A Limited Case for Reification by Type Inference

Jason Gross

CoqPL 2021

Abstract

Proof by reflection is a common and well-studied automation tool. Reification—generally written using $\mathcal{L}_{tac}$, OCaml, typeclasses, or canonical structures—is the means by which a structured representation is derived from an unstructured representation. The reflective automation then operates on the structured representation, relying on an interpretation or denotation function to justify a correspondence between the structured and unstructured representations.

A couple of years ago, I presented a trick for blazing fast reification in two lines of $\mathcal{L}_{tac}$—using the `pattern` tactic—which I termed reification by parametricity. While I still advocate for parametricity as the preferred method of domain-specific reification, I would like to present here yet another method.

While reification typically requires meta-programming features, I was surprised and delighted to discover that, in some restricted cases, reification can be performed entirely by a combination of the notation system and type inference. In some sense, this is trivial: by redefining the basic syntactic notations, a term can be "reified" merely by writing the same symbols in another scope. In another sense, though, this trick is quite surprising: we use the notation system merely to insert "reify here" functions at every atom, and the reification itself is in fact performed by type inference. My hope is that the audience will walk away with this new trick in their toolbox, and that some day some problem will come along demanding a slight generalization of this trick, and that generalization will be new and interesting in its own right. This, after all, is how reification by parametricity was discovered.

I propose to present the one example I have for this trick: reifying the type structure of a function in a way that allows manipulations of the arguments, such as uncurrying, reassociation of the uncurried structure, and reordering. I will present the simple code for this example in detail. My goal will be that the audience understand completely how it works, why it works, and how it might be used elsewhere.

We reproduce the code here without explanation on the following page.

```
Inductive curry_types := ccons (A : Type) (rest : curry_types) | cnil.

Fixpoint denoteCurried (A : curry_types) : Type :=
  match A with
  | cnil => unit
  | ccons A As => A * denoteCurried As
  end.

Definition curry {A As B} (f : A -> denoteCurried As -> B)
  : denoteCurried (ccons A As) -> B
  := fun '(a, b) => f a b.

Fixpoint denoteCurried_rev' (A : curry_types) (so_far : Type) : Type
  := match A with
     | cnil => so_far
     | ccons A As => denoteCurried_rev' As (so_far * A)
     end.
Definition denoteCurried_rev (A : curry_types) : Type
  := match A with
     | cnil => unit
     | ccons A As => denoteCurried_rev' As A
     end.

Fixpoint curry_rev' {T} (A : curry_types) (so_far : Type)
  : denoteCurried_rev' A so_far -> (so_far * denoteCurried A -> T) -> T
  := match A with
     | cnil => fun v k => k (v, tt)
     | ccons A As => fun v k => curry_rev' As _ v (fun '(sf, a, v) => k (sf, (a, v)))
     end.
Definition curry_rev {T} (A : curry_types)
  : denoteCurried_rev A -> (denoteCurried A -> T) -> T
  := match A with
     | cnil => fun v k => k v
     | ccons A As => fun v k => curry_rev' As A v k
     end.

Definition rev_curry {A B} (f : denoteCurried A -> B) : denoteCurried_rev A -> B
  := fun v => curry_rev _ v f.
Notation "'->curry' x , .. , y => v"
  := (curry (fun x => .. (curry (fun y (_ : denoteCurried cnil) => v)) .. ))
       (at level 200, x closed binder, y closed binder, v at level 100).

Eval cbv -["+"] in ->curry x, y, z, w => x + y + z + w.
(* = fun '(a, (a0, (a1, (a2, _)))) => a + a0 + a1 + a2
   : denoteCurried (ccons nat (ccons nat (ccons nat (ccons nat cnil)))) -> nat *)
Eval cbv -["+"] in rev_curry (->curry x, y, z, w => x + y + z + w).
(* = fun '(sf1, a1, a0, a) => sf1 + a1 + a0 + a
   : denoteCurried_rev (ccons nat (ccons nat (ccons nat (ccons nat cnil)))) ->
     nat *)
```