e s})\). If s is not a simplifiable case (s is 0 or 1), then we try \((\text{simpo \`(* . ,e*) temp})\), and \((== \`(* . ,t*) \text{ temp})\), using temp as a way to recur through the rest of the expressions being added. This method works because through recursion, we can check that each part of the multiplication does not contain a simplifiable piece. simpo handles addition similarly.

If comp is ground, \((\text{simpo comp simp})\) will either succeed exactly once or fail, but if comp is fresh, then simpo can cause an infinite loop. When given a ground comp, simpo will always succeed or fail because there is at most one way to simplify any concrete expression. If comp is fresh, then simpo could succeed, but if it is an impossible relation, simpo can try longer and longer comps, never succeeding, and cause an infinite loop.

An example of this is that running \((\text{simpo comp simp})\) with comp as \(1^1\) and simp as a logic variable succeeds because \(1^1\) simplified is 1. Running with comp as a logic variable and simp as (the unsimplified) \(1^1\) diverges, searching for a comp forever.

¹We have released the dxo code under an MIT licence at https://github.com/JShermanSteele/dxo .
We can also construct a case with a ground comp and partially bound simp where simpo will fail. If comp is still \(1^1\), simplifying to 1, but we require simp to be some addition expression, simpo will fail. As explained above, with a ground comp, simpo will always fail or succeed and never diverge.

\[
\text{(run \(24\) (expr deriv) (do \('x\ expr deriv) ))}
\]

\[
\Rightarrow
\]

\[
\text{('...}
\]

\[
\text{((^ (var x) (num (0 1))))} \quad ; x^2 = \text{expr}
\]

\[
\text{* (^ (var x) (num (1))) (num (0 1)))} \quad ; x^1 * 2 = \text{deriv}
\]

\[
\text{...}
\]

\[
\text{((^ (var x) (num (1 _ .0 . _1)))))} \quad ; x^a \text{ where } a \text{ is odd} = \text{expr}
\]

\[
\text{* (^ (var x) (num (0 _ .0 . _1)))} \quad ; x^{a-1} * a = \text{deriv}
\]

\[
\text{(num (1 _ .0 . _1))))}
\]

\[
\text{...}
\]

\[
\text{((^ (var x) (num ())) (^ (var x) (num ()))))} \quad ; x^0 * x^0 = \text{expr}
\]

\[
\text{(+ (num ())) (^ (var x) (num ())))} \quad ; 0 * x^0 + x^0 = \text{deriv}
\]

\[
\text{(* (num ())) (^ (var x) (num ())))} \quad ; x^0 * x^0 = \text{deriv}
\]

\[
\text{(* (num ())) (num ()))}
\]

expr is either a variable, a number, an exponent expression, a sum, or a multiplication, so do can go through these cases for all expr-deriv relations. Since the derivative of a sum is the sum of the derivatives of its parts, when expr and deriv are sums, each term is related using do. Since the sum can have any positive number of terms, a helper function, map-do-o, relates each respective term in the sums. If expr is a multiplication, do must
use the multiplication rule that
\[
\frac{d}{dx}(ab) = \frac{da}{dx}b + a \frac{db}{dx},
\]
and recur down the list of subexpressions being multiplied. To improve the code for this process, we wrote a helper function, `multruleo`, that relates the list of subexpressions being multiplied and the multiplication’s derivative. If the list has length more than one, `multruleo` separates the first term, \(e\), and the rest, \(e^*\). Applying the multiplication rule gets \(\frac{de}{dx}(e^*) + e^* \frac{d}{dx}(e)\), which is recursive with `multruleo` because \(\frac{d}{dx}(e)\) is the related derivative argument to `multruleo` with \(e^*\) as the first argument.

We could recur through \(expr\) using `do` with \(`(\ast . ,e^*)`\) as an argument, but our approach is simpler because it does not exit `multruleo` while going through \(expr\)’s multiplication.

If \(expr\) is an exponent expression, the second subexpression in the exponent must be a number by `do`’s grammar, so \(\frac{d}{dx}[x^n] = n(x^{n-1})\) where \(n\) is any number. There are three clauses for constants, \(\frac{d}{dx}[x^0] = 0\), \(\frac{d}{dx}[x^m] = 0\), and \(\frac{d}{dx}[n] = 0\) where \(n\) and \(m\) are any numbers. Finally, the derivative of just \(x\) is one.

Since `do` orders `deriv` a certain way, some integratable `derivs` will fail. This is why `do` should be used with `reordero`. For example,

\[
\text{(run 1 (EXPR)}
\text{(do 'x expr '(\ast (^ (var x) (num (1))) (num (0 1)))) \; x^1 \ast 2 = deriv)}
\Rightarrow
\text{('(\ast (var x) (num (0 1)))) \; x^2 = expr} \]  

produces an answer, but switching `deriv`’s multiplication order loops infinitely:

\[
\text{(run 1 (EXPR)}
\text{(do 'x expr '(\ast (num (0 1)) ('(\ast (var x) (num (1)))))) \; 2 \ast x^1 = deriv)}
\]  

Similarly to `simpo`, `do` always succeeds or fails running “forwards,” when `expr` and `x` are ground. With `expr` fresh, `do` will succeed if possible, but can loop infinitely just like `simpo`.

Unfortunately, `do`’s approach to differentiation uses polynomial rules instead of a deeper encoding of differentiation. In the future, we would like to support expressions containing multiple variables, as well as expressions like \(2^x\) and \(\sin\). Mainly, we are working on replacing `do` with relations that do forwards automatic differentiation and backwards automatic differentiation.
3.3 evalo

evalo is useful for solving equations and for evaluation. (evalo env expr value) relates an expression, expr, and its value where each variable in expr has a value given by the env association list. For example we can look for expressions that evaluate to 8 where we have a variable, \( z = 2 \):

\[
\text{(run 200 (expr) (evalo '((z . (0 1))) expr '(0 0 0 1))) ;z=2, value=8}
\]

\[\Rightarrow '\ldots '(* (var z) (num (0 0 1))) ;z*4 = expr\]
\[\ldots '^ (num (0 1)) (num (1 1))) ;2^3 = expr\]
\[\ldots '(+ (var z) (num ())) (num (0 1 1))) ;z + 0 + 6 = expr\]

evalo goes through expr evaluating lists into minikanren numbers. evalo has base cases of expr being just a number or variable which are evaluated to the number or the variable’s value from env respectively. To relate variables and their values, evalo calls lookupo which checks down the association list, env. If expr is a sum, multiplication, or an exponent expression, then evalo relates the first term with its value, the rest with its value, and then puts them together. The evalo code for addition does this:

\[
(== (^
(** +, ,rest) expr))
\]

(conde

\[
((== () rest) (evalo env c value))
\]

\[
(=/= () rest)
\]

\[
(evalo env *(+, ,rest) evrest)
\]

\[
(pluso evc evrest value)))
\]

This will recur through rest and sum all the parts to relate expr with value.

An interesting use of evalo is to solve algebra problems by making env fresh, for example looking for Pythagorean triples. So we set up \( x^2 + y^2 = z^2 \) and also a \( z * z = z \text{-squared} \) relation to make sure that \( z \) is a natural number to find the classic Pythagorean triple!

\[
\text{(run 1 (env)}
\]
\[
\text{(fresh (xv yv zw z2v))}
\]
\[
(== '((x .. xv) (y .. yv) (z .. zw) (z-squared .. z2v)) env)
\]
\[
\text{(evalo env (+( cvarov x) (num (0 1))) (cvarov y) (num (0 1)))) z2v)
\]
\[
\text{(o zv zv z2v))}
\]

\[
'(((x) (y) (z) (z-squared)))
\]

;\( x = 0, y = 0, z = 0 \)

Not what we wanted, alas. Setting non-zero constraints, though, produces:

\[
\text{(run 1 (env}}
\]
\[
\text{(fresh (xv yv zw z2v))}
\]
\[
\text{(poso xv)}
\]
\[
\text{(poso yv)}
\]
\[
\text{(poso zv)}
\]
\[
(== '((x .. xv) (y .. yv) (z .. zw) (z-squared .. z2v)) env)
\]
\[
\text{(evalo env (+( cvarov x) (num (0 1))) (cvarov y) (num (0 1)))) z2v)
\]

, Vol. 1, No. 1, Article . Publication date: July 2020.
\[(\ast o z v z v z2v))\]

\[
\begin{align*}
(x & 1 1) & ; x = 3 \\
(y & 0 0 1) & ; y = 4 \\
(z & 1 0 1) & ; z = 5 \\
(z = \text{squared} & 1 0 0 1 1)) & ; z^2 = 25 \\
\end{align*}
\]

which is a 3-4-5 right triangle.

If either env or expr is fresh, evalo can loop infinitely, trying more and more complicated envs or exprs. If they are both ground, then evalo will terminate since we wrote evalo to run simply forwards in this case.

We would like to make evalo infinite loop less in the future. Also we could maybe make it have the ability to keep some variables evaluated but evaluate everything else.

### 3.4 reordero

(reordero e1 e2) relates two equivalent expressions, where one is made by changing the order of subexpressions in an addition or multiplication in any level of the expression. As said above, it is useful for taking integrals with do. We can use reordero to find all reorderings of an expression:

\[
\begin{align*}
\text{run} & \ast (\text{e2}) \ (\text{reordero} \ ' (+ \ (\text{num} \ (1)) \ (* \ (\text{num} \ (0 \ 1)) \ (\text{num} \ (1 \ 1))) \ ) \ \text{e2}) \ ; 1 + 2 * 3 = \ e1 \\
\Rightarrow & \ '\(+ \ (\text{num} \ (1)) \ (* \ (\text{num} \ (0 \ 1)) \ (\text{num} \ (1 \ 1)))) \ ; 1 + 2 * 3 = \ e2 \\
& (+ (* \ (\text{num} \ (0 \ 1)) \ (\text{num} \ (1 \ 1))) \ (\text{num} \ (1))) \ ; 2 * 3 + 1 = \ e2 \\
& (+ (* \ (\text{num} \ (1 \ 1)) \ (\text{num} \ (0 \ 1))) \ (\text{num} \ (0 \ 1))) \ ; 1 + 3 * 2 = \ e2 \\
& (+ (* \ (\text{num} \ (1 \ 1)) \ (\text{num} \ (0 \ 1))) \ (\text{num} \ (1))) \ ; 3 * 2 + 1 = \ e2 \\
\end{align*}
\]

(reordero e1 e2) relates e1 and e2 by having the same outer operation. For addition and multiplication the code is,

\[
\begin{align*}
\text{fresh } & (o \ e1* \ e2*) \\
(== & '\(, , e1*) e1) \\
(== & '\(, , e2*) e2) \\
\text{typeo} & o ' +or* \\
\text{reorderitemso} & \ e1* \ e2*))
\end{align*}
\]

where e1* and e2* are permutations of each other. reorderitemso just checks that e1* and e2* have the same length, and then calls reorderinnero on them. We created reorderitemso to improve speed and divergence behavior by requiring e1* and e2* have the same length before considering the relations in reorderinnero. reorderinnero does the job of relating permuted lists at any depth. (reorderinnero e1* e2*) one-by-one goes through the items in e1* and removes them from e1* and e2*, and recursively calls itself until the base case that both e1* and e2* are the empty list. The key to make it deeply reorder is that the item it removes from e1* and e2* are not exactly the same, they are related by reordero.

(\define\ reorderinnero
\lambda\ (e1* e2*)
\text{fresh } (c1 rc1 rest1 rest2)
\text{conde}
\begin{align*}
(== & '() e1*) (== '() e2*) \\
(== & '\(, , rest1) e1*) \\
\text{removeo} & rc1 e2* rest2)
\end{align*}
\)
reorder greatly reduces infinite loops because reorder checks that its arguments are the same length. This check keeps e1 and e2 the same structure and length at every depth, keeping the search finite. Checks like this would be useful to add other places in dxo to reduce divergence. A possible improvement would be making reorder able to do distribution and be able to relate equivalent expressions with different parentheseshttps://www.overleaf.com/project/5e93a4c21d9c8a00013296f3es.

4 OPEN PROBLEMS
dxo could be improved by expanding the grammars, improving the speed and termination, and using automatic differentiation. We would like to add expressions like $2^x$, $\sin(x)$, and multiple variables. Currently, dxo searches inefficiently, especially combinations of functions like anydo, so we would like to speed these up. We would also like to make more calls terminate. We are interested in improving simpo, possibly implementing Knuth-Bendix Completion relationally. Next, we would like to make a function to replace do that automatic differentiates forwards and backwards.

5 RELATED WORK
Expresso [Schünemann 2017] is a relational algebra system written in Clojure using the miniKanren-inspired core.logic library. Like dxo, it includes algebraic simplification, differentiation, and evaluation. Beyond dxo, it includes rewriting in normal form and expressions like sin.

The Reduce-Algebraic-Expressions system in Prolog [Jasoria 2019] is similar to simpo, using certain simplification identities. simplifies expressions like $((x + x)/x) \cdot (y + y - y) \Rightarrow 2 \cdot y$. It can make simplifications like $x + x \Rightarrow 2 \cdot x$ which simpo cannot since simpo currently only includes simplification rules involving 0 and 1.

The Reduce-Algebraic-Expressions system is not relational.

6 CONCLUSION
dxo applies relational programming to algebra and differentiation. It can differentiate, integrate, simplify, complicate, evaluate, create, and reorder. dxo can concisely represent non-trivial math problems and find solutions. dxo is a foundation for future exploration of relational programming in algebra.

ACKNOWLEDGMENTS
We are grateful for the work of all the relational programmers whose work we have built upon. Our work would not have existed without the efforts of Dan Friedman and Oleg Kiselyov. We would like to thank Brandon T. Willard and Alan T. Sherman for comments on a draft of this paper and sharing ideas with us. Finally, we thank the anonymous reviewers for their many helpful comments.

REFERENCES

ACM, New York, NY, USA, 8–29. https://doi.org/10.1145/2661103.2661105
A FULL IMPLEMENTATION OF DXO

(require "faster-miniKanren-master/mk.rkt")
(require "faster-miniKanren-master/numbers.rkt")

; defines ^ as expt
(define ^ (lambda (a b) (expt a b)))

; defines ZERO and ONE
(define ZERO `(num ,(build-num 0)))
(define ONE `(num ,(build-num 1)))

; multiplies: (* a b)=c
(define timeso
  (lambda (a b c)
    (fresh (bml rec d)
      (conde
        ((== (build-num 0) b) (== (build-num 0) c))
        (;(poso b)
         (=/= (build-num 0) b)
         (pluso bml (build-num 1) b)
         (timeso a bml rec)
         (pluso a rec c)))))))

; exponent: (^ a b)=c
(define expo
  (lambda (a b c)
    (fresh (bml rec)
      (conde
        ((== (build-num 0) b) (== (build-num 1) c))
        ((=/= (build-num 0) b)
         (pluso bml (build-num 1) b)
         (expo a bml rec)
         (timeso a bml rec)
         (pluso a rec c)))))

; atom?
(define atom?
  (lambda (expr)
    (cond
      ((list? expr) #f)
      ((null? expr) #f)
      (else #t)))))

; atom, null, or list
(define typeo
(lambda (expr answer)
  (fresh (a b)
    (conde
      (((= `,(a . ,b) expr) (= 'list answer) (= 'num a) (= 'var a))
      (((= () expr) (= 'null answer))
      (((= `(num ,a) expr) (= 'atom answer))
      (((= `(var ,a) expr) (= 'atom answer))
      (((= '+ expr) (= '+or* answer))
      (((= '* expr) (= '+or* answer)))))))))

;minikanren number
(define numo
  (lambda (n)
    (fresh (b rest)
      (conde
        (((= () n))
        (((= `(,b . ,rest) n)
          (conde
            (((= '1 b))
            (((= '0 b))
            (numo rest)))))))))

;empty env
(define empty-env `())

;ext-env
(define ext-env
  (lambda (x v env)
    (cons `,(x . ,v) env)))

;lookupo
(define lookupo
  (lambda (x env v)
    (fresh (env* y w)
      (conde
        (((= `((,x . ,v) . ,env*) env))
        (((= `((,y . ,w) . ,env*) env) (= x y) (lookupo x env* v)))))))))

;unbuild-numinner to every element and if list, then to list
(define unbuild-numhelper
  (lambda (expr)
    (cond
      ((null? expr) '())
      ((list? (car expr)) (cons (unbuild-numinner (car expr)) (unbuild-numhelper (cdr expr))))
      (else (cons (car expr) (unbuild-numhelper (cdr expr)))))))
; calls unbuild-numinner for every answer in minikanren
(define unbuild-num
  (lambda (expr)
    (cond
      ((null? expr) '())
      (else (cons (unbuild-numinner (car expr)) (unbuild-num (cdr expr)))))))

; undoes build-num by calling unbinary
(define unbuild-numinner
  (lambda (expr)
    (match expr
      [`() `()]
      [`(num ,b) `(num ,(unbinary b 1))]
      [`(var ,b) `(var ,b)]
      [`(+ ,e ,e*) (unbuild-numhelper `(+ ,e ,e*))]
      [`(*, ,e ,e*) (unbuild-numhelper `(* ,e ,e*))]
      [`(pow ,e ,e*) (unbuild-numhelper `(* ,e ,e*))])))

; helper for unbuild-numinner that goes from binary to base 10
(define unbinary
  (lambda (expr n)
    (cond
      ((null? expr) 0)
      ((atom? expr) expr)
      ((equal? (car expr) 1) (+ n (unbinary (cdr expr) (* 2 n))))
      ((equal? (car expr) 0) (unbinary (cdr expr) (* 2 n)))))

; l contains item
(define membero
  (lambda (item l)
    (fresh (a rest)
      (conde
        (==(`,item ,rest) l))
      (==(`,a ,rest) l) (= item a) (membero item rest))))

; l does not contain item
(define notmembero
  (lambda (item l)
    (fresh (a rest)
      (conde
        (==(`,) l))
        (==(`,a ,rest) l)
        (= item a)
        (notmembero item rest))))
(define removeo
  (lambda (item contain removed)
    (fresh (rest rest2 c)
      (conde
        ((== `(,item . ,rest) contain)
          (== rest removed))
        ((== `(,c . ,rest) contain)
          (/= item c)
          (== `(,c . ,rest2) removed)
          (removeo item rest rest2))))))

;the implementation of dxo uses these definitions from the faster-miniKanren
;implementation of miniKanren, based on the relational arithmetic system
; created by Oleg Kiselyov------------------------
(define samelengtho
  (lambda (e1 e2)
    (fresh (c1 c2 rest1 rest2)
      (conde
        ((== `() e1) (== `() e2))
        ((== `(,c1 . ,rest1) e1) (== `(,c2 . ,rest2) e2) (samelengtho rest1 rest2))))))

(define poso
  (lambda (n)
    (fresh (a d)
      (== `(,a . ,d) n)))))

(define >1o
  (lambda (n)
    (fresh (a ad dd)
      (== `(,a ,ad . ,dd) n))))

(define =lo
  (lambda (n m)
    (conde
      ((== '() n) (== '() m))
      ((== '(1) n) (== '(1) m))
      (fresh (a x b y)
        (== `(,a . ,x) n) (poso x)
        (== `(,b . ,y) m) (poso y)
        (=1o x y))))))

(define <lo
  (lambda (n m)
(conde
  ((== '() n) (poso m))
  ((== '(1) n) (>lo m))
  ((fresh (a x b y)
    (== `(a . ,x) n) (poso x)
    (== `(b . ,y) m) (poso y)
    (<lo x y))))))))

(define <=lo
  (lambda (n m)
    (conde
      ((=lo n m))
      ((<lo n m))))))

(define <lo
  (lambda (n m)
    (conde
      ((<lo n m))
      ((=lo n m))
      ((fresh (x)
        (poso x)
        (pluso n x m)))))))

(define <=o
  (lambda (n m)
    (conde
      ((== n m))
      ((<o n m))))))

(define odd-*o
  (lambda (x n m p)
    (fresh (q)
      (bound-*o q p n m)
      (*o x m q)
      (pluso `(0 . ,q) m p))))))

(define bound-*o
  (lambda (q p n m)
    (conde
      ((== '() q) (poso p))
      ((fresh (a0 a1 a2 a3 x y z)
        (== `(a0 . ,x) q)
        (== `(a1 . ,y) p)
        (conde
          ((== '() n)
           (== `(a2 . ,z) m)
(bound-o x y z '())
((= '(a3 . z) n)
 (bound-o x y z m)))))))

(define *o
 (lambda (n m p)
   (conde
     ((= () n) (= () p))
     ((poso n) (= () m) (= () p))
     ((>1o n) (= '(1) m) (= n p))
     ((fresh (x z)
       (= '(0 . x) n) (poso x)
       (= '(0 . z) p) (poso z)
       (>1o m)
       (*o x m z)))
     ((fresh (x y)
       (= '(1 . x) n) (poso x)
       (= '(0 . y) m) (poso y)
       (*o m n p)))
     ((fresh (x y)
       (= '(1 . x) n) (poso x)
       (= '(1 . y) m) (poso y)
       (odd-o x n m p)))))))

"SIMPLIFY":-----------------------------------------------

(define simpo
 (lambda (comp simp)
   (fresh ()
     (conde
       ((fresh (n)
         (= '(num ,n) comp)
         (= comp simp))
       ((fresh (v)
         (= '(var ,v) comp)
         (= comp simp))))

       ((fresh (e1 e2 s1 s2)
         (= '(^ ,e1 ,e2) comp)
         (conde
           ((= ONE s1) (= ONE simp))
           ((= ZERO s1) (=/= ZERO s2) (= ZERO simp))
           ((=/= ZERO s1) (= ZERO s2) (=/= ONE s1) (= ONE simp))
           ((=/= ZERO s1) (= ONE s1) (= ONE s2) (= s1 simp))
           (= '(^ ,s1 ,s2) simp))

   , Vol. 1, No. 1, Article . Publication date: July 2020.
\begin{verbatim}
(=/= ONE s1)
(=/= ONE s2)
(=/= ZERO s1)
(=/= ZERO s2)))
(simpo e1 s1)
(simpo e2 s2)))

((fresh (e e* s temp t* n v)
 (== `(* ,e ,e*) comp)
 (conde
     ((== '() e*) (simpo e simp))
     ((== ZERO s)(=/* '() e*)(== ZERO simp))
     ((== ONE s)(=/* '() e*) (simpo `(* ,e*) simp))
     (=/= ONE s)
     (=/= ZERO s)
     (=/= '() e*)
     (conde
         ((== ZERO temp) (== ZERO simp))
         (== ONE temp) (== s simp))
         (== `(^ ,n) temp) (== `(* ,s ,temp) simp))
         (== `(+ ,n) temp) (== `(* ,s ,temp) simp))
         (== `(num ,n) temp) (=/= ZERO temp)
         (=/= ONE temp) (== `(* ,s ,temp) simp))
         (== `(var ,v) temp) (== `(* ,s ,temp) simp))
         (==`(* ,t*) temp) (== `(* ,s ,t*) simp)))
     (simpo `(* ,e*) temp)))
(simpo e s)
)

((fresh (e e* s temp t* n v)
 (== `(+ ,e ,e*) comp)
 (conde
     ((== '() e*) (simpo e simp))
     (== ZERO s)(=/* '() e*)(simpo `(+ ,e*) simp))
     (=/= ZERO s)
     (=/= '() e*)
     (conde
         ((== ZERO temp) (== s simp))
         (== `(^ ,n) temp) (== `(+ ,s ,temp) simp))
         (== `(+ ,n) temp) (== `(+ ,s ,temp) simp))
         (== `(num ,n) temp) (=/= ZERO temp) (== `(+ ,s ,temp) simp))
         (== `(var ,v) temp) (== `(+ ,s ,temp) simp))
         (==`(+ ,t*) temp) (== `(+ ,s ,t*) simp)))
     (simpo `(+ ,e*) temp)))
(simpo e s))))))
\end{verbatim}
; takes derivative
(define do
  (lambda (x expr deriv)
    (fresh ()
      (symbolo x)
      (conde
        ((fresh (d* e* a b c d)
            (== expr `(+ . ,e*)) (== e* `,(a . ,b))
            (== deriv `(+ . ,d*)) (== d* `,(c . ,d))
            (samelen e* d*)
            (map-do-o x* e* d*))
        ((fresh (a b)
            (== expr `(^ (var ,x) (num ,(build-num 0))))
            (== deriv ZERO))
        )
        ((fresh (l e)
            (== expr `(* . ,l))
            (letrec ((multruleo (lambda (l dd)
                                (fresh (e e* d* a b)
                                  (conde
                                    [(== l `,(e))
                                      (do x e dd)
                                    ]
                                    [(== l `,(e . ,e*))
                                      (== e* `,(a . ,b))
                                      (== dd `(+ (* ,d . ,e*) (* ,e ,d*))
                                      (do x e d)
                                      (multruleo e* d*)
                                    )]
                                  )
                                )
                              )))
            )
            (multruleo l deriv)))))
        ((fresh (int intm1)
            (== expr `(^ (var ,x) (num ,int)))
            (== deriv `(* (^ (var ,x) (num ,intm1)) (num ,int))
            (minuso int (build-num 1 intm1))))
        )
        ((fresh (int1 int2)
            (== expr `(^ (num ,int1) (num ,int2)))
            (== deriv ZERO)
            (conde
              ((poso int1))
            )
        ))
    )
  )
)

"DERIVATIVE";----------------------------------------------------------------------------------
```
((== ZERO int1)(poso int2)))
)

((fresh ()
 (== expr `(var ,x))
 (== deriv ONE)))

((fresh (int)
 (== expr `(num ,int))
 (== deriv ZERO)))))))

; maps do function
(define map-do-o
 (lambda (x expr* output)
  (fresh (e* e out out*)
   (conde
    [(== expr* '()) (== output '())
    [(== expr* `(,e . ,e*)) (== output `(,out . ,out*))
     (do x e out)
     (map-do-o x e* out*)]]))

"EVALUATE";

; evaluator
(define evalo
 (lambda (env expr value)
  (fresh (m x a b rest evc evrest eva evb)
   (conde
    (== `(var ,x) expr) (lookupo x env value))
    (== `(num ,m) expr) (numo m) (== m value))
    (== `(+ ,c . ,rest) expr)
   (conde
    (== () rest) (evalo env c value))
    (=/= () rest)
    (evalo env c evc)
    (evalo env `(+ . ,rest) evrest)
    (*o evc evrest value))))
    (== `(^ ,a ,b) expr)
```
(evalo env a eva)
(evalo env b evb)
(expo eva evb value))))])

"REORDER";______________________________________________________________

; another option instead of using reordero is to always enter expressions in the same right order
; reorders expression deeply, reordering any + and * expressions
(define reordero
  (lambda (e1 e2)
    (fresh ()
      (conde
        ((== e1 e2) (typeo e1 'atom))
        ((fresh (o e1* e2*)
            (== `(' ,o . ,e1*) e1)
            (== `(' ,o . ,e2*) e2)
            (typeo o 'or*)
            (reorderitemso e1* e2*)))))
        ((fresh (a1 b1 a2 b2)
            (== `(' ,a1 ,b1) e1)
            (== `(' ,a2 ,b2) e2)
            (reordero a1 a2)
            (reordero b1 b2)))))
)

; permutes a list by calling reorderinnero, and calls reordero on the items in the list deeply
(define reorderinnero
  (lambda (e1* e2*)
    (fresh ()
      (samellengtho e1* e2*)
      (reorderinnero e1* e2*))))

; permutes and calls reordero on the items, helper for reorderitemso
(define reorderinnero
  (lambda (e1* e2*)
    (fresh (c1 rc1 rest1 rest2)
      (conde
        ((== () e1*)(== () e2*))
        ((== `(',', c1 ,., rest1) e1*)
          (removeo rc1 e2* rest2)
          (reorderinnero rest1 rest2)
          (reordero c1 rc1)
          ))))))
(define anydo
  (lambda (expr deriv x)
    (fresh (ecomp dcomp esimp dsimp dcorder)
      (project (expr deriv)
        (if (var? expr)
          (fresh ()
            (simpo deriv dsimp)
            (simpo dcorder dsimp)
            (reordero dcomp dcorder)
            (do x ecomp dcomp)
            (simpo ecomp esimp)
            (simpo expr esimp))
          (fresh ()
            (simpo expr esimp)
            (simpo ecomp esimp)
            (do x ecomp dcomp)
            (reordero dcomp dcorder)
            (simpo dcorder dsimp)
            (simpo deriv dsimp))))))

(define doitallevalo
  (lambda (ieval inte deriv deval x env)
    (fresh (icomp dcomp isimp dsimp dcorder)
      (evalo env inte ieval)
      (evalo env deriv deval)
      (do x icomp dcomp)
      (reordero dcomp dcorder)
      (simpo deriv dsimp)
      (simpo dcorder dsimp)
      (simpo inte isimp)
      (simpo icomp isimp))))