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, represented 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. By evaluating the expression with respect to the environment, evalo can produce a natural number; 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

## 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 $f'(a) = b$, and $g'(b) = a$.

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 its derivative can both contain fresh logic variables. The relations simpo, evalo, and reordero similarly benefit from this flexibility.
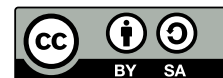
We assume the reader is familiar with core miniKanren [Byrd 2009; Byrd and Friedman 2006; Friedman et al. 2018] (==, `fresh`, `conde`, `run`), extended with disequality (=/=) and `absento` constraints [Byrd et al. 2012]. Detailed explanations to the core miniKanren language can be found in Friedman et 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 relations. Section 3 explains in detail the main relations. Section 4 discusses some open problems and possible future work. Section 5 discusses related work. We conclude the paper in Section 6. Appendix A contains the full implementation of *dxo*.

2 HIGH-LEVEL OVERVIEW

dxo is composed of four main relations, `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 expr with its derivative deriv, where the derivative is with respect to x;

(evalo env expr value) relates an expression expr with its value value, where each variable in expr is associated with a natural number by the environment env;

(reordero e1 e2) relates two equivalent expressions, e1 and e2, by changing the order of subexpressions in an addition or multiplication in any level of the other expression.

Figure 1 contains the grammar for expressions accepted by `simpo`, `evalo`, and `reordero`, and Figure 2 contains the grammar for polynomial expressions accepted as the expr for `do`. `deriv` is a subset. 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].

⟨*dxo-expression*⟩ ::=
| ⟨*numeral-or-variable*⟩
| '(+ ' ⟨*dxo-expression*⟩ ... ')'
| '(∗ ' ⟨*dxo-expression*⟩ ... ')'
| '(^ ' ⟨*dxo-expression*⟩ ⟨*dxo-expression*⟩ ')'

⟨*numeral-or-variable*⟩ :: = ⟨*tagged-numeral*⟩ | ⟨*tagged-variable*⟩

⟨*tagged-variable*⟩ ::= '(var ' ⟨*symbol*⟩ ')'

⟨*tagged-numeral*⟩ ::= '(num ' ⟨*numeral*⟩ ')'

⟨*numeral*⟩ ::= '()' | '(0 . ' ⟨*positive-numeral*⟩ ')' | '(1 . ' ⟨*numeral*⟩ ')'

⟨*positive-numeral*⟩ ::= '(0 . ' ⟨*positive-numeral*⟩ ')' | '(1 . ' ⟨*numeral*⟩ ')'

Fig. 1. Grammar for general *dxo* expressions accepted by `simpo`, `evalo`, and `reordero`.

```
⟨polynomial-expression⟩ ::=
  |  ⟨numeral-or-variable⟩
  |  '(+ ' ⟨polynomial-expression⟩ ... ')'
  |  '(* ' ⟨polynomial-expression⟩ ... ')'
  |  '(^ ' ⟨numeral-or-variable⟩ ⟨tagged-numeral⟩ ')'
```

Fig. 2. Restricted grammar for polynomial expressions accepted by do.

Using the *dxo* relations, we can solve 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 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)))
⟹
'(
  ...
  ((num _.0) (var x)                      ;  f = c (where c is any natural number),  g = x
   ((x)) ((x 1)))                         ;  b = 0,  a= 1
  ...
  ((var x) (^ (var x) (num (0 0 1 1)))    ;  f = x,  g = x¹²
   ((x 1)) ((x 0 0 1 1)))                 ;  b = 1,  a = 12
  ...
)
```

Of the 20 outputs produced by the run expression, many had $b = 0$ so we only showed two. The first shown answer shows

$$\frac{d}{dx}[c] = 0, \text{where } c \text{ is any natural number and } \frac{d}{dx}[x] = 1, \text{where } x = 5$$

The second shown answer shows

$$\frac{d}{dx}[x] = 1, \text{where } x = 12 \text{ and } \frac{d}{dx}[x^{12}] = 12, \text{where } x = 1$$

The concise run expression solving this problem shows how *dxo* benefits from the expressiveness of relational programming.

We showed a combination of the four main *dxo* relations in solving the problem in the introduction. We will shortly demonstrate another way to combine these core relations in the definition of anydo, below.

Let's use do to differentiate the polynomial $x^3 + x^0$. Mathematically,

$$\frac{d}{dx}[x^3 + x^0] = (x^2 * 3) + 0$$

The equivalent call to do succeeds:

```
(do 'x
    '(+ (^ (var x) (num (1 1))) (^ (var x) (num ())))          ; x³ + x⁰ = expr
    '(+ (* (^ (var x) (num (0 1))) (num (1 1))) (num ())))     ; (x² * 3) + 0 = deriv
```

The derivative $(x^2 * 3) + 0$ is equivalent to $1 * x^2 * 3$, so we might expect the call

```
(do 'x
    '(+ (^ (var x) (num (1 1))) (^ (var x) (num ())))          ; x³ + x⁰ = expr
    '(* (num (1)) (^ (var x) (num (0 1))) (num (1 1))))        ; 1 * x² * 3 = deriv
```

to succeed. Unfortunately, this call fails because do requires the derivative to be in canonical form, $(x^2 * 3) + 0$ in this case. This means some mathematically correct expression and derivative pairs fail as arguments to do.

We created anydo to fix this problem. (anydo expr deriv x), like do, relates an expression with its derivative with respect to x, except anydo generalizes this to simplified, complicated, or reordered forms of expr and deriv. This relaxes the restriction on deriv being in canonical form, making running "backward" more convenient. Calling anydo with the same arguments as above succeeds:

```
(anydo '(+ (^ (var x) (num (1 1))) (^ (var x) (num ())))       ; x³ + x⁰ = expr
        '(* (num (1)) (^ (var x) (num (0 1))) (num (1 1)))     ; 1 * x² * 3 = deriv
        'x)
```

anydo is centered around a call to do 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 and dcorder, with the additional step of reordering dcorder to be dcomp.

```
(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))))
```

3 *DXO* IMPLEMENTATION WALK-THROUGH

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

---

[1]We have released the *dxo* code under an MIT licence at https://github.com/JShermanSteele/dxo .

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 making all following simplifications:

- $v + 0 = v$;
- $v * 0 = 0$;
- $v * 1 = v$;
- $v^0 = 1$ $(v \neq 0)$;
- $v^1 = v$;
- $0^v = 0$ $(v \neq 0)$;
- and $1^v = 1$;

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

```
(run* (simp) (simpo `(+ (^ (num ()) (num (1 0 1) ))        ; 0^5 + 2 * 1 = comp
                        (* (num (0 1)) (num (1)))))
                   simp))
⇒
'((num (0 1)))                                              ; 2 = simp
```

simpo has base cases of (== comp simp) for comp and simp being the same number or variable. The three non-base cases, addition, multiplication, and exponentiation, deeply recursively simplify sub-expressions by checking for those simplifyable cases.

If comp is ground, (simpo comp simp) will either succeed exactly once or fail because there is at most one way to simplify any concrete expression, and simpo has no overlapping cases when running "forwards". 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 loop forever.

An example of these behaviors is that running (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.

```
(run* (simp) (simpo '(^ (num (1)) (num (1))) simp))        ; 1^1 = comp
⇒
'((num (1)))

(run 1 (comp) (simpo comp '(^ (num (1)) (num (1)))))       ; 1^1 = simp
```

There is also a case with a ground comp and partially ground simp in which simpo will fail. If comp is 1^1, which simplifies to 1, and simp is any addition expression, which may include fresh variables, simpo will fail finitely.

```
(run* (q) (simpo `(^ (num  (1)) (num (1)))                 ; 1^1 = comp
               `(+ . ,q)))                                 ; simp is some addition expression
⇒
'()
```

3.2 do

(do x expr deriv) relates a polynomial expression expr with its derivative deriv, with respect to x. For example, running do with expr and deriv fresh finds integral/derivative pairs:

```
(run 24 (expr deriv) (do 'x expr deriv))
⇒
'(...
```

```
((^ (var x) (num (0 1)))                                ;  x² = expr
 (* (^ (var x) (num (1))) (num (0 1))))                 ;  x¹ * 2 = deriv
 ...
((^ (var x) (num (1 _.0 . _.1)))                        ;  xᵃ  where a is odd = expr
 (*                                                     ;  xᵃ⁻¹ * a = deriv
   (^ (var x) (num (0 _.0 . _.1)))
   (num (1 _.0 . _.1))))
 ...
((* (^ (var x) (num ())) (^ (var x) (num ())))          ;  x⁰ * x⁰ = expr
 (+                                                     ;  0 * x⁰ + x⁰ * 0 = deriv
   (* (num ()) (^ (var x) (num ())))
   (* (^ (var x) (num ())) (num ()))))))
```

The best way to understand do is as a case analysis on expr, which is either a variable, a number, an exponentiation, an addition, or a multiplication.

Since the derivative of a sum is the sum of the derivatives of its parts, when expr and deriv are sums, the sub-expressions of expr and deriv are pair-wise related using do. Since the sum can have any positive number of terms, a helper relation, map-do-o, relates each pair 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 sub-expressions being multiplied. To improve the efficiency this process, we wrote a helper relation, multruleo, that relates the list of sub-expressions being multiplied and the multiplication's derivative. If the list has length greater than one, multruleo separates the first term $e1$ from the rest, $e2 * e3 * \ldots$. Applying the multiplication rule to $e1 * e2 * e3 * \ldots$ yields $\frac{d}{dx}[e1] * e2 * e3 * \ldots + e1 * \frac{d}{dx}[e2 * e3 * \ldots]$, which is recursive with multruleo because $\frac{d}{dx}[e2 * e3 * \ldots]$ is the related derivative argument to multruleo with $e2 * e3 * \ldots$ as the first argument. This is conde the clause in do for multiplication.

```
((fresh (l e)
   (== expr `(* . ,l))
   (letrec ((multruleo
              (lambda (l dd)
                (fresh (e e* d 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))))
```

We could recur through the multiplication by recurring with every shorter multiplication as an argument to do, but our approach is simpler because it does not exit multruleo while reccuring through expr's multiplication.

If expr is an exponentiation, 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, $\frac{d}{dx}[x^0] = 0$, $\frac{d}{dx}[n^m] = 0$, and $\frac{d}{dx}[n] = 0$ where n and m are any numbers so long as they both are not 0. Finally, the derivative of just x is one.

Since do orders deriv a certain way (for example, $x^2 * 6$ instead of $6 * x^2$), some integratable derivs will fail. This is why do should be used with reordero. For example,

```
(run 1 (expr)
  (do 'x expr '(* (^ (var x) (num (1))) (num (0 1))))))  ; x^1 * 2 = deriv
⇒
'((^ (var x) (num (0 1))))                                ; x^2 = expr
```

produces an answer, but switching deriv's multiplication order diverges:

```
(run 1 (expr)
  (do 'x expr '(* (num (0 1)) (^ (var x) (num (1))))))  ; 2 * x^1 = deriv
```

Similarly to simpo, do succeeds exactly once or fails running "forwards" when expr and x are ground. With expr fresh, do can succeed or can loop infinitely just like simpo.

3.3 evalo

evalo evaluates, and is useful for solving equations.(evalo env expr value) relates an expression expr with its value value, where each variable in expr is associated with a natural number by the environment env. For example we can look for expressions that evaluate to 8 in an environment that binds z to 2:

```
(run 200 (expr) (evalo `((z . (0 1))) expr '(0 0 0 1)))      ; z= 2, value= 8
⇒
`(...
  (* (var z) (num (0 0 1)))                                  ; z * 4 = expr
  ...
  (^ (num (0 1)) (num (1 1)))                                ; 2^3 = expr
  ...
  (+ (var z) (num ()) (num (0 1 1))))                        ; z + 0 + 6 = expr
```

evalo deeply recurs through expr, evaluating tagged little-endian binary lists into miniKanren numbers. For evalo's base cases when expr is a variable or number, value is the variable's value from env or the miniKanren number respectively.If expr is an addition, multiplication, or an exponentiation, then evalo relates the first term with its value evc, the rest with its value evrest, and adds, multiplies, or exponentiates evc and evrest to obtain value.The evalo code for addition does this:

```
((== `(+ ,c . ,rest) expr)
 (conde
   ((== '() rest) (evalo env c value))
   ((=/= '() rest)
   (evalo env c evc)
   (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!

```
(run 1 (env)
  (fresh (xv yv zv z2v)
    (== `((x . ,xv) (y . ,yv) (z . ,zv) (z-squared . ,z2v)) env)
    (evalo env `(+ (^ (var x) (num (0 1))) (^ (var y) (num (0 1)))) z2v)
```

```
    (*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:

```
(run 1 (env)
  (fresh (xv yv zv z2v)
    (poso xv)
    (poso yv)
    (poso zv)
    (== `((x . ,xv) (y . ,yv) (z . ,zv) (z-squared . ,z2v)) env)
    (evalo env `(+ (^ (var x) (num (0 1))) (^ (var y) (num (0 1)))) z2v)
    (*o zv zv z2v)))
⇒
'(((x 1 1)                                                           ;  x= 3
   (y 0 0 1)                                                         ;  y= 4
   (z 1 0 1)                                                         ;  z= 5
   (z-squared 1 0 0 1 1)))                                          ;  z² = 25
```

which is a 3-4-5 right triangle.

If either env or expr is fresh, evalo can loop forever, trying more and more complicated envs or exprs. If both env or expr are ground, then evalo will terminate since evalo runs simply forwards in this case.

3.4 reordero

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

```
(run* (e2) (reordero `(+ (num (1)) (* (num (0 1)) (num (1 1)))) e2))    ; 1 + 2 * 3 = e1
⇒
'((+ (num (1)) (* (num (0 1)) (num (1 1))))                              ; 1 + 2 * 3 = e2
  (+ (* (num (0 1)) (num (1 1))) (num (1)))                              ; 2 * 3 + 1 = e2
  (+ (num (1)) (* (num (1 1)) (num (0 1))))                              ; 1 + 3 * 2 = e2
  (+ (* (num (1 1)) (num (0 1))) (num (1))))                             ; 3 * 2 + 1 = e2
```

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

```
((fresh (o e1* e2*)
   (== `(,o . ,e1*) e1)
   (== `(,o . ,e2*) e2)
   (typeo o '+or*)
   (reorderitemso e1* e2*)))
```

where e1* and e2* are permutations of each other. reorderitemso checks that e1* and e2* have the same length, and then calls reorderinnero on them. We created reorderitemso to improve speed and divergence behavior of reordero by requiring e1* and e2* have the same length before considering the relations in reorderinnero. reorderinnero relates permuted lists at any depth. To deeply reorder, reorderinnero calls reordero on the corresponding sub-expressions for e1* and e2*.

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

reordero greatly reduces infinite loops because reorderitemso 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.

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 relations 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[Dick 1991] relationally. We have done some preliminary work making a relation to replace do that automatic differentiates forwards and backwards.

5 RELATED WORK

Expresso [Schünemann 2017] is a computer algebra system written in Clojure using the miniKanren-inspired core.logic library. expresso's original intent was to be relational, but the author made it non-relational to include more advanced features.[Schünemann 2020] 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 $((x + x)/x) * (y + y - y) \Rightarrow 2 * y$. It can make simplifications like $x + x \Rightarrow 2 * x$ 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.

REFERENCES

William E. Byrd. 2009. *Relational Programming in miniKanren: Techniques, Applications, and Implementations*. Ph.D. Dissertation. Indiana University.

William E. Byrd, Michael Ballantyne, Gregory Rosenblatt, and Matthew Might. 2017. A Unified Approach to Solving Seven Programming Problems (Functional Pearl). *Proceedings of the ACM on Programming Languages* 1, ICFP (2017), 8.

William E. Byrd and Daniel P. Friedman. 2006. From Variadic Functions to Variadic Relations: A miniKanren Perspective. In *Proceedings of the 2006 Scheme and Functional Programming Workshop (University of Chicago Technical Report TR-2006-06)*, Robby Findler (Ed.). 105–117.

William E. Byrd, Eric Holk, and Daniel P. Friedman. 2012. miniKanren, Live and Untagged: Quine Generation via Relational Interpreters (Programming Pearl). In *Proceedings of the 2012 Annual Workshop on Scheme and Functional Programming* (Copenhagen, Denmark) *(Scheme '12)*. ACM, New York, NY, USA, 8–29. https://doi.org/10.1145/2661103.2661105

A. J. J. Dick. 1991. An Introduction to Knuth–Bendix Completion. *Comput. J.* 34, 1 (01 1991), 2–15. https://doi.org/10.1093/comjnl/34.1.2 arXiv:https://academic.oup.com/comjnl/article-pdf/34/1/2/1181698/340002.pdf

Daniel P. Friedman, William E. Byrd, Oleg Kiselyov, and Jason Hemann. 2018. *The Reasoned Schemer* (2nd ed.). The MIT Press, Cambridge, MA, USA.

Shruti Jasoria. 2019. Reduce-Algebraic-Expressions. https://github.com/SJasoria/Reduce-Algebraic-Expressions. Accessed: 2020-07-14.

Oleg Kiselyov, William E. Byrd, Daniel P. Friedman, and Chung-chieh Shan. 2008. Pure, declarative, and constructive arithmetic relations (declarative pearl). In *Proceedings of the 9th International Symposium on Functional and Logic Programming (LNCS, Vol. 4989)*, Jacques Garrigue and Manuel Hermenegildo (Eds.). Springer, 64–80.

Maik Schünemann. 2017. expresso. https://github.com/clojure-numerics/expresso. Accessed: 2020-07-14.

Maik Schünemann. 2020. private correspondence.

A FULL IMPLEMENTATION OF *DXO*

```
(require "faster-miniKanren/mk.rkt")
(require "faster-miniKanren/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)))

;exponent: (^ a b)=c
(define expo
  (lambda (a b c)
    (fresh (bm1 rec)
      (conde
        ((== (build-num 0) b) (== (build-num 1) c))
        ((=/= (build-num 0) b)
         (pluso bm1 (build-num 1) b)
         (expo a bm1 rec)
         (*o 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) (=/= y x) (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 unbuld-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*))]
      [`(^ ,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)
         (=/= a item)
         (notmembero item rest))))))

;removes item from l
(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)))))))


"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)
              (=/= 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)))))))


"DERIVATIVE";----------------------------------------------------------

;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))
          (samelengtho e* d*)
          (map-do-o x e* d*)))
```

```
((fresh ()
   (== expr `(^ (var ,x) (num ,(build-num 0))))
   (== deriv ZERO)))

((fresh (l e)
   (== expr `(* . ,l))
   (letrec ((multruleo
              (lambda (l dd)
                (fresh (e e* d 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))
     ((== ZERO int1)(poso int2)))))

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

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

```
;maps do relation
(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";-----------------------------------------------------------

;evaluater
(define evalo
  (lambda (env expr value)
    (fresh (m x c 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)
            (pluso evc evrest 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 reorderitemso
  (lambda (e1* e2*)
    (fresh ()
      (samelengtho 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))))))
```

```
"ANYDO";_____

(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))))
```