Comments on "Formalising Real Numbers in Homotopy Type Theory" + "Partiality, Revisited"

Alexander Grabanski - ajg137@case.edu

July 24, 2017

When the HoTT book was written in 2013, there was no formalization of the construction of the Cauchy reals in Chapter 11 in Coq nor in Agda. This situation was remedied in 2016 by Gaëtan Gilbert (GitHub username SkySkimmer `https://github.com/SkySkimmer`) `https://arxiv.org/pdf/1610.05072.pdf`, and in the process of formalization, they managed to generalize the construction of the Cauchy reals to the Cauchy completion of any "premetric space." They also managed to prove that in the category of such spaces with Lipschitz maps as morphisms, Cauchy completion forms a monad, which gives the completion a somewhat nice computational structure.

Simplifying and generalizing the existing proofs in the HoTT book would have been publishable enough, but Gilbert went further, and defined a function out of the Cauchy reals $isPositive$ which partially decides the sign of a number (with non-termination on zero). To do this, he made use of a higher inductive-inductive construction described in "Partiality, Revisited" `https://arxiv.org/abs/1610.09254` which expresses possibly-nonterminating computations. In a very strong sense, a semi-decision procedure for the positiveness of a real number is the best we could hope for, because a decision procedure would solve the halting problem. This is simply because if we had a $T :\equiv (TuringMachine, Input)$ pair, and we had a function $H : \mathbb{N} \to \mathbb{Q}$ such that $H(n) = 1$ if T has already halted at step n, and 0 otherwise, we could express the real number:

$$S = \sum_{k=0}^{\infty} 2^{-k} H(k)$$

which would be equal to 0 if and only if T halts, so if we could decide equality of a real number with 0, we could decide the halting problem, an impossibility.

Since monads play a role both in the formalization of some results on the Cauchy reals and in the results from "Partiality, Revisited", I'll begin by describing what monads are, both from the perspective of category theory and from the perspective of their practical everyday usage in the programming language Haskell, with some examples. Then, I'll cover Gaëtan's formalization up to the "Cauchy completion forms a monad in the category of premetric spaces with Lipschitz maps" result. From there, I'll describe the results about partial functions in "Partiality, Revisited" to lead up to the partial decision procedure $isPositive$.

1 Monads

1.1 In the Context of Programming

Suppose that you're a computer programmer writing a computer algebra system in your favorite functional programming language. You're chugging along, writing some basic solution tactics, like one for solving linear systems, possibly with some foreknowledge of variables' values:

$$linsolve : EquationSet \to VariableAssignmentSet \to VariableAssignmentSet$$

which work fine and dandy, so you release version 1.0 so you can attract some angel investors in your company. However, once your application garners the attention of around three users, one of them asks you to implement a quadratic solver, because it's $\$Current\_Year\$$, not 1000BC. So, you decide to try to implement this new routine, and you give it a type signature:

$$quadsolve : EquationSet \to VariableAssignmentSet \to VariableAssignmentSet$$

However, your $VariableAssignmentSet$ only expresses a single solution to a system of equations. So this routine actually needs the type:

$$quadsolve : EquationSet \to VariableAssignmentSet \to List(VariableAssignmentSet)$$

But then, you see that to form the step-by-step sequences of solving tactics that your users expect, you'd really want to give all of your tactics the signature

$$EquationSet \to List(VariableAssignmentSet) \to List(VariableAssignmentSet)$$

so that they're composable.

Changing this would seem to require going back and making some really mind-numbing revisions to your old routines, which now number in the hundreds. You request a few weeks' more time to perform this refactor for version 2.0.

After manually converting 10 or so routines to loop over an input $List(VariableAssignmentSet)$ and apply their namesake solving tactic to each of them using a map, you get really, really bored. It would be nice to be able to just leave the signature $EquationSet \to VariableAssignmentSet \to VariableAssignmentSet$ on most routines, and pull out an $EquationSet \to VariableAssignmentSet \to List(VariableAssignmentSet)$ signature only when it's truly needed.

You try to simplify the problem by temporarily ignoring the first parameter, and reducing everything to just a few letters. You have functions $V \to V$ which you'd like to lift to $List(V) \to List(V)$ automatically using a map, but you'd also like to lift functions $V \to List(V)$ to functions $List(V) \to List(V)$ which concatenate the solution-sets. So you define:

$$liftList : (V \to List(V)) \to (List(V) \to List(V))$$

$$listList(f, L) :\equiv reduce([], concat, map(f, L))$$

Then, you'd like to be able to define an operations pipeline, which starts from a single (possibly-empty) variable-assignment context and terminates in a collection of possible solutions. At each stage, you'd like to be able to chain together operations using something of the signature:

$$>>=: List(V) \to (V \to List(V)) \to List(V)$$

which you call "bind", and you'd like to be able to inject singleton lists to get the process started, using

$$return : V \to List(V)$$

$$return(a) :\equiv Cons(a, Nil)$$

Now, note that the type of $>>=$ is nearly the same as our $liftList$, but with the first two arguments out-of-order, so we can just let:

$$>>= (x, f) :\equiv liftList(f, x)$$

Then, if you had a quadratic equation set E whose solution pinned down a parameter used in quadratic equation set F, you are happy that you can now handle this situation using:

$$(return(InitialValues) >>= quadsolve(E)) >>= quadSolve(F)$$

because the binding operator $>>=$ automatically keeps track of the possibility of multiple solution sets, with minimal work on your part.

Then, you read up on the Haskell List monad, and find that everything you've done has already been invented, and in a far more general form. Instead of $List$, a general monad in Haskell `https://wiki.haskell.org/Monad` uses some arbitrary type constructor M, which leads to the typing judgments:

$$>>=: A \to (A \to M(B)) \to M(B)$$

$$return : A \to M(A)$$

There are also some basic *Monad Laws* governing the behavior of these two operations in general. The first one listed,

$$return(a) >>= k \equiv k(a)$$

states that pipelining in a $return$-ed value is exactly the same as directly applying the whole chain to the value, meaning that our example above could have been written:

$$quadsolve(E)(InitialValues) >>= quadSolve(F)$$

The second law,

$$m >>= return \equiv m$$

merely states that $return$ acts as the identity when we treat it as if it were something of type $M(A) \to M(A)$ under the hood. The final law has a more complex expression:

$$m >>= (x \mapsto (k(x) >>= h)) \equiv (m >>= k) >>= h$$

but really, all it says is that pipelining is associative, with the lambda-abstraction thrown in there to make the type signatures fit together.

There are many more useful examples of Monads in Haskell, like the IO Monad, which allows operations which interact with the "real world", the ST Monad, which allows carrying around readable/writable state with the computational context, and the Maybe Monad, which expresses the possibility of failing computations. What all of these have in common is that they represent a way to form pipelines of operations with some special extra capabilities, like the capability to chain functions with multiple return values as seen in the $List$ monad. At the same time, Monads let us isolate these effects by forcing them into a sub-universe of types in the image of the $M(-)$ functor.

1.2 In a Categorical Context

The central thing we seemed to do to make the example of a monad work in the previous section was to define the function $liftList$, which internally used the fact that we can flatten a list of lists into a list. Generalizing to the case of M the type constructor of a monad, we could expect to need an operation μ_A for every type A which takes $M(M(A)) \to M(A)$. Then, noting that M, in the example above, is a functor (in the HoTT sense) from $\mathcal{U} \to \mathcal{U}$, we could generalize to an arbitrary category C and simply demand that $M : C \to C$ is a functor. Note that the types we had before become objects in C. From there, we can see that we want μ to be a natural transformation : $M \circ M \to M$, meaning that we should have:

$$\begin{array}{ccc} M(M(A)) & \xrightarrow{M(M(f))} & M(M(B)) \\ \downarrow{\scriptstyle \mu_A} & & \downarrow{\scriptstyle \mu_B} \\ M(A) & \xrightarrow[M(f)]{} & M(B) \end{array}$$

Which we can justify in the $List$ example by noting that the operations of mapping an element-wise operation over (1d/2d) lists and flattening to a one-dimensional list should commute.

Then, we can pick up $return$ as a natural transformation from the identity functor on C to M, call it $\eta : 1_C \to M$, since naturality in this case just demands that η successfully embeds computations performed outside of M, which will turn out to correspond to the first monad law. Graphically,

$$\begin{array}{ccc} A & \xrightarrow{f} & B \\ \downarrow{\scriptstyle \eta_A} & & \downarrow{\scriptstyle \eta_B} \\ M(A) & \xrightarrow[M(f)]{} & M(B) \end{array}$$

Then, to make the whole construction satisfy the other monad laws, we demand two coherence conditions. The first states that when we apply μ, it doesn't matter if we apply μ to the outer or the inner two nested Ms in $M(M(M(X)))$ to yield a $M(M(X))$ if we're just gonna reduce that with μ to $M(X)$. This will correspond to the third monad law (associativity) after we identify what $>>=$ is. Graphically, this coherence condition is:

$$\begin{array}{ccc} M(M(M(X))) & \xrightarrow{M(\mu_X)} & M(M(X)) \\ \Big\downarrow{\scriptstyle \mu_M(X)} & & \Big\downarrow{\scriptstyle \mu_X} \\ M(M(X)) & \xrightarrow[\mu_X]{} & M(X) \end{array}$$

Finally, we need a coherence corresponding to the second monad law, which states that $return$ (or η, in our terminology) is the identity with respect to $>>=$. Here, we'll finally reveal what $>>=$ is: Suppose that we have an $f : A \to M(B)$ which we wish to use as a computation in a binding pipeline. Then, $M(f) : M(A) \to M(M(B))$, which would be exactly what we'd need to yield a $M(A) \to M(B)$ transformation in the pipeline, so long as we squash the double-application of M using μ. So we may define:

$$a >>= f :\equiv \mu_B(M(f))(a)$$

Then, the second monad law translates to:

$$\mu_X \circ M(\eta_X) \equiv id_{M(X)}$$

and we might as well demand identity on the other side as well (which is derivable in the non-categorical context from the first monad law on the identity function in the previous section):

$$\mu_X \circ \eta_{M(X)} \equiv id_{M(X)}$$

This firmly establishes the correspondence between the two notions of a monad.

2 The Cauchy Completion Monad

2.1 Constructing the Cauchy Completion

Gilbert's formalization of the Cauchy reals begins by describing "premetric spaces", which consist of pairs $(A : \mathcal{U}, \approx\colon \mathbb{Q}_+ \times A \times A \to Prop)$ such that the relation $\approx$ is reflexive, symmetric, separated, triangular, and rounded. All of these properties have the same definitions as seen in Chapter 11 of the HoTT book, and the Cauchy reals together with $\sim_\epsilon$ satisfy them (as shown in Theorem 11.3.16 of the HoTT book). However, the proof sketch in the HoTT book was a long and irritating nested induction taking up several pages.

$\mathbb{Q}$ together with $x \approx_\epsilon y :\equiv |x - y| < \epsilon$ clearly forms a premetric space, and their initial Cauchy completion yields the Cauchy reals, so it would be nice to be able to show that the Cauchy completion of a premetric space is a premetric space. First, Gilbert defines the

Cauchy completion $\mathcal{C}T$ of a premetric space T in analogy with the HoTT's book completion of the rationals, with $\mathcal{C}T$ replacing $\mathbb{R}_C$ everywhere and T replacing $\mathbb{Q}$ everywhere, including in the definition of $CauchyApx$. For example, the point constructors become:

$$\eta : T \to \mathcal{C}T$$
$$lim : CauchyApx \to \mathcal{C}T$$

Where here, η denotes the replacement for the constructor rat. All of the higher constructors and constructors of $\sim_\epsilon$ stay the same, with the obvious exception that the constructor in the Cauchy reals of type:

$$\prod_{q,r:\mathbb{Q}} \prod_{\epsilon:\mathbb{Q}_+} (|q-r| < \epsilon) \to rat(q) \sim_\epsilon rat(r)$$

Needs to not rely on properties particular to the rationals, and instead use the premetric structure, so we get a constructor of type:

$$\prod_{q,r:T} \prod_{\epsilon:\mathbb{Q}_+} (q \sim_\epsilon r) \to \eta(q) \sim_\epsilon \eta(r)$$

From there, the induction and recursion principles of the Cauchy completion are very similar to the HoTT principles, and so the proofs of the reflexivity and symmetry of $\sim_\epsilon$ and that $\mathcal{C}T$ is a set for any T go through.

2.2 Characterizing Closeness

To get the characterization of $\sim_\epsilon$, Gilbert makes a few definitions.

First, an upper cut is a predicate on $\mathbb{Q}_+$ somewhat analogous to the upper set of a (two-sided) Dedekind cut in that it is rounded:

$$UpperCut :\equiv \sum_{U:\mathbb{Q}_+ \to Prop} isUpperCut(U)$$

$$isUpperCut(U) :\equiv (\forall \epsilon : \mathbb{Q}_+ \quad (U(\epsilon) \leftrightarrow \exists \delta < \epsilon : \mathbb{Q}_+ \quad U(\delta)))$$

If we were dealing with $\mathbb{R}_+$ instead of $\mathbb{Q}_+$, this would say that $UpperCuts$ are open intervals $(x, +\infty)$.

Now, since the ultimate goal is to characterize $\sim_\epsilon$, we could simplify the problem somewhat by fixing a point x and considering all y, ϵ such that $x \sim_\epsilon y$. These will be in the form of "balls centered at x". Intuitively, we expect that if we keep y fixed as well, the set of all ϵ satisfying this should be upward-closed, since if we think of balls of radius r centered on x, any and all $r > d(x, y)$ will do. Due to use of a metric in the construction of a "ball centered at x", we should also expect to have a version of the triangle inequality in that we can always enlarge a ball containing y to also contain a close point z. These conditions put together yield the definition:

$$Balls :\equiv \sum_{B:\mathcal{C}T \to \mathbb{Q}_+ \to Prop} (\forall y : \mathcal{C}T \quad isUpperCut(B(y)))$$

$$\wedge(\forall \epsilon, \delta : \mathbb{Q}_+, y, z : \mathcal{C}T \quad y \sim_\epsilon z \to B(y)(\delta) \to B(z)(\epsilon + \delta))$$

From there, a notion of closeness of upper cuts is defined, which states that two upper cuts U_1, U_2 are ϵ-close if any x in U_1 is such that $x + \epsilon$ is in U_2, and vice-versa. Then, two $Balls$ B_1, B_2 are ϵ-close if their corresponding upper cuts at y, $B_1(y)$ and $B_2(y)$ are ϵ-close for every y.

It's straightforward that the closeness relation on upper cuts is separated (and hence on balls), since if e.g. U_1 is ϵ-close to U_2 for every ϵ, then we can show that if x is in U_1, by roundedness, there's merely a $y < x$ in U_1, but then, since U_1 and U_2 are $(x - y)$-close, the definition of closeness yields that x is also in U_2 (a mere proposition).

Now, it's clear that every $B : Ball$ defines a collection of upper cuts $\{B(y)|y : \mathcal{C}T\}$, but in fact, any non-expanding function $f : \mathcal{C}T \to UpperCut$ yields an element of $Balls$. All that needs to be done to prove this is to show that the analogue of the triangle inequality holds, but non-expansion is good enough to guarantee this, since if y, z are ϵ-close, $F(y) \sim_\epsilon F(z)$ (here and elsewhere (for convenience) F is $pr_1 \circ f$) , and so in $F(y)(\delta) \to F(z)(\epsilon + \delta)$ due to the definition of closeness on $UpperCut$.

2.2.1 Families of Concentric Balls Centered At a Point

Then, in particular, for any $x : \mathcal{C}T$ which is $x \equiv \eta(q)$, it's possible to define the family of concentric balls at x by defining such a non-expanding function $f : \mathcal{C}T \to UpperCut$ by $(\mathcal{C}T, \sim)$-recursion. To do so, the $\sim$ part of the induction has the burden of showing that f is non-expanding. On T, f is defined by:

$$f(\eta(z)) :\equiv (\epsilon \mapsto x \sim_\epsilon z, \_)$$

where the blank contains a proof of the roundedness of the premetric on T, since that condition has the form:

$$\forall q : T, \epsilon : \mathbb{Q}_+, z : T \quad q \sim_\epsilon z \leftrightarrow (\exists \delta : \mathbb{Q}_+ \quad \delta < \epsilon \wedge q \sim_\delta z)$$

So taking q and z to be the ones from before, and identifying U with the upper cuts we're defining by $q \sim_\_ z$, this is:

$$\forall \epsilon : \mathbb{Q}_+ \quad U(\epsilon) \leftrightarrow (\exists \delta : \mathbb{Q}_+ \quad \delta < \epsilon \wedge U(\delta))$$

which is exactly a proof of cut roundedness.

For the limit case, let

$$f(lim(z)) :\equiv (\epsilon \mapsto \exists \delta < \epsilon \quad F(z_\delta)(\epsilon - \delta), \_)$$

Where the blank now contains the following proof of the roundedness of the upper cut: We need to show that

$$F(lim(z))(\epsilon) \leftrightarrow \exists \bar{\epsilon} < \epsilon \quad F(lim(z))(\bar{\epsilon})$$

which, by definition, means we need to show

$$(\exists \delta < \epsilon \quad F(z_\delta)(\epsilon - \delta)) \leftrightarrow (\exists \bar{\epsilon} < \epsilon \quad \exists \delta < \bar{\epsilon} \quad F(z_\delta)(\bar{\epsilon} - \delta))$$

But by the inductive hypothesis, we know that the upper cuts for each $F(z_\delta)$ are rounded, so the left-hand side is logically equivalent to:

$$\exists \delta < \epsilon \quad \exists \zeta < (\epsilon - \delta) \quad F(z_\delta)(\zeta)$$

whence we get the desired result equivalence by rearranging existentials and identifying

$$\zeta = \bar{\epsilon} - \delta$$

Then, for the $\sim$ part of the recursion, we'll take the closeness on $Upper$ to be the target closeness relation, and so satisfying the hypotheses of the $\sim$-recursion will directly show that the map is non-expanding. Gilbert's formalization proves all cases combining η-injected elements and limits, and the paper outlines what's needed for the proof of the $\eta - lim$ case. I'll show the $\eta - \eta$ case here, to give a flavor for what's going on:

We need to show that if we have $q, r : T$ such that $q \sim_\epsilon r$, then as upper cuts, $f(\eta(q)) \sim_\epsilon f(\eta(r))$. Now, by definition, we have that $f(\eta(q)) = \delta_1 \mapsto x \sim_{\delta_1} q$, and we also have that $f(\eta(r)) = \delta_2 \mapsto x \sim_{\delta_2} r$. Now, fix some arbitrary δ_1 in $f(\eta(q))$. We need to show that $\delta_1 + \epsilon$ is in $f(\eta(r))$, so we need to show that:

$$x \sim_{\delta_1 + \epsilon} r$$

But we know that $q \sim_\epsilon r$ and $x \sim_{\delta_1} q$, so we can get that from the triangle inequality.

After all the cases have been proved, we still have the burden of going back and also defining the case where x is not $\eta(q)$, but a limit. This is done in full in Gilbert's formalization. In total, the proof will wind up having 8 cases for the $\sim$ relation, and 4 for the base definitions, but this is better than Theorem 11.3.16's proof in the HoTT book, which required 16 closeness-relation proof cases (8 of which were omitted from the text). From all of this, we get a rather important result.

2.2.2 "Is contained in an ϵ-neighborhood of" concides with "ϵ-close"

In particular, fixing x (here taken to be $\equiv \eta(q)$) and defining f, F just like in the previous section, we have:

$$F(y)(\epsilon) \leftrightarrow x \sim_\epsilon y$$

The proof of this fact boils down to how we defined f for each case. The η case holds by definition, and in the limit case, we can use the $\sim$-constructor

$$\prod_{q:T} \prod_{y:CauchyApx} \prod_{\epsilon,\delta:\mathbb{Q}_+} (\eta(q) \sim_{\epsilon-\delta} y_\delta) \to (\eta(q) \sim_\epsilon lim(y))$$

to extract an equivalence straight from the induction hypothesis and the definition of $f(lim(z))$, which was:

$$f(lim(z)) :\equiv (\epsilon \mapsto \exists \delta < \epsilon \quad F(z_\delta)(\epsilon - \delta), \_)$$

So, from this, we get that $\sim$ is rounded and satisfies the triangle inequality, meaning that $\mathcal{C}T$ is a premetric space. From that, we show fairly simply (using the same standard arguments in HoTT, in this different context) that $\mathcal{C}T$ is Cauchy-complete.

Using this characterization can also show that continuous functions which agree on η-injected elements are equal, among many other important basic results in the theory of metric space completions.

2.3 Lipschitz Maps

Just like in the case of the Cauchy reals, we can define the notion of a Lipschitz map, and prove that Lipschitz maps can be lifted to Cauchy completions. The proof of this fact is a direct adaptation of the one in HoTT, but here's the statement:

If we have a function $f : T \to A$ from a premetric space T to a Cauchy-complete premetric space A, where f is *Lipschitz with constant* L:

$$\forall \epsilon : \mathbb{Q}_+, x, y : T \quad x \sim_\epsilon y \to f(x) \sim_{L*\epsilon} f(y)$$

then f has a lift $\bar{f} : \mathcal{C}T \to A$ which agrees with f on η-injected elements.

2.4 One Neat Trick!

One thing that'd be nice to know is that when we take the Cauchy completion of $\mathcal{C}T$, we "don't pick up anything new", meaning that the result should be equal as a type to $\mathcal{C}T$. How do we do this? Well, $id_{\mathcal{C}T}$ is a Lipschitz function (constant 1), and so it has a lift to $\bar{id_{\mathcal{C}T}} : \mathcal{C}\mathcal{C}T \to \mathcal{C}T$. We know that $\eta_T \circ \bar{id_{\mathcal{C}T}}$ is the identity, since we only need to show that it's the identity on η-injected elements by continuity. We also know that $\bar{id_{\mathcal{C}T}} \circ \eta_T$ is the identity by definition. Hence, η_T and $\bar{id_{\mathcal{C}T}}$ are mutual quasi-inverses, and so $\mathcal{C}\mathcal{C}T = \mathcal{C}T$.

This should look suspiciously familiar to the definition of μ in our discussion about monads above. In fact, the Cauchy completion is a monad in the category of pre-metric spaces with Lipschitz functions as morphisms! Let's check this: The η of the monad needs to assign a function of type $T \to \mathcal{C}T$ on every pre-metric space T. We get this by η in the construction of the Cauchy completion. $\mu \equiv \mathcal{C}$ must be natural, but it's idempotent, so this holds. The first coherence condition also is satisfied by definition of μ as a lift of the identity, as are the other coherence conditions (the unit laws).

Consequently, we can imagine having a domain-specific language (embedded in a monad) where the user is free to compose chains of Lipschitz functions and automatically lift the whole chain to the Cauchy completion.

3 Partiality

The Boolean type $\mathbf{2}$ is the target of any decision procedure ($: A \to \mathbf{2}$) on type A. In MLTT, the type $\mathbf{2}$ (sensibly) has exactly two inhabitants (up to equality): 0_2 and 1_2. However, in the programming language Haskell (mentioned earlier), this is not the case: $Bool$ has *three* inhabitants: 0_2, 1_2 and $\bot$, where we can get $\bot$ through any of the following equivalent definitions:

$$\bot = \bot$$

$$\bot = \quad let \quad f\,x = not\,(f\,x) \quad in \quad f(False)$$

$$\bot = undefined$$

That is, $\bot$ expresses both non-terminating and error states.

Most of the time, Haskell programmers will ignore this extra inhabitant of the Boolean type to facilitate equational reasoning, but it actually changes the meaning of the type

$A \to Bool$ to become a type of *partial* decision procedures. Functions of that type will only be a decision procedure on a subset of the values of A, and the particular subset on which it is defined may not be easy to express. Nevertheless, partial functions are often useful to express cases where we truly do not know an answer, because it may be impossible to in the general case (like the situation with the Halting Problem). The $\bot$ value is a direct consequence of Haskell's support for general recursive functions combined with its lazy reduction semantics.

However, the situation in MLTT is radically different, since every expression in MLTT is reducible to a normal form. This is due to the restriction of recursive functions to those definable by structural induction. For example, in Haskell, it's perfectly fine to define a function $f : \mathbb{N} \to \mathbb{N}$ where $f(n)$ can depend explicitly on n and $f(n+2)$, but in MLTT, it's only permissible to define $f(succ(n))$ using n and $f(n)$. It would seem that with this restriction in place, defining partial functions in HoTT is not possible.

However, we could also look at the inhabitant $\bot$ as representing a sort of completed infinity, since it's a value of $Bool$ which doesn't appear in a finite number of (internally-visible) steps. Instead, we could represent $\bot$ in a way that makes it into a potential infinity: instead of inhabitants of $\mathbf{2}$, we could consider functions $f : \mathbb{N} \to B$ for some suitable definition of B which gives f an interpretation as a function from a number of steps executed so far to the current computational state. One way to formalize this is through Capretta's *delay monad*, which is most naturally expressed as a co-inductive type, but since co-induction isn't native to most formulations of HoTT, we could take B to be $\mathbf{1} + \mathbf{2}$ where inL values represent computations in progress, and inR values represent finished computations (with a corresponding restriction on f to stabilize at an inR-injected value once this happens). `http://www.cs.nott.ac.uk/~psznk/docs/nicolai_uppsala_handoutversion.pdf` However, there's a problem with this definition – there's a lot more than three different such fs, up to equality! In particular, the operation "delay the computation by a step" yields infinitely many distinct values, instead of the simpler collection $0_2, 1_2, \bot$. It would seem that we need some kind of type quotient here, and HoTT might provide the proper tools to obtain it.

Something akin to the above remarks, along with the realization that the "stabilization" requirement on f is most naturally expressed by formulating a partial order, lead the authors of Partiality, Revised to formulate the following higher inductive-inductive type $A_\bot$ for partial computations of type A, together with the antisymmetric, reflexive, transitive mere relation $\sqsubseteq$:

$A_\bot$:

$$\eta : A \to A_\bot$$

$$\bot : A_\bot$$

$$sup \equiv \sqcup : IncrSeq \to A_\bot$$

$$\alpha : \prod_{x,y:A_\bot} x \sqsubseteq y \to y \sqsubseteq x \to x =_{A_\bot} y$$

$\sqsubseteq$:

$$x \sqsubseteq x$$

$$x \sqsubseteq y \wedge y \sqsubseteq z \to x \sqsubseteq z$$

$$\bot \sqsubseteq x$$

$$\prod_{s:IncrSeq} \prod_{n:\mathbb{N}} (s_n \sqsubseteq sup(s))$$

$$\prod_{s:IncrSeq} (\prod_{n:\mathbb{N}} s_n \sqsubseteq x) \to sup(s) \sqsubseteq x$$

$$\prod_{p,q:x \sqsubseteq y} p = q$$

Where in the above, $x, y, z : A_\bot$ wherever it's not stated explicitly, s_n denotes $pr_1(s)(n)$, and the type of increasing sequences $IncrSeq$ is defined by:

$$IncrSeq :\equiv \sum_{s:\mathbb{N} \to A_\bot} \prod_{n:\mathbb{N}} s(n) \sqsubseteq s(succ(n))$$

Now, since $\prod_{x,y:A_\bot} (x \sqsubseteq y) \wedge (y \sqsubseteq x)$ is a mere relation on $A_\bot$, α and Theorem 7.2.2 from HoTT imply that $A_\bot$ is a set. Due to the definition of $\sqsubseteq$, it's also clear that $\sqsubseteq$ is a partial order. But also, in analogy with the situation with the Cauchy completion, the sup constructor will also imply that the partial order is ω-*complete*, meaning that every increasing countable chain

$$s_0 \sqsubseteq s_1 \sqsubseteq s_2 \sqsubseteq ...$$

has a supremum (least upper bound). The "is an upper bound" part comes from the third-to-last constructor of $\sqsubseteq$, and the "is the least upper bound" part is derivable from the second-to-last constructor of $\sqsubseteq$.

Now, from this, we can quickly devise some examples of functions targeting $\mathbf{2}_\bot$. If we have a (partial decision problem, input) pair, and a function:

$$stateAt : \mathbb{N} \to (\mathbf{1} + \mathbf{2})$$

which represents the state (either inL for "in operation", or inR for a halted state) of the computation after n steps, then we can define a sequence

$$s(n) :\equiv rec_{\mathbf{1}+\mathbf{2}}(\mathbf{2}_\bot, \_ \mapsto \bot, x \mapsto \eta(x))(stateAt(n))$$

which is increasing so long as we can show that $stateAt$ starts at $\bot$ and stabilizes at a value upon halting, so we can use

$$sup((s, p))$$

to represent the result of the computation. p the derived proof that s is increasing.

Just like any other HIIT, $(A_\bot, \sqsubseteq)$ comes with an induction principle. However, not counting the propositional truncation on $\sqsubseteq$, this induction principle would involve proving nine different constructor-based cases for every application. Luckily, unlike some other HIITs (like the Cauchy completion), specializing the relation over $\sqsubseteq$ to be $\mathbf{1}$ and forcing the function on $A_\bot$ to target a proposition yields a widely-applicable *partiality induction principle* (Lemma 3 in Partiality, Revised):

Partiality Induction

If P : $A_\perp \to Prop$, and we can show P for $\perp$, every η-injected element, and every supremum of elements of $A_\perp$ given that they all satisfy P, then $\forall x : A_\perp \quad P(x)$.

With a catch: to be able to use this to prove useful results, we need a characterization of $\sqsubseteq$ similar to what was done for the characterization of $\sim_\epsilon$ for the Cauchy completion. In the formalization corresponding to Partiality, Revised, the following (partial) characterization is proved by full-blown $(A_\perp, \sqsubseteq)$-induction (Lemma 7):

$$\eta(a) \sqsubseteq \perp \leftrightarrow \mathbf{0}$$
$$\eta(a) \sqsubseteq \eta(b) \leftrightarrow a = b$$
$$\eta(a) \sqsubseteq sup(s) \leftrightarrow \exists n : \mathbb{N} \quad \eta(a) \sqsubseteq s_n$$

From this, we can prove (importantly) that η is injective (which follows directly from the second line), that every two non-equal, non-bottom elements are incomparable w.r.t. $\sqsubseteq$ (so $\sqsubseteq$ is a "flat" ordering), and that the η-injected elements are the maximal elements of $A_\perp$. The proof of the last fact is a good example of an application of partiality induction.

3.1 η-injected elements are maximal

We want to show that for any $y : A_\perp$, $a : A$, if

$$\eta(a) \sqsubseteq y$$

then

$$y = \eta(a)$$

To do so, from the higher constructor of $A_\perp$, it suffices to prove

$$y \sqsubseteq \eta(a)$$

So we proceed by partiality induction. The $y \equiv \eta(b)$ case follows directly from the injectivity of η, and the $y \equiv \perp$ case follows from the first line in the characterization of $\sqsubseteq$. All that remains is the sup case, where we suppose $y \equiv sup(s)$ and every s_n is such that if we have any $a : A$,

$$\eta(a) \sqsubseteq s_n \to s_n \sqsubseteq \eta(a)$$

Now, suppose that we have some $a : A$ such that

$$\eta(a) \sqsubseteq y$$

. Then, using the third line in the characterization of $\sqsubseteq$, we know that there merely exists an $n : \mathbb{N}$ such that:

$$\eta(a) \sqsubseteq s_n$$

But then, since s_n is increasing, we also know that for this n, every $m \geq n$ is such that

$$\eta(a) \sqsubseteq s_m$$

and so for all such m, by the inductive hypothesis

$$s_m \sqsubseteq \eta(a)$$

But then, everything in the s chain before n is also less than $\eta(a)$, so $\eta(a)$ is an upper bound on s, hence by constructors of $A_\perp$,

$$sup(s) \sqsubseteq \eta(a)$$

which completes the proof.

3.2 $\sqsubseteq$ is flat

The proof that the ordering is flat only appears in the formalization, but it's a a good example of another trick involving (you guessed it) a monad. In particular, there's the *double-negation* monad (see Appendix), which takes types to their double negation. This is the same process as the double-negation translation for embedding classical logic in constructive logic, and so within the double-negation monad, LEM_∞ holds.

We want to show that for any $x, y : A_\perp$

$$y \neq \perp \to x \neq \perp \to x \neq y \to x \not\sqsubseteq y$$

Before showing this, first

3.2.1 In $\neg\neg$, every element of $A_\perp$ is either $\perp$ or merely η-injected

We want to show that

$$\neg\neg(x = \perp + \exists \alpha : A \quad \eta(\alpha) = x)$$

So, proceeding by partiality induction on x, the non-supremum cases obviously imply the result. In the supremum case, $x \equiv sup(s)$ and within $\neg\neg$, every s_n is such that either $s_n = \perp$ or merely $s_n = \eta(\alpha_n)$. Then, by $\neg\neg LEM_\infty$ an analogue of Markov's principle (see Appendix), either all of s_n are $\perp$, in which case $\perp$ is an upper bound, or there merely exists an n for which s_n is merely $\eta(\alpha_n)$. In that case, since s is increasing, and all η-injected elements are maximal, x is merely s_n.

3.2.2 Back to the main task

Now, in the situation we described before, we can take each one of the premises and lift them to the double-negation monad, yielding

$$\neg\neg(y \neq \perp)$$

$$\neg\neg(x \neq \perp)$$

$$\neg\neg(x \neq y)$$

and suppose that

$$\neg\neg(x \sqsubseteq y)$$

from which we want to derive $\mathbf{0}$. From there, note that by the preceding proof, both x and y are either $\perp$ or merely η-injected in $\neg\neg$, but the $\perp$ case for each derives a double-negated $\mathbf{0}$ (equivalent to $\mathbf{0}$) when combined with the hypotheses. On the other hand, if both x and y are η-injected in $\neg\neg$, by the characterization of $\sqsubseteq$, since $x \sqsubseteq y$, $x = y$, which also proves a double-negated $\mathbf{0}$. So $\sqsubseteq$ is a flat order.

As a consequence of this result and the preceding revelations about the structure of $A_\perp$, $A_\perp$ appears to capture

4 Partiality and Topology

With that excursion into the (rather neat!) definition of $A_\perp$ out of the way, let's start merging the two seemingly-different topics. First, an excursion into Topology:

4.1 Sierpinski Space

(For another neat exposition of the first few ideas here, see `http://www.math3ma.com/mathema/2016/10/6/the-sierpinski-space-and-its-special-property`)

In topology, if we consider the discrete topology (everything is an open set) and the indiscrete topology (only $\emptyset$ and A) to be "boring" examples of topologies, the smallest "interesting" example of a topology pops up on the two-element set. Let S be a topological space with underlying set $\mathbf{2}$, where the open sets are:

$$\emptyset, \{1\}, \{0, 1\}$$

That is, the singleton containing 0 is closed, but the singleton containing 1 is open. S is called the *Sierpinski space*.

In the category of topological spaces, this space S has a very interesting property: Consider any continuous map $f : A \to S$. Since f is continuous, all preimages of open sets are open, and so in particular, $f^{-1}(\{1\})$ is open. By itself, this fact is nothing terribly special. However, suppose that we have an open set $B \subseteq A$. Then, if we let $f_B : A \to S$ be the indicator function for the proposition "element $\_\_$ belongs to B", we can see that f is in fact continuous, since the pre-images of $\emptyset, S$ are $\emptyset, A$, respectively, and the pre-image of the singleton containing 1 is B. In other words, we can identify morphisms $f : A \to S$ in Top with open subsets of A.

4.2 Back to $\mathbf{1}_\perp$

Taking inspiration from the identification between morphisms targeting S and open sets, we could try to do something interesting: We know that semidecision procedures on A have type $A \to \mathbf{1}_\perp$, since semi-decision procedures yield possibly-nonterminating computations with values in $\mathbf{1}$. Here, we'll refer to $\eta(*)$ as just $*$ for convenience. What would happen if we attempted to define a topology on A by fiat, taking pre-images of $*$ under semi-decision procedures to be the open subsets of A? Call this attempt at a topological

space A_S. Now, it's clear that the constant functions at $\bot$ and $*$ yield $\emptyset$ and A, respectively, but to show that A_S is a topological space, we still need to verify that the collection of open sets is closed under arbitrary unions and finite intersections.

First, for unions, note that if we have two semidecision procedures $f, g : A \to \mathbf{1}_\bot$, the union of the induced sets in A_S would be the induced set of $a \mapsto f(a) \cup g(a)$ where $\cup : \mathbf{1}_\bot \to \mathbf{1}_\bot \to \mathbf{1}_\bot$ is the least upper bound operator (or "join") on the poset $\mathbf{1}_\bot$, which is definable (Definition 5.5 in Gilbert) by $(\mathbf{1}_\bot, \sqsubseteq)$ recursion as:

$$\bot \cup y :\equiv y$$

$$* \cup y :\equiv *$$

$$sup(s) \cup y :\equiv sup((n \mapsto s_n \cup y, p))$$

where p is a proof that $n \mapsto s_n \cup y$ is increasing, based on the inductive hypothesis that s is increasing and that $- \cup y$ is order-preserving (which is the burden of the [simple] $\sqsubseteq$ part of the recursion to show).

Returning to the computational view of f and g, we can interpret $f(a) \cup g(a)$ computationally as a multi-tasking semi-decision procedure which regularly alternates between performing the computation $f(a)$ and the computation $g(a)$, and halts upon the first "halt" state encountered.

However, we need to support *arbitrary* unions, not just finite ones. Unfortunately, we will not be able to do this in general, but we can get pretty close: Suppose as an extra hypothesis that the space of all semidecision procedures $: A \to \mathbf{1}_\bot$ is *countable*. This is a very reasonable assumption from the POV of computability, since there are only countably many programs for a universal turing machine. Then, we only need to show that A_S is closed under *countable* unions. (Equivalently, this is the same as demanding that the resulting topological space will be second-countable).

To do so, we can leverage the sup constructor of $\mathbf{1}_\bot$ in a fairly straightforward way: Suppose that we have a countable collection of elements $e : \mathbb{N} \to \mathbf{1}_\bot$. Then, define the increasing sequence s by setting s_n to the join of all e_m for $m \leq n$. Then, $sup(s)$ is the least upper bound of all of the e_n. Call this construction $\cup(e)$. Then, if $c : \mathbb{N} \to (A \to \mathbf{1}_\bot)$ is a countable family of semidecision procedures on A, the open set corresponding to $a \mapsto \cup(n \mapsto c(n)(a))$ is the union of the open sets corresponding to each $c(n)$.

Proceeding similarly to the definition of $\cup$, we can define $\cap$ (the "meet" of two elements of $\mathbf{1}_\bot$, also known as their greatest lower bound.) However, unlike $\cup$, $\cap$ does not extend to a countable collection of elements of $\mathbf{1}_\bot$. First, it would seem to be merely *difficult* to define, because the extension of $\cup$ relied on the presence of sup, and we have no inf to work with here. But on further inspection, extending $\cap$ is impossible, since if we could, we could write out all of the semi-decision (a fortiori decision) procedures "$\_$ is a Turing machine which halts in n steps" and then intersect them to solve the Halting Problem.

4.3 A Familiar Topology in Disguise

Okay, so aside from the inspiration it gave for the above remarks on $\mathbf{1}_\bot$, what does the Sierpinski space S have anything to do with $\mathbf{1}_\bot$? Well, consider the type of all semidecision procedures $: \mathbf{1}_\bot \to \mathbf{1}_\bot$ on $\mathbf{1}_\bot$. We can define the constant function at $\bot$, the

constant function at $*$, and $id_{\mathbf{1}_\perp}$, but that's it! We can't define any other such procedures (up to equality), since $(\mathbf{1}_\perp, \sqsubseteq)$-recursion requires preservation of an ordering, and the (desirable) undefinability of $\inf$ ensures that the $\sup$ case won't go through in the case of an order-reversing mapping.

Now, notice that the open sets corresponding to $\_ \mapsto \perp$, $\_ \mapsto *$, and $id_{\mathbf{1}_\perp}$ are $\emptyset, \{\perp, *\}$, and $\{*\}$, respectively. Those are just the open sets of the Sierpinski space S under the identifications $\perp \leftrightarrow 0$, $* \leftrightarrow 1$!

5 Keeping it Real: Semidecision procedures on $\mathbb{R}_C$

Time to extract something practical out of all of the theory above to wrap up the discussion. We want a semi-decision procedure on $(x, q) : \mathbb{R}_C \times \mathbb{Q}$ for the proposition $x < rat(q)$, where $q : \mathbb{Q}$ is arbitrary. In other words (and applying currying), we want something of type:

$$\prod_{x:\mathbb{R}_C} \prod_{q:\mathbb{Q}} \sum_{s:\mathbf{1}_\perp} s = * \leftrightarrow x < rat(q)$$

So that we can get the semi-decision procedure by post-composition with the first projection. We'll do this by $\mathbb{R}_C$-induction. Note that

$$\sum_{s:\mathbf{1}_\perp} s = * \leftrightarrow x < rat(q)$$

is a mere proposition (by results in Chapter 3), and so the $eq_{\mathbb{R}_C}$ step of the induction may be (thankfully) omitted.

In the case that $x :\equiv rat(r)$, note that $x < rat(q) \leftrightarrow r < q$ by an easy lemma established in Gilbert's formalization. Then, since the ordering on $\mathbb{Q}$ is decidable, we can send an element of $r < q$ to $*$ and an element of $\neg(r < q)$ to $\perp$ for the element $s : \mathbf{1}_\perp$. The proof that $s = * \leftrightarrow x < rat(q)$ falls out by construction.

In the case that $x :\equiv lim(y)$ for $y : CauchyApx$ with every y_ϵ carrying an associated semidecision procedure s_ϵ where $s_\epsilon(q) = * \leftrightarrow y_\epsilon < rat(q)$, note first that $\mathbb{Q}_+$ is countable. Since every y_ϵ is ϵ-close to the limit x, to show that $x < rat(q)$, it will suffice to show that $y_\epsilon < rat(q) - \epsilon$ for some ϵ. So if $E : \mathbb{N} \to \mathbb{Q}_+$ is an enumeration of $\mathbb{Q}_+$, let

$$s = \cup(n \mapsto s_\epsilon(E(n)))$$

Then, $s = * \to x < rat(q)$ since by the third line in the characterization of $\sqsubseteq$, there's merely an n for which s_n is $*$. For that s_n, by the inductive hypothesis (and going back along E), there's merely an ϵ such that $y_\epsilon < rat(q - \epsilon)$. But then, $x \sim_\epsilon y_\epsilon$, so $x < rat(q - \epsilon)$.

But also, $x < rat(q) \to s = *$ since if $x < rat(q)$, there's merely an r such that $x < rat(r)$ and $r < q$. But then, setting $\epsilon :\equiv q - r$ yields that $x_\epsilon < rat(r) = rat(q - (q - r)) = rat(q - \epsilon)$, and so by the third line in the characterization of $\sqsubseteq$, $*$ lower-bounds s, and so $s = *$.

As a result, comparisons between elements of $\mathbb{R}_C$ and $\mathbb{Q}$ are semi-decidable. By interleaving two semi-decision procedures for $x < rat(0)$ and $-x < rat(-0)$, Gilbert manages to also demonstrate a partial function $isPositive : \mathbb{R}_C \to \mathbf{2}_\perp$ which returns $\eta(0_2)$ for negative numbers, $\eta(1_2)$ for positive numbers, and $\perp$ for zero.

Tying this back to the earlier musings on topology, it's somewhat interesting to note that since $\mathbb{R}$ is second-countable, we could consider a revised notion of topological spaces which weakens the demand for arbitrary unions to countable ones, and still be able to obtain the usual topology on $\mathbb{R}$. Then, what we have just shown (using the Sierpinski space, and the fact that intervals with rational endpoints form a base for $\mathbb{R}$) is that the topology generated by semi-decision processes on $\mathbb{R}_C$ is at least as fine as its usual topology.

6 Appendix: $\neg\neg$ is a monad

The usage of the $\neg\neg$ monad was demonstrated earlier in the proof that $\sqsubseteq$ is a flat ordering, but here's a demonstration that it actually is a monad:

First, let the monadic type constructor M be $M(-) \equiv \neg\neg-$. We need to define what M does to morphisms $f : A \to B$ to give it a structure as an endofunctor, so note that we need $M(f)$ to be of type $((A \to 0) \to 0) \to (B \to 0) \to 0$. Noting that we can precompose the second argument with f and then pass the result to the first argument to get a zero, we can define:

$$M(f)(m_A)(L_B) :\equiv m_A(L_B \circ f)$$

Then, $M(id_A)$ is $id_{\neg\neg A}$ by direct calculation, and we can also verify that it preserves composition of morphisms, since

$$M(f \circ g)(a)(b) \equiv a(b \circ f \circ g)$$

and

$$M(f) \circ M(g) \equiv (a \mapsto (b \mapsto a(b \circ f))) \circ (c \mapsto (d \mapsto c(d \circ g)))$$
$$\equiv c \mapsto (b \mapsto c(b \circ f \circ g))$$

which is equivalent to $M(f \circ g)$ upon α-renaming and η-extensionality.

Then, for $\eta_A : A \to \neg\neg A$, using a function devised in Chapter 1 exercises,

$$\eta_A(a)(f) :\equiv f(a)$$

To show η is natural, we need

$$\eta_B \circ f = M(f) \circ \eta_A$$

Now,

$$(\eta_B \circ f)(a) \equiv g \mapsto g(f(a))$$
$$(M(f) \circ \eta_A)(a) \equiv L_B \mapsto L_B \circ f(a)$$

so it is.

From there, the natural transformation $\mu : M \circ M \to M$ is derivable as:

$$\mu_A(f)(a) :\equiv f(\eta_A(a))$$

since we need $\mu_A : \neg\neg\neg\neg A \to \neg\neg A$, and if $f : \neg\neg\neg A \to 0$, $a : \neg A$, we can use η to push a double-negation in front of a, then apply f. The naturality of μ follows from the naturality

of η, and so do all of the coherence conditions, after boiling the appearances of μ down into ηs and rewriting.

Earlier, we used the fact that LEM_∞ holds within the $\neg\neg$ monad, which means that within $\neg\neg$, we have classical logic. $\neg\neg LEM_\infty$ was proved as an exercise in Chapter 1 of HoTT. In particular, we can prove a variant of Markov's principle,

$$\forall n : \mathbb{N} \quad (\neg\neg\neg\forall n : \mathbb{N} \quad P(n)) \to (\neg\neg\exists n : \mathbb{N} \quad \neg P(n))$$

by first supposing (by contradiction) that the conclusion isn't true, and then using that to explicitly construct a proof of

$$\neg\neg\forall n : \mathbb{N} \quad P(n)$$

by checking $P(n)$ on each n, which results in a $\mathbf{0}$.