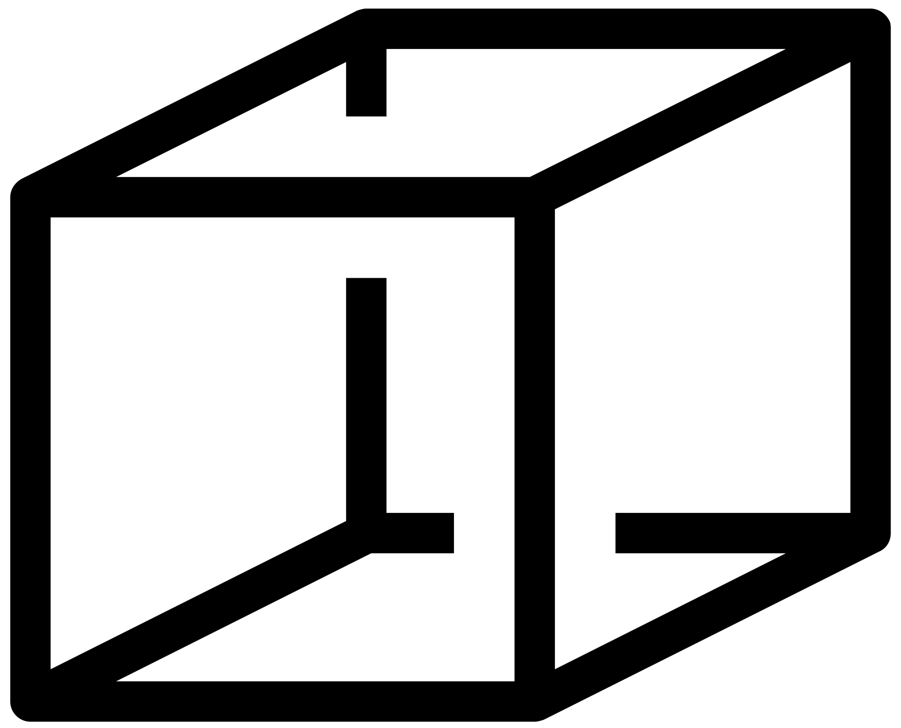
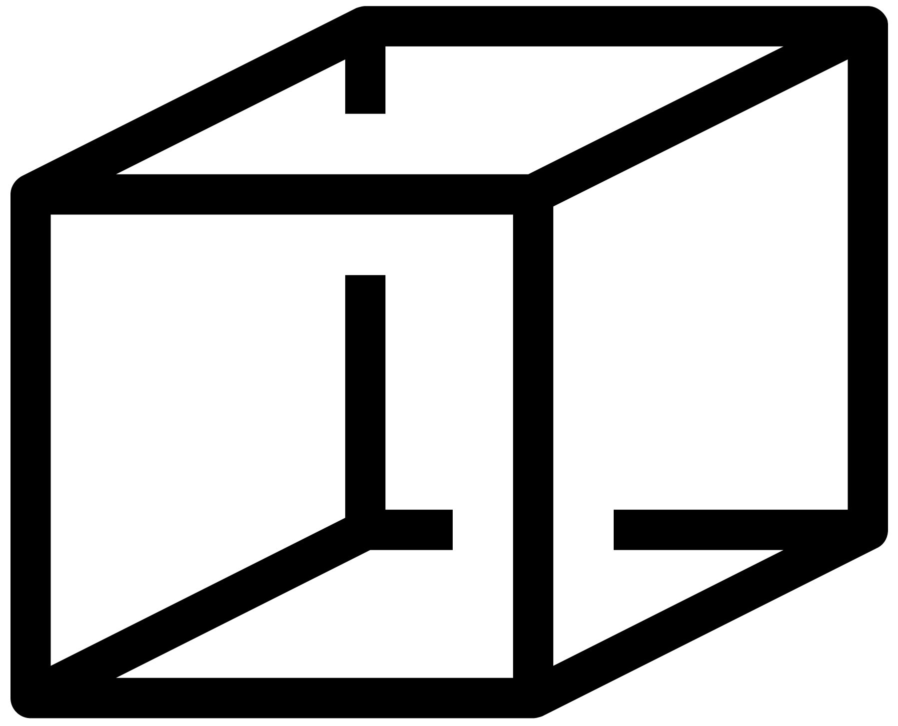
Bicubical Directed Type Theory



Matthew Weaver

General Examination
May 7th, 2018



Advised by Andrew Appel and Dan Licata

Bicubical Directed Type Theory

- Bicubical directed type theory is a constructive model of type theory
- It extends cubical type theory with an second notion of path that is directed
- We define a particularly well behaved universe of types in our model and construct directed univalence for this universe
- This is joint work with Dan Licata

What is it good for?

- Bicubical directed type theory provides a constructive setting for category theory
- Homotopy/cubical type theory has not only made it easier to formalize existing proofs in homotopy theory, but inspired new proofs
  - Directed type theory could do the same for category theory

What is it good for?

Today we'll focus on one specific application that seems most relevant to this audience:

formal verification of computational structures

A New Foundation for Formal Verification

- We're at a point where formal verification of real, large-scale software systems and computational structures is becoming tractable

CAKEML
A Verified Implementation of ML

seL4
Security. Performance. Proof.

deep spec

u r F erifi

wh ion le sc

mputational structures is becoming tractable

Core Spec

Kami

Verified Software Toolchain

CertiCoq

CAKEML

A Verified Implementation of ML

seL4

Security. Performance. Proof.

CERTIKOS

deep spec

Vellvm

verified LLVM

A New Foundation for Formal Verification

- While there has been some improvement, these proof-developments are unavoidably massive and time-consuming to develop
- The ease of verification is limited by the proof theory used in these projects
- Directed type theory provides a new setting for these proofs with primitives that correspond to fundamental concepts in computer science
- This change in foundational theory results in proofs and programs that are shorter and easier to write

But First:
The Simply Typed Lambda Calculus

(the old-fashioned way)

Let's Formalize STLC

- Let's define the simply typed lambda calculus inside of Agda,
- and then prove that our definition is invariant under weakening:

$$\frac{\Gamma \vdash t : \tau}{\Gamma,\ x : \tau' \vdash t : \tau}$$

- Warning: This may get a bit ugly

Let's Formalize STLC

```
data Ty : Type where
  A   : Ty
  _⇒_ : Ty → Ty → Ty
```

```
data Ctx : Type where
  •   : Ctx
  _,_ : Ctx → Ty → Ctx
```

Let's Formalize STLC

```
Var : Ctx → Type
Var •       = ⊥
Var (Γ , τ) = (Var Γ) + ⊤
```

Var ($x_1 : \tau_1, x_2 : \tau_2, \ldots, x_n : \tau_n$)
$:= \{x_1, x_2, \ldots, x_n\}$

```
data Tm (Γ : Ctx) : Type where
  var : Var Γ → Tm Γ
  abs : (τ : Ty) → Tm (Γ , τ) → Tm Γ
  app : Tm Γ → Tm Γ → Tm Γ
```

Tm t :=
| var x
| λ τ . t
| t t'

Let's Formalize STLC

```
getTy : (Γ : Ctx) → Var Γ → Ty
getTy • x = abort x
getTy (Γ , τ) (inr x) = τ
getTy (Γ , τ) (inl x) = getTy Γ x
```

Let's Formalize STLC

```
data _⊢_∈_ (Γ : Ctx) : Tm Γ → Ty → Type where

  tvar : (x : Var Γ)
       → ----------------------------
         Γ ⊢ var x ∈ getTy Γ x

  tabs : {τ τ' : Ty} {t : Tm (Γ , τ)}
         (_ : Γ , τ ⊢ t ∈ τ')
       → ----------------------------
         Γ ⊢ (abs τ t) ∈ τ ⇒ τ'

  tapp : {τ τ' : Ty} {t t' : Tm Γ}
         (_ : Γ ⊢ t ∈ τ ⇒ τ')
         (_ : Γ ⊢ t' ∈ τ)
       → ----------------------------
         Γ ⊢ app t t' ∈ τ'
```

Let's Formalize STLC

Now let's show everything is invariant under weakening of contexts

Let's Formalize STLC

```
wk-Var : ∀ Γ τ, Var Γ → Var (Γ , τ)
wk-Var Γ τ = inl
```

```
wk-Tm : ∀ Γ τ, Tm Γ → Tm (Γ , τ)
wk-Tm Γ τ (var x)    = var (wk-Var Γ τ x)
wk-Tm Γ τ (app t t') = app (wk-Tm Γ τ t)
                           (wk-Tm Γ τ t')
wk-Tm Γ τ (abs τ' t) = abs τ' ???     : Tm (Γ , τ , τ')

                   wk-Tm (Γ , τ') τ t : Tm (Γ , τ' , τ)
```

Let's Formalize STLC

```
Loc : Ctx → Type
Loc •         = ⊤
Loc (Γ , τ) = (Loc Γ) + ⊤
```

```
wk-Ctx :(Γ : Ctx) → Ty → Loc Γ → Ctx
wk-Ctx •          τ  l        = • , τ
wk-Ctx (Γ , τ') τ (inr l)  = (Γ , τ') , τ
wk-Ctx (Γ , τ') τ (inl l)  = (wk-Ctx Γ τ l) , τ'
```

l

$\Gamma = \bullet, \tau_1, ..., \tau_n, \tau_{n+1}, ...$ —wk-Ctx Γ τ l→

Let's Formalize STLC

```
Loc : Ctx → Type
Loc •        = ⊤
Loc (Γ , τ) = (Loc Γ) + ⊤
```

```
wk-Ctx :(Γ : Ctx) → Ty → Loc Γ → Ctx
wk-Ctx •          τ  l         = • , τ
wk-Ctx (Γ , τ') τ (inr l)  = (Γ , τ') , τ
wk-Ctx (Γ , τ') τ (inl l)  = (wk-Ctx Γ τ l) , τ'
```

l

$\Gamma = \bullet, \tau_1, ..., \tau_n, \tau_{n+1}, ...$ $\xrightarrow{\text{wk-Ctx } \Gamma\ \tau\ l}$ $\bullet, \tau_1, ..., \tau_n, \tau, \tau_{n+1}, ...$

Let's Formalize STLC

```
wk-Var : ∀ Γ τ l, Var Γ → Var (wk-Ctx Γ τ l)
wk-Var • τ l x = abort x
wk-Var (Γ , τ') τ (inr l) x = inl x
wk-Var (Γ , τ') τ (inl l) (inr x) = inr x
wk-Var (Γ , τ') τ (inl l) (inl x) = inl (wk-Var Γ τ l x)
```

Let's Formalize STLC

```
wk-Tm : ∀ Γ τ l, Tm Γ → Tm (wk-Ctx Γ τ l)
wk-Tm Γ τ l (var x)     = var (wk-Var Γ τ l x)
wk-Tm Γ τ l (app t t') = app (wk-Tm Γ τ l t)
                             (wk-Tm Γ τ l t')
wk-Tm Γ τ l (abs τ' t) = abs τ' (wk-Tm (Γ , τ') τ (inl l) t)
```

Let's Formalize STLC

```
wk-Tc : ∀ Γ τ l {t} {τ'}, Γ ⊢ t ∈ τ'
      → ------------------------------------
        (wk-Ctx Γ τ l) ⊢ (wk-Tm Γ τ l t) ∈ τ'

wk-Tc Γ τ l (tvar x)       = coe (λ τ' → _ ⊢ _ ∈ τ')
                                 (wk-getTy Γ τ l x)
                                 (tvar (wk-Var Γ τ l x))
wk-Tc Γ τ l (tabs tc)      = tabs (wk-Tc (Γ , _) τ (inl l) tc)
wk-Tc Γ τ l (tapp tc tc') = tapp (wk-Tc Γ τ l tc)
                                  (wk-Tc Γ τ l tc')
```

Let's Formalize STLC

- We know the only interesting part of weakening is its action on variables
- The type theory doesn't, resulting in verbose but trivial programs and proofs

Let's Formalize STLC

What if we want to weaken by multiple variables at once?

- We can iterate our previously defined weakening functions, which is inefficient but maintains our proof guarantees
- We can reimplement a more efficient version and redo all of the proofs

Let's Formalize STLC

- When it comes to weakening, we demonstrate there is an inclusion of the types in the type families

```
Var Γ ⊆ Var (Γ , τ)          Tm Γ ⊆ Tm (Γ , τ)
```

- Can we potentially gain insight by comparing this to subtyping?

```
Var Γ <: Var (Γ , τ)          Tm Γ <: Tm (Γ , τ)
```

Let's Formalize STLC

- Let's consider a type theory where we can specify that, for any type family `F : Ctx → Type`, it must be the case that `F Γ <: F (Γ , τ)`
- We would like this relation to have congruence rules, like subtyping
  - e.g. we can use that we know how to weaken variables to define how to weaken terms,
  - ... and use both of these to define how to weaken typing derivations

Let's Formalize STLC

- We don't want to restrict every `F : Ctx → Type` to those where there is a unique way for `F Γ <: F (Γ , τ)`
  - e.g. we could also implement our variables to be reversed (inside-out)
- Therefore this theory must keep track of which proof of this relation we are using: `p : F Γ <: F (Γ , τ)`
- p is a special function specifying how to turn a `F Γ` into a `F (Γ , τ)`

Let's Formalize STLC

- Thus, in this theory, A `<:` B has some qualities of subtyping, but is computationally relevant
- Can we define a theory with a notion that strike this balance between functions and subtyping? Yes!

Review of Subtyping

- We equip a type theory with a new judgement: `A <: B` for types `A` and `B`

$$\frac{t : A \quad A <: B}{t : B}$$

- Example:

```
record student : Type where              record person : Type where
  name     : String                        name     : String
  birthday : Date              <:          birthday : Date
  school   : String
```

Merging Subtyping and Functions

- As I already hinted towards a theory where subtyping looks like functions, let's be explicit when we use subtyping in our syntax:

$$\frac{t : A \quad A <: B}{t : B}$$

Merging Subtyping and Functions

- As I already hinted towards a theory where subtyping looks like functions, let's be explicit when we use subtyping in our syntax:

$$\frac{t : A \qquad A <: B}{\mathtt{cast}_{A<:B}\ t : B}$$

Merging Subtyping and Functions

- As I already hinted towards a theory where subtyping looks like functions, let's be explicit when we use subtyping in our syntax:

$$\frac{\texttt{A <: B}}{\texttt{cast}_{\texttt{A<:B}} : \texttt{A} \rightarrow \texttt{B}}$$

Merging Subtyping and Functions

- What might subtyping look like were it internally visible in the language?

$$\frac{\texttt{A : Type} \qquad \texttt{B : Type}}{\texttt{A <: B : Type}}$$

Merging Subtyping and Functions

- What might subtyping look like were it internally visible in the language?

$$\frac{\texttt{p : A <: B}}{\texttt{cast}_{\texttt{A<:B}}\ \texttt{p : A} \rightarrow \texttt{B}}$$

Merging Subtyping and Functions

- What might subtyping look like were it internally visible in the language?

$$\frac{\texttt{f : A → B}}{\texttt{dua f : A <: B}}$$

Merging Subtyping and Functions

- What might subtyping look like were it internally visible in the language?

$$\frac{\texttt{f : A} \to \texttt{B}}{\texttt{cast}_{\texttt{A<:B}}\ \texttt{(dua f)} \equiv_{\beta} \texttt{f}}$$

Merging Subtyping and Functions

- What might subtyping look like were it internally visible in the language?

$$\frac{\texttt{p : A <: B}}{\texttt{dua (cast}_{\texttt{A<:B}}\texttt{ p)} \equiv_\eta \texttt{p}}$$

Merging Subtyping and Functions

- What might subtyping look like were it internally visible in the language?

$$\frac{\texttt{A : Type} \qquad \texttt{B : Type}}{\texttt{A <: B : Type}}$$

$$\frac{\texttt{p : A <: B}}{\texttt{cast}_{\texttt{A<:B}}\ \texttt{p : A → B}} \qquad \frac{\texttt{f : A → B}}{\texttt{dua f : A <: B}}$$

$$\frac{\texttt{p : A <: B}}{\texttt{dua (cast}_{\texttt{A<:B}}\ \texttt{p)} \equiv_{\eta} \texttt{p}} \qquad \frac{\texttt{f : A → B}}{\texttt{cast}_{\texttt{A<:B}}\ \texttt{(dua f)} \equiv_{\beta} \texttt{f}}$$

Merging Subtyping and Functions

- Subtyping is now just a wrapper for the function type...
- ...with no additional structure or payoff.
- Yet.

Beyond Subtyping

- Let's think back to the STLC:
- Using this odd perspective of subtyping, we've proven that, `∀ Γ τ`,

```
Var Γ <: Var (Γ , τ)            Tm Γ <: Tm (Γ , τ)
```

- As mentioned before, we always want that, for every `F : Ctx → Type`,

```
F Γ <: F (Γ , τ)
```

Beyond Subtyping

- Let's extend our theory to make this restriction possible!
- As is typical in dependent type theory, let's not distinguish types and terms

```
A : Type    x : A    y : A
---------------------------
      x <: y : Type
```

Beyond Subtyping

- Let's extend our theory to make this restriction possible!
- As is typical in dependent type theory, let's not distinguish types and terms

$$\frac{\texttt{A : Type} \qquad \texttt{x : A} \qquad \texttt{y : A}}{\texttt{Hom x y : Type}}$$

- We call terms of `Hom x y` "morphisms" or "directed paths" from x to y

Beyond Subtyping

We equip every type with a proof-relevant binary relation that is reflexive

$$\frac{\texttt{x : A}}{\texttt{id x : Hom x x}}$$

Beyond Subtyping

We equip every type with a proof-relevant binary relation that is reflexive, transitive

$$\frac{\texttt{x y z : A} \qquad \texttt{p : Hom x y} \qquad \texttt{q : Hom y z}}{\texttt{p} \circ \texttt{q : Hom x z}}$$

Beyond Subtyping

We equip every type with a proof-relevant binary relation that is reflexive, transitive and congruent

```
   x : A            y : A
f : A → B    p : Hom x y
─────────────────────────
ap f p : Hom (f x) (f y)
```

- The type theory insures that all functions preserve this relation

Nontrivial Morphisms

- Now that we have morphisms in all types, let's define datatypes where this morphism structure is nontrivial
- The Idea: allow inductive types to include constructors for both terms and the morphisms
- The induction principle has cases corresponding to both kinds of constructors

Nontrivial Morphisms

```
data Ctx : Type where
  •   : Ctx
  _,_ : Ctx → Ty → Ctx
  wk  : ∀ Γ τ, Hom Γ (Γ , τ)
```

```
Ctx-rec : (A  : Type)
          (c₁ : A)
          (c₂ : A → Ty → A)
          (c₃ : ∀ a τ, Hom a (c₂ a τ))
        → ----------------------------
          Ctx → A
```

```
Ctx-rec A c₁ c₂ c₃ • ≡β c₁

Ctx-rec A c₁ c₂ c₃ (Γ , τ) ≡β c₂ (Ctx-rec A c₁ c₂ c₃ Γ) τ

ap (Ctx-rec A c₁ c₂ c₃) (wk Γ τ) ≡β c₃ (Ctx-rec A c₁ c₂ c₃ Γ) τ
```

Nontrivial Morphisms

```
data Ctx : Type where
  •   : Ctx
  _,_ : Ctx → Ty → Ctx
  wk  : ∀ Γ τ, Hom Γ (Γ , τ)
```

```
Ctx-rec : (A  : Type)
          (c₁ : A)
          (c₂ : A → Ty → A)
          (c₃ : ∀ a τ, Hom a (c₂ a τ))
        → -----------------------------
          Ctx → A
```

- Note there are no cases in both the definition and the recursion principle corresponding to the fact morphisms are reflexive, transitive and congruent

What's up with ap?

```
   x : A           y : A
f : A → B      p : Hom x y
──────────────────────────────
ap f p : Hom (f x) (f y)
```

- This rule states that everything is covariant:

Given `A A' B : Type`, and `p : Hom A A'`,

```
ap (λ X → (X → B)) p : Hom (A → B) (A' → B)
```

```
──────────────────────
A →           ' → B
```

What's up with `ap`?

- In this framework, morphisms in `Type` can be thought of as describing how two types are related, and are not (just) functions
  - Given `F : Type → Type`, `ap F` is the proof that F sends related inputs to related outputs
- We'd like to define a universe of types where morphisms are functions
- Let's call it UCov
  - Given `F : UCov → UCov`, `ap F` maps a function `f : A → B` to a function `ap F f : F A → F B`

Universe for Subtyping

- In order for F : A → UCov to typecheck, F must be covariant
  - e.g. λ X → (A → X) : UCov → UCov typechecks
  - e.g. λ X → (X → B) : UCov → UCov does not typecheck

```
      B <: B'
-----------------
A → B <: A → B'

      A <: A'
-----------------
A → B <: A' → B
```

- As morphisms coincide with functions, UCov is equipped with the following:

```
dua : {A B : UCov}
      (A → B)
    → -------------
      Hom A B
```

```
dcoe : {A : Type} (F : A → UCov)
       {x y : A} (p : Hom x y)
     → -------------------------
       F x → F y
```

Universe for Subtyping

```
dcoe : {A : Type} (F : A → UCov)
       {x y : A} (p : Hom x y)
     → -------------------------
       F x → F y
```

- As functions are morphisms in UCov, this is the same as saying:

$$\frac{F : A \to \mathrm{UCov} \qquad \mathrm{Hom}\ x\ y}{\mathrm{Hom}\ (F\ x)\ (F\ y)}$$

Universe for Subtyping

```
dcoe : {A : Type} (F : A → UCov)
       {x y : A} (p : Hom x y)
     → --------------------------
       F x → F y
```

- As functions are morphisms in UCov, this is the same as saying:

$$\frac{\texttt{F : A → UCov} \qquad \texttt{Hom x y}}{\texttt{F x <: F y}}$$

Universe for Subtyping

```
dcoe : {A : Type} (F : A → UCov)
       {x y : A} (p : Hom x y)
     → ---------------------------
       F x → F y
```

- As functions are morphisms in UCov, this is the same as saying:

$$\frac{F : Ctx \to UCov}{F\ \Gamma <: F\ (\Gamma\ ,\ \tau)}$$

Universe for Subtyping

- We can also prove that UCov is closed under various type-formers:

$$\frac{}{\top : \mathtt{UCov}}$$

$$\frac{}{\bot : \mathtt{UCov}}$$

$$\frac{A : \mathtt{UCov} \quad B : \mathtt{UCov}}{A \times B : \mathtt{UCov}}$$

$$\frac{A : \mathtt{UCov} \quad B : \mathtt{UCov}}{A + B : \mathtt{UCov}}$$

$$\frac{F : \mathtt{UCov} \to \mathtt{UCov} \text{ polynomial}}{\mu\ F : \mathtt{UCov}}$$

(i.e. inductive types)

Universe for Subtyping

```
A : UCov    B : UCov
--------------------
  A × B : UCov
```

- Because we have `dcoe` for `UCov`, this closure property is a proof that there is a unique solution to the following:

```
 A <: A'    B <: B'
--------------------
 A × B <: A' × B'
```

- Thus, by working in `UCov`, we get the congruence properties we wanted

The Payoff

Let's check out what it's like to use this type theory

Let's Formalize STLC (Again)

- 0
+ 0

```
data Ty : Type where
  A   : Ty
  _⇒_ : Ty → Ty → Ty
```

Let's Formalize STLC (Again)

- 0
+ 0

```
data Ty : UCov where
  A   : Ty
  _⇒_ : Ty → Ty → Ty
```

Let's Formalize STLC (Again)

- 0
+ 0

```
data Ctx : Type where
  •   : Ctx
  _,_ : Ctx → Ty → Ctx
```

☹️

Let's Formalize STLC (Again)

- 0
+ 1

```
data Ctx : Type where
  •   : Ctx
  _,_ : Ctx → Ty → Ctx
  wk  : ∀ Γ τ, Hom Γ (Γ , τ)
```

Let's Formalize STLC (Again)

- 0

+ 1

```
Var : Ctx → Type
Var •          = ⊥
Var (Γ , τ)  = (Var Γ) + ⊤
```

Let's Formalize STLC (Again)

- 0

+ 2

```
Var : Ctx → UCov
Var •          = ⊥
Var (Γ , τ)  = (Var Γ) + ⊤
Var (wk Γ τ) = dua inl     : Hom (Var Γ) (Var (Γ , τ))
```

Let's Formalize STLC (Again)

- 0
+ 2

```
data Tm (Γ : Ctx) : Type where
  var : Var Γ → Tm Γ
  abs : (τ : Ty) → Tm (Γ , τ) → Tm Γ
  app : Tm Γ → Tm Γ → Tm Γ
```

Let's Formalize STLC (Again)

- 0

+ 2

```
data Tm (Γ : Ctx) : UCov where
  var : Var Γ → Tm Γ
  abs : (τ : Ty) → Tm (Γ , τ) → Tm Γ
  app : Tm Γ → Tm Γ → Tm Γ
```

Let's Formalize STLC (Again)

- 0

+ 2

Let's first consider weakening terms

Let's Formalize STLC (Again)

- 0
+ 2

```
Loc : Ctx → Type
Loc •          = ⊤
Loc (Γ , τ)   = (Loc Γ) + ⊤
```

```
wk-Ctx : (Γ : Ctx) → Ty → Loc Γ → Ctx
wk-Ctx •           τ   l         = • , τ
wk-Ctx (Γ , τ') τ (inr l)  = (Γ , τ') , τ
wk-Ctx (Γ , τ') τ (inl l)  = (wk-Ctx Γ τ l) , τ'
```

☹️

Let's Formalize STLC (Again)

- 0
+ 2

```
wk-Var : ∀ Γ τ l, Var Γ → Var (wk-Ctx Γ τ l)
wk-Var • τ l x = abort x
wk-Var (Γ , τ') τ (inr l) x = inl x
wk-Var (Γ , τ') τ (inl l) (inr x) = inr x
wk-Var (Γ , τ') τ (inl l) (inl x) = inl (wk-Var Γ τ l x)
```

Let's Formalize STLC (Again)

- 8

+ 2

```
wk-Var : ∀ Γ τ, Var Γ → Var (Γ , τ)
wk-Var Γ τ = dcoe Var (wk Γ τ)
```

```
dcoe : {A : Type} (F : A → UCov)
       {x y : A} (p : Hom x y)
     → ---------------------------
       F x → F y
```

Let's Formalize STLC (Again)

- 8

+ 2

```
wk-Tm : ∀ Γ τ l, Tm Γ → Tm (wk-Ctx Γ τ l)
wk-Tm Γ τ l (var x)     = var (wk-Var Γ τ l x)
wk-Tm Γ τ l (abs τ' t) = abs τ' (wk-Tm (Γ , τ') τ (inl l) t)
wk-Tm Γ τ l (app t t') = app (wk-Tm Γ τ l t)
                             (wk-Tm Γ τ l t')
```

☹️☹️

Let's Formalize STLC (Again)

- 11
+ 2

```
wk-Tm : ∀ Γ τ, Tm Γ → Tm (Γ , τ)
wk-Tm Γ τ = dcoe Tm (wk Γ τ)
```

😊😊😊

Let's Formalize STLC (Again)

That's not fair, though:
I only implemented the outermost weakening in our new theory...right?

Wrong!!!

Let's Formalize STLC (Again)

```
wk' : ∀ Γ τ τ', Hom (Γ , τ) (Γ , τ' , τ)
wk' Γ τ τ' = ap (λ Γ → Γ , τ) (wk Γ τ')
```

```
              x : A            y : A
f : A → B         p : Hom x y
─────────────────────────────────────
ap f p : Hom (f x) (f y)
```

```
wk-Var' : ∀ Γ τ τ', Var (Γ , τ) → Var (Γ , τ' , τ)
wk-Var' Γ τ τ' = dcoe Var (wk' Γ τ τ')
```

```
wk-Tm' : ∀ Γ τ τ', Tm (Γ , τ) → Tm (Γ , τ' , τ)
wk-Tm' Γ τ τ' = dcoe Tm (wk' Γ τ τ')
```

Let's Formalize STLC (Again)

```
wk'' : ∀ Γ τ τ', Hom Γ (Γ , τ , τ')
wk'' Γ τ τ' = wk Γ τ ∘ wk (Γ , τ) τ'
```

```
wk-Var'' : ∀ Γ τ τ', Var Γ → Var (Γ , τ , τ')
wk-Var'' Γ τ τ' = dcoe Var (wk'' Γ τ τ')
```

```
wk-Tm'' : ∀ Γ τ τ', Tm Γ → Tm (Γ , τ , τ')
wk-Tm'' Γ τ τ' = dcoe Tm (wk' Γ τ τ')
```

Let's Formalize STLC (Again)

- In general, we specify that we want to weaken from Γ to `Γ'` by providing a morphisms from Γ to `Γ'`
  - Before, this data was provided by a triple containing a context, location in that context and the type by which to weaken
- `dcoe F` is the function that executes weakening for the type family F
- In summary: the type theory implemented weakening by arbitrary many variables in arbitrary locations automatically!

Let's Formalize STLC (Again)

- 11
+ 2

Now let's quickly consider weakening our typing derivations

(Note: this is more speculative than what's been shown previously)

Let's Formalize STLC (Again)

- 11
+ 3

```
getTy : (Γ : Ctx) → Var Γ → Ty
getTy • x = abort x
getTy (Γ , τ) (inr x) = τ
getTy (Γ , τ) (inl x) = getTy Γ x
getTy (wk Γ τ) = id (getTy Γ) : Hom (λ x → getTy Γ x)
                                    (λ x → getTy (Γ , τ) (inl x))
```

Let's Formalize STLC (Again)

```
data _⊢_∈_ (Γ : Ctx) : Tm Γ → Ty → UCov where

  tvar : (x : Var Γ)
       → ----------------------------
         Γ ⊢ var x ∈ getTy Γ x

  tabs : {τ τ' : Ty} {t : Tm (Γ , τ)}
         (_ : Γ , τ ⊢ t ∈ τ')
       → ----------------------------
         Γ ⊢ (abs τ t) ∈ τ ⇒ τ'

  tapp : {τ τ' : Ty} {t t' : Tm Γ}
         (_ : Γ ⊢ t ∈ τ ⇒ τ')
         (_ : Γ ⊢ t' ∈ τ)
       → ----------------------------
         Γ ⊢ app t t' ∈ τ'
```

- 11

+ 3

Let's Formalize STLC (Again)

- 11
+ 3

```
wk-Tc : ∀ Γ τ l {t} {τ'}, Γ ⊢ t ∈ τ'
      → -----------------------------------
        (wk-Ctx Γ τ l) ⊢ (wk-Tm Γ τ l t) ∈ τ'

wk-Tc Γ τ l (tvar x)        = coe (λ τ' → _ ⊢ _ ∈ τ')
                                  (wk-getTy Γ τ l x)
                                  (tvar (wk-Var Γ τ l x))
wk-Tc Γ τ l (tabs tc)       = tabs (wk-Tc (Γ , _) τ (inl l) tc)
wk-Tc Γ τ l (tapp tc tc') = tapp (wk-Tc Γ τ l tc)
                                   (wk-Tc Γ τ l tc')
```

😭😭😭😭

Let's Formalize STLC (Again)

- 26

+ 3

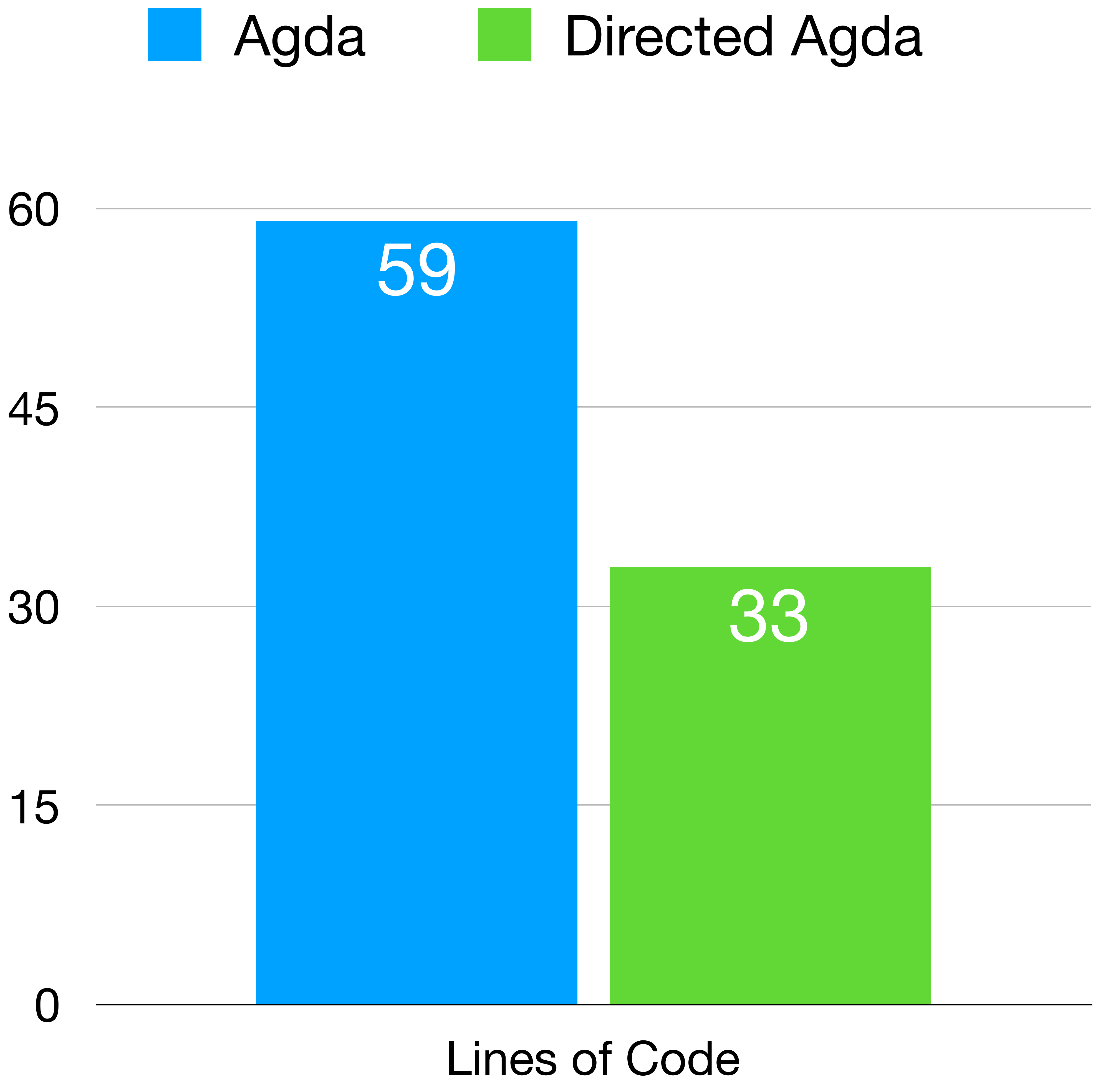
```
wk-Tc : ∀ Γ τ {t} {τ'}, Γ ⊢ t ∈ τ'
      → -----------------------------------
        Γ , τ ⊢ (wk-Tm Γ τ t) ∈ τ'

wk-Tc Γ τ l = dcoe (λ (Γ , t) → Γ ⊢ t ∈ τ') (ΣHom Tm (wk Γ τ) t)


(ΣHom Tm (wk Γ τ) t : Hom (Γ          , t)
                          ((Γ , τ) , dcoe Tm (wk Γ τ) t)
```

🤩🤩🤩🤩🤩

Let's Formalize STLC (Again)



Let's Formalize STLC (Again)

```
weak Tm (wk Γ τ ∘ wk (Γ , τ) τ') : Tm Γ → Tm (Γ, τ, τ')
```

- This function traverses the term once, and at each variable applies the function `inl` twice
- We get generic programs for free with
  - strong semantic guarantees
  - efficient computation

Let's Formalize STLC (Again)

- We can internally witness that weakening for `Tm` and type checking is *uniquely* determined by `Var : Ctx → UCov`
  - The definition we get for free must be the one we wrote by hand before
  - We can use this fact in later proofs!

So how do we make any of this work?

Math!!!

Defining Bicubical Directed Type Theory

- We define this type theory using categorical semantics
- Types are interpreted as mathematical objects called bicubical sets
- It is an extension of the model of cartesian cubical type theory by Carlo Anguli, Guillaume Brunerie, Thierry Coquand, Favonia, Bob Harper and Dan Licata
- Our approach to augmenting their work with directed paths is based off of the work of Emily Riehl and Mike Shulman that uses bisimplicial sets (as opposed to bicubical sets)
- We construct our universe internally using a method developed by Dan Licata, Ian Orton, Andy Pitts and Bas Spitters

Defining Bicubical Directed Type Theory

- Like the cartesian cubical model, our model is constructive
  - i.e. everything actually computes
- Our main contribution is the *construction* of a covariant universe `UCov` s.t.
  - $\mathtt{A} \to \mathtt{B} \simeq \mathtt{Hom}_{\mathtt{UCov}}\ \mathtt{A}\ \mathtt{B}$
  - This equivalence is called directed univalence
  - (caveat: we currently only have a constructive proof that $\mathtt{A} \to \mathtt{B}$ is a retract of $\mathtt{Hom}_{\mathtt{UCov}}\ \mathtt{A}\ \mathtt{B}$)

Our Formalization

- Our approach to this is based off of that done by Ian Orton and Andy Pitts
- Use Agda...
  - ...but only Π, Σ, ≡ w/ `uip`, ⊤, ⊥, `Prop`
- Build theory as a shallow embedding in this basic dependent type theory

Our Formalization

- Types and terms of Agda coincide with the types and terms of our model
- We use `_≡_` to encode the judgmental equality in our model
  - More generally, we use `Prop` to contain judgements of the metatheory of our model
- Precisely corresponds to a categorical model of type theory
  - Despite this fact, is 100% syntactic

Our Formalization

```
Hom : (A : Type) → A → A → Type
Hom A x y = Σ p : 𝟚 → A , p 𝟘 ≡ x × p 𝟙 ≡ y
```

Our Formalization

```
dcom-dua : ∀ {l1 l2 : Level} {Γ : Set l1}
             (x : Γ → 2)
             (A B : Γ → Set l2)
             (f : (θ : Γ) → A θ → B θ)
             → relCov A
             → relCov B
             → relCov1 (duaF x A B f)
dcom-dua x A B f dcomA dcomB p α t b =
  glue _ _ _ _
           (v-elimd01 _ (\ xp1=0  → fst (tleft xp1=0))
                        (\ xp1=1  →  fst b' ))
           (fst b' ,
            v-elimd01 _ (\ xp1=0 → fst (snd b') (inr xp1=0))
                        (\ xp1=1 → id)) ,
           (\ pα → glue-cong _ _ _ _ _ _ _
                        (λ= (v-elimd01 _
                               (\ xp1=0 → ! (tleft-α pα xp1=0))
                               (\ xp1=1 → fst (snd b') (inl pα) ∘ unglue-α (t ``1 pα) (inr xp1=1) )))
                        (fst (snd b') (inl pα)) ∘ Glueη (t ``1 pα)) where

  back-in-time : ((x o p) ``1 == ``0) → (y : _) → (x o p) y == ``0
  back-in-time eq y = transport (\ h → (x o p) y ≤ h) eq (dimonotonicity≤ (x o p) y ``1 id)

  -- when the result in is in A, compose in A
  tleft-fill : (y : 2) (xp1=0 : x (p ``1) == ``0) → _
  tleft-fill y xp1=0 =
    dcomA p y α
           (\ z pα → coe (Glue-α _ _ _ _ (inl (back-in-time xp1=0 z))) (t z pα))
           (coe (Glue-α _ _ _ _ (inl (back-in-time xp1=0 ``0 ))) (fst b) ,
               (λ pα → ((ap (coe (Glue-α _ _ _ _ (inl _))) (snd b pα)) ∘ ap (\ h → (coe (Glue-α _ _ _ _ (inl h)) (t ``0 pα))) uip)))

  tleft = tleft-fill ``1

  -- on α, the composite in A is just t 1
  tleft-α : (pα : α) → (xp1=0 : x(p ``1) == ``0) →
         fst (tleft xp1=0) == coe (Glue-α _ _ _ _ (inl xp1=0)) (t ``1 pα)
  tleft-α pα xp1 = (ap (\ h → coe (Glue-α _ _ _ _ (inl h)) (t ``1 pα)) uip) ∘ ! (fst (snd (tleft xp1)) pα)

  -- unglue everyone to B and compose there, agreeing with f (tleft-fill) on xp1 = 0
  b' : Σ \ (b' : B (p ``1)) → _
  b' = dcomB p ``1
             (α ∨ (x (p ``1) == ``0))
             ((\ z → case (\ pα →  unglue (t z pα))
                          (\ xp1=0 → f (p z) (fst (tleft-fill z xp1=0)))
                          (\ pα xp1=0 → ap (f (p z)) (fst (snd (tleft-fill z xp1=0)) pα) ∘ ! (unglue-α (t z pα) (inl (back-in-time xp1=0 z)))  )))
             (unglue (fst b) ,
               v-elim _ (\ pα → ap unglue (snd b pα))
                        (\ xp1=0 → unglue-α (fst b) (inl (back-in-time xp1=0 ``0 )) ∘ ! (ap (f (p ``0)) (snd (snd (tleft-fill ``0 xp1=0)) id)) )
                        (\ _ _ → uip) )
```

Future Directions

- Directed Higher Inductive Types
  - A general theory for types like `Ctx`
- Extended "real world" application(s) in verification
  - i.e. demonstrate directed type theory actually works and is helpful in "the wild" (e.g. real(ish) compiler, etc...)

Bicubical Directed Type Theory

- We've defined a constructive model of type theory that extends cubical type theory with
  - Directed paths
  - A covariant universe with directed univalence (81.25%)
- These new features can make formal verification easier
- We still have to develop more of the theory (i.e. DHITs) before we can use it in practice