Encoding Numeric Computations and Infusing Heuristic Knowledge Using Integrity Constraints in stableKanren

XIANGYU GUO, Arizona State University, USA
AJAY BANSAL, Arizona State University, USA

This paper presents examples of using integrity constraints in stableKanren to encode numeric computations for problem solving. Then, we use one of the examples to introduce multiple ways to infuse heuristic knowledge and reduce solving time. stableKanren is an extension of miniKanren that supports normal logic programs under stable model semantics. stableKanren further supports numeric computation by constructing a constraint store for integrity constraints. There are three ways to extend a relational programming language with numeric computations: relational number representation, grounding numbers to symbols, and constraint store construction. We demonstrate that the numeric computations in stableKanren have a straightforward numerical representation compared to relational number representations. More importantly, stableKanren balances symbolic and numeric computation in relational programming by avoiding the grounding of all numbers to symbols. Lastly, it also has simpler syntax compared to other constraint store construction approaches. stableKanren supports combinatorial search problem solving under a declarative generate and test paradigm. Such a paradigm generates all possible combinations of solutions to the problem, then applies a set of constraints to prune out the unwanted solutions. We demonstrate that different approaches to writing programs or queries affect the solver's performance in the SEND+MORE=MONEY puzzle. The performance gradually improves as more heuristic knowledge is infused through the programs or queries. Additionally, we show how to use an external function to achieve a hybrid solution.

CCS Concepts: • **Theory of computation → Constraint and logic programming**; • **Software and its engineering** → *Constraints*; *Runtime environments*; • **Computing methodologies → Logic programming and answer set programming**.

Additional Key Words and Phrases: stableKanren, integrity constraints, numeric computations, heuristic, logic programming

1 Introduction

We present two examples of writing numeric computations in a relational programming language, stableKanren. We then use the SEND+MORE=MONEY puzzle to introduce multiple ways of infusing heuristic knowledge to guide the internal search process in stableKanren.

The foundation of relational programming is symbols, which are related through resolution and unification, resulting in a bidirectional relation rather than a unidirectional function. Friedman et al. built miniKanren to capture resolution and unification, the essence of Prolog, and show a natural way to extend functional programming to relational programming [2]. The core miniKanren implementation introduces only a few operators to users: == for *unification*, *fresh* for *existential quantification*, *conde* for *disjunction*, and a *run* interface for query. The *conjunction* is captured by the sequential evaluation process naturally, so there is no operator for conjunction. However, the CPU is unidirectional, computing numbers as a function that takes inputs and produces an output. Therefore, adding numeric computation to a relational programming language is not straightforward.

There are three approaches to add numeric computation. The first approach is to create bit-wise relations or a relational CPU and represent numbers as reversed bits, which is not easy to read and cannot utilize the capabilities of the numerical CPU. The second approach is to convert all numbers into symbols through grounding, which sacrifices top-down resolution and leaves a significant memory footprint. The third one is to construct a constraint store, where the numeric expressions obtain values along with the resolution, and evaluate the expressions once they have sufficient

values. The constraint store approach creates a boundary between symbols and numbers, thereby balancing the number representation and memory usage.

Under the constraint store approach, there are two types of constraints that can be introduced to relational programming languages: the *variable constraints* and the *integrity constraints* (Definition 2.5). The variable constraints allow numeric constraints on relational variables; these constraints are applied to the variable during resolution and are evaluated once all variables have been unified with symbols. The integrity constraints control the outcome of a relational goal; these constraints are imposed on the goal and the relation between goals, and they are evaluated once sufficient goals are successfully proved by resolution. A goal is successfully proved through resolution, which also means the goal's variables are unified with symbols. Both variable and integrity constraints are able to represent whether a variable is unified or not unified with a symbol; hence, the relational programming language that has these constraints can encode the solutions to the combinatorial search problems like nqueens, graph coloring, hamiltonian cycles, etc [1, 5].

To add integrity constraints, the relational programming language also needs to have proper semantics for negation in the normal program (Definition 2.4). Guo et al. extend miniKanren under stable model semantics [4] to support negations in stableKanren [6]. stableKanren also tracks integrity constraints through Constraints as Verifiers and Emitters CaVE (Definition 2.12) [5]. stableKanren introduces three more operators: *noto* for *negation*, *defineo* for goal function definition, and *constrainto* for controlling the goal function outcome. stableKanren allows combinatorial search problems to be solved in the declarative generate-and-test paradigm.

In this paper, we demonstrate how to utilize integrity constraints in stableKanren to encode numeric computations. Furthermore, we introduce multiple ways of infusing heuristic knowledge into stableKanren, so that the internal search—the control part of the declarative algorithm—can be guided. We review the logic behind stableKanren and introduce the preliminaries in Section 2. In Section 3, we compare and contrast different approaches of adding numeric computations to relational programming languages. Then, we show two examples in Section 4 using the verifiers (Definition 2.12) in integrity constraints to perform numeric computations. We use query variables reordering (Section 5.1), adding smaller constraints (Section 5.2), and critical emitters (Section 5.3) to reduce the running time to less than 20 seconds on the SEND+MORE=MONEY puzzle. Lastly, we present an external oracle verifier that leverages human heuristic knowledge to control the outcome of the *assign* operation, thereby achieving hybrid solving.

2 Preliminaries

This section reviews several key logic programming definitions underlying stableKanren, including definite programs (Definition 2.2), normal programs (Definition 2.4), and integrity constraints (Definition 2.5). The even negation cycles in the normal programs generate combinations, and the odd negation cycles prune out combinations [10]. Integrity constraints are a special form of normal program clauses, in which they contain odd negation cycles. We present a logic program that generates multiple solutions and utilizes integrity constraints to eliminate unwanted solutions. Lastly, we use Constraints as Verifiers and Emitters (CaVE, Definition 2.12) to encode integrity constraints in stableKanren.

2.1 Normal Programs, Integrity Constraints, and Generating Combinations

Lloyd defines *definite program*, *normal program*, and *integrity constraints* as follows [11].

Definition 2.1 (definite program clause). A *definite program clause* is a clause of the form,

$$A \leftarrow B_1, \cdots, B_n$$

where $A, B_1, \ldots, B_n$ are atoms[1].

A definite program clause contains precisely one atom A in its consequent. A is called the *head* and $B_1, \ldots, B_n$ is called the *body* of the program clause.

Definition 2.2 (definite program). A *definite program* is a finite set of definite program clauses.

Based on the definition of the definite program clause, we have the definition of *normal program clause* and *normal program*.

Definition 2.3 (normal program clause). A *normal program clause* is a clause of the form,

$$A \leftarrow B_1, \cdots, B_n, not\ B_{n+1}, \cdots, not\ B_m$$

For a normal program clause, the body of a program clause is a conjunction of literals instead of atoms; $B_1, \cdots, B_n$ are *positive literals* and $not\ B_{n+1}, \cdots, not\ B_m$ are *negative literals*[2].

Definition 2.4 (normal program). A *normal program* is a finite set of normal program clauses.

The *integrity constraint* is defined as a headless normal program clause.

Definition 2.5 (integrity constraint). An *integrity constraint* is a clause of the form,

$$\bot \leftarrow B_1, \cdots, B_n, not\ B_{n+1}, \cdots, not\ B_m$$

where $B_1, \ldots, B_m$ are atoms.

For an integrity constraint, the head of a clause is $\bot$ (false) instead of an atom. So, the integrity constraints are headless. The constraint is violated only if all body literals are evaluated as $\top$ (true). When the clause's body is evaluated as $\top$, we get a formula that says true implies false.

$$\bot \leftarrow \top$$

This formula is impossible to prove true. So, the constraint is violated. The constraint can be satisfied if any literal in the body is evaluated to be false. Therefore, the integrity constraints mean that not all literals in the body can be proven true.

2.2 Loops in Normal Programs, Generating and Eliminating Combinations

According to Van Gelder et al., there are sets of atoms named *unfounded sets* in a normal program that can help us categorize the normal programs [12].

Definition 2.6 (unfounded set). Given a normal program, the atoms inside the unfounded set are only cyclically supporting each other, forming a loop.

Replacing the atoms with literals in the unfounded set (loop) definition, we have informal definitions of *positive loop* and *negative loop*.

Definition 2.7 (positive loop). A positive loop is a loop that contains no negative literals.

Definition 2.8 (negative loop). A negative loop is a loop that has at least one negative literal involved.

Using the number of negative literals, we can further define *odd negative loop* and *even negative loop* from Definition 2.8.

Definition 2.9 (odd negative loop). An odd negative loop is a loop that involves an odd number of negative literals.

[1]Atom is evaluated to be true or false.

[2]A positive literal is just an atom, a negative literal is the negation of an atom.

Definition 2.10 (even negative loop). An even negative loop is a loop that involves an even number of negative literals.

An odd negative loop introduces a contradiction to the normal program. For example, the headless integrity constraint can be translated into an odd negative loop clause that creates a contradiction, as shown in Definition 2.11.

Definition 2.11 (alternative integrity constraint). An *alternative integrity constraint* is a clause of the form,

$$\boldsymbol{fail} \leftarrow B_1, \cdots, B_n, not\ B_{n+1}, \cdots, not\ B_m, \boldsymbol{not\ fail}$$

This translation is equivalent to the original integrity constraint in Definition 2.5, and no other transformations are needed. The *fail* and *not fail* pair introduces a contradiction when all other body literals B_i are evaluated as true.

Lin and Zhao point out that even negative loops in the normal program generate combinations, and the integrity constraints eliminate combinations [10]. For example, given a propositional normal program (a program without variables) written in Prolog syntax [7, 8] as follows.

```
pick :- not free.
free :- not pick.
```

The program has one even negative loop. According to stable model semantics [4], either *pick* or *free* can be true, but they cannot be true or false at the same time. The truth value of *pick* indicates one binary choice. The propositional example can be extended to a predicate version as follows.

```
pick(X) :- not free(X).
free(X) :- not pick(X).
```

The predicate program adds a variable, X, to increase expressiveness. However, the variable is *free variable*; a program that contains free variables cannot be evaluated. Therefore, variables X need to be grounded in values. The domain of the variable provided by a set of *num* predicates is as follows.

```
num(1). num(2). num(3).
pick(X) :- num(X), not free(X).
free(X) :- num(X), not pick(X).
```

The above program grounds the variable X to three values; therefore, it indicates three binary choices. In total, it generates $2^3 = 8$ possible combinations.

A more complex example using two variables and an integrity constraint in Listing 1,

Listing 1. A logic program produces multiple models

```
num(1). num(2). num(3).
pick(X, Y) :- num(X), num(Y), not free(X, Y).
free(X, Y) :- num(X), num(Y), not pick(X, Y).
:- pick(X, Y), pick(U, V), X = U, Y != V.
```

The program describes a 3×3 board with a total of nine positions. The first line defines three numbers. Each position has either a pick or a non-pick choice. Therefore, it generates all possible combinations, resulting in $2^9 = 512$ models, under stable model semantics. The integrity constraint on the last line indicates that we do not pick the position in the same row. For example, *pick(1, 1)* and *pick(1, 2)* violate the integrity constraint. Therefore, the total number of models reduces from 512 to 64.

2.3 Constraints as Verifiers and Emitters (CaVE) in stableKanren

Constraints as Verifiers and Emitters CaVE (CaVE) categorizes the atoms $B_1 \cdots B_m$ in an integrity constraint (Definition 2.5) into two types: an *emitter* and a *verifier*.

Definition 2.12 (emitter and verifier). An integrity constraint can be written as *emitters* and *verifiers* in a clause of the form,

$$\bot \leftarrow E_1, \cdots, E_n, not\ E_{n+1}, \cdots, not\ E_m, \tag{1}$$

$$V_1, \cdots, V_i, not\ V_{i+1}, \cdots, not\ V_j \tag{2}$$

where $E_1, \ldots, E_m$, $V_1, \ldots, V_j$ are atoms. Specifically, $E_1, \ldots, E_m$ are emitters. An *emitter* is the head atom of a normal program clause that emits values. $V_1, \ldots, V_j$ are verifiers. A *verifier* is a boolean expression that receives and verifies values from the emitter.

When resolution successfully unifies variables with values in the normal program clause, the values are emitted to the verifiers. All verifiers within the same integrity constraint are combined into a single constraint handler.

Definition 2.13 (constraint handler). A constraint handler is a concatenation (*and* operator) of verifiers and $\top$ (true).

Therefore, if all verifiers are true, the constraint handler returns true, indicating that a constraint has been violated. For example, the head atoms of the program in Listing 1 are *num*, *pick*, and *free*. The integrity constraint in Listing 1 has two emitters, *pick(X, Y)* and *pick(U, V)*; two verifiers, $X = U$ and $Y\ != V$ forming a constraint handler $(X = U) \wedge (Y! = V) \wedge \top$.

Definition 2.14 (CaVE). CaVE breaks an integrity constraint into emitters and verifiers. The emitters are inserted as checkpoints in the resolution to emit values to verifiers. Once verifiers collect sufficient values, the constraint handler verifies the constraint; the verification result controls the resolution.

During the resolution, the values emitted from *pick* are captured by the constraint handler. For example, the *pick* emits *(1, 1)*, the partial constraint handler is $(1 = U) \wedge (1! = V) \wedge \top$. Later, *pick* emits *(1, 2)*, the previous parial constraint handler becomes $(1 = 1) \wedge (1! = 2) \wedge \top$, triggered a constraint violation.

stableKanren uses CaVE to implement *constrainto* [5]. So, the integrity constraints break into two parts in *constrainto*. As a result, stableKanren supports solving combinatorial search problems using a declarative generate-and-test paradigm. For example, the logic program in Listing 1 has the stableKanren equivalent as follows.

```
(defineo (num x) (conde [(== 1 x)] [(== 2 x)] [(== 3 x)]))
(defineo (pick x y) (num x) (num y) (noto (free x y)))
(defineo (free x y) (num x) (num y) (noto (pick x y)))
(constrainto [(pick x y) (pick u v)] [(= x u) (not (= y v))])
```

The program has two parts: problem instances and a declarative algorithm. The first line is the problem instances *nums*. The declarative algorithm also has two parts: generating and pruning. The *pick* and *free* form an even negation cycles that generate all combinations. And the *constrainto* prunes out the unwanted combinations.

3 Related Work

This section reviews related work on adding numeric computation to relational programming languages. The challenge in adding numeric computation to a relational programming language is that symbolic unification only unifies symbols and lacks arithmetic operations. There are three ways to overcome the challenge.

3.1 Relational Number Representation

Friedman et al. propose a set of bit-wise relations to simulate a relational CPU [2]. All numbers are represented as a list of reversed binary bits. For example, the number nineteen has the representation of (1 1 0 0 1), the first element in the list is the lowest binary bit. The unidirectional function *xor* turns into a bidirectional relation *xoro* as follows.

```
(define (xoro x y r)
  (conde [(== 0 x) (== 0 y) (== 0 r)] [(== 0 x) (== 1 y) (== 1 r)]
         [(== 1 x) (== 0 y) (== 1 r)] [(== 1 x) (== 1 y) (== 0 r)]))
```

The *xoro* relation takes three parameters instead of two. This relation is maintained as long as all three parameters are successfully unified with their corresponding values. Similar relations are also defined for *ando*, then using *xoro* and *ando* to define *half-addero* and *full-addero*.

3.2 Converting to Symbols

Kaminski et al. design a bottom-up approach to solve logic programs [9]. The bottom-up approach consists of two stages: grounding and problem-solving. All numeric operations are converted to symbols. Then the symbols are assigned a truth value to solve the Boolean equation. For example, the integrity constraint we used to remove the same row in Listing 1 produces 9 symbols from pick(1,1) to pick(1,3) and 18 propositional constraints after grounding.

```
:-pick(1,1),pick(1,2). :-pick(1,1),pick(1,3). :-pick(2,1),pick(2,2).
                        ...... 12 more ......
:-pick(2,3),pick(2,2). :-pick(3,3),pick(3,1). :-pick(3,3),pick(3,2).
```

Each symbol will be assigned a Boolean value to check whether the propositional constraints are violated or not.

3.3 Constraint Store Construction

The last approach is to construct a constraint store to track all numeric expressions during resolution. Once the numeric expressions obtain sufficient values from unification, they can be evaluated. Alvis et al. implement cKanren where the numeric constraints are imposed on the variable by using the new operators *infd*, *plusfd*, and *=/=fd* [1]. An example cKanren constraint is as follows.

```
(define (diago qi qj d rng)
  (fresh (qi+d qj+d)
    (infd qi+d qj+d rng)
    (plusfd qi d qi+d)
    (=/=fd qi+d qj)
    (plusfd qj d qj+d)
    (=/=fd qj+d qi)))
```

It ensures no two queens are placed on the diagonal for the nqueens problem.

Guo et al. build an integrity constraint into stableKanren, where the numeric constraints are applied on the verifiers [5]. An example stableKanren constraint for the same nqueens problem is as follows.

```
(constrainto [(queen x y) (queen u v)]
             [(= (abs (- x u)) (abs (- y v)))
              (not (= x u)) (not (= y v))])
```

There are two emitters *queen* emit two positions (x, y), (u, v) to the verifiers, and the verifiers perform the numeric computation to check the constraint.

4 Numeric Computations in stableKanren

The declarative generate-and-test paradigm reduces all problems to combinatorial search problems. All numbers and their possible relations are modeled as a combination search as well. The program generates all combinations of the relations between numbers, then prunes out invalid combinations. The verifiers (Definition 2.12) are the interface between symbols and numbers. As we mentioned at the end of Section 2.3, the paradigm has two parts: problem instances and a declarative algorithm. In the following examples, the problem instances are always a sequence of numbers, depending on the actual problem size, and are omitted. We focus solely on the declarative algorithm.

4.1 Arithmetic Relations

The arithmetic relations are bidirectional counterparts of the corresponding unidirectional functions. For example, the + function in Scheme has a relation *pluso* defined as follows.

```
(defineo (pluso x y z) (nums x) (nums y) (nums z) (noto (n_pluso x y z)))
(defineo (n_pluso x y z) (nums x) (nums y) (nums z) (noto (pluso x y z)))
(constrainto [(pluso x y z)] [(not (= (+ x y) z))])
(constrainto [(n_pluso x y z)] [(= (+ x y) z)])
```

The first two lines generate all possible combinations of the three numbers (x, y, z). Some combinations do not meet the *pluso* requirement, which requires $x + y = z$. The last two lines prune out these invalid combinations.

The *minuso* can be defined using *pluso* by switching the parameter position.

```
(defineo (minuso z x y) (pluso x y z))
```

Similarly, *multipo* and *divido* are defined. The generating part is simple, so we only show the pruning part.

```
(constrainto [(multipo x y z)] [(not (= (* x y) z))])
(constrainto [(n_multipo x y z)] [(= (* x y) z)])

(constrainto [(divido x y q r)] [(or (= y 0) (>= r y)
                                     (not (= (+ (* y q) r) x)))])
(constrainto [(n_divido x y q r)] [(and (not (= y 0)) (< r y)
                                        (= (+ (* y q) r) x))])
```

The *multipo* only requires $x * y = z$, but the *divido* is slightly more complex, requiring the divisor not to be equal to 0, the remainder to be less than the divisor, in addition to $y * q + r = x$.

4.2 SEND+MORE=MONEY

The SEND + MORE = MONEY puzzle requires choosing different single-digit numbers (0-9) for each letter (SENDMORY) to satisfy the equation. An additional restriction is that there are no leading zeros in the numbers, so S and M can not be 0. The problem instances include *letters* and *values*, the *letters* will have 8 symbols (SENDMORY), and the *values* will have 10 symbols (0-9). The algorithm is shown in Listing 2.

Listing 2. SEND MORE MONEY puzzle in stableKanren with integrity constraints

```
1 (defineo (assign l v) (letters l) (values v) (noto (n_assign l v)))
2 (defineo (n_assign l v) (letters l) (values v) (noto (assign l v)))
3
4 (constrainto [(assign l1 v1) (assign l2 v2)]
5              [(eq? l1 l2) (not (= v1 v2))])
6 (constrainto [(assign l1 v1) (assign l2 v2)]
7              [(not (eq? l1 l2)) (= v1 v2)])
8
9 (defineo (assigned l) (fresh (v) (letters l) (values v) (assign l v)))
```

```
10 (constrainto [(letters l1) (noto (assigned l2))] [(eq? l1 l2)])
11
12 (constrainto [(assign 's s) (assign 'e e) (assign 'n n) (assign 'd d)
13                (assign 'm m) (assign 'o o) (assign 'r r) (assign 'y y)]
14               [(not (= (+ (* s 1000) (* e 100) (* n 10) (* d 1)
15                           (* m 1000) (* o 100) (* r 10) (* e 1))
16            (+ (* m 10000) (* o 1000) (* n 100) (* e 10) (* y 1))))])
```

The first two lines generate all possible combinations of the letters and values. The variables l and v in the *assign* get values through *letters* and *values* as we have explained in Section 2.2. In lines 4 to 7, two constraints ensure that no letter takes more than one value and no value is assigned to more than one letter. Lines 9 and 10 ensure that each letter is assigned a unique value. Lines 12 to 16 use the puzzle's equation as a constraint to eliminate invalid value assignments. The symbols are emitted by the emitters *assign* in lines 12 and 13; the verifiers receive and convert them to numbers so that the equation can be evaluated to verify the assignments in lines 14 to 16.

A query uses the *run* interface to produce an answer to the puzzle as follows.

```
> (run 1 (q) (fresh (s e n d m o r y)
                (assign 's s) (assign 'e e) (assign 'n n) (assign 'd d)
                (assign 'm m) (assign 'o o) (assign 'r r) (assign 'y y)
                (== q `(,s ,e ,n ,d ,m ,o ,r ,y))))
```

This query took 4,670 seconds to find one answer on a 2020 iMac with a 3.6 GHz Intel Core i9-10910 processor and 128GB of memory, which is not ideal for such a small puzzle.

5 Infusing Heuristic Knowledge

In this section, we use the SEND + MORE = MONEY puzzle as an example to illustrate different ways of incorporating heuristics into stableKanren. To satisfy the equation, the puzzle requires choosing different single-digit numbers (0-9) for each letter (SENDMORY). An additional restriction is that there are no leading zeros in the numbers, so S and M can not equal 0. We start from a basic, slow version of encoding in stableKanren in Section 4.2. We gradually apply techniques such as query variable reordering, adding more constraints, and critical emitters. The solving time was drastically reduced to less than 20 seconds. Lastly, we demonstrate a hybrid solution that utilizes an external oracle verifier to further reduce the time.

5.1 Variables Reordering

Reordering the query variables is one way to control stableKanren's internal search. We measure the time spent on solving the puzzle with different starting letters. The ordering of the remaining seven letters is not specified in the query and is decided internally by the solver. Therefore, we have eight queries for each starting letter as follows.

```
> (run 1 (q) (assign 's q))
```

The running time in seconds is listed in Table 1, along with the corresponding letter and value.

Letter	s	e	n	d	m	o	r	y
Value	9	5	6	7	1	0	8	2
Time	14242.16	5730.59	8721.58	9778.90	1052.88	1080.75	11074.77	3717.25

Table 1. Running time of different starting letters in the query

As we can see, the running time ranges from 1080.75 (letter 'o') to 14242.16 (letter 's') seconds. More importantly, the running time increases with the value ordering (0-9) in the *values* goal

function. The letter 'o' is assigned with 0, so it prunes out the remaining search space (1-9) if the letter is placed at the first position in the query. In contrast, the letter 's' is assigned with 9, so it has to explore the entire search space if the letter is the first in the query. Therefore, if we reorder the query variables in the query based on the values they got assigned from 0 to 9. The query variables are ordered as OMYENDRS; the query only takes around 3 seconds to find one answer. However, if the query uses *run** to perform an exhaustive search, the running time exceeds 5000 seconds. The query variables sequence (OMYENDRS) was obtained after we knew the answer, but it shows the impact of the different ordering of query variables.

5.2 Smaller Constraints

Replacing the big constraint with smaller constraints is another way to control stableKanren's internal search. The big constraint in Listing 2 requires all *assign* relations to produce a value before verifying them. In the big constraint, the equation multiplies the value of each letter by the weight of each digit it represents. Therefore, it requires that all letters have a value. The equation can be represented as constraints on each digit. The equation breaks into addition on each digit and a carry value between digits. For example, the carry value on the least significant digit is 0, and the two digits are D and E. We have $(D + E)\ mod\ 10 = Y$. And each digit creates a carry value to the next digit; the least significant bit creates $\lfloor (D + E) \div 10 \rfloor$ to the next digit. We break the big constraint into five smaller constraints as follows.

```
(constrainto [(assign 'd d) (assign 'e e) (assign 'y y)]
             [(not (= y (mod (+ d e) 10)))])

(constrainto [(assign 'd d) (assign 'e e) (assign 'n n) (assign 'r r)]
             [(not (= e (mod (+ n r (floor (/ (+ d e) 10))) 10)))])

(constrainto [(assign 'd d) (assign 'n n) (assign 'r r)
              (assign 'e e) (assign 'o o)]
             [(not (= n (mod (+ o e
                                (floor (/ (+ n r
                                (floor (/ (+ d e) 10))) 10))) 10)))])

(constrainto [(assign 'n n) (assign 'd d) (assign 'r r) (assign 'e e)
              (assign 'o o) (assign 's s) (assign 'm m)]
             [(not (= o (mod (+ m s
                                (floor (/ (+ e o
                                (floor (/ (+ n r
                                (floor (/ (+ d e) 10))) 10))) 10))) 10)))])

(constrainto [(assign 'n n) (assign 'd d) (assign 'r r) (assign 'e e)
              (assign 'o o) (assign 's s) (assign 'm m)]
             [(not (= m (floor (/ (+ s m
                            (floor (/ (+ e o
                            (floor (/ (+ n r
                            (floor (/ (+ d e) 10))) 10))) 10))) 10)))])
```

Each constraint corresponds to a digit in the SEND+MORE=MONEY equation, from the lowest significant digit to the highest significant digit. The carry values pass through the digits as the computation moves from the lowest to the highest digit. So the number of emitters in each constraint gradually grows from 3, 4, 5 to 7. We measure the time performance on solving the puzzle with different starting letters using the same queries in Section 5.1. The running time in seconds is listed in Table 2, along with the corresponding letter and value.

Letter	s	e	n	d	m	o	r	y
Value	9	5	6	7	1	0	8	2
Time	2708.52	230.41	332.39	354.59	44.26	51.32	118.26	74.22

Table 2. Running time of different starting letters in the query after adding more constraints

Compared to Table 1, the performance improved 5 to 90 times after adding more constraints. The smaller constraints have fewer emitters; hence, the constraint can be verified earlier once sufficient values are collected. The query using the sequence OMYENDRS, introduced in Section 5.1, takes only 0.46 seconds to find one answer, and 311.48 seconds if the query uses *run** to perform an exhaustive search.

5.3 Critical Emitters

In Section 5.1, the query variables sequence was created by knowing the answers, but it showed that the ordering of the query variables affects the internal search space. With additional constraints introduced in Section 5.2, we can refine the query variable sequence by identifying critical emitters. An emitter is critical if future emitters depend on the outcome of the current one. In the SEND+MORE=MONEY example, the least significant digits are more critical than the most significant digits, since the carry values are generated by the lower digits and passed through to the higher digits. So, the sequence from the lowest digits to the highest digits will be YDE, ENR, NEO, OSM, and M. Combining them and removing the duplicates, the query variables sequence is YDENROSM. An exhaustive query using *run** under the sequence YDENROSM only takes 15.69 seconds to produce the answer.

5.4 External Oracle Verifier

Solving from the lowest digits to the highest digits is not the easiest path in the puzzle. It is easier to find a partial answer starting from the highest digits. From $S + M + C_0 = MO$, the biggest possible outcome to the sum of two single digits S and M is 18; even with a potential carry C_0 from the lower digits, the sum does not exceed 19. Therefore, M must be 1. Now the equation is $S + 1 + C_0 = 1O$, C_0 is either 0 or 1 from the lower digits, so $S \geq 8$ to produce $1O$. Hence, O can either be 0 or 1. Since $M = 1$, and no letter is taking the same value, O must be 0, the equation turns into $S + 1 + C_0 = 10$. If $C_0 = 1$ from the lower digits, $S = 8$; otherwise $S = 9$. In the middle digits $E + 0 + C_1 = C_0N$ to produce $C_0 = 1$, E needs to be 9, and the carry from lower digits $C_1 = 1$. So, N needs to be 0, which conflicts with $O = 0$. The middle digits $E + 0 + C_1 = C_0N$ cannot produce $C_0 = 1$, C_0 must be 0, so $S = 9$. We can create an oracle verifier function that contains the above human reasoning, but we simplified it to storing the partial answer.

```
(define (oracle l v)
  (or (and (eq? l 's) (= v 9))
      (and (eq? l 'm) (= v 1))
      (and (eq? l 'o) (= v 0))
      (or (eq? l 'e) (eq? l 'n) (eq? l 'd) (eq? l 'r) (eq? l 'y))))

(constrainto [(assign l v)] [(not (oracle l v))])
```

Whenever the emitter *assign* finds an assignment to a letter, the oracle verifier checks the value. This hybrid solving approach reduces the time required to query the YDENROSM sequence to 5.17 seconds. If the highest digits OSM are moved to the first, the sequence becomes OSMYDENRO. It only takes 1.89 seconds to complete an exhaustive search.

6 Conclusion and Future Work

In conclusion, this paper shows the advantages of using integrity constraints (Definition 2.5) in stableKanren to encode numeric computations. Unlike creating bitwise operators and representing relational numbers in bits, the numeric computations in stableKanren are easy to understand and utilize the numeric capabilities of the CPU. Unlike grounding all numbers to symbols and removing the top-down resolution, the numeric computations in stableKanren remain top-down and only track the necessary numbers, constraints, and expressions during resolution. Unlike constraining variables and controlling the algorithm imperatively, the numeric computations in stableKanren use integrity constraints to transform symbols to numbers in verifiers and declaratively control the outcome of the goals.

All examples we showed are combinatorial search problems in stableKanren. The declarative algorithm only provides the logic of the solution, without providing any control. The solver controls the solving process. The verifiers (Definition 2.12) are the interface between symbolic and numeric computations. Currently, stableKanren has no heuristic; it only uses brute force. The solving speed depends on how the constraints are written. This paper also demonstrates how to integrate heuristic knowledge into stableKanren by reordering query variables, adding constraints, and utilizing critical emitters. Moreover, we illustrate using an external oracle verifier to achieve hybrid solving.

There are several areas for future improvement. The verifier only checks the outcome but provides no guidance on the resolution. Adding a generic heuristic function could improve the performance for some problems. The SQL engine generates and optimizes a query plan before executing the actual query. The stableKanren solver can have a built-in optimizer to reorder the query variables, identify critical emitters, and analyze constraints, as we did in Section 5. For example, the critical emitter analysis can be performed by topological sorting. The Conflict Driven Nogood Learning (CDNL) algorithm is the state-of-the-art algorithm for bottom-up solving [3]. The stablekanren solver can have a top-down version of CDNL to add generic heuristic capability. More specifically, when the symbols are numbers, constraint logic programming (CLP) techniques can be used for improving solving speed. The SEND+MORE=MONEY puzzle can be solved by cKanren in under a second. The stableKanren solver can have built-in CLP techniques and use them to prune the resolution search space. We believe that there is no unified heuristic that works for all NP-hard problems. Therefore, the stableKanren should not only provide internal, built-in heuristics but also allow external domain-specific heuristics for problem-solving. Moreover, the size of the problem instances varies; every change requires redefining instances in the program. So, a high-order relation that can generate instances based on the given number n can reduce the burden. Lastly, the emitter only emits bound values, making it cannot create an application like a relational interpreter. Hence, allowing the emitter to emit unbound variables and the verifier to leave a placeholder to verify later will be helpful.

References

[1] Claire E Alvis, Jeremiah J Willcock, Kyle M Carter, William E Byrd, and Daniel P Friedman. 2011. cKanren: miniKanren with Constraints. *Scheme Workshop* (2011). http://scheme2011.ucombinator.org/papers/Alvis2011.pdf

[2] Daniel P. Friedman, William E. Byrd, and Oleg Kiselyov. 2005. *The Reasoned Schemer*. MIT Press. doi:10.7551/mitpress/5801.001.0001

[3] Martin Gebser, Benjamin Kaufmann, André Neumann, and Torsten Schaub. 2007. Conflict-Driven Answer Set Solving. In *Proceedings of the 20th International Joint Conference on Artifical Intelligence* (Hyderabad, India) *(IJCAI'07)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 386–392. https://www.ijcai.org/Proceedings/07/Papers/060.pdf

[4] Michael Gelfond and Vladimir Lifschitz. 1988. The stable model semantics for logic programming. In *ICLP/SLP*, Vol. 88. Cambridge, MA, 1070–1080. http://www.cs.utexas.edu/users/ai-lab?gel88

[5] Xiangyu Guo and Ajay Bansal. 2024. To Be or Not To Be: Adding Integrity Constraints to stableKanren to Make a Decision. arXiv:2408.16699 [cs.PL] https://arxiv.org/abs/2408.16699

[6] Xiangyu Guo, James Smith, and Ajay Bansal. 2023. StableKanren: Integrating Stable Model Semantics with MiniKanren. In *Proceedings of the 25th International Symposium on Principles and Practice of Declarative Programming* (Lisboa, Portugal) *(PPDP '23)*. Association for Computing Machinery, New York, NY, USA, Article 5, 13 pages. doi:10.1145/3610612.3610617

[7] Intl. Organization for Standardization. 1995. *ISO/IEC 13211-1:1995: Information technology — Programming languages — Prolog — Part 1: General core.* 199 pages. https://www.iso.org/standard/21413.html

[8] Intl. Organization for Standardization. 2000. *ISO/IEC 13211-2:2000: Information technology — Programming languages — Prolog — Part 2: Modules.* https://www.iso.org/standard/20775.html

[9] Roland Kaminski and Torsten Schaub. 2023. On the foundations of grounding in answer set programming. *Theory and Practice of Logic Programming* 23, 6 (2023), 1138–1197. doi:10.1017/s1471068422000308

[10] Fangzhen Lin and Xishun Zhao. 2004. On odd and even cycles in normal logic programs. In *Proceedings of the 19th National Conference on Artifical Intelligence* (San Jose, California) *(AAAI'04)*. AAAI Press, 80–85.

[11] John W. Lloyd. 1987. *Foundations of Logic Programming, 2nd Edition.* Springer. doi:10.1007/978-3-642-83189-8

[12] Allen Van Gelder, Kenneth A. Ross, and John S. Schlipf. 1991. The Well-Founded Semantics for General Logic Programs. *J. ACM* 38, 3 (July 1991), 619–649. doi:10.1145/116825.116838