Relational Floating-Point Arithmetic

LUCAS SANDRE*, University of Toronto Mississauga, Canada
MALAIKA ZAIDI*, University of Toronto Mississauga, Canada
LISA ZHANG, University of Toronto Mississauga, Canada

We present a minimal relational floating-point arithmetic that supports comparison, addition/subtraction, and multiplication/division in a binary, normalized floating-point system with rounding by chopping. The system can be used to solve simple arithmetic problems, quadratic equations, and can reason relationally about overflow. We also show that it's runtime generally grows exponentially with respect to precision, and that multiplication runtime grows exponentially with respect to the number of 1's in the mantissa.

CCS Concepts: • **Software and its engineering** → **Functional languages**; **Constraint and logic languages**.

Additional Key Words and Phrases: miniKanren, relational programming, floating-point arithmetic

1 INTRODUCTION

Although miniKanren is typically used to solve discrete problems, there has been some interest from the community in exploring mathematical applications to relational programming. Still, mathematical libraries built on miniKanren [4] typically use natural number arithmetic [3] rather than continuous arithmetic. The attendees of the second miniKanren workshop in 2020 suggested that it should be possible to implement relational floating-point arithmetic in miniKanren. This paper presents an attempt to create a minimal, relational floating-point arithmetic system.

To that end, we present a floating-point arithmetic system that supports comparison relations, addition/subtraction, and multiplication/division. These relations support binary, normalized floating-point systems with rounding by chopping. Our implementation can be found at: https://github.com/malaikazaidi/relational-float.

We show that it is possible for miniKanren to reason relationally about floating-point numbers, and that doing so can help illustrate the quirkiness of floating-point numbers. Queries where we expect a single result can sometimes return multiple results due to floating-point rounding.

For example, consider solving the equation $x + 1 = x^2$ for the Golden ratios. Recall that the two real solutions to this equation are $x_1 = \varphi \approx 1.618$ and $x_2 = \bar{\varphi} \approx -0.618$. Depending on the precision we use in the floating-point representation, we obtain different solutions:

```
; precision = 3
> (define one '(0 (1 1 1 1 1 1 1) (0 0 1))
> (run 2 (x) (fresh (y) (fp-pluso one x y) (fp-multo x x y)))
'((0 (1 1 1 1 1 1 1 1) (0 0 1))  ; positive infinity
  (1 (0 1 1 1 1 1 1)   (1 0 1))  ; -0.625
```

*Both authors contributed equally to the paper

Authors' addresses: Lucas Sandre, University of Toronto Mississauga, Mississauga, ON, Canada, lucas.sandre@mail.utoronto.ca; Malaika Zaidi, University of Toronto Mississauga, Mississauga, ON, Canada, malaika.zaidi@mail.utoronto.ca; Lisa Zhang, University of Toronto Mississauga, Mississauga, ON, Canada, lczhang@cs.toronto.edu.

miniKanren.org/workshop/2021/8-ART2

```
; precision = 4
> (define one '(0 (1 1 1 1 1 1 1) (0 0 0 1))
> (run 2 (x) (fresh (y) (fp-pluso one x y) (fp-multo x x y)))
'((0 (1 1 1 1 1 1 1 1) (0 0 0 1))  ; positive infinity
  (0 (1 1 1 1 1 1 1)   (1 0 1 1))) ; +1.625
```

The result illustrates the effect of rounding and finite representation in a floating-point system. When we use the precision $p = 3$, the first two answers we find are $+\infty$ and -0.625. In contrast, when we use the precision $p = 4$, the first two answers are $+\infty$ and $+1.625$. A positive solution does not appear using precision $p = 3$ because there is no positive floating-point number x which has the computation $x + 1$ round to x^2. However, when the precision is set to 4, we find a positive floating-point number x that has the computation $x + 1$ round to x^2.

We revisit this example in Section 5.4. But first, we briefly describe floating-point arithmetic in Section 2 and related work in Section 3. We present our implementation in Section 4, and show results and examples in Section 5. We discuss the limitations of the implementation in Section 6.

2 FLOATING-POINT ARITHMETIC

Floating-point numbers are ubiquitous in computing as a way solve problems involving continuous values using discrete hardware. We provide a concise review here, but refer interested readers to Goldberg [2] for a more detailed overview.

A floating-point number system $F(\beta, p, L, U)$ is characterized by a base $\beta \in \mathbb{N}$, precision $p \in \mathbb{N}$, and lower and upper exponential range $L, U \in \mathbb{Z}$. In this system, a real number x is represented using discrete values $d_0 \ldots d_{p-1}$ with $d_i \in \{0, 1, \ldots \beta - 1\}$, and $E \in \{L, L+1, \ldots, U-1, U\}$ with

$$x \approx \pm(d_0 + \frac{d_1}{\beta} + \frac{d_2}{\beta^2} + \ldots \frac{d_{p-1}}{\beta^{p-1}})\beta^E$$
$$\triangleq d_0.d_1 \ldots d_{p-1} \times \beta^E \qquad \text{(alternative notation)}$$

The digits $d_0 d_1 \ldots d_{p-1}$ is called the *mantissa*, and E is called the *exponent*. Most systems require that floating-point numbers be *normalized*, meaning that the leading digit must have $d_0 \neq 0$ unless $x = 0$. Requiring normalization assures that each floating-point number has a unique representation. Most systems also make an exception to allow *denormalized numbers*, which are floating-point numbers with $E = L$ and $d_0 = 0$, and which cannot be represented with an exponent larger than L.

As an example, the IEEE-754 single precision floating-point system is the normalized system $F(\beta = 2, p = 24, L = -126, U = 127)$. Each floating-point number in this system can be stored in 32 bits. One bit is used to store the sign: with `0` being positive and `1` being negative. Eight bits are used to store the *shifted exponent*, with `00000000` representing the smallest possible exponent L and `11111111` representing U the largest possible exponent. Instead of storing a 24-bit mantissa $d_0, d_1, ..., d_{23}$, the leading digit d_0 is dropped turning it into a 23-bit *fraction* $d_1, d_2, ..., d_{23}$. The value of d_0 is one unless the number represented is a denormalized number or zero. This system also has representations for special values like positive infinity, negative infinity, and `NaN`.

Floating-point addition and multiplication can be computed using long addition and multiplication. Figure 1 shows an example addition and multiplications of two floating-point numbers in the system $F(\beta = 2, p = 4, L = -100, U = 100)$. In practise, hardware designers use more efficient algorithms for common floating-point operations [1].

Rounding rules for floating-point numbers can be complex, and IEEE systems typically use a “round to nearest even" strategy. However, other strategies could be used, like rounding by chopping.

3 RELATED WORK

Kiselyov et al. [3] described a pure, declarative and constructive arithmetic relations for natural numbers, referred to by the miniKanren community as “Oleg” numbers. These numbers are stored as a little endian list of binary values. Moreover, Kiselyov et al. [3] implements comparison, addition, multiplication, and exponentiation relations, and show these relations to be decidable.

		1.	0	1	1		×	2^5
+			1	1	1	0	×	2^4
	1.	0	0	1	0	0	×	2^6
≈	1.	0	0	1			×	2^6

(a) Addition by aligning the mantissa digits, performing column-wise addition with carry, computing the exponent, and rounding to $p = 4$ digits.

				1.	0	1	1		×	2^5
×					1	1	1	0	×	2^4
				1	0	1	1			
			1	0	1	1				
+		1	0	1	1					
	1.	0	0	1	1	0	1	0	×	2^{10}
≈	1.	0	0	1					×	2^{10}

(b) Multiplication by aligning the mantissa digits, multiplying by each nonzero digit, adding the partial results, computing the exponent, and rounding to $p = 4$ digits.

Fig. 1. Example addition and multiplication computation in $F(\beta = 2, p = 4, L = -100, U = 100)$, with rounding by chopping.

We use these "Oleg" numbers as a building block of our floating-point numbers, and use the relations `pluso` and `*o` in our arithmetic and the relation `<o` in our comparisons. We use the version of miniKanren from the faster-miniKanren repository.

We are not aware of other implementations of floating-point arithmetic in miniKanren.

4 FLOATING-POINT ARITHMETIC IN MINIKANREN

Our implementation uses the normalized floating-point system $F(\beta = 2, p, L = -126, U = 127)$, where the precision $2 \leq p \leq 16$ can be set by the user. For simplicity, we use rounding by chopping and do not allow denormalized numbers. Unless specified otherwise, we will assume $p = 16$ in all our examples.

We represent a floating-point number using a list containing its sign bit, exponent, and mantissa. The sign bit is either 0 (for a positive number) and 1 (for a negative number). The exponent is an Oleg number [3] representing the shifted exponent. The mantissa is a list of p digits, with each digit being either 0 or 1, stored in little endian format.

For example, if we choose $p = 16$, the number -8.5 in base two scientific notation is 1.0001×2^3 and is represented in our system as the following list:

```
'(1                                  ; negative=1, positive=0
  (0 1 0 0 0 0 0 1)                  ; exponent=3, shifted exponent=130
  (0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1))  ; mantissa, which includes the leading 1
```

Notice that unlike in an IEEE system, we store the leading digit $d_0 = 1$ of the mantissa. We find that the additional storage more than offsets the complexity in the algorithm.

Our system represents these special values in the following manner:

- Positive zero: `'(0 () (0 0 0 0  0 0 0 0  0 0 0 0  0 0 0 0))`
- Negative zero: `'(1 () (0 0 0 0  0 0 0 0  0 0 0 0  0 0 0 0))`
- Positive infinity: `'(0 (1 1 1 1  1 1 1 1) (0 0 0 0  0 0 0 0  0 0 0 0  0 0 0 0))`
- Negative infinity: `'(1 (1 1 1 1  1 1 1 1) (0 0 0 0  0 0 0 0  0 0 0 0  0 0 0 0))`

We exclude `NaN` values which represent undefined arithmetic operations. Queries that would otherwise produce a `NaN` value would simply not return any answers.

4.1 Comparison Operators

We implement three binary comparison relations `fp-=`, `fp-<`, and `fp-<=`. These relations compare the signs, exponents, and mantissas of the two floating-point numbers.

The equality relation `(fp-= f1 f2)` succeeds when the signs, exponents, and mantissas of `f1` and `f2` are equal. We also allow positive and negative zero to be equal.

The relation (fp-< f1 f2) can succeed in several ways, each corresponding to a conde branch:

- f1 = 0, and f2 > 0
- f1 < 0, and f2 = 0
- f1 < 0, and f2 > 0
- f1, f2 > 0, with the exponent of f2 being larger
- f1, f2 < 0, with the exponent of f2 being smaller
- f1, f2 > 0, with their exponents equal, and the mantissa of f2 being larger
- f1, f2 < 0, with their exponents equal, and the mantissa of f2 being smaller

The relation (fp-<= f1 f2) succeeds if either (fp-= f1 f2) or (fp-< f1 f2).

Example 4.1. Consider the following query to find floating-point numbers larger than or equal to one.

```
> (define one '(0 (1 1 1 1 1 1 1) (0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1)))
> (run 5 (x) (fp-<= one x)
'((0 (1 1 1 1 1 1 1) (0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1)) ; 1 is >= 1
  (0                                      ; any positive number with a valid
   (_.0 _.1 _.2 _.3 _.4 _.5 _.6 _.7)      ; 8 digits in the exponent
   (_.8 _.9 _.10 _.11 _.12 _.13 _.14 _.15 _.16 _.17 _.18 _.19 _.20 _.21 _.22 1))
  (0 (1 1 1 1 1 1 1) (1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1))  ; 1.00003051758
  (0 (1 1 1 1 1 1 1) (0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1))  ; 1.00006103516
  (0 (1 1 1 1 1 1 1) (1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1))) ; 1.00009155273
```

The first solution corresponds to the case where the equality relation holds. The second solution corresponds to the case where the exponent is strictly larger than that of one. Then the remaining three solutions correspond to the case where they share the same exponent as one but mantissa being larger than the mantissa of one.

Notice that answers 3-5 and onwards could be more concisely expressed with the following answer:

```
'(0
  (1 1 1 1 1 1 1)
  (_.0 _.1 _.2 _.3 _.4 _.5 _.6 _.7_.8 _.9 _.10 _.11 _.12 _.13 _.14 1))
```

However, we use the helper relation (<o n1 n2), and this relation grounds the digits when n1 and n2 are represented using lists of the same length. Since mantissas are always the same length p, we get an enumeration of the possible answers.

4.2 Addition (and Subtraction)

The addition relation (fp-pluso f1 f2 r) takes three floating-point numbers: the two addends and their sum. Subtraction is achieved by running the addition relation "backwards".

Our implementation of fp-pluso uses a helper relation fp-samesignaddero that assumes that the signs of f1, f2 and r are all the same, and that f2 has an exponent at least as large as that of f1. If this assumption does not hold, we rearrange the argument terms before calling the helper relation.[1] To determine which of f1 and f2 have the larger exponent, we use Kiselyov's pluso relation to compute the difference between the two exponents.

Under the above assumptions, the helper relation fp-samesignaddero performs the actual addition by (1) shifting f1's mantissa to align the corresponding bits in f2's mantissa, (2) adding f2's mantissa and f1's shifted mantissa, (3) dropping any of the least-significant bits from the sum's mantissa according to preset precision, (4) determining the sum's exponent as sometimes it is one larger than the exponent of f2, and (5) checking if overflow occurs. Each of these steps is implemented as a helper relation. For performance reasons, we reorder these five steps in the actual implementation.

```
(define (fp-samesignaddero expo1 mant1  ; the exponent & mantissa of f1
                           expo2 mant2  ; the exponent & mantissa of f2
                           expo-diff    ; expo2 - expo1
                           rexpo rmant) ; the exponent & mantissa of r
```

[1]For example, (+ (-50) 3 (-47)) would be rewritten as (+ 47 3 50).

```
    (fresh (shifted-mant1 mant-sum pre-rmant bit)
        ; (1) shift the mantissa of the smaller exponent
        (mantissa-shifto mant1 expo-diff shifted-mant1)
        ; (4) shift the exponent (sometimes we need to add 1)
        (conde ((== bit '()) (== rexpo expo2))
               ((=/= bit '()) (pluso '(1) expo2 rexpo)))
        ; (3) round result to the appropriate precision by dropping
        (drop-leastsig-bito mant-sum pre-rmant bit)
        ; (5) check if overflow occurs
        (fp-overflowo rexpo pre-rmant rmant)
        ; (2) oleg number addition of the fraction
        (pluso shifted-mant1 mant2 mant-sum)))
```

Example 4.2. To illustrate the steps involved in addition, consider adding the floating-point numbers 3.14 and 1.0. To do so, we would run the query `(run 1 (r) (fp-pluso f3.14 f1.0 r))`.

```
; the number pi
> (define f3.14 '(0 (0 0 0 0 0 0 0 1) (1 0 1 0 1 1 1 1 0 0 0 1 0 0 1 1)))
; the number 1.0
> (define f1.0  '(0 (1 1 1 1 1 1 1)   (0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1)))
```

These two numbers are both nonzero, with the same sign. The number `f1.0` has the smaller exponent, and the difference between the two exponents is one, or `'(1)` in Oleg number notation. Inside `fp-pluso`, we call `fp-samesignaddero` after swapping the order of the addends:

```
(fp-samesignaddero '(1 1 1 1 1 1 1)   '(0 0 0 0 0 0 0 0 0 0 0 0 0 0 1) ;f1.0
                   '(0 0 0 0 0 0 0 1) '(1 0 1 0 1 1 1 1 0 0 0 1 0 0 1 1) ;f3.14
                   '(1) ; the difference between expo2 and expo1
                   rexpo rmant)
```

We demonstrate the five steps of the the helper relation `fp-samesignaddero`:

```
; Step 1: align the mantissas of both addends by shifting the mantissa of the
;         smaller exponent:
(mantissa-shifto '(0 0 0 0 0 0 0 0 0 0 0 0 0 0 1) ; f1.0 mantissa
                 '(1)          ; the difference between the two exponents
                 '(0 0 0 0 0 0 0 0 0 0 0 0 0 0 1))  ; f1.0 mantissa shifted

; Step 2: with the mantissas aligned, we use the Oleg number addition
;         to obtain the sum of the two mantissas.
(pluso  '(0 0 0 0 0 0 0 0 0 0 0 0 0 0 1)      ; shifted mantissa for f1.0
        '(1 0 1 0 1 1 1 1 0 0 0 1 0 0 1 1)    ; other mantissa for f3.14
        '(1 0 1 0 1 1 1 1 0 0 0 1 0 0 0 0 1)) ; mant-sum

; Step 3: Notice how the resulting sum has one more digit than the precision,
;         and so a digit needs to be removed. Since we round by chopping,
;         we simply remove the least significant bit.
(drop-leastsig-bito  '(1 0 1 0 1 1 1 1 0 0 0 1 0 0 0 0 1) ; mant-sum from step 2
                     '(0 1 0 1 1 1 1 0 0 0 1 0 0 0 0 1)) ; pre-rmant
                     '(1)) ; last bit is dropped

; Step 4: Since a bit was dropped, the exponent of our sum should be
;         one larger than the exponent of our larger addend.
(pluso '(1)
       '(0 0 0 0 0 0 0 1)  ; exponent of f3.14
       '(1 0 0 0 0 0 0 1)) ; rexpo
```

```
; Step 5: finally, we check if there is an overflow. Since there isn't,
;         the mantissa remains unchanged.
(fp-overflowo '(1 0 0 0 0 0 0 1)                  ; rexpo
              '(0 1 0 1 1 1 1 0 0 0 1 0 0 0 0 1)  ; pre-rmant
              '(0 1 0 1 1 1 1 0 0 0 1 0 0 0 0 1)) ; same mantissa.
```

These steps gives us the resulting sum `'(0 (1 0 0 0 0 0 0 1) (0 1 0 1 1 1 1 0 0 0 1 0 0 0 0 1))`, which is roughly equal to 4.14.

4.3 Multiplication (and Division)

The multiplication relation takes three floating-point numbers: the two multiplicands and their product. As before, division is achieved by running the multiplication relation "backwards".

Our multiplication relation performs multiplication by (1) relating the signs of the three terms: the product is positive if and only if both multiplicands have the same sign, (2) computing the mantissa of the product using Oleg number multiplication, (3) keeping only the p digits of precision in the mantissa of the product, (4) Determining the products exponent: the exponent of the product is the sum of the stored exponents of the two multiplicands subtracted by the bias, however in some cases we need to add 1 depending on the length of the fraction. The bias is subtracted to account for adding the two bias-shifted exponents. (5) Checking if the product's exponent signals overflow. Again, each of these steps is implemented relationally, and for performance reasons, these steps do not appear in order in our code.

```
(define (fp-multo f1 f2 r)
    (fresh (sign1 expo1 mant1 sign2 expo2 mant2 rsign rexpo rmant)
        (fp-decompo f1 sign1 expo1 mant1) ; decomposition of f1
        (fp-decompo f2 sign2 expo2 mant2) ; decomposition of f2
        (fp-decompo r  rsign rexpo rmant) ; decomposition of r
        ;(1) relate the signs
        (xoro sign1 sign2 rsign)
        (conde
            ;... omit conde branches dealing with zeros and special values ...
            (; ... omit check that f1, f2 nonzero/finite, and r nonzero...
             (fresh (mant1mant2 pre-mantr ls-bits)
                ; (5) check for overflow
                (fp-overflowo rexpo pre-mantr rmant)
                ; (2) compute the mantissa using Oleg number *o
                (*o mant1 mant2 mant1mant2)
                ; (3) round result to the appropriate precision by dropping
                (drop-leastsig-bito mant1mant2 pre-mantr ls-bits)
                ; (4) compute the exponent of the product
                (fp-multo-compute-expoo expo1 expo2 ls-bits rexpo))))
        (expo-lengtho expo1)
        (expo-lengtho expo2)
        (expo-lengtho rexpo)))
```

Example 4.3. To illustrate the steps involved in multiplication, consider multiplying the floating-point numbers -3.14 and 2.0. To do so, we would run the query `(run 1 (r) (fp-multo -3.14 2.0 r))`

```
; the number -pi
> (define fnpi '(1 (0 0 0 0 0 0 0 1) (1 0 1 0 1 1 1 1 0 0 0 1 0 0 1 1)))
; the number 2.0
> (define f2.0  '(0 (0 0 0 0 0 0 0 1)  (0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1)))
```

We demonstrate the steps of the relation `fp-multo`:

```
; Step 1: The two multiplicands are nonzero with opposite signs
;         so the relation xoro will succeed when rsign is 1
(xoro 1  ; since -3.14 is negative
      0  ; since  2.0 is  positive
      1) ; the product should be negative

; Step 2: Compute the product of the mantissa using *o
(*o '(1 0 1 0 1 1 1 1 0 0 0 1 0 0 1 1) ; mantissa of -3.14
    '(0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1) ; mantissa of  2.0
    '(0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 1 1 1 1 0 0 0 1 0 0 1 1)) ; product

; Step 3: Keep only p=16 digits of the mantissa
(drop-leastsig-bito
  '(0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 1 1 1 1 0 0 0 1 0 0 1 1) ; from above
  '(1 0 1 0 1 1 1 1 0 0 0 1 0 0 1 1) ; keep these most significant bits
  '(0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 )) ; drop these least significant bits

; Step 4: Compute the sum of the exponents of the multiplicands
;         as a step to computing the exponent of the product
(pluso '(0 0 0 0 0 0 0 1)   ; exponent of -3.14
       '(0 0 0 0 0 0 0 1)   ; exponent of 2.0
       '(0 0 0 0 0 0 0 0 1) ; sum of the two exponents
(pluso BIAS                 ; 127
       '(0 0 0 0 0 0 0 0 1) ; sum of the two exponents, from above
       '(0 0 0 0 0 0 0 1))  ; bias-shifted sum of the two exponents

; Step 5: Check if we need to +1 to the exponent of the product.
;         Since the length of the dropped bits is 15, the answer is no.
(mantissa-length-minus1o '(0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 )) ; dropped bits
(== '(0 0 0 0 0 0 0 1) ; bias-shifted sum of the two exponents
    '(0 0 0 0 0 0 0 1)) ; the exponent of the product.
```

As a final step, we check if there is an overflow. There isn't, so we get the product `'(1 (1 0 0 0 0 0 0 1) (1 0 1 0 1 1 1 1 0 0 0 1 0 0 1 1))`, which is roughly equal to -6.28.

5 RESULTS AND EXAMPLES

We present some results and examples using the relational floating-point arithmetic to perform computations. All tests are run on a desktop tower using Racket version 7.8 and faster miniKanren with the 4 Core Intel i5-7400 @ 3.0Ghz CPU, 24 GB DDR4 @ 2400MT/s RAM, and ASROCK B250M Pro4-IB motherboard.

5.1 Comparison Relations

The comparison relations `fp-=`, `fp-<`, and `fp-=<` are all relatively fast. For example, the queries below all terminate in less than 20ms.

```
> (run (10) (x) (fp-<= x one)) ; 8ms
> (run (10) (x y) (fp-<= x y)) ; 8ms
> (run (10) (y) (fp-<= one y)) ; 17ms
```

5.2 Addition and Subtraction

Table 1 shows the runtime for queries using `fp-pluso`. As expected, queries that involve a single logic variable all finish relatively quickly. The fastest queries are those with two ground addends of the same sign, since such queries require no search. When addends have different signs, or when one of the addends is a logic variable, search is required to find the correct position of the arguments in the call to `fp-samesignaddero`.

Table 1. Addition/Subtraction Queries.

Query	N	Time (ms)
$1 + \pi = \_$	1	4
$\pi + 1 = \_$	1	2
$100 + 100\pi = \_$	1	1
$100\pi + 100 = \_$	1	1
$-100 + 100\pi = \_$	1	25
$100\pi + (-100) = \_$	1	8
$100 + (-100\pi) = \_$	1	8
$-100\pi + 100 = \_$	1	7
$1 + \_ = \pi$	1	8
$\_ + 1 = \pi$	1	4
$-1 + 1.05 = \_$	1	13
$1 + (-1.05) = \_$	1	11
$-1 + (-0.05) = \_$	1	1
$0 + \_ = 3$	1	1
$3 + 0 = \_$	1	1
$100 + 0.01 = \_$	1	1
$0.01 + 100 = \_$	1	1
$100 + (-0.01) = \_$	1	22
$(-0.01) + 100 = \_$	1	19
$100 + \_ = 0.01$	1	20
$\_ + 100 = 0.01$	1	17
$100 + \_ = -0.01$	1	2
$\_ + 100 = -0.01$	1	2
$\pi + \_ = \_$	1	1
$\pi + \_ = \_$	15	12119
$\_ + \pi = \_$	1	1
$\_ + \pi = \_$	15	25355
$\_ + \_ = \pi$	1	1
$\_ + \_ = \pi$	15	65818

Table 2. Multiplication/Division Queries.

Query	N	Time (ms)
$1 * \pi = \_$	1	3
$\pi * 1 = \_$	1	2
$125 * 20\pi = \_$	1	2181
$20\pi * 125 = \_$	1	2074
$0.01 * 0.01 = \_$	1	>900000
$0.05 * 0.05 = \_$	1	43661
$0.001 * 0.001 = \_$	1	1920
$20 * \_ = 40$	1	2122
$\_ * 20 = 40$	1	2433
$2 * \_ = 40$	1	10
$\_ * 2 = 40$	1	15
$100 * 0.01 = \_$	1	15
$0.01 * 100 = \_$	1	12
$1000 * 0.001 = \_$	1	228
$0.001 * 1000 = \_$	1	1288
$100 * \_ = 0.01$	1	>900000
$\_ * 100 = 0.01$	1	21164
$0.01 * \_ = 100$	1	>900000
$\_ * 0.01 = 100$	1	>900000
$1000 * \_ = 0.001$	1	>900000
$\_ * 1000 = 0.001$	1	68728
$0.001 * \_ = 1000$	1	54476
$\_ * 0.001 = 1000$	1	47654
$\pi * \_ = \_$	1	1
$\pi * \_ = \_$	15	8503
$\_ * \pi = \_$	1	1
$\_ * \pi = \_$	15	10403
$\_ * \_ = \pi$	1	5215
$\_ * \_ = \pi$	15	5679

Query runtime is not affected by the magnitude of the floating-point numbers, so queries like $100 + 100\pi$ are no slower than $1 + \pi$. This is because the exponent values are already shifted to a fairly large value, and the shifted exponents are capped at 2^8.

When queries have two free logic variables, it is 2x faster if we provide the first addend rather than the second addend, even though doing so does not change the semantics of the query.

5.3 Multiplication and Division

Table 2 shows the runtime for queries using `fp-multo`. As before, running forward is generally fast as there is not much searching that takes place. There is rough symmetry between the two multiplicands just like in `fp-pluso`, we have that grounding the first multiplicand is faster than grounding the second one (when the other multiplicand is a logic variable).

One interesting observation is while the magnitude of the multiplicands do not influence running time, the "size" of the mantissa does. Oleg number multiplication (`*o`) is slower when the natural number multiplicands are larger. Thus, when there are many 1's in the mantissa, floating-point multiplication is slow. Figure 2 shows the relationship between the mantissa of a number x versus the computation time to retrieve one result for x^2. Notice the runtime grows exponentially as we add more 1's to the mantissa. Moreover, runtime is slower when the 1's are concentrated in the least-significant digit.

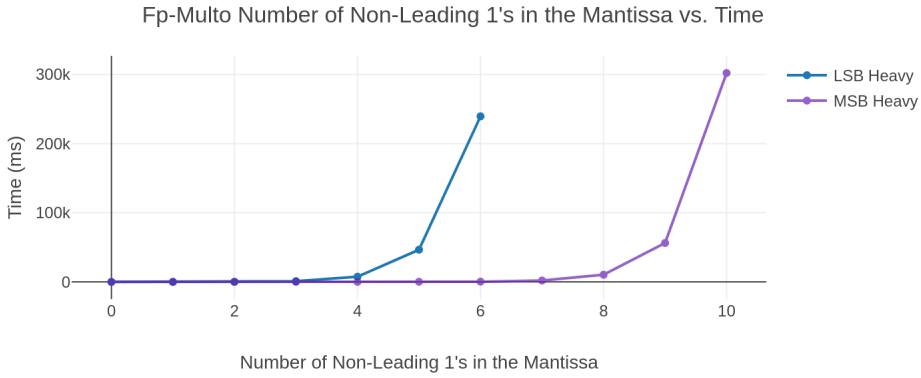

Fig. 2. The relationship between the number of non-leading 1's in a floating-point number x, and the time it takes to compute x^2. "LSB Heavy" means that the 1's are concentrated near the least significant bits, "MSB Heavy" means that the 1's are concentrated near the most significant bits.

5.4 Quadratic Equation; Impact of Precision

We explore how our floating-point system works with interesting equations such as the quadratic equation to see if our system is able to solve queries that call both `fp-pluso` and `fp-multo` together. In particular, the Golden Ratio φ is the solution to the quadratic equation $\varphi^2 - \varphi - 1 = 0$, or $\varphi + 1 = \varphi^2$, or $\varphi(\varphi - 1) = 1$. We'll represent this quadratic equation using two sets of somewhat equivalent queries:

```
(run 1 (x) (fresh (y) (fp-pluso 1 x y) (fp-multo x x y))) ; phi + 1 = phi^2
(run 1 (x) (fresh (y) (fp-pluso 1 y x) (fp-multo x y 1))) ; phi(phi - 1) = 1
```

Since the `fp-pluso` relation is typically faster than `fp-multo`, we always put the `fp-pluso` goal first. Also, we are implicitly removing the solution $\varphi = \infty$ to the equation $\varphi + 1 = \varphi^2$ since this solution doesn't illustrate the true time difference between the two queries for non-trivial solutions. With precision=8, we get

```
> precision = 8
> (run 1 (x) (fresh (y) (fp-pluso 1 x y) (fp-multo x x y)))
((0 (1 1 1 1 1 1 1) (1 1 1 1 0 0 1 1)))
cpu time: 105366 real time: 105154 gc time: 24569

> (run 1 (x) (fresh (y) (fp-pluso 1 y x) (fp-multo x y 1)))
((0 (1 1 1 1 1 1 1) (1 1 1 1 0 0 1 1)))
cpu time: 1092 real time: 1090 gc time: 51
```

Table 3 shows the results along with the precision used and the time it took to run the query to get a result. As expected, queries with a higher precision takes exponentially longer to return a result. This is because the length of the mantissa impacts how relations such as `pluso` work as it becomes slower when

Table 3. Quadratic Equation

Query	Precision	CPU Time	Query	Precision	CPU Time
$\varphi + 1 = \varphi^2$	4	57	$\varphi(\varphi - 1) = 1$	4	7
$\varphi + 1 = \varphi^2$	5	263	$\varphi(\varphi - 1) = 1$	5	13
$\varphi + 1 = \varphi^2$	6	1767	$\varphi(\varphi - 1) = 1$	6	29
$\varphi + 1 = \varphi^2$	7	13431	$\varphi(\varphi - 1) = 1$	7	N/A*
$\varphi + 1 = \varphi^2$	8	105366	$\varphi(\varphi - 1) = 1$	8	1092

the length of the mantissa is longer. Moreover, notice that when $p = 7$, we fail to find an answer for `(fresh (y) (fp-pluso 1 y x) (fp-multo x y 1))`, since all values of x and y close to the desired solution have xy and $x - y$ round to floating-point values that are not exactly 1.

5.5 Reasoning about Overflow

One advantage of using a relational floating-point system is that the system can reason about overflow. For example, consider the task of looking for a floating-point number that, when multiplied by 1000, results in an overflow. This is a more challenging task to solve using floating-point functions rather than relations. This task translates to the following query using `fp-multo`:

```
; floating-point representation of 1000:
> (define fp1000 '(0 (0 0 0 1 0 0 0 1) (0 0 0 0 0 0 0 0 0 1 0 1 1 1 1 1)))
; special value to denote +infinity
> (define fpoverflow '(0 (1 1 1 1 1 1 1 1) (0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1)))
> (run 3 (x) (fp-multo fp1000 x fpoverflow))
'((0 (1 1 1 1 1 1 1 1) (0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1)) ; infinity
  (0 (0 1 1 0 1 1 1 1) (0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1)) ; 2^119
  (0 (0 1 1 0 1 1 1 1) (1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1))); 2^119 + 2^104
```

Beyond the obvious solution of setting `x` to positive infinity, we can find other large floating-point numbers that cause the desired overflow.

5.6 Reasoning about Small Numbers

As another example, consider the query below that searches for values that, when added to fp3.4, result in the same value fp3.4. In exact arithmetic, we would expect 0 to be the only result, and our results does contain both positive and negative zeros. However, because of floating-point rounding, there are other small values that will work.

```
> (define fp3.4 '(0 (0 0 0 0 0 0 0 1) (1 0 0 1 1 0 0 1 1 0 0 1 1 0 1 1))
> (run 5 (x) (fp-pluso fp3.4 x fp3.4))
'((0 ()    (0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)) ; +0
  (1 ()    (0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)) ; -0
  (0 (1)   (_.0 _.1 _.2 _.3 _.4 _.5 _.6 _.7 _.8 _.9 _.10 _.11 _.12 _.13 _.14 1))
  (1 (1)   (_.0 _.1 _.2 _.3 _.4 _.5 _.6 _.7 _.8 _.9 _.10 _.11 _.12 _.13 _.14 1))
  (0 (0 1) (_.0 _.1 _.2 _.3 _.4 _.5 _.6 _.7 _.8 _.9 _.10 _.11 _.12 _.13 _.14 1)))
```

The results include both positive and negative floating-point numbers with small exponents, starting at -126 and going upwards.

5.7 Order of Operations

As in a non-relational floating-point arithmetic, the order of operations do matter. Laws like associativity do not hold in a floating-point arithmetic. To demonstrate, suppose we have the following three numbers in the floating-point number system $F(\beta = 2, p = 4, L = -126, U = 127)$:

```
> (define a '(0 (1 1 1 1  1 1 1) (0 0 0 1))); 1 x 2^0
> (define b '(0 (1 1 0 1  1 1 1) (0 0 0 1))); 1 x 2^(-4)
> (define c '(0 (1 1 0 1  1 1 1) (0 0 0 1))); 1 x 2^(-4)
```

When using floating-point addition, we have $(a + b) + c \neq a + (b + c)$:

```
> (run 1 (x) (fresh (y) (fp-pluso a b y) (fp-pluso y c x))) ; (a + b) + c
'((0 (1 1 1 1  1 1 1) (0 0 0 1)))                           ; = a
> (run 1 (x) (fresh (y) (fp-pluso b c y) (fp-pluso a y x))) ; a + (b + c)
'((0 (1 1 1 1  1 1 1) (1 0 0 1)))                           ; > a
```

The results of calculating `(a + b) + c` and `a + (b + c)` differs due to the order of operations. Notice that the order of operations is a different concept as the order of the conjuncts. For example, if we swap the conjuncts in the `(a + b) + c` operation, we still obtain the same result:

```
> (run 1 (x) (fresh (y) (fp-pluso y c x) (fp-pluso a b y))) ; (a + b) + c
'((0 (1 1 1 1  1 1 1) (0 0 0 1)))                           ; = a
```

5.8 Reasoning about Significant Digits

During subtraction, the relational floating-point arithmetic differentiates between significant digits and the zeros that are added. In other words when subtracting two numbers of the same sign and similar magnitude, when the mantissa of the difference has fewer significant digits than p. In a floating-point addition/subtraction function, the "missing" digits would be set to zero. However, since we expect `fp-pluso` to behave relationally, `fp-pluso` will obtain multiple answers:

```
> (run (x) (fp-pluso fp-1 fp1.0001 x)) ; -1 + (1.0001) = 0.0001
'((0
  (1 0 0 0 1 1 1)
  (_.0 _.1 _.2 _.3 _.4 _.5 _.6 _.7 _.8 _.9 _.10 _.11 _.12 _.13 1 1)))
```

Notice how the mantissa hold multiple logic variables, where a non-relational system would set these digits to zero.

6 LIMITATIONS

As seen in the results, our relational floating-point arithmetic relations are much slower than the more optimized non-relational systems. We did some optimization of the relations by rearranging the `conde` branches, but serious optimization is beyond the scope of this work. Due to the nature of miniKanren, we don't think such optimization is fruitful.

Although we demonstrate using search to perform computations like solving the quadratic equation, in practise this is not a good idea. Search is going to be much slower than doing math, and using more stable computational methods.

7 CONCLUSION

We build a minimal relational floating-point arithmetic in miniKanren, supporting floating-point numbers in the system $F(\beta = 2, p, L = -126, U = 127)$ for $2 \leq p \leq 16$. We chose simple assumptions like rounding by chopping, and not allowing denormalized numbers. Still, the relational arithmetic is able to do interesting things like reason about overflow, be explicit about cancellation, and reason about multiple results.

ACKNOWLEDGMENTS

We thank the attendees of the second miniKanren workshop in 2020 for suggesting the idea of building a relational floating-point system. We also thank Gregory Rosenblatt for showing us how to debug with `gdisplay`, and Aislinn Sandre for helpful feedback.

REFERENCES

[1] B Sreenivasa Ganesh, J E.N Abhilash, and G Rajesh Kumar. 2012. Design and Implementation of Floating Point Multiplier for Better Timing Performance. http://ijarcet.org/wp-content/uploads/IJARCET-VOL-1-ISSUE-7-130-136.pdf

[2] David Goldberg. 1991. What every computer scientist should know about floating-point arithmetic. *ACM computing surveys (CSUR)* 23, 1 (1991), 5–48.

[3] Oleg Kiselyov, William E Byrd, Daniel P Friedman, and Chung-chieh Shan. 2008. Pure, declarative, and constructive arithmetic relations (declarative pearl). In *International Symposium on Functional and Logic Programming*. Springer, 64–80.

[4] Julie Steele and William Byrd. 2020. dxo: A System for Relational Algebra and Differentiation. *arXiv preprint arXiv:2008.03441* (2020).