Formalising Free Selective Functors in Coq

Georgy Lukyanov
Newcastle University

Andrey Mokhov
Newcastle University

{g.lukyanov2@newcastle.ac.uk}
SPLV'19, August 8, 2019

```haskell
class Applicative f => Selective f where
    select :: f (b + a) -> f (b -> a) -> f a
```

```haskell
class Applicative f => Selective f where
    select :: f (b + a) -> f (b -> a) -> f a
```

Conditional effects for finite types

```haskell
ifS :: Selective f => f Bool -> f a -> f a -> f a
```

Free Selective Functors

```
Inductive Select (F : Type -> Type) (A : Set) : Set :=
    Pure     : A -> Select F A
  | MkSelect : forall (B : Set),
               Select F (B + A) -> F (B -> A) -> Select F A.
```

Free Selective Functors

```
Inductive Select (F : Type -> Type) (A : Set) : Set :=
    Pure     : A -> Select F A
  | MkSelect : forall (B : Set),
                 Select F (B + A) -> F (B -> A) -> Select F A.
```

- A is a non-uniform index

Simple Proofs: `Functor` instance and laws

```
Function Select_map {A B : Set} `{Functor F}
         (f : A -> B) (x : Select F A) : Select F B :=
  match x with
  | Pure a => Pure (f a)
  | MkSelect x y =>
    MkSelect (Select_map (Either_map f) x)
                         (fmap (fun k => f \o k) y)
  end.
```

```
forall x, Select_map id x = id x.

forall f g x,
    (Select_map f \o Select_map g) x =
    Select_map (f \o g) x.
```

Defining Selective (and Applicative) instance requires well-founded recursion

```
Fixpoint Select_depth {A : Set} {F : Type -> Type}
         (x : Select F A) : nat :=
  match x with
  | Pure a => O
  | MkSelect y _ => S (Select_depth y)
  end.
```

Well-founded recursion via depth measure

```
forall x f,
  Select_depth (Select_map f x) = Select_depth x.
```

- Use `Select_depth` with `Function` or `Equations` plugin

```
Theorem Select_Applicative_law_Interchange
        `{FunctorLaws F} :
  forall (A B : Set) (u : Select F (A -> B)) (y : A),
  u <*> pure y = pure (fun f => f y) <*> u.
Proof. induction u.
```

```
Theorem Select_Applicative_law_Interchange
        `{FunctorLaws F} :
  forall (A B : Set) (u : Select F (A -> B)) (y : A),
  u <*> pure y = pure (fun f => f y) <*> u.
Proof. induction u.
```

```
Error: Abstracting over the terms "S" and "u" leads to a term
fun (S0 : Set) (u0 : Select F S0) =>
u0 <*> pure y = pure (fun f : S0 => f y) <*> u0
  which is ill-typed.
...
The 5th term has type   "Select F S0       " which
should be coercible to  "Select F (A -> B)".
```

```
Theorem Select_Applicative_law_Interchange
        `{FunctorLaws F} :
  forall (A B : Set) (u : Select F (A -> B)) (y : A),
  u <*> pure y = pure (fun f => f y) <*> u.
Proof. induction u.
```

```
Error: Abstracting over the terms "S" and "u" leads to a term
fun (S0 : Set) (u0 : Select F S0) =>
u0 <*> pure y = pure (fun f : S0 => f y) <*> u0
  which is ill-typed.
...
The 5th term has type   "Select F S0       " which
should be coercible to  "Select F (A -> B)".
```

`S0` is a non-uniform index and this affects the generated induction principle for `Select`

```
Select_ind :
  forall (F : Type -> Type)
         (P : forall A : Set, Select F A -> Prop),
  (forall (A : Set) (a : A), P A (Pure F A a)) ->
  (forall (A B : Set) (s : Select F (B + A)),
     P (B + A) s -> forall f0 : F (B -> A),
       P A (MkSelect F A B s f0)) ->
  forall (A : Set) (s : Select F A), P A s
```

The predicate is generalised in A

```
Select_ind :
  forall (F : Type -> Type)
         (P : forall A : Set, Select F A -> Prop),
  (forall (A : Set) (a : A), P A (Pure F A a)) ->
  (forall (A B : Set) (s : Select F (B + A)),
     P (B + A) s -> forall f0 : F (B -> A),
       P A (MkSelect F A B s f0)) ->
  forall (A : Set) (s : Select F A), P A s
```

The predicate is generalised in A

It is not yet clear to me what to do with it…

Challenges of formalising Haskell concepts

- Non-structurally recursive functions (kinda solved)
- Non-uniform indices cause problems with `induction` (not solved)

Challenges of formalising Haskell concepts

- Non-structurally recursive functions (kinda solved)
- Non-uniform indices cause problems with `induction` (not solved)

Ways to go from here

- Relax the notion of equality? Work up to isomorphism?
- Come up with a different formulation of Free Selective and see if it's better for proofs?

Links

- Free Selective Functors in Coq: https://github.com/tuura/selective-theory-coq
- Selective functors in Haskell: https://github.com/snowleopard/selective
- Li-yao XIA's study of proofs for Free Applicative Functors: https://blog.poisson.chat/posts/2019-07-14-free-applicative-functors.html