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), 2 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.

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 [3]. It is known that supercompilation may adversely affect GHC optimizations yielding standalone compilation more powerful [1, 4] and cause code explosion [5]. 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, 3]. Anyway, the formalization problem of cases for useful transformers application is poorly studied.

<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

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.

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

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

2 NON-CONJUNCTIVE PARTIAL DEDUCTION

We came up with the idea.

Here we'll briefly describe it.

First we build a tree, than we generate a residual program from it. Residualization is more or less trivial, once the tree is built.

Building the tree is... Well... Complicated.

There are issues with when to unfold, when to stop.

There are a bunch of heuristics for it.

2.1 Unfolding

Unfolding is when a call to a relation is substituted by its body.

The most important question is when to unfold.

To much unfolding is sometimes is even worse than not enough unfolding.

2.2 Heuristic

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.

3 EVALUATION

Is going to be here.

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

[4] 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

[5] 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