Reimplementing Reifiers for OCanren Using the “Lightweight Higher-kinded Polymorphism” Technique

Li Yue
Programming Languages and Tools Lab
JetBrains Research
December 2021

Reimplementing Reifiers for OCanren Using the “Lightweight Higher-kinded Polymorphism” Technique
An overview

- **Background**: OCanren, injection and reification.
- **Problem**: The current implementation is unscalable due to the need for a predefined set of functors.
- **Reason**: The lack of higher-kinded polymorphism in OCaml necessitates the use of functors. Replacing functors by higher-kinded polymorphic functions would make the code less cumbersome.
- **Approach**: “Lightweight Higher-kinded Polymorphism” and its application to the problem.
- **Result**: The technique is applicable and eliminates the old functors. But a new set of predefined functors is added.
- **Limitation**: Scalability problem for OCaml is hard. We didn’t solve it, other people neither.

Background: OCanren, injection and reification

- `Cons(y,Nil)` and `Cons(2,z)` are OCanren lists, implemented in OCaml as

```
type ('a,'b) list = Nil | Cons of 'a * 'b
type var
```

- If `y:var` and `z:var` then
  - `Cons(y,Nil):(var,('a,'b)list')list'`
  - `Cons(2,z)   :(int,var)list'`

- `Cons(2,z)` reifies to `Val(Cons(Val 2,Var id_z))`
- `Cons(y,Nil)` reifies to `Val(Cons(Var id_y, Val Nil))`

Injection:
OCaml sees two incompatible types. By injection (safe type cast using unsafe OCaml features), both take the type of logical list of logical integer.

Reification:
Parsing an OCanren value to an AST.

Problem: Reification requires a predefined set of functors, hindering scalability

```
module type T1 =
  sig
    type 'a t
  end

module type T2 =
  sig
   type ('a, 'b) t
  end

module type T3 =
  sig
    type ('a, 'b, 'c) t
  end
```

```
module Fmap (T : T1) :
 sig
   val reify : …
 end

module Fmap2 (T : T2) :
 sig
   val reify : …
 end

module Fmap3 (T : T3) :
 sig
   val reify : …
 end
```

Problem Analysis

- Full scalability requires as many predefined functors as the number of possible type parameters for a type constructor.
- The current OCaml implementation supports tuple of up to 4194303 elements. OCanren implementers cannot afford to write this much functors.
- The problem with functors is twofold:
  - the duplication, and
  - functors themselves are cumbersome

Reason: Lack of higher-kinded polymorphism

- “Lower-kinded” polymorphism is abstraction over type parameters.
- e.g. int list, bool list, char list —> ‘a list
- Higher-kinded polymorphism is abstraction over type constructors.
- e.g. int list, int tree, int option —> int ‘b

Reason: Lack of higher-kinded polymorphism

- OCaml doesn't allow a type variable to occur in the position of a type constructor, lacking higher-kinded polymorphism.

- e.g. OCaml rejects `map: ('a -> 'b)-> 'a 'c -> 'b 'c`

- OCaml uses functors to realize some effect of higher-kinded polymorphism.

- We may make the reifiers implementation less cumbersome if we can just replace the set of functors by a set of higher-kinded polymorphic functions.

Approach: Lightweight higher-kinded polymorphism

- *Lightweight Higher-Kinded Polymorphism* Jeremy Yallop and Leo White, Functional and Logic Programming 2014
- Encode `'a 'b` as `('a, 'b) app`. The first `'b` is higher-kinded, the next `'b` is lower-kinded.
  - e.g. `('a -> 'b)-> 'a 'c -> 'b 'c` becomes `('a -> 'b) -> ('a,'c) app -> ('b,'c) app`

Result: The technique is applicable

- We can now define the reifiers as a set of higher-kinded polymorphic functions typed using the “lightweight” technique, instead of as a set of functors.

- The lightweight higher-kinded polymorphism technique is itself implemented with a predefined set of functors, therefore the scalability problem is not solved, but it is known to be hard and neither other people solved it.

Thanks !