# Relational Conversion for OCaml

Petr Lozov Dmitri Boulytchev

St.Petersburg State University
Saint-Petersburg, Russia
lozov.peter@gmail.com dboulytchev@math.spbu.ru

## Abstract

We address the problem of transforming typed functional programs into relational form. In this form a program can be run in various "directions" with various arguments left free, making it possible to acquire different behaviors from a single specification. We present an implementation of relational convertor for a subset of Objective Caml and evaluate it on a number of benchmarks, obtaining some relational programs never written before.

## 1. Introduction

Relational programming is an attractive technique, based on the idea of constructing programs as relations. Many logic programming languages, such as Prolog, Mercury[1], or Curry[2] to some extent can be considered as relational. In this paper we focus on `miniKanren` [1–3].

`miniKanren`[3] was originally designed as a small relational DSL, embedded in Scheme/Racket. The advantage of this approach is the flexibility in combining functional and logic features; in addition `miniKanren` possesses some quite appealing features (complete search, purity, declarativity, etc).

With relational approach, it becomes possible to give simple and elegant solutions for problems otherwise considered as tricky, tough, tedious, or boring. For example, relational interpreters can be used to derive *quines* — programs, which reduce to themselves, as well as *twines* or *trines* (a pairs or triples of programs, reducing to each other) [4]; a straightforward relational description of simply typed lambda calculus [5] inference rules works both as type inferencer and inhabitation problem solver [6]; relational list sorting can be used to generate all permutations [7], etc.

On the other hand, writing relational specification can sometimes be a tricky and error-prone task. Fortunately, many specifications can be written systematically by "generalizing" a certain functional program. From the very beginning the conversion from functional to relational form was considered as an element of relational programming thesaurus [1]. However, the traditional approach — *unnesting* — was formulated for the untyped case, worked only for specifically written programs and, to our knowledge, was never implemented.

We present a generalized form of relational conversion, which can be applied to typed terms in general form. We study the relational conversion for a certain subset of Objective Caml, retaining Hindley-Milner type system with let-polymorphism [8]. As target relational language we use `OCanren` [7] — a typed shallow embedding of `miniKanren` in Objective Caml; as a matter of fact, we transform a functional language into its *relational extension*. Our contribution includes the formal description of the semantics for both the source language and its relational extension and the proofs of static and dynamic correctness of the conversion. Due to space considerations we do not present the formal part here; the report was accepted for presentation at the symposium on Trends in Functional Programming[4].

## 2. Relational Extension and Conversion

`OCanren` can be seen as a relational extension for Objective Caml. The central notion in this extension is the *goal*, which can be an arbitrary expression of reserved goal type $\mathfrak{G}$. There are only five syntactic forms of goals:

- conjunction $g_1 \wedge g_2$ and disjunction $g_1 \vee g_2$ of goals $g_1$ and $g_2$;
- fresh variable introduction `fresh` $(x)$ $g$ for goal $g$;
- unification $t_1 \equiv t_2$ and disequality constraint $t_1 \not\equiv t_2$.

The two last forms constitute the basis for goal construction; here $t_1$ and $t_2$ are *terms*. In `OCanren` a term is an arbitrary expression of polymorphic *logic type* $\uparrow\alpha$. The simplest expression of logic type is a variable, bound in `fresh`. Another example is a value, *injected* into the logic domain with a built-in primitive "$\uparrow$", such as $\uparrow 3$ of type $\uparrow$`int`. As an example of relational specification in `OCanren`, consider the following code snippet:

```
let is_zero n b = (n ≡ ↑0  ∧  b ≡ ↑True)  ∨
                  (n ≢ ↑0  ∧  b ≡ ↑False)
```

[1] `https://mercurylang.org`

[2] `http://www-ps.informatik.uni-kiel.de/currywiki`

[3] `http://minikanren.org`

[4] https://www.cs.kent.ac.uk/events/tfp17

Function `is_zero` implements a binary relation between integers and booleans; when called with specific arguments, it returns a goal which can be executed, returning a *stream of answers*. An element of the stream contains the description of certain constraints for logical variables, which have to be respected in order for the relation to hold. For example,

```
iz_zero q p ⇝ [(q=0,p=True);(q=_.0(≠0),p=False)]
```

The returned stream of answers contains two elements: the first one assigns `q` and `p` values "0" and "`True`", respectively; the second assigns "`False`" to `p` and the constraint to be everything, except zero, to `q`; here "`_.0`" corresponds to a free variable, and ($\neq$0) to its disequality constraint.

Our relational conversion is based on the following very simple idea on the type level: we transform a source term of type $t$ into its relational counterpart of type $[t]$, where the type transformation function $[\bullet]$ is defined as follows:

$$\begin{array}{rcl} [a] & = & a \to \mathfrak{G} \\ [t_1 \to t_2] & = & [t_1] \to [t_2] \\ [\forall \alpha.t] & = & \forall \alpha.\,[t] \end{array}$$

where $a$ — arbitrary ground (non-polymorphic, non-functional type). Despite its type-based description, the conversion itself does not make use of types. The static correctness property, which we have proven, claims that for properly typed source terms the results of conversion are always properly typed in relational sense. The similar result holds for dynamic correctness.

## 3. Evaluation

We implemented the conversion[5] and applied it to a number of programs, providing their relational implementations.

First, we implemented an interpreter for a simple Scheme-like language, converted it into relational interpreter and reproduced quines, twines and trines benchmarks [4].

Next, we implemented the type inference for Hindley-Milner type system and tested it in various directions. For example, our relational inferencer is capable of inferring types:

```
nat_type_inference (↑Abst (↑"x", ↑Var ↑"x")) q ⇝
  [q=Just (TFun (TVar Z, TVar Z))]
```

as well as finding some inhabitants for a given type:

```
nat_type_inference q ↑(Just ↑TBool) ⇝
  [q=Lit (LBool _.24);
   q=Let (_.18, Lit (LInt _.32), Lit (LBool _.80));
   q=Let (_.18, Lit (LBool _.32), Lit (LBool _.80));
   q=Let (_.89, Lit (LBool _.32), Var _.89);
   ...
  ]
```

The first answer corresponds to a boolean constant (`true` of `false`), the second and third — to expressions of the form `let x = A in B`, where $A$ — some integer or boolean constant, $B$ — some boolean constant, and the fourth — to the expression `let x = B in x`, where $B$ — some boolean constant.

Our relational inferencer can also be used to deduce a complete term from an incomplete term and its desirable type:

```
nat_type_inference
  (↑Let (↑"f", q,
         ↑App (↑Var ↑"f",
               ↑Abst (↑"x",
                      ↑App (↑Var ↑"f",
                            ↑Var ↑"x")))))
  ↑(Just ↑TBool)) ⇝
  [q=Abst (_.74, Var (_.74));
   q=Abst (_.44, Abst (_.90, Var (_.90)));
   ...
  ]
```

Here we provide the type inferencer an incomplete term `let f = □ in f (λ x → f x)`, where □ is a hole, and expected type `bool`. The answers provide us with the terms, which can be plugged into the hole: the first one is `λ x → x`, the second — `λ x y → y`. Note, that this example essentially uses the polymorphic part of the type system, as the term `λ f → f (λ x → f x)` can not be typed in STLC.

Finally, we implemented an interpreter of `OCanren`-like relational language and converted it into relational form. As a result, we can run relational programs relationally. For example, for this relational program with a hole (□)

```
let rec add a b c =
   ((a ≡ Z) ∧ (b ≡ c)) ∨
   (fresh (a₀ c₀)
     (a ≡ S a₀) ∧
     □ ∧
     (add a₀ b c₀)
   )
in fresh (x y z) (add x y z)
```

and a number of answers, describing the results of addition of natural numbers in Peano form, our relational interpreter for relational language finds a feasible term to plug into the hole: `c ≡ S c₀`. Our relational language supports disequality constraints as well, which makes it different from existing works[6].

## References


[1] Daniel P. Friedman, William E.Byrd, Oleg Kiselyov. The Reasoned Schemer. The MIT Press, 2005.

[2] Jason Hemann, Daniel P. Friedman. $\mu$Kanren: A Minimal Core for Relational Programming // Proceedings of the 2013 Workshop on Scheme and Functional Programming (Scheme '13).


[5] https://bitbucket.org/peter_lozov/translator-to-minikanren

[6] https://github.com/jasonhemann/micro-in-mini, https://github.com/jpt4/muko

[3] Claire E. Alvis, Jeremiah J. Willcock, Kyle M. Carter, William E. Byrd, Daniel P. Friedman. cKanren: miniKanren with Constraints // Proceedings of the 2011 Workshop on Scheme and Functional Programming (Scheme '11).

[4] William E. Byrd, Eric Holk, Daniel P. Friedman. miniKanren, Live and Untagged: Quine Generation via Relational Interpreters (Programming Pearl) // Proceedings of the 2012 Workshop on Scheme and Functional Programming (Scheme '12).

[5] Henk Barendregt. Lambda Calculi with Types, Handbook of Logic in Computer Science (Vol. 2), 1992.

[6] William E. Byrd. Relational Programming in miniKanren: Techniques, Applications, and Implementations. PhD Thesis, Indiana University, Bloomington, IN, September 30, 2009.

[7] Dmitry Kosarev, Dmitry Boulytchev. Typed Embedding of a Relational Language in OCaml // International Workshop on ML, 2016.

[8] Benjamin Pierce. Types and Programming Languages. MIT Press, 2002.