Semi-Automated Direction-Driven Functional Conversion

EKATERINA VERBITSKAIA, JetBrains Research, Serbia
IGOR ENGEL, JetBrains Research, Germany
DANIIL BEREZUN, JetBrains Research, Netherlands

One of the most attractive applications of relational programming is program inversion. It offers an approach to solving complex problems by transforming verifiers into solvers with relatively low effort. Unfortunately, program inverters often suffer form interpretation overhead, leading to suboptimal performance compared to direct program inversions. A prior study introduced a functional conversion scheme capable of creating inversions of miniKanren specifications with respect to a known fixed direction. This paper expands upon it by providing a semi-automated functional conversion algorithm. Our evaluation demonstrates significant performance improvements achieved through functional conversion.

CCS Concepts: • **Computer systems organization → Embedded systems**; *Redundancy*; Robotics; • **Networks** → Network reliability.

Additional Key Words and Phrases: datasets, neural networks, gaze detection, text tagging

ACM Reference Format:
Ekaterina Verbitskaia, Igor Engel, and Daniil Berezun. 2023. Semi-Automated Direction-Driven Functional Conversion. 1, 1 (June 2023), 12 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

One of the most attractive applications of relational programming is *program inversion*. It comes in handy, when the program being inverted is a relational interpreter of some sort: this way an interpreter for a programming language may be used for program synthesis, a type checker — to solve type inhabitation problem and so on [3, 4]. Constructing relational interpreters out of functional implementations can be done automatically [5] by relational conversion. miniKanren along with relational conversion act as a *program inverter*. However, it is important to note that program inverters exhibit lower performance compared to directly executing an inversion of the original program due to the interpretation overhead [1, 2].

Relational programs do not exist on their own: they are a part of a host program, which utilizes query results in some way. The host languages are not expected to be able to process logic variables, nondeterminism and other aspects of relational computations. The host program usually only deals with a finite subset of answers, which have been reified into a ground representation, meaning they do not include any logic variables.

When a relation is expected to produce ground answers, and the direction in which it is intended to be run is known, then it becomes possible to convert it into a function which may execute significantly faster than its relational counterpart. Performance improvement comes from reducing interpretation overhead as well as replacing expensive unifications with considerably faster equality checks, assignments and pattern matches of a host language. An informal functional conversion

Authors' addresses: Ekaterina Verbitskaia, kajigor@gmail.com, JetBrains Research, Belgrade, Serbia; Igor Engel, abc@def.com, JetBrains Research, Bremen, Germany; Daniil Berezun, abc@def.com, JetBrains Research, Amsterdam, Netherlands.

XXXX-XXXX/2023/6-ART $15.00
https://doi.org/XXXXXXX.XXXXXXX

scheme was introduced in the paper [6]. We are building upon this research effort, presenting a semi-automatic functional conversion algorithm and implementation for a minimal core relational programming language microKanren. This paper focuses on converting to the target languages of Haskell and OCaml, although other languages can also be considered as potential target languages. Our evaluation showed performance improvement of 2.5 times for propositional formulas synthesis and up to 3 degrees of magnitude improvement for relations over Peano numbers.

2 BACKGROUND

In this section, we describe the abstract syntax of miniKanren version used in this paper and describe a concept of modes which was developed earlier for other logic languages.

2.1 Normal Form Abstract Syntax of miniKanren

To simplify the functional conversion scheme, we consider miniKanren relations to be in the superhomogeneous normal form used in the mercury programming language [?]. Converting an arbitrary miniKanren relation into the normal form is a simple syntactic transformation which me 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.

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

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 following examples:

$$C\,(x_1, x_2) \equiv C\,(C\,(y_1, y_2)\,, y_3) \Rightarrow x_1 \equiv C\,(y_1, y_2) \wedge x_2 \equiv y_3$$

$$C\,(C\,(x_1, x_2)\,, x_3) \equiv C\,(C\,(y_1, y_2)\,, y_3) \Rightarrow x_1 \equiv y_1 \wedge x_2 \equiv y_2 \wedge x_3 \equiv y_3$$

$$add^o\,(x, x, z) \Rightarrow x_1 \equiv x_2 \wedge add^o\,(x_1, x_2, z)$$

Unification in the normal form is restricted to always unify a variable with a term. We also prohibit using conjunctions inside disjunctions. The normalization procedure declares a new relation whenever this is encountered.

The complete abstract syntax of the miniKanren language used in this paper is presented in figure 1.

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

Fig. 1. Abstract syntax of miniKanren in the normal form

2.2 Modes

A mode generalizes the concept of a direction and is the terminology most commonly used in the larger logic programming community. In its most primitive form, a mode specifies which arguments of a relation are going to be known at runtime (input) and which are expected to be computed (output). Several logic programming languages has mode systems used for optimizations, with MERCURY standing out among them. MERCURY is a modern functional-logic programming with a complicated mode system capable not only to describe a direction, but also whether the relation in the given mode is deterministic, among other things.

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 $g \rightarrow g$, while *out* — $f \rightarrow g$. 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 2 shows examples of the normalized MINIKANREN relations with mode inferred for the forward and backward direction. We use superscript annotation for variables to represent their modes visually. Notice the different order of conjuncts in the bodies of the add^o relation in different modes.

$$
\begin{array}{l}
\textbf{let}\ \mathsf{double}^o\ \mathsf{x}^{g\to g}\ \mathsf{r}^{f\to g}\ = \\
\quad \mathsf{addo}^o\ \mathsf{x}_1^{g\to g}\ \mathsf{x}_2^{g\to g}\ \mathsf{r}^{f\to g}\ \wedge \\
\quad \mathsf{x}_1^{g\to g}\ \equiv\ \mathsf{x}_2^{g\to g} \\
\\
\textbf{let rec}\ \mathsf{add}^o\ \mathsf{x}^{g\to g}\ \mathsf{y}^{g\to g}\ \mathsf{z}^{f\to g}\ = \\
\quad (\mathsf{x}^{g\to g}\ \equiv\ 0\ \wedge\ \mathsf{y}^{g\to g}\ \equiv\ \mathsf{z}^{f\to g})\ \vee \\
\quad (\mathsf{x}^{g\to g}\ \equiv\ \mathsf{S}\ \mathsf{x}_1^{f\to g}\ \wedge \\
\quad\ \ \mathsf{add}^o\ \mathsf{x}_1^{g\to g}\ \mathsf{y}^{g\to g}\ \mathsf{z}_1^{f\to g}\ \wedge \\
\quad\ \ \mathsf{z}^{f\to g}\ \equiv\ \mathsf{S}\ \mathsf{z}_1^{g\to g})
\end{array}
$$

(a) Forward direction

$$
\begin{array}{l}
\textbf{let}\ \mathsf{double}^o\ \mathsf{x}^{f\to g}\ \mathsf{r}^{g\to g}\ = \\
\quad \mathsf{addo}^o\ \mathsf{x}_1^{f\to g}\ \mathsf{x}_2^{f\to g}\ \mathsf{r}^{g\to g}\ \wedge \\
\quad \mathsf{x}_1^{g\to g}\ \equiv\ \mathsf{x}_2^{g\to g} \\
\\
\textbf{let rec}\ \mathsf{add}^o\ \mathsf{x}^{f\to g}\ \mathsf{y}^{f\to g}\ \mathsf{z}^{g\to g}\ = \\
\quad (\mathsf{x}^{f\to g}\ \equiv\ 0\ \wedge\ \mathsf{y}^{f\to g}\ \equiv\ \mathsf{z}^{g\to g})\ \vee \\
\quad (\mathsf{z}^{f\to g}\ \equiv\ \mathsf{S}\ \mathsf{z}_1^{g\to g}\ \wedge \\
\quad\ \ \mathsf{add}^o\ \mathsf{x}_1^{f\to g}\ \mathsf{y}^{f\to g}\ \mathsf{z}_1^{g\to g}\ \wedge \\
\quad\ \ \mathsf{x}^{f\to g}\ \equiv\ \mathsf{S}\ \mathsf{x}_1^{g\to g})
\end{array}
$$

(b) Backward direction

Fig. 2. Normalized doubling and addition relations with mode annotations

3 FUNCTIONAL CONVERSION IN MINIKANREN

In this section, we describe the functional conversion algorithm. The reader is encouraged to first read the paper [6] 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. When relational conversion is used to create a relation, the order of the subgoals only really suits the forward direction, whereas the relation is not intended to be run in it.

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 be further pretty printed into concrete functional programming languages, in our case Haskell and OCaml.

3.1 Mode Inference

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 figure 1.

```
1  modeInfer (R_i (x_1, ..., x_{k_i}) ≡ body) = (R_i (x_1, ..., x_{k_i}) ≡ (modeInferDisj body))
2
3  modeInferDisj (⋁ (c_1, ..., c_n)) = ⋁(modeInferConj c_1, ..., modeInferConj c_n)
4
5  modeInferConj (⋀ (g_1, ..., g_n)) =
6    let (picked, theRest) = pickConjunction([g_1, ..., g_n]) in
7    let moddedPicked = modeInferBase picked in
8    let moddedConjs = modeInferConj (⋀ theRest) in
9    ⋀(moddedPicked : moddedConjs)
10
11 pickConjunction goals =
12   pickGuard goals <|>
13   pickAssignment goals <|>
14   pickMatch goals <|>
15   pickCallGuard goals <|>
16   pickCall goals <|>
17   pickGenerator 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 which 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 **??** in figure 1). Since the body is normalized, it can only be a disjunction. Each disjunct is analyzed independently (see line **??**), 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 has some initial instantiation, and some of them also have some final instantiation. Mode analyzing 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 some variables in 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 unifications, we propose to use *generators* which produce all possible 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 may 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 `pickConjunction` selects the first guard unification it can find. If no guard is present, then it searches for the first assignment (the function <|> in line **??** is responsible for this), then for the match. If all unifications in the conjunction are generators, then the search continues among relation calls. First it selects relation calls with all ground arguments, then with some ground arguments, and only if there are none of those, it picks a generator.

Once one conjunct is picked, it is analyzed (see line **??**). 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 **??**). If any new modes for any of the relations are encountered, they are also mode analyzed.

It is worth noticing that any relation can be a generator. We cannot judge the relation to be a generator solely by its mode: 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 — is 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 4). 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 3. The conversion follows quite naturally from the modded relation and the syntax of the intermediate representation.

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

Fig. 3. Abstract syntax of the intermediate language $\mathcal{F}$

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 correspond to a unification, and 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 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 a set of examples which demonstrate mode analysis and conversion results.

4.1 Multiplication Relation

Figure 4 shows the implementation of the multiplication relation $\texttt{mul}^o$, the mode analysis result for mode $\texttt{mul}^o\ \texttt{x}^{f\to g}\ \texttt{y}^{g\to g}\ \texttt{z}^{g\to g}$, 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: $\texttt{mul}^o\ \texttt{x}_1^{f\to g}\ \texttt{y}^{g\to g}\ \texttt{z}_1^{f\to g} \land \texttt{add}^o\ \texttt{y}^{g\to g}\ \texttt{z}_1^{g\to g}\ \texttt{z}^{g\to g}$. However it is possible to derive this optimal order, if determinism analysis is employed: $\texttt{add}^o\ \texttt{y}^{g\to g}\ \texttt{z}_1^{f\to g}\ \texttt{z}^{g\to g}$ is deterministic while $\texttt{mul}^o\ \texttt{x}_1^{f\to g}\ \texttt{y}^{g\to g}\ \texttt{z}_1^{f\to g}$ 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 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 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.

```
let rec mul^o x y z = conde [
  (x ≡ 0 ∧ z ≡ 0);
  (fresh (x_1 z_1)
    (x ≡ S x_1 ∧
     add^o y z_1 z ∧
     mul^o x_1 y z_1)) ]
```

(a) Implementation in miniKanren

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

Fig. 4. Multiplication relation

4.2 The Mode of Addition Relation which Needs a Generator

Consider the example of the addition relation in mode $\mathtt{add}^o\ \mathtt{x}^{g \to g}\ \mathtt{y}^{f \to g}\ \mathtt{z}^{f \to g}$ presented in figure 5. The unification in the first disjunct of this relation involves two free variables. We use a generator `gen_addoII0_x2` to generate a stream of ground values for the variable z which is passed into the function `addII0` 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 is presented in figure 5b.

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 miniKanren.

$$
\begin{array}{l}
\textbf{let rec } \mathsf{add}^o\ \mathsf{x}^{g\to g}\ \mathsf{y}^{f\to g}\ \mathsf{z}^{f\to g} = \\
\quad (\mathsf{x}^{g\to g} \equiv 0 \ \wedge\ \mathsf{y}^{f\to g} \equiv \mathsf{z}^{f\to g}) \ \vee \\
\quad (\mathsf{x}^{g\to g} \equiv \mathsf{S}\ \mathsf{x}_1^{f\to g} \ \wedge \\
\quad\ \mathsf{add}^o\ \mathsf{x}_1^{g\to g}\ \mathsf{y}^{f\to g}\ \mathsf{z}_1^{f\to g} \ \wedge \\
\quad\ \mathsf{z}^{f\to g} \equiv \mathsf{S}\ \mathsf{z}_1^{g\to g})
\end{array}
$$

(a) Mode inference result

```
genNat = msum
  [ return 0
  , do { x ← genNat
       ; return (S x) } ]
runAddoII0 x = addoII0 x genNat
```

(b) Generator of Peano numbers

```
addoI00 x0 gen_addoI00_x2 = msum
  [ do { guard (x0 == 0)
       ; (x1, x2) ← do { x2 ← gen_addoI00_x2 ; return (x2, x2) }
       ; return (x1, x2) }
  , do { x3 ← case x0 of { S y3 → return y3 ; _ → mzero }
       ; (x1, x4) ← addoI00 x3 gen_addoI00_x2
       ; let {x2 = S x4} ; return (x1, x2) } ]
```

(c) Functional conversion

Fig. 5. Addition relation when only the first argument is *in*

5 EVALUATION

To evaluate our functional conversion scheme, we implemented the proposed algorithm in Haskell. We compared execution time of several relations in different directions against their functional counterparts in the OCaml language. Here we showcase two relational programs and their conversions. The implementation of the functional conversion[1] as well as the execution code[2] can be found on Github.

5.1 Evaluator of Propositional Formulas

In this example, we converted a relational evaluator of propositional formulas: see figure 6. 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 6a), allows one to synthesize formulas which can be evaluated to the given number. The conversion of this relation does not involve any generators and is presented in figure 6b.

We run an experiment to compare 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 7. The functional conversion improved execution time of the query about 2.5 times from $724ms$ to $291ms$ for retrieving 10000 formulas.

5.2 Multiplication

In this example, we converted the multiplication relation in several directions and compared it to the relational counterparts: see figure 8. Functional conversion significantly reduced execution time in most directions.

[1]The repository of the functional conversion project https://github.com/kajigor/uKanren_transformations
[2]Evaluation code https://github.com/kajigor/miniKanren-func

```
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

Fig. 6. Evaluator of propositional formulas

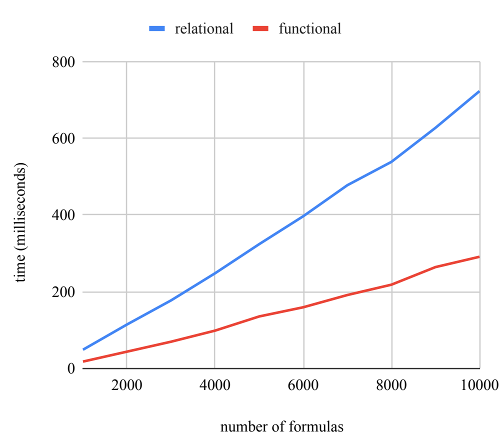
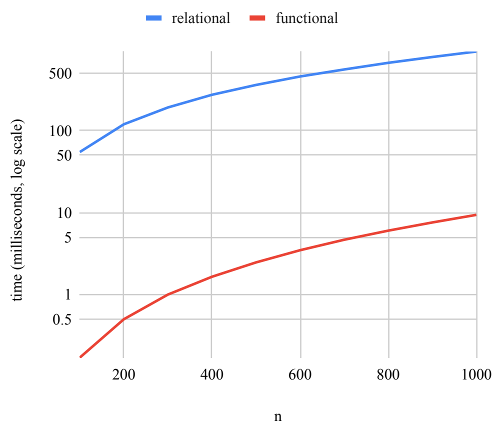
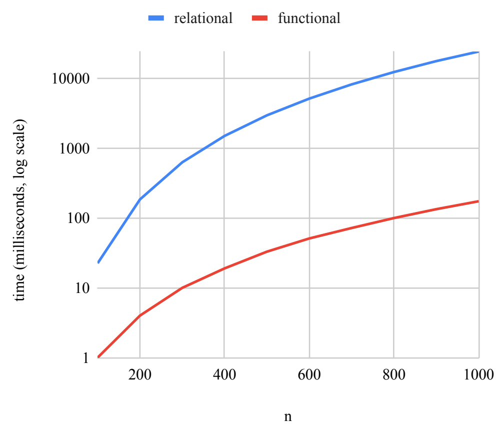


Fig. 7. Execution time of the evaluators of propositional formulas, eval [**true**; **false**; **true**] q **true**

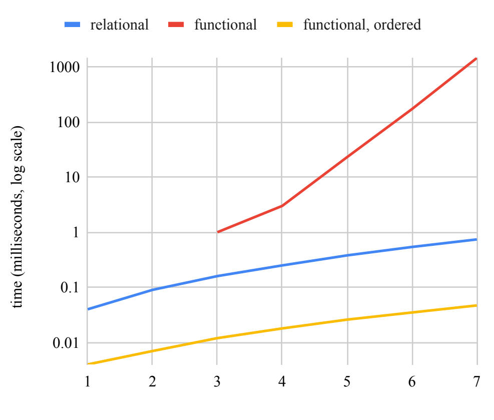
In the forward direction we run the query mul^o n 10 q with n in the range from 100 to 1000, and the functional conversion was 2 degrees of magnitude faster: 927ms vs 9.4ms for the largest n. In the direction which serves as division we run the query mul^o (n /10) q n with n ranging from 100 to 1000. Here performance improved 3 degrees of magnitude: from 24s to 1.7s for the largest n. Even more impressive was the backward direction mul^o x$^{f\to g}$ y$^{f\to g}$ z$^{g\to g}$. Querying for all 16 pairs



(a) Multiplication: `mulo n 10 q`



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



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

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

(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

Fig. 8. Execution times of the multiplication relation

of divisors of 1000 ($\mathtt{mul}^o$ `q r 1000`) took OCanren about 32.9*s*, while the functional conversion succeeded in 1.1*s*.

What was surprising is the mode $\mathtt{mul}^o\ \mathtt{x}^{g\to g}\ \mathtt{y}^{f\to g}\ \mathtt{z}^{f\to g}$. In this case the functional conversion was not only worse than its relational source, its performance degraded exponentially with the amount 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 4b). The source of this terrible behavior was the suboptimal order of the calls in the second disjunct of the $\mathtt{mul}^o$ relation in the corresponding mode (see figure 8d). In this case the call $\mathtt{add}^o\ \mathtt{y}^{f\to g}\ \mathtt{z}_1^{f\to g}\ \mathtt{z}^{f\to g}$ is put first, which generates all possible triples in the addition relation before filtering them by means of the call $\mathtt{mul}^o\ \mathtt{x}_1^{g\to g}\ \mathtt{y}^{g\to g}\ \mathtt{z}_1^{g\to g}$. The other order of calls is much better (see figure 8e): 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. It is not clear if this heuristics is universal.

5.3 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 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 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 it is. By picking other monads, we can achieve performance improvement. For example, using Maybe for division reduces its execution time by 30 times in addition to the 2 orders of magnitude improvement from the functional conversion itself: see figure **??**

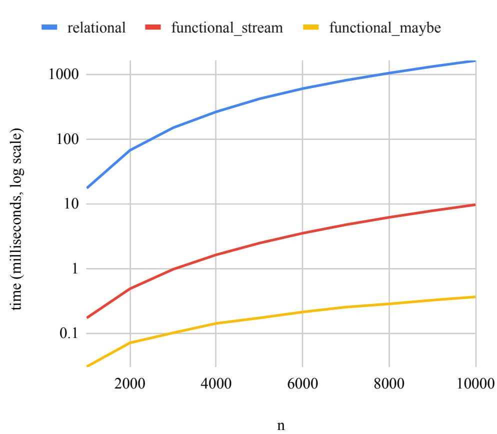


Fig. 9. Execution time of division, query `mul q 10 1000`

6 DISCUSSION

Our experiments showed that the functional conversion is capable to significantly improve performance of relational computations in the known directions. The improvement stems from eliminating costly unifications in favor of the cheaper equality checks and pattern matches. Beside 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.

It is currently not unclear 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 8e could be achieved more justifiably.

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

7 CONCLUSION AND FUTURE WORK

In this paper, we described a semi-automatic functional conversion from a miniKanren relation with a fixed direction into a functional language. 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 mode analysis with determinism check. We also plan to integrate the functional conversion with specialization techniques such as partial deduction.

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] William E. Byrd, Eric Holk, and Daniel P. Friedman. 2012. MiniKanren, Live and Untagged: Quine Generation via Relational Interpreters (Programming Pearl). In *Proceedings of the Annual Workshop on Scheme and Functional Programming* (Copenhagen, Denmark) *(Scheme '12)*. Association for Computing Machinery, New York, NY, USA, 8—-29. https://doi.org/10.1145/2661103.2661105

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

[5] Petr Lozov, Andrei Vyatkin, and Dmitry Boulytchev. 2018. Typed relational conversion. In *Trends in Functional Programming: 18th International Symposium, TFP 2017, Canterbury, UK, June 19-21, 2017, Revised Selected Papers 18*. Springer, 39–58.

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