An Empirical Study of Partial Deduction for miniKanren

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

ABSTRACT

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, Dmitry Boulytchev, and Daniil Berezun. 2020. An Empirical Study of Partial Deduction for miniKanren. 1, 1 (May 2020), 3 pages. https://doi.org/12.3456/7890123.4567890

1 INTRODUCTION

The miniKanren language family is nice.

Unpredictable running time for different directions.

Specialization sometimes helps, sometimes does not, sometimes makes everything worse.

Control issues, blah blah blah

This problem also appear in supercompilation (partial evaluation, specialization) of functional and imperative programming languages.

<Here should be review of problems in specialization of functional languages>

We came up with the new specialization algorithm, which seems nice, but is as unpredictable as the others.

Here are some numbers...

Contribution is the following

- explanation of specialization
- new specialization algorithm
- why some programs behave worse after specialization

2 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

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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2020 Association for Computing Machinery.
XXXX-XXXX/2020/5-ART $15.00
https://doi.org/12.3456/7890123.4567890

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.

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 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.

2.1 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.

2.2 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.

2.3 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.

3 EVALUATION

In the evaluation we compared the performance of our 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 be sure, it is not a cureall.

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

- propositional evaluator
  - Compicated bool expression in the last conjunctive
  - Table implementation of the boolean operation, placed as a first conjunct
- searching for a unifier of two terms
- searching for paths of a specific length in a graph

The first one demonstrates the drawback of the approach implemented in ECCE: left-to-right deterministic unfolding. The boolean expression which narrows the first conjunts is the last, so ECCE does not unfold it until too late.

The second one should be easier for this approach.

The unification is a difficult problem, specialization of which is of importance.

4 CONCLUSION

We compared some of the specialization techniques for miniKanren.

There seem to be no one good technique.