Supercompilation Strategies for MINIKANREN

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

Ekaterina Verbitskaia
JetBrains Research
Saint Petersburg, Russia
kajigor@gmail.com

In this paper we research methods of supercompilation [**?**] in the context of relational program specialization. We implement a supercompiler for MINIKANREN with different unfolding strategies and compare them.

Relational programming is a pure form of logic programming in which programs are written as *relations*. In relations *input* and *output* arguments are indistinguishable, therefore a relational program solves a whole class of problems. For instance, an addition relation describes both addition and subtraction. An interesting application of the technique is *relational interpreters* — interpreters written in a relational language. Relational interpreters are capable to both searching for a solution to a problem and verifying the solution [**?**].

MINIKANREN is a family of domain-specific languages specially designed for relational programming [**?**]. Relational programs implemented in MINIKANREN are capable of running in all directions, however, in the context of a particular task, especially for running in the backward directions, computation performance can be highly inefficient.

Specialization is a technique of automatic program optimization. A *specializer* takes a program P and a part of its input I_s and produces a new program that behaves the same way on the rest of its input I_d as the original one on all of its input [**?**]: $[\![spec(P,I_s)]\!](I_d) \equiv [\![P]\!](I_s,I_d)$.

A specializer (in form of *conjunctive partial deduction, CPD* [**?**]) has been implemented and applied to relational programs in MINIKANREN in [**?**]. Despite the fact that CPD gives a performance boost compared to original programs in most cases, there are still possible ways for improvement.

Supercompilation is a method of program transformation. Supercompilers symbolically executes a program for a given *configuration* — an expression with free variables — tracing a computation history with a *graph of configurations* and builds an equivalent *residual* program without redundant computations. Supercompilation steps are the following.

Driving is a process of symbolic execution in compliance with reduction rules which results in a possibly infinite tree. Branching appears when there are several possible ways to run the computation. The goal of *folding* is to avoid building an infinite tree by turning it into a finite graph. When a supercompiler analyses a configuration it may happen to be a renaming of the one already encountered and hence would lead to the same configuration subtree. In this situation a supercompiler links their nodes in the graph avoiding repetition of computations. *Generalization* [**?**] is another way to avoid an infinite tree when no folding operations can be done. Generalized configurations are more general and possess less information about their arguments than original ones and can be folded eventually. A predicate called *whistle* holds when a generalization step is needed. *Residualization* is a process of generating a residual program from a graph of configurations.

We adapted supercompilation for MINIKANREN the following way. An expression in the core language is *a conjunction* of two expressions, *a disjunction* of two expressions, *a unification* of two terms or *a call* of a relational definition. Disjuncts are independent, hence give rise to branches while driving. Meanwhile, a conjunction of relational calls alongside with a computed substitution forms a configura-

Submitted to:
TEASE-LP 2020

tion.

First, we pop all fresh variables to the top level and transform a configuration to a disjunctive normal form (DNF). A first step in driving performs all unifications within a disjunct and applies a substitution computed from successful unifications on previous steps. What remains is the sequence of calls to be symbolically evaluated. For further computation a supercompiler must replace at least one of the calls with its definition — *unfold* it. Choosing a specific set of calls for unfolding may change time efficiency of the specialized code and quality of specialization, but it is not clear what unfolding strategy is the best for particular tasks.

We implemented a supercompiler for MINIKANREN using a *homeomorphic embedding* [**?**] as a whistle and CPD-like abstraction algorithm for generalization [**?**]. We implemented the following unfolding strategies:

- *Full unfold* unfolds all conjuncts simultaneously, so the resulting set of configurations is a Cartesian product of the normalized definitions of conjuncts.
- *Sequential unfold* unfolds conjuncts one by one.
- *Recursive unfold* prioritizes calls that have at least one recursive call.
- *Non-recursive unfold* prioritizes calls that do not have any recursive calls.
- *Maximal size unfold* prioritizes calls with the largest definitions.
- *Minimal size unfold* prioritizes calls with the smallest definitions.

We compare strategies with the original unspecialized interpreters and the CPD specializer. As a specific implementation of MINIKANREN we use OCANREN [**?**]; the supercompiler is written in HASKELL[1].

First, we compare the performance of a solver for a path searching problem *isPath(path, graph, result)*. The *result* parameter is *True* when *path* is a path of *graph*. The relation was specialized on a query *isPath(_, _, True)*. We ran the search on a complete graph K_{10}, searching for paths of lengths 9, 11, 13 or 15. An example of running a query is in Figure 1. The results are presented in Table 1.

```
run q (fun p → length p 9 &&& isPath p k_10)
```

Figure 1: Example of a query for the first test in OCANREN.

Here we can see that supercompilation gives a huge performance boost; however, the choice of a strategy doesn't have a notable impact. This leads to an interesting conclusion that for some programs it doesn't really matters how to supercompile them. In the case of the given relation, this is due to how its conjunctions are small in size.

Secondly, we compare the performance of synthesis of propositional logic formulas *int(formula, subst, result)* that have the value of *result* with the given substitution *subst*. The relation was specialized on a query *int(_, _, True)*. We ran the search for 1000 formulas with no free variables and with a single free variable. An example of running a query for searching for formulas with a single free variable is in Figure 2. The results are presented in Table 2.

In this example, we can see that the CPD approach sometimes actually worsens programs. Also, despite that the *Full* unfold strategy could give a huge performance boost on small programs, it is unable to handle big and complex relations with high branching. *Non-recursive* and *minimal size* strategies give the best results due to the fact that they perform constant propagation better. For the non-recursive

[1] `https://github.com/RehMaar/uKanren-spec`

```
run qr (fun f q → int f (!< q))
```

Figure 2: Example of a query for the second test in OCANREN.

strategy, the reason for it is that the given relation is comprised of non-recursive calls, and the minimal size strategy excels because the logic connectives have very simple definitions.

Path length	9	11	13	15
No specialization	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
Sequential	0.014s	0.02s	0.022s	0.027s
Non recursive	0.014s	0.02s	0.022s	0.026s
Recursive	0.018s	0.02s	0.021s	0.027s
Maximal size	0.014s	0.02s	0.022s	0.026s
Minimal size	0.014s	0.02s	0.022s	0.027s

Table 1: Searching for paths in K_{10}.

Free variables in formula	0 free vars	1 free var
No specialization	0.280s	0.195s
CPD	3.330s	1.893s
Full	-	-
Sequential	0.153s	0.090s
Non recursive	0.113s	0.047s
Recursive	0.205s	0.080s
Maximal size	0.157s	0.085s
Minimal size	0.068s	0.058s

Table 2: Searching for valid formulas in a given substitution

Based on our preliminary results we conclude that supercompilation is a viable approach for MINIKANREN specialization. We believe more research should be done to find less ad hoc unfolding strategies that are justified by program properties.

References

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

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

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

[4] 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.

[5] Michael Leuschel (1998): *On the Power of Homeomorphic Embedding for Online Termination*. In Giorgio Levi, editor: *Static Analysis*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 230–245.

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

[7] Morten Heine Sørensen (1998): *Convergence of program transformers in the metric space of trees*. In: *Mathematics of Program Construction. MPC 1998.*, *Lecture Notes in Computer Science* 1422, Springer Berlin Heidelberg.

[8] 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.