Supercompilation Strategies of Relational Progams

Maria Kuklina
ITMO Univeristy
Saint Petersburg, Russia
kuklina.md@gmail.com

Ekaterina Verbitskaia
Saint Petersburg State Univeristy
Saint Petersburg, Russia
kajigor@gmail.com

In our work we research methods of supercompilation[5] in the context of relational program specialization. We implemented a supercompiler for MINIKANREN and propose different supercompilation strategies.

1 Introduction

Relational programming is a pure form of logic programming in which programs are written as *relations*. In relations "input" and "output" arguments are indistiguishable, therefore relational programs solve problems in general.

Interesting application of relational programming is *relational interpreters*. Besides computing the output form an input or running a program in an opposite "direction", relational interpreters are capable to both verify a solution and search for it.[4]

MINIKANREN is a family of domain-specific languages specially designed for relational programming.[1] Relational interpreters implemented in MINIKANREN show all their potential, however, in the context of a particular task computation performance can be highly insufficient.

Specialization is a technique of automatic program optimization. A *specializer* takes a program and a part of its input and produces a new program that behave the same way on the rest of its input as the original one on all of its input.[2] A specializer (in form of *conjunctive partial deduction, CPD*) has been implemented and applied to relational programs in MINIKANREN in [4]. Despite the fact that CPD gives a huge performance boost comparing to original programs, specialized programs still carries some overhead of interperation.

2 Supercompilation Strategies

Supercompilation is a method of program transformation. Supercompiler tries to symbolically execute a program for a given *configuration* – an expression with free variables – tracing a computation history with a *graph of configurations* and build an equivalent *residual* program removing redundant computation.

More formally, supercompilation's steps are following:

- **Driving** is a process of symbolic execution with resulting possibly infinite tree of configurations.
- The goal of **folding** is to avoid building infinite tree by turning it into a finite graph, from which the original infinite tree could be recovered. Usually it's done by adding a link to an ancestor if it's a renaming of handled configuration.
- **Generalization** is another way of avoiding an infinite tree, when no folding operations can be done. The aim of this step is to generate new goals which can be folded in finite time. Generalization step is applied only when a **whisle** decides it's neccessary.
- **Residualization** is a process of generating an actual program from a graph of configurations.

Submitted to:
TEASE-LP 2020

2.1 Supercompilation for miniKanren

We implemented a supercompiler for miniKanren using a *homeomorphic embedding* as a whisle and CPD-like abstraction algorithm for generalization. However, various strategies could be applied for a driving step.

A driving step of miniKanren supercompiler handles a configuration which takes a form of a conjunction of calls. For further computations a supercompiler has to decide which of the conjuncts to unfold.

We implemented several strategies:

- **Full unfold** strategy unfolding all conjuncts simulteniously. However, supercompilation time with full unfold strategy takes signigicant amount of resourcses, so we didn't shows a results of it for some tests.
- **First unfold** strategy always unfolds first conjunct.
- **Sequential unfold** strategy unfolds conjuncts in order.
- **Recursive unfold** strategy firstly unfolds conjuncts which have at least one recursive call.
- **Non-recursive unfold** strategy firstly unfolds conjuncts which don't have any recursive call.
- **Maximal size unfold** strategy firstly unfold conjuncts with the largest amount of conjunctions (in CNF).
- **Maximal size unfold** strategy firstly unfold conjuncts with the least amount of conjunctions (in CNF).

2.2 Results

We present testing results for the implemented supercompiler with described strategies and comparison with original interpreter and CPD specializer. As a specific implementation of miniKanren we use *OCanren* [3]; the supercompiler is written in Haskell.

First, we compare the performance of the solvers for path searching problem. We ran the search on a complete graph K_{10}, searching for path of lengths 9, 11, 13 and 15. The results are presented in **??**.

Second, we comapre the performance of generating of propositional logic formulas. We ran the search for 1000 formulas in empty substitution and in substitution with only one free variable. The results are presented in 2.2

References

[1] Oleg Kiselyov Daniel P. Friedman, William E. Byrd (2005): *The Reasoned Schemer*. The MIT Press.

[2] Neil D Jones, Carsten K Gomard & Peter Sestoft (1993): *Partial evaluation and automatic program generation*. Peter Sestoft.

[3] Dmitrii Kosarev & Dmitri Boulytchev (2018): *Typed Embedding of a Relational Language in OCaml*. *Electronic Proceedings in Theoretical Computer Science* 285, pp. 1–22, doi:10.4204/EPTCS.285.1.

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

[5] Morten Heine Sørensen & Robert Glück (1999): *Introduction to supercompilation*. In John Hatcliff, Torben Æ. Mogensen & Peter Thiemann, editors: *Partial Evaluation. Practice and Theory*, Lecture notes in computer science, Springer Verlag, pp. 246–270, doi:10.1007/3-540-47018-2_10.

Path length	9	11	13	15
Orignal	0.606s	3.98s	22.73s	120.48s
CPD	0.366s	2.27s	12.55s	63.12s
Full	0.021s	0.03s	0.035s	0.041s
First	0.014s	0.02s	0.021s	0.025s
Sequential	0.014s	0.02s	0.022s	0.027s
MaxU	0.014s	0.02s	0.022s	0.026s
MinU	0.014s	0.02s	0.022s	0.027s
RecU	0.018s	0.02s	0.021s	0.027s
NrcU	0.014s	0.02s	0.022s	0.026s

Table 1: Searching for paths in the K_{10} graph

Free variables in substitution	0 free vars	1 free var
Orignal	0.19s	0.28s
CPD	1.89s	3.33s
First	0.216s	0.15s
Sequential	0.150s	0.28s
Non recursive	0.050s	0.07s
Recursive	0.045s	0.45s
Maximal size	0.136s	0.19s
Minimal size	0.046s	0.06s

Table 2: Searching for formulas in a given substitution