An Empirical Study of Partial Deduction for miniKanren

Anonymous author(s)

Anonymous institute(s)

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

1 Introduction

A family of embedded domain-specific languages miniKanren[1] impelement relational programming — a paradigm closely related to the pure logic programming. The main primitive in miniKanren is *term unification*. Unifications are combined into *conjunctions* and *disjunctions* which form *relations*. Relations can call other relations and be recursive. A program in miniKanren consists of a set of relations. To run a miniKanren program, one specifies a *goal*: a miniKanren expression with some free variables. miniKanren searches for the values of the free variables which satisfy given goal.

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. Having specified a relation for adding two numbers, one can also compute the subtraction of two numbers or find all pairs of numbers which can be summed up to get the given one. Program synthesis can be done by running *backwards* a relational interpreter for some language. In general, it is possible to create a solver for a recognizer by translating it into miniKanren and running in the appropriate direction [14].

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 drastically in the other direction.

Specialization or partial evaluation [7] is a technique aimed at improving the performance of a program given some information about it beforehand. It

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

may either be a known value of some argument, its structure (i.e. the length of an input list) or, in case of a relational program, — the direction in which it is intended to be run. An earlier paper [14] showed that *conjunctive partial deduction* [3] can sometimes improve the performance of miniKanren programs. Unfortunately, it may also not affect the execution time of a program or even make it slower.

Control issues in partial deduction of logic programming language Prolog have been studied before [12]. The ideas described there are aimed at left-to-right evaluation strategy of Prolog. Since the search in miniKanren is complete, it is safe to reorder some relation calls within the goal for better performance. While sometimes conjunctive partial deduction gives great performance boost, sometimes it does not behave as well as it could have.

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

2 Related Work

Specialization is an attractive technique aimed to improve the performance of a program if some of its arguments are known statically. It is studied for functional, imperative and logic programing and comes in different forms: partial evaluation [7] and partial deduction [13], supercompilation [16], distillation [5] and many more.

The heart of supercompilation-based techniques is *driving* — a symbolic execution of a program through all possible execution paths. The result of driving is so-called *process tree* where nodes correspond to *configurations* which present computation state, for example, a term in case of pure functional programming languages. Each path in the tree corresponds to some concrete program execution. The two main sources for supercompilation optimizations are aggressive information propagation about variable values, equalities, and disequalities and precomputing of all deterministic semantic evaluation steps, i.e. combining of consecutive process tree nodes with no branching, also known as *deforestation* [17]. When the tree is constructed, the resulting, or *residual*, program can be extracted from the process tree by the process called *residualization*. Of course, process tree can contain infinite branches. *Whistles* — heuristics to identify possibly infinite branches — are used to ensure termination in supercompilation. If the whistle signalled during the construction of some branch, then something should be done to ensure termination. The most common approaches include either stopping driving the infinite branch completely (no specialization is done in this case and the source code is blindly copied into the residual program) or folding the process tree to a *process graph*. The main instrument to perform such a folding is *generalization*, i.e. abstracting away some computed data about the current term which makes folding possible. One source of infinite branches is con-

secutive recursive calls to the same function with an accumulating parameter: by unfolding such a call further one can only increase the term size which leads to nontermination. The accumulating parameter can be removed by replacing the call with its generalization. There are several ways to ensure process correctness and termination, the most common being *homeomorphic embedding* [6, 9] used as a whistle and most-specific generalization of terms.

While supercompilation generally improves the behaviour of input programs and distillation can even provide superlinear speedup, there are no ways to predict the effect of specialization on a given program in the general. What is worse, they rarely consider the residual program efficiency from the target language evaluator point of view. The main optimization source is computing in advance all possible intermediate and statically-known semantics steps at program transformation-time. Other criteria, like the size of the generated program or possible optimizations and execution cost of different language constructions by the target language evaluator, are usually out of consideration [7]. It is known that supercompilation may adversely affect GHC optimizations yielding standalone compilation more powerful [1, 8] and cause code explosion [15]. Moreover, it may be hard to predict the real speedup of any given program on concrete examples even disregarding problems above because of the complexity of the transformation algorithm. The worst-case for partial evaluation is when all static variables are used in a dynamic context, and there is some advice on how to implement a partial evaluator as well as a target program so that specialization indeed improved its performance [7, 2]. There is lack of research in determining the classes of programs which transformers would definitely speed up.

Conjunctive partial deduction [3] makes an effort to provide reasonable control for left-to-right evaluation strategy of Prolog. CPD constructs a tree which models goal evaluation and is similar to SLDNF tree, then a residual program is generated from this tree. Partial deduction itself resembles driving in supercompilation [4]. The specialization is done in two levels of control: the local control determines the shape of the residual programs, while the global control ensures that every relation which can be called in the residual program is indeed defined. The leaves of local control trees become nodes of the global control tree. CPD analyses these nodes at the global level and runs local control for all those which are new.

At the local level, CPD examines a conjunction of atoms by considering each atom one-by-one from left to right. An atom is *unfolded* if it is deemed safe, i.e. a whistle based on homeomorphic 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 [14].

ECCE partial deduction system [11] 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 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 choice of the proper control is left to the user.

An empitical study shown that the most well-behaved strategy of local control in CPD for Prolog is *deterministic unfolding* [10]. An atom is unfolded only if only one suitable clause head exists for it with the one exception: it is allowed to unfold an atom non-deterministically once for one local control tree. This means that if a non-deterministic atom is the leftmost within conjunction, it is most likely to be unfolded and introduce many new relation calls within the conjunction. We believe this is the core problem with CPD which limits its power when applied to miniKanren. The strategy of unfolding atoms from left to right is reasonable in the context of Prolog because it mimics the way programs in Prolog execute. But it often leads to larger global control trees and, as a result, bigger, less efficient programs. The evaluation result of a miniKanren program does not depend on the order of atoms (relation calls) within a conjunction, thus we believe a better result can be achieved by selecting a relation call which can restrict the number of branches in the tree. We describe our approach which implements this idea in the next section.

3 Conservative Partial Deduction

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

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

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

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

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

Fig. 1: Conservative partial deduction pseudo code

function `normalize` in fig. 1 is assumed to perform term normalization. The code is omitted for brevity.

There are several core ideas behind this algorithm. The first is to select an arbitrary relation to unfold, not necessarily the leftmost which is safe. The second idea is to use a 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 (line 14; `heuristically_select_a_call` stands for heuristics combination, see section 3.2 for details). If those contradictions are found, then they are exposed by considering the conjunction as a whole and replacing the selected relation call with the result of its unfolding thus *joining* the conjunction back together instead of using *split* as in CPD (lines 15–22). Joining instead of splitting is why we call our transformer *conservative* partial deduction. Finally, if the heuristics 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 heuristics selects a call to unfold (line 15), a process tree is constructed for the selected call *in isolation* (line 16). The leaves of the computed tree are examined. If all leaves are either computed substitutions or are instances of some relations accompanied with non-empty substitutions, then the leaves are collected and each of them replaces the considered call in the root conjunction (lines 19–20). If the selected call does not suit the criteria, the results of its unfolding are not propagated onto other relation calls within the conjunction, instead, the next suitable call is selected (line 22). According to the denotational semantics of miniKanren it is safe to compute individual conjuncts in any order, thus it is ok to drive any call and then propagate its results onto the other calls.

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

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

3.1 Unfolding

Unfolding in our case is done by substitution of some relation call by its body with simultaneous normalization and computation of unifications. The unfolding itself is straightforward however it is not always clear what to unfold and when to *stop* unfolding. Unfolding in functional programming languages specialization, as well as inlining in imperative one, is usually considered to be safe from the residual program efficiency point of view. It may only lead to code explosion or code duplication which is mostly left to a target program compiler optimization or even is out of consideration at all if a specializer is considered as a standalone tool [7].

Unfortunately, this is not the case for the specialization of relational programming language. Unlike functional and imperative, in logic and relational programming language unfolding may easily affect the target program efficiency [12]. Unfolding too much may create extra unifications, which is by itself a costly operation, or even introduce duplicated computations by propagating the unfolding results onto neighbouring conjuncts.

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

3.2 Less Branching Heuristics

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

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

Those relation calls which are neither static nor deterministic are examined with what we call the *less-branching* heuristics. 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 heuristics we precompute the maximum possible number of disjuncts in each relation and compare this number with the number of disjuncts when unfolding a concrete relation call. The maximum number of disjuncts is computed by unfolding the body of the relation in which all relation calls were replaced by a unification which always succeeds.

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

Fig. 2: Heuristics selection pseudocode

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

4 Evaluation

In our study we compared the new conservative partial deduction with the ECCE partial deduction system. 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 spelialized it with ECCE, and then we converted the result back to miniKanren. The coversion 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.

The set of benchmarks consist of both examples from the literature on partial deduction and programs of our own creation. The first two programs are well-known in conjunctive partial deduction literature [3]. The second two are introduced in the paper on relational interpreters [14]. The last two are examples of non-trivial relational interpreters which at the same time are observable. In the last two examples we explore how different implementations of the same relation affect the quality of specialization.

The `doubleAppend`^{o} program concatenates three lists of numbers by computing a conjunction of two calls of a relation which concatenates two lists. This

example is prominent in CPD literature, since it demonstrates *deforestation* — a transformation which gets rid of intermidiate datastructures. This transformation cannot be done with earlier partial deduction approachs, since the relation uses a variable shared between two relation calls within a conjunction which requires the transformation to treat conjunctions as a whole. We explore how the fact that our approach sometimes splits conjunctions ignoring the variable sharing affects the quality of transformation.

The `maxLength`^{o} program computes the maximal element of a list as well as this list length. In this program an input list is shared between the call of the relation to compute the maximal element and the relation to compute the length of the list. This sharing leads to the list being traversed twice during execution. This example is used to demonstrate *tupling*: the transformation in which multiple traversals of the same data structure are replaced with a single traversal which computes all the necessary results simultaneously. CPD performes tupling more often than our approach, and we explore how it affects the performance of the specialized program.

The `isPath`^{o} and `unify`^{o} relations demonstrate the relational interpretation approach in which a relation is generated from a functional program. Partial deduction is capable of improving the execution time of the generated relations in the given direction. These two relations are described in [14], thus we will only briefly describe them in this paper. Here the aim is to study if our approach can help to achieve comparable performance improvement as ECCE.

The `eval`^{o} relation implements an evaluator of a subset of propositional formulas. We consider four different implementations of this relation to explore how the way 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 `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 the functional program. We demonstrate how much these implementations differ in terms of performance before and after specialization.

In this study we only measured the execution time for the sample queries, averaging them over multiple runs. All examples of miniKanren relations in this paper are written in OCanren[2]. 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. The first two examples also contain *Ideal* row which contains the execution time of the ideal implementations of the relations presented in the paper [3].

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

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

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

Listing 1.1: Concatenation of three lists

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

Listing 1.2: Ideal implementation of concatenation of three lists

4.1 Concatenation of Three Lists

The relation `doubleAppend`^{o} concatenates three lists by a conjunction of two calls of the `append`^{o} relation (see Listing 1.1). These two calls share a variable — an intermediate list `ts`. The list `xs` is traversed twice to construct the result: first when `ts` is constructed and then when `ts` is traversed during the second call. This double traversal negatively impacts the execution time, but it can be removed by deforestation.

The better implementation of this relation (see Listing 1.2) does not traverse the first list twice. It first recursively copies `xs` into the beginning of `res`, and only when `xs` is fully consumed, it calls the `append`^{o} to concatenate `ys` and `zs`. This implementation is provided as the ideal implementation of `doubleAppend`^{o} in [3].

To automatically transform the initial program into the ideal implementation, the transformer should treat the conjuncts which share variables with care. Partial deduction splits any conjunction into its subconjunctions, regardless variable sharing, while CPD only splits conjunctions when potential non-termination is detected. Conservative partial deduction processes subconjunctions in isolation and then joins them back together, if it can help to restrict the search space. In this example, ConsPD fails to perform deforestation: ConsPD first splits the conjunction `append`^{o} `xs ys ts` ∧ `append`^{o} `ts zs rs` into two relation calls of `append`^{o}, then unfolds the first call, but since the second call is a renaming of the first one, it is not unfolded. The generated program is almost the same as the original: the only difference is that the fresh variables are introduced on the toplevel.

We run all programs in the forward and the backward direction. The forward direction `doubleAppend`^{o} `x y z` ? concatenates three given lists: the first one of length 700, the second and the third — of length 1. The backward direction `doubleAppend`^{o} ? ? ? `r` searches for the first 100 list triples whose concatenation gives the given list of length 700.

There is no significant difference in the execution time in the backward direction between different specialized versions (see Fig. 3). However the execution

	forward	backward
Original	34.8ms	1.6ms
ECCE	26.7ms	1.7ms
Ideal	26.9ms	1.7ms
ConsPD	47.6ms	1.7ms

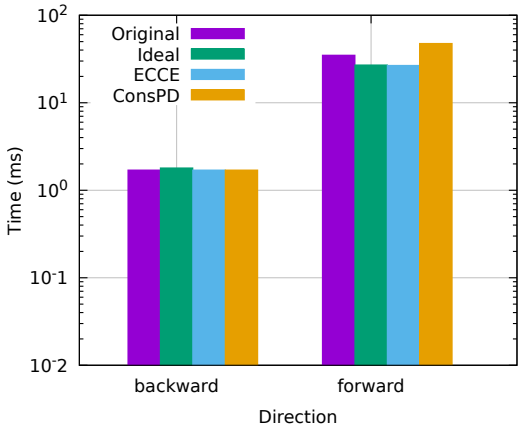

Fig. 3: Evaluation results for doubleAppendo

time in the forward direction differs: the program generated by ECCE shows the same speedup as the ideal program, while conservative partial deduction slows the program down. The reason behind the slowdown is the premature introduction of fresh variables which is significant for the input data of this size.

4.2 Maximal Element and Length of a List

The relation `maxLenght`^{o} computes the maximal element of the input list and its length (see Listing 1.3) by conjunction of two relation calls. The list `xs` is traversed twice: first to compute the maximum in the relation `max`^{o} and then to compute the length in the relation `length`^{o}. The input list contain Peano numbers which are compared by relations less-or-equal (`le`^{o}) and greater-than (`gt`^{o}). Note that the comparison relations have three arguments with the last one representing the result of comparison.

The ideal implementation provided in the CPD literature [3] is shown on Listing 1.4. This implementation traverses the input list `xs` only once and can be generated by tupling. Note, the disjunct which unifies the third argument with $\uparrow$ **false** [3] in the implementation of both `le`^{o} and `gt`^{o} can never contribute into the result of `maxLength`^{o} execution. Such disjuncts are removed by both ECCE and conservative partial deduction, thus we consider two ideal implementions: one with the original comparison relations (*Ideal*) and another in which these disjuncts are deleted (*Ideal removed*).

We measured the execution time of `maxLength`^{o} `lst m l`, where `lst` is a list of Peano numbers from 1 to 100. The results are presented in Fig. 4.

Both ECCE and ConsPD improve the performance of `maxLength`^{o} as compared to the original program. Removing the disjuncts with the result unified with $\uparrow$ **false** in the comparison relations improves the ideal program by about

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

```
let maxLength^o xs m l = max^o xs m ∧ length^o xs l
let rec length^o xs l = fresh (h t m) ( conde [
    (xs ≡ []     ∧ l ≡ zero);
    (xs ≡ h % t  ∧ l ≡ succ m ∧  length^o t m)])

let max^o xs m = max_1^o xs zero m
let rec max_1^o xs n m = fresh (h t) ( conde [
    (xs ≡ []      ∧  m ≡ n);
    (xs ≡ h % t) ∧ (conde [
      (le^o h n ↑true ∧ max_1^o t n m);
      (gt^o h n ↑true ∧ max_1^o t h m)])])

let rec le^o x y b = fresh (x_1 y_1) ( conde [
    (x ≡ zero    ∧ b ≡ ↑true);
    (x ≡ succ x_1 ∧ y ≡ zero    ∧ b ≡ ↑false);
    (x ≡ succ x_1 ∧ y ≡ succ y_1 ∧ le^o x_1 y_1 b)])
let rec gt^o x y b = fresh (x_1 y_1) ( conde [
    (x ≡ zero    ∧ b ≡ ↑false);
    (x ≡ succ x_1 ∧ y ≡ zero    ∧ b ≡ ↑true);
    (x ≡ succ x_1 ∧ y ≡ succ y_1 ∧ gt^o x_1 y_1 b)])
```

Listing 1.3: Maximum element and length of the list

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

Listing 1.4: Ideal implementation of maxlengtho

	[1..100]
Original	48.5ms
Ideal	32.2ms
Ideal removed	19.6ms
ECCE	23.7ms
ConsPD	35.7ms

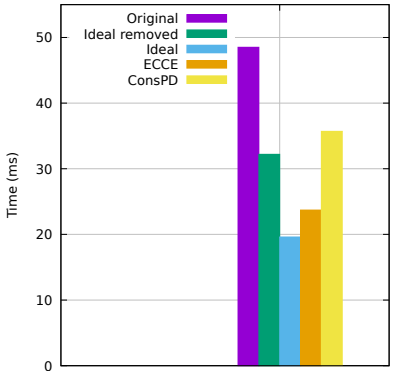

Fig. 4: Execution time of maxlengtho

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

Listing 1.5: Evaluator of formulas with boolean operation last

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

Listing 1.6: Evaluator of formulas with boolean operation second

40%. ConsPD removes disjuncts which never succeed, but does not perform tupling, thus its result is not as performant as the ideal program.

4.3 Evaluator of Logic Formulas

The relation `eval`^{o} describes an evaluation of a propositional formula under given variable assignments. The relation has three arguments. The first argument is a list of boolean values which plays a role of variable assignments. The i-th value of the substitution is the value of the i-th variable. The second argument is a formula with the following abstract syntax. A formula is either a *variable* represented with a Peano number, a *negation* of a formula, a *conjunction* of two formulas or a *disjunction* of two formulas. The third argument is the value of the formula under the given assignment.

We specialize the `eval`^{o} relation to synthesize formulas which evaluate to $\uparrow$**true**. To do so, we run the specializer for the goal with the last argument fixed to $\uparrow$**true**, while the first two arguments remain free variables. Depending on the way the `eval`^{o} is implemented, different specializers generate significantly different residual programs.

The Order of Relation Calls. One possible implementation of the evaluator is presented in Listing 1.5. Here the relation `elem`^{o} `subst v res` unifies `res` with the value of the variable `v` in the list `subst`. The relations `and`^{o}, `or`^{o}, and **not**^{o} encode corresponding boolean 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} `subst fm` $\uparrow$**true**), it fails to propagate the direction data onto recursive calls of `eval`^{o}. Knowing that `res` is $\uparrow$**true**, we can conclude that in the call `and`^{o} `v w res` variables `v` and `w` have to be $\uparrow$**true** as well. There are three possible

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

Listing 1.7: Implementation of boolean **not** 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 1.8: Implementation of boolean operation via `nand`

options for these variables in the call $\texttt{or}^o$ `v w res` and one for the call $\textbf{not}^o$. These variables are used in recursive calls of $\texttt{eval}^o$ and thus restrict the result of driving. CPD fails to recognize this, and thus unfolds recursive calls of $\texttt{eval}^o$ 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 heuristics. Since exploring the implementations of boolean connectives makes more sense, they are unfolded before recursive calls of $\texttt{eval}^o$. 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 1.6. This program is easier for ECCE to specialize which demonstrates how unequal is the behaviour of CPD for similar programs.

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 tables as demonstrated by the implementation of $\textbf{not}^o$ in Listing 1.7. 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 is, in turn, has a table-based implementation (see Listing 1.8). It will take several sequential unfoldings to derive that variables `v` and `w` should be ↑**true** when considering a call $\texttt{and}^o$ `v w` ↑**true** implemented via a basic relation. Conservative partial deduction drives the se-

	Original	ECCE	ConsPD
First-Plain	14.8ms	16.0ms	9.1ms
First-Nando	14.5ms	22.3ms	9.7ms
Last-Plain	7.4ms	14.0ms	9.1ms
Last-Nando	10.1ms	14.9ms	9.9ms

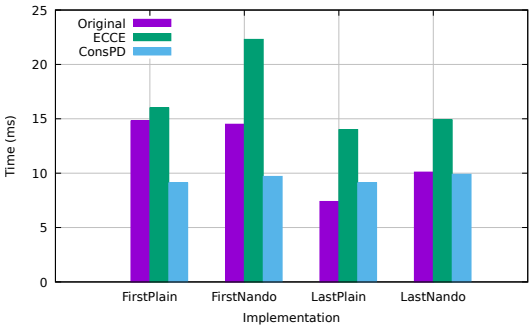

Fig. 5: Execution time of evalo

lected call until it derives useful substitutions for the variables involved while CPD with deterministic unfolding may fail to do so.

Evaluation Results. In our study we considered four implementations of `eval`^{o}:

- *FirstPlain* in which the implementations of boolean connectives are **table-based** and are placed **before** recursive calls to `eval`^{o}
- *LastPlain* in which the implementations of boolean connectives are **table-based** and are placed **after** recursive calls to `eval`^{o}
- *FirstNando* in which boolean connectives are implemented via `nand`^{o} and are placed **before** recursive calls to `eval`^{o}
- *LastNando* in which boolean connectives are implemented via `nand`^{o} andare placed **after** recursive calls to `eval`^{o}

These four implementations are very different from ECCE standpoint. We measured the time necessary to generate 1000 formulas over two variables which evaluate to $\uparrow$**true** (averaged over 10 runs). The results are presented in Fig. 1.

ConsPD 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 worsens performance of only the fastest implementation *LastPlain*, while improving it for others.

4.4 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.

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

The typing rules are straightforward and are presented below. Boolean and integer constants have the corresponding types regardless of the environment. Only terms of type integer can be summed up, multiplied or compared by 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. 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.

$$\frac{}{\Gamma \vdash IConst\ i : Int} \qquad \frac{}{\Gamma \vdash BConst\ b : Bool} \qquad \frac{\Gamma \vdash t : Int, \Gamma \vdash s : Int}{\Gamma \vdash t + s : Int}$$

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

$$\frac{\Gamma \vdash v : \tau_v,\ (\tau_v :: \Gamma) \vdash b : \tau}{\Gamma \vdash \underline{let}\ v\ b : \tau} \qquad \frac{}{\Gamma \vdash Var\ v : \tau}\ \Gamma[v] \equiv \tau \qquad \frac{\Gamma \vdash c : Bool, \Gamma \vdash t : \tau, \Gamma \vdash s : \tau}{\Gamma \vdash \underline{if}\ c\ \underline{then}\ t\ \underline{else}\ s : \tau}$$

We compared two implementations of these typing rules. The first one is obtained by unnesting of the functional program as described in [14] (*Generated*). The second program is 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 (averaged over 10 iterations; see Table 1).

	Hand-written	Generated
Original	916.9ms	11458.6ms
ConsPD	335.9ms	289.8ms
ECCE	218.6ms	382.0ms

Table 1: Running time of generating 1000 closed terms of type Int

Note that the generated program is far slower than hand-written. The principal difference between these two implementations is that the generated program contains a certain redundancy. For example, typechecking of the sum of two terms in the hand-written implementation is implemented by a single conjunc-

```
let rec typecheck^o gamma term res = conde [
  ...
  fresh (x y) ((term ≡ x + y ∧
    typecheck^o gamma x ↑(Some Integer) ∧
    typecheck^o gamma y ↑(Some Integer) ∧
    res ≡ ↑(Some Integer)));
  ...]
```

Listing 1.9: A fragment of hand-written typechecker

```
let rec typecheck^o gamma term res = conde [
  ...
  fresh (x y t1 t2) ((term ≡ x + y ∧
    conde [
      typecheck^o gamma x ↑None       ∧ res ≡ ↑None;
      typecheck^o gamma x ↑(Some t1) ∧
        typecheck^o gamma y ↑None     ∧ res ≡ ↑None;
      typecheck^o gamma x ↑(Some t1) ∧ typecheck^o gamma y ↑(Some t2) ∧
        typeEq^o t1 Integer ↑true    ∧ typeEq^o t2 Integer ↑true ∧
        res ≡ ↑(Some Integer);
    ])
  ...]
```

Listing 1.10: A fragment of generated typechecker

tion (see Listing 1.9) while the generated program is far more complicated and also uses a special relation `typeEq`^{o} to compare types (see Listing 1.10).

Most of the redundancy of the generated program is removed by specialization with respect to the known type of the term. This is why both implementations have comparable speed after specialization. The specialized programs differ too much and it is not clear why one implementation is better specialized by ECCE and the other by ConsPD.

4.5 Unification and Path Search

Besides evaluator of logic formulas we also run the specializers on the relation `unify`^{o} which searches for a unifier of two terms and a relation `isPath`^{o} specialized to search for paths in the graph. Both relations were generated from functional programs by unnesting. These two relations are described in paper [14] so we will not go into too many details here.

The `unify`^{o} relation was executed to find a unifier of three pairs of terms (see table 2) which vary in their complexity. The original program failed to terminate on two last unification problems in under 30s, while both ConsPD and ECCE generate programs which are able to find the unifiers.

The relation `isPath`^{o} was executed to search for 3 paths of length 7 in the graph with 20 vertices and 30 edges (see Table 2). Both specializers significantly improve the performance as compared to the original program, although ConsPD is about 30% worse than ECCE.

Terms	unify^o			isPath^o
	f(X, a) f(a, X)	f(a % b % nil, c % d % nil, L) f(X % XS, YS, X % ZS)	f(X, X, g(Z, t)) f(g(p, L), Y, Y)	
Original	0.23ms	—	—	24.9ms
ConsPD	0.07ms	16.5ms	397.6ms	1.6ms
ECCE	0.12ms	18.8ms	352.0ms	1.1ms

Table 2: Evaluation results of unification and path search

5 Conclusion

In this paper we discussed some issues which arise in the area of partial deduction techniques for relational programming language miniKanren. We presented a novel approach to partial deduction — conservative partial deduction — which uses a heuristics 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 10 relations which solve 6 different problems.

Our specializer improved the execution time of 10 queries to the relations, while worsening 2 queries and not affecting one query. ECCE worsened the performance of all 4 implementations of the propositional evaluator relation, while improving other queries. Conservative PD is more stable with regrads 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 better by ConsPD, while others — by ECCE. We conclude that there is still no 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.

References

1. Bolingbroke, M.C., Jones, S.L.P.: Supercompilation by evaluation. In: Gibbons, J. (ed.) Proceedings of the 3rd ACM SIGPLAN Symposium on Haskell, Haskell 2010, Baltimore, MD, USA, 30 September 2010. pp. 135–146. ACM (2010). https://doi.org/10.1145/1863523.1863540, https://doi.org/10.1145/1863523.1863540
2. Bulyonkov, M.A.: Polyvariant mixed computation for analyzer programs. Acta Inf. **21**, 473–484 (1984). https://doi.org/10.1007/BF00271642, https://doi.org/10.1007/BF00271642
3. De Schreye, D., Glück, R., Jørgensen, J., Leuschel, M., Martens, B., Sørensen, M.H.: Conjunctive partial deduction: Foundations, control, algorithms, and experiments. The Journal of Logic Programming **41**(2-3), 231–277 (1999)
4. Glück, R., Sørensen, M.H.: Partial deduction and driving are equivalent. In: International Symposium on Programming Language Implementation and Logic Programming. pp. 165–181. Springer (1994)

5. Hamilton, G.W.: 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 (2007)
6. Higman, G.: Ordering by divisibility in abstract algebras. In: Proceedings of the London Mathematical Society. vol. 2, pp. 326–336 (1952)
7. Jones, N.D., Gomard, C.K., Sestoft, P.: Partial evaluation and automatic program generation. Prentice Hall international series in computer science, Prentice Hall (1993)
8. Jonsson, P.A., Nordlander, J.: Taming code explosion in supercompilation. In: Khoo, S., Siek, J.G. (eds.) Proceedings of the 2011 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2011, Austin, TX, USA, January 24-25, 2011. pp. 33–42. ACM (2011). https://doi.org/10.1145/1929501.1929507, https://doi.org/10.1145/1929501.1929507
9. Kruskal, J.B.: Well-quasi ordering, the tree theorem, and vazsonyi's conjecture. vol. 95, pp. 210–225 (1960)
10. Leuschel, M.: Advanced techniques for logic program specialisation (1997)
11. Leuschel, M.: The ecce partial deduction system. In: Proceedings of the ILPS. vol. 97. Citeseer (1997)
12. Leuschel, M., Bruynooghe, M.: Logic program specialisation through partial deduction: Control issues. Theory and Practice of Logic Programming **2**(4-5), 461–515 (2002)
13. Lloyd, J.W., Shepherdson, J.C.: Partial evaluation in logic programming. The Journal of Logic Programming **11**(3-4), 217–242 (1991)
14. Lozov, P., Verbitskaia, E., Boulytchev, D.: Relational interpreters for search problems. In: miniKanren and Relational Programming Workshop. p. 43 (2019)
15. Mitchell, N., Runciman, C.: A supercompiler for core haskell. In: Chitil, O., Horváth, Z., Zsók, V. (eds.) 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, vol. 5083, pp. 147–164. Springer (2007). https://doi.org/10.1007/978-3-540-85373-2_9, https://doi.org/10.1007/978-3-540-85373-2_9
16. Soerensen, M.H., Glück, R., Jones, N.D.: A positive supercompiler. Journal of functional programming **6**(6), 811–838 (1996)
17. Wadler, P.: Deforestation: Transforming programs to eliminate trees. Theor. Comput. Sci. **73**(2), 231–248 (1990). https://doi.org/10.1016/0304-3975(90)90147-A, https://doi.org/10.1016/0304-3975(90)90147-A