Efficient Evaluation for Cubical Type Theories

András Kovács[1]

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

[1]Eötvös Loránd University

24 May 2023, 2nd Conference on Homotopy Type Theory

Overview

Efficiency issues in CTTs.

Overview

Efficiency issues in CTTs.

Previous approaches: optimizing cubical type formers / computation rules, definitions within CTTs.

Overview

Efficiency issues in CTTs.

Previous approaches: optimizing cubical type formers / computation rules, definitions within CTTs.

New in current work:

- Normalization-by-evaluation restructured on a basic level.
- Optimizing the case without fibrant free variables ("closed").

Overview

Efficiency issues in CTTs.

Previous approaches: optimizing cubical type formers / computation rules, definitions within CTTs.

New in current work:

- Normalization-by-evaluation restructured on a basic level.
- Optimizing the case without fibrant free variables ("closed").

WIP. We have a standalone implementation of a Cartesian CTT, plus:

- Large speedups in small benchmarks.

Overview

Efficiency issues in CTTs.

Previous approaches: optimizing cubical type formers / computation rules, definitions within CTTs.

New in current work:

- Normalization-by-evaluation restructured on a basic level.
- Optimizing the case without fibrant free variables ("closed").

WIP. We have a standalone implementation of a Cartesian CTT, plus:

- Large speedups in small benchmarks.
- Three Brunerie number definitions, all compute instantly.
  - One of these is defined but can't be computed in Agda.
  - Another can be computed in `cubicaltt` but not in `redtt`.

Overview

Efficiency issues in CTTs.

Previous approaches: optimizing cubical type formers / computation rules, definitions within CTTs.

New in current work:

- Normalization-by-evaluation restructured on a basic level.
- Optimizing the case without fibrant free variables ("closed").

WIP. We have a standalone implementation of a Cartesian CTT, plus:

- Large speedups in small benchmarks.
- Three Brunerie number definitions, all compute instantly.
  - One of these is defined but can't be computed in Agda.
  - Another can be computed in `cubicaltt` but not in `redtt`.

Many more definitions to go!

Outline

1. Normalization-by-evaluation for MLTT
2. NbE for CTT
3. Implementation & benchmarks
4. Conclusions

Outline

1. Normalization-by-evaluation for MLTT
2. NbE for CTT
3. Implementation & benchmarks
4. Conclusions

Substitution in MLTT

- The equational theory of MLTT refers to **substitution**.

Substitution in MLTT

- The equational theory of MLTT refers to **substitution**.
- Intuitive definition: recursive replacement of variables with terms.

Substitution in MLTT

- The equational theory of MLTT refers to **substitution**.
- Intuitive definition: recursive replacement of variables with terms.
- Most implementations **don't** use this.

$$(\lambda\, x\, y.\, \mathsf{big})\; t\; u \equiv ((\lambda\, y.\, \mathsf{big})[x \mapsto t])\; u \equiv \mathsf{big}[x \mapsto t][y \mapsto u]$$

Substitution in MLTT

- The equational theory of MLTT refers to **substitution**.
- Intuitive definition: recursive replacement of variables with terms.
- Most implementations **don't** use this.

$$(\lambda\, x\, y.\, \mathsf{big})\; t\; u \equiv ((\lambda\, y.\, \mathsf{big})[x \mapsto t])\; u \equiv \mathsf{big}[x \mapsto t][y \mapsto u]$$

Solution

- Separate *syntax* (program code) from *semantic values* (runtime objects).
- The syntax only supports *evaluation* into values.
- Values support efficient β-reduction, without using recursive substitution.

The separation of program code and runtime values is used in most programming languages.

The separation of program code and runtime values is used in most programming languages.

Normalization-by-evaluation extends it with

- Support for free variables in values (open evaluation).
- Mapping values back to syntax (“quotation”).

The separation of program code and runtime values is used in most programming languages.

Normalization-by-evaluation extends it with

- Support for free variables in values (open evaluation).
- Mapping values back to syntax ("quotation").

I focus on a *practical flavor* of NbE which has several differences to the nicest *formal* NbE.

Informal NbE (1)

We omit types of things for brevity.

Syntax & values

$$\begin{aligned} \Gamma,\ \Delta &: \mathsf{Con} & \sigma,\ \delta &: \mathsf{Env}\,\Gamma\,\Delta \\ t,\ u &: \mathsf{Tm}\,\Gamma & v &: \mathsf{Val}\,\Gamma \\ \sigma,\ \delta &: \mathsf{Sub}\,\Gamma\,\Delta \end{aligned}$$

Operations

$$\begin{aligned} \mathsf{eval} &: \mathsf{Env}\,\Gamma\,\Delta \to \mathsf{Tm}\,\Delta \to \mathsf{Val}\,\Gamma \\ \mathsf{quote} &: \mathsf{Val}\,\Gamma \to \mathsf{Tm}\,\Gamma \\ \mathsf{conv} &: \mathsf{Val}\,\Gamma \to \mathsf{Val}\,\Gamma \to \mathsf{Bool} \end{aligned}$$

$\mathsf{Val}\,\Gamma$ has the same structure as $\mathsf{Tm}\,\Gamma$, except each binder is replaced with a **closure**. A closure stores a variable name x, an environment $\sigma : \mathsf{Env}\,\Gamma\,\Delta$ and a $t : \mathsf{Tm}\,(\Delta, x)$.

Informal NbE (2)

$$
\begin{aligned}
&\mathsf{eval} : \mathsf{Env}\,\Gamma\,\Delta \to \mathsf{Tm}\,\Delta \to \mathsf{Val}\,\Gamma\\
&\mathsf{eval}\,\sigma\,x \qquad\quad :\equiv \sigma\,x\\
&\mathsf{eval}\,\sigma\,(\lambda\,x.\,t) :\equiv \lambda_{\mathsf{Val}}\,(x,\ \sigma,\ t)\\
&\mathsf{eval}\,\sigma\,(t\,u) \quad\ :\equiv \mathsf{case}\ \mathsf{eval}\,\sigma\,t\ \mathsf{of}\\
&\qquad \lambda_{\mathsf{Val}}\,(x,\ \delta,\ t) \to \mathsf{eval}\,(\delta,\ x \mapsto \mathsf{eval}\,\sigma\,u)\,t\\
&\qquad v \qquad\qquad\quad\ \to v\,(\mathsf{eval}\,\sigma\,u)
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{quote} : \mathsf{Val}\,\Gamma \to \mathsf{Tm}\,\Gamma\\
&\mathsf{quote}\,x \qquad\qquad\qquad :\equiv x\\
&\mathsf{quote}\,(\lambda_{\mathsf{Val}}\,(x,\ \delta,\ t)) :\equiv \lambda\,x'.\,\mathsf{quote}\,(\mathsf{eval}\,(\delta,\ x \mapsto x')\,t)\\
&\qquad\qquad\qquad\qquad\qquad\quad \text{where } x' \text{ is fresh in } \Gamma\\
&\mathsf{quote}\,(t\,u) \qquad\qquad\ :\equiv (\mathsf{quote}\,t)\,(\mathsf{quote}\,u)
\end{aligned}
$$

Outline

Cubical NbE

In the following we consider Cartesian a CTT with coe, hcom, HITs and Glue.

Cubical NbE

In the following we consider Cartesian a CTT with coe, hcom, HITs and Glue.

Terms are in triple contexts.

- $t,\, u : \mathsf{Tm}\,(\Psi; \alpha; \Gamma)$
- Ψ is a context of interval variables.
- α is a cofibration.
- Γ contains fibrant variables.

Cubical NbE

In the following we consider Cartesian a CTT with coe, hcom, HITs and Glue.

Terms are in triple contexts.

- $t,\, u : \mathsf{Tm}\,(\Psi; \alpha; \Gamma)$
- Ψ is a context of interval variables.
- α is a cofibration.
- Γ contains fibrant variables.

In analogy to MLTT NbE, cubical NbE should take a "semantic interpretation" of the context as input.

- An interval substitution $\sigma : \mathsf{Sub}^{\mathsf{I}}\,\Psi_0\,\Psi_1$.
- A cofibration implication $f : \alpha_0 \Rightarrow \alpha_1[\sigma]$.
- A value environment $\delta : \mathsf{Env}\,\Gamma_0\,(\Gamma_1[\sigma,\, f])$.

Cubical NbE

$$\begin{aligned}
\mathsf{eval} : &\forall\, \Psi_0\, \alpha_0\, \Gamma_0\, \Psi_1\, \alpha_1\, \Gamma_1 \\
&(\sigma : \mathsf{Sub}^{\mathsf{I}}\, \Psi_0\, \Psi_1) \\
&(f : \alpha_0 \Rightarrow \alpha_1[\sigma]) \\
&(\delta : \mathsf{Env}\, \Gamma_0\, (\Gamma_1[\sigma,\ f]) \\
&\to \mathsf{Tm}\,(\Psi_1; \alpha_1; \Gamma_1) \to \mathsf{Val}\,(\Psi_0; \alpha_0; \Gamma_0)
\end{aligned}$$

Cubical NbE

$$\begin{aligned}\mathsf{eval} : \forall\, \Psi_0\, \alpha_0\, \Gamma_0\, \Psi_1\, \alpha_1\, \Gamma_1 \\ (\sigma : \mathsf{Sub}^{\mathsf{I}}\, \Psi_0\, \Psi_1) \\ (f : \alpha_0 \Rightarrow \alpha_1[\sigma]) \\ (\delta : \mathsf{Env}\, \Gamma_0\, (\Gamma_1[\sigma,\ f]) \\ \to \mathsf{Tm}\, (\Psi_1; \alpha_1; \Gamma_1) \to \mathsf{Val}\, (\Psi_0; \alpha_0; \Gamma_0)\end{aligned}$$

6 out of 10 inputs are computationally relevant in implementation:

- Ψ_0 marks the next fresh interval variable.

Cubical NbE

$$
\begin{aligned}
\mathsf{eval} : \forall\, \Psi_0\, \alpha_0\, \Gamma_0\, \Psi_1\, \alpha_1\, \Gamma_1 \\
& (\sigma : \mathsf{Sub}^{\mathsf{I}}\, \Psi_0\, \Psi_1) \\
& (f : \alpha_0 \Rightarrow \alpha_1[\sigma]) \\
& (\delta : \mathsf{Env}\, \Gamma_0\, (\Gamma_1[\sigma,\ f]) \\
& \to \mathsf{Tm}\, (\Psi_1; \alpha_1; \Gamma_1) \to \mathsf{Val}\, (\Psi_0; \alpha_0; \Gamma_0)
\end{aligned}
$$

6 out of 10 inputs are computationally relevant in implementation:

- Ψ_0 marks the next fresh interval variable.
- α_0 is used for "forcing" (see later).

Cubical NbE

$$\begin{aligned}\mathsf{eval} :\ & \forall\, \Psi_0\, \alpha_0\, \Gamma_0\, \Psi_1\, \alpha_1\, \Gamma_1 \\ & (\sigma : \mathsf{Sub}^{\mathsf{I}}\, \Psi_0\, \Psi_1) \\ & (f : \alpha_0 \Rightarrow \alpha_1[\sigma]) \\ & (\delta : \mathsf{Env}\, \Gamma_0\, (\Gamma_1[\sigma,\ f]) \\ & \to \mathsf{Tm}\,(\Psi_1; \alpha_1; \Gamma_1) \to \mathsf{Val}\,(\Psi_0; \alpha_0; \Gamma_0)\end{aligned}$$

6 out of 10 inputs are computationally relevant in implementation:

- Ψ_0 marks the next fresh interval variable.
- α_0 is used for “forcing” (see later).
- Γ_0 is passed to detect when there are no fibrant free variables.

Cubical NbE

$$
\begin{aligned}
\mathsf{eval} : \forall\, \Psi_0\, \alpha_0\, \Gamma_0\, \Psi_1\, \alpha_1\, \Gamma_1 \\
&(\sigma : \mathsf{Sub}^{\mathsf{I}}\, \Psi_0\, \Psi_1) \\
&(f : \alpha_0 \Rightarrow \alpha_1[\sigma]) \\
&(\delta : \mathsf{Env}\, \Gamma_0\, (\Gamma_1[\sigma,\, f]) \\
&\to \mathsf{Tm}\,(\Psi_1; \alpha_1; \Gamma_1) \to \mathsf{Val}\,(\Psi_0; \alpha_0; \Gamma_0)
\end{aligned}
$$

6 out of 10 inputs are computationally relevant in implementation:

- Ψ_0 marks the next fresh interval variable.
- α_0 is used for "forcing" (see later).
- Γ_0 is passed to detect when there are no fibrant free variables.
- σ, δ and t are evidently required.

Trouble with interval substitution

MLTT NbE: Val substitution is inefficient.

$$-[-] : \mathsf{Val}\,\Delta \to \mathsf{Env}\,\Gamma\,\Delta \to \mathsf{Val}\,\Gamma$$

Evaluation creates **shared structure**. Recursive substitution destroys all such sharing by creating fresh copies of values.

Example for sharing:

$$\mathsf{let}\,x := f\;y\,\mathsf{in}\;(x,\,x,\,x,\,x,\,x)$$

Trouble with interval substitution

MLTT NbE: Val substitution is inefficient.

$$-[-] : \mathsf{Val}\,\Delta \to \mathsf{Env}\,\Gamma\,\Delta \to \mathsf{Val}\,\Gamma$$

Evaluation creates **shared structure**. Recursive substitution destroys all such sharing by creating fresh copies of values.

Example for sharing:

$$\mathsf{let}\, x := f\ y\, \mathsf{in}\ (x, x, x, x, x)$$

Likewise: recursive **interval substitution** destroys all structure sharing.

- MLTT NbE: no need for value substitution.
- CTT NbE: must support interval substitution on values.

Cubical NbE

Two extra operations.

1. Interval substitution

$$-[-] : \mathsf{Val}\,(\Psi_0; \alpha; \Gamma) \to (\sigma : \mathsf{Sub}^{\mathsf{I}}\,\Psi_1\,\Psi_0) \to \mathsf{Val}\,(\Psi_1; \alpha[\sigma]; \Gamma[\sigma])$$

Has trivial operational cost, only stores an explicit substitution.

Cubical NbE

Two extra operations.

1. Interval substitution

$$-[-] : \mathsf{Val}\,(\Psi_0; \alpha; \Gamma) \to (\sigma : \mathsf{Sub}^{\mathsf{I}}\,\Psi_1\,\Psi_0) \to \mathsf{Val}\,(\Psi_1; \alpha[\sigma]; \Gamma[\sigma])$$

Has trivial operational cost, only stores an explicit substitution.

2. Forcing

$$\mathsf{force} : \mathsf{Val}\,(\Psi; \alpha; \Gamma) \to \mathsf{Val}\,(\Psi; \alpha; \Gamma)$$

Computes delayed substitutions sufficiently to yield a *head normal* value.
See also: notion of forcing in lazy evaluation.

Stability annotations

Forcing has trivial cost on canonical values, for example:

$$\mathsf{force}\,(((x : A) \to B)[\sigma]) \equiv ((x : A[\sigma]) \to B[\sigma])$$

Stability annotations

Forcing has trivial cost on canonical values, for example:

$$\mathsf{force}\,(((x : A) \to B)[\sigma]) \equiv ((x : A[\sigma]) \to B[\sigma])$$

But it may have *arbitrary* cost on neutral values.

$$\begin{aligned}&\mathsf{force}\,((\mathsf{coe}_{\mathsf{Ne}}\; r\; r'\,(i.\,A)\; t)[\sigma]) \equiv \\ &\mathsf{force}\,(\mathsf{coe}\,(r[\sigma])\,(r'[\sigma])(i.\,A[\sigma])\,(t[\sigma]))\end{aligned}$$

Stability annotations

Forcing has trivial cost on canonical values, for example:

$$\mathsf{force}\,(((x : A) \to B)[\sigma]) \equiv ((x : A[\sigma]) \to B[\sigma])$$

But it may have *arbitrary* cost on neutral values.

$$\begin{gathered}\mathsf{force}\,((\mathsf{coe}_{\mathsf{Ne}}\; r\; r'\,(i.\,A)\; t)[\sigma]) \equiv \\ \mathsf{force}\,(\mathsf{coe}\,(r[\sigma])\,(r'[\sigma])(i.\,A[\sigma])\,(t[\sigma]))\end{gathered}$$

Neutrals are not necessarily stable under substitution!

Stability annotations

Forcing has trivial cost on canonical values, for example:

$$\mathsf{force}\,(((x : A) \to B)[\sigma]) \equiv ((x : A[\sigma]) \to B[\sigma])$$

But it may have *arbitrary* cost on neutral values.

$$\mathsf{force}\,((\mathsf{coe}_{\mathsf{Ne}}\; r\; r'\,(i.\,A)\; t)[\sigma]) \equiv$$
$$\mathsf{force}\,(\mathsf{coe}\,(r[\sigma])\,(r'[\sigma])(i.\,A[\sigma])\,(t[\sigma]))$$

Neutrals are not necessarily stable under substitution!

Angiuli & Sterling[1]: let's annotate neutrals with stability information.

[1] *Normalization for Cubical Type Theory*, LICS 2021

Stability annotations

Forcing has trivial cost on canonical values, for example:

$$\mathsf{force}\,(((x : A) \to B)[\sigma]) \equiv ((x : A[\sigma]) \to B[\sigma])$$

But it may have *arbitrary* cost on neutral values.

$$\begin{aligned}&\mathsf{force}\,((\mathsf{coe}_{\mathsf{Ne}}\; r\; r'\,(i.\,A)\; t)[\sigma]) \equiv \\ &\mathsf{force}\,(\mathsf{coe}\,(r[\sigma])\,(r'[\sigma])(i.\,A[\sigma])\,(t[\sigma]))\end{aligned}$$

Neutrals are not necessarily stable under substitution!

Angiuli & Sterling[1]: let's annotate neutrals with stability information.

Our implementation:

- Neutrals are annotated with *blocking sets* of interval variables.
- Only an approximation of precise predicates!
- We can quickly see if a substitution has no action on a neutral.

[1] *Normalization for Cubical Type Theory*, LICS 2021

Forcing w.r.t. cofibrations

Forcing doesn't just compute substitutions, but *cofibration weakening* as well.

$$\begin{aligned}&\mathsf{let}\, x := \mathsf{coe}\, i\, j\, (k.\, A)\, y\ \mathsf{in}\\ &\mathsf{hcom}\, 0\, 1\, [i = j \mapsto x]\, z\end{aligned}$$

x is first evaluated under some cofibration α, but then mentioned under $\alpha \wedge i = j$.

Forcing w.r.t. cofibrations

Forcing doesn't just compute substitutions, but *cofibration weakening* as well.

$$\begin{aligned}&\mathsf{let}\, x := \mathsf{coe}\, i\, j\, (k.\, A)\, y\ \mathsf{in}\\ &\mathsf{hcom}\, 0\, 1\, [i = j \mapsto x]\, z\end{aligned}$$

x is first evaluated under some cofibration α, but then mentioned under $\alpha \wedge i = j$.

Cofibration weakening is *implicit*. Any value can be "outdated" w.r.t. the current cofibration.

Forcing w.r.t. cofibrations

Forcing doesn't just compute substitutions, but *cofibration weakening* as well.

$$\text{let}\, x := \text{coe}\, i\, j\, (k.\, A)\, y \;\text{ in}$$
$$\text{hcom}\, 0\, 1\, [i = j \mapsto x]\, z$$

x is first evaluated under some cofibration α, but then mentioned under $\alpha \wedge i = j$.

Cofibration weakening is *implicit*. Any value can be "outdated" w.r.t. the current cofibration.

Contrast MLTT NbE: weakening of values has no cost!
(if we use a suitable variable representation in values, e.g. De Bruijn levels)

Closures vs. binders

We can't represent all interval binders with closures!

$$\mathsf{coe}\, r\, r'\, (i.\, A \to B)\, f \equiv \lambda\, x.\, \mathsf{coe}\, r\, r'\, (i.\, B)\, (f\, (\mathsf{coe}\, r'\, r\, (i.\, A)\, x))$$

Closures vs. binders

We can't represent all interval binders with closures!

$$\mathsf{coe}\, r\, r'\, (i.\, A \to B)\, f \equiv \lambda\, x.\, \mathsf{coe}\, r\, r'\, (i.\, B)\, (f\, (\mathsf{coe}\, r'\, r\, (i.\, A)\, x))$$

Closures vs. binders

We can't represent all interval binders with closures!

$$\mathsf{coe}\, r\, r'\, (i.\, A \to B)\, f \equiv \lambda\, x.\, \mathsf{coe}\, r\, r'\, (i.\, B)\, (f\, (\mathsf{coe}\, r'\, r\, (i.\, A)\, x))$$

Closures are "extensional", we can't efficiently inspect their bodies.

Closures vs. binders

We can't represent all interval binders with closures!

$$\mathsf{coe}\, r\, r'\, (i.\, A \to B)\, f \equiv \lambda\, x.\, \mathsf{coe}\, r\, r'\, (i.\, B)\, (f\, (\mathsf{coe}\, r'\, r\, (i.\, A)\, x))$$

Closures are "extensional", we can't efficiently inspect their bodies.

- coe, hcom: we need to peek under interval binders, so we use *explicit weakenings* as semantic binders.
- Other cases (e.g. dependent paths, path abstractions): we use closures.

Defunctionalization (1)

We actually need many different kinds of closures. Again consider:

$$\mathsf{coe}\, r\, r'\, (i.\, A \to B)\, f \equiv \lambda\, x.\, \mathsf{coe}\, r\, r'\, (i.\, B)\, (f\, (\mathsf{coe}\, r'\, r\, (i.\, A)\, x))$$

The $\lambda\, x.$ abstraction has to act on semantic values.

Defunctionalization (1)

We actually need many different kinds of closures. Again consider:

$$\mathsf{coe}\, r\, r'\, (i.\, A \to B)\, f \equiv \lambda\, x.\, \mathsf{coe}\, r\, r'\, (i.\, B)\, (f\, (\mathsf{coe}\, r'\, r\, (i.\, A)\, x))$$

The $\lambda\, x.$ abstraction has to act on semantic values.

Previously: a closure $(x,\, \sigma,\, t)$ can be **applied** to some value u by computing $\mathsf{eval}\, (\sigma,\, x \mapsto u)\, t$.

Defunctionalization (1)

We actually need many different kinds of closures. Again consider:

$$\mathsf{coe}\, r\, r'\, (i.\, A \to B)\, f \equiv \lambda\, x.\, \mathsf{coe}\, r\, r'\, (i.\, B)\, (f\, (\mathsf{coe}\, r'\, r\, (i.\, A)\, x))$$

The $\lambda\, x.$ abstraction has to act on semantic values.

Previously: a closure $(x,\, \sigma,\, t)$ can be **applied** to some value u by computing $\mathsf{eval}\, (\sigma,\, x \mapsto u)\, t$.

We add a new closure, storing $(r,\, r',\, A,\, B,\, f)$, which can be **applied** to some value x by computing $\mathsf{coe}\, r\, r'\, (i.\, B)\, (f\, (\mathsf{coe}\, r'\, r\, (i.\, A)\, x))$.

Defunctionalization (1)

We actually need many different kinds of closures. Again consider:

$$\mathsf{coe}\, r\, r'\, (i.\, A \to B)\, f \equiv \lambda\, x.\, \mathsf{coe}\, r\, r'\, (i.\, B)\, (f\, (\mathsf{coe}\, r'\, r\, (i.\, A)\, x))$$

The $\lambda\, x.$ abstraction has to act on semantic values.

Previously: a closure $(x,\, \sigma,\, t)$ can be **applied** to some value u by computing $\mathsf{eval}\, (\sigma,\, x \mapsto u)\, t$.

We add a new closure, storing $(r,\, r',\, A,\, B,\, f)$, which can be **applied** to some value x by computing $\mathsf{coe}\, r\, r'\, (i.\, B)\, (f\, (\mathsf{coe}\, r'\, r\, (i.\, A)\, x))$.

The semantic type of closures is the sum type of all such closures. The *generic application* function is a big case split on them.

Defunctionalization (1)

We actually need many different kinds of closures. Again consider:

$$\mathsf{coe}\, r\, r'\, (i.\, A \to B)\, f \equiv \lambda\, x.\, \mathsf{coe}\, r\, r'\, (i.\, B)\, (f\, (\mathsf{coe}\, r'\, r\, (i.\, A)\, x))$$

The $\lambda\, x.$ abstraction has to act on semantic values.

Previously: a closure $(x,\ \sigma,\ t)$ can be **applied** to some value u by computing $\mathsf{eval}\, (\sigma,\ x \mapsto u)\, t$.

We add a new closure, storing $(r,\ r',\ A,\ B,\ f)$, which can be **applied** to some value x by computing $\mathsf{coe}\, r\, r'\, (i.\, B)\, (f\, (\mathsf{coe}\, r'\, r\, (i.\, A)\, x))$.

The semantic type of closures is the sum type of all such closures. The *generic application* function is a big case split on them.

Defunctionalization: representing higher-order functions with first-order data and a first-order generic application.

Defunctionalization (2)

Interval substitution has action on closures:

$$
\begin{aligned}
(\mathsf{eval}_{\mathsf{cl}}(x, \delta, t))[\sigma] &\equiv \mathsf{eval}_{\mathsf{cl}}(x, \delta[\sigma], t) \\
(\mathsf{coeFun}_{\mathsf{cl}}(r, r', A, B, f))[\sigma] &\equiv \mathsf{coeFun}_{\mathsf{cl}}(r[\sigma], r'[\sigma], A[\sigma], B[\sigma], f[\sigma])
\end{aligned}
$$

Defunctionalization (2)

Interval substitution has action on closures:

$$\begin{aligned}(\mathsf{eval}_{\mathsf{cl}}(x,\,\delta,\,t))[\sigma] &\equiv \mathsf{eval}_{\mathsf{cl}}(x,\,\delta[\sigma],\,t)\\ (\mathsf{coeFun}_{\mathsf{cl}}(r,\,r',\,A,\,B,\,f))[\sigma] &\equiv \mathsf{coeFun}_{\mathsf{cl}}(r[\sigma],\,r'[\sigma],\,A[\sigma],\,B[\sigma],\,f[\sigma])\end{aligned}$$

Fun fact: we have **37** different closures in the implementation. It's a bit tedious!

Defunctionalization (2)

Interval substitution has action on closures:

$$
\begin{aligned}
(\mathsf{eval}_{\mathsf{cl}}(x,\, \delta,\, t))[\sigma] &\equiv \mathsf{eval}_{\mathsf{cl}}(x,\, \delta[\sigma],\, t) \\
(\mathsf{coeFun}_{\mathsf{cl}}(r,\, r',\, A,\, B,\, f))[\sigma] &\equiv \mathsf{coeFun}_{\mathsf{cl}}(r[\sigma],\, r'[\sigma],\, A[\sigma],\, B[\sigma],\, f[\sigma])
\end{aligned}
$$

Fun fact: we have **37** different closures in the implementation. It's a bit tedious!

Ideally, we'd just write higher-order binders in semantics, and automatically generate for each one:

1. The closure data definition.
2. The generic application definition.
3. The definition of the action of substitution.

Defunctionalization (2)

Interval substitution has action on closures:

$$(\mathsf{eval}_{\mathsf{cl}}(x, \delta, t))[\sigma] \equiv \mathsf{eval}_{\mathsf{cl}}(x, \delta[\sigma], t)$$
$$(\mathsf{coeFun}_{\mathsf{cl}}(r, r', A, B, f))[\sigma] \equiv \mathsf{coeFun}_{\mathsf{cl}}(r[\sigma], r'[\sigma], A[\sigma], B[\sigma], f[\sigma])$$

Fun fact: we have **37** different closures in the implementation. It's a bit tedious!

Ideally, we'd just write higher-order binders in semantics, and automatically generate for each one:

1. The closure data definition.
2. The generic application definition.
3. The definition of the action of substitution.

Seems like a major challenge. In the long term we'd want some *logical framework* for implementing (C)TT evaluation.

Reaping some benefits

Assumption: bounded interval scopes. When discussing costs & complexities in the following, we assume that interval contexts are small and bounded during evaluation.

Reaping some benefits

Assumption: bounded interval scopes. When discussing costs & complexities in the following, we assume that interval contexts are small and bounded during evaluation.

1. There is exactly one computation rule in our CTT which performs *arbitrary* interval substitution: coercion over Glue.

Reaping some benefits

Assumption: bounded interval scopes. When discussing costs & complexities in the following, we assume that interval contexts are small and bounded during evaluation.

1. There is exactly one computation rule in our CTT which performs *arbitrary* interval substitution: coercion over Glue.
2. All other substitutions are weakenings.

Reaping some benefits

Assumption: bounded interval scopes. When discussing costs & complexities in the following, we assume that interval contexts are small and bounded during evaluation.

1. There is exactly one computation rule in our CTT which performs *arbitrary* interval substitution: coercion over Glue.
2. All other substitutions are weakenings.
3. Neutrals are stable under weakening & forcing by weakening has trivial cost.

Reaping some benefits

Assumption: bounded interval scopes. When discussing costs & complexities in the following, we assume that interval contexts are small and bounded during evaluation.

1. There is exactly one computation rule in our CTT which performs *arbitrary* interval substitution: coercion over Glue.
2. All other substitutions are weakenings.
3. Neutrals are stable under weakening & forcing by weakening has trivial cost.

If we don't coerce along Glue, interval substitution only has linear runtime overhead.

Exploiting CTT canonicity (1)

Back to MLTT for a bit:

- Consider closed evaluation of if − then − else.
- The Bool scrutinee is true or false, so we have to evaluate just one of the branches.
- In open evaluation: if the scrutinee is neutral, we may have to evaluate *both* branches.

Exploiting CTT canonicity (1)

Back to MLTT for a bit:

- Consider closed evaluation of if − then − else.
- The Bool scrutinee is true or false, so we have to evaluate just one of the branches.
- In open evaluation: if the scrutinee is neutral, we may have to evaluate *both* branches.

In CTT:

- hcom is kind of a branching structure.

Exploiting CTT canonicity (1)

Back to MLTT for a bit:

- Consider closed evaluation of if − then − else.
- The Bool scrutinee is true or false, so we have to evaluate just one of the branches.
- In open evaluation: if the scrutinee is neutral, we may have to evaluate *both* branches.

In CTT:

- hcom is kind of a branching structure.
- There are computation rules in *closed evaluation* which evaluate *all* components ("branches") of a system!

Exploiting CTT canonicity (1)

Back to MLTT for a bit:

- Consider closed evaluation of if − then − else.
- The Bool scrutinee is true or false, so we have to evaluate just one of the branches.
- In open evaluation: if the scrutinee is neutral, we may have to evaluate *both* branches.

In CTT:

- hcom is kind of a branching structure.
- There are computation rules in *closed evaluation* which evaluate *all* components ("branches") of a system!
- This is bad.

Exploiting CTT canonicity (2)

The offending rules are precisely the hcom rules for strict inductive types.

$$\mathsf{hcom}\, r\, r'\, [\alpha \mapsto i.\, \mathsf{suc}\, t]\, (\mathsf{suc}\, b) \equiv \mathsf{suc}\, (\mathsf{hcom}\, r\, r'\, [\alpha \mapsto i.\, t]\, b)$$

Exploiting CTT canonicity (2)

The offending rules are precisely the hcom rules for strict inductive types.

$$\mathsf{hcom}\, r\, r'\, [\alpha \mapsto i.\, \mathsf{suc}\, t]\, (\mathsf{suc}\, b) \equiv \mathsf{suc}\, (\mathsf{hcom}\, r\, r'\, [\alpha \mapsto i.\, t]\, b)$$

If there are no fibrant free variables, if we have:

$$\mathsf{hcom}\, r\, r'\, [\alpha \mapsto i.\, t]\, (\mathsf{suc}\, b) : \mathbb{N}$$

Then canonicity implies that $t \equiv \mathsf{suc}\, t'$ for some t'.

Exploiting CTT canonicity (2)

The offending rules are precisely the hcom rules for strict inductive types.

$$\mathsf{hcom}\, r\, r'\, [\alpha \mapsto i.\, \mathsf{suc}\, t]\, (\mathsf{suc}\, b) \equiv \mathsf{suc}\, (\mathsf{hcom}\, r\, r'\, [\alpha \mapsto i.\, t]\, b)$$

If there are no fibrant free variables, if we have:

$$\mathsf{hcom}\, r\, r'\, [\alpha \mapsto i.\, t]\, (\mathsf{suc}\, b) : \mathbb{N}$$

Then canonicity implies that $t \equiv \mathsf{suc}\, t'$ for some t'.

So we can use this rule instead[2]:

$$\mathsf{hcom}\, r\, r'\, [\alpha \mapsto i.\, t]\, (\mathsf{suc}\, b) \equiv \mathsf{suc}\, (\mathsf{hcom}\, r\, r'\, [\alpha \mapsto i.\, \mathsf{pred}\, t]\, b)$$

pred is a metatheoretic function which unwraps a suc.

[2] Used in Simon Huber: *Cubical Interpretations of Type Theory*, sec. 7.2

Exploiting CTT canonicity (3)

The pred rule can be generalized for arbitrary strict inductive types.

Exploiting CTT canonicity (3)

The pred rule can be generalized for arbitrary strict inductive types.

In a purely cubical context (no fibrant variables), no computation rule evaluates all components of a system.

Outline

1. Normalization-by-evaluation for MLTT
2. NbE for CTT
3. Implementation & benchmarks
4. Conclusions

Implementation

- `https://github.com/AndrasKovacs/cctt`
- It's called `cctt` because it's a Cartesian CTT.
- $\sim$5000 lines of Haskell.
- Features: path types, line types, bidirectional type inference, strict inductive types, parameterized HITs.
- Design is a mixture of AFH, ABCFHL and `cubicaltt`.
  - Systems and ghcom from AFH.
  - Glue type from ABCFHL.
  - HIT implementation from `cubicaltt`.
- No universe checking (type-in-type), no termination checking.
- At least 100 times faster type checking than Agda.

Transporting along Bool negation

Convert Bool negation to a path, compose it with itself N times, transport true over it. Times in seconds.

N	**Agda**	**cctt**	**Ratio**
100	0.29	0.00041	707
250	0.97	0.00095	1021
500	3.36	0.0019	1768
750	7.07	0.0030	2356
1000	12.57	0.0047	2674
10^6	N/A	5.65	N/A

Computing winding numbers

Take an integer, convert it to a path in base $=_{\mathbb{S}^1}$ base, then convert back. Times in seconds.

N	**Agda**	**cctt**	**Ratio**
100	0.34	0.0005	680
250	1.89	0.0012	1575
500	5.643	0.0023	2453
750	10.37	0.0043	2411
1000	18.52	0.0059	3138
10^6	N/A	7.98	N/A

Brunerie and the issue with hcom-s (1)

We tried the new Brunerie number definition by Ljungström and Mörtberg[3].

Problem: we did not have ghcom at that point. We had two extra empty hcom-s for each coercion along univalence.

This caused a mismatch with cubical Agda, the following did not typecheck:

```
brunerie : ℤ :=
  g10 (g9 (g8 (λ i j. f7 (λ k. η₃ (push (loop1 i) (loop1 j) k)))));
```

[3] *Formalizing $\pi_4(\mathbb{S}^3) \cong \mathbb{Z}/2\mathbb{Z}$ and Computing a Brunerie Number in Cubical Agda*

Brunerie and the issue with hcom-s (2)

Fortunately, I was able to manually insert 18 or 36 Glue types at several places to make it well-typed. One such place:

```
g9' : gbase1'' = gbase1'' → sTrunc ℤ :=
  λ p.
    unglue (unglue (unglue (unglue (unglue (unglue (unglue (unglue (unglue (unglue (
    unglue (unglue (unglue (unglue (unglue (unglue (unglue (unglue (unglue (unglue (
    unglue (unglue (unglue (unglue (unglue (unglue (unglue (unglue (unglue (unglue (
    unglue (unglue (unglue (unglue (unglue (unglue (
    (coe 0 1 (i. (codeTruncS¹' (p i)).1) spos0)
    ))))))))))))))))))))))))))))))))))));
```

Brunerie and the issue with hcom-s (2)

Fortunately, I was able to manually insert 18 or 36 Glue types at several places to make it well-typed. One such place:

```
g9' : gbase1'' = gbase1'' → sTrunc ℤ :=
  λ p.
    unglue (unglue (unglue (unglue (unglue (unglue (unglue (unglue (unglue (unglue (
    unglue (unglue (unglue (unglue (unglue (unglue (unglue (unglue (unglue (unglue (
    unglue (unglue (unglue (unglue (unglue (unglue (unglue (unglue (unglue (unglue (
    unglue (unglue (unglue (unglue (unglue (unglue (
    (coe 0 1 (i. (codeTruncS¹' (p i)).1) spos0)
    ))))))))))))))))))))))))))))))))))));
```

- The number computes to -2 in ∼50 seconds.
- Computes 60 million hcom-s in total.
- Just before the last $g10$ step, we have the set truncation of -2 wrapped in half million empty hcom-s.

Brunerie and the issue with hcom-s (2)

Fortunately, I was able to manually insert 18 or 36 Glue types at several places to make it well-typed. One such place:

```
g9' : gbase1'' = gbase1'' → sTrunc ℤ :=
  λ p.
    unglue (unglue (unglue (unglue (unglue (unglue (unglue (unglue (unglue (unglue (
    unglue (unglue (unglue (unglue (unglue (unglue (unglue (unglue (unglue (unglue (
    unglue (unglue (unglue (unglue (unglue (unglue (unglue (unglue (unglue (unglue (
    unglue (unglue (unglue (unglue (unglue (unglue (
    (coe 0 1 (i. (codeTruncS¹' (p i)).1) spos0)
    ))))))))))))))))))))))))))))))))))));
```

- The number computes to -2 in ∼50 seconds.
- Computes 60 million hcom-s in total.
- Just before the last $g10$ step, we have the set truncation of -2 wrapped in half million empty hcom-s.

“Who needs ghcom if we can easily compute a few million empty hcom-s?”

More Brunerie numbers

With the addition of ghcom:

- The Agda-computable Brunerie number definition runs in 0.5 ms, computing a mere 700 hcom-s ($\sim$100k times speedup!).

More Brunerie numbers

With the addition of ghcom:

- The Agda-computable Brunerie number definition runs in 0.5 ms, computing a mere 700 hcom-s ($\sim$100k times speedup!).
- An Agda-incomputable variant of the definition runs in 20 ms. Without ghcom it did not compute.

More Brunerie numbers

With the addition of ghcom:

- The Agda-computable Brunerie number definition runs in 0.5 ms, computing a mere 700 hcom-s ($\sim$100k times speedup!).
- An Agda-incomputable variant of the definition runs in 20 ms. Without ghcom it did not compute.
- Tom Jack's Brunerie number computes in 0.2 seconds.
  - It does not compute in `redtt`.
  - It gets stuck in Agda (an apparent bug!).
  - It computes instantly in `cubicaltt`.

More Brunerie numbers

With the addition of ghcom:

- The Agda-computable Brunerie number definition runs in 0.5 ms, computing a mere 700 hcom-s ($\sim$100k times speedup!).
- An Agda-incomputable variant of the definition runs in 20 ms. Without ghcom it did not compute.
- Tom Jack's Brunerie number computes in 0.2 seconds.
  - It does not compute in `redtt`.
  - It gets stuck in Agda (an apparent bug!).
  - It computes instantly in `cubicaltt`.

To do:

- Two more variants from Anders & Axel's paper (β_1 and β_2).
- The infamous older `cubicaltt` definitions.

Speedup from De Morgan intervals?

Tom Jack has a $\pi_3(\mathbb{S}^2)$ generator definition:

- Computes instantly in `cubicaltt` (De Morgan CTT).
- Computes in 3 minutes in `cctt`, in 96 million hcom-s.
  (Fun fact: without ghcom, it computes in 20 minutes, in **9.5 billion** hcom-s.)

Speedup from De Morgan intervals?

Tom Jack has a $\pi_3(\mathbb{S}^2)$ generator definition:

- Computes instantly in `cubicaltt` (De Morgan CTT).
- Computes in 3 minutes in `cctt`, in 96 million hcom-s.
  (Fun fact: without ghcom, it computes in 20 minutes, in **9.5 billion** hcom-s.)

The difference *appears to be* the usage of interval connections.

Could we add some connections to Cartesian CTT?

Or: implement a De Morgan CTT with our basic optimizations.

Scope size issues

How should we associate iterated path composition, e.g. $p \cdot q \cdot r$?

Scope size issues

How should we associate iterated path composition, e.g. $p \cdot q \cdot r$?

Depending on the definition, one version will be **linear time** and the other will be usually **quadratic**.

Scope size issues

How should we associate iterated path composition, e.g. $p \cdot q \cdot r$?

Depending on the definition, one version will be **linear time** and the other will be usually **quadratic**.

The quadratic version iterates the *nesting* of systems, introducing unbounded interval scopes!

Scope size issues

How should we associate iterated path composition, e.g. $p \cdot q \cdot r$?

Depending on the definition, one version will be **linear time** and the other will be usually **quadratic**.

The quadratic version iterates the *nesting* of systems, introducing unbounded interval scopes!

No pathological scopes so far in examples. Computing the Brunerie numbers needs at most 15 interval variables.

Scope size issues

How should we associate iterated path composition, e.g. $p \cdot q \cdot r$?

Depending on the definition, one version will be **linear time** and the other will be usually **quadratic**.

The quadratic version iterates the *nesting* of systems, introducing unbounded interval scopes!

No pathological scopes so far in examples. Computing the Brunerie numbers needs at most 15 interval variables.

Tom's $\pi_3(\mathbb{S}^2)$ generator needs 110 variables.

Scope size issues

How should we associate iterated path composition, e.g. $p \cdot q \cdot r$?

Depending on the definition, one version will be **linear time** and the other will be usually **quadratic**.

The quadratic version iterates the *nesting* of systems, introducing unbounded interval scopes!

No pathological scopes so far in examples. Computing the Brunerie numbers needs at most 15 interval variables.

Tom's $\pi_3(\mathbb{S}^2)$ generator needs 110 variables.

Should we improve scope asymptotics, or just tell users to not blow up scopes?

Scope size issues

How should we associate iterated path composition, e.g. $p \cdot q \cdot r$?

Depending on the definition, one version will be **linear time** and the other will be usually **quadratic**.

The quadratic version iterates the *nesting* of systems, introducing unbounded interval scopes!

No pathological scopes so far in examples. Computing the Brunerie numbers needs at most 15 interval variables.

Tom's $\pi_3(\mathbb{S}^2)$ generator needs 110 variables.

Should we improve scope asymptotics, or just tell users to not blow up scopes?

Outline

Conclusions & future work

We need more definitions!

Conclusions & future work

We need more definitions!

We need more tracing, statistics, and better ways to isolate certain optimizations.

Conclusions & future work

We need more definitions!

We need more tracing, statistics, and better ways to isolate certain optimizations.

Multiple asymptotic “bombs”, ideally we want to defuse all of them.

Conclusions & future work

We need more definitions!

We need more tracing, statistics, and better ways to isolate certain optimizations.

Multiple asymptotic "bombs", ideally we want to defuse all of them.

Unclear what computational cost is *essential* in Brunerie number definitions. So far everything runs instantly in *some* system.

Conclusions & future work

We need more definitions!

We need more tracing, statistics, and better ways to isolate certain optimizations.

Multiple asymptotic "bombs", ideally we want to defuse all of them.

Unclear what computational cost is *essential* in Brunerie number definitions. So far everything runs instantly in *some* system.

Can we add this to Agda? Yes. Some things are harder. We'd need a complete rewrite of the Agda Abstract Machine.