A Case Study in Functional Conversion and Mode Inference in miniKanren

Ekaterina Verbitskaia
kajigor@gmail.com
JetBrains Research
Belgrade, Serbia

Igor Engel
igorengel@mail.ru
JetBrains Research
Bremen, Germany

Daniil Berezun
daniil.berezun@jetbrains.com
JetBrains Research
Amsterdam, Netherlands

Abstract

Many programs which solve complicated problems can be seen as inversions of other, much simpler, programs. One particular example is transforming verifiers into solvers which can be achieved with low effort by implementing the verifier in a relational language and then executing it in the backward direction. Unfortunately, as it is common with inverse computations, interpretation overhead may lead to subpar performance compared to direct program inversion. In this paper we discuss functional conversion aimed at improving relational miniKanren specifications with respect to a known fixed direction. Our preliminary evaluation demonstrates a significant performance increase for some programs which exemplify the approach.

CCS Concepts: • **Software and its engineering → Constraint and logic languages**.

Keywords: program inversion, inverse computations, relational programming, functional programming, conversion

1 Introduction

There is a well-known observation [1, 13] that programs solving certain problems can be acquired by inverting programs solving some other, much simpler, problems. Sometimes the difference in the "simplicity" can be characterized in precise complexity-theoretic terms: for example, type checking for simple typed lambda calculus (STLC) is known to be linear-time (and rather straightforward to implement), while type inference (its inversion) is PTIME-complete [16], and type inhabitation problem (its another inversion) is PSPACE-complete [24].

In the scope of this paper we will be interested in a more concrete scenario of this generic idea, namely, in turning *verifiers* into *solvers*. A verifier is a procedure that, given an instance of the problem and some *sample*, verifies if this sample is a solution. A solver takes an instance of the problem and returns such a sample which makes the verifier to succeed. For the variety of search problems, the implementation of a verifier is straightforward; on the other hand its inversion is a solver, which as a rule is much harder to implement in an explicit manner. There are a few approaches to program inversion [2, 3], and the properties of the solver produced by inversion greatly depend on the approach utilized. We focus on the application of *relational programming* [7] as a way to run programs in the reverse direction.

Relational programming can be considered as a subfield of conventional logic programming focused on the study of implementation techniques and applications of *purely relational* specifications. In a narrow sense, relational programming amounts to writing programs in miniKanren[1] — an embedded DSL initially developed for Scheme and later ported for dozens of other host languages. Based on the same theory of first-order Horn clauses as, for example, Prolog, miniKanren employs a complete *interleaving search* [9, 19] and discourages the use of extra-logical features such as knowledge of concrete search order, "cuts", side-effects, efficient, but non-relational arithmetic, etc. In conventional logic programming the specification provided by the end-user usually encodes a certain concrete way to solve a problem. Contrary to this, miniKanren shifts the focus onto the specification of the problem itself with no certain hints on how to solve its various instances. This makes the specifications written in miniKanren short, elegant and expressive.

It is possible to directly employ the verifier-to-solver approach [5, 10] with miniKanren. It has been successfully applied in a few non-trivial projects [11, 12]. On the other hand, many useful optimization techniques cannot be applied for miniKanren programs directly since these programs lack an important part of information — the *direction* under which relational verifier turns into a solver. By taking this information into account, it is possible to make the approach more universally practical.

In this paper we present the results of our exploration in the area of *mode inference* and *functional conversion* for miniKanren. Mode analysis and inference is a relevant technique for conventional logic programming [6, 17, 21]. A mode can be considered as an implicit specification of a direction in which a relation is intended to be evaluated. Given a user-defined description of *modes* for (some) relations, mode analysis propagates the mode information through the rest of the logic program thus defining more concrete evaluation strategy for the rest of its relations.

Various notions and concrete approaches are employed for mode analysis in different settings, and we give a survey in Section 7. In our setting we consider user-defined mode specification for the top-level goal as the prescription

[1] Website of the miniKanren programming language: http://minikanren.org

$$
\begin{array}{rcl}
C\left(x_1, x_2\right) \equiv C\left(C\left(y_1, y_2\right), y_3\right) & \Rightarrow & x_1 \equiv C\left(y_1, y_2\right) \wedge x_2 \equiv y_3 \\
C\left(C\left(x_1, x_2\right), x_3\right) \equiv C\left(C\left(y_1, y_2\right), y_3\right) & \Rightarrow & x_1 \equiv y_1 \wedge x_2 \equiv y_2 \wedge x_3 \equiv y_3 \\
x \equiv C\left(y, y\right) & \Rightarrow & x \equiv C\left(y_1, y_2\right) \wedge y_1 \equiv y_2 \\
add^o\left(x, x, z\right) & \Rightarrow & add^o\left(x_1, x_2, z\right) \wedge x_1 \equiv x_2
\end{array}
$$

Figure 1. Examples of normalized goals

of the direction in which relational specification has to be evaluated to provide a solver for the problem in question. However, such a prescription cannot be directly employed in miniKanren as it contradicts the very nature of relational programming. Instead, we accompany mode inference with *functional conversion* — a transformation which, given a relational specification, a top-level goal, user-defined modes for this goal and the results of mode inference provides a regular functional program which delivers exactly the same answers as the top-level goal being evaluated in the direction prescribed by the modes. In addition, functional conversion can sometimes eliminate the interpretation overhead introduced by miniKanren implementation as a shallow DSL: it is capable of replacing unification with pattern-matching, make use of deterministic order of evaluation if such an order is discovered by mode inference, etc.

The contribution of this paper is as follows:

- We reiterate on mode inference for miniKanren, specifying concrete requirements specific for both miniKanren and our ultimate goal of putting verifier-to-solver idea to work.
- We describe a concrete approach to mode inference which takes the aforementioned requirements into account. As mode inference in general is known to be undecidable, we develop a number of heuristics specific to our case.
- We implement both mode inference and functional conversion for a reference miniKanren implementation.
- We evaluate our implementation on several benchmarks to investigate the advantages, drawbacks, and potential areas for improvement in our approach.

The rest of the paper is structured as follows. Section 2 describes the main ideas behind relational programming as well as the object language used in this paper. Related work including inverse computations and mode analysis is discussed in Section 7. Section 3 describes the scheme of functional conversion. The conversion is illustrated by examples in Section 4. The evaluation of the approach is presented in Section 5, followed by the discussion in Section 6. We conclude and sketch the directions for future work in Section 8.

2 Relational Programming and miniKanren

Relational programming as a subfield of conventional logic programming which is focused on using purely relational

```
let rec add^o x y z =
  (x ≡ 0  ∧  y ≡ z)  ∨
  (fresh x₁, z₁ in
     x ≡ S x₁  ∧
     add^o x₁ y z₁  ∧
     z ≡ S z₁)
```

Figure 2. Addition relation in miniKanren

specifications only. Using extra-logical features such as "cuts" and side-effects common in Prolog as well as the knowledge of the particular direction the relation is supposed to be run is discouraged. Since the search in miniKanren is complete [9, 19], all answers to the query will eventually be found without the programmer taking into account a particular search strategy used in the language implementation. It also means that the way in which a program is structured has no effect on which answers are found, only on the order in which they are computed.

In this paper we use a core miniKanren language usually referred to as microKanren. In its syntax, a relation is a goal comprised of disjunctions ($\vee$) or conjunctions ($\wedge$) of other goals. A base goal can be either an explicit unification of two terms ($\equiv$) or a call of a relation. An example program in miniKanren is shown in Figure 2. It relates triples of Peano natural numbers x, y, z such that x + y = z. We use a syntax notation such that constructors are denoted by identifiers which start with the uppercase letters, while identifiers which start with the lowercase letters are used as variable names. The superscript "^{o}" denotes a relation name while the keyword **fresh** introduces fresh variables into the scope. To execute a relation, one should provide a query to run. For example, the query **run** q (add^o q q (S (S 0))) finds a number which, doubled, is 2 in Peano representation, namely S 0. Some queries can compute values of several variables, and there may be infinitely many of them. For example, the query **run** q (**fresh** y,z **in** q == (y, z) $\wedge$ add^o (S 0) y z) finds all y and z such that $1 + y = z$. These answers are (0, S 0), (S 0, S (S 0)) and so forth.

To simplify the functional conversion scheme, we consider miniKanren relations to be in the superhomogeneous normal form used in the Mercury programming language [22].

$$
\begin{array}{llll}
\mathcal{D}^N_V & : & R_n\,(x_1,\ldots,x_n) = \mathbf{Disj}_V, x_i \in V & \text{normalized relation definition} \\
\mathbf{Disj}_V & : & \bigvee\,(c_1,\ldots,c_n)\,, c_i \in \mathbf{Conj}_V & \text{normal form} \\
\mathbf{Conj}_V & : & \bigwedge\,(g_1,\ldots,g_n)\,, g_i \in \mathbf{Base}_V & \text{normal conjunction} \\
\mathbf{Base}_V & : & V \equiv \mathcal{T}_V & \text{flat unification} \\
 & \mid & R_n\,(x_1,\ldots,x_n)\,, x_i \in V, i \neq j \Rightarrow x_i \neq x_j & \text{flat call}
\end{array}
$$

Figure 3. Abstract syntax of MICROKANREN in the normal form

```
let double^o x^{g→g} r^{f→g} =
  addo^o x_1^{g→g} x_2^{g→g} r^{f→g} ∧
  x_1^{g→g} ≡ x_2^{g→g}

let rec add^o x^{g→g} y^{g→g} z^{f→g} =
  (x^{g→g} ≡ 0 ∧ y^{g→g} ≡ z^{f→g}) ∨
  (x^{g→g} ≡ S x_1^{f→g} ∧
   add^o x_1^{g→g} y^{g→g} z_1^{f→g} ∧
   z^{f→g} ≡ S z_1^{g→g})
```

(a) Forward direction

```
let double^o x^{f→g} r^{g→g} =
  addo^o x_1^{f→g} x_2^{f→g} r^{g→g} ∧
  x_1^{g→g} ≡ x_2^{g→g}

let rec add^o x^{f→g} y^{f→g} z^{g→g} =
  (x^{f→g} ≡ 0 ∧ y^{f→g} ≡ z^{g→g}) ∨
  (z^{f→g} ≡ S z_1^{g→g} ∧
   add^o x_1^{f→g} y^{f→g} z_1^{g→g} ∧
   x^{f→g} ≡ S x_1^{g→g})
```

(b) Backward direction

Figure 4. Normalized doubling and addition relations with mode annotations

Converting an arbitrary MINIKANREN relation into the normal form is a simple syntactic transformation, which we omit.

In the normal form, a term is either a variable or a constructor application which is flat and linear. Linearity means that arguments of a constructor are distinct variables. To be flat, a term should not contain any nested constructors. Each constructor has a fixed arity n. Below is the abstract syntax of the term language over the set of variables V:

$$\mathcal{T}_V = V \cup \{C_n\,(x_1,\ldots,x_n) \mid x_i \in V; i \neq j \Rightarrow x_i \neq x_j\}$$

Whenever a term which does not adhere to this form is encountered in a unification or as an argument of a call, it is transformed into a conjunction of several unifications, as illustrated by the examples in Figure 1.

Unification in the normal form is restricted to always unify a variable with a term. We also prohibit using disjunctions inside conjunctions. The normalization procedure declares a new relation whenever this is encountered. This is done to limit the number of possible permutations one has to consider when doing the mode inference.

The complete abstract syntax of the MINIKANREN language used in this paper is presented in Figure 3.

3 Functional Conversion for MINIKANREN

In this section, we describe the functional conversion algorithm. The reader is encouraged to first read the paper [27] on the topic, which introduces the conversion scheme on a series of examples.

Functional conversion is done for a relation with a concrete fixed direction. The goal is to create a function which computes the same answers as MINIKANREN would, not necessarily in the same order. Since the search in MINIKANREN is complete, both conjuncts and disjuncts can be reordered freely: interleaving makes sure that no answers would be lost this way. Moreover, the original order of the subgoals is often suboptimal for any direction but the one which the programmer had in mind when they encoded the relation. In verifiers-to-solvers approach, a relational verifier is usually created automatically from an interpreter written in a functional language by means of typed relational conversion [14]. When it is used to create a relation, the order of the subgoals only really suits the forward direction, in which the relation is often not intended to be run (in this case, it is better to run the original function).

The mode inference results in the relational program with all variables annotated by their modes, and all base subgoals ordered in a way that further conversion makes sense. Conversion then produces functions in the intermediate language. It may then be pretty printed into concrete functional programming languages, in our case HASKELL and OCAML.

3.1 Mode Inference

Given an annotation for a relation, mode inference determines modes of each variable of the relation. For some modes, conjunctions in the body of a relation may need reordering to ensure that consumers of computed values come after the producers of said values so that a variable is never used before it is bound to some value. In this project, we employed the least complicated mode system, in which variables may only have an *in* or *out* mode. A mode maps variables of a relation to a pair of the initial and final instantiations. The mode *in* stands for $g \longrightarrow g$, while *out* stands for $f \longrightarrow g$. The instantiation f represents an unbound, or *free*, variable, when no information about its possible values is available. When the variable is known to be *ground*, its instantiation is g.

In this paper, we call a pair of instantiations a mode of a variable. Figure 4 shows examples of the normalized miniKanren relations with modes inferred for the forward and backward directions. We use superscript annotation for variables to represent their modes visually. Notice the different order of conjuncts in the bodies of the $\mathtt{add}^o$ relation in different modes.

We employ a simple version of mode analysis to order subgoals properly in the given direction. The mode analysis makes sure that a variable is never used before it is associated with some value. It also ensures that once a variable becomes ground, it never becomes free, thus the value of a variable is never lost. The mode inference pseudocode is presented in Listing 1.

```
1  modeInfer (R_i (x_1, ..., x_{k_i}) ≡ body) =
2    (R_i (x_1, ..., x_{k_i}) ≡ (modeInferDisj body))
3
4  modeInferDisj (⋁ (c_1, ..., c_n)) =
5    ⋁(modeInferConj c_1, ..., modeInferConj c_n)
6
7  modeInferConj (⋀ (g_1, ..., g_n)) =
8    let (picked, others) = pickConjunct [g_1, ..., g_n]
9    in let moddedPicked = modeInferBase picked
10   in let moddedConjs = modeInferConj (⋀others)
11   in ⋀(moddedPicked : moddedConjs)
12
13 pickConjunct goals =
14   pickGuard goals <|>
15   pickAssignment goals <|>
16   pickMatch goals <|>
17   pickCallWithGroundArguments goals <|>
18   pickUnificationGenerator goals <|>
19   pickCallGenerator goals
```

Listing 1. Mode inference pseudocode

Mode inference starts by initializing modes for all variables in the body of the given relation according to the given direction. All variables that are among arguments are annotated with their *in* or *out* modes, while all other variables get only their initial instantiations specified as f.

Then the body of the relation is analyzed (see line 2). Since the body is normalized, it can only be a disjunction. Each disjunct is analyzed independently (see line 5) because no data flow happens between them.

Analyzing conjunctions involves analyzing subgoals and ordering them. Let us first consider mode analysis of unifications and calls, and then circle back to the way we order them. Whenever a base goal is analyzed, all variables in it have some initial instantiation, and some of them also have some final instantiation. Mode analysis of a base goal boils down to making all final instantiations ground.

When analyzing a unification, several situations may occur. Firstly, every variable in the unification can be ground, as in $x^{g\to g} \equiv O$ or in $y^{g\to ?} \equiv z^{g\to ?}$ (here ? is used to denote that a final instantiation is not yet known). We call this case *guard*, since it is equivalent to checking that two values are the same.

The second case is when one side of a unification only contains ground variables. Depending on which side is ground, we call this either *assignment* or *match*. The former corresponds to assigning the value to a variable, as in $x^{f\to ?} \equiv S\,x_1^{g\to g}$ or $x^{g\to g} \equiv y^{f\to ?}$. The latter — to pattern matching with the variable as the scrutinee, as in $x^{g\to g} \equiv S\,x_1^{f\to ?}$. Notice that we allow for some variables on the right-hand side to be ground in matches, given that at least one of them is free.

The last case occurs when both the left-hand and right-hand sides contain free variables. This does not translate well into functional code. Any free logic variable corresponds to the possibly infinite number of ground values. To handle this kind of unification, we propose to use *generators* which produce all possible ground values a free variable may have.

We base our ordering strategy for conjuncts on the fact that these four different unification types have different costs. The guards are just equality checks which are inexpensive and can reduce the search space considerably. Assignments and matches are more involved, but they still take much less effort than generators. Moreover, executing non-generator conjuncts first can make some of the variables of the prospective generator ground thus avoiding generation in the end. This is the base reasoning which is behind our ordering strategy.

The function `pickConjunct` selects the base goal which is least likely to blow up the search space. The right-associative function `<|>` used in lines 14 through 18 is responsible for selecting the base goals in the order described. The function first attempts to pick a base goal with its first argument, and only if it fails, the second argument is called. As a result, `pickConjunct` first picks the first guard unification it can find (`pickGuard`). If no guard is present, then it searches for the first assignment (`pickAssignment`), and then for the match (`pickMatch`). If all unifications in the conjunction are generators, then we search for relation calls with some ground arguments (`pickCallWithGroundArguments`). If there are none, then we have no choice but selecting a generating unification (`pickUnificationGenerator`) and then a call with all arguments free (`pickCallGenerator`).

$$
\begin{array}{rcll}
\mathcal{F}_V & = & \textbf{Sum}\ [\mathcal{F}_V] & \text{concatenation of streams} \\
& | & \textbf{Bind}\ [([V], \mathcal{F}_V)] & \text{monadic bind for streams} \\
& | & \textbf{Return}\ [\mathcal{T}_V] & \text{return of a tuple of terms} \\
& | & \textbf{Guard}\ (V, V) & \text{equality check} \\
& | & \textbf{Match}_V\ (\mathcal{T}_V, \mathcal{F}_V) & \text{match a variable against a pattern} \\
& | & R_n([V], [G]) & \text{function call} \\
& | & \textbf{Gen}_G & \text{generator}
\end{array}
$$

Figure 5. Abstract syntax of the intermediate language $\mathcal{F}$

Once one conjunct is picked, it is analyzed (see line 9). The picked conjunct may instantiate new variables, thus this information is propagated onto the rest of the conjuncts. Then the rest of the conjuncts is mode analyzed as a new conjunction (see line 10). If any new modes for any of the relations are encountered, they are also mode analyzed.

It is worth noticing that any relation can generate infinitely many answers. We cannot judge the relation to be such generator solely by its mode: for example, the addition relation in the mode $\mathsf{add}^o\ \mathsf{x}^{g\to g}\ \mathsf{y}^{f\to g}\ \mathsf{z}^{f\to g}$ generates an infinite stream, while $\mathsf{add}^o\ \mathsf{x}^{f\to g}\ \mathsf{y}^{f\to g}\ \mathsf{z}^{g\to g}$ does not.

3.2 Conversion into Intermediate Representation

To represent nondeterminism, our functional conversion uses the basis of miniKanren — the stream data structure. A relation is converted into a function with n arguments which returns a stream of m-tuples, where n is the number of the input arguments, and m — the number of the output arguments of the relation. Since stream is a monad, functions can be written elegantly in Haskell using do-notation (see Figure 6). We use an intermediate representation which draws inspiration from Haskell's do-notation, but can then be pretty-printed into other functional languages. The abstract syntax of our intermediate language is shown in Figure 5. The conversion follows quite naturally from the modded relation and the syntax of the intermediate representation.

A body of a function is formed as an interleaving concatenation of streams (**Sum**), each of which is constructed from one of the disjuncts of the relation. A conjunction is translated into a sequence of bind statements (**Bind**): one for each of the conjuncts and a return statement (**Return**) in the end. A bind statement binds a tuple of variables (or nothing) with values taken from the stream in the right-hand side.

A base goal is converted into a guard (**Guard**), match (**Match**), or function call, depending on the goal's type. Assignments are translated into binds with a single return statement on the right. Notice, that a match only has one branch. This branch corresponds to a unification. If the scrutinee does not match the term it is unified with, then an empty stream is returned in the catch-all branch. If a term in the right-hand side of a unification has both *out* and *in* variables, then additional guards are placed in the body of the branch to ensure the equality between values bound in the pattern and the actual ground values.

Generators (**Gen**) are used for unifications with free variables on both sides. A generator is a stream of possible values for the free variables, and it is used for each variable from the right-hand side of the unification. The variable from the left-hand side of the unification is then simply assigned the value constructed from the right-hand side. Our current implementation works with an untyped deeply embedded miniKanren, in which there is not enough information to produce generators automatically. We decided to delegate the responsibility to provide generators to the user: a generator for each free variable is added as an argument of the relation. When the user is to call the function, they have to provide the suitable generators.

4 Examples

In this section, we provide some examples which demonstrate mode analysis and conversion results.

4.1 Multiplication Relation

Figure 6 shows the implementation of the multiplication relation $\mathtt{mul}^o$, the mode analysis result for mode $\mathtt{mul}^o\ \mathsf{x}^{f\to g}\ \mathsf{y}^{g\to g}\ \mathsf{z}^{g\to g}$, and the results of functional conversion into Haskell and OCaml.

Note that the unification comes last in the second disjunct. This is because before the two relation calls are done, both variables in the unification are free. Our version of mode inference puts the relation calls before the unification, but the order of the calls depends on their order in the original relation. There is nothing else our mode inference uses to prefer the order presented in the figure over the opposite: $\mathtt{mul}^o\ \mathsf{x}_1^{f\to g}\ \mathsf{y}^{g\to g}\ \mathsf{z}_1^{f\to g} \wedge \mathsf{add}^o\ \mathsf{y}^{g\to g}\ \mathsf{z}_1^{g\to g}\ \mathsf{z}^{g\to g}$. However, it is possible to derive this optimal order, if determinism analysis is employed: $\mathsf{add}^o\ \mathsf{y}^{g\to g}\ \mathsf{z}_1^{f\to g}\ \mathsf{z}^{g\to g}$ is deterministic while $\mathtt{mul}^o\ \mathsf{x}_1^{f\to g}\ \mathsf{y}^{g\to g}\ \mathsf{z}_1^{f\to g}$ is not. Putting nondeterministic computations first makes the search space larger, and thus should be avoided if another order is possible.

Functional conversions in both languages are similar, modulo the syntax. The Haskell version employs do-notation, while we use let-syntax in the OCaml code. Both are syntactic sugar for monadic computations over streams. We use the following convention to name the functions: we add a

```
let rec mulᵒ x y z =
  (x ≡ 0  ∧  z ≡ 0)  ∨
  (fresh x1, z1 in
    (x ≡ S x1  ∧
    addᵒ  y z1 z  ∧
    mulᵒ  x1 y z1))
```

(a) Implementation in miniKanren

```
let rec mulᵒ x^{f→g} y^{g→g} z^{g→g} =
  (x^{f→g} ≡ 0  ∧  z^{g→g} ≡ 0)  ∨
  (addᵒ  y^{g→g} z1^{f→g} z^{g→g}  ∧
  mulᵒ  x1^{f→g} y^{g→g} z1^{g→g}  ∧
  x^{f→g} ≡ S x1^{g→g})
```

(b) Mode inference result

```
muloOII x1 x2 = msum
  [ do { let {x0 = 0}
       ; guard (x2 == 0)
       ; return x0 }
  , do { x4 ← addoIOI x1 x2
       ; x3 ← muloOII x1 x4
       ; let {x0 = S x3}
       ; return x0 } ]
addoIOI x0 x2 = msum
  [ do { guard (x0 == 0)
       ; let {x1 = x2}
       ; return x1 }
  , do { x3 ← case x0 of
             { S y3 → return y3
             ; _ → mzero }
       ; x4 ← case x2 of
             { S y4 → return y4
             ; _ → mzero }
       ; x1 ← addoIOI x3 x4
       ; return x1 } ]
```

(c) Functional conversion into Haskell

```
let rec muloOII x1 x2 = msum
  [ ( let* x0 = return 0 in
      let* _ = guard (x2 = 0) in
      return x0 )
  ; ( let* x4 = addoIOI x1 x2 in
      let* x3 = muloOII x1 x4 in
      let* x0 = return (S x3) in
      return x0 ) ]
and addoIOI x0 x2 = msum
  [ ( let* _ = guard (x0 = 0) in
      let* x1 = return x2 in
      return x1 )
  ; ( let* x3 = match x0 with
        | S y3 → return y3
        | _ → mzero in
      let* x4 = match x2 with
        | S y4 →  return y4
        | _ → mzero in
      let* x1 = addoIOI x3 x4 in
      return x1 ) ]
```

(d) Functional conversion into OCaml

Figure 6. Multiplication relation

suffix to the relation's name whose length is the same as the number of the relation's arguments. The suffix consists of the letters `I` and `O` which denote whether the argument in the corresponding position is *in* or *out*. The function `msum` uses the interleaving function `mplus` to concatenate the list of streams constructed from disjuncts. To check conditions, we use the function `guard` which fails the monadic computation if the condition does not hold. Note that even though patterns for the variable `x0` in the function `addoIOI` are disjunct in two branches, we do not express them as a single pattern match. Doing so would improve readability, but it does not make a difference when it comes to the performance, according to our evaluation.

4.2 The Mode of Addition Relation which Needs a Generator

Consider the example of the addition relation in the mode $\mathsf{add}^o\ \mathsf{x}^{g\to g}\ \mathsf{y}^{f\to g}\ \mathsf{z}^{f\to g}$ presented in Figure 7. The unification in the first disjunct of this relation involves two free variables. We use a generator `gen_addoIIO_x2` to generate a stream of ground values for the variable `z` which is passed into the function `addIIO` as an argument. It is up to the user to provide a suitable generator. One of the possible generators which produces all Peano numbers in order and an example of its usage are presented in Figure 7b.

The generators which produce an infinite stream should be inverse eta-delayed in OCaml and other non-lazy languages. Otherwise, the function would not terminate trying to eagerly produce all possible ground values before using any of them.

It is possible to automatically produce generators from the data type of a variable, but it is currently not implemented, as we work with an untyped version of microKanren.

5 Evaluation

To evaluate our functional conversion scheme, we implemented the proposed algorithm in Haskell. We compared

```
let rec add^o x^{g→g} y^{f→g} z^{f→g} =
  (x^{g→g} ≡ O ∧ y^{f→g} ≡ z^{f→g}) ∨
  (x^{g→g} ≡ S x_1^{f→g} ∧
   add^o x_1^{g→g} y^{f→g} z_1^{f→g} ∧
   z^{f→g} ≡ S z_1^{g→g})
```

(a) Mode inference result

```
genNat = msum
  [ return O
  , do { x ← genNat
       ; return (S x) } ]
runAddoIIO x = addoIIO x genNat
```

(b) Generator of Peano numbers

```
addoIOO x0 gen_addoIOO_x2 = msum
  [ do { guard (x0 == O)
       ; (x1, x2) ← do { x2 ← gen_addoIOO_x2 ; return (x2, x2) }
       ; return (x1, x2) }
  , do { x3 ← case x0 of { S y3 → return y3 ; _ → mzero }
       ; (x1, x4) ← addoIOO x3 gen_addoIOO_x2
       ; let {x2 = S x4} ; return (x1, x2) } ]
```

(c) Functional conversion

Figure 7. Addition relation when only the first argument is *in*

execution time of several OCanren relations in different directions against their functional counterparts in the OCaml language. Here we showcase three relational programs and their conversions. The implementation of the functional conversion[2] as well as the execution code[3] can be found on Github.

5.1 Evaluator of Propositional Formulas

In this example, we converted a relational evaluator of propositional formulas: see Figure 8. It evaluates a propositional formula `fm` in the environment `st` to get the result `u`. A formula is either a boolean literal, a numbered variable, a negation of another formula, a conjunction or a disjunction of two formulas. Converting it in the direction when everything but the formula is *in* (see Figure 8a), allows one to synthesize formulas which can be evaluated to the given value. The conversion of this relation does not involve any generators and is presented in Figure 8b.

We ran an experiment to compare the execution time of the relational interpreter vs. its functional conversion. In the experiment, we generated from 1000 to 10000 formulas which evaluate to true and contain up to 3 variables with known values. The results are presented in Figure 9. The functional conversion improved execution time of the query about 2.5 times from 724*ms* to 291*ms* for retrieving 10000 formulas.

[2]The repository of the functional conversion project https://github.com/kajigor/uKanren_transformations

[3]Evaluation code https://github.com/kajigor/miniKanren-func

5.2 Multiplication

In this example, we converted the multiplication relation in several directions and compared them to the relational counterparts: see Figure 10. Functional conversion significantly reduced execution time in most directions.

In the forward direction, we run the query $\texttt{mul}^o$ `n 10 q` with `n` in the range from 100 to 1000, and the functional conversion was 2 orders of magnitude faster: 927*ms* vs 9.4*ms* for the largest `n`, see Figure 10a. In the direction which serves as division, we run the query $\texttt{mul}^o$ (`n`/10) `q n` with `n` ranging from 100 to 1000. Here, performance improved 3 orders of magnitude: from 24*s* to 0.17*s* for the largest `n`, see Figure 10b. Even more impressive was the backward direction $\texttt{mul}^o\ \texttt{x}^{f\to g}\ \texttt{y}^{f\to g}\ \texttt{z}^{g\to g}$. Querying for all 16 pairs of divisors of 1000 ($\texttt{mul}^o$ `q r 1000`) took OCanren about 32.9*s*, while the functional conversion succeeded in 1.1*s*.

What was surprising was the mode $\texttt{mul}^o\ \texttt{x}^{g\to g}\ \texttt{y}^{f\to g}\ \texttt{z}^{f\to g}$. In this case, the functional conversion was not only worse than its relational counterpart, its performance degraded exponentially with the number of answers asked. It took almost 1450*ms* to find the first 7 pairs of numbers `q` and `r` such that `10 * q = r`, while OCanren was able to execute the query in 0.74*ms* (see Figure 10c). The source of this terrible behavior was the suboptimal order of the calls in the second disjunct of the $\texttt{mul}^o$ relation in the corresponding mode (see Figure 10d). In this case, the call $\texttt{add}^o\ \texttt{y}^{f\to g}\ \texttt{z}_1^{f\to g}\ \texttt{z}^{f\to g}$ is put first, which generates all possible triples in the addition relation before filtering them by the call $\texttt{mul}^o\ \texttt{x}_1^{g\to g}\ \texttt{y}^{g\to g}\ \texttt{z}_1^{g\to g}$. The other order of calls is much better (see Figure 10e): it is an order of magnitude faster than its relational source. To achieve the better of these two orders automatically, we delay picking any call with all arguments free. A call of this kind

```
let rec eval^o st^{g→g} fm^{f→g} u^{g→g} =
  ( fm^{f→g} ≡  Lit u^{g→g}) ∨
  ( elem^o z^{f→g} st^{g→g} u^{g→g} ∧
    fm^{f→g} ≡ Var z^{g→g} ) ∨
  ( not^o v^{f→g} u^{g→g} ∧
    eval^o st^{g→g} x^{f→g} v^{g→g} ∧
    fm^{f→g} ≡ Neg x^{g→g} ) ∨
  ( or^o v^{f→g} w^{f→g} u^{g→g} ∧
    eval^o st^{g→g} x^{f→g} v^{g→g} ∧
    eval^o st^{g→g} y^{f→g} w^{g→g} ∧
    fm^{f→g} ≡ Disj x^{g→g} y^{g→g}) ∨
  ( and^o v^{f→g} w^{f→g} u^{g→g} ∧
    eval^o st^{g→g} x^{f→g} v^{g→g} ∧
    eval^o st^{g→g} y^{f→g} w^{g→g} ∧
    fm^{f→g} ≡ Conj x^{g→g} y^{g→g}) ∨
```

(a) Mode inference result

```
evaloIOI x0 x2 = msum
  [ do { let {x1 = Lit x2}
       ; return x1 }
  , do { x7 ← elemoOII x0 x2
       ; let {x1 = Var x7}
       ; return x1 }
  , do { x5 ← notoOI x2
       ; x3 ← evaloIOI x0 x5
       ; let {x1 = Neg x3}
       ; return x1 }
  , do { (x5, x6) ← oroOOI x2
       ; x3 ← evaloIOI x0 x5
       ; x4 ← evaloIOI x0 x6
       ; let {x1 = Disj x3 x4}
       ; return x1 }
  , do { (x5, x6) ← andoOOI x2
       ; x3 ← evaloIOI x0 x5
       ; x4 ← evaloIOI x0 x6
       ; let {x1 = Conj x3 x4}
       ; return x1 } ]
```

(b) Functional conversion

Figure 8. Evaluator of propositional formulas

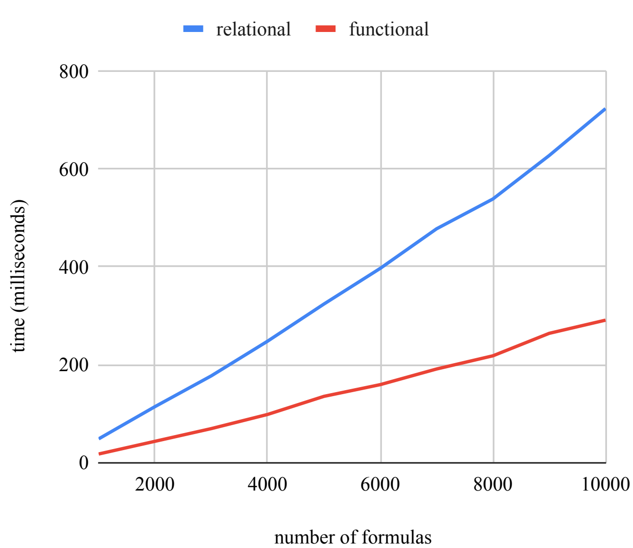


Figure 9. Execution time of the evaluators of propositional formulas, `eval [true; false; true] q true`

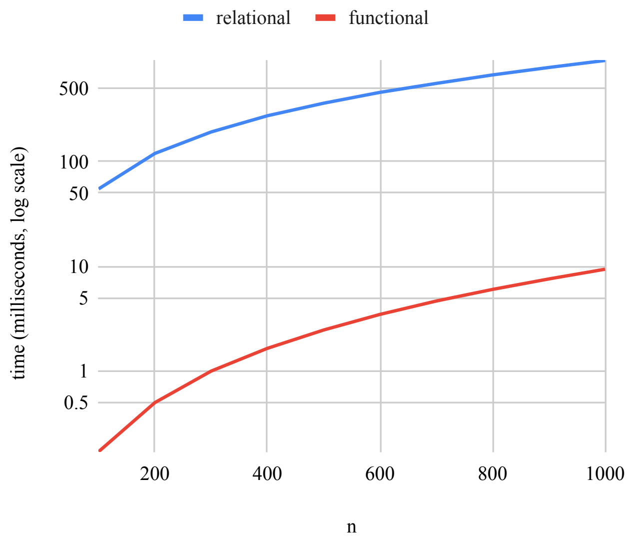
always works as a generator of every tuple of values which are in relation. It is a reasonable heuristics to postpone their execution until its arguments become more instantiated.

5.3 Relational Sorting

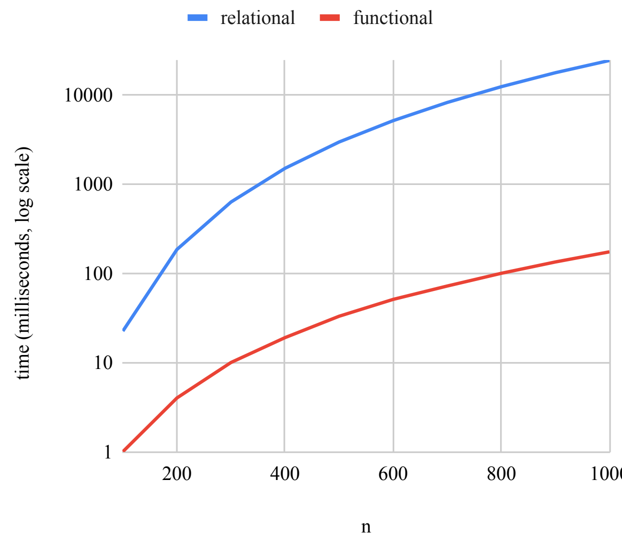
This program is written in truly relational style. By definition, a sorted list has its smallest element in its head followed by a sorted list. The implementation of the $\texttt{sort}^o$ corresponds to this definition literally: see Figure 12.

This relation can be used for both sorting a list and generating permutations, depending on which argument is passed into it. One drawback this implementation has is that its performance in the two directions is drastically different and hinges on the order of two relation calls to $\texttt{smallest}^o$ and $\texttt{sort}^o$. When the call to $\texttt{smallest}^o$ comes first, sorting works fine while permutation generation times out on lists of length 4. Reordering two calls makes it possible to generate permutations for longer lists, however sorting direction starts to time out on lists of length 5.

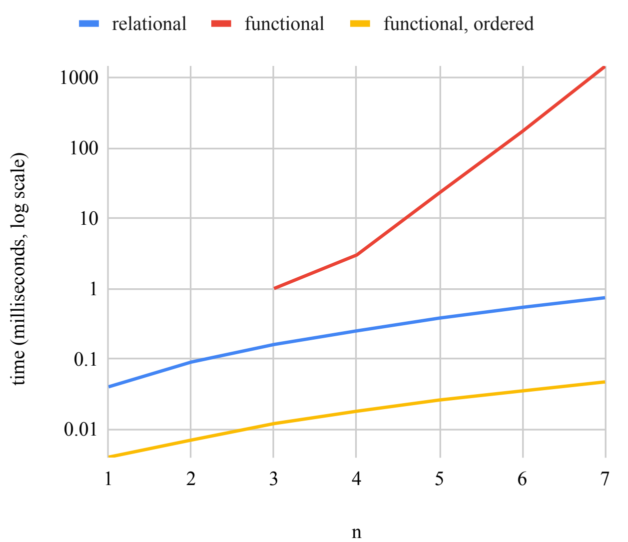
The only way a programmer can implement the relation in such a way that both directions work well, is by duplicating a conjunction with the two orders mentioned. Even though it leads to somewhat decent performance, it is far from elegant and also increases the amount of work to be done to compute any answer. Mode analysis is a better approach to reordering the conjuncts according to the direction needed. Accompanied by the functional conversion, it also improves the performance significantly: see Figure 11. Table 11a demonstrates execution time of sorting, while Table 11b — of generating permutations. Execution of a query was aborted after reaching the timeout of 30 seconds. Both tables contain columns with execution times of a relation



(a) Multiplication: `mulo n 10 q`



(b) Division: `mulo (n/10) q n`



(c) Generation: `take n (mulo 10 q r)`

$$
\begin{array}{l}
\textbf{let rec}\ \mathtt{mul}^o\ \mathtt{x}^{g\to g}\ \mathtt{y}^{f\to g}\ \mathtt{z}^{f\to g}\ = \\
\quad (\mathtt{x}^{g\to g} \equiv \mathtt{O}\ \wedge\ \mathtt{z}^{f\to g} \equiv \mathtt{O})\ \vee \\
\quad (\mathtt{x}^{g\to g} \equiv \mathtt{S}\ \mathtt{x}_1^{f\to g}\ \wedge \\
\quad\ \mathtt{add}^o\ \mathtt{y}^{f\to g}\ \mathtt{z}_1^{f\to g}\ \mathtt{z}^{f\to g}\ \wedge \\
\quad\ \mathtt{mul}^o\ \mathtt{x}_1^{g\to g}\ \mathtt{y}^{g\to g}\ \mathtt{z}_1^{g\to g}\ )
\end{array}
$$

(d) Inefficient mode

$$
\begin{array}{l}
\textbf{let rec}\ \mathtt{mul}^o\ \mathtt{x}^{g\to g}\ \mathtt{y}^{f\to g}\ \mathtt{z}^{f\to g}\ = \\
\quad (\mathtt{x}^{g\to g} \equiv \mathtt{O}\ \wedge\ \mathtt{z}^{f\to g} \equiv \mathtt{O})\ \vee \\
\quad (\mathtt{x}^{g\to g} \equiv \mathtt{S}\ \mathtt{x}_1^{f\to g})\ \wedge \\
\quad\ \mathtt{mul}^o\ \mathtt{x}_1^{g\to g}\ \mathtt{y}^{f\to g}\ \mathtt{z}_1^{f\to g}\ \wedge \\
\quad\ \mathtt{add}^o\ \mathtt{y}^{g\to g}\ \mathtt{z}_1^{g\to g}\ \mathtt{z}^{f\to g}\ )
\end{array}
$$

(e) Efficient mode

Figure 10. Execution times of the multiplication relation

with the calls sorted in the two ways described, and the execution time of the result of functional conversion. Notice, that the functional version is significantly faster than the relational version with the optimal order of calls.

It is worth noting that this relation executes too slowly to be practical even after the functional conversion. It comes from the properties of the algorithm as well as using Peano numbers. However this relation is illustrative of the ways relational programs are supposed to be written and that their execution in the reverse direction can be improved by using sophisticated analyses rather than resorting to inelegant software engineering practices.

5.4 Deterministic Directions

Running in some directions, relations produce deterministic results. For example, any forward direction of a relation created by the relational conversion produces a single result, since it mimics the original function. The guard directions

	Relation		Function
	`sorto` `smallesto`	`smallesto` `sorto`	
`[3;2;1;0]`	0.077s	0.004s	0.000s
`[4;3;2;1;0]`	timeout	0.005s	0.000s
`[31;...;0]`	timeout	1.058s	0.006s
`[262;...;0]`	timeout	timeout	1.045s

(a) Sorting direction

	Relation		Function
	`smallesto` `sorto`	`sorto` `smallesto`	
`[0;1;2]`	0.013s	0.004s	0.004s
`[0;1;2;3]`	timeout	0.005s	0.005s
`[0;...;6]`	timeout	0.999s	0.021s
`[0;...;8]`	timeout	timeout	1.543s

(b) Permutation generation direction

Figure 11. Relational sorting evaluation results

```
let rec sort^o xs sorted =
  (xs ≡ [] ∧ sorted ≡ []) ∨
  (fresh smallest, others, sorted_1 in
     xs ≡ smallest : sorted_1 ∧
     sort^o others sorted_1 ∧
     smallest^o xs smallest others)
```

Figure 12. Relational sorting in miniKanren

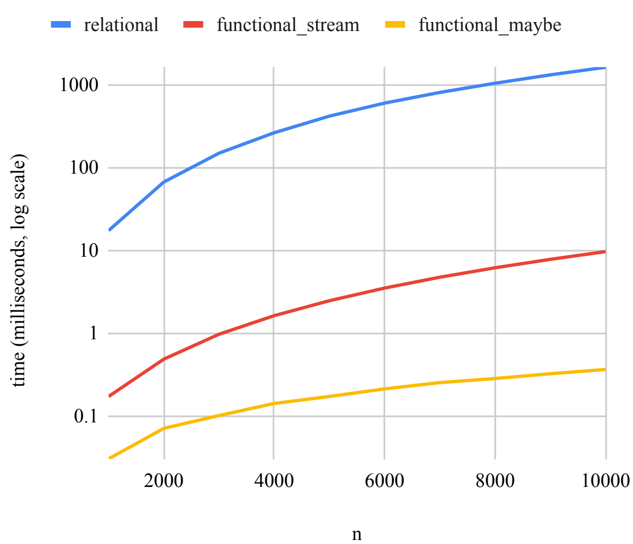


Figure 13. Execution time of division: `take n (mul q 10 1000)`

are semi-deterministic: they may fail, but if they succeed, they produce a single unit value. If the addition relation is run with one of the first two arguments *out*, it acts as subtraction and is also deterministic.

For such directions, there is no need to model nondeterminism with the Stream monad. Semi-determinism can be expressed with a Maybe monad, while deterministic directions can be converted into simple functions. Our implementation of functional conversion only restricts the computations to be monadic, it does not specify which monad to use. By picking other monads, we can achieve performance improvement. For example, using Maybe for division reduces its execution time 30 times in addition to the 2 orders of magnitude improvement from the functional conversion itself: see Figure 13.

6 Discussion

Our experiments indicated that the functional conversion is capable of improving performance of relational computations significantly in the known directions. The improvement stems from eliminating costly unifications in favor of the cheaper equality checks and pattern matches. Besides this, we employed some heuristics which push lower-cost computations to happen sooner while delaying higher-cost ones. It is also possible to take into account determinism of some directions and improve performance of them even more by picking an appropriate monad.

We used heuristics to guide the mode analysis and there are other projects [4, 15] which do the same achieving satisfactory results. It is not currently clear if the heuristics we used are universal enough. However, it is always safe to run any deterministic computations because they never increase the search space. We believe that it is necessary to integrate determinism check in the mode analysis so that the more efficient modes such as the one presented in Figure 10e could be achieved more justifiably.

We also think that further integration with specialization techniques such as partial deduction may benefit the conversion even more [26]. For example, the third argument of the propositional evaluator can be either **true** or **false**. Specializing the evaluator for these two values may help to shave off even more time.

7 Related Work

A mode generalizes the concept of a direction; this terminology is commonly used in the conventional logic programming community. In its most primitive form, a mode specifies which arguments of a relation will be known at runtime (input) and which are expected to be computed (output). Several logic programming languages have mode systems used for optimizations [23, 25, 28], with Mercury[4] standing out among them. Mercury is a modern functional-logic programming language with a complicated mode system capable not only of describing directions, but also specifying if a relation in the given mode is deterministic, among other things [17, 21].

The mode system of Mercury is *prescriptive* which means that the mode dictates the data flow. Mercury translates the logic subset of the language into a functional programming language according to the mode assigned to the relation. The semantics of a Mercury program exists only when the mode is assigned. This is not the case for a miniKanren program whose semantics is the bag of answers it produces [19] regardless of the direction, data flow or the order of subgoals within the definition. In our paper we aim to create a *descriptive* mode system for miniKanren which does not impose constraints on its execution. As another consequence, we are free to compare the execution time of programs with and without any optimizations, which Mercury papers do not usually do.

There are multiple papers describing automatic mode inference of logic programs [6, 18, 20]. The most common way to implement mode inference is by abstract interpretation as introduced in [8]. Mercury utilizes this approach [21] in its implementation to guide the compilation. This mode system proved to be not expressive enough in the context of mode polymorphism, so they researched the use of constraint systems for mode inference [17]. While being more precise, this system proved to be too slow to be used in the compiler.

Moreover, the compiler of Mercury is highly complicated and demands many annotations from the end-user. They include type, mode, uniqueness, and determinism specifications. Many miniKanren languages are embedded into host languages which are not typed and thus we cannot rely on type information in our conversion. It is also impossible to do what Mercury compiler does as a light-weight embedded DSL which is one of the design principles of the miniKanren family. Thus, our goal is to develop the simplest approach to mode analysis which is capable of improving the performance of the verifier-to-solver approach with the least amount of annotations needed from the user — ideally, only the top-level relation call should be annotated.

8 Conclusion and Future Work

In this paper, we described a semi-automatic functional conversion of a miniKanren relation with a fixed direction into a functional language. The conversion and mode analysis used are rather simple and do not rely on the type system which will make it easier to implement as a part of other miniKanren implementations. We implemented the proposed conversion and applied it to a set of relations, resulting in significant performance enhancement, as demonstrated in our evaluation. As part of the future work, we plan to augment the mode analysis with a determinism check. We also plan to integrate the functional conversion with specialization techniques such as partial deduction.

[4] Website of the Mercury programming language: https://mercurylang.org/

References

[1] Sergei Abramov and Robert Glück. 2000. Combining Semantics with Non-standard Interpreter Hierarchies. In *FST TCS 2000: Foundations of Software Technology and Theoretical Computer Science*, Sanjiv Kapoor and Sanjiva Prasad (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 201–213.

[2] Sergei Abramov and Robert Glück. 2002. *Principles of Inverse Computation and the Universal Resolving Algorithm.* Springer Berlin Heidelberg, Berlin, Heidelberg, 269–295. https://doi.org/10.1007/3-540-36377-7_13

[3] Bogdan Aman, Gabriel Ciobanu, Robert Glück, Robin Kaarsgaard, Jarkko Kari, Martin Kutrib, Ivan Lanese, Claudio Antares Mezzina, Łukasz Mikulski, Rajagopal Nagarajan, et al. 2020. Foundations of reversible computation. *Reversible Computation: Extending Horizons of Computing: Selected Results of the COST Action IC1405 12* (2020), 1–40.

[4] Lukas Bulwahn. 2011. Smart test data generators via logic programming. In *Technical Communications of the 27th International Conference on Logic Programming (ICLP'11)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[5] William E Byrd, Michael Ballantyne, Gregory Rosenblatt, and Matthew Might. 2017. A unified approach to solving seven programming problems (functional pearl). *Proceedings of the ACM on Programming Languages* 1, ICFP (2017), 1–26.

[6] Saumya K Debray and David S Warren. 1988. Automatic mode inference for logic programs. *The Journal of Logic Programming* 5, 3 (1988), 207–229.

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

[8] Gerda Janssens and Maurice Bruynooghe. 1992. Deriving descriptions of possible values of program variables by means of abstract interpretation. *The Journal of Logic Programming* 13, 2-3 (1992), 205–258.

[9] Oleg Kiselyov, Chung-chieh Shan, Daniel P. Friedman, and Amr Sabry. 2005. Backtracking, Interleaving, and Terminating Monad Transformers: (Functional Pearl). In *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming* (Tallinn, Estonia) *(ICFP '05)*. Association for Computing Machinery, New York, NY, USA, 192—-203. https://doi.org/10.1145/1086365.1086390

[10] Dmitry Kosarev, Petr Lozov, and Dmitry Boulytchev. 2020. Relational synthesis for pattern matching. In *Asian Symposium on Programming Languages and Systems*, Bruno C. d. S. Oliveira (Ed.). Springer, Springer International Publishing, Cham, 293–310.

[11] Dmitrii Kosarev, Peter Lozov, Denis Fokin, and Dmitry Boulytchev. 2022. On a Declarative Guideline-Directed UI Layout Synthesis. (2022).

[12] Peter Lozov, Dmitry Kosarev, Dmitry Ivanov, and Dmitry Boulytchev. 2023. Relational Solver for Java Generics Type System. In *Logic-Based Program Synthesis and Transformation*. Springer International Publishing.

[13] Petr Lozov, Ekaterina Verbitskaia, and Dmitry Boulytchev. 2019. Relational interpreters for search problems. In *Relational Programming Workshop*. 43.

[14] Petr Lozov, Andrei Vyatkin, and Dmitry Boulytchev. 2018. Typed Relational Conversion. In *Trends in Functional Programming*, Meng Wang and Scott Owens (Eds.). Springer International Publishing, Cham, 39–58.

[15] Gergely Lukácsy. 2008. Semantic technologies based on logic programming. (2008).

[16] HARRY G. MAIRSON. 2004. FUNCTIONAL PEARL Linear lambda calculus and PTIME-completeness. *Journal of Functional Programming*

14, 6 (2004), 623–633. https://doi.org/10.1017/S0956796804005131

[17] David Overton, Zoltan Somogyi, and Peter J Stuckey. 2002. Constraint-based mode analysis of Mercury. In *Proceedings of the 4th ACM SIGPLAN international conference on Principles and practice of declarative programming*. 109–120.

[18] Olivier Ridoux, Patrice Boizumault, and Frédéric Malésieux. 1999. Typed Static Analysis: Application to Groundness Analysis of Prolog and λ Prolog. In *International Symposium on Functional and Logic Programming*. Springer, 267–283.

[19] Dmitry Rozplokhas, Andrey Vyatkin, and Dmitry Boulytchev. 2020. Certified Semantics for Relational Programming. In *Asian Symposium on Programming Languages and Systems*, Bruno C. d. S. Oliveira (Ed.). Springer, Springer International Publishing, Cham, 167–185. https://doi.org/10.1007/978-3-030-64437-6_9

[20] Jan-Georg Smaus, Patricia M Hill, and Andy King. 2000. Mode analysis domains for typed logic programs. In *Logic-Based Program Synthesis and Transformation: 9th International Workshop, LOPSTR'99, Venice, Italy, September 22-24, 1999 Selected Papers 9*. Springer, 82–101.

[21] Zoltan Somogyi. 1987. A System of Precise Models for Logic Programs.. In *ICLP*. Citeseer, 769–787.

[22] Zoltan Somogyi, Fergus Henderson, and Thomas Conway. 1996. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *The Journal of Logic Programming* 29, 1-3 (1996), 17–64.

[23] James A Thom and Justin Zobel. 1986. *NU-Prolog*. Technical Report. Citeseer.

[24] Pawel Urzyczyn. 1997. Inhabitation in typed lambda-calculi (a syntactic approach). In *Typed Lambda Calculi and Applications*, Philippe de Groote and J. Roger Hindley (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 373–389.

[25] Peter Van Roy and Alvin M. Despain. 1992. High-performance logic programming with the Aquarius Prolog compiler. *Computer* 25, 1 (1992), 54–68.

[26] Ekaterina Verbitskaia, Daniil Berezun, and Dmitry Boulytchev. 2021. An Empirical Study of Partial Deduction for miniKanren. *arXiv preprint arXiv:2109.02814* (2021).

[27] Ekaterina Verbitskaia, Daniil Berezun, and Dmitry Boulytchev. 2022. On a Direction-Driven Functional Conversion. In *Relational Programming Workshop*.

[28] David HD Warren. 1977. Implementing Prologcompiling predicate logic programs. *Research Reports 39 and 40, Dpt. of Artificial Intelligence, Univ. of Edinburgh* (1977).