Semi-Automated Direction-Driven Functional Conversion

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

A clear and well-documented LaTeX document is presented as an article formatted for publication by ACM in a conference proceedings or journal publication. Based on the "acmart" document class, this article presents and explains many of the common variations, as well as many of the formatting elements an author may use in the preparation of the documentation of their work.

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. 2018. Semi-Automated Direction-Driven Functional Conversion. *J. ACM* 37, 4, Article 111 (August 2018), 6 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 sorts: this way an interpreter for a programming language may be used for program synthesis, a type checker — to solve type inhabitation problem (cite relational interpreters for search). Building relational interpreters out of functional implementations can be done automatically (Cite Lozov's conversion), but the resulting relations are often rather slow. Expertise and effort are required to manually create a relational interpreter with optimal performance. Utilizing program transformations to improve performance of the generated relational interpreters may be a better way to achieve better inversions.

Relational programs do not exist on their own: they are a part of a host program, which utilizes query results in some way. Since host languages are rarely logic, they 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 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 replacing expensive unifications with considerably faster equality checks, assignments and pattern matches of a host language. An informal functional conversion scheme was introduced in the paper (cite last year's miniKanren paper). We are building upon this research effort, presenting a semi-automatic

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.

0004-5411/2018/8-ART111 $15.00
https://doi.org/XXXXXXX.XXXXXXX

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.

(Some evaluation numbers)

2 BACKGROUND

In this section, we describe the abstract syntax of miniKanren version used in this paper and describe mode analysis — one of the steps used in our conversion — which was developed earlier for other languages.

2.1 Abstract Syntax of miniKanren

To simplify the functional conversion scheme, we consider miniKanren relations to be in the normal form (find the proper name). 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.

$$\mathcal{T}_V = V \cup \{C_i\left(x_1, \ldots, x_{k_i}\right) \mid x_j \in V\}$$

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:

$$C\left(x_1, x_2\right) \equiv C\left(C\left(y_1, y_2\right), y_3\right) \iff x_1 \equiv C\left(y_1, y_2\right) \wedge x_2 \equiv y_3$$

$$C\left(C\left(x_1, x_2\right), x_3\right) \equiv C\left(C\left(y_1, y_2\right), y_3\right) \iff x_1 \equiv y_1 \wedge x_2 \equiv y_2 \wedge x_3 \equiv y_3$$

$$add^o\left(x, x, z\right) \iff x_1 \equiv x_2 \wedge add^o\left(x_1, x_2, z\right)$$

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 when this is encountered.

Inverse eta-delay is used in miniKanren to make sure that the computation is sufficiently lazy. Usually, it is allowed to be used anywhere in a goal, but it is only needed to be used on calls. This is why we decided to represent inverse eta-delay as a flag which accompanies calls.

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

$\mathcal{D}_V^N$	$=$	$R_i\left(x_1, \ldots, x_{k_i}\right) \equiv \mathbf{Disj}_V, x_j \in V$	normalized relation definition
$\mathbf{Disj}_V$	$=$	$\bigvee\left(c_1, \ldots, c_n\right), c_i \in \mathbf{Conj}_V$	normal form
$\mathbf{Conj}_V$	$=$	$\bigwedge\left(g_1, \ldots, g_n\right), g_i \in \mathbf{Base}_V$	normal conjunction
$\mathbf{Base}_V$	$=$	$V \equiv \mathcal{T}_V$	flat unification
	$\mid$	$R_i^d\left(x_1, \ldots, x_{k_i}\right), d \in \mathbf{Delay}, x_j \in V$	flat call
$\mathbf{Delay}$	$=$	$\{\text{Delay}, \text{NoDelay}\}$	

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

example of some relation in the concrete syntax, query and result (?)

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 associated with 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* — $f \longrightarrow g$. The instantiation f represents an unbound, of 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 will call a pair of instantiations a mode of a variable, and annotate all variables with their modes, as can be seen in the example (stick some example).

Figure 2 shows examples of MINIKANREN relation with mode inferenced for the forward direction and backward direction. We use superscript annotation for variables to represent modes visually. Notice the different order of conjuncts in the bodies of the add^o relation in different modes.

```
let double^o x^{g→g} r^{f→g} =
  addo^o x1^{g→g} x2^{g→g} r^{f→g} ∧
  x1^{g→g} ≡ x2^{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 x1^{f→g} ∧
   add^o x1^{g→g} y^{g→g} z1^{f→g} ∧
   z^{f→g} ≡ S z1^{g→g}) ]
```

(a) Forward direction

```
let double^o x^{f→g} r^{g→g} =
  addo^o x1^{f→g} x2^{f→g} r^{g→g} ∧
  x1^{g→g} ≡ x2^{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 z1^{g→g} ∧
   add^o x1^{f→g} y^{f→g} z1^{g→g} ∧
   x^{f→g} ≡ S x1^{g→g}) ]
```

(b) Backward direction

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

3 FUNCTIONAL CONVERSION IN MINIKANREN

(1) Discuss the motivation and importance of functional conversion in MINIKANREN.
(2) Present the core contribution: an algorithm for functional conversion.
(3) Explain the algorithm's steps and principles behind functional conversion.

In this section, we describe the functional conversion algorithm. The reader is encouraged to first read the paper (cite) 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 very 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.

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. These two considerations are enough to produce a function, but they do not really influence the performance.

While a relation is executed in the given direction, more and more variables become ground. There are different ways a variable becomes ground. For example, a unification $v \equiv t$, where t is a ground term, and v a free variable, serves as an assignment of the value t to the variable v. We translate this kind of unification into an assignment in a functional language, and it is a very cheap operation. Another cheap kind of unification is $v \equiv t$, where both v and t are ground. This is just an equality check, which is not costly at all, especially taking into the account that terms are flat. Executing such kind of computations early on can only make the search space smaller and thus only improve the performance (is this even true?).

To the contrary, if we handle a unification $v \equiv t$ where v is free, and t contains some free variables, is a nightmare. Since the resulting from the conversion function should return ground values which cannot contain any logic variables, we have to make some way to produce the ground values out of thin air. Having a logic variable somewhere in a term means that it may be replaced by any possible value which makes sense in the context. In case of the typed miniKanren embeddings, it means that it may be replaced by any value of the corresponding] type, and the free logic variable in reality corresponds to often infinite set of possible values. We proposed to use the *generators* to make a stream of all values. Most of the data types are recursive, and thus produce an infinite stream of possible values. Considering this early on in the computations does not limit the search space what so ever, and should be avoided if possible.

We decided to order conjunctions in such a way that they compute as many guards and assignments early, in hopes that this would allow us to avoid generators.

The whole other thing is relations. Each relation can work as a generator, and even though for some relations it is clear, for other it is not. We cannot judge the relation to be a generator solely by its mode: addoIOO generates an infinite stream, while addoOOI does not (check). We can be sure that any relation with only input variables serves as guard and can be called earlier safely.

4 ALGORITHM FOR FUNCTIONAL CONVERSION

(1) Describe the algorithm in detail, step by step.
(2) Discuss the key techniques and transformations involved in functional conversion.
(3) Provide pseudocode or code snippets to illustrate the algorithm.

4.1 Mode Analysis

We only consider two possible modes for variables: *in* and *out*. The first mode is a shorthand for a mapping *ground* $\longrightarrow$ *ground* and correspond to the input variables. The second mode is a shorthand for *free* $\longrightarrow$ *ground* corresponding to the output variables.

The mode inference is run on relation with a fixed direction, whose body has been normalized. First, we initialize all encounters of variables which come from arguments of the relation. For each of these variables, we know that they are either *ground* or *free*, and can propagate this information to the body of the relation.

```
-- Augments variables with the initial mode information:
-- * input variables have initial instantiation g
-- * all other variables have initial instantiation f
initModes :: $\mathcal{D}_V^N \to \mathcal{D}_{(V,M)}^N$

modeInfer ($R_i\left(x_1,\ldots,x_{k_i}\right) \equiv body$) =
  ($R_i\left(x_1,\ldots,x_{k_i}\right) \equiv$ (modeInferDisj body)

modeInferDisj ($\bigvee(c_1,\ldots,c_n)$) =
  $\bigvee$(modeInferConj $c_1,\ldots,$ modeInferConj $c_n$)

modeInferConj ($\bigwedge(g_1,\ldots,g_n)$) =
  let (picked, theRest) = pickConjunction([$g_1,\ldots,g_n$]) in
  let moddedPicked = modeInferBase picked in
  let moddedConjs = modeInferConj ($\bigwedge$ theRest) in
  $\bigwedge$(moddedPicked : moddedConjs)

modeInferBase
```

Listing 1. Mode inference pseudocode

Then we start inference for goal in the body of the relation. This goal is always a disjunction, and each disjunction can be analyzed independently.

To analyze a unification, both sides of it are examined and the information about groundness is propagated.

Once a variable become ground, it stays ground in the subsequent conjuncts.

The order of conjuncts influences the performance of a relation. Some conjuncts, such as guards (introduce the 4 types of unifications sooner), are better to be run as soon as possible: they can either fail or succeed without producing multiple results. Running them sooner can only make the search space smaller. Another example is assignments and matches, which can produce one result in the worst case. Generators, on the other hand, should be delayed as much as possible, since they generally produce infinitely many results.

In light of these considerations, we make a priority list of different conjuncts and pick conjuncts in the order of that priority.

The pseudocode for the mode inference can be found in figure (reference the figure).

5 IMPLEMENTATION AND EXAMPLES

(1) Explain how the algorithm can be implemented in practice.
(2) Provide examples of miniKanren programs before and after functional conversion.
(3) Demonstrate how functional conversion improves code readability and maintainability.

5.1 Implementation

5.1.1 Stream Monad.

5.1.2 Generators.

5.1.3 Determinism Check.

5.2 Examples

5.2.1 *Addition Relation.*

5.2.2 *Multiplication Relation.*

5.2.3 *Propositional Evaluator.*

5.2.4 *Logic Riddles.* Water Pouring Riddle

Wolf-Goat-Cabbage Riddle

Trucks in Desert Riddle

6 EVALUATION AND COMPARISON

(1) Evaluate the effectiveness and performance of the functional conversion algorithm.
(2) Compare the converted functional programs with their original miniKanren counterparts.
(3) Analyze the impact of functional conversion on expressiveness and efficiency.

7 BENEFITS AND LIMITATIONS

(1) Highlight the benefits of functional conversion in miniKanren.
(2) Discuss how functional conversion leads to more modular and reusable code.
(3) Address any limitations or challenges associated with the algorithm.

8 RELATED WORK

(1) Discuss related work on functional conversion or similar approaches in other programming languages or paradigms.
(2) Compare the proposed algorithm with existing techniques.

9 CONCLUSION AND FUTURE WORK

(1) Summarize the key points discussed in the paper.
(2) Emphasize the contribution of the functional conversion algorithm to miniKanren.
(3) Discuss potential areas for future research and improvements to the algorithm.

9.1 Future Work

- Determinism check
- Pair it with a partial deduction

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009