Towards Modular Composition of Inductive Types Using Lean Meta-programming

Ramy Shahin

Qualgebra

TYPES 2025 – Glasgow, Scotland – June 12th 2025

Motivating Example

```
inductive T where
| Bool
| N
| Fn (τ₁ τ₂: T)

inductive Term where
| True
| False
| If (c t₁ t₂: Term)
| Zero
| Succ (t: Term)
| Pred (t: Term)
| V   (x: Var)
| Abs (x: Var) (τ: T) (b: Term)
| App (t₁ t₂: Term)

inductive Val: Term → Prop
| T: Val .True
| F: Val .False
| Z: Val .Zero
| S (v: Term): Val v → Val (.Succ v)
| A (x: Var) (τ: T) (b: Term):
    Val (.Abs x τ t)
```

```
def count: Term → Nat
| .True          => 1
| .False         => 1
| .If c t₁ t₂    => 1 + count c + count t₁+ count t₂
| .Zero          => 1
| .Succ t        => 1 + count t
| .Pred t        => 1 + count t
| .V _           => 2
| .Abs _ _ b     => 3 + count b
| .App t₁ t₂     => 1 + count t₁ + count t₂

inductive TRel: Term → T → Prop
| TT: TRel .True .Bool
| FF: TRel .False .Bool
| If: TRel c .Bool → TRel t₁ τ → TRel t₂ τ →
    TRel (.If c t₁ t₂) τ
| Z: TRel .Zero .N
| S: TRel t .N → TRel (.Succ t) .N
| P: TRel t .N → TRel (.Pred t) .N
| V (x: Var) (τ: T): Γ x = τ → TRel Γ (.V x) τ
| Abs (x: Var) (b: Term) (τ₁ τ₂: T):
    TRel (augment Γ x τ₁) b τ₂ →
    TRel Γ (.Abs x τ₁ b) (.Fn τ₁ τ₂)
| App (t₁ t₂: Term) (τ₁ τ₂: T):
    TRel Γ t₁ (.Fn τ₁ τ₂) → TRel Γ t₂ τ₁ →
    TRel Γ (.App t₁ t₂) τ₂
```

Motivating Example

```
inductive T where
| Bool
| N
| Fn (τ₁ τ₂: T)
```

```
inductive Term where
| True
| False
| If (c t₁ t₂: Term)
| Zero
| Succ (t: Term)
| Pred (t: Term)
| V   (x: Var)
| Abs (x: Var) (τ: T) (b: Term)
| App (t₁ t₂: Term)
```

```
inductive Val: Term → Prop
| T: Val .True
| F: Val .False
| Z: Val .Zero
| S (v: Term): Val v → Val (.Succ v)
| A (x: Var) (τ: T) (b: Term):
    Val (.Abs x τ t)
```

```
def count: Term → Nat
| .True          => 1
| .False         => 1
| .If c t₁ t₂    => 1 + count c + count t₁+ count t₂
| .Zero          => 1
| .Succ t        => 1 + count t
| .Pred t        => 1 + count t
| .V             => 2
| .Abs       b   => 3 + count b
| .App t₁ t₂     => 1 + count t₁ + count t₂
```

```
inductive TRel: Term → T → Prop
| TT: TRel .True .Bool
| FF: TRel .False .Bool
| If: TRel c .Bool → TRel t₁ τ → TRel t₂ τ →
    TRel (.If c t₁ t₂) τ
| Z: TRel .Zero .N
| S: TRel t .N → TRel (.Succ t) .N
| P: TRel t .N → TRel (.Pred t) .N
| V (x: Var) (τ: T): Γ x = τ → TRel Γ (.V x) τ
| Abs (x: Var) (b: Term) (τ₁ τ₂: T):
    TRel (augment Γ x τ₁) b τ₂ →
    TRel Γ (.Abs x τ₁ b) (.Fn τ₁ τ₂)
| App (t₁ t₂: Term) (τ₁ τ₂: T):
    TRel Γ t₁ (.Fn τ₁ τ₂) → TRel Γ t₂ τ₁ →
    TRel Γ (.App t₁ t₂) τ₂
```

Motivating Example

```
inductive T where
| Bool
| N
| Fn (τ₁ τ₂: T)
```

```
inductive Term where
| True
| False
| If (c t₁ t₂: Term)
| Zero
| Succ (t: Term)
| Pred (t: Term)
| V   (x: Var)
| Abs (x: Var) (τ: T) (b: Te
| App (t₁ t₂: Term)
```

```
inductive Val: Term → Prop
| T: Val .True
| F: Val .False
| Z: Val .Zero
| S (v: Term): Val v → Val (.Succ v)
| A (x: Var) (τ: T) (b: Term):
    Val (.Abs x τ t)
```

```
def count: Term → Nat
 .True          => 1
 .False         => 1
 .If c t₁ t₂    => 1 + count c + count t₁+ count t₂
 .Zero          => 1
 .Succ t        => 1 + count t
                            nt t
                           t b
                           t t₁ + count t₂
```

$$\texttt{scattering} \propto \frac{1}{\texttt{modularity} \land \texttt{reusability}}$$

```
                          → Prop
                           t₁ τ → TRel t₂ τ →
 Z: TRel .Zero .N
 S: TRel t .N → TRel (.Succ t) .N
 P: TRel t .N → TRel (.Pred t) .N
| V (x: Var) (τ: T): Γ x = τ → TRel Γ (.V x) τ
| Abs (x: Var) (b: Term) (τ₁ τ₂: T):
   TRel (augment Γ x τ₁) b τ₂ →
   TRel Γ (.Abs x τ₁ b) (.Fn τ₁ τ₂)
| App (t₁ t₂: Term) (τ₁ τ₂: T):
   TRel Γ t₁ (.Fn τ₁ τ₂) → TRel Γ t₂ τ₁ →
   TRel Γ (.App t₁ t₂) τ₂
```

Boolean Module

```
namespace Boolean

inductive T where
| Bool

inductive Term where
| True
| False
| If (c t₁ t₂: Term)

def countNodes: Term → Nat
| .True => 1
| .False => 1
| .If c t₁ t₂ => 1 + countNodes c + countNodes t₁ + countNodes t₂

inductive Val: Term → Prop
| T: Val .True
| F: Val .False

inductive TRel: Term → T → Prop
| TT: TRel .True .Bool
| FF: TRel .False .Bool
| If: TRel c .Bool → TRel t₁ τ → TRel t₂ τ → TRel (.If c t₁ t₂) τ

end Boolean
```

Nat Module

```
namespace Nat

inductive T where
| N

inductive Term where
| Zero
| Succ (t: Term)
| Pred (t: Term)

def countNodes: Term → Nat
| .Zero => 1
| .Succ t => 1 + countNodes t
| .Pred t => 1 + countNodes t

inductive Val: Term → Prop
| Z: Val .Zero
| S (v: Term): Val v → Val (.Succ v)

inductive TRel: Term → T → Prop where
| Z: TRel .Zero .N
| S: TRel t .N → TRel (.Succ t) .N
| P: TRel t .N → TRel (.Pred t) .N

end Nat
```

STLC Module

```
namespace STLC

inductive T: Type
| Fn (τ₁ τ₂: T)

abbrev Var := String

abbrev Context := Var → T
def augment (Γ: Context) (x: Var) (τ: T): Context := λv ↦ if v=x then τ else Γ v

inductive Term where
| V   (x: Var)
| Abs (x: Var) (τ: T) (b: Term)
| App (t₁ t₂: Term)

def countNodes: Term → Nat
| .V _        => 2
| .Abs _ _ b  => 3 + countNodes b
| .App t₁ t₂  => 1 + countNodes t₁ + countNodes t₂

inductive Val: Term → Prop
| A (x: Var) (τ: T) (b: Term): Val (.Abs x τ t)

inductive TRel: Context → Term → T → Prop where
| V (x: Var) (τ: T): Γ x = τ → TRel Γ (.V x) τ
| Abs (x: Var) (b: Term) (τ₁ τ₂: T):
    TRel (augment Γ x τ₁) b τ₂ → TRel Γ (.Abs x τ₁ b) (.Fn τ₁ τ₂)
| App (t₁ t₂: Term) (τ₁ τ₂: T):
    TRel Γ t₁ (.Fn τ₁ τ₂) → TRel Γ t₂ τ₁ → TRel Γ (.App t₁ t₂) τ₂

end STLC
```

Inductive Type Composition

```
Namespace Boolean
inductive T where
| Bool

…
end Boolean
```

```
namespace Nat
inductive T where
| N

…
end Nat
```

```
namespace STLC
inductive T: Type
| Fn (τ₁ τ₂: T)

…
end STLC
```

```
inductive T := Boolean.T |+ Nat.T |+ STLC.T
```

inductive T where
| Bool
| N
~~| Fn (τ₁ τ₂: STLC.T)~~
| Fn (τ₁ τ₂: T)

Composition and Extension

```
namespace Boolean

…
inductive Term where
| True
| False
| If (c t₁ t₂: Term)

…
end Boolean
```

```
namespace Nat

…

inductive Term where
| Zero
| Succ (t: Term)
| Pred (t: Term)

…
end Nat
```

```
namespace STLC

inductive Term where
| V   (x: Var)
| Abs (x: Var) (τ: T) (b: Term)
| App (t₁ t₂: Term)

…
end STLC
```

```
inductive Term := Boolean.Term |+ Nat.Term |+ STLC.Term
| isZero (t: Term)
```

crosscuts Boolean and Nat

```
inductive Term where
| True
| False
| If (c t₁ t₂: Term)
| Zero
| Succ (t: Term)
| Pred (t: Term)
| V   (x: Var)
| Abs (x: Var) (τ: T) (b: Term)
| App (t₁ t₂: Term)
| isZero (t: Term)
```

Dependencies

```
Boolean.TRel: Boolean.Term → Boolean.T → Prop
Nat.TRel: Nat.Term → Nat.T → Prop
STLC.TRel: STLC.Term → STLC.T → Prop
```

```
inductive TRel: Context → Term → T → Prop := Boolean.TRel |+ Nat.TRel |+ STLC.TRel
| iz: TRel Γ t T.N → TRel Γ (.isZero t) T.Bool
```

```
T = Boolean.T |+ Nat.T + STLC.T
Term = Boolean.Term |+ Nat.Term |+ STLC.Term
| isZero …
```

Subtyping and Coercion

```
inductive T := Boolean.T |+ Nat.T |+ STLC.T
```

Boolean.T <: T
Nat.T <: T
STLC.T <: T

```
instance: Coe Boolean.T T where
  coe := λ x ↦ match x with
               | Boolean.T.Bool => T.Bool

instance: Coe Nat.T T where
  coe := λ x ↦ match x with
               | Nat.T.N => T.N

instance: Coe STLC.T T where
  coe := λ x ↦ match x with
               | STLC.T.Fn τ₁ τ₂ => T.Fn τ₁ τ₂
```

recursive coercion

```
def τ: T := Boolean.T.Bool
```

✓

Subtyping and Dependent Coercion

```
inductive T := Boolean.T |+ Nat.T |+ STLC.T
```

Boolean.T <: T
Nat.T <: T
STLC.T <: T

```
instance : CoeDep T (T.Bool) Boolean.T where coe := Boolean.T.Bool
instance : CoeDep T (T.N) Nat.T where coe := Nat.T.N
instance (a : STLC.T) (b : STLC.T) : CoeDep T (T.Fn a b) STLC.T where coe := STLC.T.Fn a b
```

```
def t: Boolean.T := T.Bool
```

✓

```
def b := T.Bool
def s: Boolean.T := b
```

✗

Multiplexing Functions

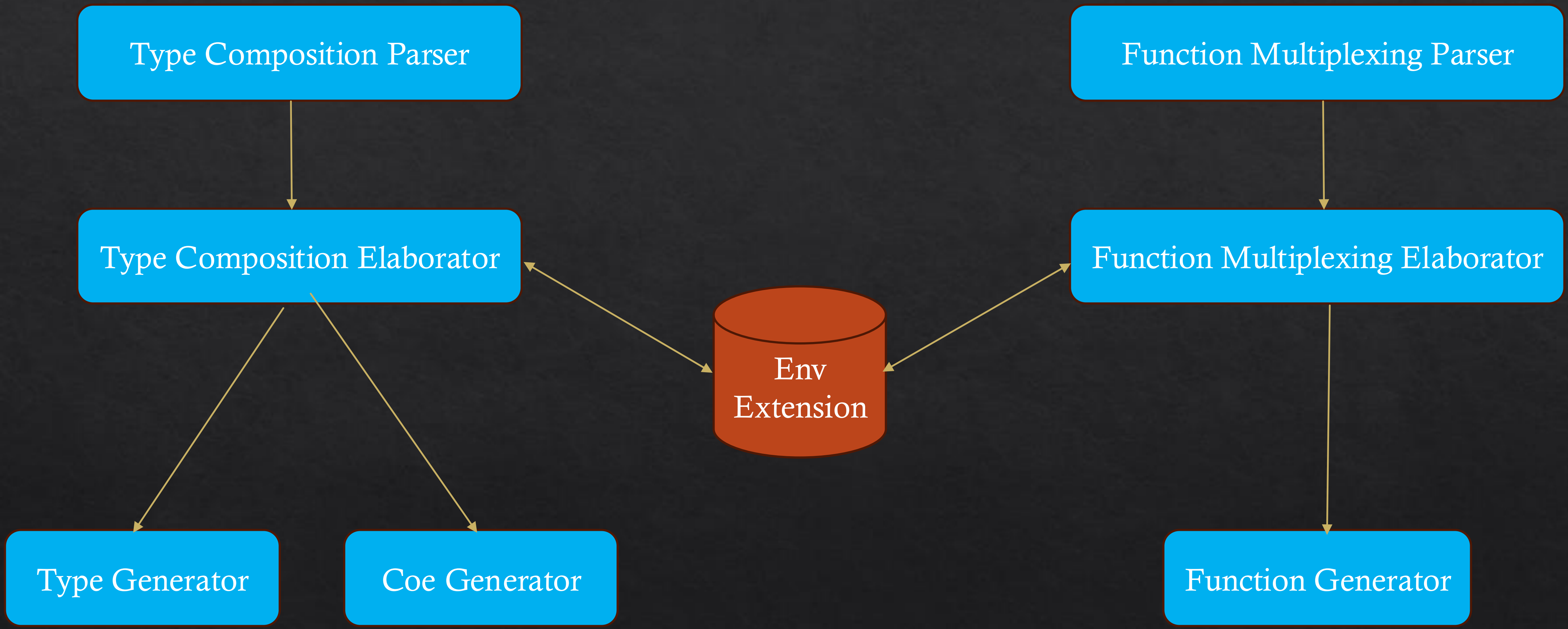
```
fn countNodes := Boolean.countNodes |+ Nat.countNodes |+ STLC.countNodes
| isZero t => 1 + countNodes t
```

Assumption: same pattern-matching structure

Adjusting function calls (including recursive ones)

```
def countNodes: Term → Nat
  | .True => 1
  | .False => 1
  | .If c t₁ t₂ => 1 + countNodes c + countNodes t₁ + countNodes t₂
  | .Zero => 1
  | .Succ t => 1 + countNodes t
  | .Pred t => 1 + countNodes t
  | .V _        => 2
  | .Abs _ _ b => 3 + countNodes b
  | .App t₁ t₂ => 1 + countNodes t₁ + countNodes t₂
  | .isZero t => 1 + countNodes t
```

Architecture



Limitations

- Full support of higher-order types, indexed-types, dependent types
- Assumptions on multiplexed functions
- Mutual recursion
- Composing feature modules instead of individual types/functions
- (Partial?) composition of theorems and proof objects
- Function reuse instead of rewriting
    - Recursion? Modifying fixpoint operators?
    - Cost of function calls? Inlining?

Thank You

Questions

https://github.com/qualgebra/LeanToolkit/tree/TYPES2025