An Empirical Study of Partial Deduction for MINIKANREN

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

Daniil Berezun
Saint Petersburg State University
JetBrains Research
Saint Petersburg, Russia
d.berezun@2009.spbu.ru

Dmitry Boulytchev
Saint Petersburg State University
JetBrains Research
Saint Petersburg, Russia
dboulytchev@math.spbu.ru

We study conjunctive partial deduction, an advanced specialization technique aimed at improving the performance of logic programs, in the context of relational programming language MINIKANREN. We identify a number of issues, caused by MINIKANREN peculiarities, and describe a novel approach to specialization based on partial deduction and supercompilation. The results of the evaluation demonstrate successful specialization of relational interpreters. Although the project is at an early stage, we consider it as the first step towards an efficient optimization framework for MINIKANREN.

1 Introduction

A family of embedded domain-specific languages MINIKANREN[1] implement relational programming — a paradigm closely related to pure logic programming. The minimal core of the language, also known as MICROKANREN, can be implemented in as little as 39 lines of SCHEME [11]. An introduction to the language and some of its extensions in a series of examples can be found in the book [7]. The formal certified semantics for MINIKANREN is described in [29].

Relational programming is a paradigm based on the idea of describing programs as relations. The core feature of relational programming is the ability to run a program in various directions by executing goals with free variables. The distribution of free variable occurrences determines the direction of relational search. For example, having specified a relation for adding two numbers, one can also compute the subtraction of two numbers or find all pairs of numbers which can be summed up to get the given one. One of the most prominent applications of relational programming amounts to implementing interpreters as relations. By running a relational interpreter for some language *backwards* one can do program synthesis. In general, it is possible to create a solver from a recognizer by translating it into MINIKANREN and running it in the appropriate direction [24].

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 execution time of a program in MINIKANREN is highly unpredictable and varies greatly for various 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 dramatically in the other direction.

Partial evaluation [13] is a technique for specialization, i.e. improving the performance of a program given some information about it beforehand. It may either be a known value of some argument, its structure (e.g. the length of an input list) or, in case of a relational program, the direction in which the relation is intended to be run. An earlier paper [24] has shown that *conjunctive partial deduction* [6] can sometimes improve the performance of MINIKANREN programs. Depending on the particular *control* decisions, it may also not affect the execution time of a program or even make it slower.

[1] MINIKANREN language web site: http://minikanren.org. Access date: 28.02.2021

Submitted to:
VPT 2021

$$
\begin{array}{lcll}
\mathscr{T} & = & \mathscr{X} \cup \{C_i^{k_i}(t_1,\ldots,t_{k_i}) \mid t_j \in \mathscr{T}\} & \text{terms over the set of variables } \mathscr{X} \\
\mathscr{G} & = & \mathscr{T} \equiv \mathscr{T} & \text{unification} \\
 & & \mathscr{G} \wedge \mathscr{G} & \text{conjunction} \\
 & & \mathscr{G} \vee \mathscr{G} & \text{disjunction} \\
 & & \textbf{fresh } \mathscr{X} \,.\, \mathscr{G} & \text{fresh variable introduction} \\
 & & R_i^{k_i}(t_1,\ldots,t_{k_i}),\ t_j \in \mathscr{T} & \text{relational symbol invocation} \\
\mathscr{S} & = & \{R_i^{k_i} = \lambda\ x_1^i \ldots x_{k_i}^i \,.\, g_i;\}\ g & \text{specification}
\end{array}
$$

Figure 1: The syntax of the source language

Control issues in partial deduction of the logic programming language PROLOG have been studied before [19]. Under the left-to-right evaluation strategy of PROLOG, atoms in the right-hand side of a clause cannot be arbitrarily reordered without changing the observable behavior of a program. In contrast, due to the completeness of the search, MINIKANREN is less sensitive to the order of conjuncts: no answers can be added or lost, and the only difference caused by the order of conjuncts may be the divergence/convergence of the search in the case when all answers are found. This opens yet another possibility for optimization, not taken into account by the approaches initially developed in the context of conventional logic programming.

In this paper we make the following contributions. We study issues which conjunctive partial deduction faces being applied for MINIKANREN. We also describe a novel approach to partial deduction for relational programming, *conservative partial deduction*. We implemented this approach and compared it with the existing specialization system (ECCE) for several programs. We report here the results of the comparison and discuss why some MINIKANREN programs run slower after specialization.

2 Background

In this section we provide some background on relational programming and relational interpreters.

2.1 MINIKANREN

This paper considers the minimal relational core of the MINIKANREN language. The syntax of the language is presented in Fig. 1. A specification of the MINIKANREN program consists of a set of *relation definitions* accompanied by a top-level *goal* which plays the role of a query. Goals, being the central syntactic category of the language, can take the form of either term *unification*, *conjunction* or *disjunction* of goals, a *fresh* syntactic variable introduction, or a relation *call*. We consider the alphabet of constructors $\{C_i^{k_i}\}$ and relational symbols $\{R_i^{k_i}\}$ to be predefined and accompanied with their arities.

The formal semantics of the language is best described in [29]. Here we only briefly introduce the semantics. A stream of substitutions for free variables within the query goal is computed during the execution of a MINIKANREN program. Depending on the kind of the goal, one of the following situations is possible.

1. Term unification $t_1 \equiv t_2$ computes the most-general unification in the context of the current substitution. If it succeeds, the unifier is added into the current substitution and then it is returned as a singleton stream. Otherwise, an empty stream is returned.

```
let rec add^o x y z = conde [
    (x ≡ zero ∧ y ≡ z);
    (fresh (p) (x ≡ succ p ∧ add^o p (succ y) z) ) ]

let rec eval^o fm res = conde [fresh (x y xr yr) (
    (fm ≡ num res);
    (eval^o x xr ∧ eval^o y yr ∧
      conde [
        (fm ≡ sum x y ∧ add^o xr yr res);
        (fm ≡ prod x y ∧ ...);
        ... ] )
```

Listing 1: Evaluator of arithmetic expressions

2. Introduction of a fresh variable $fresh\ x.g$ allocates a new *semantic* variable, substitutes it for all fresh occurrences of x within g, then evaluates the goal.
3. An execution of a relational call $R_i^{k_i}(t_1,\ldots,t_{k_i})$ is done by first substituting the terms t_j for the respective formal parameters and then running the resulting goal.
4. When executing a conjunction $g_1 \wedge g_2$, first the goal g_1 is run in the context of the current substitution which results in the stream of substitutions, in each of which g_2 is run. The resulting stream of streams is then concatenated.
5. Disjunction $g_1 \vee g_2$ applies both goals to the current substitution, incrementally switching evaluation steps between the subgoals until all results (if any) are found.

Consider the relation `add`^{o} in Listing 1. It defines the relation between three Peano numbers x, y and z, such that $x + y = z$, using the OCANREN language[2]. The keyword **conde** provides syntactic sugar for a disjunction, while **zero** and **succ** are constructors. The query **fresh** (z) (add^o (**succ zero**) (**succ zero**) z) results in the only substitution [z $\mapsto$ **succ** (**succ zero**)], while the query **fresh** (x y) (add^o x y (**succ** (**succ zero**))) executes to three valid substitutions: [x $\mapsto$ **zero**, y $\mapsto$ **succ** (**succ zero**)], [x $\mapsto$ **succ zero**, y $\mapsto$ **succ zero**], [x $\mapsto$ **succ** (**succ zero**), y $\mapsto$ **zero**].

The *interleaving* search [15] is at the core of MINIKANREN. It evaluates disjuncts incrementally, passing control from one to the other. This search strategy is what makes the search in MINIKANREN complete. It also allows for reordering of both disjuncts and conjuncts within a goal which may improve the efficiency of a program. This reordering generally leads to the reordering of the answers computed by a MINIKANREN program. The denotational semantics of MINIKANREN ignores the order of the answers because the search is complete and thus all possible answers will be found eventually.

2.2 Relational Interpreters

The kind of relational programs most interesting to us is relational interpreters. They may be used to solve complex problems such as generating quines [4] or to solve search problems by only implementing

[2]OCANREN: statically typed MINIKANREN embedding in OCAML. The repository of the project: `https://github.com/JetBrains-Research/OCanren`. Access date: 28.02.2021

programs which check that a solution is correct [24]. The latter application is the focus of our research project thus we provide a brief description of it.

Search problems are notoriously complicated. In fact, they are much more complex than verification — checking that some candidate solution is indeed a solution. The ability of MINIKANREN programs to be evaluated in different directions along with the complete semantics of the language allows for automatic generation of a solver from a verifier using relational conversion [25]. Unfortunately, generated relational interpreters are often inefficient, since the conversion introduces a lot of extra unifications and boilerplate. This kind of inefficiency is a prime candidate for specialization.

Consider the relational interpreter `eval`^{o} `fm res` in Listing 1. It evaluates an arithmetic expression `fm` which can take the form of a number (**num** `res`) or a binary expression such as the **sum** `x y` or **prod** `x y`. Running the interpreter backwards synthesizes expressions which evaluate to the given number. For example one possible answer to the query `eval`^{o} `fm` (**succ** (**succ** **zero**)) is **sum** (**num** (**succ** **zero**)) (**sum** (**num** **zero**) (**num** (**succ** **zero**))).

3 Related Work

Specialization is an attractive technique aimed to improve the performance of a program making use of its static properties such as known arguments or its environment. Specialization is studied for functional, imperative, and logic programing and comes in different forms: partial evaluation [13] and partial deduction [23], supercompilation [32, 31], distillation [10], and many others.

The heart of supercompilation-based techniques is *driving* — a symbolic execution of a program through all possible execution paths. The result of driving is a possibly infinite *process tree* where nodes correspond to *configurations* which represent computation states. For example, in the case of pure functional programming languages, the computational state might be a term. Each path in the tree corresponds to some concrete program execution. The two main sources for supercompilation optimizations are aggressive information propagation about variables' values, equalities and disequalities, and precomputing of deterministic semantic evaluation steps. The latter process, also known as *deforestation* [37], means combining of consecutive process tree nodes with no branching. Of course, the process tree can contain infinite branches. *Whistles* — heuristics to identify possibly infinite branches — are used to ensure supercompilation termination. If a whistle signals during the construction of some branch, then something should be done to ensure termination. The most common approaches are either to stop driving the infinite branch completely (no specialization is done in this case and the source code is blindly copied into the residual program) or to fold the process tree to a *process graph*. When the process graph is constructed, the resulting, or *residual*, program can be extracted from the graph by the process called *residualization*. The main instrument to perform folding is some form of *generalization*. Generalization, abstracting away some computed data about the current term, makes folding possible. For example, 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 correctness and termination of a program transformer [30], most-specific generalization (anti-unification) and *homeomorphic embedding* [12, 16] as a whistle being common.

While supercompilation generally improves the behaviour 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. What is worse, the efficiency of a residual program from the target language evaluator point of view is rarely considered in the literature. The main optimization source is computing in advance all pos-

sible intermediate and statically-known semantics steps at program 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 [13]. Partial evaluation in logic programming should be done with care to not interfere with the compiler optimizations [33]. It is also known that supercompilation may adversely affect GHC optimizations making standalone compilation more powerful [2, 14] and cause code explosion [26]. Moreover, it may be hard to predict the real speedup of any given program using concrete benchmarks even disregarding the 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 some advice on how to implement a partial evaluator as well as a target program so that specialization indeed improves its performance [13, 3]. There is a lack of research in determining the classes of programs which transformers would definitely speed up.

Conjunctive partial deduction [6] makes an effort to provide reasonable control for the left-to-right evaluation strategy of PROLOG. CPD constructs a tree which models goal evaluation and is similar to an SLDNF tree; then a residual program is generated from the tree. Partial deduction itself resembles driving in supercompilation [9]. 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 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 nodes 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, i.e. a whistle based on homeomorphic embedding does not signal for the atom. When an atom is unfolded, a clause whose 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 is more than one suitable head, then several branches are added into the tree which corresponds to the disjunction in the residualized program. An adaptation of CPD for the MINIKANREN programming language is described in [24].

ECCE partial deduction system [18, 20] is the most mature implementation of CPD for PROLOG. ECCE provides various implementations of both local and global control as well as several degrees of post-processing. Unfortunately there is no automatic procedure to choose what control setting is likely to improve input programs the most. The choice of the proper control is left to the user.

An empirical study has shown that the most well-behaved strategy of local control in CPD for PROLOG is *deterministic unfolding* [17]. An atom is unfolded only if precisely one suitable clause head exists for it with the one exception: it is allowed to unfold an atom non-deterministically once for each local control tree. This means that if a non-deterministic atom is the leftmost one within a conjunction, it is most likely to be unfolded, introducing many new relation calls within the conjunction. We believe this is the core problem of 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 programs in PROLOG execute. Special care should be taken when unfolding non-leftmost atoms in PROLOG: one should ensure that it does not duplicate code, as well as that no side-effects are done out of order [1, 22]. However in MINIKANREN leftmost unfolding often leads to larger global control trees and, as a result, bigger, less efficient programs. On the contrary, according to the denotational semantics, the results of evaluation of a MINIKANREN program do not depend on the order of relation calls (atoms) within conjunctions, 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.

```
1  | conspd goal = residualize ∘ drive_disj ∘ normalize(goal, empty_substitution)
2  |
3  | drive_disj :: [(Disjunction, Substitution)] → Process_Graph
4  | drive_disj [(c_1, subst_1), ..., (c_n, subst_n)] = ⋁_{i=1}^{n} t_i ← drive_conj(c_i, subst_i)
5  |
6  | drive_conj :: (Conjunction, Substitution) → Process_Graph
7  | drive_conj ([], subst) = create_success_node (subst)
8  | drive_conj ([r_1, ..., r_n], subst) =
9  |   C@(r_1, ..., r_n) ← propagate_substitution subst onto 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 ← unfold_in_isolation(r, subst)
17 |     | | | | ls ← leaves(t)
18 |     | | | | if trivial(ls)
19 |     | | | | then
20 |     | | | | | ⋁_{i=1}^{k} t_i ← drive_conj(C[r ↦ get_call(i, ls)], get_subst(i, ls))
21 |     | | | | else
22 |     | | | | | t ⋀ drive_conj(C \ r, subst)
23 |     | | | Nothing ⇒ ⋀_{i=1}^{n} t_i ← drive_disj ∘ normalize ∘ unfold(r_i, subst)
24 |
25 | heuristically_select_a_call :: Conjunction → Maybe Call
26 | heuristically_select_a_call C =
27 |   find isStatic C <|> find isDeterministic C <|> find isLessBranching C
```

Figure 2: Conservative partial deduction pseudo code

4 Conservative Partial Deduction

In this section, we describe a novel approach to relational program specialization. This approach draws inspiration from both conjunctive partial deduction and supercompilation. The aim was to create a specialization algorithm which would be simpler than conjunctive partial deduction and use properties of MINIKANREN to improve the performance of the input programs.

The algorithm pseudocode is shown in Fig. 2. We use the following notation in the pseudocode. The circle, ∘, represents function composition. The *at* sign, @, is used to name a pattern matched value. The arrow, ←, is used to bind a variable on the left-hand side. The arrow can also be used to pattern match the value on the right-hand side (see line 9). We use symbols ⋀ and ⋁ to represent creating, respectively, a conjunction and a disjunction node in a process graph. The data type `Maybe a = Just a | Nothing` is taken from HASKELL and represents optional value. The binary function `x <|> y` combines two optional values: `Just x <|> y = Just x`, while `Nothing <|> y = y`.

Functions `drive_disj` and `drive_conj` describe how to process disjunctions and conjunctions respectively. 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 the scope and all unifications to be computed at each step. Thus driving is declared to be the function `drive_disj` (line 4).

A driving process (along with generalization and folding) creates a process graph, from which a

residual program is later created. The process graph is meant to mimic the execution of the input program. The nodes of the process graph include a *configuration* which describes the state of program evaluation at some point. In our case a configuration is a conjunction of relation calls. The substitution computed at each step is also stored in the graph node, although it is not included in the configuration. This means that only the goal, and not the substitution, is passed into the *whistle* to determine potential non-termination.

Residualization is done by the function `residualize`. Residualization traverses the process graph and generates the MINIKANREN goal as well as new relations whenever needed. A conjunction is created from a conjunction node, a disjunction — from a disjunction node, while substitutions are generated into a conjunction of unifications. Transient nodes — the nodes which have a single child node — correspond to intermediate functions in residual programs and are removed during residualization to improve the performance of programs. We also employ *redundant argument filtering* as described in [21].

Those disjuncts in which unifications fail are removed while driving. Each other disjunct takes the form of a possibly empty conjunction of relation calls accompanied with a substitution computed from unifications. Any MINIKANREN term can be trivially transformed into the described form. The function `normalize` in Fig. 2 is assumed to perform term normalization. The code is omitted for brevity but can be found in the implementation of the approach on github[3].

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 heuristics combination, see section 4.2 for details). If those contradictions are found, then they are exposed by considering the conjunction as a whole and replacing the selected relation call with the result of its unfolding thus *joining* the conjunction back together instead of using *split* as in CPD (lines 15–22). Joining instead of splitting is why we call our transformer *conservative* partial deduction. Finally, if the heuristic fails to select a potentially good call, then the conjunction is split 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). This is done by driving the call until all leaves of the process tree are either substitutions, failures or recursive calls to the relations unfolded within the tree (example process tree is provided in Figure 4). No folding is performed by `unfold_in_isolation` and each recursive call is unfolded at most once. The leaves of the computed tree are examined. If all leaves are either computed substitutions or are instances of some relations accompanied with non-empty substitutions, then the leaves are collected and each of them replaces the considered call in the root conjunction (line 20). If the selected call does not suit the criteria, the results of its unfolding are not propagated onto other relation calls within the conjunction, instead, 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 okay to drive any call and then propagate its results onto the other calls.

This process creates branchings whenever a disjunction is examined (lines 4, 20). When a goal is fully computed to a substitution (line 7), then a success node is created and driving stops. At each step, we make sure to not drive a conjunction which has already been examined. To do this, we check if the current conjunction is a renaming of any other configuration in the graph (line 11). If it is, then we fold by creating a special node which is then residualized into a call to the corresponding relation.

In this approach, we do not generalize in the same fashion as CPD or supercompilation. This decision was motivated by keeping the complexity of the approach to the minimum. Our conjunctions are always

[3]The project repository: `https://github.com/kajigor/uKanren_transformations/`. Access date: 28.02.2021

split into individual calls and are joined back together only if it is meaningful, for example, leads to contradictions. If the need for generalization arises, i.e. homeomorphic embedding of conjunctions [6] is detected, then we immediately stop driving this conjunction (line 12). When residualizing such a conjunction, we just generate a conjunction of calls to the input program before specialization.

4.1 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 the context of specialization of functional programming languages, as well as inlining in specialization of imperative languages, is usually considered to be safe from the 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 [13].

Unfortunately, this is not the case for the specialization of a relational programming language. Unlike functional and imperative languages, in logic and relational programming languages unfolding may easily affect the target program's efficiency [19, 8]. Unfolding too much may create extra unifications, which is by itself a costly operation, or even introduce duplicated computations by propagating the results of unfolding onto neighbouring conjuncts.

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

4.2 Less Branching Heuristic

This heuristic is aimed at selecting a relation call within a conjunction which is both safe to unfold and may lead to discovering contradictions within the conjunction. An unsafe unfolding leads to an uncontrollable increase of 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 (non-recursive) conjunct 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.

Those relation calls which are neither static nor deterministic are examined with what we call the *less-branching* heuristic. It identifies the case when the unfolded relation contains fewer 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. The 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.

The pseudocode describing our heuristic is shown in Fig. 2 (lines 25-27). Selecting a good relation call can fail. We express this by using `Maybe` data type for the result (line 25). The implementation works as follows: we first select those relation calls which are static, and only if there are none, we proceed to consider deterministic unfoldings and then we search for those which are less branching. We believe this heuristic provides a good balance in unfolding.

```
let rec eval^o s fm res = conde [fresh (x y z v w) (
    ( fm ≡ conj x y ∧ eval^o s x v ∧ eval^o s y w ∧ and^o v w res );
    ( fm ≡ disj x y ∧ eval^o s x v ∧ eval^o s y w ∧ or^o  v w res );
    ( fm ≡ neg x    ∧ eval^o s x v ∧ not^o v res );
    ( fm ≡ var v    ∧ elem^o s v res ))]

let not^o x y = nand^o x x y
let or^o  x y z = nand^o x x xx ∧ nand^o y y yy ∧ nand^o xx yy z
let and^o x y z = nand^o x y xy ∧ nand^o xy xy z
let nand^o 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 )]

let elem^o n s v = conde [ fresh (h t m) (
  ( n ≡ zero ∧ s ≡ h : t ∧ h ≡ v );
  ( n ≡ succ m ∧ s ≡ h : t ∧ elem^o m t v ))]
```

Listing 2: Evaluator of propositional formulas

5 Example

In this section we demonstrate by example how conservative partial deduction works. The example program is a relational interpreter of propositional formulas under given variable assignments. The complete code of the example program is provided in Listing 2.

The relation `eval`^{o} has three arguments. The first argument, `s`, is a list of boolean values which plays the role of variable assignments. The *i*-th value of the substitution is the value of the *i*-th variable. The second argument, `fm`, 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, `res`, is the value of the formula under the given assignment.

The relation `eval`^{o} is in canonical normal form: it is a single disjunction which consists of 4 conjunctions of unifications and relation calls, and all its fresh variables are introduced at the top level. The unification in each conjunction determines the shape of the input formula and binds variables to the corresponding subformulas. For each of the subformulas, the relation `eval`^{o} is called recursively. Then the results of the evaluation of the subformulas are combined by the corresponding boolean connective to get the result for the input formula. For example, when the formula is a conjunction of two subformulas `x` and `y`, the results of their execution `v` and `w` are combined by the call to the relation `and`^{o} to compute the result: `and`^{o} `v w res`. If the input formula is a variable, then its value is looked up in the substitution list by means of the relation `elem`^{o}.

Consider the goal **fresh** `(s fm)` (`eval`^{o} `s fm` **true**). The partially constructed process graph for this goal is presented in Fig. 3. The following notation is used. Rectangle nodes contain configurations, while diamond nodes correspond to splitting. Each configuration contains a goal along with a substitution in angle brackets. To visually differentiate constructors from variables within goals, we made them bold. For brevity, we only put a fragment of the substitution computed at each step in the corresponding node.

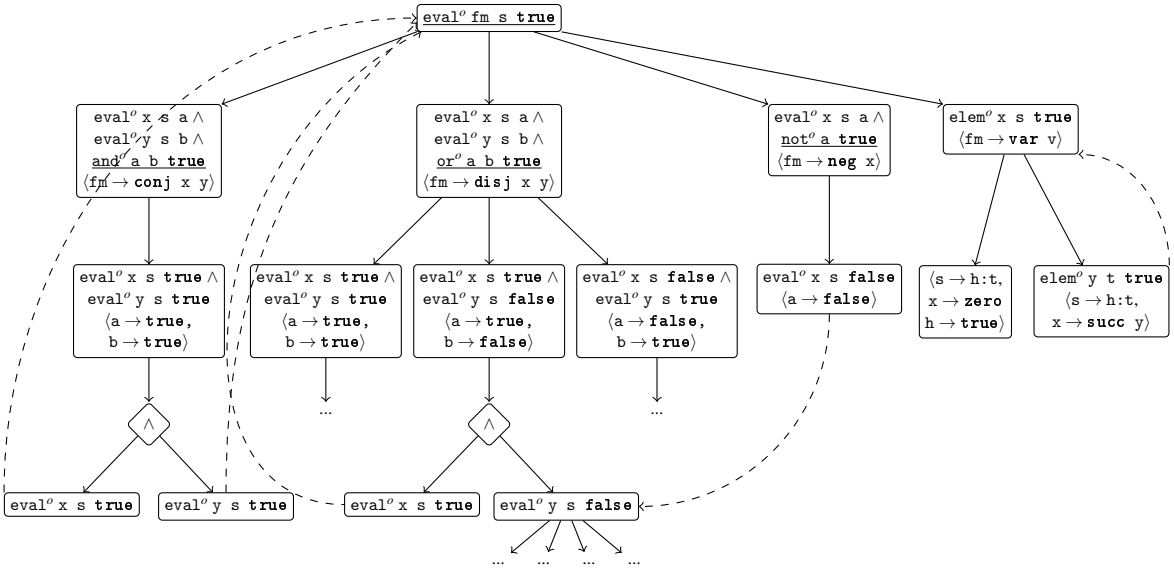

Figure 3: Partially constructed process graph for the relation eval^o.

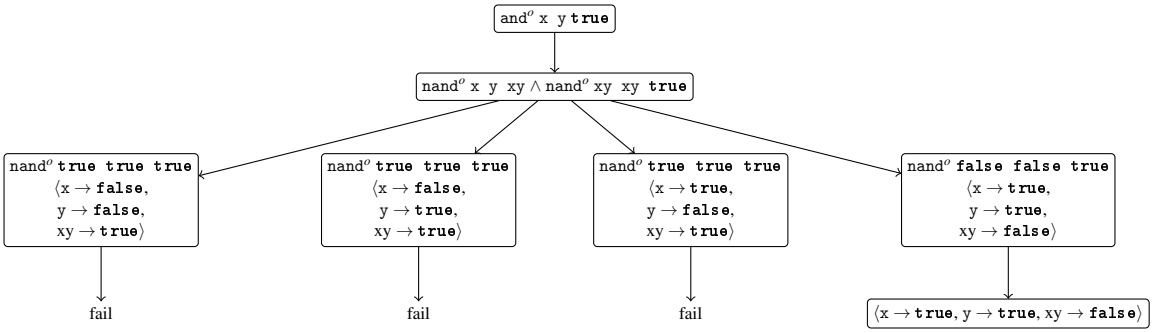

Figure 4: Unfolding of and^o.

The leaf node which contains only the substitution and no goal in its configuration is a success node. The call selected to be unfolded is underlined in a conjunction. Nodes corresponding to failures are not present within the process graph. Dashed arrows mark renamings.

Conservative partial deduction starts by unfolding the single relation call in this goal once. Besides introducing fresh variables into the context, unifications in each conjunct are computed into substitutions. This produces 4 branches, each of which is processed further.

Consider the first branch, in which the input formula `fm` is a conjunction of subformulas `x` and `y`. First, each relation call within a conjunction is examined separately to select one of the calls to unfold. To do so, we unfold each of them in isolation and use the less branching heuristic. Both recursive calls to `eval`^{o} are done with three distinct fresh variables, and are not selected according to the less branching heuristic. By unfolding the call to `and`^{o} several times, we determine it has a single non-failing branch (see Fig. 4), which is less than the same relation would have if called on all free and distinct variables, thus this call is selected to be unfolded. The result of unfolding the call is a single substitution which

```
let rec eval^o_true s fm = conde [fresh (x y z v w) (
    ( fm ≡ conj x y ∧ eval^o_true s x ∧eval^o_true s y );
    ( fm ≡ disj x y ∧ (conde [
        ( eval^o_true s x ∧ eval^o_true s y );
        ( eval^o_true s x ∧ eval^o_false s y );
        ( eval^o_false s x ∧ eval^o_true s y );
    ]);
    ( fm ≡ neg x ∧ eval^o_false s x v );
    ( fm ≡ var v ∧ elem^o_true s v ))]

let rec eval^o_false s fm = conde [fresh (x y z v w) (
    ( fm ≡ conj x y ∧ (conde [
        ( eval^o_false s x ∧ eval^o_false s y );
        ( eval^o_true s x ∧ eval^o_false s y );
        ( eval^o_false s x ∧ eval^o_true s y );
    ]);
    ( fm ≡ disj x y ∧ eval^o_true s x ∧eval^o_true s y );
    ( fm ≡ neg x ∧ eval^o_true s x v );
    ( fm ≡ var v ∧ elem^o_false s v ))]

let elem^o_true n s = conde [ fresh (h t m) (
    ( n ≡ zero ∧ s ≡ true : t );
    ( n ≡ succ m ∧ s ≡ h : t ∧ elem^o_true m t ))]

let elem^o_false n s = conde [ fresh (h t m) (
    ( n ≡ zero ∧ s ≡ false : t );
    ( n ≡ succ m ∧ s ≡ h : t ∧ elem^o_false m t ))]
```

Listing 3: Specialized evaluator of propositional formulas

associates the variables `a` and `b` with **true**. By applying the computed substitution to the goal we get a conjunction of two calls to the $\texttt{eval}^o$ relation with the last argument being **true**. This conjunction embeds the root goal, thus we split the conjunction and both calls become leaves which rename the root. This finishes the processing of the first branch.

Consider the second branch, in which the input formula `fm` is a disjunction of subformulas `x` and `y`. Similarly to the first branch, the heuristic selects the call to the boolean relation to be unfolded which produces three possible substitutions for variables `a` and `b`. The substitutions are propagated into the goals and three branches, each of which embeds the root goal, are added into the process graph. Each goal is then split, and the calls with the last argument being **true** rename the root. Finally, the call $\texttt{eval}^o$ `y s` **false** is processed, which creates 4 branches similar to the branches of the root goal.

The third branch is driven until a call with the last argument being **false** is encountered. Since it renames one of the nodes which is already present in the process graph, we stop exploring this branch and add the back edge.

The last branch, in which the input formula `fm` is a variable `v`, contains the single call to the relation

$\texttt{elem}^o$. The unfolding of this call produces two leaves: a success node and a renaming of the parent node. This finishes the construction of the process graph.

The process graph is then residualized into a specialized version of the $\texttt{eval}^o$ relation (see Listing 3). This program does not contain any calls to boolean connectives. Neither does the program contain the original, not specialized, relation $\texttt{eval}^o$.

It is worth noting that the result produced by the Conservative Partial Deduction is not ideal. For example, in the definition of the $\texttt{eval}^o_{true}$, when the input formula `fm` is a disjunction of subformulas `x` and `y`, the recursive call $\texttt{eval}^o_{true}$ `s x` is done twice in two disjuncts. The ideal version of the relation $\texttt{eval}^o_{true}$ should contain this recursive call only once. This can be done, for example, by common subexpression elimination [27]. However, ideally, `y` should not be evaluted at all, since the value of the formula `fm` does not depend on it. It is unclear if and how this kind of transformation can be done automatically. Such transformation would require, first, realising that a disjunction of two relation calls $\texttt{eval}^o_{true}$ `s y` and $\texttt{eval}^o_{false}$ `s y` exhaust all possible values for `y`. Secondly, the transformation would have to examine if a relation restricts values of a given argument regardless of the other arguments' values.

6 Evaluation

We implemented[4] the conservative partial deduction for MINIKANREN and compared it with the ECCE partial deduction system. ECCE is designed for the PROLOG programming language and cannot be directly applied for programs written in MINIKANREN. Nevertheless, the languages show resemblance, and it is valuable to check if the existing methods for PROLOG can be used directly in the context of relational programming. To be able to compare our approach with ECCE, we converted each input program first to the pure subset of PROLOG, then specialized it with ECCE, and then we converted the result back to MINIKANREN. The conversion to PROLOG is a simple syntactic conversion. In the conversion from PROLOG to MINIKANREN, for each Horn clause a conjunction is generated in which unifications are placed before any relation call. All programs are run as MINIKANREN programs in our experiments. In the final subsection we discuss some limitations of both our approach and ECCE.

We chose two problems for our study: evaluation of a subset of propositional formulas and typechecking for a simple language. The problems illustrate the approach of using relational interpreters to solve search problems [24]. For both these problems we considered several possible implementations in MINIKANREN which highlight different aspects relevant in specialization.

The $\texttt{eval}^o$ relation implements an evaluator of a subset of propositional formulas described in Section 5. We consider four different implementations of this relation to explore how the way the program is implemented can affect the quality of specialization. Depending on the implementation, ECCE generates programs of varying performance, while the execution times of the programs generated by our approach are similar.

The $\texttt{typecheck}^o$ relation implements a typechecker for a tiny expression language. We consider two different implementations of this relation: one written by hand and the other generated from a functional program, which implements the typechecker, as described in [24]. We demonstrate how much these implementations differ in terms of performance before and after specialization.

In this study we measured the execution time for the sample queries, averaging them over multiple runs. All examples of MINIKANREN relations in this paper are written in OCANREN. The queries were run on a laptop running Ubuntu 18.04 with quad core Intel Core i5 2.30GHz CPU and 8 GB of RAM.

[4] The project repository: `https://github.com/kajigor/uKanren_transformations/`. Access date: 28.02.2021

```
let rec eval^o s fm res = conde [fresh (x y z v w) (
    (fm ≡ conj x y ∧ eval^o s x v ∧ eval^o s y w ∧ and^o v w res);
    (fm ≡ disj x y ∧ eval^o s x v ∧ eval^o s y w ∧ or^o  v w res);
    (fm ≡ neg x    ∧ eval^o s x v ∧ not^o v res));
    (fm ≡ var v    ∧ elem^o s v res)]
```

Listing 4: Evaluator of formulas with boolean operation last

```
let rec eval^o s fm res = conde [fresh (x y z v w) (
    (fm ≡ conj x y ∧ and^o v w res ∧ eval^o s x v ∧ eval^o s y w);
    (fm ≡ disj x y ∧ or^o  v w res ∧ eval^o s x v ∧ eval^o s y w);
    (fm ≡ neg x    ∧ not^o v res   ∧ eval^o s x v);
    (fm ≡ var v    ∧ elem^o s v res))]
```

Listing 5: Evaluator of formulas with boolean operation second

The tables and graphs use the following denotations. *Original* represents the execution time of a program before any transformations were applied; *ECCE* — of the program specialized by ECCE with the default conjunctive control setting; *ConsPD* — of the program specialized by our approach.

6.1 Evaluator of Logic Formulas

The relation `eval`^{o} describes an evaluation of a propositional formula under given variable assignments presented in section 5. We specialize the `eval`^{o} 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 `eval`^{o} is implemented, different specializers generate significantly different residual programs.

6.1.1 The Order of Relation Calls

One possible implementation of the `eval`^{o} relation is presented in Listing 4. Here the relation `elem`^{o} `s v res` unifies `res` with the value of the variable `v` in the list `s`. The relations `and`^{o}, `or`^{o}, and `not`^{o} encode corresponding boolean connectives.

Note, the calls to boolean relations `and`^{o}, `or`^{o}, and `not`^{o} are placed last within each conjunction. This poses a challenge for the CPD-based specializers such as ECCE. 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} `s fm` **true**), it fails to propagate the direction data onto recursive calls of `eval`^{o}. Knowing that `res` is **true**, we can conclude that the variables `v` and `w` have to be **true** as well in the call `and`^{o} `v w res`. There are three possible options for these variables in the call `or`^{o} `v w res` and one for the call `not`^{o} `v res`. These variables are used in recursive calls of `eval`^{o} and thus restrict the result of its execution. CPD fails to recognize this, and thus unfolds recursive calls of `eval`^{o} applied to fresh variables. It leads to over-unfolding, large residual programs and poor performance.

The conservative partial deduction first unfolds those calls which are selected according to the heuristic. Since exploring the implementations of boolean connectives makes more sense, they are unfolded before the recursive calls of `eval`^{o}. The way conservative partial deduction treats this program is the same as it treats the other implementation in which boolean connectives are moved to the left, as shown

```
let not^o x y = conde [
  (x ≡ true ∧ y ≡ false;
   x ≡ false ∧ y ≡ true)]
```

Listing 6: Implementation of boolean not^o as a table

```
let not^o  x y = nand^o x x y
let or^o   x y z = nand^o x x xx ∧ nand^o y y yy ∧ nand^o xx yy z
let and^o x y z = nand^o x y xy ∧ nand^o xy xy z
let nand^o 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 7: Implementation of boolean operations via nand^o

in Listing 5. This program is easier for ECCE to specialize which demonstrates how unequal the behaviour of CPD for similar programs is.

6.1.2 Unfolding of Complex Relations

Depending on the way a relation is implemented, it may take a different number of driving steps to reach the point when any useful information is derived through its unfolding. Partial 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 `eval`^{o} boolean connectives can effectively restrict variables within the conjunctions and should be unfolded until they do. But depending on the way they are implemented, the different number of driving steps should be performed for that. The simplest way to implement these relations is by mimicking a truth table as demonstrated by the implementation of `not`^{o} in Listing 6. It is enough to unfold such relation calls once to derive useful information about variables.

The other way to implement boolean connectives is to express them using a single basic boolean relation such as `nand`^{o} which, in turn, has a table-based implementation (see Listing 7). It takes several sequential unfoldings to derive that the variables `v` and `w` should be **true** when considering a call `and`^{o} `v w` **true** implemented via a basic relation. Conservative 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.

6.1.3 Evaluation Results

In our study we considered 4 implementations of `eval`^{o} summarised in the Table 1. They differ in the way the boolean connectives are implemented (see column *Implementation*) and whether they are placed before or after the recursive calls to `eval`^{o} (see column *Placement*). These four implementations are very different from the standpoint of ECCE. We measured the time necessary to generate 1000 formulas over two variables which evaluate to **true** (averaged over 10 runs). The results are presented in Fig. 5.

Conservative partial deduction generates programs with comparable performance for all four implementations, while the quality of ECCE specialization differs significantly. ECCE worsens performance

	Implementation	Placement
FirstPlain	table-based	before
LastPlain	table-based	after
FirstNando	via nand^o	before
LastNando	via nand^o	after

Table 1: Different implementations of eval^o

	Original	ECCE	ConsPD
First-Plain	1.59s	1.61s	0.92s
First-Nando	1.43s	2.24s	0.96s
Last-Plain	0.98s	1.43s	0.97s
Last-Nando	1.09s	1.54s	0.91s

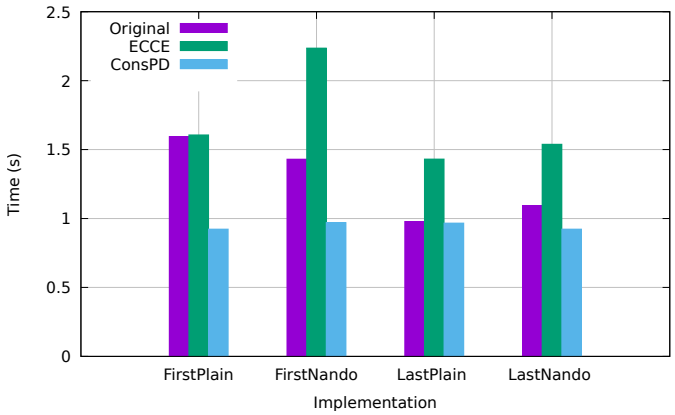

Figure 5: Execution time of eval^o

for every implementation as compared to the original program. ConsPD does not worsen performance for any implementation. Its effect is most significant for the implementations in which the boolean connectives are placed first within conjunctions.

6.1.4 The Order of Answers

It is important to note that different implementations of the same MINIKANREN relation produce answers in different orders. Nevertheless, since MINIKANREN search is complete, all answers will be found eventually. Unfortunately, it is not guaranteed that the first 1000 formulas generated with different implementations of `eval`^{o} will be the same. For example, 983 formulas are the same among the first 1000 formulas generated by the Original *FirstPlain* relation and the same relation after the ConsPD transformation. At the same time, only 405 formulas are the same between the Original and ECCE *LastNando* relations.

The reason why implementations differ so much in the order of the answers stems from the canonical search strategy employed in MINIKANREN. Most MINIKANREN implementations employ *interleaving* search [15] which is left-biased. It means that the leftmost disjunct in a relation is being executed longer than the disjunct on the right. This property is not local which makes it very hard to estimate the performance of a given relation.

In practice it means that if a specializer reorders disjuncts, then the performance of relations after specialization may be unpredictable. For example, by putting the disjuncts of the `eval`^{o} relation in the opposite order, one produces a relation which runs much faster than the original, but it generates completely different formulas at the same time. Most of the first 1000 formulas in this case are multiple negations of a variable, while the original relation produces more diverse set of answers. Computing

a negation of a formula only takes one recursive `eval`^{o} call thus finding such answers is faster than conjunctions and disjunctions. Meanwhile, the formulas generated by the reordered relation are less diverse and may be of less interest.

Although neither ECCE nor ConsPD reorder disjuncts, they remove disjuncts which cannot succeed. Thus they influence the order of answers and performance of relations. Both methods reduce the number of unifications needed to compute each individual answer thus performing specialization. In general, it is not possible to guarantee the same order of answers after specialization. Exploring how different specializations influence the execution order is a fascinating direction for future research.

6.2 Typechecker-Term Generator

This relation implements a typechecker for a tiny expression language. Being executed in the backward direction it serves as a generator of terms of the given type. The abstract syntax of the language is presented below. The variables are represented with de Bruijn indices, thus let-binding does not specify which variable is being bound.

$$\begin{array}{lllll} type\ term = & BConst\ of\ Bool & |\ IConst\ of\ Int & |\ Var\ of\ Int & \\ & |\ term + term & |\ term * term & |\ term = term & |\ term < term \\ & |\ \underline{let}\ term\ \underline{in}\ term & |\ \underline{if}\ term\ \underline{then}\ term\ \underline{else}\ term & & \end{array}$$

The typing rules are straightforward and are presented in Fig. 6. Boolean and integer constants have the corresponding types regardless of the environment. Only terms of type integer can be added, multiplied or compared by the less-than operator. Any terms of the same type can be checked for equality. Addition and multiplication of two terms of suitable types have integer type, while comparisons have boolean type. The if-then-else expression typechecks only if its condition is of type boolean, while both then- and else-branches have the same type. An environment Γ is a list, in which the i-th element is the type of the variable with the i-th de Bruijn index. To typecheck a let-binding, first, the term being bound is typechecked and is added in the beginning of the environment Γ, and then the body is typechecked in the context of the new environment. Typechecking a variable with the index i boils down to getting an i-th element of the list.

$$\frac{}{\Gamma \vdash IConst\ i : Int} \qquad \frac{}{\Gamma \vdash BConst\ b : Bool} \qquad \frac{}{\Gamma \vdash Var\ v : \tau}\ \Gamma[v] \equiv \tau$$

$$\frac{\Gamma \vdash t : Int, \Gamma \vdash s : Int}{\Gamma \vdash t + s : Int} \qquad \frac{\Gamma \vdash t : \tau, \Gamma \vdash s : \tau}{\Gamma \vdash t = s : Bool} \qquad \frac{\Gamma \vdash v : \tau_v,\ (\tau_v :: \Gamma) \vdash b : \tau}{\Gamma \vdash \underline{let}\ v\ b : \tau}$$

$$\frac{\Gamma \vdash t : Int, \Gamma \vdash s : Int}{\Gamma \vdash t * s : Int} \qquad \frac{\Gamma \vdash t : Int, \Gamma \vdash s : Int}{\Gamma \vdash t < s : Bool} \qquad \frac{\Gamma \vdash c : Bool, \Gamma \vdash t : \tau, \Gamma \vdash s : \tau}{\Gamma \vdash \underline{if}\ c\ \underline{then}\ t\ \underline{else}\ s : \tau}$$

Figure 6: Typing rules implemented in typecheck^o relation

We compared two implementations of these typing rules. The first one is obtained by unnesting of a functional program, which implements the typechecker, as described in [24] (*Generated*). It is worth noting that the unnesting introduces a lot of redundancy in the form of extra unifications and thus creates programs which are very inefficient. Thus we contrast this implementation with the program hand-written in OCANREN (*Hand-written*). Each implementation has been specialized with ConsPD and ECCE. We measured the time needed to generate 1000 closed terms of type integer (see Fig. 7).

	Original	ECCE	ConsPD
Hand-written	0.92s	0.22s	0.34s
Generated	11.46s	0.38s	0.29s

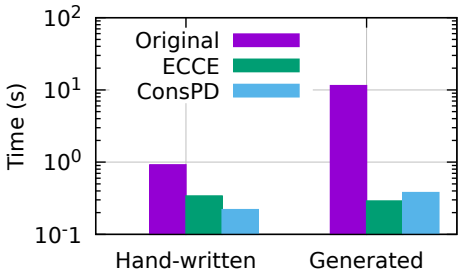

Figure 7: Execution time of generating 1000 closed terms of type integer

```
let rec typecheck^o gamma term res = conde [
  ...
  fresh (x y) ((term ≡ x + y ∧
    typecheck^o gamma x (some integer) ∧
    typecheck^o gamma y (some integer) ∧
    res ≡ (some integer)));
  ...]
```

Listing 8: A fragment of the hand-written typechecker

```
let rec typecheck^o gamma term res = conde [
  ...
  fresh (x y t1 t2) ((term ≡ x + y ∧
    conde [
      typecheck^o gamma x none       ∧ res ≡ none;
      typecheck^o gamma x (some t1) ∧
        typecheck^o gamma y none     ∧ res ≡ none;
      typecheck^o gamma x (some t1) ∧ typecheck^o gamma y (some t2) ∧
        typeEq^o t1 integer true     ∧ typeEq^o t2 integer true ∧
        res ≡ (some integer);
    ])
  ...]
```

Listing 9: A fragment of the generated typechecker

As expected, the generated program is far slower than the hand-written one. The principal difference between these two implementations is that the generated program contains a certain redundancy introduced by unnesting. For example, typechecking of the sum of two terms in the hand-written implementation consists of a single conjunction (see Listing 8) while the generated program is far more complicated and also uses a special relation $\texttt{typeEq}^o$ to compare types (see Listing 9). This relation has to be unfolded early to determine types of the subterms, much like the boolean relations implemented via a basic relation had to be unfolded in the previous problem.

Most redundancy of the generated program is removed by specialization with respect to the known type. This is why both implementations have comparable speed after specialization. ECCE shows bigger

```
let doubleAppend^o x y z res = fresh (t) (
    append^o x y t ∧ append^o t z res )

let append^o x y res = conde [ fresh (h t r) (
    ( x ≡ [] ∧ res ≡ y );
    ( x ≡ h : t ∧ append^o t y r ∧ res ≡ h : r ))]
```

Listing 10: Inefficient implementation of concatenation of three lists

```
let maxLength^o x m l = fresh (t) (
    max^o x m ∧ length^o x l )

let length^o x l = conde [ fresh (h t r) (
    ( x ≡ [] ∧ l ≡ zero );
    ( x ≡ h : t ∧ length^o t r ∧ l ≡ succ r ))]

let max^o x m = max_1^o x zero m
let rec max_1^o x n m = fresh (h t) ( conde [
    (x ≡ [] ∧ m ≡ n);
    (x ≡ h : t) ∧ (conde [
      (le^o h n true ∧ max_1^o t n m);
      (gt^o h n true ∧ max_1^o t h m)])])
```

Listing 11: Inefficient implementation of maxLength^o

speedup for the hand-written program than ConsPD and vice versa for the generated implementation. We believe that this difference can be explained by too much unfolding. ECCE performs a lot of excessive unfolding for the generated program and only barely changes the hand-written program. At the same time ConsPD specializes both implementations to comparable programs performing an average amount of unfolding. This shows that the heuristic we presented gives more stable, although not the best, results.

6.3 Discussion: Tupling and Deforestation

Tupling [28, 5] and deforestation are among the important transformations conjunctive partial deduction is capable of. Deforestation is often demonstrated by the `doubleAppend`^{o} program which concatenates three lists by calling the concatenation relation `append`^{o} twice in a conjunction (see Listing 10). The two calls to `append` lead to double traversal of the first list, which is inefficient. The program may be transformed in such a way so as to only traverse the first list once (see [6] for details), which conjunctive partial deduction does.

Conjunctive partial deduction achieves this effect by considering the conjunction of two `append`^{o} calls as a whole. At the local control level, it first unfolds the leftmost call, propagates the computed substitutions onto the rightmost call, and then unfolds the rightmost call in the context of the substitutions. When the first list is not empty, this leads to discovering a renaming of a conjunction of two `append`^{o} calls. By renaming this conjunction into a new predicate, deforestation is achieved in this example.

Unfortunately, conservative partial deduction does not succeed at this transformation on this example.

This happens because ConsPD splits the conjunction of two calls to `append`^{o}, since none of them is selected by the less branching heuristic. Splitting the conjunction leads to information loss and makes it so there is no renaming of the whole conjunction in the process graph.

A similar thing happens when considering the common example on which tupling is demonstrated in literature on CPD: the `maxLength`^{o} program. The original implementation of this program computes the maximum element of the list along with the length of the list by conjunction of two calls to the relations `max`^{o} and `length`^{o} respectively (see Listing. 11). This implementation also traverses the input list twice when run in the forward direction. By tupling, this program may be transformed so that the list is traversed once while both the maximum value and the length of the list are computed simultaneously, and CPD is capable to achieve this transformation with the default settings. Conservative partial deduction also splits much too early and thus fails to yield any useful transformation for this program.

It is worth noting that determinate unfolding performed by CPD plays a huge role in these examples. The default unfolding strategy implemented in ECCE allows for only a single non-determinate unfolding per a local control tree. When considering conjunctions `append`^{o} `x y t` $\wedge$ `append`^{o} `t z res` and `max`^{o} `x m` $\wedge$ `length`^{o} `x l`, it unfolds the leftmost call which produces several branches in the tree. The rightmost call is only considered, if unfolding of its conjunction with the result of unfolding of the leftmost call produces only one result. This indeed happens in these two examples. If it was not to happen, then the conjunction would have been split at the global level and no deforestation or tupling would have been achieved. It is not that hard to modify the examples in such a way so that CPD fails to transform them in a meaningful way. For example, one extra disjunct can be added into the `append`^{o} relation, or the calls to `max`^{o} and `length`^{o} may be reordered. This is evidence of how non-trivial and fragile these transformers are. More research should be done to make sure useful transformations are possible for many input programs.

7 Conclusion

In this paper we discussed some issues which arise in the area of partial deduction techniques for the relational programming language MINIKANREN. We presented a novel approach to partial deduction — conservative partial deduction — which uses a heuristic to select a suitable relation call to unfold at each step of driving. We compared this approach with the most sophisticated implementation of conjunctive partial deduction — ECCE partial deduction system — on 6 relations which solve 2 different problems.

Our specializer improved the execution time of all queries. ECCE worsened the performance of all 4 implementations of the propositional evaluator relation, while improving the other queries. Conservative partial deduction is more stable with regards to the order of relation calls than ECCE which is demonstrated by the similar performance of all 4 implementations of the evaluator of logic formulas.

Some queries to the same relation were improved more by ConsPD, while others — by ECCE. We conclude that there is still not one good technique which definitively speeds up every relational program. More research is needed to develop models capable of predicting the performance of a relation which can be used in specialization of MINIKANREN. There are some papers which estimate the efficiency of partial evaluation in the context of logic and functional logic programming languages [34, 35], and may facilitate achieving this goal. Employing a combination of offline and online transformations as done in [36] may also be the step towards more effective and predictable partial evaluation. Other directions for future research include exploring how specialization influences the execution order of a MINIKANREN program, improving ConsPD so that it succeeds at deforestation and tupling more often, and coming up with a larger, more impressive, set of benchmarks.

Acknowledgements

We gratefully acknowledge the anonymous referees and the participants of VPT-2021 workshop for fruitful discussions and many useful suggestions.

References

[1] Elvira Albert, Germán Puebla & John P. Gallagher (2006): *Non-leftmost Unfolding in Partial Evaluation of Logic Programs with Impure Predicates*. In Patricia M. Hill, editor: *Logic Based Program Synthesis and Transformation*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 115–132, doi:10.1007/11680093_8.

[2] Maximilian C. Bolingbroke & Simon L. Peyton Jones (2010): *Supercompilation by evaluation*. In Jeremy Gibbons, editor: *Proceedings of the 3rd ACM SIGPLAN Symposium on Haskell, Haskell 2010, Baltimore, MD, USA, 30 September 2010*, ACM, pp. 135–146, doi:10.1145/1863523.1863540.

[3] Mikhail A. Bulyonkov (1984): *Polyvariant Mixed Computation for Analyzer Programs*. *Acta Inf.* 21, pp. 473–484, doi:10.1007/BF00271642.

[4] William E. Byrd, Eric Holk & Daniel P. Friedman (2012): *miniKanren, live and untagged: Quine generation via relational interpreters (programming pearl)*. In: *Proceedings of the 2012 Annual Workshop on Scheme and Functional Programming*, pp. 8–29, doi:10.1145/2661103.2661105.

[5] Wei-Ngan Chin (1993): *Towards an automated tupling strategy*. In: *Proceedings of the 1993 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pp. 119–132, doi:10.1145/154630.154643.

[6] Danny De Schreye, Robert Glück, Jesper Jørgensen, Michael Leuschel, Bern Martens & Morten Heine B. Sørensen (1999): *Conjunctive partial deduction: Foundations, control, algorithms, and experiments*. *The Journal of Logic Programming* 41(2-3), pp. 231–277, doi:10.1016/S0743-1066(99)00030-8.

[7] Daniel P. Friedman, William E. Byrd & Oleg Kiselyov (2005): *The Reasoned Schemer*. The MIT Press, doi:10.7551/mitpress/5801.001.0001.

[8] John P. Gallagher (1993): *Tutorial on specialisation of logic programs*. In: *Proceedings of the 1993 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pp. 88–98, doi:10.1145/154630.154640.

[9] Robert Glück & Morten Heine B. Sørensen (1994): *Partial deduction and driving are equivalent*. In: *International Symposium on Programming Language Implementation and Logic Programming*, Springer, pp. 165–181, doi:10.1007/3-540-58402-1_13.

[10] 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*, pp. 61–70, doi:10.1145/1244381.1244391.

[11] Jason Hemann & Daniel P. Friedman: *μKanren: A Minimal Functional Core for Relational Programming*.

[12] Graham Higman (1952): *Ordering by divisibility in abstract algebras*. In: *Proceedings of the London Mathematical Society*, 2, pp. 326–336, doi:10.1112/plms/s3-2.1.326.

[13] Neil D. Jones, Carsten K. Gomard & Peter Sestoft (1993): *Partial evaluation and automatic program generation*. Prentice Hall international series in computer science, Prentice Hall.

[14] Peter A. Jonsson & Johan Nordlander (2011): *Taming code explosion in supercompilation*. In Siau-Cheng Khoo & Jeremy G. Siek, editors: *Proceedings of the 2011 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2011, Austin, TX, USA, January 24-25, 2011*, ACM, pp. 33–42, doi:10.1145/1929501.1929507.

[15] Oleg Kiselyov, Chung-chieh Shan, Daniel P. Friedman & Amr Sabry (2005): *Backtracking, Interleaving, and Terminating Monad Transformers: (Functional Pearl)*. *SIGPLAN Not.* 40(9), p. 192203, doi:10.1145/1090189.1086390.

[16] Joseph B. Kruskal (1960): *Well-quasi ordering, the tree theorem, and Vazsonyi's conjecture.* 95, pp. 210–225, doi:10.2307/1993287.

[17] Michael Leuschel (1997): *Advanced techniques for logic program specialisation*. Available at `https://lirias.kuleuven.be/retrieve/390873`.

[18] Michael Leuschel (1997): *The ecce partial deduction system*. In: *Proceedings of the ILPS'97 Workshop on Tools and Environments for (Constraint) Logic Programming, German Puebla, Universidad Politcnica de Madrid Tech. Rep. CLIP7/97.1*.

[19] Michael Leuschel & Maurice Bruynooghe (2002): *Logic program specialisation through partial deduction: Control issues*. *Theory and Practice of Logic Programming* 2(4-5), pp. 461–515, doi:10.1017/S147106840200145X.

[20] Michael Leuschel, Daniel Elphick, Mauricio Varea, Stephen-John Craig & Marc Fontaine (2006): *The Ecce and Logen partial evaluators and their web interfaces*. In John Hatcliff & Frank Tip, editors: *Proceedings of the 2006 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation, 2006, Charleston, South Carolina, USA, January 9-10, 2006*, ACM, pp. 88–94, doi:10.1145/1111542.1111557.

[21] Michael Leuschel & Morten Heine B. Sørensen (1996): *Redundant argument filtering of logic programs*. In: *International Workshop on Logic Programming Synthesis and Transformation*, Springer, pp. 83–103, doi:10.1007/3-540-62718-9_6.

[22] Michael Leuschel & Germán Vidal (2014): *Fast offline partial evaluation of logic programs*. *Information and Computation* 235, pp. 70–97, doi:10.1016/j.ic.2014.01.005.

[23] John W. Lloyd & John C. Shepherdson (1991): *Partial evaluation in logic programming*. *The Journal of Logic Programming* 11(3-4), pp. 217–242, doi:10.1016/0743-1066(91)90027-M.

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

[25] Petr Lozov, Andrei Vyatkin & Dmitry Boulytchev (2017): *Typed relational conversion*. In: *International Symposium on Trends in Functional Programming*, Springer, pp. 39–58, doi:10.1007/978-3-319-89719-6_3.

[26] Neil Mitchell & Colin Runciman (2007): *A Supercompiler for Core Haskell*. In Olaf Chitil, Zoltán Horváth & Viktória Zsók, editors: *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* 5083, Springer, pp. 147–164, doi:10.1007/978-3-540-85373-2_9.

[27] Steven Muchnick (1997): *Advanced compiler design implementation*. Morgan kaufmann.

[28] Alberto Pettorossi (1984): *A powerful strategy for deriving efficient programs by transformation*. In: *Proceedings of the 1984 ACM Symposium on LISP and functional programming*, pp. 273–281, doi:10.1145/800055.802044.

[29] Dmitry Rozplokhas, Andrey Vyatkin & Dmitry Boulytchev (2020): *Certified Semantics for Relational Programming*. In: *Asian Symposium on Programming Languages and Systems*, Springer, pp. 167–185, doi:10.1007/978-3-030-64437-6_9.

[30] Morten Heine B. Sørensen (1998): *Convergence of program transformers in the metric space of trees*. In: *International Conference on Mathematics of Program Construction*, Springer, pp. 315–337, doi:10.1007/BFb0054297.

[31] Morten Heine B. Sørensen, Robert Glück & Neil D. Jones (1996): *A positive supercompiler*. *Journal of functional programming* 6(6), pp. 811–838, doi:10.1017/S0956796800002008.

[32] Valentin F. Turchin (1986): *The Concept of a Supercompiler*. *ACM Trans. Program. Lang. Syst.* 8(3), p. 292325, doi:10.1145/5956.5957.

[33] Raf Venken & Bart Demoen (1988): *A partial evaluation system for prolog: some practical considerations*. *New Generation Computing* 6(2-3), pp. 279–290, doi:10.1007/BF03037142.

[34] Germán Vidal (2004): *Cost-augmented partial evaluation of functional logic programs*. *Higher-Order and Symbolic Computation* 17(1), pp. 7–46, doi:10.1023/B:LISP.0000029447.02190.42.

[35] Germán Vidal (2008): *Trace Analysis for Predicting the Effectiveness of Partial Evaluation*. In: *International Conference on Logic Programming*, Springer, pp. 790–794, doi:10.1007/978-3-540-89982-2_78.

[36] Germán Vidal (2011): *A Hybrid Approach to Conjunctive Partial Evaluation of Logic Programs*. In María Alpuente, editor: *Logic-Based Program Synthesis and Transformation*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 200–214, doi:10.1007/978-3-642-20551-4_13.

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