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

Abstract

We present an implementation of relational programming language miniKanren as a set of combinators and syntax extension for OCaml. The key feature of our approach is *polymorphic unification*, which can be used to unify data structures of almost 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 relational and functional code integration.

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.

We have chosen miniKanren[1] 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 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 the capstone constructs of miniKanren — unification — has to work for different data structures, which may have types, different beyond parametricity.

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 the "`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.
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.
3. We provide a simplified way to integrate relational and functional code. Our approach utilizes well-known pattern [5, 6] for variadic function implementation and makes it possible to hide refinement of the answers phase from an end-user.

The source code of our implementation is accessible from `https://github.com/dboulytchev/OCanren`. An extended version of this abstract can be downloaded from `http://oops.math.spbu.ru/papers/ocanren.pdf`.

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

2. Polymorphic Unification

We consider it rather natural to employ polymorphic unification in the language, already equipped with polymorphic comparison — a convenient, but somewhat disputable feature.

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.

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

$$x^\tau \mid C^\tau(t_1^{\tau_1}, \ldots, t_k^{\tau_k})$$

where x^τ denotes a logical variable of type τ, C^τ — some constructor of type τ, $t_i^{\tau_i}$ — some terms of types τ_i. We reflect by $t_1^\tau[t_2^\rho]$ the fact of t_2^ρ being a subterm of t_1^τ, and assume, that ρ is unambiguously determined by t_1, τ, and a position of t_2 "inside" t_1.

We show, that the following three invariants are maintained for any substitution s, involved in unification:

1. if $t_1[x^\tau]$ and $t_2[x^\rho]$ — 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 $(s\ \ x^\tau) = t^\tau$ (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 function `unify` is not directly accessible at the user level; it is used to implement both unification ("`===`") and disequality ("`=/=`") goals. The implementation generally follows [3].

3. Logic Variables and Injection

Unification, considered in Section 2, works for values of type α `logic`. Any value of this type can be seen as either a value of type α, or a logical variable of type α.

Free variables can be created solely using the "`fresh`" construct of miniKanren. Note, that since the unification is implemented in an untyped manner, we can not use simple pattern matching to

distinguish logical variables from other logical values. A special attention was paid to implement variable recognition in constant time.

Apart from variables, other logical values can be obtained by injection; conversely, sometimes a logical value can be projected to regular one. We supply two functions[2] for these purposes

```
val (↑) : α → α logic
val (↓) : α logic → α
```

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:

```
type (α, β) list = Nil | Cons of α * β

type α glist = (α, α glist) list
type α llist = (α logic, α llist) list logic

let rec inj_list l = ↑(map_list (↑) inj_list l)
let rec prj_list l = map_list (↓) prj_list (↓ l)
```

Here "list" is a custom type for lists; note, that it made more polymorphic, than usual — we abstracted it from itself and made it non-recursive. 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 single type-specific function $\texttt{map}_{\texttt{list}}$, we easily provided injection (of type α glist $\rightarrow$ α llist) and projection (of type α llist $\rightarrow$ α glist).

4. Refinement and Top-Level Primitives

The result of 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:

```
val refine : subst → α logic → α 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, that disequality can be established only for equally typed terms, which justifies type-safety of reification.

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:

```
run n̄ (fun l1 ... ln → G) (fun a1 ... an → H)
```

[2] "inj" and "prj" in concrete syntax.

Here $\overline{n}$ stands for a *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 the types of $l_1 \ldots l_n$: if l_i has type t logic, then a_i has type 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 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, by tradition, q, qr, qrs 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 [5, 6].

As a final example we consider a program, which calculates the list of all permutations of given list of integers, using relational sorting (some supplementary function definitions are omitted):

```
let minmax^o a b min max = conde [
    (min === a) &&& (max === b) &&& (le^o a b);
    (max === a) &&& (min === b) &&& (gt^o a b)]
let rec smallest^o 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^o h s' s max)
      (smallest^o t s' t')]
let rec sort^o x y = conde [
    (x === ↑Nil) &&& (y === ↑Nil);
    fresh (s xs xs')
      (y === ↑(Cons (s, xs')))
      (sort^o xs xs')
      (smallest^o x s xs)]
let perm l = map prj_nat_list @@ run q
  (fun q → fresh (r) (sort^o (inj_nat_list l) r)(sort^o q r))
  (take ~n:(fact @@ length l))
```

Here take is a function for a stream, which returns the specified number of its first items as regular list.

References

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

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

[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] Olivier Danvy. Functional Unparsing // Journal of Functional Programming, Vol. 8, Issue 6, November 1998.

[6] Daniel Fridlender, Mia Indrika. Do we need dependent types? // Journal of Functional Programming, Vol. 10, Issue 4, July 2000.