Efficient Evaluation for Cubical Type Theories

András Kovács[1]

j.w.w. Evan Cavallo, Tom Jack, Anders Mörtberg

[1]University of Gothenburg

21 April 2024, Conference on Homotopy Type Theory and Computing, Abu Dhabi

Overview

We'd like to speed up cubical type theories.

- There are computations that we'd like to do but can't, e.g. Brunerie numbers.
- Speed is important for just plain user experience and scalability.

Overview

We'd like to speed up cubical type theories.

- There are computations that we'd like to do but can't, e.g. Brunerie numbers.
- Speed is important for just plain user experience and scalability.

In this talk:

- A demo of a fast new CTT implementation.
- An overview of important ingredients:
  1. Environment machines
  2. Explicit interval substitution
  3. Defunctionalization
  4. Shortcuts in closed cubical evaluation
  5. Avoiding empty compositions

  If we skip any of these, we get blow-ups!

Computation in CTTs

Based on **substitution**.

In plain MLTT:

```
(λ x. t) u ≡ t[x ↦ u]
```

Naive implementation: traverse `t`, replace `x`'s occurrences, build new term.

Rarely used in practice!

```
(λ x y. big) t u ≡ (big[x ↦ t])[y ↦ u]
```

Environment machines

Evaluation takes an **environment** and a **term** as input, and returns a **value**.

```
eval []                         ((λ x y. (x, y)) true false) ≡
eval []                         ((λ x y. (x, y)) true false) ≡
eval [x ↦ true]                 (λ y. (x, y)) false          ≡
eval [x ↦ true, y ↦ false] (x, y)                         ≡
```

```
(eval [x ↦ true, y ↦ false] x, eval [x ↦ true, y ↦ false] y) ≡
(x, y)
```

Substitution is only ever performed on *variables*, as environment lookup.

Environment machines

Evaluation takes an **environment** and a **term** as input, and returns a **value**.

```
eval []                          ((λ x y. (x, y)) true false) ≡
eval []                          ((λ x y. (x, y)) true false) ≡
eval [x ↦ true]                  (λ y. (x, y)) false          ≡
eval [x ↦ true, y ↦ false] (x, y)                             ≡
```

```
(eval [x ↦ true, y ↦ false] x, eval [x ↦ true, y ↦ false] y) ≡
(x, y)
```

Substitution is only ever performed on *variables*, as environment lookup.

NB: this is another view on *normalization-by-evaluation*. I'll focus on the operational perspective, not the formally-nice perspective.

Closures

How to handle λ-s?

```
eval [] (let x = true in let f = λ y. x in f false)   ≡
eval [x ↦ true] (let f = λ y. x in f false)           ≡
eval [x ↦ true, f ↦ <[x ↦ true], y. x>] (f false)     ≡
eval [x ↦ true, y ↦ false] x                          ≡
true
```

The **closure** `<[x ↦ true], y. x>` stores the current environment and the function body. Evaluation of the body resumes when the closure is applied.

Interval substitution in CTT (1)

The same thing doesn't work for interval substitution!

Problem: coercion looks under an interval binder and substitutes it.

```
coe r r' (i. A → B)           t ≡ ...
coe r r' (i. Glue A [α ↦ f]) t ≡ ...   (using [i ↦ r'])
```

More complicated setup:

- Evaluates takes an extra **interval environment** as input.
- Closures store both environments.
- We use closures everywhere except in coercion.

Interval substitution in CTT (2)

We still need interval substitution.

But its action is delayed as much as possible.

- Interval substitution computes nothing, just stores an *explicit substitution*.
- There is a *weak head forcing* operation which computes substitutions until we get to a neutral or a head form.

```
eval γ (t u) ≡ case (force t) of
  <γ', x. t'> → eval [γ', x ↦ eval γ u] t'
  t'(neutral) → t' (eval γ u)
```

Forcing pushes substitutions inside closure environments:

```
force (<γ, x. t>[σ]) ≡ <γ[σ], x.t>
```

Defunctionalization

How to implement?

```
coe r r' (i. A → B) f ≡
  λ x. coe r r' (i. B) (f (coe r' r (i. A) x))
```

The right hand side should be a suspended computation, resumed when we apply it to an argument.

It's a new kind of closure that stores `r`, `r'`, `A`, `B` and `f`!

Every semantic binder becomes a closure.

```
apply (EvalLam γ x t)      u ≡ eval [γ, x ↦ u] t
apply (CoeFun r r' A B f) u ≡ coe r r' (i. B)
                                  (f (coe r' r (i. A) u))
...
```

Annoying: we have 30+ closures in the implementation!

Exploiting canonicity

Problem: composition for non-higher inductive types.

```
hcom r r' [α ↦ i. suc t] (suc b) ≡ suc (hcom r r' [α ↦ i. t] b)
```

We need to force all components of the system to check for "`suc t`"-s!

Canonicity: in a purely cubical context, every $\mathbb{N}$ term is `zero` or `suc`.

In a purely cubical context, if the base of a `hcom` is `suc`, all components of the system must be also `suc`. Hence, we can admit:[1]

```
hcom r r' [α ↦ i. t] (suc b) ≡ suc (hcom r r' [α ↦ i. pred t] b)
```

where `pred` is a metatheoretic function that peels off a `suc`. Importantly, we can compute a `pred`-ed value lazily.

[1] see also in Simon Huber's thesis.

Exploiting canonicity

Generalizing this shortcut to all non-higher inductive types, we get:

Closed evaluation computes at most one component of each system.

Moreover:

Closed evaluation only depends on evaluation in cubical atomic contexts.

(Cubical atomic context: only contains interval variables)

Avoiding empty compositions

HIT-s have *formal compositions* as canonical inhabitants.

We can compose values with nothing:

```
hcom 0 1 [] (loop i)
```

is a canonical element of S^1 that's path equal to `loop i`.

Early CTT implementations suffered from an explosion of empty compositions.

We borrow a simple "naive" solution from Carlo Angiuli's thesis. We don't need fancy versions because of our optimized closed evaluation.

Prospects, Agda

`cctt` implements a Cartesian CTT that can be interpreted into classical homotopy theory. Recent work by Christian Sattler suggests that an extension with $\wedge$ and $\vee$ still works out.

Agda implements the CHM[2] theory, which currently has no such interpretation.

We don't know *for sure* that Agda's CTT is wrong.

Agda would benefit from a major cubical overhaul, but changing the core theory and not changing it could both turn out to be mistakes!

[2] Coquand-Huber-Mörtberg.