Typed Embedding of a Relational Language in OCaml

Dmitry Kosarev  Dmitri Boulytchev
St.Petersburg State University
Saint-Petersburg, Russia
Dmitrii.Kosarev@protonmail.ch  dboulytchev@math.spbu.ru

1. Introduction

Relational programming [1] 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 purely theoretical standpoint, this approach may have a practical value: some problems look much simpler, if they are considered as queries to relational specification. There is a number of appealing examples, confirming this observation: a type checker for simply typed lambda calculus (and, at the same time, type inferencer and solver for the inhabitation problem), an interpreter (capable of generating “quines” — programs, producing themselves as result) [4], list sorting (capable of producing all permutations), etc.

Many logic programming languages, such as Prolog, Mercury¹, or Curry² to some extent can be considered as relational. We have chosen miniKanren³ as model language, because it was specifically designed as relational DSL, embedded in Scheme/Racket. Being rather a minimalistic language, which can be implemented with just a few data structures and combinators, miniKanren found its way in dozens of host languages, including Haskell, Standard ML, and OCaml.

There is, however, a predictable glitch in implementing miniKanren for a strongly typed language. Designed in a metaprogramming-friendly and dynamically typed realm of Scheme/Racket, original miniKanren implementation pays very little attention to what has a significant importance in (specifically) ML or Haskell. In particular, one of 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 libraries⁴ as well as existing OCaml implementation⁵ take this approach. As a result, the original implementation can be retold with all its elegance; relational specifications, however, become weakly typed. Another approach is to utilize ad hoc polymorphism and provide type-specific unification for each “interesting” type: Molog⁶ and MiniKanrenT⁷, 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 Haskell⁸, is desirable. There is, actually, another potential approach, but we do not know, if anybody tried it: to implement unification for generic representation of types as sum-of-products and fixpoints of functors [10, 11]. Thus, unification would work for any types, for which representation is provided. We assume that implementing representation would require less boilerplate code.

As follows from this exposition, typed embedding of miniKanren in OCaml can be done with a combination of datatype-generic programming [8] and ad hoc polymorphism. There are a number of generic frameworks for OCaml (for example, [9]). On the other hand, the support for ad hoc polymorphism in OCaml is weak; there is nothing comparable in power with Haskell type classes, and despite sometimes 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 example, module implicits [5]) require patching the compiler, which we tend to avoid.

We present an implementation of a set of relational combinators in OCaml, which, technically speaking, corresponds to μKanren [2] with disequality constraints [3]; syntax extension for “fresh” construct is added as well. The contribution of our work is as follows:

1. Our implementation is based on polymorphic unification, which, like polymorphic comparison, can be used for almost arbitrary types. The implementation of polymorphic unification uses unsafe features and relies on intrinsic knowledge of runtime representation of values; we show, however, that this does not compromise type safety. Practically, we applied purely ad hoc approach since the features, which would provide less ad hoc solution, are not yet integrated into the mainstream language.

2. We describe a uniform and scalable pattern for using types for relational programming, which helps in converting typed data to- and from relational domain. With this pattern, only one generic feature (“map/morphism/Functor”) is needed, and thus virtually any generic framework for OCaml can be used. Despite being rather a pragmatic observation, this pattern, as we believe, would lead to more regular and easy to maintain relational specifications.

References

1. https://mercurylang.org
2. http://www-ps.informatik.uni-kiel.de/currywiki
6. https://github.com/acfoltzer/Molog
7. https://github.com/jvranish/MiniKanrenT
8. https://wiki.haskell.org/Template_Haskell
3. We provide a simplified way to integrate relational and functional code. Our approach utilizes well-known pattern [6, 7] for variadic function implementation and makes it possible to hide refinement of answers phase from an end-user.

The rest of the paper is organized as follows: in the next section we discuss polymorphic unification, and show, that standard unification with triangular substitution respects typing. Then we present our approach to handle user-defined types by injecting them into logic domain. Next section describes top-level primitives and addresses the problem of relational and functional code integration. Then, we present a complete example of relational specification, written with the aid of our library. The final section concludes.

We expect from reader some familiarity with basic concepts behind original miniKanren implementation as well as principles of relational programming.

2. Polymorphic Unification

We consider it rather natural to employ polymorphic unification in the language, already equipped with polymorphic comparison — a convenient, but somewhat disputable8 feature. Like polymorphic comparison, polymorphic unification performs traversal of values, exploiting intrinsic knowledge of their runtime representation. The undeniable benefit of this solution is that in order to perform unification for user types no “boilerplate” code is needed. On the other hand, all pitfalls of polymorphic comparison are inherited as well; in particular, unification can loop 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 them. Another drawback is that the implementation of polymorphic unification relies on runtime representation of values and have to be fixed every time the representation changes. Finally, as it is written in unsafe manner using 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 (mapping from logical variables to terms), which later is used to refine answers and reify constraints. So, generally speaking, we have to show, that no ill-typed terms are constructed as a result.

Polymorphic unification is introduced via the following function:

```
val unify : α logic → α logic → subst option → subst option
```

where “α logic” stands for the type α, injected into the logic domain, “subst” — for the type of substitution. 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). Note, the type of substitution is not polymorphic, which means, that compiler completely loses the track of types for values, stored in a substitution. These types are recovered later during refinement of answers.

To justify the correctness of unification, we consider a set of typed terms, each of which has one of two forms

\[ x^\tau \mid C^\tau(t_1^\tau, \ldots, t_k^\tau) \]

where \( x^\tau \) denotes a logical variable of type \( \tau \), \( C^\tau \) — some constructor of type \( \tau \), \( t_i^\tau \) — some terms of types \( \tau_i \). We reflect by \( t_1^\tau \) the fact of \( t_2^\tau \) being a subterm of \( t_1^\tau \), and assume, that \( \rho \) is unambiguously determined by \( t_1 \), \( \tau \), and a position of \( t_2 \) “inside” \( t_1 \).

Outside unification the compiler maintains typing, which means, that all terms, subterms, and variables agree in their types in all contexts. However, as our implementation resorts to unsafe features, we have to manually repeat this work for unification code.

We argue, that the following three invariants are maintained for any substitution \( s \), involved in unification:

1. if \( t_1^\tau[x^\tau] \) and \( t_2^\tau[x^\tau] \) — two arbitrary terms (in particular, \( t_1 \) and \( t_2 \) may be the same), bound in \( s \) and containing occurrences of variable \( x \), then \( \rho = \tau \) (different occurrences of the same variable in \( s \) are attributed with the same type);
2. if \( (s \ x^\tau) \) is defined, then \( (s \ x^\tau) \rightarrow t^\tau \) (a substitution always binds a variable to a term of the same type);
3. each variable in \( s \) preserves its type, assigned by the compiler (from the first two invariants it follows, that this type is unique; note also, that all variables are created and have their types assigned outside unification, in a type-safe world).

The initial (empty) substitution trivially fulfills these invariants; hence, it is sufficient to show, that they are preserved by unification.

The following snippet presents the implementation of unification with triangular substitution in only a little bit more abstract form, than actual code (for example, “occurs check” is omitted):

```
let rec walk s = function
| x^\tau when x ∈ dom s → walk s (s x)^\tau
| t^\tau → t^\tau

let rec unify t_1^\tau t_2^\tau = function
| None → None
| Some s as subst →
  match walk s t_1, walk s t_2 with
  | x_1^\tau, x_2^\tau when x_1 = x_2 → subst
  | x_1^\tau, (t_2^\tau)^\tau → Some (s[x_1 ← t_2^\tau])
  | (t_1^\tau)^\tau, x_2^\tau → Some (s[x_2 ← t_1^\tau])
  | C^\tau(t_1^\tau_1, \ldots, t_k^\tau_k), C^\tau(p_1^\tau, \ldots, p_k^\tau) →
    unify t_1^\tau_1 p_1^\tau (… (unify t_k^\tau_k p_k^\tau (sub))…)
  | … → None
```

Type annotations, included in the snippet above, can be justified by the following reasonings10:

1. Line 2: the type of \( (s \ x^\tau) \) is \( \tau \) due to invariant 2; hence, the type of walk result coincides with the type of its second argument (technically, an induction on the number of recursive invocations of walk is needed).
2. Line 9: the substitution is left unchanged, hence all invariants are preserved.
3. Line 10 (and, symmetrically, line 11): first, note, that \( (s \ x_1) \) is undefined (otherwise walk would not return \( x_1 \)). Then, \( x_1 \) and \( t_2 \) have the same type, which justifies the preservation of invariant 2. Finally, either \( x_1 = t_1 \) (and, then, \( \tau \) is the type of \( x_1 \), assigned by the compiler), or \( x_1 \) is retrieved from \( s \) with type \( \tau \) — both cases justify invariants 1 and 3. The same applies to the pair \( t_2 \) and \( t_1 \).
4. The previous paragraph justifies the base case for inductive proof on the number of recursive invocations of unify.

Function unify is not directly accessible at the user level; it used to implement both unification (“==” and “=/=”) goals. The implementation generally follows [3].

8Sec, for example, https://blogs.janestreet.com/the-perils-of-polymorphic-compare

9We omit verbal description of unification algorithm; the details can be found in [2].
3. Logic Variables and Injection

Unification, considered in Section 2, works for values of type \( \alpha \) logic. Any value of this type can be seen as either value of type \( \alpha \), or logical variable of type \( \alpha \). The type itself is made abstract, but its values can be uncovered after refinement (see Section 4).

Free variables solely can be created using "fresh" construct of miniKanren. Note, since the unification is implemented in untyped manner, we cannot use simple pattern matching to distinguish logical variables from other logical values. Special attention was paid to implement variable recognition in constant time.

Apart from variables, other logical values can be obtained by injection; conversely, sometimes logical value can be projected to a regular one. We supply two functions\(^{\dagger}\) for these purposes:

\[
\begin{align*}
\text{val} \ (\uparrow) & : \alpha \rightarrow \alpha \ \text{logic} \\
\text{val} \ (\downarrow) & : \alpha \ \text{logic} \rightarrow \alpha
\end{align*}
\]

As expected, injection is total, while projection is partial. Using these functions and type-specific "map", which can be derived automatically using a number of existing frameworks for generic programming, one can easily provide injection and projection for user-defined datatypes. We consider user-defined list type as an example:

\[
\begin{align*}
\text{type} \ (\alpha, \beta) \ \text{list} & = \text{Nil} \ | \ \text{Cons} \ \text{of} \ \alpha \ast \beta \\
\text{type} \ \alpha \ \text{glist} & = (\alpha, \alpha \ \text{glist}) \ \text{list} \\
\text{type} \ \alpha \ \text{llist} & = (\alpha \ \text{logic}, \alpha \ \text{llist}) \ \text{list} \ \text{logic}
\end{align*}
\]

let rec inj_list l = ↑(maplist (↑) inj_list l)
let rec prj_list l = mapList (↓) prj_list (↓ l)

Here "list" is a custom type for lists; note, that it is made more polymorphic, than usual — we abstracted it from itself and made it non-recursive (pragmatically speaking, it is desirable to make a type fully abstract, thus logic variables can be placed in arbitrary positions).

Then we provided two specialized versions — "glist" ("ground" list), which corresponds to regular, non-logic lists, and "llist" ("logical" list), which corresponds to logical lists with logical elements. Using a single type-specific function maplist, we easily provided injection (of type \( \alpha \ \text{glist} \rightarrow \alpha \ \text{llist} \)) and projection (of type \( \alpha \ \text{llist} \rightarrow \alpha \ \text{glist} \)).

In context of these definitions, now we can implement relational list concatenation, which is one of first-step examples of miniKanren programming:

\[
\begin{align*}
\text{let rec append}^{\dagger} \ x \ y \ xy = \\
\quad \text{conde} [ \\
\quad \quad (x \ \text{===} \ \uparrow \text{Nil}) \ \&\& \ (y \ \text{===} \ y) ; \\
\quad \quad \text{fresh} \ (h \ t \ ty) \\
\quad \quad (x \ \text{===} \ ↑(\text{Cons}) (h \ t)) \\
\quad \quad (y \ \text{===} \ ↑(\text{Cons}) (h \ t y)) \\
\quad \quad (\text{append}^{\dagger} \ t \ y \ ty) \\
\]
\]

Note, in the definition of append\(^{\dagger}\) we used only default injection ("↑"). Customized version most likely would appear in some top-level goal, for example:

\[
\begin{align*}
\text{fun} \ q \rightarrow \text{append}^{\dagger} \ \text{(inj_list [1; 2; 3])} \\
\quad \quad \quad \text{(inj_list [4; 5; 6])} \\
\quad \quad \quad q
\end{align*}
\]

\(^{\dagger}\)"inj" and "prj" in concrete syntax.

4. Refinement and Top-Level Primitives

The result of a relational program is a stream of substitutions, each of which represents a certain answer. As a rule, a substitution binds many intermediate logical variables, created by "fresh" in the course of execution. A meaningful answer has to be refined.

In our implementation refinement is represented by the following function:

\[
\text{val} \ \text{refine} : \ \text{subst} \rightarrow \alpha \ \text{logic} \rightarrow \alpha \ \text{logic}
\]

This function takes a substitution and a logical value and recursively substitutes all logical variables in that value w.r.t. the substitution until no occurrences of bound variables are left. Since in our implementation the type of substitution is not polymorphic, refine is also implemented in an unsafe manner. However, it is easy to see, that refine does not produce ill-typed terms. Indeed, all original types of variables are preserved in a substitution due to invariant 3 from Section 2. Unification does not change unified terms, so all terms, bound in a substitution, are well-typed. Hence, refine always substitutes some subterm in a well-typed term with another term of the same type, which preserves well-typedness.

In addition to performing substitutions, refine also reifies disequality constrains. Reification attaches to each free variable in a refined term a list of refined terms, describing disequality constraint for that free variable. Note, disequality can be established only for equally typed terms, which justifies type-safety of reification. Note also, additional care has to be taken to avoid infinite looping, since refinement and reification are mutually recursive, and refinement of a variable can be potentially invoked from itself due to a chain of disequality constraints.

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

\[
\text{val} \ \text{destruct} : \ \alpha \ \text{logic} \rightarrow \\
\quad \quad [\ \text{Var of} \ \text{int} \ast \alpha \ \text{logic list} \ | \ \text{Value of} \ \alpha]
\]

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

We did not make refine accessible for an end-user; instead we provided a set of top-level combinators, which should be used to surround relational code and perform refinement in a transparent manner. Note, from pragmatic standpoint only variables, supplied as arguments for the top-level goal, have to be refined (the original miniKanren implementation follows the same convention).

The toplevel primitive in our implementation is run, which takes three arguments. The exact type of run is rather complex and non-instructive, so we better describe the typical form of its application:

\[
\text{run} \ \pi \ (\text{fun} \ l_1 \ldots l_n \rightarrow G) \ (\text{fun} \ a_1 \ldots a_n \rightarrow H)
\]

Here \( \pi \) stands for numeral, which describes the number of parameters for two other arguments of run, \( l_1 \ldots l_n \) — free logical variables, \( G \) — a goal (which can make use of \( l_1 \ldots l_n \)), \( a_1 \ldots a_n \) — refined answers for \( l_1 \ldots l_n \), respectively, and, finally, \( H \) — a handler (which can make use of \( a_1 \ldots a_n \)). The types of \( l_1 \ldots l_n \) are inferred from \( G \), and the types of \( a_1 \ldots a_n \) are inferred from types of \( l_1 \ldots l_n \); if \( l_i \) has type \( \text{t logic} \), then \( a_i \) has type \( \text{t logic stream} \). In other words, user-defined handler takes streams of refined answers for all variables, supplied to the top-level goal. All streams \( a_i \) contains 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, by tradition, \( q \), \( qr \), \( qrr \) etc.), and a successor function, which can be applied to existing numeral to increment the number of expected arguments. The technique, used to implement them, generally follows [6, 7].
5. An Example

Here we present an example of relational specification, written with the aid of our library. For this example we take list sorting; specifically, we present 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 much declarative as possible to demonstrate the advantages of relational approach. From this standpoint, we can claim, that sorted version of empty list is empty list, and sorted version of non-empty list is its smallest element, concatenated with sorted version of list, containing all its remaining elements.

The following snippet literally implements this definition:

```ocaml
let rec sort x y = conde [
  (x === ↑Nil) &&& (y === ↑Nil);
  fresh (s xs xs')
  (cons (s, xs') y)
  (sort x xs) (smallest'' x s xs)
]
```

The meaning of the expression

```
smallest'' 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''` can be implemented using case analysis (note, that “l” here is a non-empty list):

```ocaml
let rec smallest'' l s l' = conde [
  (l === ↑(Cons (s, ↑Nil))) &&& (l' === ↑Nil);
  fresh (h t s' t' max)
  (l' === ↑(Cons(max,t')))
  (l === ↑(Cons(h,t)))
  (minmax h s' s max)
  (smallest'' t s' t')
]
```

Finally, we implement relational minimum-maximum calculation primitive:

```ocaml
let minmax a b min max = conde [
  (min === a) &&& (max === b) &&& (le a b);
  (max === a) &&& (min === b) &&& (gt a b)]
```

Here "le" and "gt" are built-in comparison goals for natural numbers in Peano form.

Having relational `sort''`, we can implement sorting for regular integer lists:

```ocaml
let sort l = run q (sorto @® inj_nat_list l)
  (fun q s -> prj_nat_list @® Stream.hd q)
```

Here `Stream.hd` is a function, which takes a head from a lazy stream of answers.

It is interesting, that since `sort''` is relational, it can be used to calculate the list of all permutations for a given list. Indeed, each permutation, being sorted, 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:

```ocaml
let perm l = map prj_nat_list @®
  run q (fun q → fresh (r)
    (sort'' (inj_nat_list l) r)
    (sort'' q r)
  )
  (Stream.take "n:(fact @® length 1))
```

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

6. Conclusion

We presented 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.

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

We also want to express our gratitude to William Byrd, who infected us with relational programming, and for the extra time he sacrificed as both our tutor and friend.

References