A Complexity Study for Interleaving Search

DMITRY ROZPLOKHAS and DMITRY BOULYTCHEV, Saint Petersburg State University and JetBrains Research, Russia

We study the worst-case time complexity of relational programs for the canonical implementation of miniKanren. We propose a model that breaks the evaluation time in miniKanren into several different parts and provides methods to estimate the complexity of these parts. These methods can be combined into an approach for analyzing the complexity of recursive relations using the ideas originating from symbolic execution. Our approach puts a number of limitations on the program being analyzed but we show that is still practically applicable by applying it to a number of typical relations in miniKanren.

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

Additional Key Words and Phrases: miniKanren, time complexity, interleaving search, triangular substitution, symbolic execution

ACM Reference Format:
Dmitry Rozplokhas and Dmitry Boulytchev. . A Complexity Study for Interleaving Search. 1, 1 (August), 24 pages.

1 INTRODUCTION

This paper deals with the problem of the worst-case time complexity estimations for relational program evaluation in the canonical implementation of miniKanren based on interleaving search [5]. Despite its simple implementation, the search in miniKanren has a number of subtleties affecting the performance, which are hard to think about intuitively altogether. It is easy to overlook some of them and the behavior of the search in certain cases can be really surprising.

As a motivative example, consider two implementations of a standard recursive relation calculating the length of a list (see Fig. 1). They differ only in the orders of conjuncts. Although the $\texttt{length}^o_d$ relation can be seen as a more direct definition of a function as a relation (all steps of usual length evaluation written up in order), it is well-known that the $\texttt{length}^o$ with the recursive call in the end is much faster when running this relation backward (in fact, the search diverges if we run $\texttt{length}^o_d$ backward, while for $\texttt{length}^o$ it terminates). What is less known and what we find more unexpected is the fact that if we run both relations forward (specifying a list and asking for its length) $\texttt{length}^o$ is still much faster than $\texttt{length}^o_d$, although they perform the same number of unifications. One can see the comparative time of the search in Fig. 2. The difference is even more staggering if we switch off occurs checks in unifications. It's OK because for simple queries like this occurs checks are never violated. In the same Fig. 2 you can see that the *asymptotic complexity* becomes different in this case: it is linear for $\texttt{length}^o$ and quadratic for $\texttt{length}^o_d$.

Authors' address: Dmitry Rozplokhas, rozplokhas@gmail.com; Dmitry Boulytchev, dboulytchev@math.spbu.ru, Saint Petersburg State University and JetBrains Research, Russia.

XXXX-XXXX//8-ART $15.00
https://doi.org/

```
length^o = λ a n →
  ((a ≡ Nil) ∧ (n ≡ O)) ∨
  (fresh (h t n')
     (a ≡ Cons(h, t)) ∧
     (n ≡ S(n')) ∧
     (length^o t n'))
  )
```

```
length^o_d = λ a n →
  ((a ≡ Nil) ∧ (n ≡ O)) ∨
  (fresh (h t n')
     (a ≡ Cons(h, t)) ∧
     (length^o_d t n') ∧
     (n ≡ S(n')))
  )
```

Fig. 1. Example of two implementations of the length calculating relation

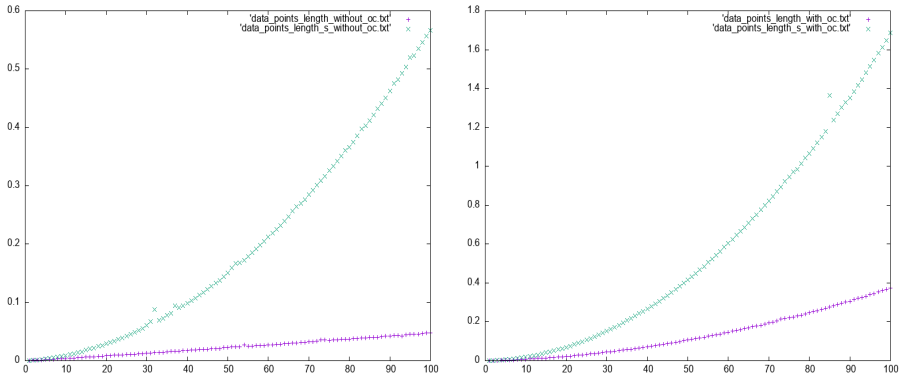

Fig. 2. Time (in seconds) of the search for the relations $\texttt{length}^o$ (purple) and $\texttt{length}^o_d$ (green) depending on the lentgh of the list. Left: without occurs check. Right: with occurs check.

After investigating the execution for this example in detail we found that the difference is caused not by unifications but by the process of *scheduling* of goals during the search. During the execution of a program in miniKanren a lazy structure is build that decomposes the goals into unifications, performs these unifications in a certain order, and passes the results appropriately. For the $\texttt{length}^o_d$ relation this structure becomes linear in size just because of the order in conjunctions and increases the time of scheduling significantly. This kind of effect is hard to predict and measure without a formal model for performance in miniKanren.

This paper presents such a model. We study evaluation in a specific (canonical) implementation of miniKanren with goals evaluated to lazy streams in program-written order, but we rely only on the basic principles of implementation of miniKanren, so the model should be easily adoptable for a large class of regular implementations. We state that the total time of the search in miniKanren breaks into three separate parts: the time of scheduling (T_s) that breaks the evaluation into a sequence of unifications, the time required to perform these unifications, and the time of reifications (T_r) that reconstruct the result in an expected form in the end. The time of unifications can be further divided into the time of occurs checks (T_{occ}) during the unification and the time (T_{uni}) of the rest of the unification algorithm (this division will help us to see how large is the part of the total time that occurs check, which often can be omitted, takes). So the total time of the search can be calculated as the sum of four components:

$$T = T_s + T_{uni} + T_{occ} + T_r$$

$$
\begin{array}{lcll}
C & = & \{C_i^{k_i}\} & \text{constructors} \\
\mathcal{T}_X & = & X \cup \{C_i^{k_i}(t_1, \ldots, t_{k_i}) \mid t_j \in \mathcal{T}_X\} & \text{terms over the set of variables } X \\
\mathcal{D} & = & \mathcal{T}_\emptyset & \text{ground terms} \\
\mathcal{X} & = & \{x, y, z, \ldots\} & \text{syntactic variables} \\
\mathcal{A} & = & \{x^?, y^?, z^? \ldots\} & \text{logic variables} \\
\mathcal{R} & = & \{R_i^{k_i}\} & \text{relational symbols with arities} \\
\mathcal{G} & = & \mathcal{T}_\mathcal{X} \equiv \mathcal{T}_\mathcal{X} & \text{equality} \\
 & & \mathcal{G} \wedge \mathcal{G} & \text{conjunction} \\
 & & \mathcal{G} \vee \mathcal{G} & \text{disjunction} \\
 & & \textbf{fresh } \mathcal{X} \text{ . } \mathcal{G} & \text{fresh variable introduction} \\
 & & R_i^{k_i}(t_1, \ldots, t_{k_i}),\ t_j \in \mathcal{T}_\mathcal{X} & \text{relational symbol invocation} \\
\mathcal{S} & = & \{R_i^{k_i} = \lambda\ x_1^i \ldots x_{k_i}^i \text{ . } g_i;\}\ g & \text{specification}
\end{array}
$$

Fig. 3. The syntax of miniKanren

We show how these components can be estimated and compared to each other in terms of asymptotics. The scheduling time complexity can be measured precisely since we link it to a specific value which we call *scheduling factor*, defined in terms of existing formal semantics of miniKanren (we recall the existing formal descriptions of miniKanren in section 2) and can be calculated using a number of equations (section 3). The other time components are hard to estimate precisely in general, as they are connected to the unification process, but we identify two criteria that determine a wide range of cases for which these time components can be estimated easily (section 4). These separate methods for estimation of different components of the time of the search can be put together in one approach for calculating manually the time complexity for a given query to a recursive relation (if this query satisfies the stated criteria) using the principles of symbolic execution (section 5). We then show the applicability of our method by applying it for a number of realistic miniKanren relations (section 6).

2 BACKGROUND: SYNTAX AND OPERATIONAL SEMANTICS

In this section, we recollect some known formal descriptions for miniKanren language that will be used as a basis for our development. Specifically, we restate the syntax of the basic version of the language and the operational semantics for program evaluation. The descriptions here are taken from [10] (with a few non-essential adjustments for presentation purposes) to make the paper self-contained, more details and explanations can be found there.

The syntax of the basic version of miniKanren is shown in Fig. 3. All data is presented using terms $\mathcal{T}_X$ built from a fixed set of constructors C with known arities and variables from a given set X. We parameterize the terms with an alphabet of variables since in the semantic description we will need *two* kinds of variables: *syntactic* variables $\mathcal{X}$, used for bindings in the definitions, and *logic* variables $\mathcal{A}$, which are introduced and unified during the evaluation.

In this version of miniKanren there are five types of goals: unification of two terms, conjunction and disjunction of goals (the "conde" operator from the canonical versions of miniKanren, split in the two for simplicity), fresh logic variable introduction, and invocation of some relational definition. For the sake of brevity, in code snippets, we abbreviate immediately nested "**fresh**" constructs into the one, writing "**fresh** $x\ y \ldots g$" instead of "**fresh** x . **fresh** y . $\ldots g$".

$$
\begin{array}{lcll}
\Sigma & = & \mathcal{A} \to \mathcal{T}_{\mathcal{A}} & \text{substitutions} \\
E & = & \Sigma \times \mathbb{N} & \text{environments} \\
 & & & \\
S & = & \langle \mathcal{G}, E \rangle & \text{task} \\
 & & S \oplus S & \text{sum} \\
 & & S \otimes \mathcal{G} & \text{product} \\
\hat{S} & = & \diamond \mid S & \text{states} \\
L & = & \circ \mid E & \text{labels}
\end{array}
$$

Fig. 4. States and labels in the LTS for miniKanren

The *specification* $\mathcal{S}$ consists of a set of relational definitions and a top-level goal. A top-level goal represents a search procedure that returns a stream of substitutions for the free variables of the goal. The language we consider is first-order, as goals can not be passed as parameters, returned, or constructed at runtime.

During the evaluation of miniKanren program an environment, consisting of substitution for logic variables and a counter of allocated logic variables, is threaded through the computation and updated in every unification and fresh variable introduction. The substitution in the environment at a given point and a given branch of evaluation contains all the information about relations between the logical variables at this point. We will later refer to the substitution in the environment at a given point as a "current substitution" in informal explanations. The different branches are combined via *interleaving search* procedure [7]. The answers for the given query are extracted from the final environments (they are the values of the queried variables in the substitution in the final environment).

This search procedure is formally described by operational semantics in the form of a labeled transition system. This semantics corresponds to the canonical implementation of interleaving search.

The form of states and labels in the transition system is defined in Fig. 4. Non-terminal states S have a tree-like structure with intermediate nodes corresponding to partially evaluated conjunctions ("$\otimes$") or disjunctions ("$\oplus$"). A leaf in the form $\langle g, e \rangle$ determines a task to evaluate a goal g in an environment e. For a conjunction node, its right child is always a goal since it cannot be evaluated unless some result is provided by the left conjunct. We also need a terminal state $\diamond$ to represent the end of the evaluation. The label "$\circ$" is used to mark those steps which do not provide an answer; otherwise, a transition is labeled by an updated environment.

The transition rules are shown in Fig. 5. The introduced transition system is completely deterministic. Therefore a derivation sequence for a state s determines a certain *trace* — a sequence of states and labeled transitions between them. It may be either finite (ending with the terminal state $\diamond$) or infinite. We will denote by $\mathcal{T}r^{st}(s)$ the sequence of states in the trace for initial state s and by $\mathcal{T}r^{ans}(s)$ the sequence of answers in the trace for initial state s. The sequence $\mathcal{T}r^{ans}(s)$ corresponds to the stream of answers in the reference miniKanren implementations.

3 SCHEDULING COMPLEXITY

We may notice that operational semantics described in the previous section can be used to calculate the number of elementary scheduling steps. In this section, we define a specific value that estimates the scheduling time and give some equations to calculate this value for a given *semantic state*. However, our ultimate goal is to provide a way to deduce the complexity factor recursively for

$$\langle t_1 \equiv t_2,\ (\sigma, n)\rangle \xrightarrow{\circ} \diamond,\ \nexists\, mgu\,(t_1\sigma, t_2\sigma) \qquad \text{[UnifyFail]}$$

$$\langle t_1 \equiv t_2,\ (\sigma, n)\rangle \xrightarrow{(mgu\,(t_1\sigma, t_2\sigma)\circ\sigma),\ n)} \diamond \qquad \text{[UnifySuccess]}$$

$$\langle g_1 \vee g_2,\ e\rangle \xrightarrow{\circ} \langle g_1,\ e\rangle \oplus \langle g_2,\ e\rangle \qquad \text{[Disj]}$$

$$\langle g_1 \wedge g_2,\ e\rangle \xrightarrow{\circ} \langle g_1,\ e\rangle \otimes g_2 \qquad \text{[Conj]}$$

$$\langle \mathbf{fresh}\ x\,.\,g,\ (\sigma, n)\rangle \xrightarrow{\circ} \langle g\,[\alpha_{n+1}\,/_x],\ (\sigma, n+1)\rangle \qquad \text{[Fresh]}$$

$$\frac{R_i^{k_i} = \lambda\, x_1 \ldots x_{k_i}\,.\,g}{\langle R_i^{k_i}\,(t_1, \ldots, t_{k_i}),\ e\rangle \xrightarrow{\circ} \langle g\,[t_1\,/_{x_1} \ldots t_{k_i}\,/_{x_{k_i}}],\ e\rangle} \qquad \text{[Invoke]}$$

$$\frac{s_1 \xrightarrow{\circ} \diamond}{(s_1 \oplus s_2) \xrightarrow{\circ} s_2} \qquad \text{[DisjStop]}$$

$$\frac{s_1 \xrightarrow{e} \diamond}{(s_1 \oplus s_2) \xrightarrow{e} s_2} \qquad \text{[DisjStopAns]}$$

$$\frac{s \xrightarrow{\circ} \diamond}{(s \otimes g) \xrightarrow{\circ} \diamond} \qquad \text{[ConjStop]}$$

$$\frac{s \xrightarrow{e} \diamond}{(s \otimes g) \xrightarrow{\circ} \langle g,\ e\rangle} \qquad \text{[ConjStopAns]}$$

$$\frac{s_1 \xrightarrow{\circ} s_1'}{(s_1 \oplus s_2) \xrightarrow{\circ} (s_2 \oplus s_1')} \qquad \text{[DisjStep]}$$

$$\frac{s_1 \xrightarrow{e} s_1'}{(s_1 \oplus s_2) \xrightarrow{e} (s_2 \oplus s_1')} \qquad \text{[DisjStepAns]}$$

$$\frac{s \xrightarrow{\circ} s'}{(s \otimes g) \xrightarrow{\circ} (s' \otimes g)} \qquad \text{[ConjStep]}$$

$$\frac{s \xrightarrow{e} s'}{(s \otimes g) \xrightarrow{\circ} (\langle g,\ e\rangle \oplus (s' \otimes g))} \qquad \text{[ConjStepAns]}$$

Fig. 5. Operational semantics of interleaving search

a given query. This problem will be addressed in Section 5, which will make use of recurrent equations presented here.

We also restrict our considerations only by the case when the evaluation of a goal in question terminates. Indeed, if the search diverges, no reasonable complexity estimation for the time of the whole search can be given. A much more interesting question would be the complexity estimation for coming up with some *specific* answer; however for now this problem seems to be too hard to tackle.

Our first idea is to take the number of states $d\,(s)$ in the *finite* trace for a given state s:

$$d\,(s) \stackrel{\text{def}}{=} |\mathcal{T}r^{st}(s)|$$

However, it turns out, that this value alone is not enough to provide an accurate scheduling complexity estimation. The reason is that some elementary steps in the semantics are not elementary

in existing implementations. Namely, a careful analysis of the semantics discovers that it involves navigation to the leftmost leaf of the state, which in implementations corresponds to a number of steps, proportional to the length of the leftmost branch of the state in question. Here we provide an *ad-hoc* definition for this value, $t(s)$, which we call the *scheduling factor*:

$$t(s) \stackrel{\text{def}}{=} \sum_{s_i \in \mathcal{T}r^{st}(s)} lh(s_i)$$

where

$$\begin{aligned} lh(\langle g,\ e\rangle) &\stackrel{\text{def}}{=} 1 \\ lh(s_1 \oplus s_2) &\stackrel{\text{def}}{=} lh(s_1) + 1 \\ lh(s \otimes g) &\stackrel{\text{def}}{=} lh(s) + 1 \end{aligned}$$

The following lemma provides a fundamental relation between these two estimations of the scheduling complexity:

Lemma 3.1. *For any state s*

$$d(s) \le t(s) \le d^2(s)$$

Our next goal is to come up with the equations which relate the scheduling complexity of states with the scheduling complexity of their (immediate) substates. We take scheduling factor $t(s)$ as a value that determines the scheduling complexity T_s, but we will also need to calculate $d(s)$ as it will be used in the equations for $t(s)$.

The following lemma, obvious from the definitions, will be enough to deal with a basic (leaf state) case:

Lemma 3.2. *If*

$$\langle g,\ e\rangle \xrightarrow{\circ} s'$$

or

$$\langle g,\ e\rangle \xrightarrow{a} s'$$

then

$$d(\langle g,\ e\rangle) = d(s') + 1$$

and

$$t(\langle g,\ e\rangle) = t(s') + 1$$

In state of the form $s_1 \oplus s_2$ the substates are evaluated separately, one step at a time for each substate, so the total number of semantic steps is just a sum. However, for the scheduling factor, there is an extra summand $cost_{\oplus}(s_1 \oplus s_2)$ since the heights of the states in the trace are increased by one additional $\oplus$-node on the top. This additional node exists in the trace until one of the substates is evaluated completely, so the scheduling factor is increased by the number of steps before such event. So we have the following lemma.

Lemma 3.3. *For any two states s_1 and s_2*

$$\begin{aligned} d(s_1 \oplus s_2) &= d(s_1) + d(s_2) \\ t(s_1 \oplus s_2) &= t(s_1) + t(s_2) + cost_{\oplus}(s_1 \oplus s_2) \end{aligned}$$

where

$$cost_{\oplus}(s_1 \oplus s_2) = \min\{2 \cdot d(s_1) - 1, 2 \cdot d(s_2)\}$$

For the states of the form $s \otimes g$ the reasoning is the same, but the result is more complicated. In $\otimes$-state the left substate is evaluated until an answer is found, which is then taken as *an environment* for the evaluation of the right subgoal. Thus, in the equations for $\otimes$-states we sum the evaluation time of the second goal for all the answers generated for the first substate. The tasks of evaluating the right subgoal in different environments are added to the evaluation of the left substate by the creation of an $\oplus$-state, so for scheduling factor there is an additional summand $cost_{\oplus}(\langle g,\ a_i\rangle \oplus s'_i)$ for each answer with s'_i being the state after discovering the given answer. There is also an extra summand $cost_{\otimes}(s \otimes g)$ to the scheduling factor because of the $\otimes$-node that increases the height in the trace, analogous to the one caused by $\oplus$-nodes. We can notice that $\otimes$-node is always placed immediately over the left substate so this addition is exactly the number of steps for the left substate. Therefore the resulting equations for $\otimes$-states are as follows.

Lemma 3.4. *For any state s and any goal g*

$$\begin{aligned} d(s \otimes g) &= d(s) + \sum_{a_i \in \mathcal{T}r^{ans}(s)} d(\langle g,\ a_i\rangle) \\ t(s \otimes g) &= t(s) + \Big(\sum_{a_i \in \mathcal{T}r^{ans}(s)} (t(\langle g,\ a_i\rangle) + cost_{\oplus}(\langle g,\ a_i\rangle \oplus (s'_i \otimes g)))\Big) + cost_{\otimes}(s \otimes g) \end{aligned}$$

where

$$cost_{\oplus}(s_1 \oplus s_2) = \min\{2 \cdot d(s_1) - 1, 2 \cdot d(s_2)\}$$

$$cost_{\otimes}(s \otimes g) = d(s)$$

s'_i is the next state in the trace for s after the transition labeled with the answer a_i

After unfolding the auxiliary definitions the last equation becomes too cumbersome to use directly, so we will use some its approximations instead. One option is to go with the first argument of "min" in $cost_{\oplus}(\langle g,\ a_i\rangle \oplus s'_i)$. It should be a good approximation in the case when there are several answers passed to the second goal and for none of them the number of steps surpasses the *overall* number of steps for all other answers (the second argument of "min" will include the sum for the rest of the answers).

Corollary 3.5. *For any state s and any goal g*

$$t(s \otimes g) \le t(s) + d(s) + \sum_{a_i \in \mathcal{T}r^{ans}(s)} (t(\langle g,\ a_i\rangle) + 2 \cdot d(\langle g,\ a_i\rangle))$$

In the case when there is only one answer, however, we should rather go with the second argument of "min".

In this case the number of steps $d(s'_1 \otimes g)$ is equal to the number of steps $d(s'_1)$ since no more answers are produced, and we can approximate it by the length $d(s)$ of the whole trace for s.

Corollary 3.6. *For any state s and any goal g, if $\mathcal{T}r^{ans}(s) = \{a\}$, then*

$$t(s \otimes g) \leq t(s) + 3 \cdot d(s) + t(\langle g,\ a\rangle)$$

Finally, since we estimate only up to a multiplicative constant (in particular, it does not matter by what constants we multiply the values of $d(\cdot)$ when calculating the scheduling factor) we can derive from these results two compact scheduling time approximations for goals in the form of sequences of disjuncts/conjuncts. These two approximations work regardless of the associativity/grouping of subformulas; thus a certain constant c_k, depending only on k, comes in.

For conjunctions, we have the following one.

Lemma 3.7. *Let $g = g_1 \wedge \cdots \wedge g_k$ and let A_i be a set of all answers that are passed to g_i at some stage starting from some initial environment e_0*

$$\begin{aligned} A_1 &= \{e_0\} \\ A_{i+1} &= \bigcup_{a \in A_i} \mathcal{T}r^{ans}(\langle g_i,\ a\rangle) \end{aligned}$$

Then

$$\begin{aligned} d(\langle g,\ e\rangle) &= \sum_{1 \leq i \leq k} \sum_{a \in A_i} d(\langle g_i,\ a\rangle) \\ t(\langle g,\ e\rangle) &\leq \sum_{1 \leq i \leq k} \sum_{a \in A_i} t(\langle g_i,\ a\rangle) + c_k \cdot \sum_{1 \leq i \leq k} \sum_{a \in A_i} d(\langle g_i,\ a\rangle), \end{aligned}$$

In the case when all A_i contain only one answer

$$t(\langle g,\ e\rangle) \leq \sum_{1 \leq i \leq k} \sum_{a \in A_i} t(\langle g_i,\ a\rangle) + c_k \cdot \sum_{1 \leq i \leq k-1} \sum_{a \in A_i} d(\langle g_i,\ a\rangle)$$

When applying the estimation from corollary 3.5 we have an extra summand in the form of the number of steps (multiplied by some constant) for all conjuncts. The only exception is the case when every conjunct produces no more than one answer, then we can use the corollary 3.6 everywhere instead and drop out the additional number of steps for the last conjunct. Besides that, a constant number of steps is required to turn each conjunction into a $\otimes$-state, but we may integrate this extra constant into c_k.

For disjunctions, the lemma is the following one.

Lemma 3.8. *Let $g = g_1 \vee \cdots \vee g_k$ and $1 \leq l \leq k$; then*

$$\begin{aligned} d(\langle g,\ e\rangle) &\leq \sum_{1 \leq i \leq k} d(\langle g_i,\ e\rangle) \\ t(\langle g,\ e\rangle) &\leq \sum_{1 \leq i \leq k} t(\langle g_i,\ e\rangle) + c_k \cdot \sum_{\substack{1 \leq i \leq k \\ i \neq l}} d(\langle g_i,\ e\rangle). \end{aligned}$$

Roughly speaking, if we have a disjunct g_m with a number of steps larger than all the steps for other disjuncts combined, then when applying lemma 3.3 we again will have an extra summand in the form of the number of steps for all disjuncts except for the g_m (it will always be the largest argument of "min"). But if we can drop out the *largest* the number of steps among disjuncts, we can also drop out any other instead, that's where arbitrary l comes from. The case when there is no such g_m has to be considered separately; it is simpler since then all the numbers of steps are the same up to a multiplicative constant.

4 UNIFICATION AND REIFICATION COMPLEXITY

Syntactic unification of terms is a core operation for logic programming in whole and relational programming in particular. However, the performance characteristics of conventional unification algorithms are rather hard to assess. The known worst-case estimations say very little about the behavior of unification in *practically important cases*, and, in general, the very notion of "practical importance" is hard to formalize (which constitutes a generic problem for applied complexity as well).

The practical observations witness, that while the worst-case complexity for the conventional unification algorithm is exponential, in the majority of cases met in practical logic programming the time for each unification problem instance throughout the program execution is linear or even constant on the size of the input.

miniKanren has a distinctive way of implementing unification fitting in accordance with its ideology. First, since miniKanren aims at the purely functional implementation of an embedded logical language it uses a triangular form of substitution [1] which allows a simple extension in a non-mutable fashion. Such substitutions are lazy in the sense that they hold a partially substituted value for each variable, so to obtain a fully substituted value it may be necessary to apply a substitution repeatedly. In particular, a full cycle of substitution application is needed at the end of the search to get the result for a queried variable. This process is called *reification*. miniKanren uses the conventional Robinson's algorithm for unification [9], adjusted for triangular substitutions [5]. Second, since miniKanren commits to adhere to logical consistency, by default it always performs *occurs checks* during the unification. Occurs check ensures that a binding being added into the substitution does not introduce any circularity, which is crucial for establishing the soundness of unification results. However, being rarely violated, occurs check introduces a significant performance penalty, so some logical languages (such as Prolog) omit it.

In this section, we show how the complexity of unification can be assessed for many practical cases. Specifically, we present two dynamic criteria for relational programs, under which every unification (omitting occurs check) in the program performs a constant number of basic operations. At the same time, the occurs check, which complexity can be estimated separately, adds a significant overhead to the execution time and often increases the asymptotic complexity. A number of programs satisfying given tests and showing the impact of occurs check are listed in section 6.

The actual time of unification depends on a concrete choice for a data structure to represent triangular substitutions (which are, abstractly, maps from integers to terms). Therefore we parameterize our estimations by two values — $\texttt{lookup}(\sigma)$ and $\texttt{add}(\sigma)$, — which represent, respectively, the worst-case asymptotic complexity for lookup and add operations w.r.t. to a substitution σ. The obvious candidate data structure is standard library maps for a host language (and many implementations like miniKanren-with-symbolic-constraints and OCanren follow this recipe). For this data structure both operations have logarithmic complexity, so we expect this multiplier to be negligible. However, some implementations like microKanren use associative lists for simplicity of presentation (which have linear-time lookup and constant-time addition complexities) or more complex data structures like random-access lists (which have a log-time lookup and average constant-time addition complexities), so we keep this parameterization for the general case. The review of the performance of different date structures for triangular substitutions is given in [3].

The basic building block for the unification with triangular substitution is a *walk* operation. This operation checks whether a given variable is mapped by a given substitution to a term with a constructor at the top level. "Walk" continually looks up the substitution until a binding to a non-variable is found or until there is no binding at all. This process can diverge only when there is a circular binding in the substitution, which, in turn, is excluded by the occurs check, so

the substitutions are always consistent in this sense [8]. Nevertheless "walk" can require a linear number (on the size of substitution) of lookups. However, the variable-to-variable bindings occur rarely in practice, and usually "walk" finds the required binding in one step. We can take the absence of variable-to-variable bindings as our first criterion: for *flat* substitutions (substitutions without such bindings) "walk" always makes only one lookup. We can relax this requirement by allowing a constant number, independent of the input parameters of the topmost goal, of variable-to-variable bindings.

Definition 4.1. A substitution σ is called *constant-factor flat* if the number variable-to-variable bindings in σ does not depend on the input parameters of the topmost goal.

Lemma 4.2. *If during the evaluation of a goal all substitutions are constant-factor flat, then the time of any walk during that evaluation on substitution σ is* $\mathtt{lookup}(\sigma)$.

The unification of two terms goes by the standard recursive descent. Each time a variable is encountered in a term being unified a "walk" step is performed. If it ends up with an unbound variable the occurs check is performed and, if it succeeded, the substitution is extended. As the substitution grows during the process the unified terms grow, too, and the descent can go beyond the size of initial terms. But we argue that this happens not that often. For example, for a linear case (when any variable occurs in unified terms at most once) the extensions of the substitution during the unification do not affect the unification in other branches, so the recursion will stop at the minimal height of the terms.

Lemma 4.3. *Given two terms t_1 and t_2 and a current constant-factor flat substitution σ, if any variable occurs at most once in at most one of the terms $\{t_1\sigma, t_2\sigma\}$, then the time this unification takes, excluding occurs checks, is* $O(\min\{|t_1\sigma|, |t_2\sigma|\}) \cdot (\mathtt{lookup}(\sigma) + \mathtt{add}(\sigma))$.

In particular, if the size of one of those two terms does not depend on the input parameters (which is usually the case) the unification performs a constant number of basic operations. This is our second criterion: linearity and constant size of one of the terms in every unification.

In the presence of occurs checks, however, we need to also go through every term we add in the substitution. This changes "min" in the estimation above to "max", making a huge difference. Roughly speaking, in average the number of basic operations for every unification with occurs checks is approximately an average size of all terms unified in the program (which is usually linear of the input). Therefore occurs checks add a huge time overhead for program execution in miniKanren, both for asymptotics (see section 6) and for observable time [4]. This fact calls for an investigation into ways of going around occurs checks in miniKanren. Although simply omitting them is not an option for miniKanren, there are other known approaches (mostly explored for Prolog), for example, static tests ensuring that occurs checks for a given program will never be violated [2]. As far as we know, for now, there are no such solutions adopted for miniKanren.

For now, as we estimate the time of every individual unification it might be not clear how it relates to the estimations for the scheduling time. But since we consider cases in which unification is relatively fast (constant number of basic operations), the number of unifications during an execution plays a more important role. And the number of unifications can be simply limited by the number of semantic steps $d(s_{init}(input))$ (because every unification is a separate step in the semantics). Similarly, although the time of basic operations depends on the size of substitution in different points of execution, logical variables for these bindings come either from the input (where there is usually a constant number of them) or from fresh variable allocations during the evaluation (each of which is a separate step in the semantics). So the number of allocated logic variables and, therefore, the maximum possible number of bindings are limited by $FV(input) + d(s_{init}(input))$.

```
prefix^o = λ n p →
  (p ≡ Nil) ∨
  (fresh (n' pt pti)
     (n ≡ S(n')) ∧
     (prefix^o n' pt) ∧
     (incr_list^o pt pti) ∧
     (p ≡ Cons(S(O), pti))
  )
```

```
incr_list^o = λ a r →
  ((a ≡ Nil) ∧ (r ≡ Nil)) ∨
  (fresh (h t tr)
     (a ≡ Cons(h, t)) ∧
     (r ≡ Cons(S(h), tr)) ∧
     (incr_list^o t tr)
  )
```

Fig. 6. Relational Prefixes Example

So, for example, for a usual case, when our two criteria are satisfied and the input contains a constant number of logic variables, for a standard implementation and without occurs checks the total time of unifications T_{uni} is $O\left(d\left(s_{init}\left(input\right)\right) \cdot \log d\left(s_{init}\left(input\right)\right)\right)$

The time of reification T_r can be estimated in the same way, since reification simply goes through the resulting term similarly to occur check. So in the case when the resulting substitution is constant-factor flat, the number of basic operations for the reification is limited by the size of the output (multiplied by a constant). This time is usually dominated by the time of unification and scheduling, but not always (see examples in section 6).

5 COMPLEXITY ANALYSIS VIA SYMBOLIC EXECUTION SCHEMES

In the previous sections, we presented some methods to estimate the time complexity for scheduling and unification/reification (for the latter two only for some practical cases) in miniKanren, but they work only for relational search in general, not for a specific relational program. In this section, we show how the latter task can be formulated and how those methods can be combined to solve it using a notion of *symbolic execution*. Specifically, we add symbolic variables to miniKanren and use *symbolic execution schemes* to build recursive inequalities for all components of our performance model. These inequalities then can be solved to provide a symbolic representation for asymptotic estimations.

The application of symbolic execution for time complexity analysis is well-known and was explored for logic programming in particular [6]. Usually, symbolic execution graphs are used to capture all the details of program execution which are significant for performance, and then the standard techniques for time (or other) analysis of rewriting systems are applied. In contrast, we need symbolic execution graphs only as a neat representation of a general scheme of a relational search for a given program and then bring in performance details using *ad hoc* methods described in the previous sections. So we use a restricted version of a graph that corresponds precisely to a body of a relation, not unfolding any relational calls. For this reason, we refer to them as "symbolic execution schemes" rather than "symbolic execution graphs". This also means that we suppose that we know what answers any relational call in the program produces before we start the time complexity analysis.

To present the whole process in a clearer way we will go through it with a specific artificial example, in which almost all important details are presented. The example is a relational program for generating all prefixes of a list $[1, \ldots, n]$ (with numbers represented in Peano encoding with constructors `O` and `S`). Consider the following creative recursive solution: we either take an empty prefix or take any prefix for the same task for $n - 1$ (if $n > 0$), increment all the elements and add 1

at the beginning. The relation $\texttt{prefix}^o$ in Fig. 6, relating a number n to some prefix p, follows this description directly. It uses a straightforwardly implemented relation $\texttt{incr}-\texttt{list}^o$ that increments all numbers in a given list. This relation provides the required results: if we instantiate n with some Peano number and put a free logical variable for p then p will be bound to every prefix exactly once. It is an inefficient solution in many ways, but it is nice for presentation.

Now we want to estimate the time the search will take depending on a number we put as an input. To make our reasoning more precise we introduce the notion of *symbolic variables*, which we will denote with an overline ($\overline{a}, \overline{b}, \ldots$) as opposed to the usual logic variables, which we will denote using a question mark ($a^?, b^?, \ldots$). The symbolic variables can be considered on two levels. At the level of symbolic execution, each symbolic variable in miniKanren stands for some ground term (a term without logic variables inside), but we do not know which term exactly. At the metalevel, where we reason about the complexity of a program, a symbolic variable $\overline{x}$ stands for a representation of some object x from metatheory (it can be a number, a string, or a graph, for example) as a ground term, and we analyze how the program behaves depending on this object or some of its parameters. We will distinguish between these two levels throughout the whole process of complexity analysis. For our example we consider the parametric query $\texttt{prefix}^o\ \overline{k}\ a^?$ with the first parameter instantiated by some number k represented as a ground Peano term and second parameter left as a free logic variable, and ask how much time the search and its different stages will take depending on the value of k.

Our approach estimates the time complexity for some specific relational call with symbolic variables as arguments, not for a relation in general. We name every call we encounter to use these names in our notations throughout the analysis (for example $pref = \texttt{prefix}^o\ \overline{k}\ a^?$). During the analysis we separately compute a number of factors for the query that correspond to components of the overall time of the search in our model: the number of semantic steps $d^{pref}(k)$ and the scheduling factor $t_s^{pref}(k)$, which correspond to the number of semantic steps and the scheduling factor defined in section 3, $t_{uni}^{pref}(k)$, which is the number of basic operations performed during unifications in the execution of the call, excluding basic operations in occurs checks, $t_{occ}^{pref}(k)$, which is the number of basic operations performed during occurs checks, and $t_r^{pref}(k)$, which is the number of basic operations performed during the reification.

To achieve this, we build a symbolic execution scheme, mirroring the body of the examined relation, identify recursive calls, and reconstructing recursive inequalities for all the aforementioned factors by using the estimations described in the previous sections. We put a number of restrictions for the examined relational call for our approach (however, as can be seen from the section 6, the huge variety of real examples satisfy them): the two criteria from section 4 should be satisfied (which we can check using the symbolic execution, too), relations should be in disjunctive normal form.

We need to know also two extra pieces of information to perform the analysis for a given call. First, to know how to proceed after recursive calls we need to know the answers that the calls produce. We describe them by sets of substitutions binding all the logical variables in the query to terms, possibly containing fresh logic variables and symbolic variables, the latter we then specify in the metatheory (for example, $ANS^{pref}(k) = \{[a^? \leftarrow \overline{p}] \mid p \textit{ is a prefix of the list } [1, \ldots, k]\}$). Second, we need to know all information for non-recursive relational calls in the scheme (the values of all the complexity factors, produced answers, whether the requirements are satisfied). So we need to go and analyze these calls using the same approach before we can examine the given call or reuse the information if we have already analyzed relevant calls before. For this reason, we require the absence of mutual recursion in the examined calls (it should be eliminated using standard techniques) and analyze them in the order of topological sorting. For $pref$ call we will

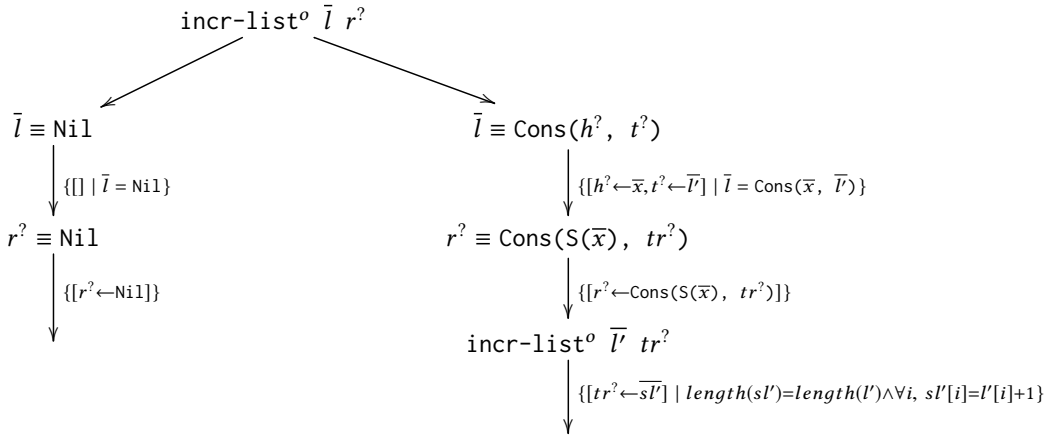

Fig. 7. Symbolic execution scheme for the *incr* call

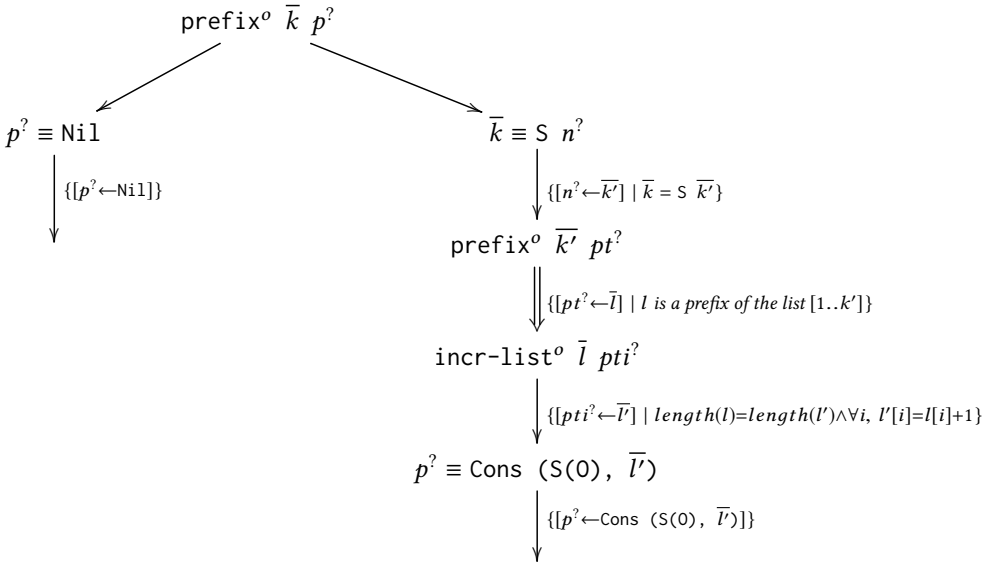

Fig. 8. Symbolic execution scheme for the $pref$ call

need this information for the internal call $incr$ = incr−list^o $\bar{l}$ $r^?$, so we will analyze it first along the way.

The symbolic execution scheme for the $pref$ call is shown in Fig. 8 and the scheme for the internal call *incr* is shown in Fig. 7. A symbolic execution scheme shows unifications and internal relational calls evaluated during the search for the initial call and the answers that are threaded through the search. The initial call is at the top. For simplicity we work only with the relations in disjunctive normal form, each disjunct is represented as a separate column on the scheme. The nodes of the column are the unifications and relational calls in the given conjunct, they are

$U(w^?, w^?)$	$= \epsilon$	
$U(w^?, t)$	$= \bot$	*if* $w^? \in FV(t)$
$U(w^?, t)$	$= [w^? \leftarrow t]$	*if* $t \neq w^? \wedge w^? \notin FV(t)$
$U(t, w^?)$	$= U(w^?, t)$	*if* t *is not a logic variable*
$U(\overline{x}, \overline{y})$	$= [\overline{x} \leftarrow \overline{y}]$	
$U(\overline{x}, f(t_1, \ldots, t_k))$	$= [\overline{x} \leftarrow f(\overline{x_1}, \ldots, \overline{x_k})] \circ U(\overline{x_1}, t_1) \circ \cdots \circ U(\overline{x_k}, t_k)$	*where* $\overline{x_i}$ *are fresh*
$U(f(t_1, \ldots, t_k), \overline{x})$	$= U(\overline{x}, f(t_1, \ldots, t_k))$	
$U(f(t_1, \ldots, t_k), f(t'_1, \ldots, t'_k))$	$= U(t_1, t'_1) \circ \cdots \circ U(t_k, t'_k)$	
$U(f(t_1, \ldots, t_k), g(t'_1, \ldots, t'_{k'}))$	$= \bot$	*if* $f \neq g$

Fig. 9. Unification for terms with logic and symbolic variables; t stands for an arbitrary term.

written down sequentially in the same order as in the relation and connected by arrows. Arrows are labeled with the description of a set of answers, produced by the previous node. This description is represented as a set of lists of bindings for logical variables by which the substitution is extended, the generator of the set (the condition after the '|' symbol) is described in terms of metatheory. For the analysis we need to distinguish cases when multiple answers are produced so we denote by a single arrow $\downarrow$ the sets that we know to have no more than one answer, and put a double arrow $\Downarrow$ in other cases. The answers produced by internal relational calls are given as a prerequisite for the analysis. The unifications may produce new substitutions for both logic variables and symbolic variables. The definition of unification with both logic and symbolic variables is shown on Fig. 9. Bindings for logical variables in the result are extensions of the substitution in the environment after this unification, and bindings for symbolic variables are conditions for the objects in metatheory represented by these symbolic variables, under which we continue to execute the current branch. For example, the unification for $\overline{x} \equiv f(t_1, \ldots, t_k)$ will succeed only for object x such that its representation is $f(\overline{x_1}, \ldots, \overline{x_k})$, where $\overline{x_i}$ are the representations which are the terms unifiable with t_i. So we add bindings for symbolic variables to the generator of the set in the form of equalities. We apply bindings for both logic and symbolic variables in all nodes after we get them to show the fully substituted values of terms.

This scheme presents all the information we need to check the criteria and calculate the complexity of all the factors using the results from the previous sections.

(1) To check that all substitutions are flat during the evaluation we need to know that all non-recursive internal calls satisfy this condition and to check that no variable-to-variable bindings are added during the evaluation of the body of the relation. To check this we can simply check that rhs of all bindings on arrows after unifications are not logical variables (then the value in the binding necessarily has a constructor on the top-level).
If there are no recursive calls in the scheme, we can allow variable-to-variable bindings after substitutions, since there will be at most a constant number of them and substitutions will always be constant-factor flat.
The second criterion (linearity and constant size of one of the terms for every unification) we can easily check directly: for every unification on the scheme each logical variable should occur at most once and one of the terms should have no symbolic variables. We also need the criterion to be satisfied for all the internal calls.
For the calls $incr$ and $pref$ both criteria are satisfied.

(2) To estimate $d^{incr}(l)$ and $d^{pref}(k)$ we use lemmas 3.7 and 3.8. Specifically, we just add the corresponding value for every internal call summed up for all the answers for which the call is executed and also add a constant to handle the rest (unifications and fresh variable

introductions). We know the value of $d^q(\dots)$ for every internal call q: for non-recursive internal calls we have the estimation up to a multiplicative constant from the previous analysis, for recursive internal calls it's just the value of the same function with a different argument.
So, for the *incr* call we have:

$$d^{incr}(l) \le C + \sum_{\overline{l} = \mathsf{Cons}(\overline{x},\ \overline{l'})} d^{incr}(l')$$

Considering two cases when l is an empty and a non-empty list we can simplify the inequality above into the following two:

$$\begin{array}{lcl} d^{incr}([]) & \le & C \\ d^{incr}(x : l') & \le & C + d^{incr}(l') \end{array}$$

which we can easily solve and get $d^{incr}(l) \in O(len(l))$.
And for the $pref$ call we have:

$$d^{pref}(k) \le C + \sum_{\overline{k} = \mathsf{S}\ \overline{k'}} (d^{pref}(k') + \sum_{l \text{ is a prefix of the list } [1..k']} d^{incr}(l))$$

Which we can rewrite and simplify again by considering two cases and substituting calculated complexity for $d^{incr}(l)$:

$$\begin{array}{lcl} d^{pref}(0) & \le & C \\ d^{pref}(k'+1) & \le & C + d^{pref}(k') + \sum\limits_{i \in [0..k']} C \cdot i \\ & \le & d^{pref}(k') + C \cdot k'^2 \end{array}$$

From which we get $d^{pref}(k) \in O(k^3)$.

(3) For $t_s^{incr}(l)$ and $t_s^{pref}(k)$ we do basically the same using the same lemmas 3.7 and 3.8. The difference is that for every internal call q along with $t_s^q(\dots)$ we have to add $d^q(\dots)$ multiplied by a constant (for recursive calls we use the complexity calculated at the previous step). There is a possible exception, however (identified in the lemmas): for one column that has only single arrows, we can omit additional $d^q(\dots)$ for the last call in the column (if the column ends with a call). By lemma 3.8 we can pick any column, it might make difference only when this value $d^q(\dots)$ dominates all the other values. In particular, in the case when a relation has one recurisve call, if it is in the end of a conjunction we omit additional value $d^q(\dots)$ for this call, and when it is not in the end we can not omit this additional value (and can omit the additional value $d^q(\dots)$ for the last non-recursive call instead, which will be dominated by the value $t^q(\dots)$ for this call with a multiplicative factor anyway). This omission is precisely the reason for the change in complexity when the recursive call is moved to the end, like in the initial example of $\texttt{length}^o$ and $\texttt{length}_d^o$ relations in section 1. Likewise, for *incr* call we omit additional value $d^{incr}(l')$ for the recursive call and get the same inequality as the one for the number of steps:

$$t_s^{incr}(l) \le C + \sum_{\overline{l} = \mathsf{Cons}(\overline{x},\ \overline{l'})} t_s^{incr}(l')$$

So, $t_s^{incr}(l)$ is also in $O(len(l))$.

In contrast, in $pref$ call the recursive call is not in the end, so we have additional value $d^{pref}(k')$ for it, which affects the resulting complexity:

$$t_s^{pref}(k) \leq C + \sum_{\overline{k} = \mathtt{S}\ \overline{k'}} (\quad t_s^{pref}(k') + C \cdot d^{pref}(k') + \sum_{l \text{ is a prefix of the list } [1..k']} (t_s^{incr}(l) + C \cdot d^{incr}(l)))$$

After the simplification, we get the following two inequalities:

$$\begin{aligned} t_s^{pref}(0) &\leq C \\ t_s^{pref}(k'+1) &\leq C + t_s^{pref}(k') + C \cdot k'^3 + \sum_{i \in [0..k']} (C \cdot i + C \cdot i) \\ &\leq t_s^{pref}(k') + C \cdot k'^3 \end{aligned}$$

And after solving them we get $t_s^{pref}(k) \in O(k^4)$.

(4) To estimate $t_{uni}^{incr}(l)$ and $t_{uni}^{pref}(k)$ we just do the same summation, counting the number of unifications in the scheme and in the internal calls.
For *incr* we have the following inequality:

$$t_{uni}^{incr}(l) \leq 1 + (\sum_{\overline{l} = \mathtt{Nil}} 1) + 1 + \sum_{\overline{l} = \mathtt{Cons}(\overline{x},\ \overline{l'})} (1 + t_{uni}^{incr}(l'))$$

The simplified version is the following:

$$\begin{aligned} t_{uni}^{incr}([]) &\leq C \\ t_{uni}^{incr}(x : l') &\leq C + t_{uni}^{incr}(l') \end{aligned}$$

And the result is $t_{uni}^{incr}(l) \in O(len(l))$.
And for $pref$ we have the following inequality:

$$t_{uni}^{pref}(k) \leq 1+1+ \sum_{\overline{k} = \mathtt{S}\ \overline{k'}} (t_{uni}^{pref}(k') + \sum_{l \text{ is a prefix of the list } [1..k']} (t_{uni}^{incr}(l) + \sum_{l' :\ length(l) = length(l') \wedge \forall i,\ l'[i] = l[i]+1} 1))$$

The simplified version is the following:

$$\begin{aligned} t_{uni}^{pref}(0) &\leq C \\ t_{uni}^{pref}(k'+1) &\leq C + t_{uni}^{pref}(k') + \sum_{i \in [0..k']} (C \cdot i + 1) \\ &\leq t_{uni}^{pref}(k') + C \cdot k'^2 \end{aligned}$$

And the result is $t_{uni}^{pref}(k) \in O(k^3)$.

(5) To estimate $t_{occ}^{incr}(l)$ and $t_{occ}^{pref}(k)$ we just do the same summation, counting the sizes of rhs in bindings on arrows after every unification on the scheme and the same in the internal calls.
For *incr* we have the following inequality:

$$t_{occ}^{incr}(l) \leq (\sum_{\overline{l} = \mathtt{Nil}} |\mathtt{Nil}|) + \sum_{\overline{l} = \mathtt{Cons}(\overline{x},\ \overline{l'})} (|\overline{x}| + size(l') + |\mathtt{Cons(S(\overline{x}),\ tr)}| + t_{occ}^{incr}(l')),$$

where $size\,(l') = \sum_{y \in l'} |\overline{y}|$.

The simplified version is the following:

$$\begin{array}{rcl} t^{incr}_{uni}\,([]) & \leq & C \\ t^{incr}_{uni}\,(x : l') & \leq & C + 2|\overline{x}| + size(l') + t^{incr}_{uni}\,(l') \end{array}$$

And the result is $t^{incr}_{uni}\,(l) \in O\,(len\,(l) \cdot size\,(l))$.

And for $pref$ we have the following inequality:

$$t^{pref}_{occ}\,(k) \leq C + \sum_{\overline{k} = \mathtt{S}\ \overline{k'}} (k' + t^{pref}_{occ}\,(k') + \sum_{l \text{ is a prefix of the list } [1..k']} (t^{incr}_{occ}\,(l) + \sum_{l' \colon length(l) = length(l') \wedge \forall i,\ l'[i] = l[i]+1} |\mathtt{Cons\ (S(0),}\ \overline{l'}\mathtt{)}|))$$

The simplified version is the following.

$$\begin{array}{rcl} t^{pref}_{occ}\,(0) & \leq & C \\ t^{pref}_{occ}\,(k'+1) & \leq & C + k' + t^{pref}_{occ}\,(k') + \sum_{i \in [0..k']} (C \cdot i^3 + C \cdot i^2 + C) \\ & \leq & t^{pref}_{occ}\,(k') + C \cdot k'^4 \end{array}$$

And the result is $t^{pref}_{occ}\,(k) \in O\,(k^5)$.

(6) Finally, to estimate $t^{pref}_{r}\,(k)$ we just sum the sizes of all answers from $ANS^{pref}\,(k)$.

$t^{pref}_{r}\,(k) = \sum_{l \text{ is a prefix of the list } [1..k]} size(l) \leq \sum_{i \in [0..k]} C \cdot i^2 \leq C \cdot k^3$

So $t^{pref}_{r}\,(k) \in O\,(k^3)$.

This way we get the complexity for all the components of the search. Now we can combine them to get the complete estimation. To get the time related to the unification we should multiply t^{pref}_{uni}, t^{pref}_{occ}, t^{pref}_{r} by a multiplier ($\mathtt{lookup}\,(|\sigma|) + \mathtt{add}\,(|\sigma|)$) which we can estimate by ($\mathtt{lookup}\,(d^{pref}\,(k)) + \mathtt{add}\,(d^{pref}\,(k))$). So, for example, for an implementation with standard-library maps for substitutions the complete time of the search for the call $\mathtt{prefix}^o\ \overline{k}\ a^?$ is $O\,(k^4 + k^3 \log k + k^5 \log k + k^3 \log k) = O\,(k^5 \log k)$ with occurs checks and $O\,(k^4 + k^3 \log k + k^3 \log k) = O\,(k^4)$ without.

6 EVALUATION

We have applied the described approach of time complexity analysis to a number of standard miniKanren relations. Specifically, we have analyzed relational operations for basic data types: for Peano numbers (addition, multiplication, comparison) and lists (length, map, append, reverse), the definitions can be found in appendix A. These are well-known relations with simple declarative recursive definitions, often used as examples of relational programming, yet we could not find any formal analysis of their time complexity. All the examples we studied satisfy the requirements of our method and the extracted recursive inequalities are easily solvable, which supports the claim that our method, although not universal, is adequately applicable in practice.

For every relation, we analyzed several queries (specifying different arguments with symbolic variables) corresponding to different reasonable usages (for example, we examined usages of addition relation for addition, subtraction, and decomposition of a number into a sum of two numbers). For every query, we took the known optimal order of conjuncts in the relation (as the time of the search hugely depends on this order [4]), which may be different for different queries. We could perform the analysis with other orders the same way, but the results are not that useful

Query	t_s	t_{uni}	t_{occ}	t_r
$\texttt{le}^o\ \overline{n}\ \overline{m}$	$\min(n, m)$	$\min(n, m)$	nm	0
$\texttt{le}^o\ x^?\ \overline{m}$	m	m	m^2	m^2
$\texttt{le}^o\ \overline{n}\ y^?$	n	n	n^2	n
$\texttt{plus}^o\ \overline{n}\ \overline{m}\ r^?$	n	n	$n^2 + m$	$n + m$
$\texttt{plus}^o\ \overline{n}\ y^?\ \overline{k}$	$\min(n, k)$	$\min(n, k)$	nk	$\max(n - m, 0)$
$\texttt{plus}^o\ x^?\ y^?\ \overline{k}$	k	k	k^2	k^2
$\texttt{mult}^o\ \overline{n}\ \overline{m}\ r^?$	n^2m	nm	$nm^2 + n^2m$	nm
$\texttt{mult}^o\ x^?\ \overline{m+1}\ \overline{k}$	k	k	$k^2 + mk$	$\frac{k}{m}$
$\texttt{mult}^o\ (\texttt{S}\ x^?)\ (\texttt{S}\ y^?)\ \overline{k}$	k^2	k^2	k^3	k^2
$\texttt{length}^o_d\ \overline{l}\ r^?$	$len^2(l)$	$len(l)$	$len(l) \cdot size(l)$	$len(l)$
$\texttt{length}^o\ \overline{l}\ r^?$	$len(l)$	$len(l)$	$len(l) \cdot size(l)$	$len(l)$
$\texttt{length}^o\ a^?\ \overline{n}$	n	n	n^2	n
$\texttt{incr\_list}^o\ \overline{l}\ r^?$	$len(l)$	$len(l)$	$len(l) \cdot size(l)$	$size(l)$
$\texttt{incr\_list}^o\ a^?\ \overline{l}$	$len(l)$	$len(l)$	$len(l) \cdot size(l)$	$size(l)$
$\texttt{append}^o\ \overline{l_1}\ \overline{l_2}\ r^?$	$len(l_1)$	$len(l_1)$	$len(l_1) \cdot size(l_1) + size(l_2)$	$size(l_1) + size(l_2)$
$\texttt{append}^o\ a^?\ b^?\ \overline{l}$	$len(l)$	$len(l)$	$len(l) \cdot size(l)$	$len(l) \cdot size(l)$
$\texttt{reverse}^o\ \overline{l}\ r^?$	$len^3(l)$	$len^2(l)$	$len^2(l) \cdot size(l)$	$size(l)$
$\texttt{reverse}^o\ a^?\ \overline{l}$	$len^2(l)$	$len^2(l)$	$len^2(l) \cdot size(l)$	$size(l)$

Fig. 10. Calculated complexities for the example queries. $len\,(\cdot)$ stands for length of a given list, $size\,(\cdot)$ stands for a total size of representation of a given list (sum of sizes of representations of all elements). Note that the aspects of the search related to unification and reification (t_{uni}, t_{occ}, and t_r) are all measured in the number of basic operations on substitution that take $\texttt{lookup}\,(\sigma)$ + $\texttt{add}\,(\sigma)$ time each.

and in those cases, the search often diverges. The results — the complexities for all the factors characterizing different aspects of the search — are shown in Fig. 10. Note that sometimes the time depends not on the size of the terms substituted for symbolic variables, but on some characteristics of the values these terms represent (in these cases, length of the list). In all cases, the time is polynomial on the size of the input.

From these results, we can infer some conclusions about the factors influencing the time of evaluation in miniKanren.

First, the overhead of occurs checks is immense in terms of time complexity. In all cases, this component dominates all others, usually incrementing the degree of the polynomial. This contrast is shown more clearly in Fig. 11 where the total time of the search for the standard implementation with and without occurs check is given.

Second, we can see that sometimes a rather counter-intuitively running execution "backwards" (specifying the supposed result in a relation instead of supposed arguments) is faster than the intended "forward" execution. For example, the division via multiplication relation is faster than multiplication itself, the list reversing is faster if we specify the result and ask for a suitable argument. If we look closer into these examples we can see that the reason for this is the fact that the optimal order of conjuncts for backward execution has the recursive call at the end while the optimal order of conjuncts for forward execution does not. When the recursive call is not the last it forces us to add the number of semantic steps for this call when calculating the scheduling factor, which often increases the complexity. This supports a well-known rule of thumb in miniKanren which

Query	without occurs checks	with occurs checks
$\texttt{le}^o\ \overline{n}\ \overline{m}$	$min(n,m)\cdot\log min(n,m)$	$nm\cdot\log min(n,m)$
$\texttt{le}^o\ x^?\ \overline{m}$	$m^2\log m$	$m^2\log m$
$\texttt{le}^o\ \overline{n}\ y^?$	$n\log n$	$n^2\log n$
$\texttt{plus}^o\ \overline{n}\ \overline{m}\ r^?$	$(n+m)\log n$	$(n^2+m)\log n$
$\texttt{plus}^o\ \overline{n}\ y^?\ \overline{k}$	$\min(n,k)\cdot\log\min(n,k)$	$nk\cdot\log\min(n,k)$
$\texttt{plus}^o\ x^?\ y^?\ \overline{k}$	$k^2\log k$	$k^2\log k$
$\texttt{mult}^o\ \overline{n}\ \overline{m}\ r^?$	$n^2m\log nm$	$(nm^2+n^2m)\log nm$
$\texttt{mult}^o\ x^?\ \overline{m+1}\ \overline{k}$	$k\log k$	$(k^2+mk)\log k$
$\texttt{mult}^o\ (\texttt{S}\ x^?)\ (\texttt{S}\ y^?)\ \overline{k}$	$k^2\log k$	$k^3\log k$
$\texttt{length}^o_d\ \overline{l}\ r^?$	$len^2(l)\cdot\log len(l)$	$len(l)\cdot size(l)\cdot\log len(l)$
$\texttt{length}^o\ \overline{l}\ r^?$	$len(l)\cdot\log len(l)$	$len(l)\cdot size(l)\cdot\log len(l)$
$\texttt{length}^o\ a^?\ \overline{n}$	$n\log n$	$n^2\log n$
$\texttt{incr\_list}^o\ \overline{l}\ r^?$	$len(l)\cdot\log len(l)$	$len(l)\cdot size(l)\cdot\log len(l)$
$\texttt{incr\_list}^o\ a^?\ \overline{l}$	$len(l)\cdot\log len(l)$	$len(l)\cdot size(l)\cdot\log len(l)$
$\texttt{append}^o\ \overline{l_1}\ \overline{l_2}\ r^?$	$(size(l_1)+size(l_2))\cdot\log len(l_1)$	$(len(l_1)\cdot size(l_1)+size(l_2))\cdot\log len(l_1)$
$\texttt{append}^o\ a^?\ b^?\ \overline{l}$	$len(l)\cdot size(l)\cdot\log len(l_1)$	$len(l)\cdot size(l)\cdot\log len(l)$
$\texttt{reverse}^o\ \overline{l}\ r^?$	$(len^3(l)+size(l))\cdot\log len(l)$	$len^2(l)\cdot size(l)\cdot\log len(l)$
$\texttt{reverse}^o\ a^?\ \overline{l}$	$(len^2(l)+size(l))\cdot\log len(l)$	$len^2(l)\cdot size(l)\cdot\log len(l)$

Fig. 11. Complexities of the total time of the search for the example queries with and without occurs check. The standard implementation of substitution is considered (using standard library maps), so $\log|\sigma|$ is taken as a time of basic operations on substitution. For other implementations of substitutions, this factor should be changed to the appropriate one.

recommends placing recursive calls at the end of conjunctions whenever possible. We can now see that one reason for it is smaller time penalty of scheduling discipline.[1]

To check how well our estimates correspond to the reality we implemented a simple embedding of miniKanren into OCaml and measured the time of the search for the example queries, building graphs for the time vs. parameters in the estimated complexity (distinct plot for each parameter). [2] The referenced embedding follows the standard microKanren implementation but with standard library maps for substitutions and with the possibility to switch occurs check on and off. For time measuring we used standard OCaml library benchmark. The precision we were able to achieve is rather limited, so the plots are not always smooth and can have deviations (especially for small sizes of the input), but the overall trend is usually clear. Because of the problems with precision, for now, we are able to adequately measure only the total time of the search, with or without occurs check, so we are basically verifying only the Fig. 11. The example of the resulting graph is shown in Fig. 12 (the time of addition of two numbers depending on the first argument). The character of a time dependency function is not always visible from the graph (when the degree of the polynomial is greater than one), but after studying each graph individually (for example, by placing the plot

[1]There are other reasons. For example, executing all constraints before possibly diverging relational calls is one of the methods of ensuring termination of relational programs [4].

[2]The implementation and the results of the measuring can be found at https://www.dropbox.com/sh/ciceovnogkeeibz/AAAoclpTSDeY3OMagOBJHNiSa

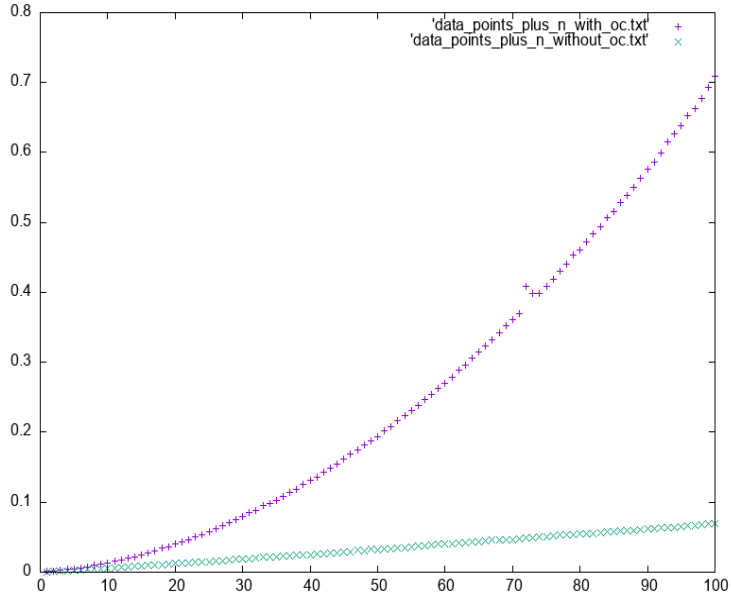

Fig. 12. The graph for the addition query showing the dependency of total time (in seconds) of the search for the query $\texttt{plus}^o\ \overline{n}\,\overline{m}\,r^?$ on the value of n with a fixed m. Purple dots show the time for the case when occurs checks are performed, green dots — for the case when they are not performed.

between two polynomials of the same degree) we are reasonably convinced that all the complexity estimates from the Fig. 11 are confirmed.

7 CONCLUSION

We presented a first attempt to build a theory that would allow one to calculate adequate worst-case time complexity estimations for relational programs in miniKanren. While our research is not completed the current results still allow explaining some observable phenomena in some relational programs behavior.

For now, we confine ourselves to the problem of estimating the time of the full search for a given query. Estimating the time before the first (or some specific) answer is found is an important and probably more practical task. Unfortunately, the described model can not be adjusted naturally to this case. The reason for this is that the reasoning about time (scheduling time in particular) in our terms becomes non-compositional for the case of interrupted search: if the answer is found in some branch, the search is cut short in other branches too. We can still calculate the number of semantic steps for this case, just focusing on one branch in such cases. But for the scheduling factor this will not work, because the size of states can be different in different branches, so the equal number of semantic steps can take different amounts of time in different branches. This picture requires more complicated notions with non-trivial dependencies between them and the model becomes impractical. We are currently looking into ways to tame it.

Among other directions for future research, we can mention relaxing the requirements which we put on terms and substitutions in order to provide accurate unification time estimations. It would be also interesting to come up with an automated (or semi-automated) procedure to deduce the complexity estimations in symbolic forms by making use of the symbolic execution model presented in Section 5.

REFERENCES

[1] Franz Baader and Wayne Snyder. 2001. Unification Theory. In *Handbook of Automated Reasoning (in 2 volumes)*, John Alan Robinson and Andrei Voronkov (Eds.). Elsevier and MIT Press, 445–532. https://doi.org/10.1016/b978-044450813-3/50010-2

[2] Joachim Beer. 1988. The Occur-Check Problem Revisited. *J. Log. Program.* 5, 3 (1988), 243–261. https://doi.org/10.1016/0743-1066(88)90012-X

[3] David Bender, Lindsey Kuper, William Byrd, and Daniel Friedman. 2015. Efficient representations for triangular substitutions: A comparison in miniKanren. (03 2015).

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

[5] Daniel P. Friedman, William E. Byrd, and Oleg Kiselyov. 2005. *The reasoned schemer.* MIT Press.

[6] Jürgen Giesl, Thomas Ströder, Peter Schneider-Kamp, Fabian Emmes, and Carsten Fuhs. 2012. Symbolic evaluation graphs and term rewriting: a general methodology for analyzing logic programs. In *Principles and Practice of Declarative Programming, PPDP'12, Leuven, Belgium - September 19 - 21, 2012*, Danny De Schreye, Gerda Janssens, and Andy King (Eds.). ACM, 1–12. https://doi.org/10.1145/2370776.2370778

[7] Oleg Kiselyov, Chung-chieh Shan, Daniel P. Friedman, and Amr Sabry. 2005. Backtracking, interleaving, and terminating monad transformers: (functional pearl). In *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming, ICFP 2005, Tallinn, Estonia, September 26-28, 2005*, Olivier Danvy and Benjamin C. Pierce (Eds.). ACM, 192–203. https://doi.org/10.1145/1086365.1086390

[8] Ramana Kumar and Michael Norrish. 2010. (Nominal) Unification by Recursive Descent with Triangular Substitutions. In *Interactive Theorem Proving, First International Conference, ITP 2010, Edinburgh, UK, July 11-14, 2010. Proceedings (Lecture Notes in Computer Science, Vol. 6172)*, Matt Kaufmann and Lawrence C. Paulson (Eds.). Springer, 51–66. https://doi.org/10.1007/978-3-642-14052-5_6

[9] John Alan Robinson. 1965. A Machine-Oriented Logic Based on the Resolution Principle. *J. ACM* 12, 1 (1965), 23–41. https://doi.org/10.1145/321250.321253

[10] Dmitry Rozplokhas, Andrey Vyatkin, and Dmitry Boulytchev. 2020. Certified Semantics for Relational Programming. In *Programming Languages and Systems - 18th Asian Symposium, APLAS 2020, Fukuoka, Japan, November 30 - December 2, 2020, Proceedings (Lecture Notes in Computer Science, Vol. 12470)*, Bruno C. d. S. Oliveira (Ed.). Springer, 167–185. https://doi.org/10.1007/978-3-030-64437-6_9

A DEFINITIONS OF THE EVALUATED RELATIONS

Here are the definitions of the miniKanren relations we used for evaluation. Different queries to the same relation may require different orders in conjunctions.

(1) Comparison of Peano numbers
Definition:

```
leᵒ = λ x y →
  (x ≡ O) ∨
  (fresh (x' y')
     (x ≡ S(x')) ∧
     (y ≡ S(y')) ∧
     (leᵒ x' y')
  )
```

For queries:

- $\texttt{le}^o\ \overline{n}\ \overline{m}$
- $\texttt{le}^o\ x^?\ \overline{m}$
- $\texttt{le}^o\ \overline{n}\ y^?$

(2) Sum of Peano numbers

Definition:

```
plus^o = λ x y r →
  ((x ≡ O) ∧ (y ≡ r)) ∨
  (fresh (x' r')
     (x ≡ S(x')) ∧
     (r ≡ S(r')) ∧
     (plus^o x' y r')
  )
```

For queries:

- `plus`^{o} $\overline{n}$ $\overline{m}$ $r^?$
- `plus`^{o} $\overline{n}$ $y^?$ $\overline{k}$
- `plus`^{o} $x^?$ $y^?$ $\overline{k}$

(3) Product of Peano numbers
Definition #1:

```
mult^o = λ x y r →
  ((x ≡ O) ∧ (r ≡ O)) ∨
  (fresh (x' r')
     (x ≡ S(x')) ∧
     (mult^o x' y r') ∧
     (plus^o r' y r)
  )
```

For query:

- `mult`^{o} $\overline{n}$ $\overline{m}$ $r^?$

Definition #2:

```
mult^o = λ x y r →
  ((x ≡ O) ∧ (r ≡ O)) ∨
  (fresh (x' r')
     (x ≡ S(x')) ∧
     (plus^o r' y r) ∧
     (mult^o x' y r')
  )
```

For queries:

- `mult`^{o} $x^?$ $\overline{m+1}$ $\overline{k}$
- `mult`^{o} $\mathtt{S}(x^?)$ $\mathtt{S}(y^?)$ $\overline{k}$

(4) Length of a list
Definition #1:

```
length^o_d = λ a r →
  ((a ≡ Nil) ∧ (x ≡ O)) ∨
  (fresh (h t r')
     (a ≡ Cons(h, t)) ∧
     (length^o_d t r') ∧
     (r ≡ S(r'))
  )
```

For query:

- $\mathtt{length}^o_d$ $\overline{l}$ $r^?$

Definition #2:

```
length^o = λ a r →
  ((a ≡ Nil) ∧ (x ≡ 0)) ∨
  (fresh (h t r')
     (a ≡ Cons(h, t)) ∧
     (r ≡ S(r')) ∧
     (length^o t r')
  )
```

For queries:

- `length`^{o} $\overline{l}$ $r^?$
- `length`^{o} $a^?$ $\overline{n}$

(5) Incrementing all elements in a list
Definition:

```
incr_list^o = λ a r →
  ((a ≡ Nil) ∧ (r ≡ Nil)) ∨
  (fresh (h t tr)
     (a ≡ Cons(h, t)) ∧
     (r ≡ Cons(S(h), tr)) ∧
     (incr_list^o t tr)
  )
```

For queries:

- `incr_list`^{o} $\overline{l}$ $r^?$
- `incr_list`^{o} $a^?$ $\overline{l}$

(6) Concatination of two lists
Definition:

```
append^o = λ a b r →
  ((a ≡ Nil) ∧ (b ≡ r)) ∨
  (fresh (h t tb)
     (a ≡ Cons(h, t)) ∧
     (r ≡ Cons(h, tb)) ∧
     (append^o t b tb)
  )
```

For queries:

- `append`^{o} $\overline{l_1}$ $\overline{l_2}$ $r^?$
- `append`^{o} $a^?$ $b^?$ $\overline{l}$

(7) Inversion of a list
Definition #1:

```
reverse^o = λ a r →
  ((a ≡ Nil) ∧ (r ≡ Nil)) ∨
  (fresh (h t tb)
     (a ≡ Cons(h, t)) ∧
     (reverse^o t tr) ∧
     (append^o tr Cons(h, Nil) r)
  )
```

For query:

- `reverse`^{o} $\overline{l}$ $r^?$

Definition #2:

```
reverse^o = λ a r →
  ((a ≡ Nil) ∧ (r ≡ Nil)) ∨
  (fresh (h t tb)
     (a ≡ Cons(h, t)) ∧
     (append^o tr Cons(h, Nil) r) ∧
     (reverse^o t tr)
  )
```

For query:

```
- reverse^o a^? l̄
```