THE UNIVERSITY OF BRITISH COLUMBIA

# Approximate Normalization for Gradual Dependent Types

*Joseph Eremondi, Éric Tanter, Ronald Garcia*
*University of British Columbia, University of Chile/INRIA*
*ICFP 2019*

# Our Contributions

# GDTL:
# Gradual Dependently Typed Language

## GDTL:

## Gradual Dependently Typed Language

- Full spectrum, universe hierarchy

## GDTL:

## Gradual Dependently Typed Language

- Full spectrum, universe hierarchy
- Can replace any type or term with **?**

# GDTL:

# Gradual Dependently Typed Language

- Full spectrum, universe hierarchy
- Can replace any type or term with **?**
- Embeds fully typed & untyped calculi

# GDTL:

# Gradual Dependently Typed Language

- Full spectrum, universe hierarchy
- Can replace any type or term with **?**
- Embeds fully typed & untyped calculi
- Decidable typechecking

# Our Contributions

## GDTL:

## Gradual Dependently Typed Language

- Full spectrum, universe hierarchy
- Can replace any type or term with **?**
- Embeds fully typed & untyped calculi
- Decidable typechecking

This talk

# Our Contributions

## GDTL:

## Gradual Dependently Typed Language

- Full spectrum, universe hierarchy
- Can replace any type or term with **?**
- Embeds fully typed & untyped calculi
- Decidable typechecking

This talk

- Proof of gradual type safety

**GDTL:**

**Gradual Dependently Typed Language**

- Full spectrum, universe hierarchy
- Can replace any type or term with **?**
- Embeds fully typed & untyped calculi
- Decidable typechecking

$\Bigg\}$ This talk

- Proof of gradual type safety
- *Gradual Guarantees* (Siek et al 2015): reducing precision of term won't create new static or dynamic failures

# GDTL:

# Gradual Dependently Typed Language

- Full spectrum, universe hierarchy
- Can replace any type or term with **?**
- Embeds fully typed & untyped calculi
- Decidable typechecking

This talk

- Proof of gradual type safety
- *Gradual Guarantees* (Siek et al 2015): reducing precision of term won't create new static or dynamic failures

See paper

# Why Gradual Dependent Types?

# Dependent Types: Pain and Promise

# Expectation

```
$> compile ./myprogram
>> 0 bugs detected!
```

## Expectation

```
$> compile ./myprogram
>> 0 bugs detected!
```

## Reality

```
$> compile ./myprogram
>> Type mismatch between
        Vec Nat (m+n)
    and
        Vec Nat (n+m)
```

## Expectation

```
$> compile ./myprogram
>> 0 bugs detected!
```

## Reality

```
$> compile ./myprogram
>> Type mismatch between
       Vec Nat (m+n)
     and
       Vec Nat (n+m)
```

- Popular for proof assistants

## Expectation

```
$> compile ./myprogram
>> 0 bugs detected!
```

## Reality

```
$> compile ./myprogram
>> Type mismatch between
       Vec Nat (m+n)
   and
       Vec Nat (n+m)
```

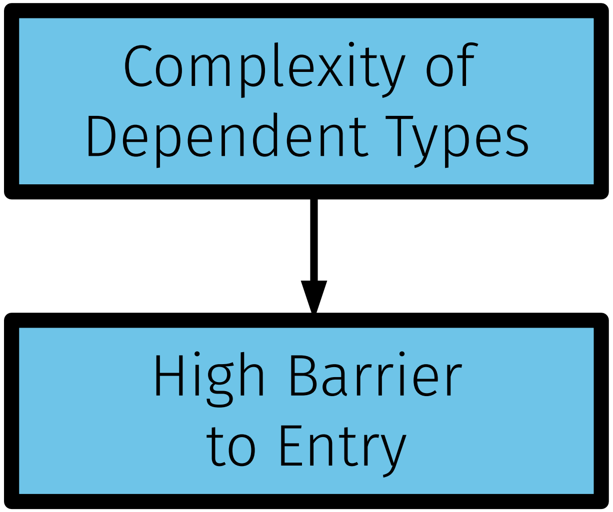
- Popular for proof assistants
- Not popular for programming

# A Vicious Cycle

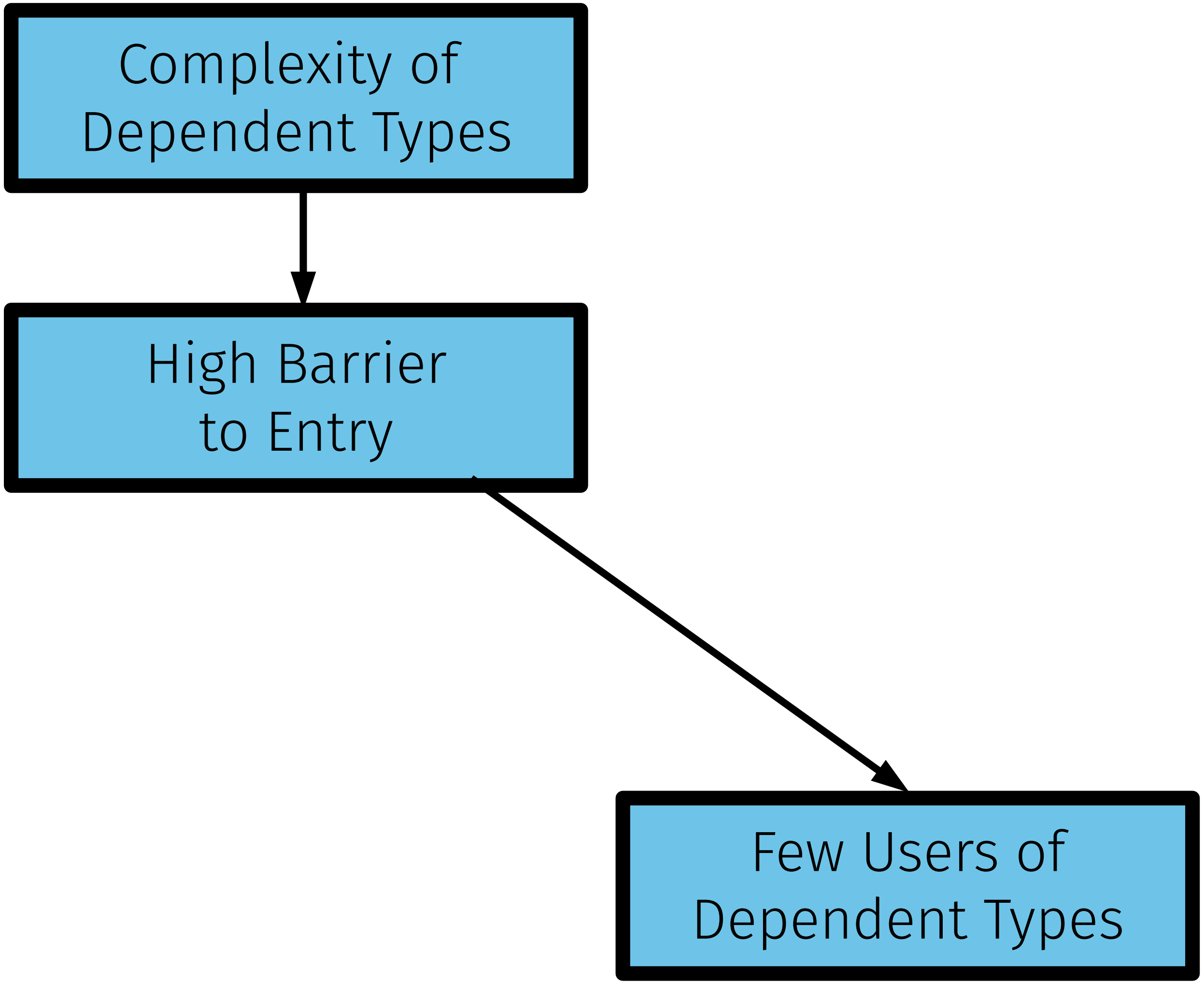
# A Vicious Cycle

Complexity of Dependent Types

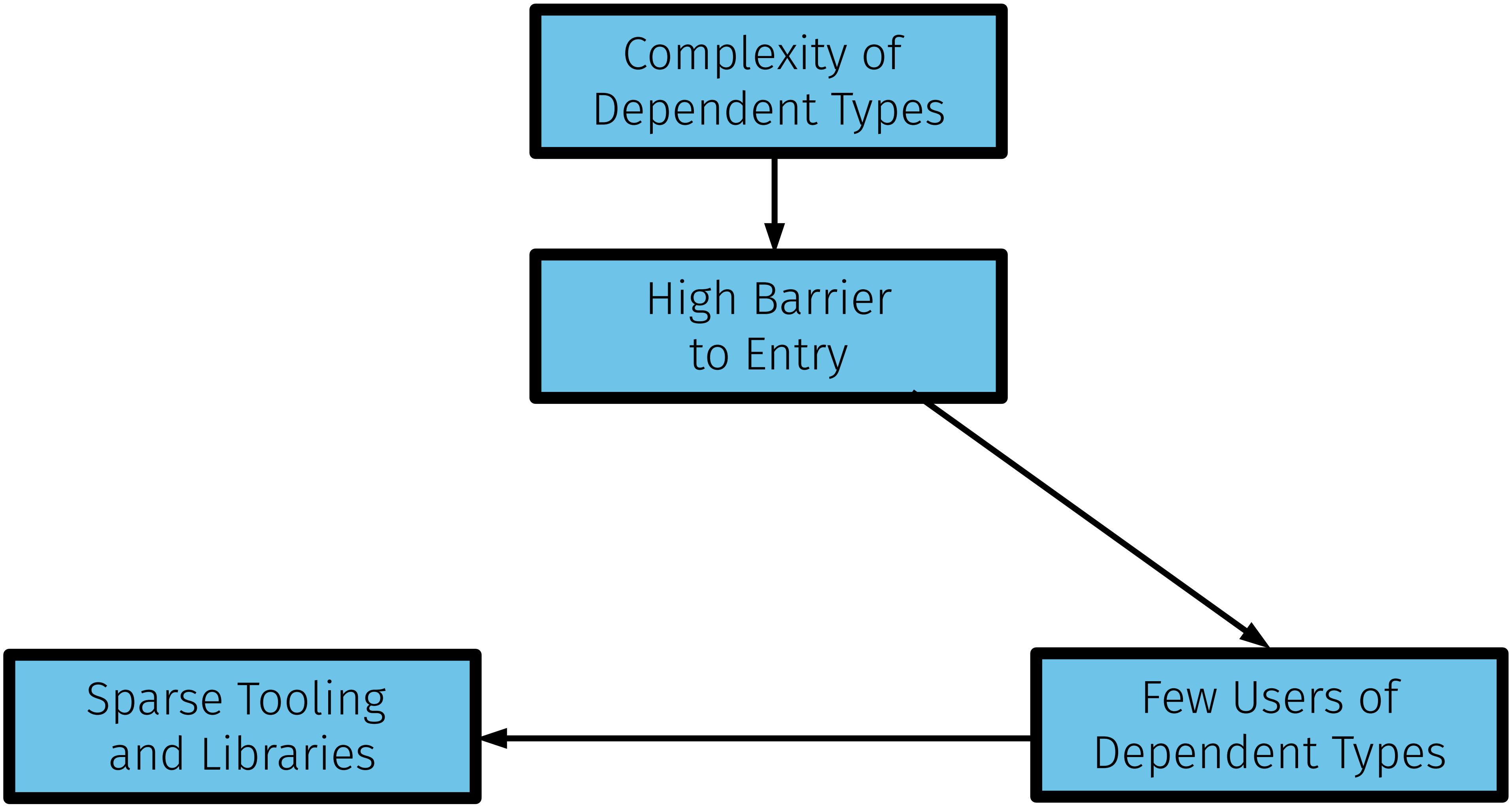
# A Vicious Cycle



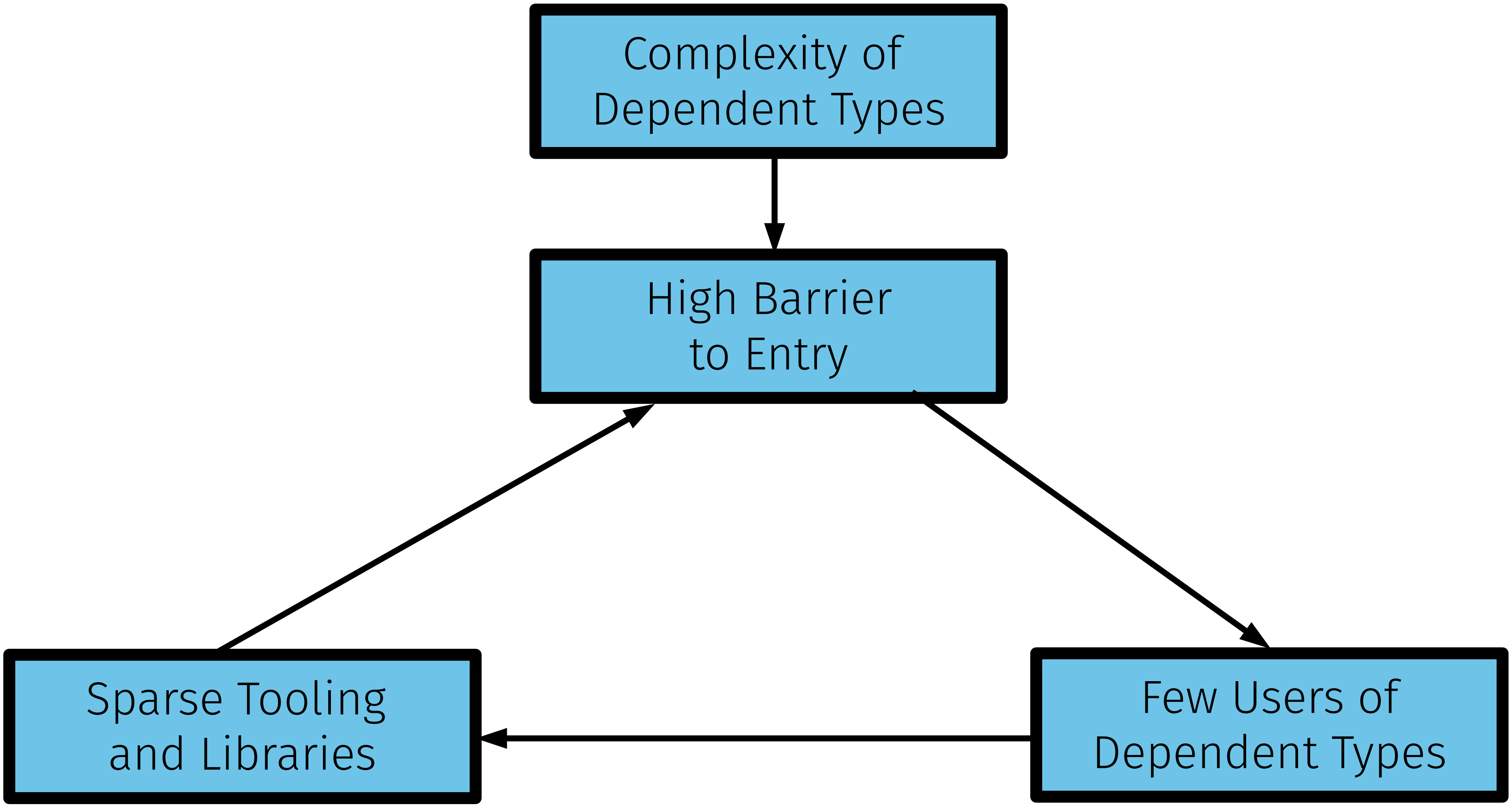
# A Vicious Cycle



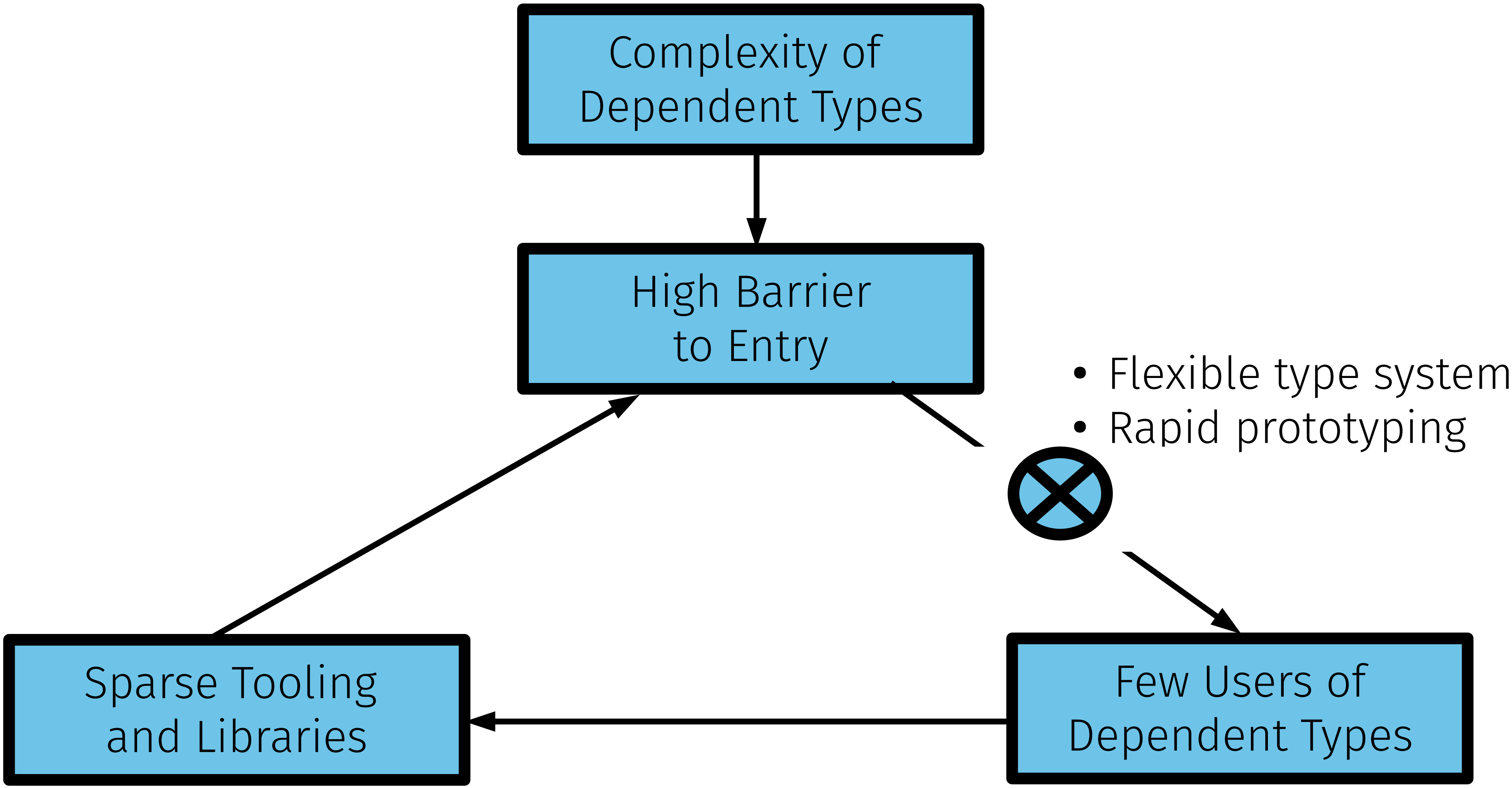
# A Vicious Cycle



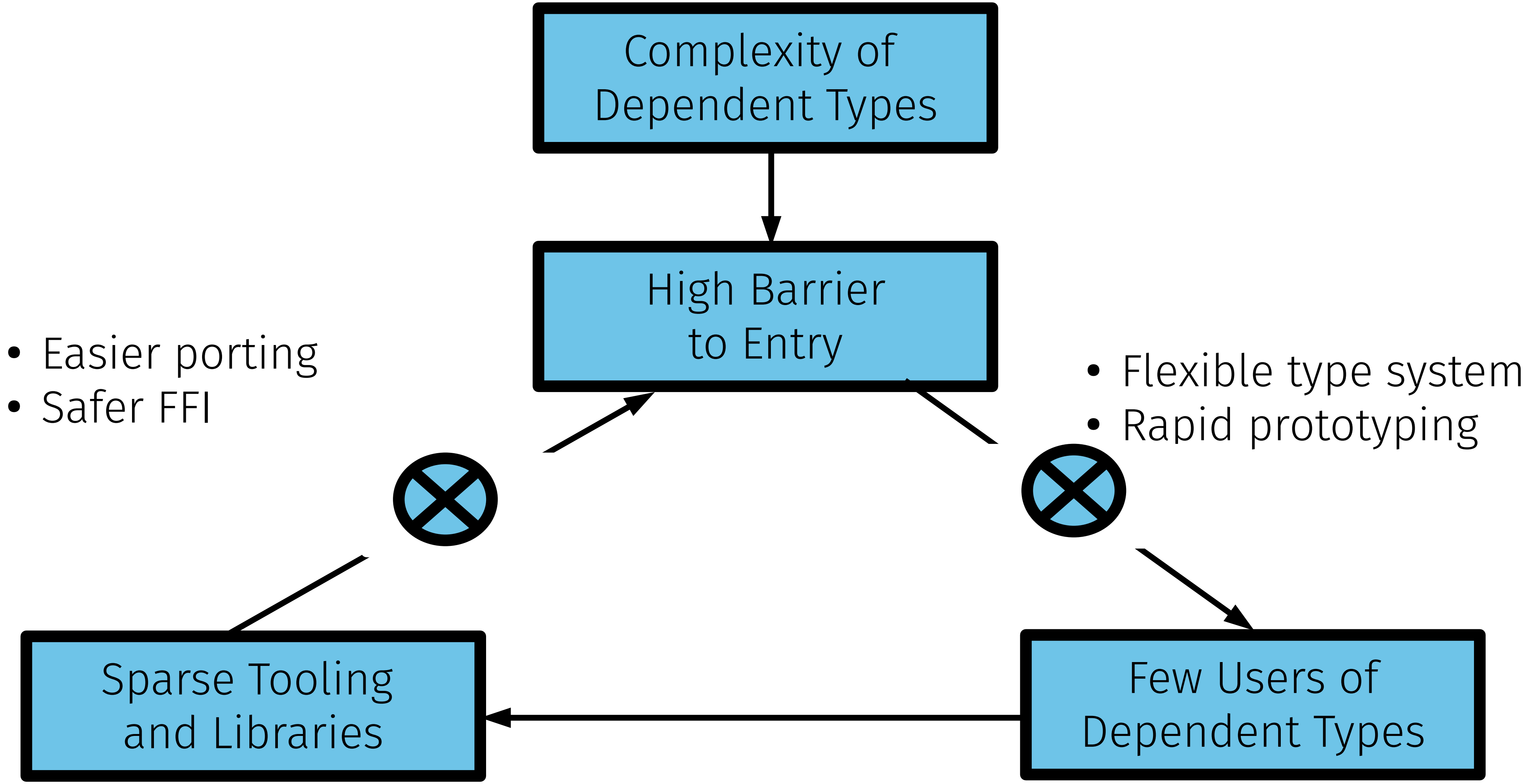
# A Vicious Cycle



# A Vicious Cycle



# A Vicious Cycle



# Goals For
# Gradual Dependent Types

# Motivation: Lists vs. Vectors

# Motivation: Lists vs. Vectors

```
data List a
  where
    Nil : List a
    Cons : a
         -> List a
         -> List a

head : List a -> a
```

No size knowledge
in type

```
data List a
  where
    Nil : List a
    Cons : a
         -> List a
         -> List a

head : List a -> a
```

No size knowledge in type

```
data List a
  where
    Nil : List a
    Cons : a
         -> List a
         -> List a
```

```
head : List a -> a
```

Error on `Nil`

No size knowledge in type

```
data List a
  where
    Nil : List a
    Cons : a
         -> List a
         -> List a

head : List a -> a
```

Error on `Nil`

No size knowledge in type

```
data List a
  where
    Nil : List a
    Cons : a
         -> List a
         -> List a

head : List a -> a
```

```
data Vec (a : Type) (n : Nat)
 where
    Nil : Vec a 0
    Cons : a
         -> Vec a n
         -> Vec a (n + 1)

head : Vec a (n + 1) -> a
```

Error on `Nil`

# Motivation: Lists vs. Vectors

No size knowledge in type

```
data List a
  where
    Nil : List a
    Cons : a
         -> List a
         -> List a

head : List a -> a
```

Error on `Nil`

Length in *type index*

```
data Vec (a : Type) (n : Nat)
 where
    Nil : Vec a 0
    Cons : a
         -> Vec a n
         -> Vec a (n + 1)

head : Vec a (n + 1) -> a
```

# Motivation: Lists vs. Vectors

No size knowledge in type

```
data List a
  where
    Nil  : List a
    Cons : a
         -> List a
         -> List a

head : List a -> a
```

Error on `Nil`

Length in *type index*

```
data Vec (a : Type) (n : Nat)
 where
    Nil  : Vec a 0
    Cons : a
         -> Vec a n
         -> Vec a (n + 1)
```

```
head : Vec a (n + 1) -> a
```

Won't typecheck for `Nil`

# Porting Code to Dependent Types

```
sort : List Int -> List Int
```

## Porting Code to Dependent Types

```
sort : List Int -> List Int

sort Nil = Nil
```

```
sort : List Int -> List Int
sort Nil = Nil
sort (Cons head tail) =
```

```
sort : List Int -> List Int

sort Nil = Nil

sort (Cons head tail) =
  sort (filter (<= head) tail))
```

```
sort : List Int -> List Int

sort Nil = Nil

sort (Cons head tail) =
  sort (filter (<= head) tail))
  ++ [head]
```

```
sort : List Int -> List Int

sort Nil = Nil

sort (Cons head tail) =
  sort (filter (<= head) tail))
  ++ [head]
  ++ sort (filter (> head) tail))
```

```
sort : List Int -> List Int

sort Nil = Nil

sort (Cons head tail) =
  sort (filter (<= head) tail))
  ++ [head]
  ++ sort (filter (> head) tail))


filter : (Int -> Bool)
       -> List Int -> List Int
```

```
       Vec Int n
sort : List Int -> List Int

sort Nil = Nil

sort (Cons head tail) =
  sort (filter (<= head) tail))
  ++ [head]
  ++ sort (filter (> head) tail))


filter : (Int -> Bool)
       -> List Int -> List Int
```

```
       Vec Int n    Vec Int n
sort : List Int -> List Int

sort Nil = Nil

sort (Cons head tail) =
  sort (filter (<= head) tail))
  ++ [head]
  ++ sort (filter (> head) tail))


filter : (Int -> Bool)
       -> List Int -> List Int
```

# Porting Code to Dependent Types

```
         Vec Int n    Vec Int n
sort : List Int -> List Int

sort Nil = Nil ✓

sort (Cons head tail) =
  sort (filter (<= head) tail)
  ++ [head]
  ++ sort (filter (> head) tail)


filter : (Int -> Bool)
       -> List Int -> List Int
```

# Porting Code to Dependent Types

```
       Vec Int n   Vec Int n
sort : List Int -> List Int

sort Nil = Nil ✓

sort (Cons head tail) =
  sort (filter (<= head) tail))
  ++ [head]
  ++ sort (filter (> head) tail))
```

Need proof that lengths sum to n

```
filter : (Int -> Bool)
       -> List Int -> List Int
```

# Porting Code to Dependent Types

`Vec Int n` `Vec Int n`

`sort :` ~~`List Int`~~ `->` ~~`List Int`~~

```
sort Nil = Nil ✓

sort (Cons head tail) =
  sort (filter (<= head) tail))
  ++ [head]
  ++ sort (filter (> head) tail))
```

Need proof that lengths sum to n

```
filter : (Int -> Bool)
```

`->` ~~`List Int`~~ `-> List Int`

`Vec Int n`

# Porting Code to Dependent Types

```
          Vec Int n    Vec Int n
sort : List Int -> List Int

sort Nil = Nil ✓

sort (Cons head tail) =
  sort (filter (<= head) tail))
  ++ [head]
  ++ sort (filter (> head) tail))
```

Need proof that lengths sum to n

```
filter : (Int -> Bool)
         -> List Int -> List Int
            Vec Int n   Vec Int ___
```

# How To Solve?

# How To Solve?

| | | |
|---|---|---|
| Static Dependent Types | | |
| | | |
| | | |

# How To Solve?

| | | |
|---|---|---|
| Static Dependent Types | Existential Types, Inductive Proof | |
| | | |
| | | |

# The Static Approach

# The Static Approach

```
rewriteFilterLength :
```

# The Static Approach

```
rewriteFilterLength :
  (v : Vec Int n)
```

# The Static Approach

```
rewriteFilterLength :
  (v : Vec Int n)
  -> (p : Int -> Bool)
```

## The Static Approach

```
rewriteFilterLength :
  (v : Vec Int n)
  -> (p : Int -> Bool)
  -> Vec Int
       (length (filter p v)
        + 1 + length (filter (not . p) v))
```

```
rewriteFilterLength :
  (v : Vec Int n)
  -> (p : Int -> Bool)
  -> Vec Int
        (length (filter p v)
         + 1 + length (filter (not . p) v)
  ->  Vec Int n
```

```
rewriteFilterLength :
  (v : Vec Int n)
  -> (p : Int -> Bool)
  -> Vec Int
        (length (filter p v)
         + 1 + length (filter (not . p) v)
  ->  Vec Int n
```

Relies on induction, commutativity, etc.

```
sort : Vec Int n -> Vec Int n

sort Nil = Nil

sort (Cons head tail) =

  sort (filter (<= head) tail))
  ++ [head]
  ++ sort (filter (> head) tail))


filter : (Int -> Bool)
       -> Vec Int n ->   Vec Int ?
```

# Gradual Proof Terms

```
sort : Vec Int n -> Vec Int n

sort Nil = Nil

sort (Cons head tail) =
  rewriteFilterLength (
  sort (filter (<= head) tail))
  ++ [head]
  ++ sort (filter (> head) tail))
                                 )

filter : (Int -> Bool)
       -> Vec Int n -> Vec Int ?
```

# How To Solve?

| | | |
|---|---|---|
| Static Dependent Types | Existential Types, Inductive Proof | |
| | | |
| | | |

## How To Solve?

| Static Dependent Types | Existential Types, Inductive Proof | ✗ Significant effort required |
|---|---|---|
| | | |
| | | |

## How To Solve?

| | | |
|---|---|---|
| Static Dependent Types | Existential Types, Inductive Proof | ✗ Significant effort required |
| Non-dependent Gradual Types | | |
| | | |

# How To Solve?

| | | |
|---|---|---|
| Static Dependent Types | Existential Types, Inductive Proof | ✗ Significant effort required |
| Non-dependent Gradual Types | `filter` returns **`?`**<br>unknown *<u>type</u>* | |
| | | |

# How To Solve?

| | | |
|---|---|---|
| Static Dependent Types | Existential Types, Inductive Proof | ✗ Significant effort required |
| Non-dependent Gradual Types | `filter` returns **`?`** unknown *type* | ✗ Can have non-list return |
| | | |

# How To Solve?

| | | |
|---|---|---|
| Static Dependent Types | Existential Types, Inductive Proof | ✗ Significant effort required |
| Non-dependent Gradual Types | `filter` returns `?` unknown *type* | ✗ Can have non-list return |
| Gradual Dependent Types | | |

## How To Solve?

| | | |
|---|---|---|
| Static Dependent Types | Existential Types, Inductive Proof | ✗ Significant effort required |
| Non-dependent Gradual Types | `filter` returns **`?`** unknown *type* | ✗ Can have non-list return |
| Gradual Dependent Types | `filter` returns **`Vec Int ?`** unknown *length* | |

# How To Solve?

| | | |
|---|---|---|
| Static Dependent Types | Existential Types, Inductive Proof | ✗ Significant effort required |
| Non-dependent Gradual Types | `filter` returns `?`<br>unknown *type* | ✗ Can have non-list return |
| Gradual Dependent Types | `filter` returns **`Vec Int ?`**<br>unknown *length* | ✓ Precise in type, flexible in length! |

## How To Solve?

| | | |
|---|---|---|
| Static Dependent Types | Existential Types, Inductive Proof | ✗ Significant effort required |
| Non-dependent Gradual Types | `filter` returns `?` unknown *type* | ✗ Can have non-list return |
| Gradual Dependent Types | `filter` returns `Vec Int ?` unknown *length* | ✓ Precise in type, flexible in length! |

Our approach!

# The GDTL Solution

```
sort : Vec Int n -> Vec Int n

sort Nil = Nil

sort (Cons head tail) =
  sort (filter (<= head) tail))
  ++ [head]
  ++ sort (filter (> head) tail))


filter : (Int -> Bool)
       -> Vec Int n ->
```

# The GDTL Solution

```
sort : Vec Int n -> Vec Int n

sort Nil = Nil

sort (Cons head tail) =
  sort (filter (<= head) tail))
  ++ [head]
  ++ sort (filter (> head) tail))


filter : (Int -> Bool)
       -> Vec Int n -> Vec Int ?
```

# The GDTL Solution

```
sort : Vec Int n -> Vec Int n

sort Nil = Nil

sort (Cons head tail) =
  sort (filter (<= head) tail))
  ++ [head]
  ++ sort (filter (> head) tail))
```

**?**+1+**?** evals to **?**, is *consistent* with n

```
filter : (Int -> Bool)
       -> Vec Int n -> Vec Int ?
```

# Gradual Type Safety

# Gradual Type Safety

# Gradual Type Safety

```
head : Vec a (n+1) -> a
```

```
head : Vec a (n+1) -> a

x : Vec a 0
```

# Gradual Type Safety

```
head : Vec a (n+1) -> a

x : Vec a 0

theHead = head x
```

```
head : Vec a (n+1) -> a

x : Vec a 0

theHead = head x
```

- Does not typecheck

# Gradual Type Safety

```
head : Vec a (n+1) -> a

x : Vec a 0

theHead = head x
```

```
head : Vec a (n+1) -> a

x : Vec a ?

theHead = head x
```

```
head : Vec a (n+1) -> a

x : Vec a ?

theHead = head x
```

- Typechecks!

```
head : Vec a (n+1) -> a

x : Vec a ?

theHead = head x
```

```
x ↦ Nil
```

```
head : Vec a (n+1) -> a

x : Vec a ?

theHead = head x
```

x ↦ Nil

- Runtime error

```
head : Vec a (n+1) -> a

x : Vec a ?

theHead = head x
```

x ↦ Nil

x ↦ Cons 1 Nil

# Gradual Type Safety

```
head : Vec a (n+1) -> a

x : Vec a ?

theHead = head x
```

x ↦ 

- Runs successfully

x ↦ Cons 1 Nil

```
sort : Vec Int n -> Vec Int n

sort Nil = Nil

sort (Cons head tail) =

  sort (filter (<= head) tail))
  ++ [head]
  ++ sort (filter (> head) tail))
```

Need proof that lengths sum to n

```
filter : (Int -> Bool)
       -> Vec Int n ->  Vec Int ?
```

# Filling in the Proof

```
sort : Vec Int n -> Vec Int n

sort Nil = Nil

sort (Cons head tail) =

  sort (filter (<= head) tail))
  ++ [head]
  ++ sort (filter (> head) tail))
```

Need proof that lengths sum to n

```
filter : (Int -> Bool)
       -> Vec Int n ->   Vec Int ?
```

# Gradual Proof Terms

```
rewriteFilterLength :
  (v : Vec Int n)
  -> (p : Int -> Bool)
  -> Vec Int
       (length (filter p v)
        + 1 + length (filter (not . p) v)
  ->  Vec Int n
```

```
rewriteFilterLength :
  (v : Vec Int n)
  -> (p : Int -> Bool)
  -> Vec Int
        (length (filter p v)
         + 1 + length (filter (not . p) v)
  ->  Vec Int n



rewriteFilterLength = ?
```

```
rewriteFilterLength :
  (v : Vec Int n)
  -> (p : Int -> Bool)
  -> Vec Int
        (length (filter p v)
         + 1 + length (filter (not . p) v)
  ->  Vec Int n
```

```
rewriteFilterLength = ?
```

Like Idris/Agda typed holes

```
sort : Vec Int n -> Vec Int n

sort Nil = Nil

sort (Cons head tail) =

  sort (filter (<= head) tail))
  ++ [head]
  ++ sort (filter (> head) tail))


filter : (Int -> Bool)
       -> Vec Int n ->   Vec Int ?
```

```
sort : Vec Int n -> Vec Int n

sort Nil = Nil

sort (Cons head tail) =
  rewriteFilterLength (
  sort (filter (<= head) tail))
  ++ [head]
  ++ sort (filter (> head) tail))
                                )

filter : (Int -> Bool)
       -> Vec Int n ->  Vec Int ?
```

```
sort : Vec Int n -> Vec Int n

sort Nil = Nil

sort (Cons head tail) =
  rewriteFilterLength (
  sort (filter (<= head) tail))
  ++ [head]
  ++ sort (filter (> head) tail))
                                )

filter : (Int -> Bool)
      -> Vec Int n -> Vec Int ?
```

This code typechecks *and runs!*

# Semantics of **?** in GDTL

# Semantics of **?** in GDTL

- **?** has type **?**, can use at any type

# Semantics of **?** in GDTL

- **?** has type **?**, can use at any type
- Eliminating **?** produces **?**

## Semantics of **?** in GDTL

- **?** has type **?**, can use at any type
- Eliminating **?** produces **?**
- Runtime checks ensure safety

## Semantics of **?** in GDTL

- **?** has type **?**, can use at any type
- Eliminating **?** produces **?**
- Runtime checks ensure safety

```
subst : a = b -> P a -> P b
```

- **?** has type **?**, can use at any type
- Eliminating **?** produces **?**
- Runtime checks ensure safety

```
subst : a = b -> P a -> P b

badProof : 0 = 1
badProof = ?
```

- **?** has type **?**, can use at any type
- Eliminating **?** produces **?**
- Runtime checks ensure safety

```
subst : a = b -> P a -> P b

badProof : 0 = 1
badProof = ?

head ((subst badProof nil) :: Vec Int 1)
```

# Semantics of **?** in GDTL

- **?** has type **?**, can use at any type
- Eliminating **?** produces **?**
- Runtime checks ensure safety

```
subst : a = b -> P a -> P b

badProof : 0 = 1
badProof = ?

head ((subst badProof nil) :: Vec Int 1)
```

- Typechecks
- Runtime type error

# GDTL: A Gradual Dependently Typed Language

Statics + Dynamics mostly using
*Abstracting Gradual Typing (Garcia et. al. 2016)*

Statics + Dynamics mostly using
*Abstracting Gradual Typing (Garcia et. al. 2016)*

**Main extensions:**

Statics + Dynamics mostly using
*Abstracting Gradual Typing (Garcia et. al. 2016)*

**Main extensions:**

Type/Term Overlap

Statics + Dynamics mostly using *Abstracting Gradual Typing (Garcia et. al. 2016)*

**Main extensions:**

Type/Term Overlap $\longrightarrow$ **?** as unknown type *and* term

Statics + Dynamics mostly using
*Abstracting Gradual Typing (Garcia et. al. 2016)*

**Main extensions:**

Type/Term Overlap $\longrightarrow$ **?** as unknown type *and* term

Type Indices

Statics + Dynamics mostly using
*Abstracting Gradual Typing (Garcia et. al. 2016)*

**Main extensions:**

Type/Term Overlap ⟶ **?** as unknown type *and* term

Type Indices ⟶ **?** as type index

Statics + Dynamics mostly using
*Abstracting Gradual Typing (Garcia et. al. 2016)*

**Main extensions:**

Type/Term Overlap → **?** as unknown type *and* term

Type Indices → **?** as type index

Proof term

Statics + Dynamics mostly using
*Abstracting Gradual Typing (Garcia et. al. 2016)*

**Main extensions:**

Type/Term Overlap ⟶ **?** as unknown type *and* term

Type Indices ⟶ **?** as type index

Proof term ⟶ **?** as a term *at runtime*

# What’s the Catch?

# Dependent Types

# Dependent Types

Evaluate terms
*at compile time*

# Dependent Types

Evaluate terms
*at compile time*

Strongly
normalizing

# Dependent Types

Evaluate terms
*at compile time*

Strongly
normalizing

Failure free

| Dependent Types | Gradual Types |
|---|---|
| Evaluate terms *at compile time* | |
| Strongly normalizing | |
| Failure free | |

| **Dependent Types** | **Gradual Types** |
|---|---|
| Evaluate terms *at compile time* | Evaluating has effects |
| Strongly normalizing | |
| Failure free | |

## Dependent Types

Evaluate terms
*at compile time*

Strongly
normalizing

Failure free

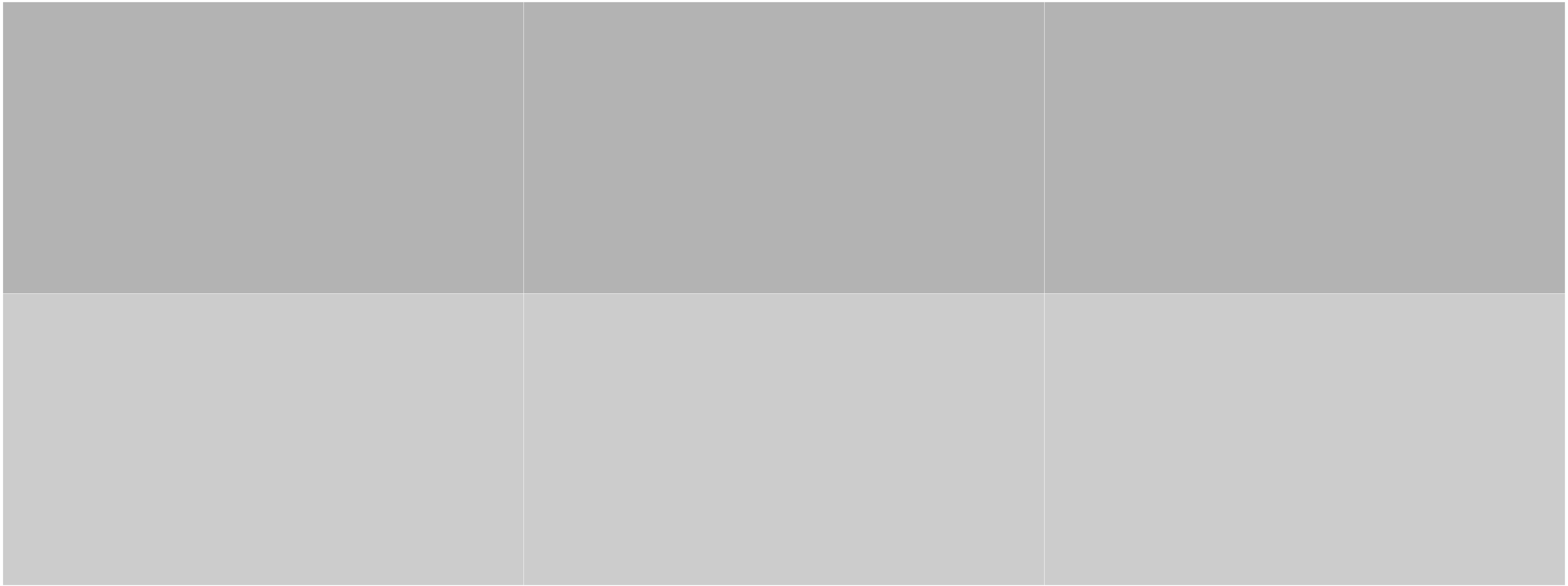
## Gradual Types

Evaluating has
effects

Can diverge
i.e. $\lambda(x : \mathbf{?}).\, x\ x$

| **Dependent Types** | **Gradual Types** |
|---|---|
| Evaluate terms *at compile time* | Evaluating has effects |
| Strongly normalizing | Can diverge i.e. $\lambda(x : \mathbf{?}).\, x\ x$ |
| Failure free | Type errors in evaluation |

# **Key Idea:** Approximate Normalization



## **Key Idea:** Approximate Normalization

Exploit the phase distinction:

# **Key Idea:** Approximate Normalization

Exploit the phase distinction:

| **Compile-time Normalization** | | |
|---|---|---|
| | | |

Exploit the phase distinction:

| | | |
|---|---|---|
| **Compile-time Normalization** | Always terminates | |
| | | |

# **Key Idea:** Approximate Normalization

Exploit the phase distinction:

| | | |
|---|---|---|
| **Compile-time Normalization** | Always terminates | Approximate results |
| | | |

# **Key Idea:** Approximate Normalization

Exploit the phase distinction:

| | | |
|---|---|---|
| **Compile-time Normalization** | Always terminates | Approximate results |
| **Runtime Evaluation** | | |

## **Key Idea:** Approximate Normalization

Exploit the phase distinction:

| | | |
|---|---|---|
| **Compile-time Normalization** | Always terminates | Approximate results |
| **Runtime Evaluation** | May diverge | |

## **Key Idea:** Approximate Normalization

Exploit the phase distinction:

| | | |
|---|---|---|
| **Compile-time Normalization** | Always terminates | Approximate results |
| **Runtime Evaluation** | May diverge | Exact results |

# Compile-Time - Approximation #1: Termination

**Based on *Hereditary Substitution***
(Watkins et al 2003)

**Based on *Hereditary Substitution***
(Watkins et al 2003)

- Static version: normalization is *structurally recursive* on types

**Based on *Hereditary Substitution***
(Watkins et al 2003)

- Static version: normalization is *structurally recursive* on types
- Our version:

**Based on *Hereditary Substitution***
(Watkins et al 2003)

- Static version: normalization is *structurally recursive* on types
- Our version:

Types structurally decreasing?

## **Compile-Time -** Approximation #1: Termination

**Based on *Hereditary Substitution***
(Watkins et al 2003)

- Static version: normalization is *structurally recursive* on types
- Our version:

Types structurally decreasing?

***Yes***

**Based on *Hereditary Substitution***
(Watkins et al 2003)

- Static version: normalization is *structurally recursive* on types
- Our version:

Types structurally decreasing?

***Yes***

Proceed normally

## Based on *Hereditary Substitution*

(Watkins et al 2003)

- Static version: normalization is *structurally recursive* on types
- Our version:

Types structurally decreasing?

***Yes***

***No***

Proceed normally

## Based on *Hereditary Substitution*

(Watkins et al 2003)

- Static version: normalization is *structurally recursive* on types
- Our version:

Types structurally decreasing?

***Yes*** → Proceed normally

***No*** → Return **?**

# Hereditary Substitution Examples

# Hereditary Substitution Examples

$$(\lambda x.\, x(\lambda y.\, yy))(\lambda z.\, zz)$$

# Hereditary Substitution Examples

$$(\lambda x.\, x(\lambda y.\, yy))(\lambda z.\, zz)$$

$$\rightsquigarrow$$

$$(\lambda z.\, zz)(\lambda y.\, yy)$$

# Hereditary Substitution Examples

$$(\lambda x.\, x(\lambda y.\, yy))(\lambda z.\, zz)$$

$$\rightsquigarrow$$

$$(\lambda z.\, zz)(\lambda y.\, yy)$$

$$\rightsquigarrow$$

$$(\lambda y.\, yy)(\lambda y.\, yy)$$

# Hereditary Substitution Examples

$$(\lambda x.\, x(\lambda y.\, yy))(\lambda z.\, zz)$$

$$\rightsquigarrow$$

$$(\lambda z.\, zz)(\lambda y.\, yy)$$

$$\rightsquigarrow$$

$$(\lambda y.\, yy)(\lambda y.\, yy)$$

$$\rightsquigarrow$$

$$(\lambda y.\, yy)(\lambda y.\, yy)$$

# Hereditary Substitution Examples

$$(\lambda x.\, x(\lambda y.\, yy))(\lambda z.\, zz) \qquad x : (\mathbf{?} \to \mathbf{?}) \to \mathbf{?}$$

$$\rightsquigarrow$$

$$(\lambda z.\, zz)(\lambda y.\, yy)$$

$$\rightsquigarrow$$

$$(\lambda y.\, yy)(\lambda y.\, yy)$$

$$\rightsquigarrow$$

$$(\lambda y.\, yy)(\lambda y.\, yy)$$

# Hereditary Substitution Examples

$$(\lambda x.\, x(\lambda y.\, yy))(\lambda z.\, zz)$$

$$\leadsto$$

$$(\lambda z.\, zz)(\lambda y.\, yy)$$

$$\leadsto$$

$$(\lambda y.\, yy)(\lambda y.\, yy)$$

$$\leadsto$$

$$(\lambda y.\, yy)(\lambda y.\, yy)$$

$$x : (\mathbf{?} \to \mathbf{?}) \to \mathbf{?}$$

$$\curlyvee$$

$$z : \mathbf{?} \to \mathbf{?}$$

# Hereditary Substitution Examples

$$(\lambda x.\, x(\lambda y.\, yy))(\lambda z.\, zz)$$

$$\rotatebox{-90}{$\rightsquigarrow$}$$

$$(\lambda z.\, zz)(\lambda y.\, yy)$$

$$\rotatebox{-90}{$\rightsquigarrow$}$$

$$(\lambda y.\, yy)(\lambda y.\, yy)$$

$$\rotatebox{-90}{$\rightsquigarrow$}$$

$$(\lambda y.\, yy)(\lambda y.\, yy)$$

$$x : (\mathbf{?} \to \mathbf{?}) \to \mathbf{?}$$

$$\curlyvee$$

$$z : \mathbf{?} \to \mathbf{?}$$

$$\curlyvee$$

$$y : \mathbf{?}$$

# Hereditary Substitution Examples

$$
\begin{array}{ll}
(\lambda x.\, x(\lambda y.\, yy))(\lambda z.\, zz) & x : (\mathbf{?} \to \mathbf{?}) \to \mathbf{?} \\
\rotatebox[origin=c]{-90}{$\rightsquigarrow$} & \curlyvee \\
(\lambda z.\, zz)(\lambda y.\, yy) & z : \mathbf{?} \to \mathbf{?} \\
\rotatebox[origin=c]{-90}{$\rightsquigarrow$} & \curlyvee \\
(\lambda y.\, yy)(\lambda y.\, yy) & y : \mathbf{?} \\
\rotatebox[origin=c]{-90}{$\rightsquigarrow$} & \not\curlyvee \\
(\lambda y.\, yy)(\lambda y.\, yy) & y : \mathbf{?}
\end{array}
$$

$$
\begin{array}{ll}
(\lambda x.\, x(\lambda y.\, yy))(\lambda z.\, zz) & x : (\mathbf{?} \to \mathbf{?}) \to \mathbf{?} \\
\rightsquigarrow & \curlyvee \\
(\lambda z.\, zz)(\lambda y.\, yy) & z : \mathbf{?} \to \mathbf{?} \\
\rightsquigarrow & \curlyvee \\
(\lambda y.\, yy)(\lambda y.\, yy) & y : \mathbf{?} \\
\rightsquigarrow & \not\curlyvee \\
(\lambda y.\, yy)(\lambda y.\, yy) & y : \mathbf{?} \\
\rightsquigarrow & \\
\mathbf{?} &
\end{array}
$$

# Compile-Time - Approximation #2: Downcasts

Order types by *precision*

Order types by *precision*

Losing type information : ✓ **Safe**

Order types by *precision*

Losing type information : ✓ **Safe**

Gaining type information: ✗ **Unsafe**

Order types by *precision*

Losing type information : ✓ **Safe**

Gaining type information: ✗ **Unsafe**

Unsafe operation: approximate!

Order types by *precision*

Losing type information : ✓ **Safe**

Gaining type information: ✗ **Unsafe**

Unsafe operation: approximate!

$$((3 :: \mathbb{N}) :: \mathbf{?}) :: \mathbb{B}$$

Order types by *precision*

Losing type information : ✓ **Safe**

Gaining type information: ✗ **Unsafe**

Unsafe operation: approximate!

$$((3 :: \mathbb{N}) :: \mathbf{?}) :: \mathbb{B} \leadsto ((3 :: \mathbf{?}) :: \mathbb{B})$$

Order types by *precision*

Losing type information : ✓ **Safe**

Gaining type information: ✗ **Unsafe**

Unsafe operation: approximate!

$$((3 :: \mathbb{N}) :: \mathbf{?}) :: \mathbb{B} \rightsquigarrow ((3 :: \mathbf{?}) :: \mathbb{B}) \rightsquigarrow \mathbf{?}$$

- Terms annotated with *evidence*

- Terms annotated with *evidence*
    - Most-precise currently-known type info

- Terms annotated with *evidence*
    - Most-precise currently-known type info
- Combined using precision-meet

## **Run-Time -** Execution

- Terms annotated with *evidence*
  - Most-precise currently-known type info
- Combined using precision-meet
  - Runtime error if meet does not exist

# Wrapping Up

# What We Built

# What We Built

**GDTL:**

# GDTL:

# Gradual Dependently Typed Language

**GDTL:**

**Gradual Dependently Typed Language**

- Full spectrum, universe hierarchy

**GDTL:**

**Gradual Dependently Typed Language**

- Full spectrum, universe hierarchy
- Can replace any type or term with **?**

**GDTL:**

**Gradual Dependently Typed Language**

- Full spectrum, universe hierarchy
- Can replace any type or term with **?**
- Embeds fully typed & untyped calculi

## GDTL:

## Gradual Dependently Typed Language

- Full spectrum, universe hierarchy
- Can replace any type or term with **?**
- Embeds fully typed & untyped calculi
- Decidable typechecking via ***approximate normalization***

## GDTL:

## Gradual Dependently Typed Language

- Full spectrum, universe hierarchy
- Can replace any type or term with **?**
- Embeds fully typed & untyped calculi
- Decidable typechecking via ***approximate normalization***
- Proof of gradual type safety

# GDTL:

# Gradual Dependently Typed Language

- Full spectrum, universe hierarchy
- Can replace any type or term with **?**
- Embeds fully typed & untyped calculi
- Decidable typechecking via ***approximate normalization***
- Proof of gradual type safety
- Gradual Guarantees

# Future Work

# Future Work

- Inductives and Pattern Matching

## Future Work

- Inductives and Pattern Matching
- Type Inference and Unification

# Future Work

- Inductives and Pattern Matching
- Type Inference and Unification
- Blame and Error Reporting

## Future Work

- Inductives and Pattern Matching
- Type Inference and Unification
- Blame and Error Reporting
- Eventual Goal: Idris frontend

## GDTL:
## Gradual Dependently Typed Language

- Full spectrum, universe hierarchy
- Can replace any type or term with **?**
- Embeds fully typed & untyped calculi
- Decidable typechecking via ***approximate normalization***
- Proof of gradual type safety
- Gradual Guarantees