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), 9 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. Program synthesis can be done by running *backwards* a relational interpreter for some language. In general, it is possible to create a solver for a recognizer by translating it into miniKanren and running in the appropriate direction [11].

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 down the computation drastically in the other direction.

Specialization or partial evaluation [6] 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 [11] 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 [9]. 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 some relation calls within the goal for better performance. While sometimes

[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

conjunctive partial deduction gives great performance boost, sometimes it does not behave as well as it could have.

In this paper we show on examples some issues which conjunctive partial deduction faces. 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 WORK

Specialization is an attractive technique aimed to improve the performance of a program if some of its arguments are known statically. It is studied for functional, imperative and logic programing and comes in different forms: partial evaluation [6] and partial deduction [10], supercompilation [13], distillation [4] and many more.

The heart of supercompilation-based techniques is *driving* — a symbolic execution of a program through all possible execution paths. The result of driving is so-called *process tree* where nodes correspond to *configurations* which present computation state, for example, a term in case of pure functional programming languages. Each path in the tree corresponds to some concrete program execution. The two main sources for supercompilation optimizations are aggressive information propagation about variable values, equalities, and disequalities and precomputing of all deterministic semantic evaluation steps, i.e. combining of consecutive process tree nodes with no branching, also known as *deforestation* [14]. When the tree is constructed, the resulting, or *residual*, program can be extracted from the process tree by the process called *residualization*. Of course, process tree can contain infinite branches. *Whistles* — heuristics to identify possibly infinite branches — are used to ensure termination in supercompilation. If the whistle signalled during the construction of some branch, then something should be done to ensure termination. The most common approaches include either stopping driving the infinite branch completely (no specialization is done in this case and the source code is blindly copied into the residual program) or folding the process tree to fold in a *process graph*. The main instrument to perform such a folding is *generalization*, i.e. abstracting away some computed data about the current term which makes folding possible. One source of infinite branches is consecutive recursive calls to the same function with an accumulating parameter: by unfolding such a call further one can only increase the term size which leads to nontermination. The accumulating parameter can be removed by replacing the call with its generalization. There are several ways to ensure process correctness and termination, the most common being *homeomorphic embedding* [5, 8] used as a whistle and most-specific generalization of terms.

While supercompilation generally improves the behavior of input programs and distillation can even provide superlinear speedup, there are no ways to predict the effect of specialization on a given program in general case. What is worse they rarely consider the residual program efficiency from the point of view of the target language evaluator. The main optimization source is computing in advance all possible intermediate and statically-known semantics steps at transformation-time. Other criteria like the size of the generated program or possible optimizations and execution cost of different language constructions by the target language evaluator are usually out of consideration [6]. It is known that supercompilation may adversely affect GHC optimizations yielding standalone compilation more powerful [1, 7] and cause code explosion [12]. Moreover, it may be hard to predict the real speedup of any given program on concrete examples even disregarding problems above because of the complexity of the transformation algorithm. The worst case for partial evaluation is when all static variables are used in a dynamic context, and there is a lot of advice on how to implement a partial evaluator as well as a target program so that specialization really improved its performance [2, 6]. There is lack of research in determining the classes of programs which transformers would definitely speed up.

Conjunctive partial deduction [3] makes an effort to provide reasonable control for left-to-right evaluation strategy of Prolog. CPD constructs a tree which models goal evaluation and resembles SLD-NF tree. After the tree is constructed, a residual program is generated from it. The specialization is done in two levels of control: the local control determines the shape of the residual programs, while the global control ensures that every relation which can be called in the residual program is indeed defined. The leaves of local control trees become nodes of the global control tree. CPD analyses these nodes at the global level and runs local control for all those which are new.

At the local level CPD examines a conjunction of atoms by considering each atom one-by-one from left to right. An atom is *unfolded* if it is deemed safe. When an atom is unfolded, a clause which head can be unified with the atom is found, and a new node is added into the tree where the atom in the conjunction is replaced with the body of that clause. If there are more than one suitable head, then several branches are added into the tree which correspond to the disjunction in the residualized program. An adaptation of CPD for the miniKanren programming language is described in [11].

The most well-behaved strategy of local control in CPD for Prolog is *deterministic unfolding*. An atom is unfolded only if only one suitable clause head exist for it with the one exception: it is allowed to unfold an atom non-deterministically once for one local control tree. This means that if a non-deterministic atom is the left-most within a conjunction, it is most likely to be unfoldled and introduce a lot of We believe this is the core problem with CPD which limits its power when applied to miniKanren. The strategy of unfolding atoms from left to right is reasonable in the context of Prolog because it mimics the way it executes. But it often leads to larger global control trees and, as a result, bigger less efficient programs. The evaluation result of a miniKanren program does not depend on the order of atoms (relation calls) within a conjunction, thus we believe a better result can be achieved by selecting a relation call which can restrict the number of branches in the tree. We describe our approach which implements this idea in the next section.

3 NON-CONJUNCTIVE PARTIAL DEDUCTION

In this section we 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 algorithm pseudocode is shown on Fig. 1. For the sake of brevity and clarity, we provide functions `drive_disj` and `drive_conj` which describe how to process disjunctions and conjunctions respectively. Driving itself is a trivial combination of presented functions (line 2).

A driving process creates a process tree, from which a residual program is later created. The process tree is meant to mimic the execution of the input program. 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.

Hereafter, we consider all goals and relation bodies to be in *canonical normal form* — a disjunction of conjunctions of either calls or unifications. Moreover, we assume all fresh variables to be introduced into scope and all unifications to be computed at each step. Those disjuncts in which unifications fail are removed. Other disjuncts take form of a conjunction of relation calls accompanied with a substitution computed from unificaitons. Any miniKanren term can be trivially transformed to the described form. In Fig. 1 function `normalize` is assumed to perform term normalization. The code is omitted for brevity.

There are several core ideas behind this algorithm. The first is to select an arbitrary relation to unfold, not necessarily the leftmost which is safe. The second idea is to use a heuristic which decides if unfolding a relation call can lead to discovery of contradictions between conjuncts which in turn leads to restriction of the answer set at specialization-time (line 14; `heuristically_select_a_call` stands for heurictics combination, see section 3.2

```
1  ncpd goal = residualize ∘ drive ∘ normalize (goal)
2  drive      = drive_disj ∪ drive_conj
3
4  drive_disj :: Disjunction → Process_Tree
5  drive_disj D@(c_1, ..., c_n) = ⋁_{i=1}^{n} t_i ← drive_conj (c_i)
6
7  drive_conj :: (Conjunction, Substitution) → Process_Tree
8  drive_conj ((r_1, ..., r_n), subst) =
9    C@(r_1, ..., r_n) ← propagate_substitution subst on r_1, ..., r_n
10   case whistle (C) of
11    | instance (C', subst')         ⇒ create_fold_node (C', subst')
12    | embedded_but_not_instance ⇒ create_stop_node (C , subst )
13    | otherwise ⇒
14    | | case heuristically_select_a_call (r_1, ..., r_n) of
15    | | | Just r ⇒
16    | | | | t ← drive ∘ normalize ∘ unfold (r)
17    | | | | if trivial ∘ leafs (t)
18    | | | | then
19    | | | | | C' ← propagate_substitution (C \ r, extract_substitution (t))
20    | | | | | drive C'[r ↦ extract_calls (t)]
21    | | | | else
22    | | | | | t ⋀ drive (C \ r, subst)
23    | | | Nothing ⇒ ⋀_{i=1}^{n} t_i ← drive ∘ normalize ∘ unfold (r_i)
```

Fig. 1. Non-conjunctive Partial Deduction Pseudo Code

for details). If those contradictions are found, then they are exposed by considering the conjunction as a whole and substituting the result of the call unfolding back into the conjunction the call was selected from thus *joining* the conjunction back together instead of using *split* as in CPD (lines 15–22). Finally, if the heuristic fails to select a potentially good call, then the conjunction is splitted into individual calls which are driven in isolation and are never joined (line 23).

When the heuristic selects a call to unfold (line 15), a process tree is constructed for the selected call *in isolation* (line 16). The leaves of the computed tree are examined. If all leaves are either computed substitutions or are renamings of some relations accompanied with non-empty substitutions, then the leaves are collected and replace the considered call in the root conjunction (lines 19–20). 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 (line 22). According to the 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.

This process creates branchings whenever a disjunction is examined lines 4–5). 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 (line 11). If it is, then we create a special node which then is residualized into a call to the corresponding relation. This is known as *folding* in supercompilation.

We decided not to perform generalization in this approach in the same fashion as CPD or supercompilation does. Our conjunctions are always splitted into individual calls and are joined back together if it is sensible. If the need for generalization arises, i.e. embedding is detected, then we immediately stop driving this conjunction (line 12). When redisualizatin such conjunction, we just generate a conjunction of calls to the input program before specialization.

3.1 Unfolding

Unfolding in our case is done by substitution of some relation call by its body with simultaneous normalization and computation of unifications. The unfolding itself is straightforward however it is not always clear what to unfold and when to *stop* unfolding. Unfolding in functional programming languages specialization, as well as inlining in imperative one, is usually considered to be safe from a residual program efficiency point of view. It may only lead to code explosion or code duplication which is mostly left to a target program compiler optimization or even is out of consideration at all if a specializer is considered as a standalone tool [6].

Unfortunately, this is not the case for the specialization of relational programming language. Unlike functional and imperative, in logic and relational programming language unfolding may easily affect the target program efficiency [9]. Unfolding too much may create extra unifications, which is by itself a costly operation, or even introduce duplicated computations by propagating the unfolding results onto neighboring conjuncts.

There is a fine edge between too much unfolding and not enough unfolding. The former may be even worse than the latter. We believe that the following heuristic provides a reasonable approach to controling unfolding.

3.2 Heuristic

This heuristic is aimed at selecting the relation call within a conjunction which is both safe to unfold and may lead to discovering contradictions within the conjunction. The unsafe unfolding leads to uncontrollable increase in the number of relation calls in a conjunction. It is best to first unfold those relation calls which can be fully computed up to substitutions.

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 part of heuristic is less trivial, we call it the branching heuristic. It considers the case when the unfolded relation contains less disjuncts than it could possibly have. This means that we found some contradiction, some computations were gotten rid of, and thus the answer set was restricted, which is desirable when unfolding. To compute this heuristic we precompute the maximum possible number of disjuncts in each relation and compare this number with the number of disjuncts when unfolding a concrete relation call. Maximum number of disjuncts is computed by unfolding the body of the relation in which all relation calls were replaced by a unification which always succeeds.

```
1  heuristically_select_a_call :: Conjunction → Maybe Call
2  heuristically_select_a_call C = find heuristic C
3
4  heuristic :: Call → Bool
5  heuristic r = isStatic r || isDeterministic r || isLessBranching r
```

Fig. 2. Heuristic selection pseudocode

The pseudocode describing heuristic is shown in fig. 2. Selecting a good relation call can fail (line 1). The implementation works such that we first select those relation calls which are static, and only if there are none, we proceed to look at deterministic unfoldings and then we search for those which are less branching. This heuristic provides a good balance in unfolding.

4 EVALUATION

In our study we compared the CPD adaptation for miniKanren and the new non-conjunctive partial deduction. We have also employed the branching heuristic instead of the deterministic unfolding in the CPD to check wether it can improve the quality of the specialization.

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.

The last two relations are described in [11] thus we will not describe them here.

4.1 Evaluator of Logic Formulas

The relation `eval`^{o} describes evaluation of a subset of first-order logic formulas in a given substitution. It has 3 arguments. The first argument is a list of boolean values which serves as a substitution. The *i*-th value of the substitution 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 the value of the formula in the given substitution.

In this paper we specialize the relation to synthesize formulas which evaluate to ↑**true**. To do so we run the specializer for the goal with the last argument fixed to ↑**true**, while the first two arguments remain free variables. Depending on the way the `eval`^{o} is implemented, different specializers generate significantly different residual programs.

4.1.1 The Order of Relation Calls. One possible implementation of the evaluator in the syntax of OCanren is presented in listing 1. In this implementation the relation `elem`^{o} `subst v res` unifies `res` with the value of the variable `v` in the list `subst`. The relations `and`^{o}, `or`^{o}, and **not**^{o} encode corresponding boolean operations.

```
let rec evalᵒ subst fm res = conde [
  fresh (x y z v w) (
    (fm ≡ var v  ∧  elemᵒ subst v res);
    (fm ≡ conj x y  ∧  evalᵒ st x v  ∧ evalᵒ st y w  ∧ andᵒ v w res);
    (fm ≡ disj x y  ∧  evalᵒ st x v  ∧ evalᵒ st y w  ∧ orᵒ v w res);
    (fm ≡ neg x  ∧  evalᵒ st x v  ∧ notᵒ v res))]
```

Listing 1. Evaluator of formulas with boolean operation last

Note, that the calls to boolean relations `and`^{o}, `or`^{o}, and **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 `eval`^{o} `subst fm` ↑**true**), it fails to propagate the direction data onto recursive calls of `eval`^{o}. Knowing that `res` is ↑**true**, we can conclude that in the call `and`^{o} `v w res` variables `v` and `w` have to be ↑**true** as well. There are three possible options for these variables in the call `or`^{o} `v w res` and one for the call **not**^{o}. These variables are used in recursive calls of `eval`^{o} and thus restrict the result of driving them. CPD fails to recognize this, and thus unfold recursive calls of `eval`^{o} applied to fresh variables. It leads to over-unfolding, big residual programs and less than optimal performance.

The non-conjunctive partial deduction will first unfold those calls which are selected with the heuristic. Since exploring boolean operations makes more sense, they are unfolded before recursive calls of `eval`^{o}. The way non-conjunctive partial deduction treats this program is the same as it treats the other implementation in which boolean operations are moved to the left, as shown in listing 2. This program is easier for CPD to transform which demonstrates how unequal is the behavior of CPD for the similar programs.

4.1.2 Unfolding of Complex Relations. Depending on the way a relation is implemented, it may take different number of driving steps to reach the point when any useful information is derived through its unfolding. Partial

```
let rec evalᵒ subst fm res = conde [
  fresh (x y z v w) (
    (fm ≡ var v ∧ elemᵒ subst v res);
    (fm ≡ conj x y ∧ andᵒ_table v w res ∧ evalᵒ st x v ∧ evalᵒ st y w);
    (fm ≡ disj x y ∧ orᵒ_table v w res ∧ evalᵒ st x v ∧ evalᵒ st y w);
    (fm ≡ neg x ∧ notᵒ_table v res ∧ evalᵒ st x v))]
```

Listing 2. Evaluator of formulas with boolean operation second

deduction tries to unfold every relation call unless it is unsafe, but not all relation calls serve to restrict the search space and thus should be unfolded. In the implementation of $\texttt{eval}^o$ boolean operations can effectively restrict variables within the conjunctions and should be unfolded until they do. But depending on the way they are implemented, different number of driving steps should be performed to do so. The simplest way to implement these relations is with a table as demonstrated with the implementation of $\mathbf{not}^o$ in listing 3. It is enough to unfold such relation calls once to derive useful information about variables.

```
let notᵒ_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 basic boolean relation such as $\texttt{nand}^o$ which is in turn has a table-based implementation (see listing 4). It will take several sequential unfoldings to derive that variables v and w should be ↑**true** when considering a call $\texttt{and}^o$ v w ↑**true** implemented via a basic relation. Non-conjunctive partial deduction drives the selected call until it derives useful substitutions for the variables involved while CPD with deterministic unfolding may fail to do so.

```
let notᵒ x y = nandᵒ x x y

let orᵒ x y z = nandᵒ x x xx ∧ nandᵒ y y yy ∧ nandᵒ xx yy z

let andᵒ x y z = nandᵒ x y xy ∧ nandᵒ xy xy z

let nandᵒ 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

4.2 Evaluation Results

In our study we considered two implementations of $\texttt{eval}^o$: one we call `plain` and the other — `last` and compared how specializers behave on them. The `plain` relation uses table-based boolean operations and places them further

	last	plain	unify	isPath
Original	>60.00s	>60.00s	>300.00s	19.86s
CPD	31.31s	5.46s	2.35s	4.66s
Non-CPD	4.99s	5.05s	14.90s	3.00s
Branching	17.21s	6.17s	>300.00s	N/A

Table 1. Evaluation results

to the left in each conjunction. The relation `last` employs boolean operations implemented via $\texttt{nand}^o$ and place them at the end of each conjunction. These two programs are complete opposites from the standpoint of CPD.

We measured time necessary to generate 10000 formulas over two variables which evaluate to $\uparrow$**true**. We compared the results of specialization of the goal $\texttt{eval}^o$ `subst fm` $\uparrow$**true** by our implementation of CPD, the new non-conjunctive partial deduction, and the CPD modified with the branching heuristic. Our evaluation confirmed that CPD behaves very differently on these two implementations of the same relation: the running time of the specialized `last` relation is almost 6 times as high as the running time of the specialized `plain` relation. The running time of two programs generated with our novel non-conjunctive partial deduction is very close and it is a little bit better than the best by CPD. CPD with the branching heuristic still gives different quality transformations. The results are shown in table 1.

Besides evaluator of logic formulas we also run the transformers on the relation `unify` which searches for a unifier of two terms and a relation `isPath` specialized to search for paths in the graph. These two relations are described in paper [11] so we will not go into too much details here.

The `unify` relation was executed to find 5 unifiers of two terms $f(X, X, g(Z, t))$ and $f(g(p, L), Y, Y)$. The original miniKanren program fails to terminate on this goal in 300 seconds. On this example the most performant is the program generated by CPD (2.35 seconds) while the program generated by adding branching heuristic also fails to terminate in 300 seconds. The non-conjunctive partial deduction shows some improvement with the residual program terminated within 15 seconds. While driving this program, non-conjunctive partial deduction does too much unfolding which negatively impact the running time as compared to CPD.

The last test executed `isPath` relation to search for 5 paths in the graph with 20 vertices and 30 edges. On this program non-conjunctive partial deduction showed better transformation results than CPD, although the difference is not that drastic.

All evaluation results are presented in the table 1. Each column correspond to the relation being run as described above. The row marked "Original" contains the running time of the original miniKanren relation before specialization, "CPD" and "Non-CPD" correspond to conjunctive and non-conjunctive partial deduction respectively while "Branching" is for the CPD modified with the branching heuristic.

5 CONCLUSION

In this paper we discussed some issues which arise in partial deduction of a relational programming language miniKanren. We presented a novel approach to partial deduction which uses a heuristic to select the most suitable relation call to unfold at each step of driving. We compared this approach to the earlier implementation of conjunctive partial deduction. Our evaluation showed that there is still not one good technique which definitively speeds up every relational program.

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] Geoff W Hamilton. 2007. Distillation: extracting the essence of programs. In *Proceedings of the 2007 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*. 61–70.

[5] G. Higman. 1952. Ordering by divisibility in abstract algebras. In *Proceedings of the London Mathematical Society*, Vol. 2. 326–336.

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

[7] 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

[8] J. B. Kruskal. 1960. Well-quasi ordering, the tree theorem, and Vazsonyi's conjecture. *Trans. Amer. Math. Soc.* 95, 210–225.

[9] 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.

[10] John W. Lloyd and John C Shepherdson. 1991. Partial evaluation in logic programming. *The Journal of Logic Programming* 11, 3-4 (1991), 217–242.

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

[12] 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

[13] Morten Heine Soerensen, Robert Glück, and Neil D. Jones. 1996. A positive supercompiler. *Journal of functional programming* 6, 6 (1996), 811–838.

[14] Philip Wadler. 1990. Deforestation: Transforming Programs to Eliminate Trees. *Theor. Comput. Sci.* 73, 2 (1990), 231–248. https://doi.org/10.1016/0304-3975(90)90147-A