On Fair Relational Conjunction*

PETR LOZOV and DMITRY BOULYTCHEV, Saint Petersburg State University, Russia and JetBrains Research, Russia

We present a new, more symmetric evaluation strategy for conjunctions in miniKanren. Unlike the original unfair directed conjunction, our approach controls the order of conjunct execution based on the properties of structurally recursive relations. In this paper we describe operational semantics for both "classical" and "fair" conjunctions. We also compare the performance of classical and fair conjunctions on a number of examples and discuss the results of the evaluation.

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

Additional Key Words and Phrases: relational programming, miniKanren, evaluation strategies, operational semantics

1 INTRODUCTION

miniKanren [7, 8] is known for its capability to express solutions for complex problems [4, 6, 11] in the form of compact declarative specifications. This minimalistic language has various extensions [1, 3, 5, 14] designed to increase its expressiveness and declarativeness. However, *conjunction* in miniKanren has somewhat imperative flavor. The evaluation of conjunction is asymmetrical and the order of conjuncts affects not only the performance of the program but even the convergence. As a result, the order of conjuncts determines control flow in a relational program. We may call this directed behavior of conjunction *unfair*.

The problem of unfair behavior of conjunction has already been examined from several different points of view. One aspect of the unfair behavior of conjunction is to prioritize the evaluation of the independent branches which conjunction generates. In the original miniKanren a higher priority is given to an earlier branch. However, there is an approach [10] which allows one to balance the evaluation time of different branches. This makes the conjunction behavior fairer, but the order of the conjuncts still affects both performance and convergence. Also, the conjunction can be made fairer if we are capable of detecting the divergence in its conjuncts. The approach [12] detects the divergence at run time and performs switching of conjuncts. In this case, the data that was received during the evaluation of the diverging conjunct is erased. The approach turned out to be efficient in practice. However, the conservative rearrangement of the conjuncts does not use the information obtained when evaluating the conjunct before the rearrangement. There are also examples for this approach where the order of the conjuncts affects convergence.

The contribution of this paper is a more declarative approach to the evaluation of relational programs of miniKanren. This approach executes the conjuncts alternately, choosing a more optimal execution order. The fair conjunction that we propose is comparable in efficiency to the classic unfair one, but the order of the conjuncts weakly affects both efficiency and convergence. Our approach also demonstrates a more convergent behavior: we

*The reported study was funded by RFBR, project number 19-31-90053

Authors' address: Petr Lozov, lozov.peter@gmail.com; Dmitry Boulytchev, dboulytchev@math.spbu.ru, Saint Petersburg State University, Russia , JetBrains Research, Russia.

miniKanren.org/workshop/2020/8-ART3

present some examples where classical conjunction diverges for any order of conjuncts while the fair conjunction converges.

The paper is organized as follows. In Section 2 we discuss the advantages and drawbacks of the classical directed conjunction. Section 3 contains a description of miniKanren syntax as well as operational semantics based on the unfolding operation. In Section 4 we present the semantics of a naive fair conjunction, and in Section 5 we extend these semantics by controlling the conjunction order based on structural recursion. Section 6 is devoted to the evaluation and performance comparison on different semantics in the form of interpreters. The final section concludes.

2 DIRECTED CONJUNCTION

In this section we consider the classic directed conjunction in the original miniKanren and demonstrate its advantages and drawbacks on examples.

In miniKanren there is a significant difference between disjunction and conjunction evaluation strategy. The arguments of disjunction are evaluated in an *interleaving* manner, switching elementary evaluation steps between disjuncts, which provides the completeness of the search. Conjunction, on the other hand, waits for the answers from the first conjunct and then calculates the second conjunct in the context of each answer.

On the one hand, this strategy is easy to implement and allows one to explicitly specify the order of evaluation when this order is essential. On the other hand, the strategy amounts to non-commutativity: the convergence of a conjunction can depend on the order of its arguments. For example, the relation

```
let rec freeze^o x = x ≡ true ∧ freeze^o x
```

either converges in one recursion step or diverges. The conjunction (`x ≡ false ∧ freeze^o x`) converges, but the same conjunction with the reverse order of conjuncts (`freeze^o x ∧ x ≡ false`) diverges. Indeed, in the first case the first conjunct produces exactly one answer which contradicts the unification in the body of `freeze^o`. In the second case the relation `freeze^o` diverges and does not produce any answer. As a result we will never begin to evaluate the second conjunct.

```
1  let rec append^o x y xy =
2    (x ≡ [] ∧ y ≡ xy) ∨
3    fresh (e xs xys) (
4      x ≡ e : xs ∧
5      xs ≡ e : xys ∧
6      append^o xs y xys)
```

```
7  let rec revers^o x y =
8    (x ≡ [] ∧ y ≡ []) ∨
9    fresh (e xs ys) (
10     x ≡ e : xs ∧
11     revers^o xs ys ∧
12     append^o ys [e] y)
```

Fig. 1. Relational list reversing

The problem in this particular example can be alleviated by using a conventional rule for miniKanren, which says that unifications must be moved first in a cluster of conjunctions. But the rule does not work for clusters with more than one relational call. Consider, for instance, the relation `revers^o` (Fig. 1), which associates an arbitrary list with the list containing the same elements in reverse order. In this relation we can see a couple of conjuncts: `revers^o` on line 11 and `append^o` on line 12. With this particular order, the call (`revers^o [1, 2, 3] q`) converges, but in reverse order it diverges after an answer is found. Moreover, the reverse order negatively affects the performance of the answer evaluation.

At the same time the call (`revers^o q [1, 2, 3]`) for a given order of conjuncts diverges, and for the reverse order it converges. As a result a different order of conjuncts is desirable depending on their runtime values.

In the following sections we describe an approach for automatically determining a "good" order during program evaluation.

3 THE SEMANTICS OF DIRECTED CONJUNCTION

In this section we introduce a small-step operational semantics to define the behavior of the original miniKanren with directed conjunction.

Note, although at the moment there exists a certified semantics [13] of miniKanren, however, it makes a distinction between the first and the second conjuncts, which greatly complicates the task of rearranging the conjuncts in the process of program evaluation. Therefore, for this research, we have developed a new semantics that does not distinguish between conjuncts from the start.

The semantics which we propose are based on unfolding of relational calls. At each step we select a call from the current state of the program and unfold it. This process continues until no calls remain in the state. If at some step the state becomes empty then the evaluation converges. Otherwise, the evaluation diverges. Below we define these semantics formally.

$$
\begin{array}{rcll}
\mathcal{C} & = & \{C_1^{k_1}, C_2^{k_2}, \ldots\} & \text{constructors with arities} \\
\mathcal{X} & = & \{x_1, x_2, \ldots\} & \text{syntax variables} \\
\mathcal{A} & = & \{\alpha_1, \alpha_2, \ldots\} & \text{semantic variables} \\
\mathcal{T}_{\mathcal{X}} & = & \mathcal{X} \mid C_i^{k_i}(\mathcal{T}_{\mathcal{X}}^1, \ldots, \mathcal{T}_{\mathcal{X}}^{k_i}) & \text{syntax terms} \\
\mathcal{T}_{\mathcal{A}} & = & \mathcal{A} \mid C_i^{k_i}(\mathcal{T}_{\mathcal{A}}^1, \ldots, \mathcal{T}_{\mathcal{A}}^{k_i}) & \text{semantic terms} \\
\mathcal{F} & = & \{F_1^{k_1}, F_2^{k_2}, \ldots\} & \text{names of relations with arities} \\
\mathcal{G} & = & \mathcal{T}_{\mathcal{X}} \equiv \mathcal{T}_{\mathcal{X}} & \text{unification} \\
 & \mid & \mathcal{G} \vee \mathcal{G} & \text{disjunction} \\
 & \mid & \mathcal{G} \wedge \mathcal{G} & \text{conjunction} \\
 & \mid & \textbf{fresh } (\mathcal{X})\ \mathcal{G} & \text{fresh variable introduction} \\
 & \mid & F_i^{k_i}(\mathcal{T}_{\mathcal{X}}^1, \ldots, \mathcal{T}_{\mathcal{X}}^{k_i}) & \text{relation call} \\
\mathcal{S} & = & F_i^{k_i} = \lambda \mathcal{X}_1 \ldots \mathcal{X}_n.\ \mathcal{G} & \text{relations}
\end{array}
$$

Fig. 2. The syntax of relational language

First, we define the syntax of the language (Fig. 2). We define the set of syntax terms $\mathcal{T}_{\mathcal{X}}$ and the set of semantic terms $\mathcal{T}_{\mathcal{A}}$ using constructors $\mathcal{C}$, syntactic variables $\mathcal{X}$ and semantic variables $\mathcal{A}$. Syntactic variables $\mathcal{X}$ are needed to introduce relation arguments and fresh variables. Semantic variables $\mathcal{A}$ can not occur in the source program, but they are introduced during the evaluation of the **fresh** construct.

Also we define the set $\mathcal{F}$ of relation names with arities. Next, we describe the set of goals $\mathcal{G}$. A goal is either a unification of terms, or a disjunction, conjunction, introduction of fresh variable, or a call of a relation. Finally, we define the set of relations $\mathcal{S}$. Each relation consists of a name with arity, a list of argument names, and a goal as its body.

In addition to syntax, we define an intermediate state of relational program.

$$
\begin{array}{rcll}
c_i & = & F_{j_i}^{k_{j_i}}\ (t_{\mathcal{A}}^1, \ldots, t_{\mathcal{A}}^{k_{j_i}}) & \text{calls} \\
C & = & c_1 \wedge \cdots \wedge c_n & \text{conjunction of calls} \\
\mathfrak{T} & = & \mathfrak{T} \vee \mathfrak{T} & \text{disjunction state} \\
 & \mid & \langle \sigma;\ i;\ C \rangle & \text{leaf state}
\end{array}
$$

The state has a shape of a disjunction tree. An internal node "$\vee$" corresponds to the disjunction of two descendant states. Its leaves contain intermediate substitutions σ, a counter of semantic variables i and a conjunction of calls C. A substitution σ is a mapping from semantic variables to semantic terms. The counter of semantic variables is necessary to evaluate the **fresh** construct, which introduces a semantic variable with a new counter. The conjunction of calls contains calls c_i which must be evaluated in this branch. The number of conjuncts in a conjunction may be zero, which we denote as ϵ.

We expand the set of states with an empty state

$$\mathfrak{T}+ = \varnothing.$$

which corresponds to the completion of the evaluation.

Also we introduce two auxiliary functions for working with the program state. The first one, *union*, combines two extended states:

$$union\,(T_1,\,T_2) = \begin{cases} T_2, & \text{if } T_1 = \varnothing \\ T_1, & \text{if } T_1 \neq \varnothing \text{ and } T_2 = \varnothing \\ T_1 \vee T_2, & \text{otherwise} \end{cases}$$

If one of the states is empty *union* returns another state. If both states are not empty then the function returns the combined state.

The next auxiliary function, *push*, is needed to construct the state after the unfolding:

$$push\,(C,\,T) = \begin{cases} \varnothing, & \text{if } T = \varnothing \\ \langle \sigma;\, i;\, C_1 \wedge \bar{C} \wedge C_2 \rangle, & \text{if } T = \langle \sigma;\, i;\, \bar{C} \rangle \text{ and } C = C_1 \wedge \square \wedge C_2 \\ push\,(C, T_1) \vee push\,(C, T_2), & \text{if } T = T_1 \vee T_2 \end{cases}$$

The first argument of this function is a conjunction of calls which contains a *hole* "$\square$". The hole corresponds to a position of the call which we are unfolding. The second argument is the state to unfold. This function recursively traverses the state and for any leaf replaces its calls, making a substitution of this leaf calls into the hole in the first argument of *push*.

Now we define the semantics for the unfolding operation. This semantics (Fig. 3) evaluates a call of a relation and a substitution into a state which corresponds to the body of this relation. Since unfolding is a finite process we can describe it in a big-step style, defining a relation "$\Rightarrow$".

The top-level rule in [Unfold]. Thus, this is the only rule which infers "$\Rightarrow$". It starts the unfolding process of call F with the list of arguments $\bar{t}$ in the context of substitution σ and counter of semantic variables i. First of all, the call F is replaced by the body b of the relation. Also the arguments $\bar{x}$ are substituted by terms $\bar{t}$ and the state $\langle \sigma;\, i;\, \epsilon \rangle$ is initialized. Next, we evaluate the relation body into the corresponding state using the rest of the rules.

The rule [EmptyState] handles the empty state case. The rules [UnifyFail] and [UnifySuccess] perform unification. If the most general unifier (MGU) exists, then we apply rule [UnifySuccess], which updates the substitution σ. If the MGU does not exist, then we apply rule [UnifyFail], which leads to the empty state.

Since this semantics should unfold a call exactly once, we leave all the nested calls unchanged. This behavior is specified by rule [Call]. The nested call is not evaluated, but added to the conjunction, which is contained in the state.

The rule [Fresh] corresponds to the introduction of a fresh variable. In this rule we replace a syntax variable x with a semantic variable α_i. We also increment the counter of semantic variables.

The rules [DisjGoal] and [DisjState] are required to evaluate disjunctions. The first rule evaluates both disjuncts and combines them into a new state using auxiliary function *union*. The second rule handles a disjunction

$$\frac{F = \lambda \bar{x}.b \qquad \langle \sigma;\, i;\, \epsilon \rangle \vdash b[\bar{x} \leftarrow \bar{t}] \rightsquigarrow T}{(\sigma, i) \vdash F(\bar{t}) \Rightarrow T} \quad [\textsc{Unfold}]$$

$$\varnothing \vdash g \rightsquigarrow \varnothing \quad [\textsc{EmptyState}]$$

$$\frac{\nexists\, mgu(t_1, t_2, \sigma)}{\langle \sigma;\, i;\, C \rangle \vdash (t_1 \equiv t_2) \rightsquigarrow \varnothing} \quad [\textsc{UnifyFail}]$$

$$\frac{\bar{\sigma} = mgu(t_1, t_2, \sigma)}{\langle \sigma;\, i;\, C \rangle \vdash (t_1 \equiv t_2) \rightsquigarrow \langle \bar{\sigma};\, i;\, C \rangle} \quad [\textsc{UnifySuccess}]$$

$$\langle \sigma;\, i;\, C \rangle \vdash F(\bar{t}) \rightsquigarrow \langle \sigma;\, i;\, F(\bar{t}) \wedge C \rangle \quad [\textsc{Call}]$$

$$\frac{\langle \sigma;\, i + 1;\, C \rangle \vdash g[x \leftarrow \alpha_i] \rightsquigarrow T}{\langle \sigma;\, i;\, C \rangle \vdash \textbf{fresh}\ x.\, g \rightsquigarrow T} \quad [\textsc{Fresh}]$$

$$\frac{\langle \sigma;\, i;\, C \rangle \vdash g_1 \rightsquigarrow T_1 \qquad \langle \sigma;\, i;\, C \rangle \vdash g_2 \rightsquigarrow T_1}{\langle \sigma;\, i;\, C \rangle \vdash g_1 \vee g_2 \rightsquigarrow union(T_1, T_2)} \quad [\textsc{DisjGoal}]$$

$$\frac{g \neq g_1 \vee g_2 \qquad T_1 \vdash g \rightsquigarrow T_3 \qquad T_2 \vdash g \rightsquigarrow T_4}{T_1 \vee T_2 \vdash g \rightsquigarrow union(T_3, T_4)} \quad [\textsc{DisjState}]$$

$$\frac{\langle \sigma;\, i;\, C \rangle \vdash g_1 \rightsquigarrow T \qquad T \vdash g_2 \rightsquigarrow \bar{T}}{\langle \sigma;\, i;\, C \rangle \vdash g_1 \wedge g_2 \rightsquigarrow \bar{T}} \quad [\textsc{Conj}]$$

Fig. 3. Big step semantics of unfolding

which is contained in the state. As in the previous rule we perform two independent evaluations and then combine the results into a new state.

The last rule [Conj] describes the evaluation of conjunction. In this case we evaluate the first conjunct into a state T, and then evaluate the second conjunct in the context of T. Thus, the second conjunct will be evaluated in the context of all leaves of the state T. Note, if the first conjunct is evaluated into empty state then we can only apply the rule [EmptySet] to the second conjunct. Applying this rule will result in an empty state.

Now we have everything we need to define the semantics of a relational language with directed conjunction (Fig. 4). This small-step semantics sequentially evaluates a state and periodically produces answer substitutions. If an answer is found during program evaluation it is indicated above the transition symbol "$\rightarrow$"; otherwise, "$\circ$" is indicated.

If the current state is a leaf and does not contain any calls then we have an answer in the form of a substitution. In this case, we apply the rule [Answer].

Also, if the current state is a leaf but contains at least one call, we apply rule [ConjUnfold]. In this case we unfold the leftmost call c. Then we construct a new state from the remaining calls C and the unfolding result T using the function *push*.

Finally, if the current state is a disjunction then we evaluate the left disjunct T_1. We apply either rule [Disj] or rule [DisjStep] depending on the results of the evaluation for the left disjunct. The first rule corresponds to the empty state and returns the second disjunct T_2 as a result. The second rule corresponds to a non-empty state and returns a new state ($T_2 \vee \bar{T}_1$). Rearrangement of disjuncts is a necessary action called *interleaving* [9]. It guarantees the completeness of the search.

To make initial state from a call c we need to replace all its syntactic variables with semantic ones:

$$\langle \sigma;\, i;\, \epsilon \rangle \xrightarrow{\sigma} \varnothing \qquad \text{[Answer]}$$

$$\frac{(\sigma, i) \vdash c \Rightarrow T}{\langle \sigma;\, i;\, c \wedge C \rangle \xrightarrow{\circ} push(\square \wedge C, T)} \qquad \text{[ConjUnfold]}$$

$$\frac{T_1 \xrightarrow{\alpha} \varnothing}{T_1 \vee T_2 \xrightarrow{\alpha} T_2} \qquad \text{[Disj]}$$

$$\frac{T_1 \xrightarrow{\alpha} \bar{T}_1 \qquad \bar{T}_1 \neq \varnothing}{T_1 \vee T_2 \xrightarrow{\alpha} T_2 \vee \bar{T}_1} \qquad \text{[DisjStep]}$$

Fig. 4. Semantics with directed conjunction

$$\langle \{\};\, n;\, c[x_0 \leftarrow \alpha_0, \ldots, x_{n-1} \leftarrow \alpha_{n-1}] \rangle.$$

This semantics differs from certified semantics and classical implementations primarily in step size. The unfolding operation performs many actions in a row. But in the classical case interleaving is performed after each elementary action. However, the semantics we presented has retained some common features: disjuncts are switched after each step, and conjunctions are evaluated strictly from left to right.

The order of conjuncts strongly affects the results of the evaluation precisely because of the strictly fixed order of evaluation of the conjuncts. In the following sections we propose two semantics that handles conjunctions more flexibly.

4 A NAIVE FAIR CONJUNCTION

In this section we consider the semantics of miniKanren which fairly unfolds conjuncts. Also, we discuss their advantages and drawbacks.

Instead of unfolding the leftmost call to completion we take some finite number N and bound the number of unfolding steps by N. If after N unfoldings the leftmost call is not eliminated we start to unfold the next call. This process will continue for all calls of the leaf state. When all calls are unfolded N times, we again return the leftmost once again. We call N *unfolding bound*.

To implement this behavior, we need to modify the state structure.

$$\begin{array}{rcll} c_i & = & F_{j_i}^{k_{j_i}}\,(t_{\mathcal{A}}^1, \ldots, t_{\mathcal{A}}^{k_{j_i}}) & \text{calls} \\ C & = & c_1^{m_1} \wedge \cdots \wedge c_n^{m_n} & \text{conjunction of marked calls} \\ \mathfrak{T} & = & \mathfrak{T} \vee \mathfrak{T} & \text{disjunction state} \\ & | & \langle \sigma;\, i;\, C \rangle & \text{leaf state} \end{array}$$

For each call c_i of the leaf we add a natural number m_i which specifies the remaining number of unfoldings. Therefore, each call in the state is marked with an unfolding counter.

Note that the syntax, auxiliary function *union*, the semantics of unfolding are all remained unchanged. The function *push* also preserves its behavior, but now it takes a conjunction of marked calls with the hole and a state in a new form. In addition we need yet another function *set*, which takes an old state without counters in the leaves and a natural number. This number is attached to each call in the state:

$$set(T, m) = \begin{cases} \varnothing, & \text{if } T = \varnothing \\ \langle \sigma;\ i;\ c_1^m \wedge \ldots \wedge c_n^m \rangle, & \text{if } T = \langle \sigma;\ i;\ c_1 \wedge \ldots \wedge c_n \rangle \\ set(T_1, m) \vee set(T_2, m), & \text{if } T = T_1 \vee T_2 \end{cases}$$

Now we are ready to modify the semantics. A new semantics (Fig. 5) evaluates disjuncts in the old way, but it unfolds conjuncts fairly.

$$\langle \sigma;\ i;\ \epsilon \rangle \xrightarrow{\sigma} \varnothing \qquad \text{[Answer]}$$

$$\langle \sigma;\ i;\ c_1^0 \wedge \ldots \wedge c_n^0 \rangle \xrightarrow{\circ} (\sigma, i, c_1^N \wedge \ldots \wedge c_n^N) \qquad \text{[ConjZero]}$$

$$\frac{m > 0 \qquad (\sigma, i) \vdash c_k \Rightarrow T \qquad set(T, m-1) = \bar{T}}{\langle \sigma;\ i;\ c_1^0 \wedge \ldots \wedge c_{k-1}^0 \wedge c_k^m \wedge C \rangle \xrightarrow{\circ} push(c_1^0 \wedge \ldots \wedge c_{k-1}^0 \wedge \square \wedge C, \bar{T})} \qquad \text{[ConjUnfold]}$$

$$\frac{T_1 \xrightarrow{\alpha} \varnothing}{T_1 \vee T_2 \xrightarrow{\alpha} T_2} \qquad \text{[Disj]}$$

$$\frac{T_1 \xrightarrow{\alpha} \bar{T}_1 \qquad \bar{T}_1 \neq \varnothing}{T_1 \vee T_2 \xrightarrow{\alpha} T_2 \vee \bar{T}_1} \qquad \text{[DisjStep]}$$

Fig. 5. Simple fair semantics

First of all, we note that semantics depends on unfolding bound. If the bound is 1, then the evaluation of conjunctions becomes very similar to that for disjunctions. As in the case of disjunction, we switch to the unfolding of a next conjunct after each step. If the bound is set to infinity, then the conjunction will behave like in the directed case. Indeed, the counter of the leftmost conjunct will never become zero and this conjunct will unfold until completion.

Now we consider the semantics in more detail. The rules [Answer], [Disj] and [DisjStep] remain unchanged. However, we introduced two new rules for handling conjunctions. The rule [ConjUnfold] unfolds the leftmost conjunct c_k whose counter m is greater than zero. All calls in the new state T, which we have after the unfolding, require to attach an updated counter; we do this using the function *set*. If all calls in the leaf have a zero counter, then we apply the rule [ConjZero]. This rule updates all counters in the leaf, setting them all to unfolding bound.

```
let rec repeat^o e l =
  (l ≡ []) ∨
  fresh (ls)
    (l ≡ e : ls ∧
     repeat^o e ls)

let divergence^o l =
  repeat^o C_1 l ∧
  repeat^o C_2 l
```

Fig. 6. An example to demonstrate fair conjunction superiority

This semantics more fairly distributes resources between conjuncts. Because of this, relational queries converge more often. Let's go back to the `revers`^{o} example (Fig. 1). As we said earlier, for directed conjunction the query (`revers`^{o} [1, 2, 3] `q`) converges in the specified order of conjuncts and diverges in the reverse order. At the same time, the query (`revers`^{o} `q` [1, 2, 3]) diverges in the specified conjunct order but converges in the reverse order. In the case of fair conjunction, however, both queries converge in both conjunct orders for any finite unfolding bound.

Moreover, some examples diverge for any order of conjuncts in case of directed conjunction but converge in case of fair conjunction (see Fig. 6). The relation searches for lists which on the one hand contain terms $\mathtt{C}_1$ only, and on the other, contain terms $\mathtt{C}_2$ only. Obviously, only an empty list has this property.

We can find this answer by evaluating query `divergence`^{o} `l` under directed conjunction. However, the search for other answers will diverge, and any order of conjuncts preserves this effect. At the same time fair conjunction converges for any finite unfolding bound and any order of conjuncts.

However, in practice this approach has an unstable performance. On the one hand, with a certain unfolding bound the efficiency of a fair conjunction is comparable to the directed conjunction with optimal order of conjunctions. On the other hand, with the wrong unfolding bound we can get an extremely inefficient evaluation, which is hundreds of times slower than the directed conjunction. Therefore, instead of choosing the unfolding bound once and for all we would like to determine it dynamically for each conjunct.

5 FAIR CONJUNCTION BY STRUCTURAL RECURSION

In this section we consider a generalized semantics of miniKanren with a fair conjunction which determines the unfolding bound dynamically. We also consider its specific implementation which makes use of structural recursion of relations.

In the general case we want to parameterize the semantics with an unfolding predicate *pred*. This predicate takes a substitution and a call as arguments. It returns **true** if the call needs to be unfolded further, and **false**, if we need to move on to the next conjunct. We do not get rid of the unfolding bound completely since it is still needed to handle the case when *pred* is false for all calls in a leaf.

$$\langle \sigma;\ i;\ \epsilon \rangle \xrightarrow{\sigma} \varnothing \qquad \text{[Answer]}$$

$$\frac{\bigvee_{j=1}^{n} pred(\sigma, c_j) = \bot}{\langle \sigma;\ i;\ c_1^0 \wedge \ldots \wedge c_n^0 \rangle \xrightarrow{\circ} \langle \sigma;\ i;\ c_1^N \wedge \ldots \wedge c_n^N \rangle} \qquad \text{[ConjZero]}$$

$$\frac{m_k > 0 \qquad \bigvee_{j=1}^{n} pred(\sigma, c_j) = \bot \qquad (\sigma, i) \vdash c_k \Rightarrow T \qquad set(T, m_k - 1) = \bar{T}}{\langle \sigma;\ i;\ c_1^0 \wedge \ldots \wedge c_{k-1}^0 \wedge c_k^{m_k} \wedge \ldots \wedge c_n^{m_n} \rangle \xrightarrow{\circ} push(c_1^0 \wedge \ldots \wedge c_i^0 \wedge \square \wedge \ldots \wedge c_n^{m_n}, \bar{T})} \qquad \text{[ConjUnfold]}$$

$$\frac{\bigvee_{j=1}^{k-1} pred(\sigma, c_j) = \bot \quad pred(\sigma, c_k) = \top \quad (\sigma, i) \vdash c_k \Rightarrow T \quad set(T, \max(0,\ m_k - 1)) = \bar{T}}{\langle \sigma;\ i;\ c_1^{m_1} \wedge \ldots \wedge c_{k-1}^{m_{k-1}} \wedge c_k^{m_k} \wedge C_2 \rangle \xrightarrow{\circ} push(c_1^{m_1} \wedge \ldots \wedge c_{k-1}^{m_{k-1}} \wedge \square \wedge C_2, \bar{T})} \qquad \text{[ConjUnfoldPred]}$$

$$\frac{T_1 \xrightarrow{\alpha} \varnothing}{T_1 \vee T_2 \xrightarrow{\alpha} T_2} \qquad \text{[Disj]}$$

$$\frac{T_1 \xrightarrow{\alpha} \bar{T}_1 \qquad \bar{T}_1 \neq \varnothing}{T_1 \vee T_2 \xrightarrow{\alpha} T_2 \vee \bar{T}_1} \qquad \text{[DisjStep]}$$

Fig. 7. Semantics of fair conjunction by structural recursion

The semantics parameterized by the unfolding predicate is shown in Fig. 7. Since we modify only the behavior of conjunction the rules [Answer], [Disj] and [DisjStep] remain unchanged. Three updated rules are responsible for conjunction behavior. If the predicate *pred* is $\top$ for at least one call, then we apply the rule [ConjUnfoldPred], which unfolds the leftmost such call and decrements its counter. If the predicate *pred* is $\bot$ for all calls, but there is at least one call with a nonzero counter, then we apply the rule [ConjUnfold], which unfolds the leftmost such call and decrements its counter. If the predicate is $\bot$ for all calls and all the counters are equal to zero, then we apply the rule [ConjZero], which sets all the counters to unfolding bound.

As for the predicate, we need a criterion that can tell a call that is profitable to unfold now apart from a call which is worth deferring. We propose a criterion that works correctly for structurally recursive relations. Such relations have at least one argument which structurally decreases with each step of the recursion. This property allows us to control the depth of unfolding. We propose to use the following predicate:

$$pred(\sigma, F^k(t_1, \ldots, t_k)) = \begin{cases} & \text{if } F^k \text{ is structural recursion relation,} \\ \top, & i \text{ is number of structural recursion argument,} \\ & t_i \text{ is not fresh variable in } \sigma \\ \bot, & \text{otherwise.} \end{cases}$$

As long as at least one argument along which structural recursion is performed is not a free variable, we continue to unfold this call. If all such arguments are free, then the call will diverge in the current substitution, so we proceed to evaluate the next call. Since structurally recursive arguments decrease, in a finite number of steps the evaluation will either complete, or all arguments of structural recursion will become free variables.

```
let rec appendᵒ x y xy =
  (x ≡ [] ∧ y ≡ xy) ∨
  fresh (e xs xys) (
    x ≡ e : xs ∧
    xs ≡ e : xys ∧
    appendᵒ xs y xys)
```

Fig. 8. Relational concatenation

For example, the relation append^o (Fig. 8) is structurally recursive on its first and third arguments. Indeed, the nested call append^o takes `xs` as its first argument, which is a subterm of `x`. Also, append^o takes `xys` as the third argument, which is a subterm of `xy`. If at least one of them is a fixed-length list, then the relation will converge. Otherwise, $\mathtt{x} \equiv t_1 : \ldots t_n : \alpha_1$ and $\mathtt{xy} \equiv \bar{t}_1 : \ldots \bar{t}_m : \alpha_2$. Therefore, in $max(n, m)$ steps, both arguments become free variables.

We are currently working on proving the independence of this semantics from the order of the conjuncts in the case when all relations are structurally recursive.

6 EVALUATION

In this section we present the results of the evaluation of three semantics on a set of examples; the semantics were implemented in Haskell in the form of interpreters. For evaluation we've chosen two simple programs (list reversing and list sorting) and three more complicated (the "Hanoi Towers"[1] solver, the "Bridge and torch

[1] https://en.wikipedia.org/wiki/Tower_of_Hanoi

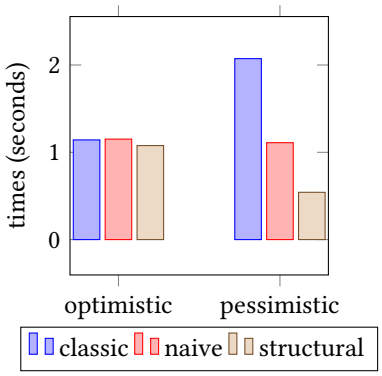

Fig. 9. revers^o evaluation for a list with a length of 90

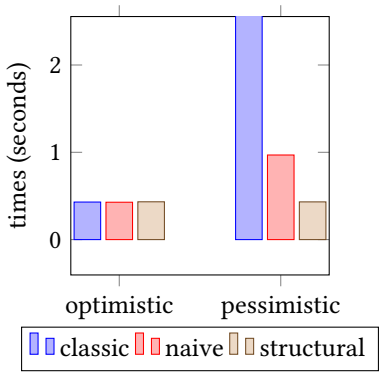

Fig. 10. sort^o evaluation for a list with a length of 5

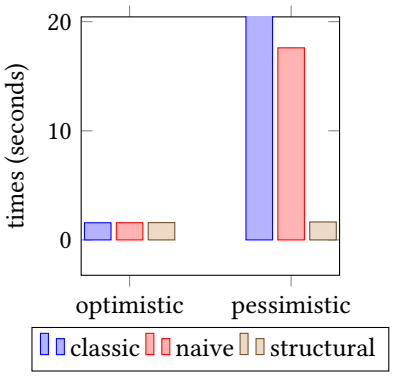

Fig. 11. "The Tower of Hanoi" solver evaluation

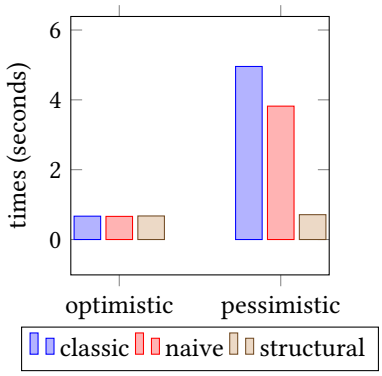

Fig. 12. "Bridge and torch problem" solver evaluation

problem"[2] solver and "Water pouring puzzle"[3] solver). Each program was written in two versions: "optimistic" (with the order of important conjuncts set to provide the best performance) and "pessimistic" (with the order of important conjuncts set to provide the worst performance). Also we evaluated list reversing and list sorting in both directions. In the case of the list reversing, queries `(revers`^{o} `[1;2;3] q)` and `(revers`^{o} `q [1;2;3])` will give the same answer `q = [3;2;1]` but the "optimistic" order of conjuncts is different for them. In the case of list sorting, queries `(sort`^{o} `[1;2;3] q)` and `(sort`^{o} `q [1;2;3])` will give different answers. The first one gives sorted list `q = [1;2;3]`, the second one gives all permutations of list `[1;2;3]`.

All benchmarks were run ten times, and the average time was taken. For the naive fair conjunction we cherry-picked the best value of unfolding bound manually. For the fair conjunction based on structural recursion the bound was set to 100.

Fig. 9-12 show the results of evaluation in the form of bar charts. In the optimistic case, the results are similar for all semantics. In the pessimistic case the evaluation time of the directed conjunction rapidly increases, the evaluation time of the naive fair conjunction also increases, but not so much. The fair conjunction based on structural recursion demonstrates a similar efficiency as in the optimistic case.

[2]https://en.wikipedia.org/wiki/Bridge_and_torch_problem
[3]https://en.wikipedia.org/wiki/Water_pouring_puzzle

The results are presented in more detail in Fig. 13. “Hanoi Towers” solver has name `hanoi`^{o}, “Bridge and torch problem” solver has name `bridge`^{o} and “Water pouring puzzle” solver has name `water`^{o}. We can conclude that forward and backward `sort`^{o} runtime in the pessimistic case increases very rapidly with increasing the list length for directed and naive fair conjunctions. In the case of the fair conjunction based on structural recursion the running time in pessimistic case increases on a par with that in the optimistic one. Also the solver `water`^{o} very slow in the pessimistic case for directed and naive fair. However, fair conjunction based on structural recursion pessimistic case is no different from an optimistic case.

relation	size	directed conjunction		naive fair conjunction		structural recursion	
		optimistic	pessimistic	optimistic	pessimistic	optimistic	pessimistic
forward revers^o	30	0.465	0.532	0.468	0.461	0.438	0.425
	60	0.579	0.828	0.577	0.658	0.545	0.450
	90	1.142	2.073	1.151	1.110	1.077	0.542
backward revers^o	30	0.541	0.573	0.550	0.540	0.516	0.518
	60	0.637	0.867	0.642	0.805	0.636	0.637
	90	1.268	1.778	1.274	1.359	1.289	1.273
forward sort^o	3	0.418	0.432	0.420	0.420	0.424	0.425
	4	0.424	3.924	0.424	0.455	0.429	0.429
	5	0.430	>300	0.428	0.969	0.433	0.432
	6	0.434	>300	0.430	11.577	0.434	0.437
	30	1.664	>300	1.636	>300	1.723	1.751
backward sort^o	3	0.511	0.511	0.509	0.513	0.516	0.518
	4	0.525	0.823	0.534	0.539	0.534	0.530
	5	0.667	69.725	0.692	1.443	0.689	0.697
	6	2.880	>300	2.891	56.107	2.921	2.936
hanoi^o	-	1.574	>300	1.579	17.604	1.585	1.646
bridge^o	-	0.669	4.956	0.663	3.820	0.675	0.712
water^o	-	3.132	>300	3.168	>300	3.220	3.414

Fig. 13. The results of evaluation: running times of benchmarks in seconds

To summarize, the fair conjunction based on structural recursion does not introduce any essential overhead in comparison with directed conjunction in an optimistic case. At the same time it weakly depends on the order of the conjuncts, and thus demonstrates much better performance in the pessimistic case.

7 CONCLUSION

In this paper we proposed a new approach for the execution of relational programs with structural recursion. This approach reduces the performance impact of conjunct order.

In the future we plan to generalize the proposed approach to a larger class of programs. We also plan to formalize our approach in Coq [2] proof assistant system and prove that the convergence of the execution does not depend on the order of the conjuncts.

REFERENCES

[1] Claire E. Alvis, Jeremiah J. Willcock, and William E. Byrd. 2011. cKanren: miniKanren with Constraints. *Workshop on Scheme and Functional Programming* (2011).

[2] Yves Bertot and Pierre Castéran. 2004. *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*. Springer. https://doi.org/10.1007/978-3-662-07964-5

[3] William E. Byrd. 2009. *Relational Programming in miniKanren: Techniques, Applications, and Implementations.* Ph.D. Dissertation. Indiana University.
[4] William E. Byrd, Michael Ballantyne, Gregory Rosenblatt, and Matthew Might. 2017. A Unified Approach to Solving Seven Programming Problems (Functional Pearl). *Proc. ACM Program. Lang.* 1, ICFP, Article 8 (Aug. 2017), 26 pages. https://doi.org/10.1145/3110252
[5] William E. Byrd and Daniel P. Friedman. 2007. αkanren: A Fresh Name in Nominal Logic Programming. *Proceedings of the 2007 Annual Workshop on Scheme and Functional Programming* (2007), 79–90.
[6] William E. Byrd, Eric Holk, and Daniel P. Friedman. 2012. miniKanren, Live and Untagged: Quine Generation via Relational Interpreters (Programming Pearl). *Workshop on Scheme and Functional Programming* (2012).
[7] Daniel P. Friedman, William E. Byrd, and Oleg Kiselyov. 2005. *The Reasoned Schemer.* The MIT Press.
[8] Jason Hemann, Daniel Friedman, William Byrd, and Matthew Might. 2016. A small embedding of logic programming with a simple complete search. 96–107. https://doi.org/10.1145/2989225.2989230
[9] Oleg Kiselyov, Chung-chieh Shan, Daniel P. Friedman, and Amr Sabry. 2005. Backtracking, Interleaving, and Terminating Monad Transformers: (Functional Pearl). *SIGPLAN Not.* 40, 9 (Sept. 2005), 192–203. https://doi.org/10.1145/1090189.1086390
[10] Kuang-Chen Lu, Weixi Ma, and Daniel P. Friedman. 2019. Towards a miniKanren with fair search strategies. *The miniKanren and Relational Programming Workshop.*
[11] Joseph Near, William Byrd, and Daniel Friedman. 2008. αleanTAP: A Declarative Theorem Prover for First-Order Classical Logic, Vol. 5366. 238–252. https://doi.org/10.1007/978-3-540-89982-2_26
[12] Dmitri Rozplokhas and Dmitri Boulytchev. 2018. Improving Refutational Completeness of Relational Search via Divergence Test. In *Proceedings of the 20th International Symposium on Principles and Practice of Declarative Programming, PPDP 2018, Frankfurt am Main, Germany, September 03-05, 2018.* 18:1–18:13. https://doi.org/10.1145/3236950.3236958
[13] Dmitry Rozplokhas, Andrey Vyatkin, and Dmitri Boulytchev. 2019. Certified Semantics for miniKanren. *The miniKanren and Relational Programming Workshop.*
[14] Cameron Swords and Daniel P. Friedman. 2013. rKanren: Guided Search in miniKanren. *Proceedings of the 2013 Annual Workshop on Scheme and Functional Programming* (2013).