Adding Cubes to Agda

Andrea Vezzosi vezzosi@chalmers.se

June 29, 2017

Cubical Type Theory (CTT) [1] provides an extension of Martin-Löf Type Theory (MLTT) where we can interpret the univalence axiom while preserving the canonicity property [2], i.e. every closed term [1] actually computes to a value. The typing and equality rules of CTT come as a fairly well-behaved extension of the ones of MLTT and the denotational model and prototype implementation[2] help clarifying the system further.

Given the above it felt reasonable to introduce the features of CTT into a more mature proof assistant like Agda, and here we report the status of this endeavour. In short:

- The univalence axiom is proven as a theorem and we successfully tested its computational behavior on small examples.
- comp computes for any parametrized data or record types, including coinductive ones, but it is stuck for inductive families
- The interaction of the path type and copatterns gives extensionality principles for coinductive records.
- The interval $\mathbb{I}$ is an actual type, we also have restriction types $A[\varphi \mapsto u]$ and types for partial elements $\mathsf{Partial}\ \varphi\ A$. Their sort makes sure comp does not apply to them.

Examples are collected at `https://github.com/Saizan/cubical-demo`.

Implementation. The main principle used during the development was to fit as much as possible within Agda existing support for builtins and primitives[3] to avoid changing both the surface syntax and the internal representation of terms too much. This way we saved the time that would be required to propagate this kind of widely-affecting changes. In doing so we had to sometimes deviate from the rules of CTT and so rely directly on the denotational semantics.

[1] or even ones mentioning only interval variables

[2] `https://github.com/mortberg/cubicaltt`

[3] previously used mainly for efficient representations of basic types like naturals, integers and strings.

Primarily, we have that the interval is an actual type, although of sort Set_ω so that comp does not apply to it [4]. As a consequence the open terms of type $\mathbb{I}$ are not just the canonical ones (variables of type $\mathbb{I}$ and the demorgan algebra operations) but they also contain neutral elements like applications of functions, which complicates operations like $\forall i.\varphi$ and checking judgements like $\Gamma, \varphi \vdash \varphi = 1_{\mathbb{F}}$ which are handled in the CTT prototype by scrutinizing the normal form of φ as a disjunction of conjunctions of atoms of the form $i = 0$ or $i = 1$. So for example in the context $\Gamma = f : \mathbb{I} \to \mathbb{I}, i : \mathbb{I}$ the term $\forall i.f i$ is simply stuck, since different substitutions for f might or might not use the i variable and give different results. Also, the judgement $\Gamma, f\ i = 1 \vdash f\ i = 1$ will fail, because constraints involving non-canonical terms are ignored [5].

Having $\mathbb{I}$ as a type is not just a source of problems though, it is sometimes convenient to build a "path" without having to specify the endpoints, or e.g. internally prove that unglue is an equivalence for any φ. It is however less clear whether there is a good use for functions that return elements of $\mathbb{I}$.

The Path type is a case in which we actually altered the internal representation to avoid reconstructing types during reduction while still satisfying $p\ 0 = x$ whenever $p : \mathsf{Path}\ x\ y$. The internal representation annotates path application with the endpoints x and y, so that reduction can look them up when needed. This fits well into Agda's typechecker because it elaborates abstract syntax into an internal representation: the typechecker can use the available type information to fill in the endpoints statically without burdening the user.

Further work is required to handle inductive families: suppose we have a family $\mathsf{T} : (\mathsf{N} \to \mathsf{N}) \to \mathsf{Set}$, with constructor $\mathsf{c} : \mathsf{T} f$ and some $p : f = g$ where f and g are definitionally different, which canonical element should $\mathsf{transport}\ \mathsf{T}\ p\ \mathsf{c} : \mathsf{T}\ g$ compute to? One solution would be to actually desugar c into $\mathsf{c} : \forall h \to f = h \to \mathsf{T}\ h$, so that $\mathsf{transport}\ \mathsf{T}\ p\ (\mathsf{c}\ \mathsf{refl})$ could compute to $\mathsf{c}\ p$, but we would like this to be hidden from the user. Also, we want to allow pattern matching for Id or even Path instead of explicit calls to J. Higher Inductive Types are also missing so far, but we hope to have a plan for them in short time.

References

[1] C. Cohen, T. Coquand, S. Huber, and A. Mörtberg. Cubical type theory: a constructive interpretation of the univalence axiom. *CoRR*, abs/1611.02108, 2016.

[2] S. Huber. Canonicity for cubical type theory. *CoRR*, abs/1607.04156, 2016.

[4] Set_ω is a sort which is not itself an element of an higher universe, it is closed under Π types, but it cannot be the domain of a quantification itself. It was originally used as the sort of types that use universe polymorphism.

[5] The situation is similar to pattern matching for inductive families, when we have argument of type e.g. $T\ (g\ x)$ with a constructor $d : T\ 1$: Agda will complain that it cannot unify $g\ x$ with 1, however there it actually refuses to match.