An Empirical Study of Partial Deduction for MINIKANREN

Ekaterina Verbitskaia
JetBrains Research
Saint Petersburg, Russia
kajigor@gmail.com

Daniil Berezun
Saint Petersburg State University
JetBrains Research
Saint Petersburg, Russia
d.berezun@2009.spbu.ru

Dmitry Boulytchev
Saint Petersburg State University
JetBrains Research
Saint Petersburg, Russia
dboulytchev@math.spbu.ru

We study conjunctive partial deduction, an advanced specialization technique aimed at improving the performance of logic programs, in the context of relational programming language MINIKANREN. We identify a number of issues, caused by MINIKANREN peculiarities, and describe a novel approach to specialization based on partial deduction and supercompilation. The results of the evaluation demonstrate successful specialization of relational interpreters. Although the project is at an early stage, we consider it as the first step towards an efficient optimization framework for MINIKANREN.

1 Introduction

A family of embedded domain-specific languages MINIKANREN[1] implement relational programming — a paradigm closely related to pure logic programming. The minimal core of the language, also known as MICROKANREN, can be implemented in as little as 39 lines of SCHEME [7]. An introduction to the language and some of its extensions in a series of examples can be found in the book The Reasoned Schemer [6]. The formal certified semantics for MINIKANREN is described in [26].

Relational programming is a paradigm based on the idea of describing programs as relations. The core feature of relational programming is the ability to run a program in various directions by executing goals with free variables. The distribution of free variable occurrences determines the direction of relational search. For example, 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. One of the most prominent applications of relational programming amounts to implementing interpreters as relations. By running a relational interpreter for some language *backwards* one can do program synthesis. In general, it is possible to create a solver from a recognizer by translating it into MINIKANREN and running it in the appropriate direction [22].

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 execution time of a program in MINIKANREN is highly unpredictable and varies greatly for various 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 dramatically in the other direction.

Partial evaluation [12] is a technique for specialization, i.e. improving the performance of a program given some information about it beforehand. It may either be a known value of some argument, its structure (e.g. the length of an input list) or, in case of a relational program, the direction in which the relation is intended to be run. An earlier paper [22] has shown that *conjunctive partial deduction* [5] can sometimes improve the performance of MINIKANREN programs. Depending on the particular *control* decisions, it may also not affect the execution time of a program or even make it slower.

[1]MINIKANREN language web site: `http://minikanren.org`. Access date: 28.02.2021

Submitted to:
VPT 2021

$$\begin{array}{rcll}
\mathscr{T} & = & \mathscr{X} \cup \{C_i^{k_i}(t_1,\dots,t_{k_i}) \mid t_j \in \mathscr{T}\} & \text{terms over the set of variables } \mathscr{X} \\
\mathscr{G} & = & \mathscr{T} \equiv \mathscr{T} & \text{unification} \\
 & & \mathscr{G} \wedge \mathscr{G} & \text{conjunction} \\
 & & \mathscr{G} \vee \mathscr{G} & \text{disjunction} \\
 & & \textbf{fresh } \mathscr{X} \,.\, \mathscr{G} & \text{fresh variable introduction} \\
 & & R_i^{k_i}(t_1,\dots,t_{k_i}),\ t_j \in \mathscr{T} & \text{relational symbol invocation} \\
\mathscr{S} & = & \{R_i^{k_i} = \lambda\, x_1^i \dots x_{k_i}^i \,.\, g_i;\}\ g & \text{specification}
\end{array}$$

Figure 1: The syntax of the source language

Control issues in partial deduction of the logic programming language PROLOG have been studied before [18]. Under the left-to-right evaluation strategy of PROLOG, atoms in the right-hand side of a clause cannot be arbitrarily reordered without changing the semantics of a program. In MINIKANREN, the subgoals of a conjunction/disjunction can be freely switched without changing the denotational semantics of the program. This opens yet another possibility for optimization, not taken into account by the approaches initially developed in the context of conventional logic programming.

In this paper, we study issues which conjunctive partial deduction faces being applied for MINIKANREN. We also describe a novel approach to partial deduction for relational programming, *conservative partial deduction*. We implemented this approach and compared it with the existing specialization system (ECCE) for several programs. We report here the results of the comparison and discuss why some MINIKANREN programs run slower after specialization.

2 Background

In this section we provide some background on relational programming and relational interpreters.

2.1 MINIKANREN

This paper considers the minimal relational core of the MINIKANREN language. The syntax of the language is presented in Fig. 1. A specification of the MINIKANREN program consists of a set of *relation definitions* accompanied by a top-level *goal* which plays the role of a query. Goals, being the central syntactic category of the language, can take the form of either a term *unification*, *conjunction* or *disjunction* of goals, a *fresh* syntactic variable introduction, or a relation *call*. We consider the alphabet of constructors $\{C_i^{k_i}\}$ and relational symbols $\{R_i^{k_i}\}$ to be predefined and accompanied with their arities.

The formal semantics of the language is best described in [26]. Here we only briefly introduce the semantics. A stream of substitutions for free variables within the query goal is computed during the execution of a MINIKANREN program. Depending on the kind of the goal, one of the following situations is possible.

1. Term unification $t_1 \equiv t_2$ computes the most-general unification in the context of the current substitution. If it succeeds, the unifier is added into the current substitution and then it is returned as a singleton stream. Otherwise, an empty stream is returned.

2. Introduction of the fresh variable $fresh\ x.g$ allocates a new *semantic* variable, substitutes it for all fresh occurrences of x within g, then evaluates the goal.

```
let rec add^o x y z = conde [
    (x ≡ zero  ∧  y ≡ z);
    (fresh (p) (x ≡ succ p  ∧  add^o p (succ y) z) ) ]

let rec eval^o fm res = conde [fresh (x y xr yr) (
    (fm ≡ num res);
    (eval^o x xr  ∧  eval^o y yr  ∧
      conde [
        (fm ≡ sum x y  ∧  add^o xr yr res);
        (fm ≡ prod x y  ∧  ...);
        ... ] )
```

Listing 1: Evaluator of arithmetic expressions

3. An execution of a relational call $R_i^{k_i}(t_1,\dots,t_{k_i})$ is done by first substuting the terms t_j for the respective formal parameters and then running the resulting goal.
4. When executing a conjunction $g_1 \wedge g_2$, first the goal g_1 is run in the context of the current substitution which results in the stream of substitutions, in each of which g_2 is run. The resulting stream of streams is then concatenated.
5. Disjunction $g_1 \vee g_2$ applies both goals to the current substitution and concatenates the results.

Consider the relation $\texttt{add}^o$ in Listing 1. It defines the relation between three Peano numbers x, y and z, such that $x+y=z$, using the OCANREN language[2]. The keyword **conde** provides syntactic sugar for a disjunction, while **zero** and **succ** are constructors. The query **fresh** (z) ($\texttt{add}^o$ (**succ zero**) (**succ zero**) z) results in the only substitution [z $\mapsto$ **succ** (**succ zero**)], while the query **fresh** (x y) ($\texttt{add}^o$ x y (**succ** (**succ zero**))) executes to three valid substitutions: [x $\mapsto$ **zero**, y $\mapsto$ **succ** (**succ zero**)], [x $\mapsto$ **succ zero**, y $\mapsto$ **succ zero**], [x $\mapsto$ **succ** (**succ zero**), y $\mapsto$ **zero**].

The *interleaving* search [14] is at the core of MINIKANREN. It evaluates disjuncts incrementally, passing control from one to the other. This search strategy is what makes the search in MINIKANREN complete. It also allows for reordering of both disjuncts and conjuncts within a goal which may improve the efficiency of a program. This reordering generally leads to the reordering of the answers computed by a MINIKANREN program. The denotational semantics of MINIKANREN ignores the order of the answers because the search is complete and thus all possible answers will be found eventually.

2.2 Relational Interpreters

The kind of relational programs most interesting to us is relational interpreters. They may be used to solve complex problems such as generating quines [4] or to solve search problems by only implementing programs which check that a solution is correct [22]. The latter application is the focus of our research project thus we provide a brief description of it.

Search problems are notoriously complicated. In fact, they are much more complex than verification — checking that some candidate solution is indeed a solution. The ability of MINIKANREN programs

[2]OCANREN: statically typed MINIKANREN embedding in OCAML. The repository of the project: `https://github.com/JetBrains-Research/OCanren`. Access date: 28.02.2021

to be evaluated in different directions along with the complete semantics of the language allows for automatic generation of a solver from a verifier using relational conversion [23]. Unfortunately, generated relational interpreters are often inefficient, since the conversion introduces a lot of extra unifications and boilerplate. This kind of inefficiency is a prime candidate for specialization.

Consider the relational interpreter `eval`^{o} `fm res` in Listing 1. It evaluates an arithmetic expression `fm` which can take the form of a number (**num** `res`) or a binary expression such as the **sum** `x y` or **prod** `x y`. Running the interpreter backwards synthesizes expressions which evaluate to the given number. For example one possible answer to the query `eval`^{o} `fm` (**succ** (**succ** **zero**)) is **sum** (**num** (**succ** **zero**)) (**sum** (**num** **zero**) (**num** (**succ** **zero**))) .

3 Related Work

Specialization is an attractive technique aimed to improve the performance of a program making use of its static properties such as known arguments or its environment. Specialization is studied for functional, imperative, and logic programing and comes in different forms: partial evaluation [12] and partial deduction [21], supercompilation [27], distillation [10], and many others.

The heart of supercompilation-based techniques is *driving* — a symbolic execution of a program through all possible execution paths. The result of driving is a possibly infinite *process tree* where nodes correspond to *configurations* which represent computation states. For example, in the case of pure functional programming languages, the computational state might be a term. Each path in the tree corresponds to some concrete program execution. The two main sources for supercompilation optimizations are aggressive information propagation about variables' values, equalities and disequalities, and precomputing of deterministic semantic evaluation steps. The latter process, also known as *deforestation* [33], means combining of consecutive process tree nodes with no branching. When the tree is constructed, the resulting, or *residual*, program can be extracted from the process tree by the process called *residualization*. Of course, the process tree can contain infinite branches. *Whistles* — heuristics to identify possibly infinite branches — are used to ensure supercompilation termination. If a whistle signals during the construction of some branch, then something should be done to ensure termination. The most common approaches are either to stop driving the infinite branch completely (no specialization is done in this case and the source code is blindly copied into the residual program) or to fold the process tree to a *process graph*. The main instrument to perform such a folding is some form of *generalization*. Generalization, abstracting away some computed data about the current term, 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 correctness and termination of a program transformer [28], most-specific generalization (anti-unification) and *homeomorphic embedding* [11, 15] as a whistle being common.

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 general. What is worse, the efficiency of a residual program from the target language evaluator point of view is rarely considered in the literature. 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 [12]. Partial evaluation in logic programming should be done with care to not interfere with the compiler optimizations [29]. It is also known that supercompilation may adversely affect GHC optimizations making standalone compilation more powerful [2, 13] and cause code explosion [24]. Moreover, it may be hard to predict the real speedup of any given program using concrete benchmarks even disregarding the 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 improves its performance [12, 3]. There is a lack of research in determining the classes of programs which transformers would definitely speed up.

Conjunctive partial deduction [5] makes an effort to provide reasonable control for the left-to-right evaluation strategy of PROLOG. CPD constructs a tree which models goal evaluation and is similar to an SLDNF tree; then a residual program is generated from the tree. Partial deduction itself resembles driving in supercompilation [9]. 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 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 embedding 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 [22].

ECCE partial deduction system [17] is the most mature implementation of CPD for PROLOG. ECCE provides various implementations of both local and global control as well as several degrees of post-processing. Unfortunately there is no automatic procedure to choose what control setting is likely to improve input programs the most. The choice of the proper control is left to the user.

An empirical study has shown that the most well-behaved strategy of local control in CPD for PROLOG is *deterministic unfolding* [16]. An atom is unfolded only if precisely one suitable clause head exists for it with the one exception: it is allowed to unfold an atom non-deterministically once for each local control tree. This means that if a non-deterministic atom is the leftmost one within a conjunction, it is most likely to be unfolded, introducing many new relation calls within the conjunction. We believe this is the core problem of 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. Special care should be taken when unfolding non-leftmost atoms in PROLOG: one should ensure that it does not duplicate code, as well as that no side-effects are done out of order [1, 20]. However in MINIKANREN leftmost unfolding often leads to larger global control trees and, as a result, bigger, less efficient programs. On the contrary, according to the denotational semantics, the results of evaluation of a MINIKANREN program do not depend on the order of relation calls (atoms) within conjunctions, 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.

```
1  | conspd goal = residualize ∘ drive ∘ normalize (goal)
2  | drive       = drive_disj
3  |
4  | drive_disj :: Disjunction → Process_Tree
5  | drive_disj (c_1, ..., c_n) = $\bigvee_{i=1}^{n}$ 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 onto 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 $\bigwedge$ drive (C \ r, subst)
23 |    | | | Nothing ⇒ $\bigwedge_{i=1}^{n}$ t_i ← drive ∘ normalize ∘ unfold (r_i)
```

Figure 2: Conservative partial deduction pseudo code

4 Conservative Partial Deduction

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

The algorithm pseudocode is shown in Fig. 2. We use the following notation in the pseudocode. The circle, $\circ$, represents function composition. The *at* sign, @, is used to name a pattern matched value. The arrow, $\leftarrow$, is used to bind a variable on the left-hand side. The arrow can also be used to pattern match the value on the right-hand side (see line 9). We use symbols $\bigwedge$ and $\bigvee$ to represent creating, respectively, a conjunction or a disjunction node in a process tree.

Functions `drive_disj` and `drive_conj` describe how to process disjunctions and conjunctions respectively. 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. Thus driving is declared to be the function `drive_disj` (line 2).

A driving process (along with generalization and folding) 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 a 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. This means that only the goal, and not the substitution, is passed into the *whistle* to determine potential non-termination.

Residualization is done by the function `residualize`. Residualization traverses the process tree in preorder and generates the MINIKANREN goal as well as new relations whenever needed. A conjunction is created from a conjunction node, a disjunction — from a disjunction node, while substitutions are generated into a conjunction of unifications. Transient nodes are removed during residualization to improve the performance of programs. We also employ *redundant argument filtering* as described in [19].

Those disjuncts in which unifications fail are removed while driving. Each other disjunct takes the form of a possibly empty conjunction of relation calls accompanied with a substitution computed from unifications. Any MINIKANREN term can be trivially transformed into the described form. The function `normalize` in Fig. 2 is assumed to perform term normalization. The code is omitted for brevity but can be found in the implementation of the approach on github[3].

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 4.2 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 15–22). Joining instead of splitting is why we call our transformer *conservative* 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 okay 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 to not drive a conjunction which has already been 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 is then residualized into a call to the corresponding relation.

In this approach, we do not generalize in the same fashion as CPD or supercompilation. This decision was motivated by keeping the complexity of the approach to the minimum. Our conjunctions are always split into individual calls and are joined back together only if it is meaningful, for example, leads to contradictions. If the need for generalization arises, i.e. homeomorphic embedding of conjunctions [5] is detected, then we immediately stop driving this conjunction (line 12). When residualizing such a conjunction, we just generate a conjunction of calls to the input program before specialization.

4.1 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 the context of specialization of

[3]The project repository: `https://github.com/kajigor/uKanren_transformations/`. Access date: 28.02.2021

```
1 heuristically_select_a_call :: Conjunction → Maybe Call
2 heuristically_select_a_call C =
3   find isStatic C <|> find isDeterministic C <|> find isLessBranching C
```

Figure 3: Heuristic selection pseudocode

functional programming languages, as well as inlining in specialization of imperative languages, 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 [12].

Unfortunately, this is not the case for the specialization of a relational programming language. Unlike functional and imperative languages, in logic and relational programming languages unfolding may easily affect the target program's efficiency [18, 8]. Unfolding too much may create extra unifications, which is by itself a costly operation, or even introduce duplicated computations by propagating the results of unfolding 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.

4.2 Less Branching Heuristic

This heuristic is aimed at selecting a relation call within a conjunction which is both safe to unfold and may lead to discovering contradictions within the conjunction. An unsafe unfolding leads to an uncontrollable increase of 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.

The pseudocode describing our heuristic is shown in Fig. 3. 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.

5 Example

In this section we demonstrate by example how conservative partial deduction works. The example program is a relational interpreter of propositional formulas under given variable assignments. The complete code of the example program is provided in Listing 2.

```
let rec evalᵒ s fm res = conde [fresh (x y z v w) (
    ( fm ≡ conj x y  ∧  evalᵒ s x v  ∧  evalᵒ s y w  ∧  andᵒ v w res );
    ( fm ≡ disj x y  ∧  evalᵒ s x v  ∧  evalᵒ s y w  ∧  orᵒ  v w res );
    ( fm ≡ neg x     ∧  evalᵒ s x v  ∧  notᵒ v res );
    ( fm ≡ var v     ∧  elemᵒ s v res ))]

let notᵒ x y = nandᵒ x x y
let orᵒ  x y z = nandᵒ x x xx  ∧  nandᵒ y y yy  ∧ nandᵒ xx yy z
let andᵒ x y z = nandᵒ x y xy  ∧  nandᵒ xy xy z
let nandᵒ 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 )]

let elemᵒ n s v = conde [ fresh (h t m) (
  ( n ≡ zero  ∧  s ≡ h % t  ∧  h ≡ v );
  ( n ≡ succ m  ∧  s ≡ h % t  ∧  elemᵒ m t v ))]
```

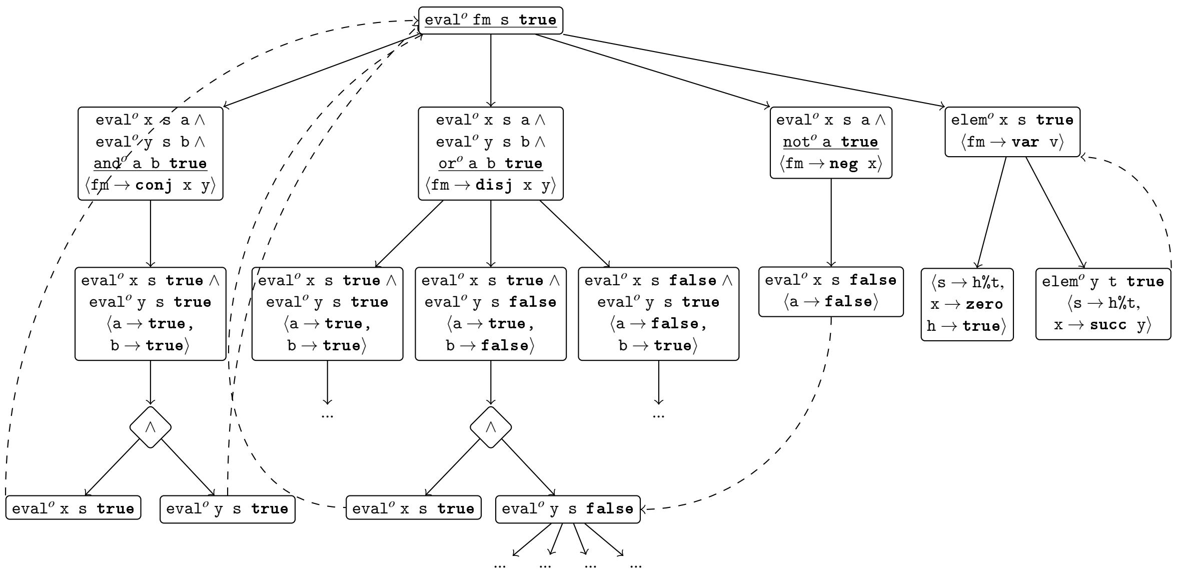
Listing 2: Evaluator of propositional formulas

The relation `eval`^{o} has three arguments. The first argument, `s`, is a list of boolean values which plays the role of variable assignments. The *i*-th value of the substitution is the value of the *i*-th variable. The second argument, `fm`, 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, `res`, is the value of the formula under the given assignment.

The relation `eval`^{o} is in canonical normal form: it is a single disjunction which consists of 4 conjunctions of unifications and relation calls, and all its fresh variables are introduced at the top level. The unification in each conjunction determines the shape of the input formula and binds variables to the corresponding subformulas. For each of the subformulas, the relation `eval`^{o} is called recursively. Then the results of the evaluation of the subformulas are combined by the corresponding boolean connective to get the result for the input formula. For example, when the formula is a conjunction of two subformulas `x` and `y`, the results of their execution `v` and `w` are combined by the call to the relation `and`^{o} to compute the result: `and`^{o} `v w res`. If the input formula is a variable, then its value is looked up in the substitution list by means of the relation `elem`^{o}.

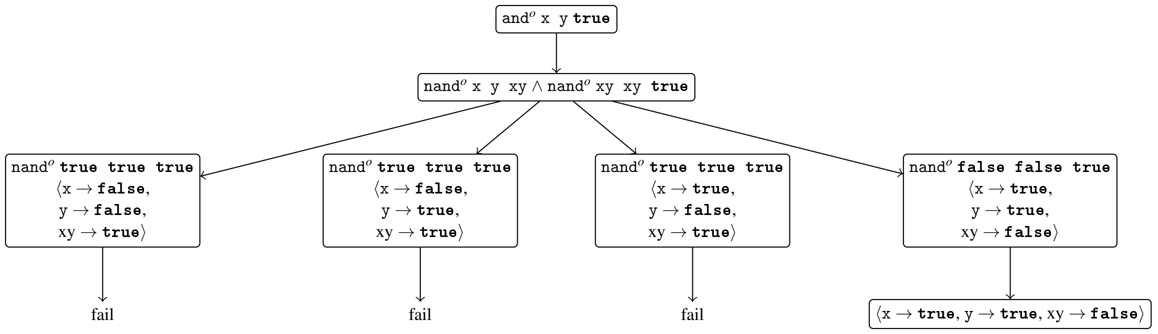
Consider the goal **fresh** `(s fm) (eval`^{o} `s fm` **true**`)`. The partially constructed process graph for this goal is presented in Fig. 4. The following notation is used. Rectangle nodes contain configurations, while diamond nodes correspond to splitting. Each configuration contains a goal along with a substitution in angle brackets. To visually differentiate constructors from variables within goals, we made them bold. For brevity, we only put a fragment of the substitution computed at each step in the corresponding node. The leaf node which contains only the substitution and no goal in its configuration is a success node. The call selected to be unfolded is underlined in a conjunction. Nodes corresponding to failures are not present within the process tree. Dashed arrows mark renamings.

Conservative partial deduction starts by unfolding the single relation call in this goal once. Besides introducing fresh variables into the context, unifications in each conjunct are computed into substitutions.



eval^o fm s true
eval^o x s a ∧ eval^o y s b ∧ and^o a b true ⟨fm → conj x y⟩
eval^o x s a ∧ eval^o y s b ∧ or^o a b true ⟨fm → disj x y⟩
eval^o x s a ∧ not^o a true ⟨fm → neg x⟩
elem^o x s true ⟨fm → var v⟩
eval^o x s true ∧ eval^o y s true ⟨a → true, b → true⟩
eval^o x s true ∧ eval^o y s true ⟨a → true, b → true⟩
eval^o x s true ∧ eval^o y s false ⟨a → true, b → false⟩
eval^o x s false ∧ eval^o y s true ⟨a → false, b → true⟩
eval^o x s false ⟨a → false⟩
⟨s → h%t, x → zero h → true⟩
elem^o y t true ⟨s → h%t, x → succ y⟩
∧
∧
eval^o x s true
eval^o y s true
eval^o x s true
eval^o y s false

Figure 4: Partially constructed process graph for the relation eval^o.



and^o x y true
nand^o x y xy ∧ nand^o xy xy true
nand^o true true true ⟨x → false, y → false, xy → true⟩
nand^o true true true ⟨x → false, y → true, xy → true⟩
nand^o true true true ⟨x → true, y → false, xy → true⟩
nand^o false false true ⟨x → true, y → true, xy → false⟩
fail
fail
fail
⟨x → true, y → true, xy → false⟩

Figure 5: Unfolding of and^o.

This produces 4 branches, each of which is processed further.

Consider the first branch, in which the input formula `fm` is a conjunction of subformulas `x` and `y`. First, each relation call within a conjunction is examined separately to select one of the calls to unfold. To do so, we unfold each of them in isolation and use the less branching heuristic. Since both recursive calls to `eval`^{o} are done with three distinct fresh variables, they are not selected according to the less branching heuristic. By unfolding the call to `and`^{o} several times, we determine it has a single non-failing branch (see Fig. 5), which is less than the same relation would have if called on all free and distinct variables, thus this call is selected to be unfolded. The result of unfolding the call is a single substitution which associates the variables `a` and `b` with **true**. By applying the computed substitution to the goal we get a conjunction of two calls to the `eval`^{o} relation with the last argument being **true**. This conjunction embeds the root goal, thus we split the conjunction and both calls become leaves which rename the root. This finishes the processing of the first branch.

Consider the second branch, in which the input formula `fm` is a disjunction of subformulas `x` and `y`.

```
let rec eval$^o_{true}$ s fm = conde [fresh (x y z v w) (
    ( fm ≡ conj x y ∧ eval$^o_{true}$ s x ∧eval$^o_{true}$ s y );
    ( fm ≡ disj x y ∧ (conde [
        ( eval$^o_{true}$ s x ∧ eval$^o_{true}$ s y );
        ( eval$^o_{true}$ s x ∧ eval$^o_{false}$ s y );
        ( eval$^o_{false}$ s x ∧ eval$^o_{true}$ s y );
    ]);
    ( fm ≡ neg x ∧ eval$^o_{false}$ s x v );
    ( fm ≡ var v ∧ elem$^o_{true}$ s v ))]

let rec eval$^o_{false}$ s fm = conde [fresh (x y z v w) (
    ( fm ≡ conj x y ∧ (conde [
        ( eval$^o_{false}$ s x ∧ eval$^o_{false}$ s y );
        ( eval$^o_{true}$ s x ∧ eval$^o_{false}$ s y );
        ( eval$^o_{false}$ s x ∧ eval$^o_{true}$ s y );
    ]);
    ( fm ≡ disj x y ∧ eval$^o_{true}$ s x ∧eval$^o_{true}$ s y );
    ( fm ≡ neg x ∧ eval$^o_{true}$ s x v );
    ( fm ≡ var v ∧ elem$^o_{false}$ s v ))]

let elem$^o_{true}$ n s = conde [ fresh (h t m) (
    ( n ≡ zero ∧ s ≡ true % t );
    ( n ≡ succ m ∧ s ≡ h % t ∧ elem$^o_{true}$ m t ))]

let elem$^o_{false}$ n s = conde [ fresh (h t m) (
    ( n ≡ zero ∧ s ≡ false % t );
    ( n ≡ succ m ∧ s ≡ h % t ∧ elem$^o_{false}$ m t ))]
```

Listing 3: Specialized evaluator of propositional formulas

Similarly to the first branch, the heuristic selects the call to the boolean relation to be unfolded which produces three possible substitutions for variables a and b. The substitutions are propagated into the goals and three branches, each of which embeds the root goal, are added into the process tree. Each goal is then split, and the calls with the last argument being **true** rename the root. Finally, the call eval^o y s **false** is processed, which creates 4 branches similar to the branches of the root goal.

The third branch is driven until a call with the last argument being **false** is encountered. Since it renames one of the nodes which is already present in the process tree, we stop exploring this branch and add the back edge.

The last branch, in which the input formula fm is a variable v, contains the single call to the relation elem^o. The unfolding of this call produces two leaves: a success node and a renaming of the parent node. This finishes the construction of the process tree.

The process tree is then residualized into a specialized version of the eval^o relation (see Listing 3). This program does not contain any calls to boolean connectives. Neither does the program contain the original, not specialized, relation eval^o.

It is worth noting that the result produced by the Conservative Partial Deduction is not ideal. For example, in the definition of the $\texttt{eval}^o_{true}$, when the input formula `fm` is a disjunction of subformulas `x` and `y`, the recursive call $\texttt{eval}^o_{true}$ `s x` is done twice in two disjuncts. The ideal version of the relation $\texttt{eval}^o_{true}$ should contain this recursive call only once. This can be done, for example, by common subexpression elimination [25]. However, ideally, `y` should not be evaluted at all, since the value of the formula `fm` does not depend on it. It is unclear if and how this kind of transformation can be done automatically. Such transformation would require, first, realising that a disjunction of two relation calls $\texttt{eval}^o_{true}$ `s y` and $\texttt{eval}^o_{false}$ `s y` exhaust all possible values for `y`. Secondly, the transformation would have to examine if a relation restricts values of a given argument regardless of the other arguments' values.

6 Evaluation

We implemented[4] the conservative partial deduction for miniKanren and compared it with the ECCE partial deduction system. ECCE is designed for the Prolog programming language and cannot be directly applied for programs written in miniKanren. Nevertheless, the languages show resemblance, and it is valuable to check if the existing methods for Prolog can be used directly in the context of relational programming. To be able to compare our approach with ECCE, we converted each input program first to the pure subset of Prolog, then specialized it with ECCE, and then we converted the result back to miniKanren. The conversion to Prolog is a simple syntactic conversion. In the conversion from Prolog to miniKanren, for each Horn clause a conjunction is generated in which unifications are placed before any relation call. All programs are run as miniKanren programs in our experiments. In the final subsection we discuss some limitations of our approach and ECCE.

We chose two problems for our study: evaluation of a subset of propositional formulas and typechecking for a simple language. The problems illustrate the approach of using relational interpreters to solve search problems [22]. For both these problems we considered several possible implementations in miniKanren which highlight different aspects relevant in specialization.

The $\texttt{eval}^o$ relation implements an evaluator of a subset of propositional formulas described in Section 5. We consider four different implementations of this relation to explore how the way the program is implemented can affect the quality of specialization. Depending on the implementation, ECCE generates programs of varying performance, while the execution times of the programs generated by our approach are similar.

The $\texttt{typecheck}^o$ relation implements a typechecker for a tiny expression language. We consider two different implementations of this relation: one written by hand and the other generated from a functional program, which implements the typechecker, as described in [22]. We demonstrate how much these implementations differ in terms of performance before and after specialization.

In this study we measured the execution time for the sample queries, averaging them over multiple runs. We also measured the number of unifications done in search of each individual answer. All examples of miniKanren relations in this paper are written in OCanren. 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.

The tables and graphs use the following denotations. *Original* represents the execution time of a program before any transformations were applied; *ECCE* — of the program specialized by ECCE with the default conjunctive control setting; *ConsPD* — of the program specialized by our approach.

[4]The project repository: `https://github.com/kajigor/uKanren_transformations/`. Access date: 28.02.2021

```
let rec evalᵒ s fm res = conde [fresh (x y z v w) (
    (fm ≡ conj x y  ∧  evalᵒ s x v  ∧  evalᵒ s y w  ∧  andᵒ v w res);
    (fm ≡ disj x y  ∧  evalᵒ s x v  ∧  evalᵒ s y w  ∧  orᵒ  v w res);
    (fm ≡ neg x     ∧  evalᵒ s x v  ∧  notᵒ v res));
    (fm ≡ var v     ∧  elemᵒ s v res)]
```

Listing 4: Evaluator of formulas with boolean operation last

```
let rec evalᵒ s fm res = conde [fresh (x y z v w) (
    (fm ≡ conj x y  ∧  andᵒ v w res  ∧  evalᵒ s x v  ∧  evalᵒ s y w);
    (fm ≡ disj x y  ∧  orᵒ  v w res  ∧  evalᵒ s x v  ∧  evalᵒ s y w);
    (fm ≡ neg x     ∧  notᵒ v res   ∧  evalᵒ s x v);
    (fm ≡ var v     ∧  elemᵒ s v res))]
```

Listing 5: Evaluator of formulas with boolean operation second

6.1 Evaluator of Logic Formulas

The relation eval^o describes an evaluation of a propositional formula under given variable assignments presented in section 5. We specialize the eval^o relation to synthesize formulas which evaluate to **true**. To do so, we run the specializer for the goal with the last argument fixed to **true**, while the first two arguments remain free variables. Depending on the way eval^o is implemented, different specializers generate significantly different residual programs.

6.1.1 The Order of Relation Calls

One possible implementation of the eval^o relation is presented in Listing 4. Here the relation elem^o s v res unifies res with the value of the variable v in the list s. The relations and^o, or^o, and not^o encode corresponding boolean connectives.

Note, the calls to boolean relations and^o, or^o, and not^o are placed last within each conjunction. This poses a challenge for the CPD-based specializers such as ECCE. 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 s fm **true**), it fails to propagate the direction data onto recursive calls of eval^o. Knowing that res is **true**, we can conclude that the variables v and w have to be **true** as well in the call and^o v w res. There are three possible options for these variables in the call or^o v w res and one for the call not^o v res. These variables are used in recursive calls of eval^o and thus restrict the result of its execution. CPD fails to recognize this, and thus unfolds recursive calls of eval^o applied to fresh variables. It leads to over-unfolding, large residual programs and poor performance.

The conservative partial deduction first unfolds those calls which are selected according to the heuristic. Since exploring the implementations of boolean connectives makes more sense, they are unfolded before the recursive calls of eval^o. The way conservative partial deduction treats this program is the same as it treats the other implementation in which boolean connectives are moved to the left, as shown in Listing 5. This program is easier for ECCE to specialize which demonstrates how unequal the behaviour of CPD for similar programs is.

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

Listing 6: Implementation of boolean not^o as a table

```
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 7: Implementation of boolean operations via nand^o

6.1.2 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 $\texttt{eval}^o$ boolean connectives 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 by mimicking a truth table as demonstrated by the implementation of $\texttt{not}^o$ in Listing 6. It is enough to unfold such relation calls once to derive useful information about variables.

The other way to implement boolean connectives is to express them using a single basic boolean relation such as $\texttt{nand}^o$ which, in turn, has a table-based implementation (see Listing 7). It takes several sequential unfoldings to derive that the variables `v` and `w` should be **true** when considering a call $\texttt{and}^o$ `v w` **true** implemented via a basic relation. Conservative 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.

6.1.3 Evaluation Results

In our study we considered 4 implementations of $\texttt{eval}^o$ summarised in the Table 1. They differ in the way the boolean connectives are implemented (see column *Implementation*) and whether they are placed before or after the recursive calls to $\texttt{eval}^o$ (see column *Placement*). These four implementations are very different from the standpoint of ECCE. We measured the time necessary to generate 1000 formulas over two variables which evaluate to **true** (averaged over 10 runs). The results are presented in Fig. 6.

Conservative partial deduction generates programs with comparable performance for all four implementations, while the quality of ECCE specialization differs significantly. ECCE worsens performance for every implementation as compared to the original program. ConsPD does not worsen performance for any implementation. Its effect is most significant for the implementations in which the boolean connectives are placed first within conjunctions.

	Implementation	Placement
FirstPlain	table-based	before
LastPlain	table-based	after
FirstNando	via nand^o	before
LastNando	via nand^o	after

Table 1: Different implementations of eval^o

	Original	ECCE	ConsPD
First-Plain	1.59s	1.61s	0.92s
First-Nando	1.43s	2.24s	0.96s
Last-Plain	0.98s	1.43s	0.97s
Last-Nando	1.09s	1.54s	0.91s

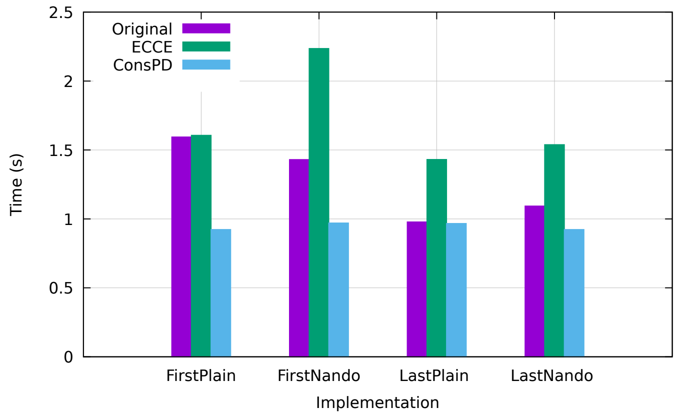


Figure 6: Execution time of eval^o

6.1.4 The Order of Answers

It is important to note that different implementations of the same miniKanren relation produce answers in different orders. Nevertheless, since miniKanren search is complete, all answers will be found eventually. Unfortunately, it is not guaranteed that the first 1000 formulas generated with different implementations of `eval`^{o} will be the same. For example, 983 formulas are the same among the first 1000 formulas generated by the Original *FirstPlain* relation and the same relation after the ConsPD transformation. At the same time, only 405 formulas are the same between the Original and ECCE *LastNando* relations.

The reason why implementations differ so much in the order of the answers stems from the canonical search strategy employed in miniKanren. Most miniKanren implementations employ *interleaving* search [14] which is left-biased. It means that the leftmost disjunct in a relation is being executed longer than the disjunct on the right. This property is not local which makes it very hard to estimate the performance of a given relation.

In practice it means that if a specializer reorders disjuncts, then the performance of relations after specialization may be unpredictable. For example, by putting the disjuncts of the `eval`^{o} relation in the opposite order, one produces a relation which runs much faster than the original, but it generates completely different formulas at the same time. Most of the first 1000 formulas in this case are multiple negations of a variable, while the original relation produces more diverse set of answers. Computing a negation of a formula only takes one recursive `eval`^{o} call thus finding such answers is faster than conjunctions and disjunctions. Meanwhile, the formulas generated by the reordered relation are less diverse and may be of less interest.

Although neither ECCE nor ConsPD reorder disjuncts, they remove disjuncts which cannot succeed.

Thus they influence the order of answers and performance of relations. Both methods reduce the number of unifications needed to compute each individual answer thus performing specialization. In general, it is not possible to guarantee the same order of answers after specialization. Exploring how different specializations influence the execution order is a fascinating direction for future research.

6.2 Typechecker-Term Generator

This relation implements a typechecker for a tiny expression 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.

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

The typing rules are straightforward and are presented in Fig. 7. Boolean and integer constants have the corresponding types regardless of the environment. Only terms of type integer can be added, multiplied or compared by the less-than operator. Any terms of the same type can be checked for equality. Addition and multiplication of two terms of suitable types have integer type, while comparisons have boolean type. The 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 an i-th element of the list.

$$\frac{}{\Gamma \vdash IConst\ i : Int} \qquad \frac{}{\Gamma \vdash BConst\ b : Bool} \qquad \frac{}{\Gamma \vdash Var\ v : \tau}\ \Gamma[v] \equiv \tau$$

$$\frac{\Gamma \vdash t : Int, \Gamma \vdash s : Int}{\Gamma \vdash t + s : Int} \qquad \frac{\Gamma \vdash t : \tau, \Gamma \vdash s : \tau}{\Gamma \vdash t = s : Bool} \qquad \frac{\Gamma \vdash v : \tau_v,\ (\tau_v :: \Gamma) \vdash b : \tau}{\Gamma \vdash \underline{let}\ v\ b : \tau}$$

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

Figure 7: Typing rules implemented in typecheck^o relation

We compared two implementations of these typing rules. The first one is obtained by unnesting of a functional program, which implements the typechecker, as described in [22] (*Generated*). It is worth noting that the unnesting introduces a lot of redundancy in the form of extra unifications and thus creates programs which are very inefficient. Thus we contrast this implementation with the program hand-written in OCanren (*Hand-written*). Each implementation has been specialized with ConsPD and ECCE. We measured the time needed to generate 1000 closed terms of type integer (see Fig. 8).

As expected, the generated program is far slower than the hand-written one. The principal difference between these two implementations is that the generated program contains a certain redundancy introduced by unnesting. For example, typechecking of the sum of two terms in the hand-written im-

	Original	ECCE	ConsPD
Hand-written	0.92s	0.22s	0.34s
Generated	11.46s	0.38s	0.29s

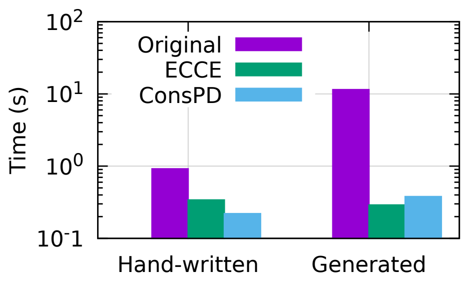


Figure 8: Execution time of generating 1000 closed terms of type integer

```
let rec typecheckº gamma term res = conde [
  ...
  fresh (x y) ((term ≡ x + y ∧
     typecheckº gamma x (some integer) ∧
     typecheckº gamma y (some integer) ∧
     res ≡ (some integer)));
  ...]
```

Listing 8: A fragment of the hand-written typechecker

```
let rec typecheckº gamma term res = conde [
  ...
  fresh (x y t1 t2) ((term ≡ x + y ∧
    conde [
      typecheckº gamma x none       ∧ res ≡ none;
      typecheckº gamma x (some t1) ∧
        typecheckº gamma y none     ∧ res ≡ none;
      typecheckº gamma x (some t1) ∧ typecheckº gamma y (some t2) ∧
        typeEqº t1 integer true     ∧ typeEqº t2 integer true ∧
        res ≡ (some integer);
    ])
  ...]
```

Listing 9: A fragment of the generated typechecker

plementation consists of a single conjunction (see Listing 8) while the generated program is far more complicated and also uses a special relation `typeEq`^{o} to compare types (see Listing 9).

Most redundancy of the generated program is removed by specialization with respect to the known type. This is why both implementations have comparable speed after specialization. ECCE shows bigger speedup for the hand-written program than ConsPD and vice versa for the generated implementation. We believe that this difference can be explained by too much unfolding. ECCE performs a lot of excessive unfolding for the generated program and only barely changes the hand-written program. At the same time ConsPD specializes both implementations to comparable programs performing an average amount of unfolding. This shows that the heuristic we presented gives more stable, although not the best, results.

```
let doubleAppendᵒ x y z res = fresh (t) (
    appendᵒ x y t ∧ appendᵒ t z res )

let appendᵒ x y res = conde [ fresh (h t r) (
    ( x ≡ [] ∧ res ≡ y );
    ( x ≡ h % t ∧ appendᵒ t y r ∧ res ≡ h % r ))]
```

Listing 10: Inefficient implementation of concatenation of three lists

```
let maxLengthᵒ x m l = fresh (t) (
    maxᵒ x m ∧ lengthᵒ x l )

let lengthᵒ x l = conde [ fresh (h t r) (
    ( x ≡ [] ∧ l ≡ zero );
    ( x ≡ h % t ∧ lengthᵒ t r ∧ l ≡ succ r ))]

let maxᵒ x m = max₁ᵒ x zero m
let rec max₁ᵒ x n m = fresh (h t) ( conde [
    (x ≡ [] ∧ m ≡ n);
    (x ≡ h % t) ∧ (conde [
      (leᵒ h n true ∧ max₁ᵒ t n m);
      (gtᵒ h n true ∧ max₁ᵒ t h m)])])
```

Listing 11: Inefficient implementation of maxLength^o

6.3 Discussion: Tupling and Deforestation

Tupling and deforestation are among the important transformations conjunctive partial deduction is capable of. Deforestation is often demonstrated by the doubleAppend^o program which concatenates three lists by calling the concatenation relation append^o twice in a conjunction (see Listing 10). The two calls to append lead to double traversal of the first list, which is inefficient. The program may be transformed in such a way so as to only traverse the first list once (see [5] for details), which conjunctive partial deduction does.

Conjunctive partial deduction achieves this effect by considering the conjunction of two append^o calls as a whole. At the local control level, it first unfolds the leftmost call, propagates the computed substitutions onto the rightmost call, and then unfolds the rightmost call in the context of the substitutions. When the first list is not empty, this leads to discovering a renaming of a conjunction of two append^o calls. By renaming this conjunction into a new predicate, deforestation is achieved in this example.

Unfortunately, conservative partial deduction does not succeed at this transformation on this example. This happens because ConsPD splits the conjunction of two calls to append^o, since none of them is selected by the less branching heuristic. Splitting the conjunction leads to information loss and makes it so there is no renaming of the whole conjunction in the process tree.

A similar thing happens when considering the common example on which tupling is demonstrated in literature on CPD: the maxLength^o program. The original implementation of this program computes the maximum element of the list along with the length of the list by conjunction of two calls to the corresponding relations: `max`^{o} and `length`^{o} (see Listing. 11). This implementation also traverses the input list twice when run in the forward direction. By tupling, this program may be transformed so that the list is traversed once while both the maximum value and the length of the list are computed simultaneously, and CPD is capable to achieve this transformation with the default settings. Conservative partial deduction also splits too much too early and thus fails to yield any useful transformation for this program.

It is worth noting that determinate unfolding performed by CPD plays a huge role in these examples. The default unfolding strategy implemented in ECCE allows for only a single non-determinate unfolding per a local control tree. When considering conjunctions `append`^{o} `x y t` $\wedge$ `append`^{o} `t z res` and `max`^{o} `x m` $\wedge$ `length`^{o} `x l`, it unfolds the leftmost call which produces several branches in the tree. The rightmost call is only considered, if unfolding of its conjunction with the result of unfolding of the leftmost call produces only one result. This indeed happens in these two examples. If it was not to happen, then the conjunction would have been split at the global level and no deforestation or tupling would have been achieved. It is not that hard to modify the examples in such a way so that CPD fails to transform them in a meaningful way. For example, one extra disjunct can be added into the `append`^{o} relation, or the calls to `max`^{o} and `length`^{o} may be reordered. This is evidence of how non-trivial and fragile these transformers are. More research should be done to make sure useful transformations are possible for many input programs.

7 Conclusion

In this paper we discussed some issues which arise in the area of partial deduction techniques for the relational programming language miniKanren. We presented a novel approach to partial deduction — conservative partial deduction — which uses a heuristic to select a suitable relation call to unfold at each step of driving. We compared this approach with the most sophisticated implementation of conjunctive partial deduction — ECCE partial deduction system — on 6 relations which solve 2 different problems.

Our specializer improved the execution time of all queries. ECCE worsened the performance of all 4 implementations of the propositional evaluator relation, while improving the other queries. Conservative partial deduction is more stable with regards to the order of relation calls than ECCE which is demonstrated by the similar performance of all 4 implementations of the evaluator of logic formulas.

Some queries to the same relation were improved more by ConsPD, while others — by ECCE. We conclude that there is still not one good technique which definitively speeds up every relational program. More research is needed to develop models capable of predicting the performance of a relation which can be used in specialization of miniKanren. There are some papers which estimate the efficiency of partial evaluation in the context of logic and functional logic programming languages [30, 31], and may facilitate achieving this goal. Employing a combination of offline and online transformations as done in [32] may also be the step towards more effective and predictable partial evaluation. Other directions for future research include exploring how specialization influences the execution order of a miniKanren program, improving ConsPD so that it succeeds at deforestation and tupling more often, and coming up with a larger, more impressive, set of benchmarks.

Acknowledgements

We gratefully acknowledge the anonymous referees and the participants of VPT-2021 workshop for fruitful discussions and many useful suggestions.

References

[1] Elvira Albert, Germán Puebla & John P. Gallagher (2006): *Non-leftmost Unfolding in Partial Evaluation of Logic Programs with Impure Predicates*. In Patricia M. Hill, editor: *Logic Based Program Synthesis and Transformation*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 115–132, doi:10.1007/11680093_8.

[2] Maximilian C. Bolingbroke & Simon L. Peyton Jones (2010): *Supercompilation by evaluation*. In Jeremy Gibbons, editor: *Proceedings of the 3rd ACM SIGPLAN Symposium on Haskell, Haskell 2010, Baltimore, MD, USA, 30 September 2010*, ACM, pp. 135–146, doi:10.1145/1863523.1863540.

[3] Mikhail A. Bulyonkov (1984): *Polyvariant Mixed Computation for Analyzer Programs*. *Acta Inf.* 21, pp. 473–484, doi:10.1007/BF00271642.

[4] William E Byrd, Eric Holk & Daniel P Friedman (2012): *miniKanren, live and untagged: Quine generation via relational interpreters (programming pearl)*. In: *Proceedings of the 2012 Annual Workshop on Scheme and Functional Programming*, pp. 8–29, doi:10.1145/2661103.2661105.

[5] Danny De Schreye, Robert Glück, Jesper Jørgensen, Michael Leuschel, Bern Martens & Morten Heine Sørensen (1999): *Conjunctive partial deduction: Foundations, control, algorithms, and experiments*. *The Journal of Logic Programming* 41(2-3), pp. 231–277, doi:10.1016/S0743-1066(99)00030-8.

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

[7] Jason Hemann Daniel P Friedman: *μKanren: A Minimal Functional Core for Relational Programming*.

[8] John P Gallagher (1993): *Tutorial on specialisation of logic programs*. In: *Proceedings of the 1993 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pp. 88–98, doi:10.1145/154630.154640.

[9] Robert Glück & Morten Heine Sørensen (1994): *Partial deduction and driving are equivalent*. In: *International Symposium on Programming Language Implementation and Logic Programming*, Springer, pp. 165–181, doi:10.1007/3-540-58402-1_13.

[10] Geoff W Hamilton (2007): *Distillation: extracting the essence of programs*. In: *Proceedings of the 2007 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pp. 61–70, doi:10.1145/1244381.1244391.

[11] G. Higman (1952): *Ordering by divisibility in abstract algebras*. In: *Proceedings of the London Mathematical Society*, 2, pp. 326–336, doi:10.1112/plms/s3-2.1.326.

[12] Neil D. Jones, Carsten K. Gomard & Peter Sestoft (1993): *Partial evaluation and automatic program generation*. Prentice Hall international series in computer science, Prentice Hall.

[13] Peter A. Jonsson & Johan Nordlander (2011): *Taming code explosion in supercompilation*. In Siau-Cheng Khoo & Jeremy G. Siek, editors: *Proceedings of the 2011 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2011, Austin, TX, USA, January 24-25, 2011*, ACM, pp. 33–42, doi:10.1145/1929501.1929507.

[14] Oleg Kiselyov, Chung-chieh Shan, Daniel P. Friedman & Amr Sabry (2005): *Backtracking, Interleaving, and Terminating Monad Transformers: (Functional Pearl)*. *SIGPLAN Not.* 40(9), p. 192203, doi:10.1145/1090189.1086390.

[15] J. B. Kruskal (1960): *Well-quasi ordering, the tree theorem, and Vazsonyi's conjecture*. 95, pp. 210–225, doi:10.2307/1993287.

[16] Michael Leuschel (1997): *Advanced techniques for logic program specialisation*.

[17] Michael Leuschel (1997): *The ecce partial deduction system*. In: *Proceedings of the ILPS*, 97, Citeseer.

[18] Michael Leuschel & Maurice Bruynooghe (2002): *Logic program specialisation through partial deduction: Control issues*. *Theory and Practice of Logic Programming* 2(4-5), pp. 461–515, doi:10.1017/S147106840200145X.

[19] Michael Leuschel & Morten Heine Sørensen (1996): *Redundant argument filtering of logic programs*. In: *International Workshop on Logic Programming Synthesis and Transformation*, Springer, pp. 83–103, doi:10.1007/3-540-62718-9_6.

[20] Michael Leuschel & Germán Vidal (2014): *Fast offline partial evaluation of logic programs*. *Information and Computation* 235, pp. 70–97, doi:10.1016/j.ic.2014.01.005.

[21] John W. Lloyd & John C Shepherdson (1991): *Partial evaluation in logic programming*. *The Journal of Logic Programming* 11(3-4), pp. 217–242, doi:10.1016/0743-1066(91)90027-M.

[22] Petr Lozov, Ekaterina Verbitskaia & Dmitry Boulytchev (2019): *Relational Interpreters for Search Problems*. In: *miniKanren and Relational Programming Workshop*, p. 43.

[23] Petr Lozov, Andrei Vyatkin & Dmitry Boulytchev (2017): *Typed relational conversion*. In: *International Symposium on Trends in Functional Programming*, Springer, pp. 39–58, doi:10.1007/978-3-319-89719-6_3.

[24] Neil Mitchell & Colin Runciman (2007): *A Supercompiler for Core Haskell*. In Olaf Chitil, Zoltán Horváth & Viktória Zsók, editors: *Implementation and Application of Functional Languages, 19th International Workshop, IFL 2007, Freiburg, Germany, September 27-29, 2007. Revised Selected Papers*, *Lecture Notes in Computer Science* 5083, Springer, pp. 147–164, doi:10.1007/978-3-540-85373-2_9.

[25] Steven Muchnick et al. (1997): *Advanced compiler design implementation*. Morgan kaufmann.

[26] Dmitry Rozplokhas, Andrey Vyatkin & Dmitry Boulytchev (2020): *Certified Semantics for Relational Programming*. In: *Asian Symposium on Programming Languages and Systems*, Springer, pp. 167–185, doi:10.1007/978-3-030-64437-6_9.

[27] Morten Heine Soerensen, Robert Glück & Neil D. Jones (1996): *A positive supercompiler*. *Journal of functional programming* 6(6), pp. 811–838, doi:10.1017/S0956796800002008.

[28] Morten Heine B Sørensen (1998): *Convergence of program transformers in the metric space of trees*. In: *International Conference on Mathematics of Program Construction*, Springer, pp. 315–337, doi:10.1007/BFb0054297.

[29] Raf Venken & Bart Demoen (1988): *A partial evaluation system for prolog: some practical considerations*. *New Generation Computing* 6(2-3), pp. 279–290, doi:10.1007/BF03037142.

[30] Germán Vidal (2004): *Cost-augmented partial evaluation of functional logic programs*. *Higher-Order and Symbolic Computation* 17(1), pp. 7–46, doi:10.1023/B:LISP.0000029447.02190.42.

[31] Germán Vidal (2008): *Trace Analysis for Predicting the Effectiveness of Partial Evaluation*. In: *International Conference on Logic Programming*, Springer, pp. 790–794, doi:10.1007/978-3-540-89982-2_78.

[32] Germán Vidal (2011): *A Hybrid Approach to Conjunctive Partial Evaluation of Logic Programs*. In María Alpuente, editor: *Logic-Based Program Synthesis and Transformation*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 200–214, doi:10.1007/978-3-642-20551-4_13.

[33] Philip Wadler (1990): *Deforestation: Transforming Programs to Eliminate Trees*. *Theor. Comput. Sci.* 73(2), pp. 231–248, doi:10.1016/0304-3975(90)90147-A.

Appendix

We thank the reviewers for thourough reading of our paper and providing such thoughtful feedback. We addressed most of the comments in the revised version of the paper. Nevertheless, in the following subsections we provide our answers to the questions asked (our answers are written in cursive).

Review 1

The paper considers the optimisation of miniKanren relational programs using some form of (conjunctive) partial evaluation. In particular, the authors propose a novel approach, called conservative partial deduction, and reports on an experimental evaluation that compares the results obtained with ECCE (an online partial evaluator for Prolog based on conjunctive partial deduction) and the authors' tool. The results are very promising and show the advantages of the new approach.

All in all, the paper contains interesting ideas and I recommend it to be accepted for VPT 2021.

Detailed comments for authors:

- In order for the paper to be as self-complete as possible, I'd suggest to add a brief introduction to miniKanren (syntax, informal semantics, a couple of examples) at the beginning of the paper.

  Added. See subsection 2.1.

- There are already some works that considered non-leftmost unfolding during partial evaluation, e.g.,

  - E. Albert, G. Puebla, J.P. Gallagher: Non-leftmost Unfolding in Partial Evaluation of Logic Programs with Impure Predicates. LOPSTR 2005: 115-132
  - M. Leuschel, G. Vidal: Fast offline partial evaluation of logic programs. Inf. Comput. 235: 70-97 (2014)

  Thank you for bringing these papers to our attention. We cited them in the paper.

- As for the effectiveness of partial evaluation, you can check this one

  - G. Vidal: Cost-Augmented Partial Evaluation of Functional Logic Programs. High. Order Symb. Comput. 17(1-2): 7-46 (2004)

  where the effectiveness of partial evaluation is estimated, and this one

  - G. Vidal: Trace Analysis for Predicting the Effectiveness of Partial Evaluation. ICLP 2008: 790-794

  which aims at predicting the effectiveness achieved by partial evaluation (though it's just a preliminary approach).

  Thank you! These papers do provide a nice way to estimate how efficient a transformation is and we will likely employ these ideas in future versions of our specializer, although this would likely require some tweaking for the interleaving semantics of miniKanren.

- The formalisation of conservative partial deduction using pseudocode is nice, but a more formal approach (as in [3]) might be useful to prove a number of properties. You could consider that as future work.

  We agree that our approach will benefit from a more formal description and will consider it as future work.

- In Sect. 3.1, you mention that unfolding too much might introduce extra unifications or duplicate computations. I think that this is not possible as long as partial evaluation considers a fixed left-to-right unfolding strategy at PE time. This is actually a problem when considering non-leftmost unfolding strategies at PE time.

  Section 5.1.1 of the paper [5] states that non-determinate unfolding may add unifications regardless of wether the unfolding strategy is left-to-right or not. Of course, in pure PROLOG *the effect of using unfolding strategies which are not left-to-right is more pronounced. Nevertheless, interleaving semantics of* MINIKANREN *complicates everything even further.*

- Another unfolding strategy that might be related to your heuristics is presented in

  - G. Vidal: A Hybrid Approach to Conjunctive Partial Evaluation of Logic Programs. LOPSTR 2010: 200-214

  where the notion of "strongly regular logic programs" is introduced in order to characterise the predicates whose unfolding cannot produce infinitely growing conjunctions at PE time.

  This notion seems to be related to our heuristic. Also, combining online and offline methods seems like a good idea. We will definitely consider this direction in future research.

- The experimental evaluation is certainly promising. Nevertheless, it would be great if you could put the implemented tool publicly available, including the source code of the considered benchmarks so that the readers can replicate the experiments.

  The tool is available on github: `https://github.com/kajigor/uKanren_transformations/`*, the experiments, currently not very well structured, are also on github in a different repository:* `https://github.com/kajigor/mk-transformers-bench`

Review 2

This paper looks at partial deduction for the relational programming language MINIKANREN. The execution of programs in MINIKANREN differs from that in Prolog in that the subgoals of a conjunction or disjunction can be reordered. This creates many more degrees of freedom in the way a program can be executed, and also in the optimisations that can be performed.

The paper looks at issues faced by conjunctive partial deduction of MINIKANREN and describes a new approach to partial deduction for relational languages called conservative partial deduction. The main issue with conjunctive partial deduction of MINIKANREN is the unfolding strategy. For Prolog, this is usally done by deterministic unfolding where all atoms except one are only unfolded if it matches with at most one clause head. One atom, usually the leftmost one, can be unfolded non-deterministically. However, this can add many new relation calls to a conjunction. Although this works well for Prolog, it does not work so well for MINIKANREN as it does not match its order of evaluation and can result in large less efficient programs.

The solution proposed to this problem in this paper is called conservative partial deduction, which is inspired by both partial deduction and supercompilation. One aim is to create a speialization algorithm that is simpler than conjunctive partial deduction, but I am not sure this has been achieved as it still seems quite complex.

We agree that our approach is not the simplest, but we still strive towards this goal.

The conservative aspect of the algorithm is in the unfolding of relation calls. This involves deciding 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. It is unclear how exactly this is preferrable to unfolding any other relation call.

The core idea behind this approach is in removing (at specialization-time) those computations which will definitely fail at run-time. To do so we examine which calls within a conjunction interact in such a way as to lead to failures (one may say, the calls contradict each other), and if there are such calls, we process as a whole conjunction rather than splitting and driving them separately, loosing the information about contradictions by doing so.

It might actually add many new relation calls to a conjunction and result in large less efficient programs. The result of the unfolding is joined back into the conjunction rather than being split as is done in conjunctive partial deduction. It also just stops on encountering an embedding rather than trying to generalise and further transform.

This decision was made in pursuit of reducing the algorithm's complexity. There is no reason not to do generalization of goals which do not require splitting.

The algorithm does not take advantage of the fact that subgoals in the language can be reordered, so it appears to miss out on many further opportunities for optimisation.

On the contrary, whenever the algorithm decides that some call within a conjunction may lead to a contradiction, it unfolds the call and replaces it with the result of unfolding. This, at specialization-time, changes the order in which the calls are executed (see how boolean connectives are processed in different implementations of the $\texttt{eval}^o$ *relation). The residualized program will have the calls (or rather the result of unfolding of the calls) in roughly the same order, which was our intention from the very beginning.*

The evaluation of the proposed technique is rather poor, with only variations of two test programs being evaluated. For the evaluator of logic formulas, the variations are on the way the boolean connectives are implemented, and whether these connectives are placed before or after the recursive call. For the type checker, the variations are on whether it was implemented by hand or automatically generated from a corresponding functional program. This is not enough to conclude whether the transformation works well in practice.

We agree this evaluation is not enough, and we are working on the better set of benchmarks. These programs were chosen to demonstrate typical issues which occur in developing efficient relational interpreters. Our goal in selecting them was to provide non-trivial examples which are illustrative and can be easily comprehended by the reader.

The transformed programs do all perform better than the original program, but ECCE does perform better than the described transformation for one program variation. The order of the answers produced by transformed programs can also be different to that of the original program, which I see as very problematic as the transformation is changing the behaviour of the program.

The search in MINIKANREN *is complete, thus all possible answers will be found, given enough time. This is why the* MINIKANREN *community does not focus on the order in which answers are found. The denotational semantics of a* MINIKANREN *program is a set of relations on the free variables of a goal rather that the list of answers in any particular order. See [26] for the proof of the equivalence of the denotational and operational semantics of* MINIKANREN.

Although the paper is well written, the contributions it makes to the area are minimal. It does not seem appropriate to apply conjunctive partial deduction to transform MINIKANREN programs, as conjunctive partial deduction follows the evaluation strategy of logic languages such as Prolog rather than that of relational languages such as MINIKANREN. It would seem to be more appropriate to devise a new transformation that follows the search strategy used by the MINIKANREN implementation.

We believe that using a well-established framework for specialization of a language related to MINIKANREN *is the most reasonable first step. If it had worked right away, we would have just ap-*

plied CPD and moved on to the next challenge. Alas, it did not work as well as we hopped, thus we proceeded to developing a different transformation.

For the correctness of the transformation, it should be proved that the transformation does not change the order of answers produced for the program.

According to the denotational semantics, the order of answers does not matter. Once we establish the best specialization method for MINIKANREN*, we will prove its correctness with respect to the denotational semantics of* MINIKANREN.

A more thorough evaluation of the described technique is also required. I am therefore on the fence as to whether to accept this paper or not.

We agree with the need for better evaluation.

Detailed Comments =================

Section 1, second last para: This opens a yet another possibility → This opens yet another possibility

Section 2, para 2: is so-called process tree → is a so-called process tree

Section 2, para 2: Of course, process tree → Of course, the process tree

Section 2, para 2: signalls → signals

Section 2, para 3: the efficiency of residual program → the efficiency of a residual program

Section 2, para 3: yielding standalone compilation more powerful → making standalone compilation more powerful

Section 2, final para: empitical → empirical

Section 3, para 5: use a heuristics which decides → use a heuristic which decides

Section 3, para 5: if the heuristics fails to select → if the heuristic fails to select

Section 3, final para: only if it is meaningful - what is meant by "meaningful"?

Here it was used as a more formal synonym for "if it makes sense".

Section 3.1, para 2: Unlike functional and imperative languages, in logic and relational programming languages unfolding may easily affect the target programs efficiency - actually unfolding in functional and imperative languages can also affect the program's efficiency if paramters are non-linear.

We agree that unfolding may decrease the performance of programs with non-linear arguments for functional and imperative languages. In logic and relational programming languages the effect seems to be more drastic because of how costly unification is.

Section 3.2, Figure 2: this figure is not really necessary and does not add very much

Section 3.2, para 3: the less-branching heuristics → the less-branching heuristic

Section 4, para 4: generated from the functional program - you have not really said what this functional program is

We meant the straightforward implementation of the interpreter in a functional language which was then translated into MINIKANREN *by means of relational conversion, according to the approach described in [22]. We clarified this in the paper.*

Section 4.1.1, para 2: direction data - it is not clear what is meant by this

We specialize a MINIKANREN *goal in which some of the arguments are known statically. Which arguments are known and which are free variables determines the "direction" in which relation call is executed. See the second and the third paragraphs of the seciton 1.*

Section 4.1.2, para 1: mimicking a truth tables → mimicking a truth table

Section 4.2, para 3: in form of extra unifications → in the form of extra unifications

Section 5, para 1: which uses a heuristics → which uses a heuristic

Section 5, para 2: with regrads to → with regards to

Section 5, para 3: were improved better → were improved more

Review 3

This paper seems to show that programs can be constructed that perform better using one algorithm (for MiniKanren) than another (ECCE for Prolog). As mentioned on page 2, ”there are no ways to predict the effect of specialization on a given program in general”. So it is hard to see what general conclusions can be drawn from this empirical study.

Many parts of the paper are written clearly but overall, more clarification of the purpose and significance of the experiment is needed.

In this paper we demonstrated some issues which arise in miniKanren *specialization and proposed an approach to deal with them. The proposed algorithm does not yet demonstrate the degree of specialization we strive for. We still consider it a step towards our goal.*

This paper reminds me of some of the discussions about control of partial evaluation of logic programs in the 1990s. It seems that some of these issues are being rediscovered in the context of MiniKanren, for example the handling of deterministic choices. See for example Section 4 of [2].

MiniKanren and Prolog are similar languages, as shown by the experiments in which programs are translated from one language to the other. However they have differences in their execution strategy and the translation could affect the execution. This is a vital aspect that needs to be addressed. Each language implementation is optimised for its respective execution strategy.

Since the languages are similar, it was natural to try solutions for specialization which already exist first. Hence we run the experiments and discovered that CPD did not solve our problem. If CPD worked as well for miniKanren *as we expected, we would just apply it and move on to the next challenge.*

However, all the experimental results seem to be achieved by running the programs as MiniKanren programs. I.e. a Prolog program extracted from ECCE is executed as a MiniKanren program!

This seriously detracts from the value of the experiment. I think that the experiments should go both ways; i.e. run all the programs both as Prolog and as MiniKanren.

Although running miniKanren *relations as* Prolog *programs is a nice experiment on its own, it does not advance us to the goal of running relational interpreters backwards efficiently, thus we did not do it.*

Detailed comments.

page 1. impelement ⇒ implement

page 2. “What is worse, the efficiency of residual program from the target language evaluator point of view is rarely considered in the literature.” This is certainly not true of PE for Prolog. See for example references [1], [2].

Thank you for the referces. We added them into the paper.

page 3. While it is true that CPD (ECCE) optimises programs for a left-to-right strategy. CPD can unfold any atom in a conjunction. This preserves its logical semantics. It is not true to state that CPD only “unfolds atoms from left to right”.

We realize that CPD can use different unfolding strategies. We were referring to the settings which proved to be the most beneficial in [16].

page 3. The strategy for CPD provided by ECCE can be varied, as mentioned. However, the experiments apparently only use the default settings.

We did use only the default settings, see the previous answer.

page 3. “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.” This is misleading, possible false. Unfolding the leftmost atoms can clearly NEVER increase the size of the search space when executing left to right. It merely pushes forward some choices (which often can improve indexing in the head clause and improve performance). So what is meant by "larger global trees" and "less efficient programs". Please provide a clear example.

However, unfolding non-deterministic non-leftmost atoms CAN increase the search space of a left-right execution, since goals to the left of the unfolded atom are duplicated.

All this was mentioned in the context of MINIKANREN, *where the execution strategy is not left to right, not in the context of* PROLOG. *We fixed this in the text.*

page 3. A feature of CPD that seems to be ignored is that a conjunction can be unfolded, i.e. it is not always "atom-by-atom" but a conjunction p,q can be treated as a single predicate pq. This allows more powerful specialisations, loop fusion, etc. It is not clear whether MiniKanren can achieve these specialisations.

We realize that CPD treats conjunctions as a single predicate. Conservative partial deduction does exactly that too: once a conjunction is encountered which is a renaming of another conjunction, we do not proceed to split it or unfold any of the relation calls within it. We note that it is a renaming and residualize it into a new predicate in the similar manner CPD does.

page 4. Note that putting formulas in canonical form (disjunction of conjunctions) blows up the size of the program and can also increase the search. This is a critical part of the experiment that needs to be addressed. It would be better to define a new predicate for each construct, and then interpret it. E.g.

```
p :- s,(q \/ r).
```

should be translated as

```
p :- s,qr.
qr :- q.
qr :- r.
```

rather than

```
p :- s,q.
p :- s,r.
```

Executing the second form repeats execution of s, but the first form does not. If the normalize function uses the second form, it casts serious doubt about the whole experiment.

This is an important comment, thank you. Normalization does deteriorate the performance of programs in the exact cases you described. However duplicating s may also prove beneficial, if running it with q or r leads to contradictions. There is no way to spot this kind of interactions without duplicating calls first. Making sure that such duplication does not seep into the transformed program is the task for residualization. For example, some kind of common subexpression elimination can be employed fot it.

page 5. The mechanism of generalisation seems very crude. It might work for the given examples but seems too simple to be used in general. More powerful generalisation using abstract interpretation and related methods have been widely studied.

The decision to generalize only by splitting was made in pursuit of lowering the algorithm's complexity and in attempt to single out the effects our specialization method has on the program. It is not hard to integrate more intricate generalization. This is left for future work.

page 6. Please explain more clearly why the two problems were chosen. It is stated that they are examples of relational interpreters to solve search problems. This is a special kind of program - can the results be expected to apply to other programs?

The goal of our project is to improve backwards execution of relational interpreters (see section 2.2 for context). We chose these two programs as examples to demonstrate the common issues faced when dealing with automatically generated relational interpreters. The examples are meant to be small and observable, so the reader can fully comprehend them.

The issues are not specific for interpreters: any program may have relation calls which restrict substitutions considerably, but are not placed leftmost within a conjunction. Also, one of the best pratices in programming is to not write the same code twice, thus many relations use calls to other relations in their definitions. Multiple unfoldings are often needed to get useful results in such a case. Thus we believe our approach will apply to other kinds of relational programs.

page 9, Fig 3. It should be made absolutely clear whether the programs are all run in MiniKanren (not Prolog). In my opinion this casts doubt on the validity of the experiments, since ECCE is designed to optimised execution for Prolog.

We added the explanation in the first paragraph of section 6

[1] Raf Venken, Bart Demoen: A Partial Evaluation System for Prolog: some Practical Considerations. New Gener. Comput. 6(2&3): 279-290 (1988)

[2] J.P. Gallagher: Tutorial on Logic Program Specialisation" (PEPM 1993).

Review 4

The paper reports on testing the well-known ECCE partial deduction system, in order to compare with a partial deduction for a simplest model variant of Prolog named relation language system being developed by the authors.

miniKanren *was not developed by the authors. We clarified it in the introduction and added a background subsection on* miniKanren.

It is surely in the scope of the VPT workshop. The submitted paper is quite good smoothly written. It is rather a kind of a tutorial based on the simplest model relation language than a research paper. I have actually found no research news in this paper. The authors are studying the partial deduction method and try, with a youthful enthusiasm, to dubiously comment some aspects of the simplest version of ("positive") supercompilation, experimenting with simple program samples, and fix their understanding "on the white lists of paper". Nevertheless, studying the program specialization methods is laudable and should be supported. The introduction, the section "Related works", and the section "References" take five pages from 13 pages used for the whole paper, i.e., these three sections together take more than 38 percents of the paper size. Unfortunately, the paper contains no transformation example of miniKanren program given in details, thus the reader is not able to follows the details and be sure in soundness of the short comments.

We added an example: see section 5.

The paper authors do not refer to many well known published papers that of course were read by them and indirectly used in the submitted tutorial.

We added citations to more of the papers read by us, as well as to the papers the reviewers suggested.

Since the authors do not explicitly formulate their contribution, then I think perhaps the authors understand themselves that the "conservative partial deduction" was studied in published literature and its variants were used in a number of partial deduction systems.

Links to these papers would be appreciated.

Thus I am forced to estimate this submission only with "a border paper" score.

Abstract:

1. We identify a number of issues, caused by MINIKANREN peculiarities, and describe a novel approach to specialization based on partial deduction and supercompilation. — This reviewer is not sure the described approach is novel, but it is certainly useful for empirical study of partial deduction.

Sec. Introduction
Page 1.

2. — In the first three paragraphs of this section the authors write their current understanding the basic concepts of the logical programming, assuming the readers are not familiar with the terms of ”relation language”, ”partial deduction”, and ”supercompilation”. I think the whole account of these paragraphs can be written in a couple sentences.

   Our intent was to write a self-contained paper which can be understood by readers with different backgrounds.

3. Specialization or partial evaluation [7] is a technique aimed at improving the performance of a program given some information about it beforehand. — Specialization is a wider concept, than the partial evaluation. The last one is just one of the program specialization methods.

   We agree. We rewrote the sentence to avoid confusing the two terms.

4. Typo: (i.e. the length of an input list) $\rightarrow$ (e.g. the length of an input list)

   Fixed.

5. Control issues in partial deduction of logic programming language PROLOG have been studied before [13]. Unlike Prolog, where atoms in the right-hand side of a clause cannot be arbitrarily reordered without changing the semantics of a program, in MINIKANREN the subgoals of conjunction/disjunction can be freely switched. This opens a yet another possibility for optimization, not taken into account by approaches initially developed in the context of conventional logic programming. — Actually, studying a program specialization in terms of any programming language starts (and goes on) only for a pure functional fragment of the language.

Page 2.

6. In this paper, we study issues which conjunctive partial deduction faces being applied for MINIKANREN. We also describe a novel approach to partial deduction for relational programming, conservative partial deduction. We implemented this approach and compared it with the existing specialization system (ECCE) for several programs. We report here the results of the comparison and discuss why some MINIKANREN programs run slower after specialization. — It is the contribution? https://github.com/JetBrains-Research/OCanren

   No, this is not the contribution. However, we used this miniKanren *implementation to run the programs mentioned in the paper. The contributions of the paper are listed in the last paragraph of the Introduction.*

Sec. RelatedWork
Page 2.

7. Specialization is an attractive technique aimed to improve the performance of a program if some of its arguments are known statically.

   — It is wrong. Actually the program specialization takes also into account contexts of intermediate computation, for example, composition of function (or relation) calls as your use themselves, trying to permutate calls of the commutative and associative basic logical connectives.

   The sentence was rewritten to reflect this wider context.

8. The heart of supercompilation-based techniques is driving - a symbolic execution of a program through all possible execution paths.

— As a rule, a program cannot be executed "through all possible execution paths", since the number of the paths are usually unbounded and lengths of the most of the paths are usually infinite. Your next sentence "The result of driving is so-called process tree where nodes correspond to configurations which represent computation state." contradicts to the previous one.

A process tree can be infinite, thus representing an unbound number of the paths. Infinite data structures are not something particularly hard to deal with both in code and in theoretical reasoning.

9. The two main sources for supercompilation optimizations are aggressive information propagation about variables' values, equalities and disequalities, and precomputing of all deterministic semantic evaluation steps.

   — It seems to me you mean only deterministic programming languages. Hence, by definition, any "deterministic semantic evaluation step" is deterministic. It should be something like the following. "...precomputing all evaluation actions that can be uniformly done over unknown values of the program variables".

   By deterministic semantic step we mean the situation in which there is a node in the process tree which has only one branch — also known as transient nodes.

10. Generalization, abstracting away some computed data about the current term, makes folding possible.

    — As a rule, a (local) generalization is a two-arity function depending on two configurations and, maybe, some nontrivial properties of the configuration automatically discovered by supercompilation before the corresponding generalization invocation starts.

11. The accumulating parameter can be removed by replacing the call with its generalization.

    — As a rule, the accumulating parameter cannot be removed by generalization; even more the task to recognize the fact that a parameter accumulating is undecidable.

    We are not stating that generalization can identify any accumulating parameter and remove it. Sometimes accumulating parameters can be identified, for example, by homeomorphic embedding, and then removed by most specific generalization.

12. There are several ways to ensure process correctness and termination, most-specific generalization (anti-unification) and homeomorphic embedding [6, 10] as a whistle being the most common.

    — In general, the most-specific generalization is not able prevent nontermination. Actually, it depends on the data set of the corresponding programming language. For example, if the data set is the set of finite strings in a given alphabet, then your statement is wrong. I.e. there are functional programming languages manipulating the character strings, rather than the LISP lists. In the case a programming language is Turing complete, using the most-specific generalization, as a rule, does not lead to any interesting transformations.

    We were referring to the resutls on the convergence of transfromers from the paper [28].

13. 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 [7].

    — The authors do not understand that any (physical) computing system specified by a given program is unclosed. Hence, its performance efficiency depends on an external environment. The environment comprises, for example, (maybe) a number of sequential compilers transforming the source and residual programs to a code supported by hardware, and the hardware properties.

We understand this. However abstracting away from such details is a common practice in estimating how well some transformation works.

Page 3.

14. Unfortunately there is neither an automatic procedure to choose what control setting is likely to improve input programs the most nor any informal recommendations on how to choose the best settings.

    — The sentence above should be rewritten.

    The sentence was rewritten.

Sec. 3 Conservative Partial Deduction
Page 4.

15. A driving process creates a process tree, from which a residual program is later created.

    — The driving does not include the generalization and folding actions. Unfortunately, the authors being freshmen and a fresh woman do not understand the basic concepts of supercompilation.

    We mentioned generalization and folding in the sentence.

16. — In this section a transformation example demonstrating the "conservative partial deduction" algorithm should given in details. You have spent only 13 pages and have at least two pages for such an example. There are many problems hidden behind the pseudocode given in Figure 1. Without such an example it is not certainly that the authors understand the problems. For example, what is the strategy looking for the configuration C'? (1) Does the configuration belong to the path rooted at the initial goal and ending at the configuration C'? (2) Or, maybe, does your algorithm look among all generated configurations in the process tree? What is happened when there are at least two such configurations? Following the pseudocode one may guesses that the strategy follows the (1)-st variant. Actually, it is not the best choice. As far as I know the most specialization methods based on the the (2)-nd variant. It is quite natural since the (1)-st strategy leads to generating lager ! residual programs as compared with the (2)-nd one (or code duplication).

    Currently, the less branching heuristics guides the choice of the suitable configuration. It works in the following manner. First it searches for the a static call within a conjunction. If there is no static calls, it then tries to find a deterministic call. If there are none, then it searches for the call which branchess less, than it does when all arguments are distinct free variables. It does select the leftmost suitable call, but only in the order described. If the leftmost call is less-branching, but the rightmost call is static, then the rightmost call will be selected and then relaced in the conjunction with the result of its unfolding.

17. `heuristically_select_a_call( r_1 , ... , r_n )`

    — Hence, you have no idea on the strategy choosing the next atom. Actually, the set of the strategies controlling the supercompilation process is the key problem hidden by you, since the optimization task per se is undecidable.

    The strategy of choosing the next atom is exactly what is described in the section 4.2

18. The substitution computed at each step is also stored in the tree node, although it is not included in the configuration.

    — What does that mean? Only either an example mentioned above or a precise definition may provide information on the subject.

It means that the substitution is stored in the process tree, since it is later used in residualization. Substitutions are ignored by the whistle: only goals in configurations are checked for renaming and embedding. We clarified it in the paper.

19. Each other disjunct takes the form of a possibly empty conjunction of relation calls accompanied with a substitution computed from unifications. Any MINIKANREN term can be trivially transformed into the described form. The function normalize in Fig. 1 is assumed to perform term normalization. The code is omitted for brevity.

    — The code should be presented somewhere in the paper or in an appendix.

    The implementation of the approach can be found on the github, the link is added into the paper.

20. There are several core ideas behind this algorithm.

    — Ok. See my remarks above.

21. The first is to select an arbitrary relation to unfold, not necessarily the leftmost which is safe. heuristically_select_a_call stands for heuristics combination, see section 3.2 for details ...

    — Let us see below.

22. The second idea is to use a heuristics 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

    — It is not a news. There is a huge literature devoted to this subject. For example, a large series of works by A. Pettorossi, M. Proietti and theirs coauthors, unfortunately, not cited by the authors of the paper being reviewed.

    The list of these works would be greatly appreciated.

23. — Typo (here and many times along the papers): a heuristics -$\longrightarrow$ a heuristic

    Fixed.

Page 5.

24. In this approach, we do not generalize in the same fashion as CPD or supercompilation. Our conjunctions are always split into individual calls and are joined back together only if it is meaningful. If the need for generalization arises, i.e. homeomorphic embedding of conjunctions [3] is detected, then we immediately stop driving this conjunction (line 12). When residualizing such a conjunction, we just generate a conjunction of calls to the input program before specialization.

    — In order to make convincing the decisions above you have to demonstrate them by means of interesting non-trivial examples.

    We believe the examples in the evaluation are interesting, non-trivial, and demonstrate the benefits of the decisions made. The fact that we only do splitting in generalization and do not perform most-specific generalization of the terms embedded was an attempt to reduce the complexity of the system so that it can be easier to comprehend. This limitation is not set in stone is not hard to remove.

Sec. 3.1 Unfolding

25. Unfolding too much may create extra unifications, which is by itself a costly operation, or even introduce duplicated computations by propagating the results of unfolding onto neighbouring conjuncts.

— Actually none can reason on any efficiency without fixing an efficiency model. As I have noted above the (physical) computing system is unclosed. The efficiency can be seen as kind of energy of the computing system. The task we are interested in is not quite mathematical and sometimes we should look for physical arguments.

26. We believe that the following heuristics provides a reasonable approach to unfolding control.

    — I do not think that the statement above looks convincing. Do you?

    — Only a meaningful list of interesting program transformation examples may convince that.

    We believe the examples in the evaluation are interesting, non-trivial, and demonstrate the benefitsof the decisions made.

3.2 Less Branching Heuristics
Page 6.
4 Evaluation

27. ECCE is designed for PROLOG programming language and cannot be directly applied for programs, written in MINIKANREN. To be able to compare our approach with ECCE, we converted each input program first to the pure subset of PROLOG, then specialized it with ECCE, and then we converted the result back to MINIKANREN. The conversion to PROLOG is a simple syntactic conversion. In the conversion from PROLOG to MINIKANREN, for each Horn clause a conjunction is generated in which unifications are placed before any relation call.

    — Actually ECCE can be (almost) directly applied for programs, written in MINIKANREN. You have to implement an interpreter of MINIKANREN in Prolog and specialize the interpreter w.r.t. your MINIKANREN programs. I expect you are not able to specialize a given Prolog interepreter, written in MINIKANREN, w.r.t. any Prolog program. Are you able?

    This would definitely be a fun experiment to run. We did not do it, if that is what you are asking for.

Page 7.

28. 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. The tables and graphs use the following denotations. Original represents the execution time of a program before any transformations were applied; ECCE - of the program specialized by ECCE with default conjunctive control setting; ConsPD-of the program specialized by our approach. Figure 3: Execution time of eval^o.

    — I guess you have a series of examples demonstrating a different behaviour of ConsPD as compared with ECCE, but hide such examples. Do not have you?

    ConsPD currently fails to perform deforestation and tupling on some programs which ECCE successfully transforms. We added a section 6.3 which discusses it. Battling this shortcomming is an ongoing project.

    — Have you explored some properties of the operating systems when the examples were specialized by both ConsPD and ECCE? For instance, how much of RAM was occupied, how many program were loaded in the operating system before starting and during the specialization?

    We did not, since currently we are only interested in the execution time.

Sec. 4.1 Evaluator of Logic Formulas

29. We specialize the eval^o relation to synthesize formulas which evaluate to ”true”. To do so, we run the specializer for the goal with the last argument fixed to ”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.

    — Those are interesting experiments. The specialization results are quite expected.

    Thank you.

Sec. 4.1.1 The Order of Relation Calls
Page 8.

30. Knowing that res is “true”, we can conclude that in the call and^o v w res variables v and w have to be “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. CPD fails to recognize this, and thus unfolds recursive calls of eval^o applied to fresh variables. It leads to over-unfolding, large residual programs and poor performance.

    — The problem discussing by you above is a problem of parallel evaluation. Parallel evaluation is natural for the relation programming. It is widely known that the simplest way to solve the problem is to use the breadth-first unfolding.

    If we understand what you meant correctly, then breadth-first unfolding would not be enough since there is often a need to synchronize how much individual relation calls are unfolded to find contradictions.

Sec. 4.1.2 Unfolding of Complex Relations

31. 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.

    — Here, for the first time in this paper, you use the “driving” term in a correct context, writing on “a different number of driving steps”. Given an interpreter (an operation semantics of a programming language), the interpreter has a concept of a step repeated during evaluation (computation) of the programs on their given/fixed input data. The driving is a meta-extension of the step over the parameterized input data. Unfortunately, the data sets of Prolog and relation programming languages themselves include parameters named “free variables”. That leads to a confusion.

Page 9.

32. Typo: Figure 3: Execution time of evalo $\rightarrow$ Figure 3: Execution time of eval^o

    Fixed.

Sec. 4.1.4 The Order of Answers
Page 10.

33. We believe that, in general, it is not possible to guarantee the same order of answers after specialization.

    — That is evidently since, in general, the problem discussed is undecidable.

    Rephrased

Sec. 4.2 Typechecker-Term Generator
Page 11.

34. For example, typechecking of the sum of two terms in the hand-written implementation consists of a single conjunction (see Listing 5) while the generated program is far more complicated and also uses a special relation typeEq^o to compare types (see Listing 6).

    — Here the residual programs should be specialized once again. Is able your model specializer discussed to remove such redundant relation definitions from the residual programs?

    Our specializer is able to remove many redundant relation definitions. We did not attempt running the specializer multiple times.

Sec. 5 Conclusion
Page 12.

35. We compared this approach with the most sophisticated implementation of conjunctive partial deduction- ECCE partial deduction system-on 6 relations which solve 2 different problems.

    — Do your really "believe" that such a benchmark size is quite convincing? Don't you?

    We agree that the evaluation is not convincing as a proper benchmark set. Our goal was to illustrate address the issues which arise often in relational interpreters. Adding more non-trivial programs into the evaluation is an ongoing project.

Thanks for your enthusiasm.

Review 5

Specialization is a transformation of a given program with respect to (a set of) given restrictions to its behavior. Relational programming offers the ability to run a program in various directions and to execute goals with free variables. The paper describes a novel approach to relational programs specialization. It aims at improving the conjunctive partial deduction (CPD) approach such that it accelerates the generated programs.

In CPD, the specialization is done at the local level, where the shape of the residual program is determined, and at the global level, where every relation in the residual program is defined. At the local level, CPD considers atoms one-by-one from left to right. The paper proposes heuristics to relax the order of atom consideration and to decide if unfolding a relation call can lead to the discovery of contradictions between conjuncts. This leads to the restriction of the answer set at the specialization run.

The main algorithm is described in pseudocode (Alg.1), but it is not easy to follow for readers without a proper background. Below are some suggestions to improve the presentation:

- residualize – is not explained.

  Added a paragraph in section 4 which explains residualization. The residualization is also mentioned further in the description of the algorithm.

- I feel the combination of functional notation and set-theoretic notation is confusing. For instance, line 1 suggests that drive is a function, but in line 2 drive is defined as disjunction drive_disj $\cup$ drive_conj. Then in line 5, drive_disj is defined using drive_conj. So, if I understand correctly, drive can be replaced just by drive_disj, and line 2 of the algorithm can be completely removed.

  You are right, driving is, in essence, just `drive_disj`. We wanted to avoid dealing with corner cases in pseudocode (when a disjunction contains only one disjunct, for example), and we see how it made the matters more confusing.

- What do the C@ and D@ prefixes denote?

  C and D are just names given to the pattern matched values. This is a piece of syntax we borrowed from Haskell. The explanation of the notation used is added before the pseudocode discussion.

- The presentation of the main idea of joining instead of splitting is somewhat dry. The motivation is more-or-less understandable, but the technical description is hard to follow. An example would greatly help here.

The paper also presents a heuristic to control the unfolding by computing if the unfolded relation contains fewer disjuncts than it could possibly have. Still, I suggest the authors polish the technical sections and add a running example.

We added an example: see section 5.

The related work section actually presents both the related work and background. For readability reasons, I suggest separating it into two sections; as well as introducing the notation and giving more background on MINIKANREN.

Done: see section2.

My main criticism is about the evaluation. There are two case studies: 1) a subset of propositional formulas and 2) type checking for a simple language. The approach is implemented in OCANREN (statically typed MINIKANREN embedding in OCAML) and compared to ECCE (the most mature implementation of CPD for PROLOG). The implementation of the approach (called ConsPD) outperforms ECCE almost always. But the running times in both cases are very short (seconds, or fractions of seconds). An experiment over larger benchmarks would look more convincing. Compare to SAT solvers – they are pretty good at solving formulas with thousands and millions of variables. The task considered in this paper is way simpler because no actual solving is done. I then suggest generating some really large propositional formulas and copmare ECCE and ConsPD on them.

We agree that the evaluation is not as remarkable as it should be. We are still working on the best way to specialize MINIKANREN. *These particular example programs were chosen to demonstrate common non-trivial issues which arise in* MINIKANREN *specialization. They were purposely kept short, so that we were able to point at specific issues: ordering of the relation calls, and how much they should be unfolded to get any useful data from it. Developing better, more comprehensive benchmarks is future work.*

To sum up, I think the paper gives an interesting contribution, but the presentation and experiments could be improved.