Wildcard Logic Variables

How to say in miniKanren that natural number is less than 5?

Dmitry Kosarev
Saint Petersburg State University
Russia
Dmitrii.Kosarev@pm.me

Daniil Berezun
JetBrains Research
The Netherlands
daniil.berezun@jetbrains.com

Peter Lozov
Saint Petersburg State University
Russia
lozov.peter@gmail.com

Abstract

We propose a new kind of logic variables – wildcard variables – as a limited form of universal quantification. Combined with disequality constraints they extend the expressive power of OCanren – typed dialect of miniKanren, and enrich subset of OCaml programs that could be automatically converted to relational ones. We also report our progress on applying this idea to a task of synthesizing pattern matching compilation scheme.

CCS Concepts: • **Software and its engineering → Constraint and logic languages**; **Source code generation**.

Keywords: relational programming, relational interpreters, pattern matching, constraint programming

1 Introduction

Relational and logic programming are powerful techniques for enumerating the space of possible answers for a query. Constraints allow us to prune search space and to make path to there right answer short. Some constraints (for example, disequality [Byrd 2009]) are universally applicable, but it's OK to invent new special constraints for specific tasks.

The miniKanren family of languages includes different implementations, both statically and dynamically types. There are some peculiarities in statically typed implementation OCanren [Kosarev and Boulytchev 2016] relatively to "official" implementation [Ballantyne and et al. 2017]. For example, the following result of the query is decent in languages like Scheme, but in statically typed OCaml the result looks weird.

```
; Scheme
> (run 1 (q)  (≠ q #t) (≠ q #f) )
((_.0 (≠ ((_.0 #f)) ((_.0 #t)))))

(* OCaml *)
> run ... (λq → (q ≠ !!true) &&& (q ≠ !!false))
q=_.0 [≠ false; ≠ true];
```

Indeed, in Scheme there are infinitely many possible values for variables that are neither `#t`, nor `#f`. But in OCanren the compiler prohibits unification of variables with non-unifiable types, so variable `q` could be bound in a substitution only to **`true`**, **`false`** or another fresh variable; and expected result of the query is empty stream.

The example above could be repaired by introduction of finite domain constraints [Alvis et al. 2011], but in case of proper algebraic data types they doesn't help. Let's imagine that we want to express that *a variable holds a list, but couldn't begin from a constructor* `Cons`. The naïve attempt in OCanren, **`fresh`** `(h tl) (q≠ cons h tl)` doesn't give us what we desire. It states that there are some `h` and `tl` such that `q` is not equal, but we expected that fact for any possible `h` and `tl`. This form of universal quantification is currently not expressible in OCanren.

Another example of algebraic data types are Peano numbers. Unification allows us to express that a peano number `q` is greater or equal a constant: `q ≡ S(S(S _ .10))`. But disequality constraints are not powerful enough to express that a number is *less* than a constant.

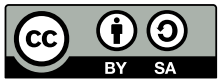
In this paper we introduce *wildcard logic variables* (denoted as `__`) which are able to solve problem like above. The disequality `q ≠ S(S(S __))` states that two values are not equal no matter what we would substitute instead of `__`, which will effectively filter out `S(S(S Z))`, `S(S(S(S Z)))`, i.e. all numbers greater or equal three. This *single* disequality is an only constraint that is required to describe finitely "a peano number is less then constant N". In default implementations of miniKanren we could write a disjunction of three cases but this will hurt performance of the search.

2 Informal Description

In this section we describe the essence of wildcard variables and how they interact with other features of miniKanren.

2.1 Wildcards and Disequality Constraints

Checking and storing disequality constraints may be performed in clever manner [Ballantyne and et al. 2017]. Disequality constraints are stored as a conjunction of disjunctions of pairs (CNF): a fresh variable and a term (or an another variable). While the search progresses, new bindings are propagated to inequalities which allows simplification. For example, if one of the bindings is impossible to be unified, then it is being removed from the disjunction clause. If whole



miniKanren 2022, September 15 2022, Ljubljana, Slovenia

ACM ISBN 978-1-4503-XXXX-X/18/06.

disjunction becomes non-unifiable, then it could be removed from CNF. If a disjunction clause becomes unifiable in any substitution, then it becomes violated and whole CNF too.

In our implementation wildcard variables may occur in program many times. But internally it is a single logic variable (predefined "constant" is some sense), that it treated in a special way. On creation of a disequality constraint we perform unification which gives an updated substitution and a list of recently introduced bindings. Our implementation of wildcard unification doesn't add anything to substitution, but adds new bindings as usual. Our disequality constraints implementation evaluates these bindings and stores in every conjunction clause not only pairs, but also a set of variables that should not be wildcards.

Let's discuss details of checking disequality constraints in presence of wildcards using examples. Below we will use the mantra *"It should be a way to make two values not equal, in spite of we could substitute anything instead of wildcard."* to make decisions about simplification of constraints. We start from simple cases, and leave the complicated cases (a disequality between fresh variable and wildcard) to the end.

Consider the disequality $(1\ \_.10) \not\equiv (\_\_\ 2)$ where either first components of pair should be not equal, or second ones. With first components we are going to consider worst case scenario and substitute a number 1 instead of wildcard. The constraint above simplifies to a shorter disequality between $\_.10$ and 2. All disequalities between ground values and wildcard variable will be simplified immediately.

On disequality of two wildcard variables we again consider worst case scenario: we substitute, for example, 42 instead of both and get a violation of disequality constraints.

Disequality between wildcard and complex value, for example $\texttt{cons}\ 1\ \_.10 \not\equiv \_\_$, could be also simplified. We consider a worst case scenario and use $\texttt{cons}\ \_\_\ \_\_$ instead of wildcard. (But this simplification requires deep understanding of variable's domain, and currently not implemented.)

Disequalities between fresh variables and complex values with wildcards inside are left as they are. Later, we could get more information about fresh variable and simplify the constraint. For example, $(\texttt{cons}\ \_\_\ \_\_ \not\equiv \_.10)\ \&\ (\texttt{cons}\ \_.11\ \_.12 \equiv \_.10)$ simplifies to "either $\_.11$ or $\_.12$ is not wildcard".

The case above leads us to the most complicated case: a disequality between fresh variable and wildcard. If we consider fresh variables as existential ones, the decision may look trivial: if variable q should not be equal wildcard, we will easily violate this constraint by substituting q instead of wildcard. But actually this substitution could be not possible, for example, if variable q has an empty domain. Moreover, we should use hypothesis that any fresh variables have non empty domain, to allow attaching a domain information to variable q later in the search. Without this hypothesis our miniKanren implementation would be less declarative.

2.2 Unification

The primary usage of wildcard variables is in the context of disequality constraints. Despite there is only a single wildcard variable in runtime, we need a special combinator (similar to `call_fresh`) that creates wildcard variable. This is specific to OCanren – typed embedding of miniKanren to OCaml – where we can unify only logic values of the same type. That's why we need many instances of differently typed wildcard variables.

The proposed wildcard variables are not designed to be used in unification, but we decided to allow wildcard syntax `__` in unification anyway. This "wildcards" have a different semantics from proposed in this paper, they looks resembles more traditional wildcards from pattern matching in OCaml: they are just a convenient syntax to avoid manual creation of fresh variables. In the example below we demonstrate two implementations of a relation that extract tail of the list: with wildcard syntax and without them. All occurrences of `__` are translated to calls of primitive combinators by a macro expansion.

```
let tlo xs tl =
  (xs ≡ List.conso __ tl)

let tlo xs tl =
  fresh (h) (xs ≡ List.conso h tl)
```

The macro expansion is responsible for insertion of wildcard variables into right places, but end user could create meaningless OCanren expression bypassing macro expansion. Right now we don't defend against that. But it should be doable if we add for every logic variable a phantom type variable[1] which says whether logic variable can be used in unification, disequality or both.

3 Related Works

The described approach with wildcard logic variable strongly reminds a form of universal quantification. Nowadays, the "official" implementation [Ballantyne and et al. 2017] of miniKanren in Scheme don't yet support any form of universal quantification. In this section we will observe two approaches two universal quantification, and two areas where our wildcard logic variables could be helpful.

3.1 Eigen

One of the form of universal quantification are eigen variables [Byrd 2013]. This approach allows to introduce new fresh named existential variables as usual, and new eigen named variables, which are unifiable with themselves and with fresh variables introduced in their scope. The primary purpose of eigen variables is synthesis a fixpoint combinator in combinatory logic, and this task is being solved without any usage of disequality or other constraints. The interaction

[1] Behavioural types by KC Sivaramakrishnan (accessed: August 29, 2022)

between eigen variables and constraints is not handled by the implementation of eigen dialect of miniKanren. During personal interactions over email we are told that the performance of current implementation of eigen and disequality constraints is troublesome, because it seems to be required to recheck possible equalities between all introduced logic variables and eigen variables.

The one could consider our approach as a simplified form of eigen variables. We don't attach any names to our wildcard variables, so rechecking all possible pairs of inequalities doesn't make much sense. This makes out approach less expressive than eigen but definitely improves search termination. Say, from one point of view the following goal can't be expressed via wildcard variables.

```
(run 1 (q) (fresh (a b) (eigen (x)
    (≠ `(,x ,x) `(,a ,b))) (== a 7) (== b 7)))
```

From another point of view, this goal diverges since it calculation collects all disequality constraints and then simplifies process them once.

Next, eigen variable may occur before fresh variables while it is not the case with wildcards. Consider the following example with eigen variable:

```
(run 1 (q) (eigen (x) (fresh (y) (≠ x y))))
```

It succeeds since whatever for any given x, one can choose a y distinct from x. While wildcarded expression

```
(run 1 (q) (fresh (y) (__ ≠ y)))
```

fails as well as its eigen equivalent

```
(run 1 (q) (fresh (y) (eigen (x) (≠ x y)))).
```

Also, our wildcard variables should be used in disequality constraints. Antagonistically, the implementation of eigen variables uses them in unifications. The detailed comparison of expressiveness of wildcards and eigen variables in disequality constraint requires further studying.

3.2 Universal Quantification and Implication

An interesting idea from [Jin et al. 2021] is to mine examples from the domain of universally quantified variable one by one, cut these points from the domain and wait until it becomes empty. This approach is promising for finite domains, but for recursively described ones it could lead to divergence. We are looking forward for upgraded implementations of the approach, to check it out for tasks that are important for us.

3.3 noCanren

Writing relational programs well requires gaining some skill. The tempting idea is to generate relational programs from functional ones. A subset of OCaml could be automatically converted [Lozov et al. 2018] to OCanren, but current language restrictions make the usage of it inconvenient. For example, to have a decent semantics of a relation programs, it's required for every pattern matching in functional program that all it's branches are non-overlapping. This shortcoming exists because of normal disequality constraints are not powerful enough to express desired result, and the manual process of making pattern non-overlapping leads to exponential increasing of a number of patterns. We believe that adding wildcards is a step to solve this embarrassment (section 4).

3.4 Relational Synthesis of Pattern Matching

The state of art approach to compile pattern matching in OCaml to intermediate representation is translation to backtracking automaton [Le Fessant and Maranget 2001]. In theory we could implement a relational interpreter of intermediate representation, and synthesize a compilation scheme that behave on a pack of examples as we desire [Kosarev et al. 2020]. In practice, a number of examples is finite but large. It depends more on a number of inhabitants of scrutinee's type until certain depth, than on a number of patterns in pattern matching. Ideally, we want to have for exhaustive pattern matching as many examples as we have branches. The wildcard variables allow us to make a step in that direction. But they we are currently far away from finishing that task, because of hidden complications of the task. We describe our progress in section 5.

4 noCanren

Translation of pattern matching from functional program to relational **conde** clauses could be non-trivial if a few branches overlap. For a functional program from Figure 1a1 one could try a straightforward encoding without any constraints:

```
let straightforward q rez =
  conde
    [ fresh ()
        (rez ≡ !!1)
        (q ≡ Std.pair !!true __)
    ; fresh (l r)
        (rez ≡ !!2)
        (q ≡ Std.pair l r)
    ]
```

Unfortunately, this encoding to relational program will have a different semantics from functional one. If scrutinee is a pair of **true**'s, the stream of answers will have both (rez≡1) and (rez≡2) from the first and the second branches of **conde**, respectively. The first answer is expected, but the second one contradicts behaviour of the functional program: the second branch should not be tested if previous one fits the scrutinee. This issue appears only if branches overlap, and to get a proper semantics the approach of relational conversion [Lozov et al. 2018] and associated tool[2] decided to

[2] https://github.com/Lozov-Petr/noCanren (accessed: August 1st, 2022)

```
match x, y with
| T, _ → 1
| _, _ → 2
```

(a1) Pattern matching

```
if x
then 1
else 2
```

(a2) Optimal implementation

(a) Simple patterns matching example

```
match x, y, z with
| _, F, T → 1
| F, T, _ → 2
| _, _, F → 3
| _, _, T → 4
```

(b1) Pattern matching

```
if y then
  if x then
    if z then 4 else 3
  else 2
else
  if z then 1 else 3
```

(b2) Optimal implementation

(b) Pattern matching compilation example (from [Le Fessant and Maranget 2001])

```
let naive_rel q rez =
  conde
    [ fresh (tmp)
        (q ≡ pair !!true tmp)
        (rez ≡ !!1)
    ; fresh (tmp l r)
        (q ≠ pair !!true tmp)
        (q ≡ pair l r)
        (rez ≡ !!2)
    ]
```

(c) Naïve compilation with disequality constraints

```
let better_rel q rez =
  conde
    [ fresh ()
        (q ≡ pair !!true __)
        (rez ≡ !!1)
    ; fresh ()
        (q ≠ pair !!true __)
        (rez ≡ !!2)
    ]
```

(d) Better compilation with wildcards

Figure 1. Pattern matching compilation example

```
let enc_with_diseq q rez =
  let _T = !!true in
  let _F = !!false in
  let w = Std.triple in
  conde
    [ fresh (fresh1) (rez ≡ !!1)
        (q ≡ w fresh1 _F _T)
    ; fresh (fresh1 x) (rez ≡ !!2)
        (q ≡ w _F _T fresh1)
        (q ≠ w x _F _T)
    ; fresh (fresh1 fresh2 x y z) (rez ≡ !!3)
        (q ≡ w fresh1 fresh2 _F)
        (q ≠ w x _F _T)
        (q ≠ w _F _T z)
    ; fresh (x y z fresh1 fresh2) (rez ≡ !!4)
        (q ≠ w x _F _T)
        (q ≠ w _F _T z)
        (q ≠ w x y _F)
        (q ≡ w fresh1 fresh2 _T) ]
```

(a) With disequality constraints

```
let enc_with_wc q rez =
    let _T = !!true in
    let _F = !!false in
    let w = Std.triple in
    conde
      [ fresh () (rez ≡ !!1)
          (q ≡ w __ _F _T)
      ; fresh () (rez ≡ !!2)
          (q ≡ w _F _T __)
          (q ≠ w __ _F _T)
      ; fresh () (rez ≡ !!3)
          (q ≡ w __ __ _F)
          (q ≠ w __ _F _T)
          (q ≠ w _F _T __)
      ; fresh () (rez ≡ !!4)
          (q ≠ w __ _F _T)
          (q ≠ w _F _T __)
          (q ≠ w __ __ _F)
          (q ≡ w __ __ _T) ]
```

(b) With wildcards

Figure 2. Two possible encodings of an example from Fig. 1b

```
1  q=((_.13, false, true), 1);
2  q=((false, true, _.15), 2);
3  q=((_.13, _.14, false), 3);
4  q=((_.13 [≠ false; ≠ _.22], _.14, true), 4);
5  q=((_.13 [≠ false], _.14, false), 3);
6  q=((_.13 [≠ _.22], _.14, true), 4);
7  q=((_.13, _.14 [≠ true], false), 3);
8  q=((_.13 [≠ _.22], _.14 [≠ true], true), 4);
9  q=((_.13 [≠ false], _.14 [≠ false], true), 4);
10 q=((_.13, _.14 [≠ false], true), 4);
11 q=((_.13, _.14 [≠ false; ≠ true], true), 4);
```

(a) With disequality constraints

```
q=((_.13, false, true), 1);
q=((false, true, _.15), 2);

q=((_.13 [≠ false], _.14, false), 3);

q=((_.13, _.14 [≠ true], false), 3);
q=((_.13 [≠ false], _.14 [≠ false], true), 4);

q=((_.13, _.14 [≠ false; ≠ true], true), 4);
```

(b) With wildcards

Figure 3. The output of running two encodings from Figure 2 where scrutinee is a triple of three fresh variables (_ .13, _ .14, _ .15) . One can observe that disequality constraints generate more (bogus) answers. Also, the last answer raises the idea that finite domain constraints may be useful for that example.

forbid overlapping branches and not to use any constraints in generated code. The developer needs to manually rewrite branches of pattern matching, which is annoying and could lead to exponential increasing of the number of branches.

For an example shown in Figure 1a we also provide encodings (Figure 1) with default disequality constraints and with wildcards.

An execution of this pattern matching on four possible ground scrutinees, encoding using wildcards provides four expected results: if first element of pair is **true**, return 1; on **false** – 2. Running naïve compilation scheme (Figure 1c) discovers two more answers: pairs (**true**,**false**) and (**true**,**true**) may return 2. Indeed, two branches of conde are not ordered, and running this scrutinees will not fail a disequality constraint: it will be simplified to ≠ **true** and ≠ **false** respectively. From this example one could conclude that disequality constraints are not expressive enough to handle relational conversion of pattern matching.

One also could try to encode a more complicated example from [Le Fessant and Maranget 2001] with four branches and a scrutinee being a triple of booleans (Figure 1b). In Figure 2 one could observe relational encoding of this matching, and in Figure 3 the result of querying where a scrutinee is a triple of three fresh variables. One could see that all answers returned by "wildcardful" relation are also returned by a relation with disequality constraints. Extra answers on the left should be considered bogus. For example, answer in line 3 about branch 3 is incorrect: the described scrutinee should be already matched by branch 2. Also, answers in lines 8 and 10 should subsume each other. Finally, the last answer demonstrates the need of finite domain constraints which allows considering disequality constraint containing ambivalent information such as [≠ **false**; ≠ **true**] violated.

The above demonstrates how *wildcard* usage allows one to compile pattern matching in a very elegant way without the need of disjoint **conde**-s.

5 Synthesis of Pattern Matching

In this section we briefly describe the task of synthesis of pattern matching compilation scheme [Kosarev et al. 2020] and how wildcard patterns may improve the situation.

The pattern matching expression match a scrutinee v from a set of values $\mathcal{V}$ of algebraic data types with a finite set of patterns $\mathcal{P}$, and produces an integer – index of the pattern, such that it matches provided scrutinee and the ones before it doesn't. For simplicity we suppose that the set patterns is exhaustive, i.e. it is impossible to provide the scrutinee which doesn't fit any pattern.

$$\begin{array}{rcl} C & = & \{C_1^{k_1}, \ldots, C_n^{k_n}\} \\ \mathcal{V} & = & C\,\mathcal{V}^* \\ \mathcal{P} & = & \_ \mid C\,\mathcal{P}^* \end{array}$$

$$\langle v; p_1, \ldots, p_k \rangle \longrightarrow i, \quad 1 \leqslant i \leqslant k; v \in \mathcal{V}; p_1, \ldots, p_k \in \mathcal{P}$$

For every synthesis task the patterns and indexes are ground. Type information is also available. For every subvalue in scrutinee we know which constructors makes sense to match, it's arities and type information of constructors' arguments.

The relation "$\rightarrow$" gives us a *declarative* semantics of pattern matching. Since we are interested in synthesizing implementations, we need a *programmatical* view on the same problem. Thus, we introduce a language $\mathcal{S}$ (the "switch" language) of test-and-branch constructs, and a evaluator "$\rightarrow_{\mathcal{S}}$" that matches a scrutinee to an integer. In the original paper [Kosarev et al. 2020] this language has built-in

```
1  q=(if S[0] then 4 else (if S[2] then (if S[1] then 2 else 1) else 3));
2  q=(if S[0] then 4 else (if S[2] then (if S[1] then 2 else 1) else (if S[1] then 3 else _.1494)));
3  q=(if S[0] then 4 else (if S[2] then 1 else (if S[1] then 2 else 3)));
4  q=(if S[0] then 4 else (if S[2] then (if S[1] then _.25863 else 1) else (if S[1] then 2 else 3)));
5  q=(if S[0] then 4 else (if S[1] then (if S[2] then 2 else 3) else 1));
6  q=(if S[0] then 4 else (if S[2] then (if S[1] then 2 else 1) else (if S[1] then _.1493 else 3)));
7  q=(if S[0] then 4 else (if S[1] then (if S[2] then 2 else 3) else (if S[2] then 1 else _.35148)));
8  q=(if S[0] then 4 else (if S[1] then 2 else (if S[2] then 1 else 3)));
9  q=(if S[0] then (if S[1] then 4 else 3) else (if S[1] then 2 else 1));
10 q=(if S[0] then (if S[1] then 4 else 3) else (if S[1] then 2 else (if S[2] then 1 else _.99286)));
```

Figure 4. First ten unexpected results while compiling pattern matching from Figure 1b1

switch construction that distinguishes tags (integers) of algebraic constructors. But for now we are only testing our approach on tuples of booleans, so we have only if-then-else construction in our language $\mathcal{S}$, despite that fact that **switch** supercedes **if** – **then** – **else**.

$$
\begin{array}{rcl}
\mathcal{M} & = & \bullet \\
 & | & \mathcal{M}\,[\mathbb{N}] \\
\mathcal{S} & = & \textbf{return}\ \mathbb{N} \\
 & | & \textbf{switch}\ \mathcal{M}\ \textbf{with}\ [C \to \mathcal{S}]^*\ \textbf{otherwise}\ \mathcal{S} \\
 & | & \textbf{if}\ \mathcal{M}\ \textbf{startswith}\ C\ \textbf{then}\ \mathcal{S}\ \textbf{else}\ \mathcal{S}
\end{array}
$$

We can formulate the *pattern matching synthesis problem* as follows: for a given ordered sequence of patterns $p_1, \ldots, p_k$ find a switch program $v \vdash \pi$, such that

$$
\begin{gathered}
\forall v \in \mathcal{V},\ \forall 1 \leqslant i \leqslant n : \langle v;\, p_1, \ldots, p_n \rangle \longrightarrow i \\
\Longleftrightarrow \\
v \vdash \pi \longrightarrow_{\mathcal{S}} i
\end{gathered}
$$

The description above uses universal quantification, and can't be immediately transformed into relational specification, because *recursive* data types may make $\mathcal{V}$ infinite. But there is another observation that makes this synthesis problem representable in miniKanren. For every synthesis task we have a ground set of patterns, and they check any scrutinee only into finite depth. This allows one to cut the set of possible scrutinees until certain depth and replace universal quantification by a finite conjunction. The downside of this encoding is an exponential blowup of search space:

- Increasing amount of constructors in types, increases amount of examples required, which hurts performance.
- Increasing depth of constructors hurts performance, but it is expected.
- Changing number of patterns while preserving the same maximum depth doesn't affect performance at all. This is unexpected.

To reduce the number of required examples, we are going to use wildcard variables to say that scrutinee doesn't fit previous branches. In other words, every branch of pattern matching have a single corresponding example. That example will state via unification that scrutinee fits current branch, and also will state via disequality constraints with wildcards that scrutinee doesn't fit previous branches.

During the search OCanren accumulates inequalities between sub parts of scrutinee. More precisely, every **if** – **then** – **else** may introduce disequality between tags of algebraic constructors. It's rather easy to get into situation described in section 1, where boolean constructor is not equal both **true** and **false**. To get rid of these bogus answers we enhance OCanren by finite domain constraints using Z3 [de Moura and Bjørner 2008] under the hood. Adding this constraints complicates implementation of relational engine, because we can't no longer say two values are not equal because of their corresponding sub values are not equal (the inequality of sub values may contradict finite domain constraints). We speculate that in presence of finite domain constraints it may be better to store disequality constraints as DNF instead of CNF, but for now we use traditional[3] implementation.

Unfortunately, wildcards can't solve the pattern matching synthesis problem themselves in a way one can expect. The result of pattern matching synthesis for program in Figure 1b1 is shown in Figure 4. We do believe that the expected reference answer corresponding to one from Figure 1b2 will be eventually found by OCanren but it is definitely far from being the first. We will demonstrate the main source of such a behaviour by example below.

Consider pattern matching with the only branch `__, false, true` → 1. The first answer produced by the interpreter on this example is `q = 1`. This is absolutely correct answer since on all scrutinees having **false** as the second element and **true** as the third element of a triple it produces 1. In other words, the produced answer should have the same or bigger domain than the expected reference

[3]The recommended efficient way to represent disequality constraints is available online [Ballantyne and et al. 2017]

answer. This can be seen as an analog of conservative approximation of the reference answer. Further interpretation of other branches will make the answer produced on the previous step more precise but still conservative.

Processing the next example the interpreter will figure out *some* scrutinee that satisfies disequality constraints and unifications from the given example. Then it will change the program being synthesized to handle this particular scrutinee. In other words, each interpreter step is some conservative approximation of the reference answer. Thus, it is able to produce the reference answer iff on each step the approximation is precise. This means that the reference answer should be eventually produced by the interpreter but after an unpredictable amount of time.

Finally, the interpreter ends up with a stream of answers (see Figure 4) with each answer being in some sense a conservative approximation of the reference one.

6 Conclusion and Future Work

In the paper we introduced a new kind of logic variables: *wildcard* logic variables. For some extend they could be seen as a rework to eigen variables designed to fit with disequality constraints.

Wildcards allow a nice opportunity to represent answer on queries like "a peano number that is less than constant N" as a single answer with a constraints, instead of enumerating of N answers. Although, this particular use case may be expressed with eigen variables, which requires further investigation. Also, wildcard variables allow to relax restrictions of pattern matching when we do relational conversion from functional programs to relational ones. In the paper we also studied potential application of wildcards to relational compilation of pattern matching. The goal is to significantly reduce a number of required examples for synthesis, but complete solving of this task is an ongoing work.

Experiments with wildcards involve reasoning about domains of logic variables, which should require adding constraints on domains of variables. For now, we only have finite domain constraints implemented via Z3, but on this stage it's not obvious how to implement a careful description of algebraic data types domains. We also haven't tried yet to measure or optimize the performance of relational search with wildcard variables.

While working on implementation of synthesis we came up with idea, that overspecifying existential variables in scrutinee could be overcomed by forbidding matching on these subparts of scrutinee for a concrete example. When relational interpreter meets **if** – **then** – **else** on a forbidden value, it ignores the value of a condition, and expects that branches **then** and **else** should either fail or return the same answer. It's not obvious for us how to do it relationally, but we couldn't manage to implement it in a non-relational manner anyway. We believe that this idea is worth spending some extra time.

References

C. E. Alvis, J. Willcock, K. Carter, W. E. Byrd, and D. Friedman. 2011. cKanren: miniKanren with Constraints. In *Proceedings of the 2011 Annual Workshop on Scheme and Functional Programming*.

M. Ballantyne and et al. 2017. *faster-miniKanren implementation in Scheme*. https://github.com/michaelballantyne/faster-minikanren

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

W. E. Byrd. 2013. Relational Synthesis of Programs. (2013). http://webyrd.net/cl/cl.pdf

L. de Moura and N. Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

E. Jin, G. Rosenblatt, M. Might, and Zhang L. 2021. Universal Quantification and Implication in miniKanren. https://www.cs.toronto.edu/~lczhang/jin_universal2021.pdf

D. Kosarev and D. Boulytchev. 2016. Typed Embedding of a Relational Language in OCaml. 285 (2016), 1–22. https://doi.org/10.4204/EPTCS.285.1

D. Kosarev, P. Lozov, and D. Boulytchev. 2020. Relational Synthesis for Pattern Matching. In *Programming Languages and Systems*, Bruno C. d. S. Oliveira (Ed.). Springer International Publishing, Cham, 293–310.

F. Le Fessant and L. Maranget. 2001. Optimizing Pattern Matching. *SIGPLAN Not.* 36, 10 (Oct. 2001), 26–37. https://doi.org/10.1145/507669.507641

P. Lozov, A. Vyatkin, and D. Boulytchev. 2018. Typed Relational Conversion. In *Trends in Functional Programming*, Meng Wang and Scott Owens (Eds.). Springer International Publishing, Cham, 39–58.

A Simple Wildcard Examples

Table 1. A few example of relational queries involving wildcard variables

Example query	Result
(__ ≠ _ .20)	**success**
(1, __) ≠ (__, 1)	fail
(_ .10, 2, __) ≠ (1, __, 2)	_ .10 ≠ 1
(_ .10, _ .11) ≠ (1, __)	_ .10 ≠ 1
fresh (a b) (q ≡ pair a b) (q ≠ pair !!1 __) (q ≡ pair __ !!1)	q ≡ (a [≠ 1], 1), b ≡ 1