An Empirical Study of Partial Deduction for miniKanren

Ekaterina Verbitskaia[1][0000−1111−2222−3333], Daniil Berezun[1][1111−2222−3333−4444], and Dmitry Boulytchev[1,2][2222−−3333−4444−5555]

[1] JetBrains Research, Saint-Petersburg 197374, Russia
[2] {ekaterina.verbitskaya,daniil.berezun,db}@jetbrains.com
https://research.jetbrains.org/
[3] Saint Petersburg State University, *!!!!*, Saint Petersurg, Russia
dboulytchev@math.spbu.ru

Abstract. We explore partial deduction for miniKanren: a specialization technique aimed at improving the performance of a relation in the given direction. We describe a novel approach to specialization of miniKanren based on partial deduction and supercompilation. On several examples, we demonstrate issues which arise during partial deduction.

1 Introduction

The core feature of the family of relational programming languages miniKanren[4] is their ability to run a program in different directions. Having specified a relation for adding two numbers, one can also compute the subtraction of two numbers or find all pairs of numbers which can be summed up to get the given one. Program synthesis can be done by running *backwards* a relational interpreter for some language. In general, it is possible to create a solver for a recognizer by translating it into miniKanren and running in the appropriate direction [**?**].

The search employed in miniKanren is complete which means that every answer will be found, although it may take a long time. The promise of miniKanren falls short when speaking of performance. The running time of a program in miniKanren is highly unpredictable and varies greatly for different directions. What is even worse, it depends on the order of the relation calls within a program. One order can be good for one direction, but slow down the computation drastically in the other direction.

Specialization or partial evaluation [**?**] is a technique aimed at improving the performance of a program given some information about it beforehand. It may either be a known value of some argument, its structure (i.e. the length of an input list) or, in case of a relational program, — the direction in which it is intended to be run. An earlier paper [**?**] showed that *conjunctive partial deduction* [**?**] can sometimes improve the performance of miniKanren programs.

[4] miniKanren language web site: http://minikanren.org

Unfortunately, it may also not affect the running time of a program or even make it slower.

Control issues in partial deduction of logic programming language Prolog have been studied before [**?**]. The ideas described there are aimed at left-to-right evaluation strategy of Prolog. Since the search in miniKanren is complete, it is safe to reorder some relation calls within the goal for better performance. While sometimes conjunctive partial deduction gives great performance boost, sometimes it does not behave as well as it could have.

In this paper, we show on examples some issues which conjunctive partial deduction faces. We also describe a novel approach to partial deduction of a relational programming language miniKanren. We compare it to the existing specialization algorithms on several programs and discuss why some miniKanren programs run slower after specialization.

2 Related Work

Specialization is an attractive technique aimed to improve the performance of a program if some of its arguments are known statically. It is studied for functional, imperative and logic programing and comes in different forms: partial evaluation [**?**] and partial deduction [**?**], supercompilation [**?**], distillation [**?**] and many more.

The heart of supercompilation-based techniques is *driving* — a symbolic execution of a program through all possible execution paths. The result of driving is so-called *process tree* where nodes correspond to *configurations* which present computation state, for example, a term in case of pure functional programming languages. Each path in the tree corresponds to some concrete program execution. The two main sources for supercompilation optimizations are aggressive information propagation about variable values, equalities, and disequalities and precomputing of all deterministic semantic evaluation steps, i.e. combining of consecutive process tree nodes with no branching, also known as *deforestation* [**?**]. When the tree is constructed, the resulting, or *residual*, program can be extracted from the process tree by the process called *residualization*. Of course, process tree can contain infinite branches. *Whistles* — heuristics to identify possibly infinite branches — are used to ensure termination in supercompilation. If the whistle signalled during the construction of some branch, then something should be done to ensure termination. The most common approaches include either stopping driving the infinite branch completely (no specialization is done in this case and the source code is blindly copied into the residual program) or folding the process tree to a *process graph*. The main instrument to perform such a folding is *generalization*, i.e. abstracting away some computed data about the current term which makes folding possible. One source of infinite branches is consecutive recursive calls to the same function with an accumulating parameter: by unfolding such a call further one can only increase the term size which leads to nontermination. The accumulating parameter can be removed by replacing the call with its generalization. There are several ways to ensure process correctness

and termination, the most common being *homeomorphic embedding* [**?**,**?**] used as a whistle and most-specific generalization of terms.

While supercompilation generally improves the behaviour of input programs and distillation can even provide superlinear speedup, there are no ways to predict the effect of specialization on a given program in the general. What is worse, they rarely consider the residual program efficiency from the target language evaluator point of view. The main optimization source is computing in advance all possible intermediate and statically-known semantics steps at program transformation-time. Other criteria, like the size of the generated program or possible optimizations and execution cost of different language constructions by the target language evaluator, are usually out of consideration [**?**]. It is known that supercompilation may adversely affect GHC optimizations yielding standalone compilation more powerful [**?**,**?**] and cause code explosion [**?**]. Moreover, it may be hard to predict the real speedup of any given program on concrete examples even disregarding problems above because of the complexity of the transformation algorithm. The worst-case for partial evaluation is when all static variables are used in a dynamic context, and there is some advice on how to implement a partial evaluator as well as a target program so that specialization indeed improved its performance [**?**,**?**]. There is lack of research in determining the classes of programs which transformers would definitely speed up.

Conjunctive partial deduction [**?**] makes an effort to provide reasonable control for left-to-right evaluation strategy of Prolog. CPD constructs a tree which models goal evaluation and is similar to SLD-NF tree, then a residual program is generated from this tree. Partial deduction itself resembles driving in supercompilation [**?**]. The specialization is done in two levels of control: the local control determines the shape of the residual programs, while the global control ensures that every relation which can be called in the residual program is indeed defined. The leaves of local control trees become nodes of the global control tree. CPD analyses these nodes at the global level and runs local control for all those which are new.

At the local level, CPD examines a conjunction of atoms by considering each atom one-by-one from left to right. An atom is *unfolded* if it is deemed safe, i.e. a whistle based on homeomorphic does not signal for the atom. When an atom is unfolded, a clause whose head can be unified with the atom is found, and a new node is added into the tree where the atom in the conjunction is replaced with the body of that clause. If there is more than one suitable head, then several branches are added into the tree which corresponds to the disjunction in the residualized program. An adaptation of CPD for the miniKanren programming language is described in [**?**].

The most well-behaved strategy of local control in CPD for Prolog is *deterministic unfolding* [**?**]. An atom is unfolded only if only one suitable clause head exists for it with the one exception: it is allowed to unfold an atom nondeterministically once for one local control tree. This means that if a nondeterministic atom is the leftmost within conjunction, it is most likely to be unfolded and introduce many new relation calls within the conjunction. We be-

lieve this is the core problem with CPD which limits its power when applied to miniKanren. The strategy of unfolding atoms from left to right is reasonable in the context of Prolog because it mimics the way programs in Prolog execute. But it often leads to larger global control trees and, as a result, bigger, less efficient programs. The evaluation result of a miniKanren program does not depend on the order of atoms (relation calls) within a conjunction, thus we believe a better result can be achieved by selecting a relation call which can restrict the number of branches in the tree. We describe our approach which implements this idea in the next section.

3 Non-conjunctive Partial Deduction

In this section, we describe a novel approach to relational programs specialization. This approach draws inspiration from both conjunctive partial deduction and supercompilation. The aim was to create a specialization algorithm which is simpler than conjunctive partial deduction and uses properties of miniKanren to improve the performance of the input programs.

The algorithm pseudocode is shown in Fig. 1. For the sake of brevity and clarity, we provide functions `drive_disj` and `drive_conj` which describe how to process disjunctions and conjunctions respectively. Driving itself is a trivial combination of presented functions (line 2).

A driving process creates a process tree, from which a residual program is later created. The process tree is meant to mimic the execution of the input program. The nodes of the process tree include a *configuration* which describes the state of program evaluation at some point. In our case configuration is a conjunction of relation calls. The substitution computed at each step is also stored in the tree node, although it is not included in the configuration.

Hereafter, we consider all goals and relation bodies to be in *canonical normal form* — a disjunction of conjunctions of either calls or unifications. Moreover, we assume all fresh variables to be introduced into the scope and all unifications to be computed at each step. Those disjuncts in which unifications fail are removed. Other disjuncts take the form of possibly empty conjunction of relation calls accompanied with a substitution computed from unifications. Any miniKanren term can be trivially transformed into the described form. In Fig. 1 function `normalize` is assumed to perform term normalization. The code is omitted for brevity.

There are several core ideas behind this algorithm. The first is to select an arbitrary relation to unfold, not necessarily the leftmost which is safe. The second idea is to use a heuristic which decides if unfolding a relation call can lead to discovery of contradictions between conjuncts which in turn leads to restriction of the answer set at specialization-time (line 14; `heuristically_select_a_call` stands for heuristics combination, see section 3.2 for details). If those contradictions are found, then they are exposed by considering the conjunction as a whole and replacing the selected relation call with the result of its unfolding thus *joining* the conjunction back together instead of using *split* as in CPD (lines 15–

```
1  | ncpd goal = residualize ∘ drive ∘ normalize (goal)
2  | drive     = drive_disj ∪ drive_conj
3  |
4  | drive_disj :: Disjunction → Process_Tree
5  | drive_disj D@(c_1, ..., c_n) = ⋁_{i=1}^{n} t_i ← drive_conj (c_i)
6  |
7  | drive_conj :: (Conjunction, Substitution) → Process_Tree
8  | drive_conj ((r_1, ..., r_n), subst) =
9  |   C@(r_1, ..., r_n) ← propagate_substitution subst on r_1, ..., r_n
10 |   case whistle (C) of
11 |     | instance (C', subst')      ⇒ create_fold_node (C', subst')
12 |     | embedded_but_not_instance ⇒ create_stop_node (C , subst )
13 |     | otherwise ⇒
14 |     | | case heuristically_select_a_call (r_1, ..., r_n) of
15 |     | | | Just r ⇒
16 |     | | | | t ← drive ∘ normalize ∘ unfold (r)
17 |     | | | | if trivial ∘ leafs (t)
18 |     | | | | then
19 |     | | | | | C' ← propagate_substitution (C \ r, extract_substitution (t))
20 |     | | | | | drive C'[r ↦ extract_calls (t)]
21 |     | | | | else
22 |     | | | | | t ⋀ drive (C \ r, subst)
23 |     | | | Nothing ⇒ ⋀_{i=1}^{n} t_i ← drive ∘ normalize ∘ unfold (r_i)
```

Fig. 1. Non-conjunctive Partial Deduction Pseudo Code

22). Joining instead of splitting is why we call our transformer *non-conjunctive* partial deduction. Finally, if the heuristic fails to select a potentially good call, then the conjunction is split into individual calls which are driven in isolation and are never joined (line 23).

When the heuristic selects a call to unfold (line 15), a process tree is constructed for the selected call *in isolation* (line 16). The leaves of the computed tree are examined. If all leaves are either computed substitutions or are instances of some relations accompanied with non-empty substitutions, then the leaves are collected and each of them replaces the considered call in the root conjunction (lines 19–20). If the selected call does not suit the criteria, the results of its unfolding are not propagated onto other relation calls within the conjunction, instead, the next suitable call is selected (line 22). According to the denotational semantics of miniKanren it is safe to compute individual conjuncts in any order, thus it is ok to drive any call and then propagate its results onto the other calls.

This process creates branchings whenever a disjunction is examined (lines 4–5). At each step, we make sure that we do not start driving conjunction which we have already examined. To do this, we check if the current conjunction is a renaming of any other configuration in the tree (line 11). If it is, then we fold the tree by creating a special node which then is residualized into a call to the corresponding relation.

We decided not to perform generalization in this approach in the same fashion as CPD or supercompilation does. Our conjunctions are always split into individual calls and are joined back together only if it is sensible. If the need for generalization arises, i.e. homeomorphic embedding of conjunctions [**?**] is detected, then we immediately stop driving this conjunction (line 12). When

residualizing such conjunction, we just generate a conjunction of calls to the input program before specialization.

3.1 Unfolding

Unfolding in our case is done by substitution of some relation call by its body with simultaneous normalization and computation of unifications. The unfolding itself is straightforward however it is not always clear what to unfold and when to *stop* unfolding. Unfolding in functional programming languages specialization, as well as inlining in imperative one, is usually considered to be safe from the residual program efficiency point of view. It may only lead to code explosion or code duplication which is mostly left to a target program compiler optimization or even is out of consideration at all if a specializer is considered as a standalone tool [**?**].

Unfortunately, this is not the case for the specialization of relational programming language. Unlike functional and imperative, in logic and relational programming language unfolding may easily affect the target program efficiency [**?**]. Unfolding too much may create extra unifications, which is by itself a costly operation, or even introduce duplicated computations by propagating the unfolding results onto neighbouring conjuncts.

There is a fine edge between too much unfolding and not enough unfolding. The former is maybe even worse than the latter. We believe that the following heuristic provides a reasonable approach to unfolding control.

3.2 Heuristic

This heuristic is aimed at selecting the relation call within a conjunction which is both safe to unfold and may lead to discovering contradictions within the conjunction. The unsafe unfolding leads to an uncontrollable increase in the number of relation calls in a conjunction. It is best to first unfold those relation calls which can be fully computed up to substitutions.

We deem every static (non-recursive) conjunct to be safe because they never lead to growth in the number of conjunctions. Those calls which unfold deterministically, meaning there is only one disjunct in the unfolded relation, are also considered to be safe.

Those relation calls which are neither static nor deterministic are examined with what we call the *less-branching* heuristic. It identifies the case when the unfolded relation contains fewer disjuncts than it could possibly have. This means that we found some contradiction, some computations were gotten rid of, and thus the answer set was restricted, which is desirable when unfolding. To compute this heuristic we precompute the maximum possible number of disjuncts in each relation and compare this number with the number of disjuncts when unfolding a concrete relation call. The maximum number of disjuncts is computed by unfolding the body of the relation in which all relation calls were replaced by a unification which always succeeds.

```
1 heuristically_select_a_call :: Conjunction → Maybe Call
2 heuristically_select_a_call C = find heuristic C
3
4 heuristic :: Call → Bool
5 heuristic r = isStatic r || isDeterministic r || isLessBranching r
```

Fig. 2. Heuristic selection pseudocode

The pseudocode describing heuristic is shown in fig. 2. Selecting a good relation call can fail (line 1). The implementation works such that we first select those relation calls which are static, and only if there are none, we proceed to consider deterministic unfoldings and then we search for those which are less branching. We believe this heuristic provides a good balance in unfolding.

4 Evaluation

In our study we compared the CPD adaptation for miniKanren and the new non-conjunctive partial deduction. For some programs we have also employed the branching heuristic instead of the deterministic unfolding in the CPD to check whether it can improve the quality of the specialization. We measured the execution time of each specialized program for some queries and compared it with the execution time of the original program.

We used the following programs to test the specializers on.

- A program to compute a concatenation of three lists.
- A program to compute both the length of the list and its maximal element.
- Two implementations of an evaluator of logic formulas.
- Two implementations of a typechecker for a small expression language.
- A program to compute a unifier of two terms.
- A program to search for paths of a specific length in a graph.

The last two relations are described in [**?**] thus we will only briefly describe them in this paper.

We focused on these particular examples because they are examples of relational interpreters which are natural and observable. In this study we only measured the execution time for sample queries. The measured time was averaged over multiple runs for every query. The queries were run on a laptop running Ubuntu 18.04 with quad core Intel Core i5 2.30GHz CPU and 8 GB of RAM.

intro sentence. Row Original contains the execution time of the original version of the program before any transformations. Row Ecce corresponds to the result of the transformation with Ecce conjunctive partial deduction system. Row Non-CPD — translation by our implementation of nonconjunctive partial deduction. Etalon — the etalon implementation of the same program. CPD — the earlier implementation of the conjunctive partial deduction for miniKanren. Brancing — the earlier implemenctation of CPD for miniKanren with the branching heuristic.

4.1 Concatenation of Three Lists

The relation `doubleAppend`^{o} concatenates three lists by a conjunction of two calls of the `append`^{o} relation (see fig. 1.1). These two calls share a variable — an intermediate list `ts` — which should be treated with care during specialization. Partial deduction does ignore the sharing of the variables which renders the approach to be unproductive for this kind of relations. This is why this relation is particularly well known in the conjunctive partial deduction work.

The first list gets traversed twice to construct the result. First, when the intermidiate list `ts` is constructed during the first `append`^{o} call and then, when `ts` is read during the second call. This double traversal negatively impacts the execution time.

```
let doubleAppend^o xs ys zs res =
  fresh (ts) (
    append^o xs ys ts ∧ append^o ts zs res)

let rec append^o xs ys zs = conde [
  (xs ≡ [] ∧ ys ≡ zs);
  fresh (h t r) (
    xs ≡ (h % t) ∧
    zs ≡ (h % r) ∧
    append^o t ys r)]
```

Listing 1.1. Concatenation of three lists

The better implementation of this relation (see fig. 1.2) does not traverse the first list twice. This etalon implementation is generated with the conjunctive partial deduction.

```
let doubleAppend^o xs ys zs res = conde [
  (xs ≡ [] ∧ append^o ys zs res);
  fresh (h t r) (
    xs ≡ h % t ∧
    res ≡ h % r ∧
    doubleAppend^o t ys zs r)]
```

Listing 1.2. Etalon implementation of concatenation of three lists

We run all translated programs in the forward and the backward direction. The forward direction `doubleAppend`^{o} `x y z` ? concatenates three given lists: the first one of length 700, the second and the third — of length 1. When run in

the backward direction `doubleAppend`^{o} ? ? ? `r` , it searches for the first 100 list triples whose concatenation gives the given list of length 700.

There is no significant difference in the execution time in the backward direction between different specialized versions 1. However the execution time in the backward direction differs: the program generated by Ecce shows the same speedup as the etalon program, while non-conjunctive partial deduction slows the program down. WHY?

	forward	backward
Original	0.0040s	0.0195s
Ecce	0.0031s	0.0201s
Non-CPD	0.0055s	0.0202s
Etalon	0.0031s	0.0203s

Table 1. Evaluation results for doubleAppendo

4.2 Maximum Element and Length of a List

The relation `maxLenght`^{o} computes both the length of the list and its maximum element (see fig. 1.3) by concatenation of two relation calls. This relation traverses the list `xs` twice: first to compute the maximum and then to compute the length of the list. It can be transformed in such a way that the list is only traversed once while computing both components of the result simultaneously. This transformation is called *tupling* and can be achieved with conjunctive partial deduction.

The etalon implementation is shown in fig. 1.4. It uses a recursive relation to compute the length and maximum element simultaneously while traversing the list once.

We measured the execution time of `maxLength`^{o} `lst m l`, where `lst` is a list of Peano numbers from 1 to 100. Note, that the second disjunct in the implementation of both `le`^{o} and `gt`^{o} can never contribute into the result of `maxLength`^{o} execution. Thus among others we compared two etalon implentations: one includes these disjuncts, while the other removes them (see table 2)

	[1..100]
Original	2.286ms
Etalon	4.623ms
Etalon removed	3.133ms
Ecce	1.701ms
Non-CPD	2.578ms

Table 2. Execution time of maxlengtho

```
let maxLength^o xs m l = max^o xs m ∧ length^o xs l
let rec length^o xs l = conde [
  (xs ≡ [] ∧ l ≡ zero);
  (fresh (h t m) (
    xs ≡ h % t ∧ l ≡ succ m ∧ length^o t m))]
let max^o xs m = max_1^o xs zero m
let rec max_1^o xs n m = conde [
  (xs ≡ [] ∧ m ≡ n);
  (fresh (h t) (
    (xs ≡ h % t) ∧
    (conde [
      (le^o h n ↑true ∧ max_1^o t n m);
      (gt^o h n ↑true ∧ max_1^o t h m)]))))]
let rec le^o x y b = conde [
  (x ≡ zero ∧ b ≡ ↑true);
  (fresh (x_1) (x ≡succ x_1 ∧ y ≡ zero ∧ b ≡ ↑false));
  (fresh (x_1 y_1) (x ≡succ x_1 ∧ y ≡succ y_1 ∧ le^o x_1 y_1 b))]
let rec gt^o x y b = conde [
  (x ≡ zero ∧ b ≡ ↑false);
  (fresh (x_1) (x ≡succ x_1 ∧ y ≡ zero ∧ b ≡ ↑false));
  (fresh (x_1 y_1) (x ≡succ x_1 ∧ y ≡succ y_1 ∧ gt^o x_1 y_1 b))]
```

Listing 1.3. Maximum element and length of the list

```
let maxLength^o xs m l = maxLength_1^o xs m zero l
let rec maxLength_1^o xs m n l = conde [
  (xs ≡ [] ∧ m ≡ n ∧ l ≡ zero);
  (fresh (h t l_1)
     (xs ≡ h % t) ∧
     (l ≡ succ l_1) ∧
     (conde [
       (le^o h n ↑true ∧ maxLength_1^o t m n l);
       (gt^o h n ↑true ∧ maxLength_1^o t m h l)]))]
```

Listing 1.4. Etalon implementation of maxlengtho

Surprising enough, the execution time of the etalon program is worse than of the original. Ecce succeeds at improving the program performance, while non-conjunctive partial deduction worsens it a little. why

4.3 Evaluator of Logic Formulas

The relation `eval`^{o} describes an evaluation of a subset of first-order logic formulas in a given substitution. It has 3 arguments. The first argument is a list of boolean values which serves as a substitution. The i-th value of the substitution is the value of the i-th variable. The second argument is a formula with the following abstract syntax. A formula is either a *variable* represented with a Peano

number, a *negation* of a formula, a *conjunction* of two formulas or a *disjunction* of two formulas. The third argument is the value of the formula in the given substitution.

All examples of miniKanren relations in this paper are written in OCanren[5] syntax. We specialize the `eval`^{o} relation to synthesize formulas which evaluate to $\uparrow$**true**[6]. To do so, we run the specializer for the goal with the last argument fixed to $\uparrow$**true**, while the first two arguments remain free variables. Depending on the way the `eval`^{o} is implemented, different specializers generate significantly different residual programs.

The Order of Relation Calls One possible implementation of the evaluator in the syntax of OCanren is presented in listing 1.5. Here the relation `elem`^{o} `subst v res` unifies `res` with the value of the variable `v` in the list `subst`. The relations `and`^{o}, `or`^{o}, and **not**^{o} encode corresponding boolean operations.

```
let rec eval^o subst fm res = conde [
  fresh (x y z v w) (
    (fm ≡ var v ∧ elem^o subst v res);
    (fm ≡ conj x y ∧ eval^o st x v ∧ eval^o st y w ∧ and^o v w res);
    (fm ≡ disj x y ∧ eval^o st x v ∧ eval^o st y w ∧ or^o v w res);
    (fm ≡ neg x ∧ eval^o st x v ∧ not^o v res))]
```

Listing 1.5. Evaluator of formulas with boolean operation last

Note, that the calls to boolean relations `and`^{o}, `or`^{o}, and **not**^{o} are placed last within each conjunction. This poses a challenge to the CPD-based specializers. Conjunctive partial deduction unfolds relation calls from left to right, so when specializing this relation for running backwards (i.e. considering the goal `eval`^{o} `subst fm` $\uparrow$**true**), it fails to propagate the direction data onto recursive calls of `eval`^{o}. Knowing that `res` is $\uparrow$**true**, we can conclude that in the call `and`^{o} `v w res` variables `v` and `w` have to be $\uparrow$**true** as well. There are three possible options for these variables in the call `or`^{o} `v w res` and one for the call **not**^{o}. These variables are used in recursive calls of `eval`^{o} and thus restrict the result of driving them. CPD fails to recognize this, and thus unfolds recursive calls of `eval`^{o} applied to fresh variables. It leads to over-unfolding, big residual programs and less than optimal performance.

The non-conjunctive partial deduction first unfolds those calls which are selected with the heuristic. Since exploring boolean operations makes more sense, they are unfolded before recursive calls of `eval`^{o}. The way non-conjunctive partial deduction treats this program is the same as it treats the other implementation

[5] OCanren: statically typed miniKanren embedding in OCaml. The repository of the project: https://github.com/JetBrains-Research/OCanren

[6] An arrow lifts ordinary values to the logic domain

in which boolean operations are moved to the left, as shown in listing 1.6. This program is easier for CPD to transform which demonstrates how unequal is the behaviour of CPD for similar programs.

```
let rec eval^o subst fm res = conde [
  fresh (x y z v w) (
    (fm ≡ var v ∧ elem^o subst v res);
    (fm ≡ conj x y ∧ and^o v w res ∧eval^o st x v ∧eval^o st y w);
    (fm ≡ disj x y ∧ or^o v w res ∧eval^o st x v ∧eval^o st y w);
    (fm ≡ neg x ∧ not^o v res ∧eval^o st x v))]
```

Listing 1.6. Evaluator of formulas with boolean operation second

Unfolding of Complex Relations Depending on the way a relation is implemented, it may take a different number of driving steps to reach the point when any useful information is derived through its unfolding. Partial deduction tries to unfold every relation call unless it is unsafe, but not all relation calls serve to restrict the search space and thus should be unfolded. In the implementation of `eval`^{o} boolean operations can effectively restrict variables within the conjunctions and should be unfolded until they do. But depending on the way they are implemented, the different number of driving steps should be performed for that. The simplest way to implement these relations is with a table as demonstrated with the implementation of **not**^{o} in listing 1.7. It is enough to unfold such relation calls once to derive useful information about variables.

```
let not^o x y = conde [
    (x ≡ ↑true ∧ y ≡ ↑false;
     x ≡ ↑false ∧ y ≡ ↑true)]
```

Listing 1.7. Implementation of boolean **not** as a table

The other way to implement boolean operations is via one basic boolean relation such as `nand`^{o} which is, in turn, has a table-based implementation (see listing 1.8). It will take several sequential unfoldings to derive that variables `v` and `w` should be ↑**true** when considering a call `and`^{o} `v w` ↑**true** implemented via a basic relation. Non-conjunctive partial deduction drives the selected call until it derives useful substitutions for the variables involved while CPD with deterministic unfolding may fail to do so.

Evaluation Results In our study, we considered two implementations of `eval`^{o}, one we call `plain` and the other — `last`, and compared how specializers behave

```
let not^o x y = nand^o x x y

let or^o x y z = nand^o x x xx ∧nand^o y y yy ∧nand^o xx yy z

let and^o x y z = nand^o x y xy ∧nand^o xy xy z

let nand^o a b c = conde [
  ( a ≡ ↑false ∧ b ≡ ↑false ∧ c ≡ ↑true );
  ( a ≡ ↑false ∧ b ≡ ↑true ∧ c ≡ ↑true );
  ( a ≡ ↑true ∧ b ≡ ↑false ∧ c ≡ ↑true );
  ( a ≡ ↑true ∧ b ≡ ↑true ∧ c ≡ ↑false )]
```

Listing 1.8. Implementation of boolean operation via `nand`

on them. The `plain` relation uses table-based boolean operations and places them further to the left in each conjunction. The relation `last` employs boolean operations implemented via `nand`^{o} and place them at the end of each conjunction. These two programs are complete opposites from the standpoint of CPD. We measured the time necessary to generate 1000 formulas over two variables which evaluate to ↑**true** (averaged over 10 runs).

	last	plain
Original	0.695s	1.426s
CPD	1.721s	0.168s
Ecce	0.205s	0.264s
Non-CPD	0.132s	0.137s
Branching	0.468s	0.179s

Table 3. Evaluation results

4.4 Unification and Path Search

Besides evaluator of logic formulas we also run the transformers on the relation `unify` which searches for a unifier of two terms and a relation `isPath` specialized to search for paths in the graph. These two relations are described in paper [**?**] so we will not go into too many details here.

The `unify` relation was executed to find a unifier of two terms $f(X, X, g(Z, t))$ and $f(g(p, L), Y, Y)$. The original miniKanren program fails to terminate on this goal in 30 seconds. On this example, the most performant is the program generated by CPD (0.352 seconds) while the program generated by adding branching heuristic also fails to terminate in 30 seconds. The non-conjunctive partial deduction shows some improvement with the residual program terminated within 1.947 seconds. While driving this program, the non-conjunctive partial deduction

does too much unfolding which negatively impacts the running time as compared to CPD.

The last test executed `isPath` relation to search for 3 paths of length 7 in the graph with 20 vertices and 30 edges. On this program, non-conjunctive partial deduction showed better transformation results than CPD, although the difference is not that drastic.

All evaluation results are presented in the table **??**. Each column corresponds to the relation being run as described above. The row marked "Original" contains the running time of the original miniKanren relation before specialization, "CPD" and "Non-CPD" correspond to conjunctive and non-conjunctive partial deduction respectively while "Branching" is for the CPD modified with the branching heuristic.

	unify			isPath
	first	second	third	
Original	$0.247 * 10^{-4}s$	$> 30s$ (oum?)	$> 30s$ (oum?)	$23.743s$
CPD	$0.081 * 10^{-4}s$	$0.017s$	$0.343s$	$2.999s$
Non-CPD	$0.084 * 10^{-4}s$	$0.021s$	$2.198s$	$1.786s$
Ecce	$0.128 * 10^{-4}s$	$0.026s$	$1.340s$	$1.292s$
Branching	$0.205 * 10^{-4}s$	$0.101s$	$> 30s$ (oum?)	N/A

Table 4. Evaluation results of unification and path search

4.5 Typechecker-Term Generator

This relation implements a typechecker for a tiny language. Being executed in the backward direction it serves as a generator of terms of the given type. The abstract syntax of the language is presented below. The variables are represented with de Bruijn indices, thus let-binding does not specify which variable is being bound.

$$\begin{aligned} type\ term = \ & BConst\ of\ Bool \mid IConst\ of\ Int \mid Var\ of\ Int \\ & \mid term + term \mid term * term \mid term = term \mid term < term \\ & \mid \underline{let}\ term\ \underline{in}\ term \mid \underline{if}\ term\ \underline{then}\ term\ \underline{else}\ term \end{aligned}$$

The typing rules are straightforward and are presented below. Boolean and integer constants have the corresponding types regardless of the environment. Only terms of type integer can be summed up, multiplied or checked for being less or equal. Any terms of the same type can be checked for equality. If-then-else expression typechecks only if its condition is of type boolean, while both then- and else-branches have the same type. An environment Γ is an ordered list, in which the i-th element is the type of the variable with the i-th de Bruijn index. To typecheck a let-binding, first, the term being bound is typechecked and is

added in the beginning of the environment Γ, and then the body is typechecked in the context of the new environment. Typechecking a variable with the index i boils down to getting a i-th element of the list.

$$\frac{}{\Gamma \vdash IConst\ i : Int} \qquad \frac{}{\Gamma \vdash BConst\ b : Bool}$$

$$\frac{\Gamma \vdash t : Int, \Gamma \vdash s : Int}{\Gamma \vdash t + s : Int} \qquad \frac{\Gamma \vdash t : Int, \Gamma \vdash s : Int}{\Gamma \vdash t * s : Int}$$

$$\frac{\Gamma \vdash t : \tau, \Gamma \vdash s : \tau}{\Gamma \vdash t = s : Bool} \qquad \frac{\Gamma \vdash t : Int, \Gamma \vdash s : Int}{\Gamma \vdash t < s : Bool}$$

$$\frac{\Gamma \vdash v : \tau_v,\ (\tau_v :: \Gamma) \vdash b : \tau}{\Gamma \vdash \underline{let}\ v\ b : \tau} \qquad \frac{}{\Gamma \vdash Var\ v : \tau}\ \Gamma[v] = \tau$$

$$\frac{\Gamma \vdash c : Bool, \Gamma \vdash t : \tau, \Gamma \vdash s : \tau}{\Gamma \vdash \underline{if}\ c\ \underline{then}\ t\ \underline{else}\ s : \tau}$$

We compared two possible implementations of these typing rules. The first one is obtained by unnesting of the functional program as described in [**?**]. The second program is hand-written in OCanren. Each implementation has been transformed with non-conjunctive partial deducer and by Ecce.

We measured the time needed to generate 100 terms of type integer (averaged over 10 iterations). Ecce significantly worsens the execution time for both implementations. Non-conjunctive partial deduction improves the first implementation a little bit, while worsening the performance of the second implementation 3 times.

	Implementation 1	Implementation 2
Original	0.464s	0.057s
Non-CPD	0.448s	0.154s
Ecce	1.518s	2.543s

Table 5. Running time of generating 100 closed terms of type Int

why

5 Conclusion

In this paper, we discussed some issues which arise in partial deduction of a relational programming language miniKanren. We presented a novel approach to partial deduction which uses a heuristic to select the most suitable relation

call to unfold at each step of driving. We compared this approach to the earlier implementation of conjunctive partial deduction. Our evaluation showed that there is still not one good technique which definitively speeds up every relational program.