An Empirical Study of Partial Deduction for miniKanren

EKATERINA VERBITSKAIA, DANIIL BEREZUN, and DMITRY BOULYTCHEV, Saint Petersburg State University, Russia and JetBrains Research, Russia

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.

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

Additional Key Words and Phrases: relational programming, partial deduction, specialization

ACM Reference Format:
Ekaterina Verbitskaia, Daniil Berezun, and Dmitry Boulytchev. 2020. An Empirical Study of Partial Deduction for miniKanren. 1, 1 (May 2020), 7 pages. https://doi.org/12.3456/7890123.4567890

1 INTRODUCTION

The core feature of the family of relational programming languages miniKanren[1] is their ability to run a program in different directions. Having specified a relation for adding two numbers, one can also compute the substraction of two numbers or find all pairs of numbers which can be summed up to get the given one. By running a relational interpreter *backwards* one can obtain a *solver* [7] thus solving a much more complicated problem.

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

Specialization or partial evaluation [4] is a technique aimed at improving performance of a program given some information about it beforehand. It may either be a known value of some argument, its structure (i.e. the length of an input list) or, in case of relational program, — the direction in which it is intended to be run. An earlier paper [7] showed that *conjunctive partial deduction* [3] can sometimes improve the performance of miniKanren programs. Unfortunately, it may also not affect the running time of a program or even make it slower.

Control issues in partial deduction of logic programming language Prolog have been studied before [6]. 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 come relation calls within the goal for better performance. While sometimes conjunctive partial deduction gives great perforamnce boost, sometimes it does not behave as well as it could have been.

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

Authors' address: Ekaterina Verbitskaia, kajigor@gmail.com; Daniil Berezun, daniil.berezun@jetbrains.com; Dmitry Boulytchev, dboulytchev@math.spbu.ru, Saint Petersburg State University, Russia , JetBrains Research, Russia.

XXXX-XXXX/2020/5-ART $15.00
https://doi.org/12.3456/7890123.4567890

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

2 RELATED WORKS

While the aim of all meta-computation techniques is residual program efficiency, they rarely consider the residual program efficiency from the target language evaluator point of view. The main optimization source is precomputing all possible intermediate and statically-known semantics steps at transformation-time while other criteria like residual program size or possible optimizations and execution cost of different language constructions by target language evaluator are usually out of consideration [4]. It is known that supercompilation may adversely affect GHC optimizations yielding standalone compilation more powerful [1, 5] and cause code explosion [8]. Moreover, caused by transformer complexity it may be hard to predict real program speedup in each case on a bunch of examples even disregarding problems above. Wherein, for example, for partial evaluation, an apogee of useless is the use of all static variables in a dynamic context only while it is known to be usefull for some specific form of language processors [2, 4]. Anyway, the formalization problem of cases for useful transformers application is poorly studied.

3 CONJUNCTIVE PARTIAL DEDUCTION

Partial deduction is a specialization technique for logic languages.

Conjunctive partial deduction is a modification of it, in which conjunctions of atoms are considered as a whole.

An adaptation of conjunctive partial deduction to miniKanren was described in an earlier paper.

Conjunctive partial deduction has its problems. It is designed for Prolog which fixes the order in which atoms are evaluated. While driving, CPD consideres atoms from left to right which leads to the following problem. If an atom which can restrict the answer set for other atoms is last in the conjunction, then before it gets unfolded, CPD would unfold all the others which are calls for free variables. This leads to over-unfolding.

miniKanren is a language in which the set of answers does not depend on the order of calls within conjunction. The order only affects the running time. It would be lovely, if the specialization technique for miniKanren chooses the best order of conjuncts. By blindly implementing CPD, we fail to do so. This is why we created a novel specialization approach which we called non-conjunctive partial deduction.

4 NON-CONJUNCTIVE PARTIAL DEDUCTION

In this section we will describe a novel approach to specialization of relational programs. 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 performance of the input programs.

The high-level idea behind the algorithm is to select a relation call to unfold by using a heuristic which decides if the call can narrow down the answer set. A driving tree is constructed for the selected call in isolation. The leaves of the computed tree are examined. If all leaves are either computed substitutions or are calls to some relations accompanied with non-empty substitutions, then the leaves are collected and are put back into the root conjunction instead of the examined call. According to 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. The algorithm pseudocode is shown on Fig. 1.

A driving process creates a process tree, from which a residual program is later created. The nodes of process tree include a *configuraion* which describes the state of program evaluation at some point. In our case configuration

```
1  ncpd goal = residualize ∘ drive ∘ normalize (goal)
2  drive     = drive_disj ∪ drive_conj
3
4  drive_disj :: Disjunction → Process_Tree
5  drive_disj D@(c1, ..., cn) =
6    create_or_node ([ci ← drive_conj (ci)])
7
8  drive_conj :: (Conjunction, Substitution) → Process_Tree
9  drive_conj (C@(r1, ..., rn), subst) =
10   r1, ..., rn ← propogate substitution subst on r1, ..., rn
11   switch whistle (C) of
12     instance (C', subst')       → create_fold_node (C', subst')
13     embedded_but_not_instance → create_stop_node (C , subst )
14     otherwise →
15       r ← select_a_call (r1, ..., rn)
16       t ← drive ∘ normalize ∘ unfold (r)
17       if trivial ∘ leafs (t)
18       then
19         C' ← propagate_subst (C \ r, extract_subst (t))
20          drive C'[r ↦ extract_calls (t)]
21       else
22         t ⋀ drive (C \ r, subst)
```

Fig. 1. Non-conjunctive Partial Deduction Pseudo Code

is a conjunction of relation calls. The substitution computed at each step is also stored in the tree node, although it is not included into the configuration.

Each time we examine a conjunction of calls, we *split* them into separate nodes which are driven independently from each other. Among the relation calls we select one which is according to the heuristic is likely to narrow down the answer set. If the selected call does not suit the criteria, the results of its unfolding is not propagated onto other relation calls withing the conjunction and the next suitable call is selected.

This process does create branchings whenever a disjunction is examined. At each step we make sure that we do not start driving a conjunction which we have already examined. To do this, we check if the current conjunction is renaming of any other configuration in the tree. If it is, then we create a special node which then is residualized into a call to the suitable relation.

We decided not to do generalization in this approach. The generalization is used in supercompilation and partial deduction to ensure termination at the same time as some degree of specialization. The generalization of two terms is usually a *most-specific generalization*. Generalization is used to abstract away some information computed during driving. In conjunctive partial deduction generalization is modified to support treating of conjunctions. The generalization selects subconjuctions of two conjuncts which are similar (call to the same relation and their arguments have similar shape and distribution). For the subconjunctions selected a most-specific generalization is computed.

In our approach we only do splitting of a conjunction into individual relation calls. This makes any program with an accumulating parameter to be a problem. Sometimes when there is a need to do a proper generalization, it is in reality just an instance of some other goal within the tree and we can simply create a call there. Otherwise we are unable to meaningfully specialize such goal, but we can always just include the initial program in the residual program and call the corresponding relation.

4.1 Unfolding

Unfolding is a process of substitution of some relation call by its body with simultaneous computation of unifications. To unfold a relation call we do the following steps. First, the formal arguments of a relation are substituted for the actual arguments of the call in the body. All fresh variables get instantiated. The body is transformed into a canonical form (disjunction of conjunctions of either calls or unifications). All unifications are computed. Those disjuncts in which unifications fails are removed. Other disjuncts take form of a conjunction of relation calls accompanied with a substitution.

The most important question is when to unfold. Unfortunately, too much unfolding is sometimes is even worse than not enough unfolding. There is a fine edge between those. This problem is mentioned in the (CONTROL PAPER). We believe that the following heuristic provides a reasonable control.

4.2 Heuristic

The intuition behind the heuristic is to find those calls which are safe to unfold. We deem every static conjunct (non-recursive) 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.

The other more complicated case is when there are less disjuncts than there can possibly be. This signifies that at least one branch of computations is gotten rid of.

The final heuristic selects the first conjunct which suites either of the following cases. First we unfold those conjuncts which are static. Then — deterministic. Then those which are less branching. The last to be unfolded are those calls, which unfold to a substitution with not conjunction.

4.3 Residualization

Residualization is quite straightforward. A branching in the process tree becomes a disjunction. A split node becomes a conjunction. Computed substitution is residualized as a conjunction of unifications. A renaming node is just a call to a relation. Relations are created for configurations on which leaf nodes are renamed.

One other thing is that when some configuration is occurred within the tree which is an instance of a configuration for which a new relation is created, then we just create a call.

5 EVALUATION

In our study we compared the performance of our implementation of CPD, ECCE and the new non-conjunctive partial evaluator. We have also implemented the branching heuristic instead of the deterministic one in the CPD, just to ensure, it is not a cureall.

We used the following 4 programs to test the specializers on.

- Two implementations of an evaluator of logic formulas.
- A program to compute a unifier of two terms.
- A program to search for paths of a specific length in a graph

All these relations are relational interpreters. Their last argument is a boolean value which is $true$ if the other arguments are in relation and $false$ otherwise.

5.1 Evaluator of Logic Formulas

The relation $eval^o$ describes evaluation of a subset of first-order logic formulas in a given substitution. $eval^o$ has 3 arguments. The first argument is a list of boolean values which serves as a substitution. The i-th value of the list 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 a boolean value which is *true* if the given formula in the given substition is true and *false* otherwise.

One possible implementation of the evaluator in the syntax of OCanren is presented in listing 1. Here the relation $\texttt{elem}^o$ *substvres* unifies `res` with the value of the variable `v` in the list `subst`. The relations $\texttt{and}^o$, $\texttt{or}^o$, and $\textbf{not}^o$ encode corresponding boolean operations.

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

Listing 1. Evaluator of formulas with boolean operation last

Note, that the calls to boolean relations $\texttt{and}^o$, $\texttt{or}^o$, and $\textbf{not}^o$ are placed last within each conjunction. This poses a challenge to the CPD-based specializers. Conjunctive partial deduction unfolds relation calls from left to right, so when specializing this relation for running backwards (i.e. considering the goal $\texttt{eval}^o$ `subst fm` ↑**true**), it fails to propagate the direction data onto recursive calls of $\texttt{eval}^o$. It leads to over-unfolding, big residual programs and less than optimal performance.

Moving boolean operations to the left, as shown in listing 2, we get a program which is easier for CPD to specialize.

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

Listing 2. Evaluator of formulas with boolean operation second

The next complication for partial deduction is how much sequential unfoldings should be performed to restrict other calls in a conjunction in a meaningful way. In this example the way in which boolean relations are implemented affect the specialization result. The simplest way to implement these relations is with a table as shown in listing 4

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

Listing 3. Implementation of boolean **not** as a table

The other way to implement boolean operations is via one base boolean relation such as $\texttt{nand}^o$ which is in turn has a table-based implementation.

```
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
```

	last	plain
Original	>60.00s	>60.00s
Ecce	7.12s	9.22s
CPD	31.31s	5.46s
Non-CPD	4.99s	5.05s
Branches	17.21s	6.17s

Table 1. Searching for 10000 true logic formulas

	running time
Original	>300.00s
CPD	2.35s
Non-CPD	14.90s
Branches	>300.00s

Table 2. Searching for a unifier for terms f(X, X, g(Z, t)) and f(g(p, L), Y, Y)

```
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 4. Implementation of boolean operation via nand

We considered two implementations of `eval`^{o}: `eval`^{o}`_plain` and `eval`^{o}`_last` and studied how specializers behave on them. The first one uses table-based boolean operations and places them at the second place in each conjunction. The relation `eval`^{o}`_last` employs boolean operations implemented via `nand`^{o} and place them at the end of each conjunction. These two programs are complete opposites from the standpoint of CPD which the time measurements in table 1 confirmed.

We measured time necessary to generate 10000 formulas over two variables which evaluate to ↑**true**. We compared the result of specialization of the goal `eval`^{o} `subst fm` ↑**true** by our implementation of CPD, done by ECCE system and the new non-conjunctive partial deduction. Since ECCE cannot work directly on miniKanren programs, we first translated them to Prolog and then back: this translation is purely syntactical.

5.2 Search for a Unifier

The unification of two terms t and u is searching for a substitution θ such that $t\theta = u\theta$, θ is called a unifier. We search for any unifier, not necessarily most specific. The details on this benchmark can be found in the earlier paper CITE. This example demonstrates how too much unfolding can be introduced with the non-conjunctive partial deduction.

	running time
Original	19.86s
CPD	4.66s
Non-CPD	3.00s

Table 3. Searching for paths in a graph

5.3 Search for Paths in a Graph

Here we search for 5 paths in a graph. The details on this benchmark can be found in the earlier paper CITE.

6 CONCLUSION

We compared some of the specialization techniques for miniKanren.

There seem to be no one good technique.

REFERENCES

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

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

[3] Danny De Schreye, Robert Glück, Jesper Jørgensen, Michael Leuschel, Bern Martens, and Morten Heine Sørensen. 1999. Conjunctive partial deduction: Foundations, control, algorithms, and experiments. *The Journal of Logic Programming* 41, 2-3 (1999), 231–277.

[4] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. 1993. *Partial evaluation and automatic program generation.* Prentice Hall.

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

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

[7] Petr Lozov, Ekaterina Verbitskaia, and Dmitry Boulytchev. 2019. Relational Interpreters for Search Problems. In *miniKanren and Relational Programming Workshop*. 43.

[8] Neil Mitchell and Colin Runciman. 2007. A Supercompiler for Core Haskell. In *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)*, Olaf Chitil, Zoltán Horváth, and Viktória Zsók (Eds.). Springer, 147–164. https://doi.org/10.1007/978-3-540-85373-2_9