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), 7 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 a concept of modes which was developed earlier for other logic 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.

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

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:

$$C\left(x_1, x_2\right) \equiv C\left(C\left(y_1, y_2\right), y_3\right) \Rightarrow x_1 \equiv C\left(y_1, y_2\right) \land x_2 \equiv y_3$$

$$C\left(C\left(x_1, x_2\right), x_3\right) \equiv C\left(C\left(y_1, y_2\right), y_3\right) \Rightarrow x_1 \equiv y_1 \land x_2 \equiv y_2 \land x_3 \equiv y_3$$

$$add^o\left(x, x, z\right) \Rightarrow x_1 \equiv x_2 \land add^o\left(x_1, x_2, z\right)$$

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 whenever 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.

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

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 $g \longrightarrow g$, while *out* — $f \longrightarrow g$. 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 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 (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 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.

```
modeInfer (R_i (x_1, ..., x_{k_i}) ≡ body) =
  (R_i (x_1, ..., x_{k_i}) ≡ (modeInferDisj body))

modeInferDisj (⋁ (c_1, ..., c_n)) =
  ⋁(modeInferConj c_1, ..., modeInferConj c_n)

modeInferConj (⋀ (g_1, ..., g_n)) =
  let (picked, theRest) = pickConjunction([g_1, ..., g_n]) in
  let moddedPicked = modeInferBase picked in
  let moddedConjs = modeInferConj (⋀ theRest) in
  ⋀(moddedPicked : moddedConjs)

pickConjunction goals =
  pickGuard goals <|>
  pickAssignment goals <|>
  pickMatch goals <|>
  pickCallGuard goals <|>
  pickCall goals <|>
  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 (reference) in figure (reference)). Since the body is normalized, it can only be a disjunction. Each disjunct is analyzed independently, 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 first guard unification it can find. If no guard is present, then it searches for the first assignment, 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 non of those, it picks a generator.

Once one conjunct is picked, it is analyzed. 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. 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 $\texttt{add}^o\ \texttt{x}^{g \to g}\ \texttt{y}^{f \to g}\ \texttt{z}^{f \to g}$ generates an infinite stream, while $\texttt{add}^o\ \texttt{x}^{f \to g}\ \texttt{y}^{f \to g}\ \texttt{z}^{g \to g}$ does not (check).

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 the do-notation (ref to some later code sample). We use an intermediate representation which draws inspiration from the 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.

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

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.

3.3 Conversion into Concrete Languages

The intermediate representation is then translated into a concrete functional programming language. We provided two translations: into Haskell and OCaml. The first one utilizes metaprogramming framework TemplateHaskell, while the second is a custom pretty-printer. There is no do-notation in OCaml, however let-syntax provides a decent alternative.

4 EXAMPLES

(1) Provide examples of miniKanren programs before and after functional conversion.

4.1 Addition Relation

4.2 Multiplication Relation

4.3 Propositional Evaluator

4.4 Logic Riddles

Water Pouring Riddle

Wolf-Goat-Cabbage Riddle

Trucks in Desert Riddle

5 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.

6 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.

7 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.

8 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.

8.1 Future Work

- Determinism check
- Pair it with a partial deduction

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