

# Typed Embedding of a Relational Language in OCaml

Dmitrii Kosarev

Saint Petersburg State University  
Saint Petersburg, Russia

Dmitrii.Kosarev@protonmail.ch

Dmitry Boulytchev

Saint Petersburg State University  
Saint Petersburg, Russia

dboulytchev@math.spbu.ru

We present an implementation of the relational programming language miniKanren as a set of combinators and syntax extensions for OCaml. The key feature of our approach is *polymorphic unification*, which can be used to unify data structures of arbitrary types. In addition we provide a useful generic programming pattern to systematically develop relational specifications in a typed manner, and address the problem of integration of relational subsystems into functional applications.

## 1 Introduction

Relational programming [11] is an attractive technique, based on the idea of constructing programs as relations. As a result, relational programs can be “queried” in various “directions”, making it possible, for example, to simulate reversed execution. Apart from being interesting from a purely theoretical standpoint, this approach may have a practical value: some problems look much simpler when considered as queries to some relational specification [5]. There are a number of appealing examples confirming this observation: a type checker for simply typed lambda calculus (and, at the same time, a type inferencer and solver for the inhabitation problem), an interpreter (capable of generating “quines” — programs producing themselves as a result) [7], list sorting (capable of producing all permutations), etc.

Many logic programming languages, such as Prolog, Mercury [21], or Curry [13] to some extent can be considered relational. We have chosen miniKanren<sup>1</sup> as a model language, because it was specifically designed as a relational DSL, embedded in Scheme/Racket. Being rather a minimalistic language, which can be implemented with just a few data structures and combinators [14, 15], miniKanren found its way into dozens of host languages, including Scala, Haskell and Standard ML. The paradigm behind miniKanren can be described as “lightweight logic programming”<sup>2</sup>.

This paper addresses the problem of embedding miniKanren into OCaml<sup>3</sup> — a statically-typed functional language with a rich type system. A statically-typed implementation would bring us a number of benefits. First, as always, we expect typing to provide a certain level of correctness guarantees, ruling out some pathological programs, which otherwise would provide pathological results. In the context of relational programming, however, typing would additionally help us to interpret the results of queries. Often an answer to a relational query contains a number of free variables, which are supposed to take arbitrary values. In the typed case these variables become typed, facilitating the understanding of the answers, especially those with multiple free variables. Next, a number of miniKanren applications require additional constraints to be implemented. In the untyped setting, when everything can be anything, some symbols or data structures tend to percolate into undesirable contexts [7]. In order to prevent this from happening, some auxiliary constraints (“absent<sup>o</sup>”, “symbol<sup>o</sup>”, etc.) were introduced. These constraints

---

<sup>1</sup><http://minikanren.org>

<sup>2</sup>An in-depth comparison of miniKanren and Prolog can be found here: <http://minikanren.org/minikanren-and-prolog.html>

<sup>3</sup><http://ocaml.org>

play a role of a weak dynamic type system, cutting undesirable answers out at runtime. Conversely, in a typed language this work can be entrusted to the type checker (at the price of enforcing an end user to write properly typed specifications), not only improving the performance of the system but also reducing the number of constraints which have to be implemented. Finally, it is rather natural to expect better performance of a statically-typed implementation.

We present an implementation of a set of relational combinators and syntax extensions for OCaml<sup>4</sup>, which, technically speaking, corresponds to  $\mu$ Kanren [14] with disequality constraints [1]. The contribution of our work is as follows:

1. Our embedding allows an end user to enjoy strong static typing and type inference in relational specifications; in particular, all type errors are reported at compile time and the types for all logical variables are inferred from the context.
2. Our implementation is based on the *polymorphic unification*, which, like the polymorphic comparison, can be used for arbitrary types. The implementation of polymorphic unification uses unsafe features and relies on the intrinsic knowledge of the runtime representation of values; we show, however, that this does not compromise type safety.
3. We describe a uniform and scalable pattern for using types for relational programming, which helps in converting typed data to and from the relational domain. With this pattern, only one generic feature (“Functor”) is needed, and thus virtually any generic framework for OCaml can be used. Although being rather a pragmatic observation, this pattern, we believe, would lead to a more regular and easy way to maintain relational specifications.
4. We provide a simplified way to integrate relational and functional code. Our approach utilizes a well-known pattern [9, 10] for variadic function implementation and makes it possible to hide the reification of the answers phase from an end user.

The rest of the paper is organized as follows: in the next section we provide a short overview of the related works. Then we briefly introduce miniKanren in its original form to establish some notions; we do not intend to describe the language in its full bloom (interested readers can refer to [11]). In Section 4 we describe some basic constructs behind a miniKanren implementation, this time in OCaml. In Section 5 we discuss polymorphic unification, and show that unification with triangular substitution respects typing. Then we present our approach to handle user-defined types by injecting them into the logic domain, and describe a convenient generic programming pattern, which can be used to implement the conversions from/to logic domain. We also consider a simple approach and a more elaborate and efficient tagless variant (see Section 6). Section 7 describes top-level primitives and addresses the problem of relational and functional code integration. Then, in Section 8 we present a set of relational examples, written with the aid of our library. Section 9 contains the results of a performance evaluation and a comparison of our implementations with existing implementation for Scheme. The final section concludes.

The authors would like to express a deep gratitude to the anonymous reviewers for their numerous constructive comments, Michael Ballantyne, Greg Rosenblatt, and the other attendees of the miniKanren Google Hangouts, who helped the authors in understanding the subtleties of the original miniKanren implementation, Daniel Friedman for his remarks on the preliminary version of the paper, and William Byrd for all his help and support, which cannot be overestimated.

---

<sup>4</sup>The source code of our implementation is accessible from <https://github.com/dboulytchev/OCanren>.

## 2 Related Works

There is a predictable difficulty in implementing miniKanren for a strongly typed language. Designed in the metaprogramming-friendly and dynamically typed realm of Scheme/Racket, the original miniKanren implementation pays very little attention to what has a significant importance in (specifically) ML or Haskell. In particular, one of the capstone constructs of miniKanren — unification — has to work for different data structures, which may have types different beyond parametricity.

There are a few ways to overcome this problem. The first one is simply to follow the untyped paradigm and provide unification for some concrete type, rich enough to represent any reasonable data structures. Some Haskell miniKanren libraries<sup>5</sup> as well as the previous OCaml implementation<sup>6</sup> take this approach. As a result, the original implementation can be retold with all its elegance; the relational specifications, however, become weakly typed. A similar approach was taken in early works on embedding Prolog into Haskell [22].

Another approach is to utilize *ad hoc* polymorphism and provide a type-specific unification for each “interesting” type. Some miniKanren implementations, such as Molog<sup>7</sup> and MiniKanrenT<sup>8</sup>, both for Haskell, can be mentioned as examples. While preserving strong typing, this approach requires a lot of “boilerplate” code to be written, so some automation — for example, using Template Haskell [20] — is desirable. In [18] a separate type class was introduced to both perform the unification and detect free logical variables in end-user data structures. The requirement for end user to provide a way to represent logical variables in custom data structures looks superfluous for us since these logical variables would require proper handling in the rest of the code outside the logical programming subsystem.

There is, actually, another potential approach, but we do not know if anybody has tried it: implementing unification for a generic representation of types as sum-of-products and fixpoints of functors [8, 23]. Thus, unification would work for any type for which a representation is provided. We believe that implementing this representation would require less boilerplate code to be written.

As follows from this exposition, a typed embedding of miniKanren in OCaml can be done with a combination of datatype-generic programming [12] and *ad hoc* polymorphism. There are a number of generic frameworks for OCaml (for example, [25]). On the other hand, the support for *ad hoc* polymorphism in OCaml is weak; there is nothing comparable in power to Haskell type classes, and even though sometimes the object-oriented layer of the language can be used to mimic desirable behavior, the result, as a rule, is far from satisfactory. Existing proposals for *ad hoc* polymorphism (for example, modular implicits [24]) require patching the compiler, which we want to avoid. Therefore, we take a different approach, implementing polymorphic unification once and for all logical types — a purely *ad hoc* approach, since the features which would provide a less *ad hoc* solution are not yet well integrated into the language. To deal with user-defined types in the relational subsystem, we propose to use their logical representations (see Section 6), which free an end user from the burden of maintaining logical variables, and we use generic programming to build conversions from and to logical representations in a systematic manner.

---

<sup>5</sup><https://github.com/JaimieMurdock/HK>, <https://github.com/rntz/ukanren>

<sup>6</sup><https://github.com/lightyang/minikanren-ocaml>

<sup>7</sup><https://github.com/acfoltzer/Molog>

<sup>8</sup><https://github.com/jvranish/MiniKanrenT>

### 3 miniKanren — a Short Presentation

In this section we briefly describe miniKanren in its original form, using a canonical example. miniKanren is organized as a set of combinators and macros for Scheme/Racket, designed to describe a search for the solution of a certain *goal*. There are four domain-specific constructs to build *goals*:

- Syntactic unification [2] in the form  $(\equiv t_1 t_2)$ , where  $t_1, t_2$  are some *terms*; unification establishes a syntactic basis for all other goals. If there is a unifier for two given terms, the goal is considered satisfied, a most general unifier is kept as a partial solution, and the execution of current branch continues. Otherwise, the current branch backtracks.
- Disequality constraint [1] in the form  $(\neq t_1 t_2)$ , where  $t_1, t_2$  are some terms; a disequality constraint prevents all branches (starting from the current), in which the specified terms are equal (w.r.t. the search state), from being considered.
- Conditional construct in the form

```
(conde
  [g11 g21 ... gk11]
  [g12 g22 ... gk22]
  ...
  [g1n g2n ... gknn]
)
```

where each  $g_j^i$  is a goal. A conde goal considers each collection of subgoals, surrounded by square brackets, as implicit conjunction (so  $[g_1^i g_2^i \dots g_{k_i}^i]$  is considered as a conjunction of all  $g_j^i$ ) and tries to satisfy each of them independently — in other words, operates on them as a disjunction.

- Fresh variable introduction construct in the form

```
(fresh (x1 x2 ... xk)
  g1
  g2
  ...
  gn
)
```

where each  $g_i$  is a goal. This form introduces fresh variables  $x_1 x_2 \dots x_k$  and tries to satisfy the conjunction of all subgoals  $g_1 g_2 \dots g_n$  (these subgoals may contain introduced fresh variables).

As an example consider a list concatenation relation; by a well-established tradition, the names of relational objects are superscripted by “*o*”, hence `appendo`:

```
1 (define (appendo x y xy)
2   (conde
3     [( $\equiv$  '() x) ( $\equiv$  y xy)]
4     [(fresh (h t ty)
5       ( $\equiv$  '(,h . ,t) x)
6       ( $\equiv$  '(,h . ,ty) xy)
7       (appendo t y ty))]))
```

We interpret the relation “`appendo x y xy`” as “the concatenation of `x` and `y` equals `xy`”. Indeed, if the list `x` is empty, then (regardless the content of `y`) in order for the relation to hold the value for `xy` should be equal to that of `y` — hence line 3. Otherwise, `x` can be decomposed into the head `h` and the tail `t` — so we need some fresh variables. We also need the additional variable `ty` to designate the list that is in the relation `appendo` with `t` and `y`. Trivial relational reasonings complete the implementation (lines 5-7).

A goal, built with the aid of the aforementioned constructs, can be run by the following primitive:

```
run n (q1 ... qk) G
```

Here `n` is the number of requested answers (or “\*” for all answers), `qi` are fresh query variables, and `G` is a goal, which can contain these variables.

The `run` construct initiates the search for answers for a given goal and returns a (finite or infinite) list of answers — the bindings for query variables, which represent individual solutions for that query. For example,

```
run 1 (q) (appendo q '(3 4) '(1 2 3 4) )
```

returns a list `((1 2))`, which constitutes the answer for a query variable `q`. The process of constructing the answers from internal data structures of miniKanren interpreter is called *reification* [5].

## 4 Streams, States, and Goals

This section describes a top-level framework for our implementation. Even though it contains nothing more than a reiteration of the original implementation [14, 1] in OCaml, we still need some notions to be properly established.

The search itself is implemented using a backtracking lazy stream monad [17]:

```
type α stream
```

```
val mplus : α stream → α stream → α stream
```

```
val bind : α stream → (α → β stream) → β stream
```

Monadic primitives describe the shape of the search, and their implementations may vary in concrete miniKanren versions.

An essential component of the implementation is a bundle of the following types:

```
type env = ...
```

```
type subst = ...
```

```
type constraints = ...
```

```
type state = env * subst * constraints
```

Type `state` describes a point in a lazily constructed search tree: type `env` corresponds to an *environment*, which contains some supplementary information (in particular, an environment is needed to correctly allocate fresh variables), type `subst` describes a substitution, which keeps current bindings for some logical variables, and, finally, type `constraints` represents disequality constraints, which have to be respected. In the simplest case `env` is just a counter for the number of the next free variable, `subst` is a map-like structure and `constraints` is a list of substitutions.

The next cornerstone element is the *goal* type, which is considered as a transformer of a state into a lazy stream of states:

```
type goal = state → state stream
```

In terms of the search, a goal nondeterministically performs one step of the search: for a given node in a search tree it produces its immediate descendants. On the user level the type goal is abstract, and states are completely hidden.

Next to last, there are a number of predefined combinators:

```
val (&&&)      : goal → goal → goal
val (|||)      : goal → goal → goal
val call_fresh : (v → goal) → goal
...

```

Conjunction “&&&” combines the results of its argument goals using bind, disjunction “|||” concatenates the results using mplus, abstraction primitive call\_fresh takes an abstracted goal and applies it to a freshly created variable. Type  $v$  in the last case designates the type for a fresh variable, which we leave abstract for now. These combinators serve as the bricks for the implementation of conventional miniKanren constructs and syntax extensions (conde, fresh, etc.)

Finally, there are two primitive goal constructors:

```
val (≡) : t → t → goal
val (≠) : t → t → goal
```

The first one is a unification, while the other is a disequality constraint. Here, we again left the type of terms  $t$  abstract; it will be instantiated later.

In the implementation of miniKanren both of these goals are implemented using unification [1]; this is true for us as well. However, due to a drastic difference between the host languages, the implementation of efficient polymorphic unification itself leads to a number of tricks with typing and data representation, which are absent in the original version.

In this setting, the run primitive is represented by the following function:

```
val run : goal → state stream
```

This function creates an initial state and applies a goal to it. The states in the return stream describe various solutions for the goal. As the stream is constructed lazily, taking elements one by one makes the search progress.

To discover concrete answers, the state has to be queried for its contents. As a rule, a few variables are reified in a state, i.e. their bindings in the corresponding substitution are retrieved. Disequality constraints for free variables have to be reified additionally (e.g. represented as a list of “forbidden” terms). As forbidden terms can contain free variables, the constraint reification process is recursive.

In our case, the reification is a subtle part, since, as we will see shortly, it can not be implemented in a type-safe fragment of the language.

## 5 Polymorphic Unification

We consider it rather natural to employ polymorphic unification in a language already equipped with polymorphic comparison — a convenient, but somewhat disputable<sup>9</sup> feature. Like polymorphic comparison, polymorphic unification performs a traversal of values, exploiting intrinsic knowledge of their runtime representation. The undeniable benefits of this solution are, first, that in order to perform

<sup>9</sup>See, for example, <https://blogs.janestreet.com/the-perils-of-polymorphic-compare>

unification for user types no “boilerplate” code is needed, and, second, that this approach seems to deliver the most efficient implementation. On the other hand, all the pitfalls of polymorphic comparison are inherited as well; in particular, polymorphic unification loops for cyclic data structures and does not work for functional values. Since we generally do not expect any reasonable outcome in these cases, the only remaining problem is that the compiler is incapable of providing any assistance in identifying and avoiding these cases. Another drawback is that the implementation of polymorphic unification relies on the runtime representation of values and has to be fixed every time the representation changes. Finally, as it is written in an unsafe manner using the `Obj` interface, it has to be carefully developed and tested.

An important difference between polymorphic comparison and unification is that the former only inspects its operands, while the results of unification are recorded in a substitution (a mapping from logical variables to terms), which later is used to reify answers. So, generally speaking, we have to show that no ill-typed terms are constructed as a result. Overall, this property seems to be maintained vacuously, since the very nature of (syntactic) unification is to detect whether some things can be considered equal. Nevertheless there are different type systems and different unification implementations; in addition *equal things* can be *differently typed*, so we provide here a correctness justification for a well-defined abstract case, and will reuse this conclusion for various concrete cases.

First, we consider three alphabets:

$$\begin{array}{ll} \tau, \dots & - \text{ types} \\ x^\tau, \dots & - \text{ typed logic variables} \\ C_k^{\tau_1 \times \tau_2 \times \dots \times \tau_k \rightarrow \tau} (k \geq 0), \dots & - \text{ typed constructors} \end{array}$$

The set of all well-formed typed terms is defined by mutual induction for all types:

$$t^\tau = x^\tau \mid C_k^{\tau_1 \times \tau_2 \times \dots \times \tau_k \rightarrow \tau}(t^{\tau_1}, t^{\tau_2}, \dots, t^{\tau_k})$$

For simplicity from now on we abbreviate the notation  $C_k^{\tau_1 \times \tau_2 \times \dots \times \tau_k \rightarrow \tau}(t^{\tau_1}, t^{\tau_2}, \dots, t^{\tau_k})$  into  $C_k^\tau(t^{\tau_1}, t^{\tau_2}, \dots, t^{\tau_k})$ , keeping in mind that for any concrete constructor and for all its occurrences in arbitrary terms all its subterms in corresponding positions agree in types.

In this formulation we do not consider any structure over the set of types besides type equality, and we assume all terms are explicitly attributed to their types at runtime. We employ this property to implement a unification algorithm in regular OCaml, using some representation for terms and types:

```
val unify : term → term → subst option → subst option
```

where “term” stands for the type representing typed terms, and “subst” stands for the type of substitution (a partial mapping from logic variables to terms). Unification can fail (hence “option” in the result type), is performed in the context of existing substitution (hence “subst” in the third argument) and can be chained (hence “option” in the third argument).

We use exactly the same unification algorithm with triangular substitution as in the reference implementation [14]. We omit here some not-so-important details (like “occurs check”), which are kept in the actual implementation, and refrain from discussing the nature and properties of the algorithm itself (an excellent description, including a certified correctness proof, can be found in [19]).

The following snippet presents the implementation:

```
1 let rec unify t1^τ t2^τ subst =
2   let rec walk s t^τ =
3     match t^τ with
4     | x^τ when x^τ ∈ dom(s) → walk s (s x^τ)
```

```

5     | _ → tτ
6   in
7   match subst with
8     | None → None
9     | Some s →
10      match walk s t1τ, walk s t2τ with
11      | x1τ, x2τ when x1τ = x2τ → subst
12      | x1τ, q2τ → Some (s [x1τ ← q2τ])
13      | q1τ, x2τ → Some (s [x2τ ← q1τ])
14      | Cτ(t1τ1}, ..., tkτk}, Cτ(p1τ1}, ..., pkτk}) →
15          unify tkτk} pkτk} (... (unify t1τ1} p1τ1} subst)..)
16      | -, - → None

```

We remind the reader that all superscripts correspond to type attributes, which we consider here as parts of values being manipulated. For example, line 1 means that we apply unify to terms  $t_1$  and  $t_2$ , and expect their types to be equal  $\tau$ . We assume that at the top level unification is always applied to some terms of the same type and that any substitution can only be acquired from the empty one by a sequence of unifications.

We are going to show that under these assumptions all type attributes are superfluous — they do not affect the execution of unify and can be removed. Note that the only place where we were incapable of providing an explicit type attribute was in line 4, where the result of substitution application was returned. However, we can prove by induction that any substitution respects the following property: if a substitution  $s$  is defined for a variable  $x^\tau$ , then  $s x^\tau$  is attributed with the type  $\tau$  (and, consequently,  $\text{walk } s t^\tau$  always returns a term of type  $\tau$ ).

Indeed, this property vacuously holds for the empty substitution. Let  $s$  be some substitution, for which the property holds. In line 11 we return an unchanged substitution; in line 10 we perform two calls —  $\text{walk } s t_1^\tau$  and  $\text{walk } s t_2^\tau$  and match their results. However, by our induction hypothesis these results are again attributed to the type  $\tau$ , which justifies the pattern matching. In line 11 we return the substitution unchanged, in lines 12 and 13 we extend the existing substitution but preserve the property of interest. Finally, in line 15 we chain a few applications of unify; note that, again, all these calls are performed for terms of equal types, starting from a substitution possessing the property of interest. A simple induction on the chain length completes the proof.

So, type attributes are inessential — they are never analyzed and never restrict pattern matching; hence, they can be erased completely. We can notice now that for the representation of terms we can use OCaml’s native runtime representation. It can not be done, however, using regular OCaml — we have to utilize the low-level, unsafe interface Obj. Note also, we need some way to identify the occurrences of logical variables inside the terms (in the original miniKanren implementation the ambiguity between variable and non-variable datum representation is resolved by a convention — a luxury we cannot afford). We postpone the discussion on this subject until the next section.

We call our implementation *polymorphic*, since at the top level it is defined as

```
val unify :  $\alpha \rightarrow \alpha \rightarrow \text{subst option} \rightarrow \text{subst option}$ 
```

The type of substitution is not polymorphic, which means that the compiler completely loses the track of types of values stored in a substitution. These types are recovered later during the reification-of-answers phase (see Section 7). Outside the unification the compiler maintains typing, which means that all terms, subterms, and variables agree in their types in all contexts.



## 6 Term Representation and Injection

Polymorphic unification, considered in the previous section, works for the values of any type under the assumption that we are capable of identifying logical variables. The latter depends on the term representation. In the original implementation all terms are represented as a conventional S-expressions, while logical variables (in a simplest case) — as one-element vectors; it’s an end user responsibility to respect this convention and refrain from operating with vectors as a user data.

In our case we want to preserve both strong typing and type inference. Since we have chosen to use polymorphic unification, it is undesirable to represent logical variables of different types differently (while technically possible, it would compromise the lightweight approach we used so far). This means that terms with logical variables have to be typed differently from user-defined data — otherwise it would be possible to use terms in contexts where logical variables are not handled properly. At the same time we do not want term types to be completely different from user-defined types — for example, we would like to reuse user-defined constructors, etc. This considerations boil down to the idea of *logical representation* for a user-defined type. Informally, a logical representation for the type  $t$  is a type  $\rho_t$  with a couple of conversion functions:

$$\begin{aligned} \uparrow : t &\rightarrow \rho_t & - & \text{injection} \\ \downarrow : \rho_t &\rightarrow t & - & \text{projection} \end{aligned}$$

The type  $\rho_t$  repeats the structure of  $t$ , but can contain logic variables. So, the injection is total, while the projection is partial.

It is important to design representations as instances of some generic scheme (otherwise, miniKanren combinators could not be properly typed). In addition it is desirable to provide a generic way to build injection/projection pair in a uniform manner (and, even better, automatically) to lift the burden of their implementation off the end user shoulders and improve the reliability of the solution. Finally, the representation must provide a way to detect logic variable occurrences.

In this part we consider two approaches to implementing logical representations. The first is rather easy to develop and implement; unfortunately, the implementation demonstrates a poor performance for a number of important applications. In order to fix this deficiency, we develop a more elaborate technique which nevertheless reuses some components from the previous one. In Section 9 we present the results of performance evaluation for both approaches.

### 6.1 Tagged Logical Values

The first natural solution is to use tagging for representing logical representations. We introduce the following polymorphic type  $[\alpha]$ <sup>10</sup>, which corresponds to a logical representation of the type  $\alpha$ :

`type`  $[\alpha] = \text{Var of int} \mid \text{Value of } \alpha$

Informally speaking, any value of type  $[\alpha]$  is either a value of type  $\alpha$ , or a free logic variable. Note, the constructors of this type cannot be disclosed to an end user, since the only possible way to create a logic variable should still be by using the “`fresh`” construct; thus the logic type is abstract in the interface. Now, we may redefine the signature of abstraction, unification and disequality primitives in the following manner

`val` `call_fresh` :  $([\alpha] \rightarrow \text{goal}) \rightarrow \text{goal}$

---

<sup>10</sup>In concrete syntax called “ $\alpha$  logic”

```

val (  $\equiv$  )      :  $[\alpha] \rightarrow [\alpha] \rightarrow \text{goal}$ 
val (  $\neq$  )      :  $[\alpha] \rightarrow [\alpha] \rightarrow \text{goal}$ 

```

Both unification and disequality constraint would still use the same polymorphic unification; their external, visible type, however, is restricted to logical types only.

Apart from variables, other logical values can be obtained by injection; conversely, sometimes a logical value can be projected to a regular one. We supply two basic functions<sup>11</sup> for these purposes

```

val ( $\uparrow_{\forall}$ ) :  $\alpha \rightarrow [\alpha]$ 
val ( $\downarrow_{\forall}$ ) :  $[\alpha] \rightarrow \alpha$ 

```

```

let ( $\uparrow_{\forall}$ ) x = Value x
let ( $\downarrow_{\forall}$ ) = function Value x  $\rightarrow$  x | _  $\rightarrow$  failwith “not a value”

```

which can be used to perform a *shallow* injection/projection. As expected, the injection is total, while the projection is partial.

The shallow pair works well for primitive types; to implement injection/projection for arbitrary types we exploit the idea of representing regular types as fixed points of functors [23]. For our purposes it is desirable to make the functors fully polymorphic — thus a type, in which we can place a logical variable into arbitrary position, can be easily manufactured. In addition this approach makes it possible to refactor the existing code to use relational programming with only minor changes.

To illustrate this approach, we consider an iconic example — the list type. Let us have a conventional definition for a regular polymorphic list in OCaml:

```

type  $\alpha$  list = Nil | Cons of  $\alpha * \alpha$  list

```

For this type we can only place a logical variable in the position of a list element, but not of the tail, since the tail always has the type  $\alpha$  list, fixed in the definition of constructor Cons. In order to create a full-fledged logical representation, we first have to abstract the type into a fully-polymorphic functor:

```

type ( $\alpha, \beta$ )  $\mathcal{L}$  = Nil | Cons of  $\alpha * \beta$ 

```

Now, the original type can be expressed as

```

type  $\alpha$  list = ( $\alpha, \alpha$  list)  $\mathcal{L}$ 

```

and its logical representation — as

```

type  $\alpha$  listo =  $[[\alpha], \alpha$  listo)  $\mathcal{L}$ 

```

Moreover, with the aid of conventional functor-specific mapping function

```

val fmap $\mathcal{L}$  : ( $\alpha \rightarrow \alpha'$ )  $\rightarrow$  ( $\beta \rightarrow \beta'$ )  $\rightarrow$  ( $\alpha, \beta$ )  $\mathcal{L} \rightarrow$  ( $\alpha', \beta'$ )  $\mathcal{L}$ 

```

both the injection and the projection functions can be implemented:

```

let rec  $\uparrow_{\text{list}}$  l =  $\uparrow_{\forall}$ (fmap $\mathcal{L}$  ( $\uparrow_{\forall}$ )  $\uparrow_{\text{list}}$  l)
let rec  $\downarrow_{\text{list}}$  l = fmap $\mathcal{L}$  ( $\downarrow_{\forall}$ )  $\downarrow_{\text{list}}$  ( $\downarrow_{\forall}$  l)

```

As functor-specific mapping functions can be easily written or, better, derived automatically using a number of existing frameworks for generic programming for OCaml, one can easily provide injection/projection pair for user-defined data types.

---

<sup>11</sup>In concrete syntax called “inj” and “prj”.

We now can address the problem of variable identification during polymorphic unification. As we do not know the types, we cannot discriminate logical variables by their tags only and, thus, cannot simply use pattern matching. In our implementation we perform a variable test as follows:

- in an environment, we additionally keep some unique boxed value — the *anchor* — created by run at the moment of initial state generation; the anchor is inherited unchanged in all derived environments during the search session;
- we change the logic type definition into

`type [ $\alpha$ ] = Var of int * anchor | Value of  $\alpha$`

making it possible to save in each variable the anchor, inherited from the environment, in which the variable was created;

- inside the unification, in order to check if we are dealing with a variable, we test the conjunction of the following properties:
  1. the scrutinee is boxed;
  2. the scrutinee's tag and layout correspond to those for variables (i.e. the values, created with the constructor `Var` of type [ $\alpha$ ]);
  3. the scrutinee's anchor and the current environment's anchor have equal addresses.

Taking into account that the state type is abstract at the user level, we guarantee that only those variables which were created during the current run session would pass the test, since the pointer to the anchor is unique among all pointers to a boxed value and could not be disclosed anywhere but in the variable-creation primitive.

The only thing to describe now is the implementation of the reification stage. The reification is represented by the following function:

`val reify : state → [ $\alpha$ ] → [ $\alpha$ ]`

This function takes a state and a logic value and recursively substitutes all logic variables in that value w.r.t. the substitution in the state until no occurrences of bound variables are left. Since in our implementation the type of a substitution is not polymorphic, `reify` is also implemented in an unsafe manner. However, it is easy to see that `reify` does not produce ill-typed terms. Indeed, all original types of variables are preserved in a substitution; unification does not change unified terms, so all terms bound in a substitution are well-typed. Hence, `reify` always substitutes some subterms in a well-typed term with other terms of the corresponding types, which preserves the well-typedness.

In addition to performing substitutions, `reify` also reifies disequality constraints. Constraint reification attaches to each free variable in a reified term a list of reified terms, describing the disequality constraints for that free variable. Note, disequality can be established only for equally-typed terms, which justifies the type-safety of reification. Note also, additional care has to be taken to avoid infinite looping, since reification of answers and constraints are mutually recursive, and the reification of a variable can be potentially invoked from itself due to a chain of disequality constraints. In the following example

```
let foo q =
  fresh (r s)
    (q ≡ ↑v (Some r)) &&&
    (r ≠ s) &&&
    (s ≠ r)
```

the answer for the variable  $q$  will contain a disequality constraint for the variable  $r$ ; the reification of  $r$  will in turn lead to the reification of its own constraint, this time the variable  $s$ ; finally, the reification of  $s$  will again invoke the reification of  $r$ , etc.

After the reification, the content of a logical value can be inspected via the following function:

```
val destruct : [ $\alpha$ ]  $\rightarrow$  ['Var of int * [ $\alpha$ ] list | 'Value of  $\alpha$ ]
```

Constructor `'Var` corresponds to a free variable with unique integer identifier and a list of terms, representing all disequality constraints for this variable.

## 6.2 Tagless Logical Values and Type Bookkeeping

The solution presented in the previous subsection suffers from the following deficiency: in order to perform unification, we inject terms into the logic domain, making them as twice as large. As a result, this implementation loses to the original one in terms of performance in many important applications, which compromises the very idea of using OCaml as a host language.

Here we develop an advanced version, which eliminates this penalty. As a first step, let's try to eliminate the tagging with a drastic measure:

```
type [ $\alpha$ ] =  $\alpha$ 
```

What consequences would this have? Of course, we would not be able to create logical variables in a conventional way. However, we still could have a separate type of variables

```
type var = Var of int * anchor
```

and use *the same* variable test procedure. As the type  $[\alpha]$  is abstract, this modification does not change the interface. As we reuse the variable test, polymorphic unification can continue to work *almost* correctly. The problem is that now it can introduce the occurrences of free logic variables in non-logical, tagless, data structures. These free logic variables do not get in the way of unification itself (since it can handle them properly, thanks to the variable test), but they can not be disclosed to the outer world as is.

Our idea is to use this generally unsound representation for all internal actions, and perform tagging only during the reification stage. However, this scenario raises the following question: what would the type of `reify` be? It can not be simply

```
val reify : state  $\rightarrow$  [ $\alpha$ ]  $\rightarrow$  [ $\alpha$ ]
```

anymore since  $[\alpha]$  now equals  $\alpha$ . We *want*, however, it be something like

```
val reify : state  $\rightarrow$  [ $\alpha$ ]  $\rightarrow$  (“tagged” [ $\alpha$ ])
```

If  $\alpha$  is not a parametric type, we can simply test if the value is a variable, and if yes, tag it with the constructor `Var`; we tag it with `Value` otherwise, and we're done. This trick, however, would not work for parametric types. Consider, for example, the reification of a value of type `[[int] list]`. The (hypothetical) approach being described would return a value of type (“tagged” `[[int] list]`), i.e. tagged only on the top level; we need to repeat the procedure recursively. In other words, we need the following (meta) type for the reification primitive:

```
val reify : state  $\rightarrow$  [ $\alpha$ ]  $\rightarrow$  (“tagged” $[\beta]$ )
```

where  $\beta$  is the result of tagging  $\alpha$ .

These considerations can be boiled down to the following concrete implementation.

First, we roll back to the initial definition of  $[\alpha]$  — it will play the role of our “tagged” type. We introduce a new, two-parameter type<sup>12</sup>

```
type { $\alpha$ ,  $\beta$ } =  $\alpha$ 
```

Of course, this type is kept abstract at the end-user level. Informally speaking, the type  $\{\alpha, \beta\}$  designates the injection of a tagless type  $\alpha$  into a tagged type  $\beta$ ; the value itself is kept in the tagless form, but the tagged type can be used during the reify stage as a constraint, which would allow us to reify a tagless representation only to a feasible tagged one. In other words, we record the injection steps using the second type parameter of the type “ $\{\},$ ”, performing the bookkeeping on the type level rather than on the value level.

We introduce the following primitives for the type  $\{\alpha, \beta\}$ :

```
val lift :  $\alpha \rightarrow \{\alpha, \alpha\}$   
val inj  :  $\{\alpha, \beta\} \rightarrow \{\alpha, [\beta]\}$ 
```

```
let lift x = x  
let inj  x = x
```

The function `lift` puts a value into the “bookkeeping injection” domain for the first time, while `inj` plays the role of the injection itself. Their composition is analogous to what was called “ $\uparrow_{\vee}$ ” in the previous implementation:

```
val  $\uparrow_{\vee}$  :  $\alpha \rightarrow \{\alpha, [\alpha]\}$   
let  $\uparrow_{\vee}$  x = inj (lift x)
```

In order to deal with parametric types, we can again utilize generic programming. To handle the types with one parameter, we introduce the following functor:

```
module FMap (T : sig type  $\alpha$  t val fmap : ( $\alpha \rightarrow \beta$ )  $\rightarrow$   $\alpha$  t  $\rightarrow$   $\beta$  t end) :  
  sig  
    val distrib :  $\{\alpha, \beta\}$  T.t  $\rightarrow$   $\{\alpha$  T.t,  $\beta$  T.t}  
  end =  
  struct  
    let distrib x = x  
  end
```

Note, that we do not use the function “`T.fmap`” in the implementation; however, first, we need an inhabitant of the corresponding type to make sure we are indeed dealing with a functor, and next, we actually will use it in the implementation of type-specific reification, see below.

In order to handle two-, three-, etc. parameter types we need higher-kinded polymorphism, which is not supported in a direct form in OCaml. So, unfortunately, we need to introduce separate functors for the types with two-, three- etc. parameters; existing works on higher-kinded polymorphism in OCaml [26] require the similar scaffolding to be erected as a bootstrap step.

Given the functor(s) of the described shape, we can implement logic representations for all type’s constructors. For example, for standard type  $\alpha$  `option` with two constructors `None` and `Some` the implementation looks like as follows:

```
module FOption = FMap (struct
```

---

<sup>12</sup>In concrete syntax called “ $(\alpha, \beta)$  injected”.

```

  type  $\alpha$  t =  $\alpha$  option
  let fmap = fmap_option
end)

val some :  $\{\alpha, \beta\} \rightarrow \{\alpha \text{ option}, \beta \text{ option}\}$ 
val none : unit  $\rightarrow \{\alpha \text{ option}, \beta \text{ option}\}$ 

let some x = inj (FOption.distrib (Some x))
let none () = inj (FOption.distrib None)

```

In other words, we can in a very systematic manner define logic representations for all constructors of types of interest. These representations can be used in the relational code, providing a well-bookkept typing — for each logical type we would be able to reconstruct its original, tagless preimage.

With the new implementation, the types of basic goal constructors have to be adjusted:

```

val ( $\equiv$ ) :  $\{\alpha, [\beta]\} \rightarrow \{\alpha, [\beta]\} \rightarrow \text{goal}$ 
val ( $\neq$ ) :  $\{\alpha, [\beta]\} \rightarrow \{\alpha, [\beta]\} \rightarrow \text{goal}$ 

```

As always, we require both arguments of unification and disequality constraint to be of the same type; in addition we require the injected part of the type to be logical.

During the reification stage the bindings for the top-level variables, reconstructed using the final substitution, have to be properly tagged. This process is implemented in a datatype-generic manner as well: first, we have reifiers for all primitive types:

```

val reify_int : helper  $\rightarrow \{\text{int}, [\text{int}]\} \rightarrow [\text{int}]$ 
val reify_string : helper  $\rightarrow \{\text{string}, [\text{string}]\} \rightarrow [\text{string}]$ 
...

```

and, then, we add the reifier to the output signature in all FMap-like functors:

```

val reify: (helper  $\rightarrow \{\alpha, \beta\} \rightarrow \beta$ )  $\rightarrow$  helper  $\rightarrow \{\alpha \text{ T.t}, [\beta \text{ T.t}] \text{ as } \gamma\} \rightarrow \gamma$ 

```

Note, since now `reify` is a type-specific and, hence, constructed at the user-level, we refrain from passing it a state (which is inaccessible on the user level). Instead, we wrap all state-specific functionality in an abstract `helper` data type, which encapsulates all state-dependent functionality needed for `reify` to work properly.

## 7 Reification and Top-Level Primitives

In Section 4 we presented a top-level function `run`, which runs a goal and returns a stream of states. To acquire answers to the query, represented by that goal, its free variables have to be reified in these states, and we described the reification primitives in Section 6. However, the states keep answers in an untyped form, and the types of answers are recovered solely on the basis of the types of variables being reified. So, the type safety of the reification critically depends on the requirement to reify each variable only in those states, which are descendants (w.r.t. the search tree) of the state, in which that variable was created. In this section we describe a set of top-level primitives, which enforce this requirement.

We provide a set of top-level combinators, which should be used to surround relational code and perform reification in a transparent manner only in correct states. We reimplement the top-level primitive `run` to take three arguments. The exact type of `run` is rather complex and non-instructive, so we prefer to describe the typical form of its application:

$$\text{run } \bar{n} \ (\underline{\text{fun}} \ l_1 \dots l_n \rightarrow G) \ (\underline{\text{fun}} \ a_1 \dots a_n \rightarrow H)$$

Here  $\bar{n}$  stands for a *numeral*, which describes the number of parameters for two other arguments of `run`,  $l_1 \dots l_n$  — free logical variables,  $G$  — a goal (which can make use of  $l_1 \dots l_n$ ),  $a_1 \dots a_n$  — reified answers for  $l_1 \dots l_n$ , respectively, and, finally,  $H$  — a *handler* (which can make use of  $a_1 \dots a_n$ ).

The types of  $l_1 \dots l_n$  are inferred from  $G$  and always have a form

$$\{\alpha, [\beta]\}$$

since the types of variables can be constrained only in unification or disequality constraints.

The types of  $a_1 \dots a_n$  are inferred from the types of  $l_1 \dots l_n$  and have the form

$$(\alpha, \beta) \text{ reified stream}$$

where the type `reified`, in turn, is

$$\text{type } (\alpha, \beta) \text{ reified} = \langle \text{prj} : \alpha; \text{reify} : (\text{helper} \rightarrow \{\alpha, \beta\} \rightarrow \beta) \rightarrow \beta \rangle$$

Two methods of this type can be used to perform two different styles of reification: first, a value without free variables can be returned as is (using the method `prj` which checks that in the value of interest no free variables occur, and raises an exception otherwise). If the value contains some free variables, it has to be properly injected into the logic domain — this is what `reify` stands for. It takes as an argument a type-specific tagging function, constructed using generic primitives described in the previous section.

In other words a user-defined handler takes streams of reified answers for all variables supplied to the top-level goal. All streams  $a_i$  contain coherent elements, so they all have the same length and  $n$ -th elements of all streams correspond to the  $n$ -th answer, produced by the goal  $G$ .

There are a few predefined numerals for one, two, etc. arguments (called, traditionally, `q`, `qr`, `qrs` etc.), and a successor function, which can be applied to existing numeral to increment the number of expected arguments. The implementation technique generally follows [9, 10].

Thus, the search and reification are tightly coupled; it is simply impossible to perform the reification for arbitrarily-taken state and variable. This solution both guarantees the type safety and frees an end user from the necessity to call reification primitives manually.

## 8 Examples

In this section we present some examples of a relational specification, written with the aid of our library. Besides `miniKanren` combinators themselves, our implementation contains two syntax extensions — one for `fresh` construct and another for *inverse- $\eta$ -delay* [14], which is sometimes necessary to delay recursive calls in order to prevent infinite looping. In addition, we included a small relational library of data structures like lists, numbers, booleans, etc. This library is written completely on the user level using techniques described in Section 6 with no utilization of any unsafe features. The examples given below illustrate the usage of all these elements as well.

### 8.1 List Concatenation and Reversing

List concatenation and reversing are usually the first relational programs considered, and we do not wish to deviate from this tradition. We've already considered the implementation of `appendo` in original `miniKanren` in Section 3. In our case, the implementation looks familiar:

```

let rec appendo x y xy =
  (x ≡ nil ()) &&& (y ≡ xy) |||
  (fresh (h t)
    (x ≡ h % t)
    (fresh (ty)
      (h % ty ≡ xy)
      (appendo t y ty)
    )
  )
)

let rec reverso a b =
  conde [
    (a ≡ nil ()) &&& (b ≡ nil ());
    (fresh (h t)
      (a ≡ h % t)
      (fresh (a')
        (appendo a' !< h b)
        (reverso t a')
      )
    )
  ]

```

Here we make use of our implementation of relational lists, which provides convenient shortcuts for standard functional primitives:

- “`nil ()`” corresponds to “`[]`”;
- “`h % t`” corresponds to “`h :: t`”;
- “`a %< (b %< (c !< d))`” corresponds to “`[a; b; c; d]`”.

In our implementation the basic miniKanren primitive “`conde`” is implemented as a disjunction of a list of goals, not as a built-in syntax construct. We also make use of explicit conjunction and disjunction infix operators instead of nested bracketed structures which, we believe, would look too foreign here.

## 8.2 Relational Sorting and Permutations

For the next example we take list sorting; specifically, we present a sorting for lists of natural numbers in Peano form since our library already contains built-in support for them. However, our example can be easily extended for arbitrary (but linearly ordered) types.

List sorting can be implemented in miniKanren in a variety of ways — virtually any existing algorithm can be rewritten relationally. We, however, try to be as declarative as possible to demonstrate the advantages of the relational approach. From this standpoint, we can claim that the sorted version of an empty list is an empty list, and the sorted version of a non-empty list is its smallest element, concatenated with a sorted version of the list containing all its remaining elements.

The following snippet literally implements this definition:

```

let rec sorto x y =
  conde [

```



```

(x ≡ nil ()) &&& (y ≡ nil ());
fresh (s xs xs')
  (y ≡ s % xs')
  (sorto xs xs')
  (smallesto x s xs)
]

```

The meaning of the expression “smallest<sup>o</sup> x s xs” is “s is the smallest element of a (non-empty) list x, and xs is the list of all its remaining elements”. Now, smallest<sup>o</sup> can be implemented using a case analysis (note, “l” here is a non-empty list):

```

let rec smallesto l s l' =
  conde [
    (l ≡ s % nil ()) &&& (l' ≡ nil ());
    fresh (h t s' t' max)
      (l' ≡ max % t')
      (l ≡ h % t)
      (minmaxo h s' s max)
      (smallesto t s' t')
  ]

```

Finally, we implement a relational minimum-maximum calculation primitive:

```

let minmaxo a b min max =
  conde [
    (min ≡ a) &&& (max ≡ b) &&& (leo a b);
    (max ≡ a) &&& (min ≡ b) &&& (gto a b)
  ]

```

Here “le<sup>o</sup>” and “gt<sup>o</sup>” are built-in comparison goals for natural numbers in Peano form. Having relational sort<sup>o</sup>, we can implement sorting for regular integer lists:

```

let sort l =
  run q (sorto (inj_nat_list l))
    (fun qs → from_nat_list ((Stream.hd qs)#prj) )

```

Here Stream.hd is a function which takes a head from a lazy stream of answers, inj\_nat\_list — an injection from regular integer lists into logical lists of logical Peano numbers, from\_nat\_list — a projection from lists of Peano numbers to lists of integers.

It is interesting, that since sort<sup>o</sup> is relational, it can be used to calculate a list of all *permutations* for a given list. Indeed, sorting each permutation results in the same list. So, the problem of finding all permutations can be relationally reformulated into the problem of finding all lists which are converted by sorting into the given one:

```

let perm l = map (fun a → from_nat_list a#prj)
  (run q (fun q → fresh (r)
    (sorto (inj_nat_list l) r)
    (sorto q r)
  )
  (Stream.take ~n:(fact (length l))))

```

Note, for sorting the original list we used exactly the same primitive. Note also, we requested exactly fact @@ length 1 answers; requesting more would result in an infinite search for non-existing answers.

### 8.3 Type Inference for STLC

Our final example is a type inference for Simply Typed Lambda Calculus [3]. The problem and solution themselves are rather textbook examples again [11, 5]; however, with this example we show once again the utilization of generic programming techniques we described in Section 6. As a supplementary generic programming library here we used object-oriented generic transformers<sup>13</sup>; we presume, however, that any other framework could equally be used.

We first describe the type of lambda terms and their logic representation:

```

module Term = struct
  module T = struct
    @type ('varname, 'self) t =
      | V   of 'varname
      | App of 'self * 'self
      | Abs of 'varname * 'self
    with gmap

    let fmap f g x = gmap(t) f g x
  end

  include T
  include FMap2(T)

  let v   s = inj (distrib (V s))
  let app x y = inj (distrib (App (x, y)))
  let abs x y = inj (distrib (Abs (x, y)))
end

```

Now we have to repeat the work for the type of simple types:

```

module Type = struct
  module T = struct
    @type ('a, 'b) t =
      | P   of 'a
      | Arr of 'b * 'b
    with gmap

    let fmap f g x = gmap(t) f g x
  end

  include T
  include FMap2(T)

```

---

<sup>13</sup><https://github.com/dboulytchev/GT>

```

  let p s = inj (distrib (P s))
  let arr x y = inj (distrib (Arr (x, y)))
end

```

Note, the “relational” part is trivial, boilerplate and short (and could even be generated using a more advanced framework).

The relational type inferencer itself rather resembles the original implementation. The only difference (besides the syntax) is that instead of data constructors we use their logic counterparts:

```

let rec lookupo a g t =
  fresh (a' t' t1)
  (g ≡ (inj_pair a' t') % t1)
  (conde [
    (a' ≡ a) &&& (t' ≡ t);
    (a' ≠ a) &&& (lookupo a t1 t)
  ])

let infero expr typ =
  let rec infero gamma expr typ =
    conde [
      fresh (x)
        (expr ≡ v x)
        (lookupo x gamma typ);
      fresh (m n t)
        (expr ≡ app m n)
        (infero gamma m (arr t typ))
        (infero gamma n t);
      fresh (x l t t')
        (expr ≡ abs x l)
        (typ ≡ arr t t')
        (infero ((inj_pair x t) % gamma) l t')
    ]
  in
  infero (nil()) expr typ

```

## 9 Performance Evaluation

One of our initial goals was to evaluate what performance impact would choosing OCaml as a host language makes. In addition we spent some effort in order to implement miniKanren in an efficient, tagless manner, and, of course, the outcome of this decision also has to be measured. For comparison we took faster-miniKanren<sup>14</sup> — a full-fledged miniKanren implementation for Scheme/Racket. It turned out that faster-miniKanren implements a number of optimizations [5, 6] to speed up the search; moreover, the search order in our implementation initially was a little bit different. In order to make the comparison fair,

<sup>14</sup><https://github.com/webyrd/faster-miniKanren>

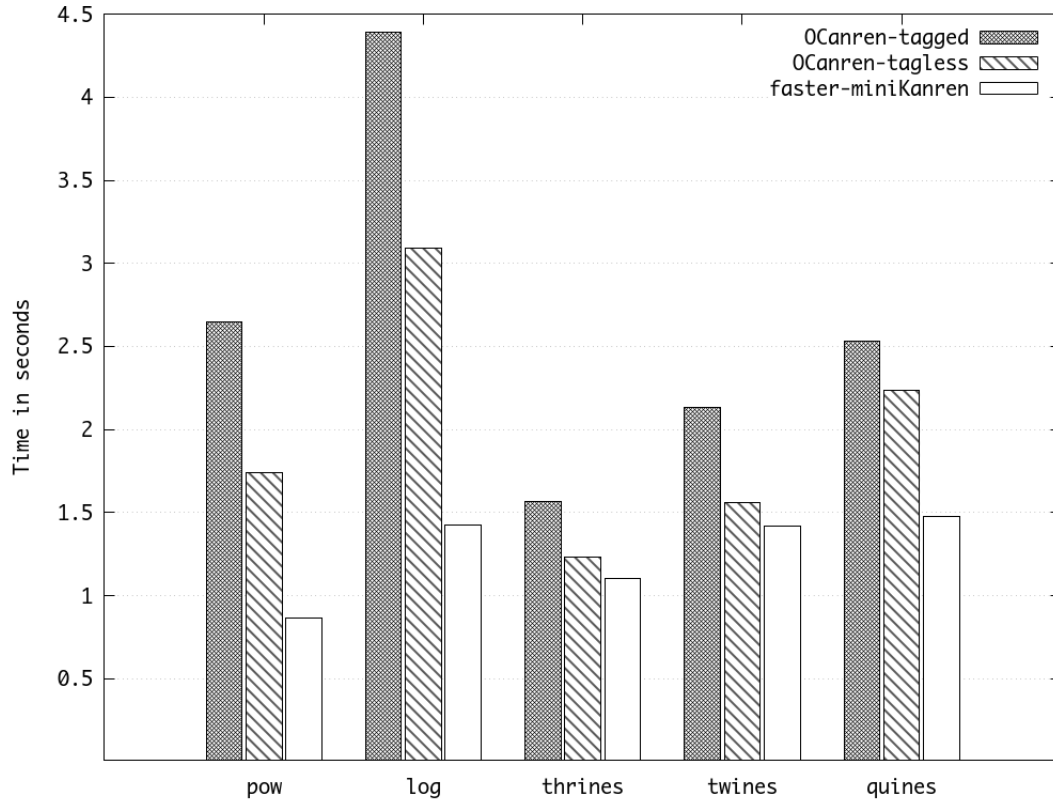


Figure 1: The Results of the Performance Evaluation

we additionally implemented all these optimizations and adjusted the search order to exactly coincide with what faster-miniKanren does.

For the set of benchmarks we took the following problems:

- **pow, log** — exponentiation and logarithm for integers in binary form. The concrete tests relationally computed  $3^5$  (which is 243) and  $\log_3 243$  (which is, conversely, 5). The implementation was adopted from [16].
- **quines, twines, trines** — self/co-evaluating program synthesis problems from [7]. The concrete tests took the first 100, 15 and 2 answers for these problems respectively.

The evaluation was performed on a desktop computer with Intel Core i7-4790K CPU @ 4.00GHz processor and 16GB of memory. For OCanren `ocaml-4.04.0+frame_pointer+flambda` was used, for faster-miniKanren — Chez Scheme 9.4.1. All benchmarks were executed in the natively compiled mode ten times, then average user time was taken. The results of the evaluation are shown on Figure 1. The whole evaluation repository with all scripts and detailed description is accessible from [GitHub](https://github.com/Kakadu/ocanren-perf)<sup>15</sup>.

The first conclusion, which is rather easy to derive from the results, is that the tagless approach indeed matters. Our initial implementation did not show essential speedup in comparison even with  $\mu$ Kanren (and was even *slower* on the logarithm and permutations benchmarks). The situation was improved drastically, however, when we switched to the tagless version.

<sup>15</sup><https://github.com/Kakadu/ocanren-perf>

Yet, in comparison with faster-miniKanren, our implementation is still lagging behind. We can conclude that the optimizations used in the Scheme/Racket version, have a different impact in the OCaml case; we save this problem for future research.

## 10 Conclusion

We presented a strongly-typed implementation of miniKanren for OCaml. Our implementation passes all tests written for miniKanren (including those for disequality constraints); in addition we implemented many interesting relational programs known from the literature. We claim that our implementation can be used both as a convenient relational DSL for OCaml and an experimental framework for future research in the area of relational programming.

## References

- [1] Claire E. Alvis, Jeremiah J. Willcock, Kyle M. Carter, William E. Byrd & Daniel P. Friedman (2011): *cKanren: miniKanren with Constraints*. In: *Proceedings of the 2011 Workshop on Scheme and Functional Programming (Scheme '11)*, doi:10.1.1.231.3635.
- [2] Franz Baader & Wayne Snyder (2001): *Handbook of Automated Reasoning*. Elsevier and MIT Press, doi:10.1016/B978-044450813-3/50010-2.
- [3] H. P. Barendregt (1992): *Handbook of Logic in Computer Science (Vol. 2)*. Oxford University Press, Inc., New York, NY, USA, doi:10.1.1.26.4391. Available at <http://dl.acm.org/citation.cfm?id=162552.162561>.
- [4] David C. Bender, Lindsey Kuper, William E. Byrd & Daniel P. Friedman (2009): *Efficient Representations for Triangular Substitutions: a Comparison in miniKanren*. Unpublished manuscript.
- [5] William E. Byrd (2009): *Relational Programming in Minikanren: Techniques, Applications, and Implementations*. Ph.D. thesis, Indianapolis, IN, USA, doi:10.1.1.363.5478. AAI3380156.
- [6] William E. Byrd & Michael Ballantyne (2017): *Personal communications*.
- [7] William E. Byrd, Eric Holk & Daniel P. Friedman (2012): *miniKanren, Live and Untagged: Quine Generation via Relational Interpreters (Programming Pearl)*. In: *Proceedings of the 2012 Annual Workshop on Scheme and Functional Programming, Scheme '12*, ACM, New York, NY, USA, pp. 8–29, doi:10.1145/2661103.2661105.
- [8] Manuel M. T. Chakravarty, Gabriel C. Ditu & Roman Leshchinskiy (2009): *Instant Generics: Fast and Easy*. doi:10.1.1.150.1033. Available at <http://www.cse.unsw.edu.au/~chak/papers/CDL09.html>.
- [9] Olivier Danvy (1998): *Functional Unparsing*. *Journal of Functional Programming* 8(6), doi:10.1017/S0956796898003104. Available at <https://dl.acm.org/citation.cfm?id=969603>.
- [10] Daniel Fridlender & Mia Indrika (2000): *Do we need dependent types?* *Journal of Functional Programming* 10(4), doi:10.1017/S0956796800003658. Available at <https://dl.acm.org/citation.cfm?id=967474>.
- [11] Daniel P. Friedman, William E. Byrd & Oleg Kiselyov (2005): *The Reasoned Schemer*. The MIT Press.
- [12] Jeremy Gibbons (2007): *Datatype-generic Programming*. In: *Proceedings of the 2006 International Conference on Datatype-generic Programming, SSDGP'06*, Springer-Verlag, Berlin, Heidelberg, pp. 1–71, doi:10.1007/978-3-540-76786-2\_1. Available at <http://dl.acm.org/citation.cfm?id=1782894.1782895>.
- [13] M. Hanus, H. Kuchen & J.J. Moreno-Navarro (1995): *Curry: A Truly Functional Logic Language*. In: *Proc. ILPS'95 Workshop on Visions for the Future of Logic Programming*, pp. 95–107, doi:10.1.1.33.7348.

- [14] Jason Hemann & Daniel P. Friedman (2013):  *$\mu$ Kanren: A Minimal Core for Relational Programming*. In: *Proceedings of the 2013 Workshop on Scheme and Functional Programming (Scheme'13)*. Available at <http://webyrd.net/scheme-2013/papers/HemannMuKanren2013.pdf>.
- [15] Jason Hemann, Daniel P. Friedman, William E. Byrd & Matthew Might (2016): *A Small Embedding of Logic Programming with a Simple Complete Search*. In: *Proceedings of the 12th Symposium on Dynamic Languages, DLS 2016*, ACM, New York, NY, USA, pp. 96–107, doi:10.1145/2989225.2989230.
- [16] Oleg Kiselyov, William E. Byrd, Daniel P. Friedman & Chung-Chieh Shan (2008): *Pure, Declarative, and Constructive Arithmetic Relations (Declarative Pearl)*. In: *Proceedings of the 9th International Conference on Functional and Logic Programming, FLOPS'08*, Springer-Verlag, Berlin, Heidelberg, pp. 64–80, doi:10.1007/978-3-540-78969-7\_7. Available at <http://dl.acm.org/citation.cfm?id=1788446.1788456>.
- [17] Oleg Kiselyov, Chung-chieh Shan, Daniel P. Friedman & Amr Sabry (2005): *Backtracking, Interleaving, and Terminating Monad Transformers: (Functional Pearl)*. *SIGPLAN Not.* 40(9), pp. 192–203, doi:10.1145/1090189.1086390.
- [18] Claessen Koen & Peter Ljungl of (2001): *Typed Logical Variables in Haskell*. In: *Electronic Notices in Theoretical Computer Science*, 41, doi:10.1016/S1571-0661(05)80544-4.
- [19] Ramana Kumar (2010): *Mechanising Aspects of miniKanren in HOL*. Bachelor Thesis, The Australian National University.
- [20] Tim Sheard & Simon Peyton Jones (2002): *Template Meta-programming for Haskell*. In: *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell, Haskell '02*, ACM, New York, NY, USA, pp. 1–16, doi:10.1145/581690.581691.
- [21] Zoltan Somogyi, Fergus Henderson & Thomas Conway (1996): *The execution algorithm of mercury, an efficient purely declarative logic programming language*. *The Journal of Logic Programming* 29(1), pp. 17 – 64, doi:10.1016/S0743-1066(96)00068-4. Available at <http://www.sciencedirect.com/science/article/pii/S0743106696000684>. High-Performance Implementations of Logic Programming Systems.
- [22] Michael Spivey & Silvija Seres (1999): *Embedding PROLOG in HASKELL*. In: *Proceedings of the 1999 Haskell Workshop*, doi:10.1.1.35.8710.
- [23] Wouter Swierstra (2008): *Data Types á la Carte*. *Journal of Functional Programming* 18(4), doi:10.1017/S0956796808006758.
- [24] Leo White, Frédéric Bour & Jeremy Yallop (2015): *Modular Implicits*. In: *Proceedings ML Family/OCaml Users and Developers workshops*, 198, pp. 22–63, doi:10.4204/EPTCS.198.2. Available at <http://arxiv.org/abs/1512.01895>.
- [25] Jeremy Yallop (2007): *Practical Generic Programming in OCaml*. In: *Proceedings of the 2007 Workshop on Workshop on ML, ML '07*, ACM, New York, NY, USA, pp. 83–94, doi:10.1145/1292535.1292548.
- [26] Jeremy Yallop & Leo White (2014): *Lightweight Higher-Kinded Polymorphism*, pp. 119–135. Springer International Publishing, Cham, doi:10.1007/978-3-319-07151-0\_8.