A Computational Interpretation of Higher Inductive Types: How to Tame Equality in your Types

Jason Gross JGROSS@CSAIL.MIT.EDU
MIT Computer Science and Artificial Intelligence Laboratory, 32 Vassar Street, Cambridge MA, 02139 USA

1. Background

An inductive data type is one generated by its constructors. For example,

```
data ℕ where
  0 : ℕ
  S : ℕ → ℕ
```

is a typical inductive data type, whose computable inhabitants are 0, S 0, S (S 0), S (S (S 0)), ... Fully applied constructors must construct an inhabitant of the inductive data type, and we require some basic positivity or termination constraints.

Sometimes, this isn't enough. A `set` isn't the same thing as a list, nor is it the same thing as a priority queue, though languages like Coq might lead us to think so. There are currently two options for encoding concepts like unordered lists in Coq: one can assume lots of axioms, which results in computational problems; or one can use setoids, which force the programmer to include lots of boiler-plate proofs in many places.

2. Higher Inductive Types

It would be really nice if we could just assert that permutations of a list are equal, and be done with it. Higher inductive types let us do precisely that. If `IsPermutation` ℓ_1 ℓ_2 is a proposition expressing the assertion that ℓ_1 is a permutation of ℓ_2 (and we had higher inductive types), then we could write

```
data unordered_list (T : Type) where
  disorder : list T → unordered_list T
  are_disordered : ∀ ℓ1 ℓ2,
          IsPermutation ℓ1 ℓ2
          → disorder ℓ1 = disorder ℓ2
  contr1 : ∀ (ℓ1 ℓ2 : unordered_list T)
             (pf1 pf2 : ℓ1 = ℓ2),
               pf1 = pf2
```

3. Computation

This is trivial, and obvious, right? The goal is building a strongly normalizing type theory with confluent reduction rules around this. That is, we don't want any of our constructions to get stuck on `are_disordered` when they shouldn't, and we certainly don't want to be able to exploid `are_disordered` to write programs that segfault or crash.

It may seem obvious how to write a computation rule for higher inductive types. Here is an example to make it seem less obvious. We prove functional extensionality, following (Shulman, 2011), which causes many problems in intensional type theory, without axioms. We can define an "interval" type with two points and a path (equality) between them:

```
data interval where
  zero : interval
  one  : interval
  seg  : zero = one
```

Now, given two functions `f` and `g` of type `∀ x : A, B x`, and a proof `H` that `∀ x, f x = g x`, we can construct the function

```
funext_helper : interval → (x : A) → B x
funext_helper zero x = f x
funext_helper one  x = g x
funext_helper seg  x = H x
```

Using the fact that `x = y → f x = f y` for all functions `f`, we can get `funext_helper zero = funext_helper one` because `seg : zero = one`. Thus we obtain that `f = g`, proving functional extensionality. So any computational interpretation of higher inductive types needs to give a computational account of functional extensionality, possibly via something like Observational Type Theory (Altenkirch et al., 2007).

References

Altenkirch, T., McBride, C., & Swierstra, W. (2007). Observational equality, now! *Proceedings of the 2007 workshop on Programming languages meets program verification* (pp. 57–68).

Shulman, M. (2011). An interval type implies function extensionality. `http://homotopytypetheory.org/2011/04/04/an-interval-type-implies-function-extensionality/`.