# From Datatype Genericity to Language Genericity

**Liang-Ting Chen**

Institute of Information Science, Academia Sinica, Taiwan

**WG6 Meeting of EuroProofNet · Leuven, Belgium · 05 April 2024**

# Motivation

- Language-formalisation frameworks are in favour of *intrinsic typing* (Allais et al, 2021; Fiore & Szamozvancev, 2022).
- Wait, where are intrinsically-typed terms from?
  - Syntax parsing, type checking and synthesis, etc.
- How to *generically* prove the correctness of
  1. the conversion between intrinsic typing and extrinsic typing, and
  2. (bidirectional) type synthesis?

# Datatype-Generic Programming

# Datatype-Generic Programming, Classically

## 'Scrap your boilerplate code'

- Datatype-generic programs (DGP) are programs parametrised in ***descriptions*** of data types, e.g., in Haskell
  - `show :: (Show a) => a -> String`, `== :: (Eq a) => a -> a -> Bool`
  - JSON conversion by the Aeson package
- ***Algorithms*** and ***theorems*** can be generically designed for various data types (Bird & de Moor, 1997).
- What counts as data types?
  - In Haskell, `GHC.Generics` defines *finite sums of finite products*.
  - Some valid Haskell data types, e.g., GADT, are not included.
  - *Dependent types* allow more interesting data types.

# Theories of Data Types

- **Syntax**
  - W-types (Martin-Löf, 1982)
  - ***Inductive families*** (Dybjer, 1994)
  - (Indexed) inductive-recursive types (IRT) (Dybjer & Setzer, 2003; Dybjer & Setzer, 2006)
  - Inductive-inductive types (IIT) by Forsberg & Setzer (2010), IIR (Forsberg, 2014)
  - Higher IIT (Kaposi & Kovács, 2018, 2020)
  - ***Quotient inductive-inductive types (QIIT)*** by Altenkirch et al. (2018) and Kaposi et al. (2019)
- **Categorical semantics**
  - ***Polynomial functors*** (Fiore, 2012; Gambino & Kock, 2013; Awodey et al. 2017)
  - Cell monads with parameters (Lumsdaine & Shulman, 2019)
- **Ongoing studies**
  - QIIR (Kaposi, 2023)
  - ***HIIR*** (Kovács, 2023), which is semantically unclear but allowed in Agda

# Theories of Data Types

- Inductive families suffice to define non-dependent type systems, incl. simply typed $\lambda$-calculus, polymorphic $\lambda$-calculus, etc.
- Quotient inductive-inductive types (QIIT) suffice to define dependent type theories modulo equality rules.
- ***Polynomial functors*** serve as a foundation of inductive families (Dagand & McBride, 2013), small IR (Hancock et al., 2013), GADT (Hamana & Fiore, 2011), etc.
    - Unfortunately DGP techniques are currently based on polynomial functors.

# Datatype-Generic Programming, Dependently

- Most dependently typed languages does *not* have a mechanism for DGP.
- Instead, it is known that the we only need one data type $\mu$ for descriptions.
  - Is that a problem?

# Inductive Families in Agda

- 'IFam I I' encodes I-indexed ***inductive families***.
- Each 'D : IFam I I' is called a ***description***.
- Every description $D$ defines a functor $[\![D]\!]$ from $\mathscr{U}^I$ to $\mathscr{U}^J$.
- (μ D, con) is the initial $[\![D]\!]$-algebra.

```
data IFam (I J : 𝑈) : 𝑈₁ where
  End : (j : J)                            → IFam I J
  Arg : (S : 𝑈) → (S       → IFam I J) → IFam I J
  Ind : (P : 𝑈) → (P → I) → IFam I J  → IFam I J
```

```
⟦_⟧ : IFam I J → (I → 𝑈) → (J → 𝑈)
⟦ End j'    ⟧ X j = j' ≡ j
⟦ Arg S T   ⟧ X j = Σ[ s ∈ S ] ⟦ T s ⟧ X j
⟦ Ind P f D ⟧ X j =
  ((p : P) → X (f p)) × ⟦ D ⟧ X j
```

```
data μ (D : IFam I I) : I → 𝑈 where
  con : ⟦ D ⟧ (μ D) i → (μ D) i
```

# Inductive Families in Agda

## Example: vectors

```
data Vec (A : 𝑈) : ℕ → 𝑈 where
  []  : Vec A 0
  _∷_ : {n : ℕ} → A
    → Vec A n → Vec A (suc n)
```

```
data VecT : 𝑈 where nilT consT : VecT

VecD : 𝑈 → IFam ℕ ℕ
VecD A = σ VecT λ where
  nilT  → End 0
  consT → Arg ℕ λ n → Arg A λ x
    → Ind ⊤ (λ _ → n) (End (suc n))

Vec : 𝑈 → ℕ → 𝑈
Vec A = μ (VecD A)

[] : Vec A 0
[] = con (nilT , refl)

_∷_ : A → Vec A n → Vec A (suc n)
_∷_ {A} {n = n} x xs =
  con (consT , n , x , (λ _ → xs) , refl)
```

# Datatype-Generic Programming via $\mu$

- DGP based on $\mu : \mathsf{Desc} \to \mathscr{U}$ is not practical.
  - Tedious to write descriptions.
  - Generic programs of different universes cannot interoperate.
  - Support for data types is lost, e.g., pattern matching, constructor overloading, and case splitting.
- The syntactic information is lost, e.g. constructors do not play a role.
- Can we make DGP more useful in a dependently typed language?

# Datatype-Generic Programming, Natively

## (Ko, Chen & Lin, 2022)

- A DGP mechanism allows us to
    1. *reflect* a data type
    2. *reify* a description
- Descriptions preserve the syntactic structure.
- Generic definitions can be reified to *remove intermediate structures.*
    - Suitable for further reasoning.
- So, what can we do ***generically on data types***?

```
data List (A : 𝑈) : 𝑈 where
  …
ListC : Named (quote List) _
  unNamed ListC = genDataC ListD (genDataT ListD List)
    where ListD = genDataD List
…
unquoteDecl foldr-fusion = defineInd foldr-fusionP foldr-fusion
-- foldr-fusion :
--   {A B C : 𝑈} (h : B → C) {e : B} {f : A → B → B}
–   {e’ : C} {f' : A → C → C}
--   (he : h e ≡ e') (hf : (a : A) (b : B) (c : C)
–     → h b ≡ c → h (f a b) ≡ f' a c)
--   (as : List A) →h (foldr e f as) ≡ foldr e' f' as

-- foldr-fusion h he hf []       = he
-- foldr-fusion h he hf (a ∷ as) =
--   hf a _ _ (foldr-fusion h he hf as)
```

# Ornaments
## (Scrap your boilerplate data types)

# Ornaments

## Relationships between structurally similar datatype descriptions

- An ***ornament*** describes how an inductive family is ***enriched over*** a base data type.
  - Ornaments are itself inductively defined.
- The universe of ornaments boils down to the following cases (Dagand & McBride, 2014):
  1. Extension by a non-inductive argument
  2. Index refinement
  3. Deletion (if reconstructible from somewhere, e.g. indices)
  4. Preservation

2. Categorically, ornaments over a ***base*** description $D$ as a polynomial are precisely ***cartesian morphisms*** into $D$ (Dagand & McBride, 2013).

# Ornaments

## Example: extending and refining numbers to vectors

```
data ℕ : 𝒰 where
  zero : ℕ
  suc  : ℕ → ℕ
data List (A : 𝒰) : 𝒰 where
  []   : List A
  _∷_  : A → List A → List A
data Vec (A : 𝒰) : ℕ → 𝒰 where
  []   : Vec A 0
  _∷_  : A → {n : ℕ} → Vec A n → Vec A (suc n)
```

# Ornaments

## Step 1: stating the ornaments

```
data ℕ : 𝓤 where
  zero : ℕ
  suc  :             ℕ → ℕ
data List (A : 𝓤) : 𝓤 where
  []   : List A
  _∷_  : A → List A → List A
```

List-Orn A

Extension by an argument

# Ornaments

## Step 1: stating the ornaments

```
data ℕ : 𝓤 where
  zero : ℕ
  suc  :           ℕ → ℕ

data List (A : 𝓤) : 𝓤 where
  []   : List A
  _∷_  : A →          List A  → List A

data Vec (A : 𝓤) : ℕ → 𝓤 where
  []   : Vec A 0
  _∷_  : A → {n : ℕ} → Vec A n → Vec A (suc n)
```

Vec-Orn

Extension by an argument

Refining by an additional index

# Ornaments

## Step 2: derive the forgetful maps from the ornaments

```
data ℕ : 𝓤 where
  zero : ℕ
  suc  :                ℕ         → ℕ
data List (A : 𝓤) : 𝓤 where
  []   : List A
  _∷_  : A →            List A    → List A
data Vec (A : 𝓤) : ℕ → 𝓤 where
  []   : Vec A 0
  _∷_  : A → {n : ℕ} → Vec A n → Vec A (suc n)
```

forget

forget

Forgetting the additional argument

Forgetting the additional index

# Ornaments

## Scrap your boilerplate data types

- Structural recursion gives rise to an ornament, called ***algebraic ornamentation***, e.g.
  - $\mathit{length} : \mathsf{List}\ A \to \mathbb{N}$ gives `Vec-Orn`.
- What has been forgotten can be ***remember***ed, e.g. $\mathit{remember} : (\mathit{xs} : \mathsf{List}\ A) \to \mathsf{Vec}\ A\ (\mathit{length}\ \mathit{xs})$.
- Every ornament derives an ***isomorphism*** using the remember-forget pair.

$$\mathsf{List}\ A \cong \Sigma\ (n : \mathbb{N})\ (\mathsf{Vec}\ A\ n)$$

# Language-Generic Programming

# Extrinsic and Intrinsic Typing

# Extrinsic and Intrinsic Typing

## How do we prove the isomorphism without induction?

$$\Gamma \vdash \tau \cong \Sigma\,(t : \Lambda)\; \Gamma \vdash t : \tau$$

# Extrinsic and Intrinsic Typing for STLC

## Step 1: Specify the ornament

```
data Λ : 𝑈 where

  var : ℕ → Λ

  app : Λ
      → Λ
      → Λ

  lam : Λ
      → Λ
```

Typing-Orn →

```
data _⊢_ : List Ty → Ty → 𝑈 where

  var : Γ ∋ τ
      → Γ ⊢ τ

  app : Γ ⊢ τ ⇒ τ'
      → Γ ⊢ τ
      → Γ ⊢ τ'

  lam : τ ∷ Γ ⊢ τ'
      → Γ ⊢ τ ⇒ τ'
```

# Extrinsic and Intrinsic Typing for STLC

## Step 2: Derive the forget map from the ornament

```
data Λ : 𝑈 where

  var : ℕ → Λ

  app : Λ
      → Λ
      → Λ

  lam : Λ
      → Λ
```

← forget

```
data _⊢_ : List Ty → Ty → 𝑈 where

  var : Γ ∋ τ
      → Γ ⊢ τ

  app : Γ ⊢ τ ⇒ τ'
      → Γ ⊢ τ
      → Γ ⊢ τ'

  lam : τ ∷ Γ ⊢ τ'
      → Γ ⊢ τ ⇒ τ'
```

```
toΛ : Γ ⊢ τ → Λ
toΛ (var i  ) = var (toℕ i)
toΛ (app t u) = app (toΛ t) (toΛ u)
toΛ (lam t  ) = lam (toΛ t)
```

# Extrinsic and Intrinsic Typing for STLC

## Step 3: Use the forgetful map to derive the ornament

```
data _⊢_∶_ : List Ty → Λ → Ty → 𝓤 where

  var : (i : Γ ∋ τ)
      → Γ ⊢ var (toℕ i) ∶ τ

  app : ∀ {t} → Γ ⊢ t ∶ τ ⇒ τ'
      → ∀ {u} → Γ ⊢ u ∶ τ
      → Γ ⊢ app t u ∶ τ'

  lam : ∀ {t} → τ ∷ Γ ⊢ t ∶ τ'
      → Γ ⊢ lam t ∶ τ ⇒ τ'
```

*algebraic ornamentation via* toΛ

```
data _⊢_ : List Ty → Ty → 𝓤 where

  var : Γ ∋ τ
      → Γ ⊢ τ

  app : Γ ⊢ τ ⇒ τ'
      → Γ ⊢ τ
      → Γ ⊢ τ'

  lam : τ ∷ Γ ⊢ τ'
      → Γ ⊢ τ ⇒ τ'
```

```
toΛ : Γ ⊢ τ → Λ
toΛ (var i  ) = var (toℕ i)
toΛ (app t u) = app (toΛ t) (toΛ u)
toΛ (lam t  ) = lam (toΛ t)
```

# Extrinsic and Intrinsic Typing for STLC

## Step 4: Derive another forgetful map from the algebraic ornamentation

```
data _⊢_∶_ : List Ty → Λ → Ty → 𝓤 where

  var : (i : Γ ∋ τ)
      → Γ ⊢ var (toℕ i) ∶ τ

  app : ∀ {t} → Γ ⊢ t ∶ τ ⇒ τ'
      → ∀ {u} → Γ ⊢ u ∶ τ
      → Γ ⊢ app t u ∶ τ'

  lam : ∀ {t} → τ ∷ Γ ⊢ t ∶ τ'
      → Γ ⊢ lam t ∶ τ ⇒ τ'
```

forget

```
data _⊢_ : List Ty → Ty → 𝓤 where

  var : Γ ∋ τ
      → Γ ⊢ τ

  app : Γ ⊢ τ ⇒ τ'
      → Γ ⊢ τ
      → Γ ⊢ τ'

  lam : τ ∷ Γ ⊢ τ'
      → Γ ⊢ τ ⇒ τ'
```

```
fromTyping : ∀ {t} → Γ ⊢ t ∶ τ → Γ ⊢ τ
fromTyping (var i  ) = var i
fromTyping (app d e) = app (fromTyping d) (fromTyping e)
fromTyping (lam d  ) = lam (fromTyping d)
```

# Extrinsic and Intrinsic Typing for STLC

## Step 5: Remember what you forget

```
data _⊢_:_ : List Ty → Λ → Ty → 𝑼 where

  var : (i : Γ ∋ τ)
      → Γ ⊢ var (toℕ i) : τ

  app : ∀ {t} → Γ ⊢ t : τ ⇒ τ'
      → ∀ {u} → Γ ⊢ u : τ
      → Γ ⊢ app t u : τ'

  lam : ∀ {t} → τ ∷ Γ ⊢ t : τ'
      → Γ ⊢ lam t : τ ⇒ τ'
```

← remember

```
data _⊢_ : List Ty → Ty → 𝑼 where

  var : Γ ∋ τ
      → Γ ⊢ τ

  app : Γ ⊢ τ ⇒ τ'
      → Γ ⊢ τ
      → Γ ⊢ τ'

  lam : τ ∷ Γ ⊢ τ'
      → Γ ⊢ τ ⇒ τ'
```

```
toTyping : (t : Γ ⊢ τ) → Γ ⊢ toΛ t : τ
toTyping (var i  ) = var i
toTyping (app t u) = app (toTyping t) (toTyping u)
toTyping (lam t  ) = lam (toTyping t)
```

# Extrinsic and Intrinsic Typing for STLC

## Step 6: Apply the theorems about the remember-forget pair

```
data _⊢_∶_ : List Ty → Λ → Ty → 𝒰 where

  var : (i : Γ ∋ τ)
      → Γ ⊢ var (toℕ i) ∶ τ

  app : ∀ {t} → Γ ⊢ t ∶ τ ⇒ τ'
      → ∀ {u} → Γ ⊢ u ∶ τ
      → Γ ⊢ app t u ∶ τ'

  lam : ∀ {t} → τ ∷ Γ ⊢ t ∶ τ'
      → Γ ⊢ lam t ∶ τ ⇒ τ'
```

forget

remember

```
data _⊢_ : List Ty → Ty → 𝒰 where

  var : Γ ∋ τ
      → Γ ⊢ τ

  app : Γ ⊢ τ ⇒ τ'
      → Γ ⊢ τ
      → Γ ⊢ τ'

  lam : τ ∷ Γ ⊢ τ'
      → Γ ⊢ τ ⇒ τ'
```

```
from-toTyping : (t : Γ ⊢ τ) → fromTyping (toTyping t) ≡ t
to-fromTyping : ∀ {t} (d : Γ ⊢ t ∶ τ)
              → (toΛ (fromTyping d) , toTyping (fromTyping d))
              ≡ ((t , d) ⦂ Σ[ t' ∈ Λ ] Γ ⊢ t' ∶ τ)
```

# Extrinsic and Intrinsic Typing for STLC

## Step 6: Apply the theorems about the remember-forget pair

$$\Gamma \vdash \tau \cong \Sigma\,(t : \Lambda)\;\Gamma \vdash t : \tau$$

```
from-toTyping : (t : Γ ⊢ τ) → fromTyping (toTyping t) ≡ t
to-fromTyping : ∀ {t} (d : Γ ⊢ t : τ)
              → (toΛ (fromTyping d) , toTyping (fromTyping d))
              ≡ ((t , d) ⸴ Σ[ t' ∈ Λ ] Γ ⊢ t' : τ)
```

# Extrinsic and Intrinsic Typing for STLC

## Another theorem about algebraic ornamentation (Dagand & McBride, 2014)

$$\Gamma \vdash t : \tau \cong \Sigma\,(d \colon \Gamma \vdash \tau)\,(\mathsf{to}\Lambda\ d \equiv t)$$

# Extrinsic and Intrinsic Typing

$$\Gamma \vdash t : \tau \cong \Sigma\, (d \colon \Gamma \vdash \tau)\, (\mathsf{to}\Lambda\ d \equiv t)$$

- The ornament from raw terms to typing derivations has to be specified manually.
- ***Everything else follows*** from the theory of ornaments (Ko & Gibbons, 2011; Dagand & McBride, 2014; Ko et al., 2022).
- Can we even ***derive the ornament*** between raw terms and typing derivations?
  - Yes, if we restrict inductive types to typed syntaxes, e.g.
    - Allais et al.'s universe of syntaxes with binding (2021)
    - Aczel's binding signatures (1978)

# Bidirectional Typing

# Bidirectional Type Systems

**A formal treatment of bidirectional typing (Chen & Ko, 2024)**

$$\Gamma \vdash_{\Sigma,\Omega} t :^{d} A$$

- Consider a class of bidirectional simple-type systems specified by a signature $(\Sigma, \Omega)$, which consist of
  - Variables
  - Annotations
  - Subsumption
  - Constructs specified by ***binding arity*** with ***modes***
    - $\Delta_i$ for context extension
    - $A_i$ for argument type
    - $d_i$ is either $\Rightarrow$ (synthesis) or $\Leftarrow$ (checking)

$$\frac{(x : A) \in \Gamma}{\Gamma \vdash_{\Sigma,\Omega} x :^{\Rightarrow} A}\ \textsc{Var}^{\Rightarrow}$$

$$\frac{\Gamma \vdash_{\Sigma,\Omega} t :^{\Leftarrow} A}{\Gamma \vdash_{\Sigma,\Omega} (t \mathbin{⦂} A) :^{\Rightarrow} A}\ \textsc{Anno}^{\Rightarrow}$$

$$\frac{\Gamma \vdash_{\Sigma,\Omega} t :^{\Rightarrow} B \qquad B = A}{\Gamma \vdash_{\Sigma,\Omega} t :^{\Leftarrow} A}\ \textsc{Sub}^{\Leftarrow}$$

$$\frac{\rho \colon \mathsf{Sub}_{\Sigma}(\Xi, \emptyset) \qquad \Gamma, \vec{x}_1 : \Delta_1\langle\rho\rangle \vdash_{\Sigma,\Omega} t_1 :^{d_1} A_1\langle\rho\rangle \quad \cdots \quad \Gamma, \vec{x}_n : \Delta_n\langle\rho\rangle \vdash_{\Sigma,\Omega} t_n :^{d_n} A_n\langle\rho\rangle}{\Gamma \vdash_{\Sigma,\Omega} \mathsf{op}_o(\vec{x}_1.\, t_1; \ldots; \vec{x}_n.\, t_n) :^{d} A_0\langle\rho\rangle}\ \textsc{Op}$$

$$\text{for } o \colon \Xi \rhd [\Delta_1]A_1^{d_1}, \ldots, [\Delta_n]A_n^{d_n} \to A_0^{d} \text{ in } \Omega$$

# Soundness & Completeness for Bidirectional Typing

## How are these two derivations related?

$$\Gamma \vdash_{\Sigma,\Omega} t :^{d} A \qquad\qquad \Gamma \vdash_{\Sigma,\Omega} t : A$$

# Soundness & Completeness for Bidirectional Typing

## Step 1: Specify ornaments over the type of raw terms

$$\Gamma \vdash_{\Sigma,\Omega} t :^{d} A$$

$$|\Gamma| \vdash_{\Sigma,\Omega} t^{d}$$

$$\Gamma \vdash_{\Sigma,\Omega} t : A$$

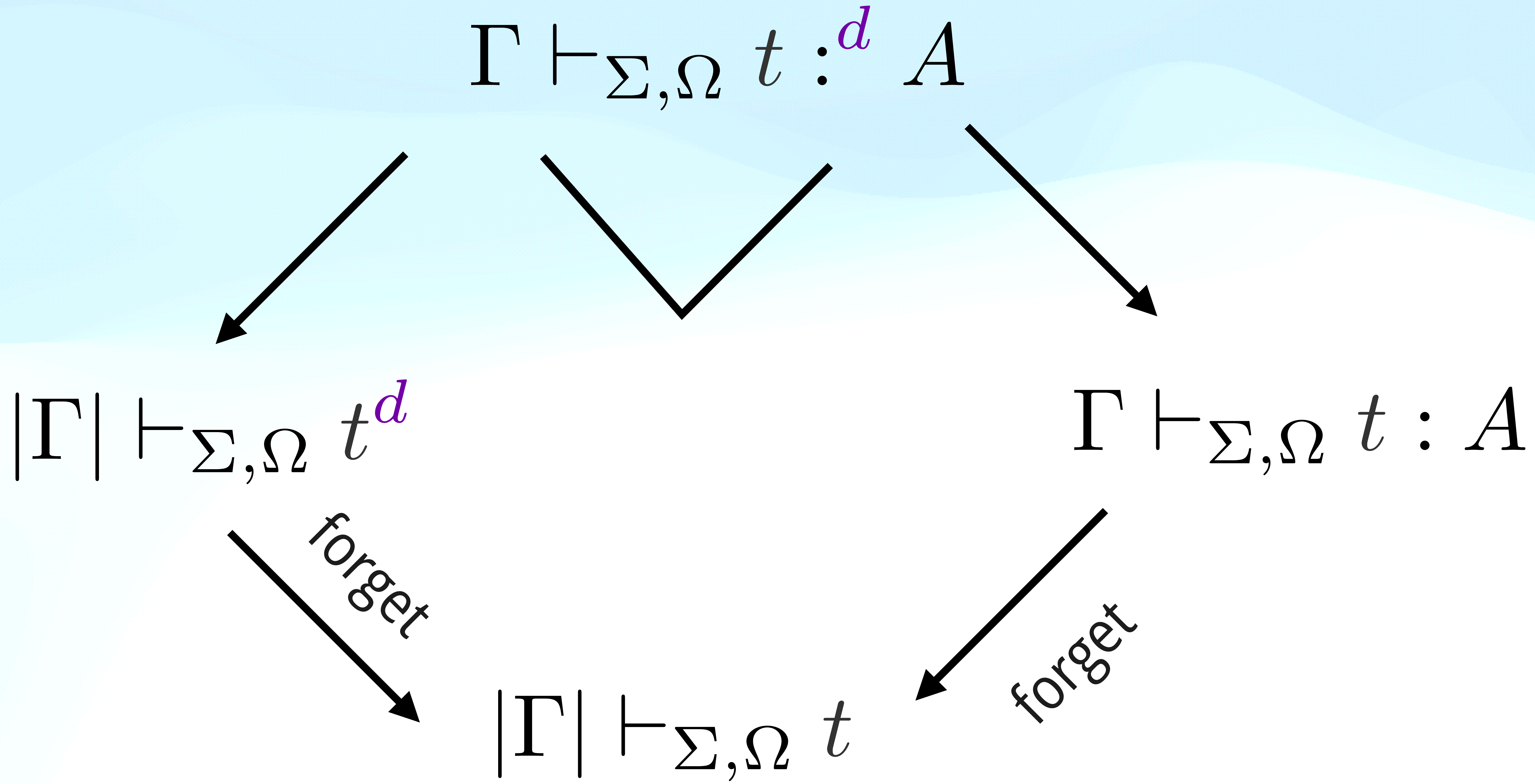
forget

forget

$$|\Gamma| \vdash_{\Sigma,\Omega} t$$

# Soundness & Completeness for Bidirectional Typing

## Step 2: Identify bidirectional typing as a pullback (Dagand & McBride, 2013; Ko, 2014)



# Soundness & Completeness for Bidirectional Typing

## Step 3: Apply the parallel composition of ornaments (Ko, 2014)

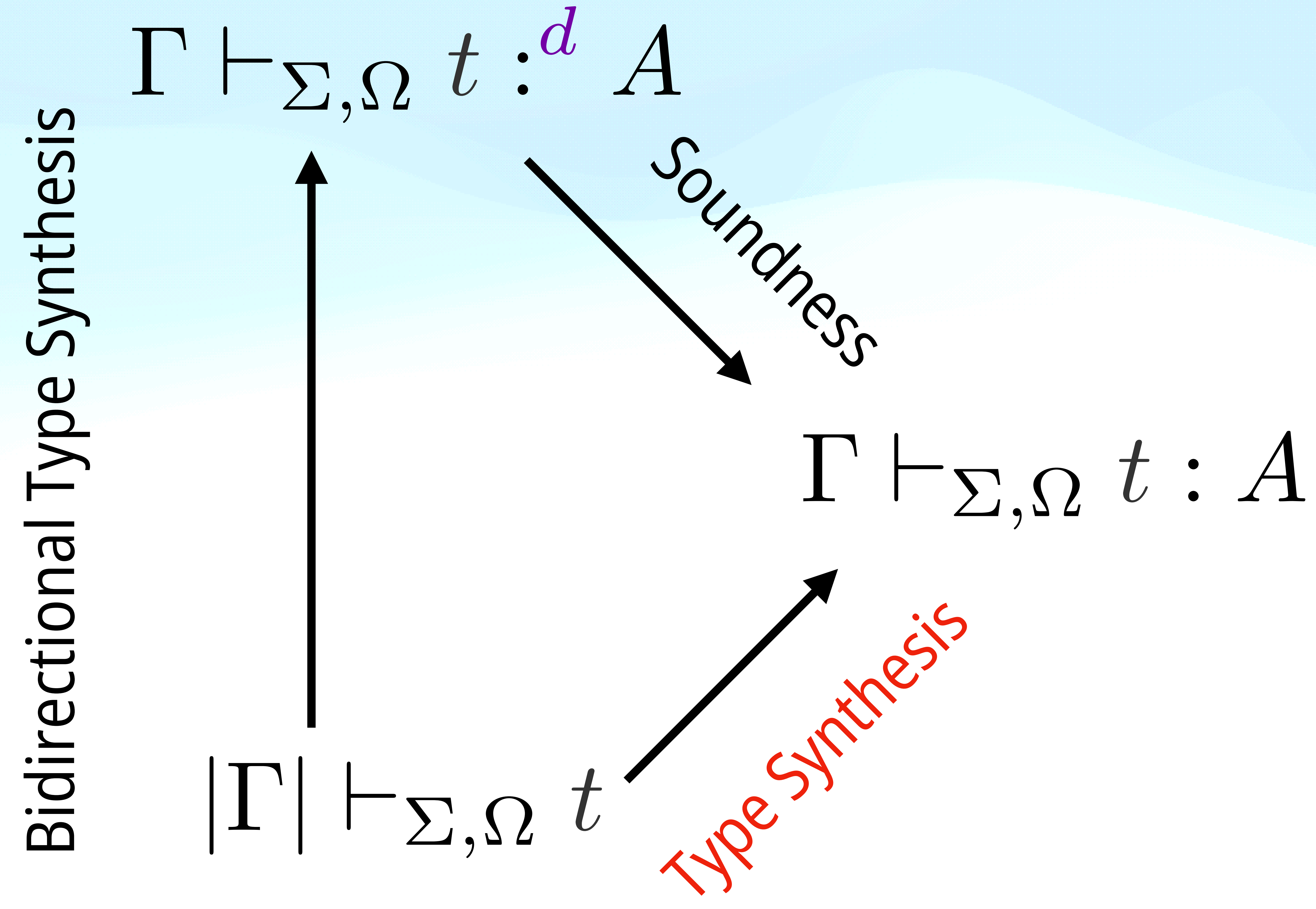
$$\Gamma \vdash_{\Sigma,\Omega} t :^{d} A \cong |\Gamma| \vdash_{\Sigma,\Omega} t^{d} \times \Gamma \vdash_{\Sigma,\Omega} t : A$$

That is, for a raw term $t$ with variables in $|\Gamma|$,

$\Gamma \vdash t :^{d} A$ if and only if $|\Gamma| \vdash t^{d}$ and $\Gamma \vdash t : A$

# Decidable Generic Bidirectional Type Synthesis

## Generic programs for more informative descriptions



# Decidable Generic Bidirectional Type Synthesis

$$|\Gamma| \vdash_{\Sigma,\Omega} t^{d}$$

$$\Gamma \vdash_{\Sigma,\Omega} t : A$$

Mode Decoration

$$|\Gamma| \vdash_{\Sigma,\Omega} t$$

# Decidable Generic Bidirectional Type Synthesis

$$\Gamma \vdash_{\Sigma,\Omega} t :^{d} A$$

Bidirectional Type Synthesis

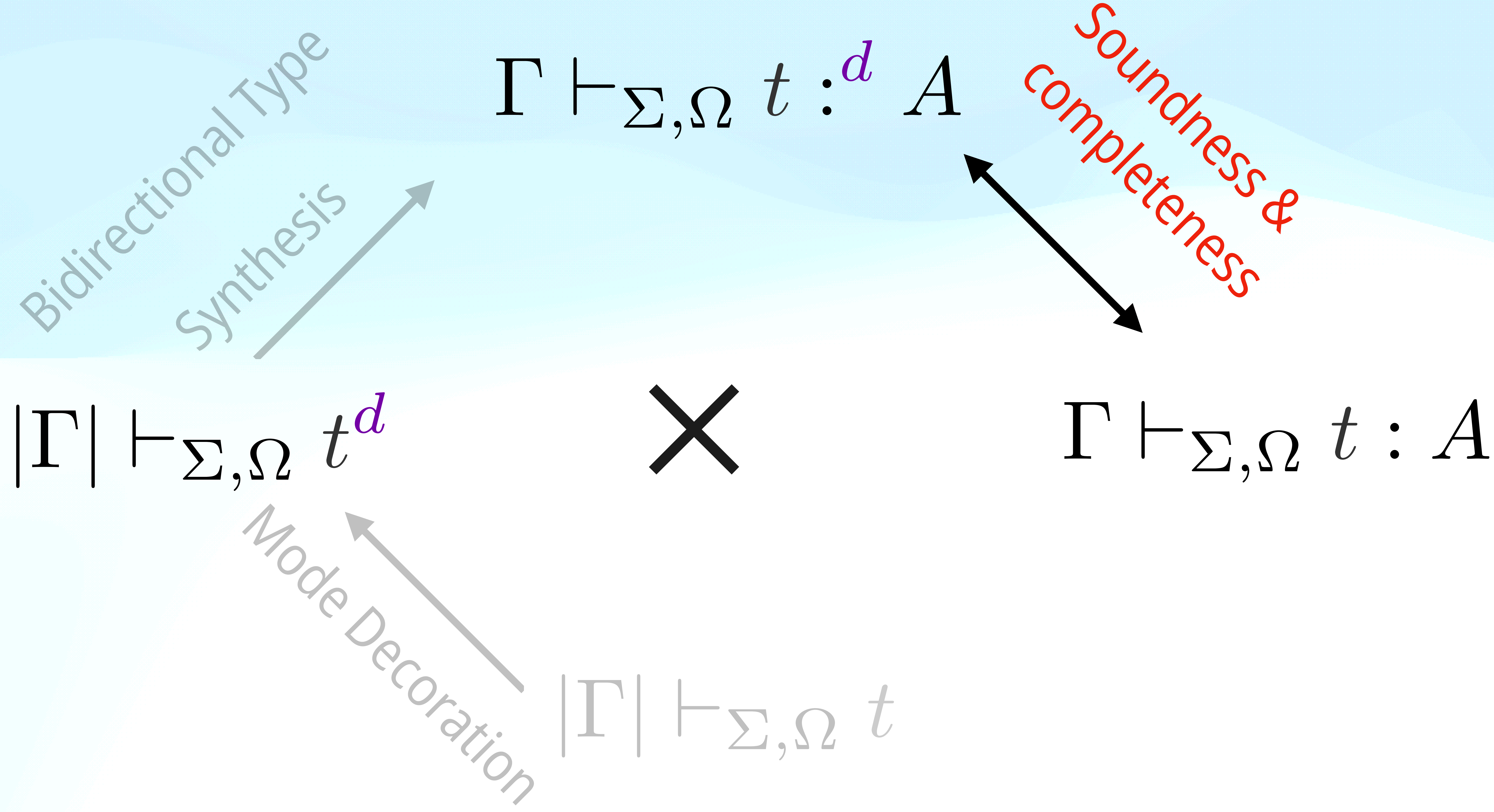
$$|\Gamma| \vdash_{\Sigma,\Omega} t^{d}$$

$$\Gamma \vdash_{\Sigma,\Omega} t : A$$

Mode Decoration

$$|\Gamma| \vdash_{\Sigma,\Omega} t$$

# Decidable Generic Bidirectional Type Synthesis



# Decidable Generic Bidirectional Type Synthesis

$$\Gamma \vdash_{\Sigma,\Omega} t :^{d} A$$

Soundness ↓

$$\Gamma \vdash_{\Sigma,\Omega} t : A$$

$$\Gamma \nvdash_{\Sigma,\Omega} t :^{d} A$$

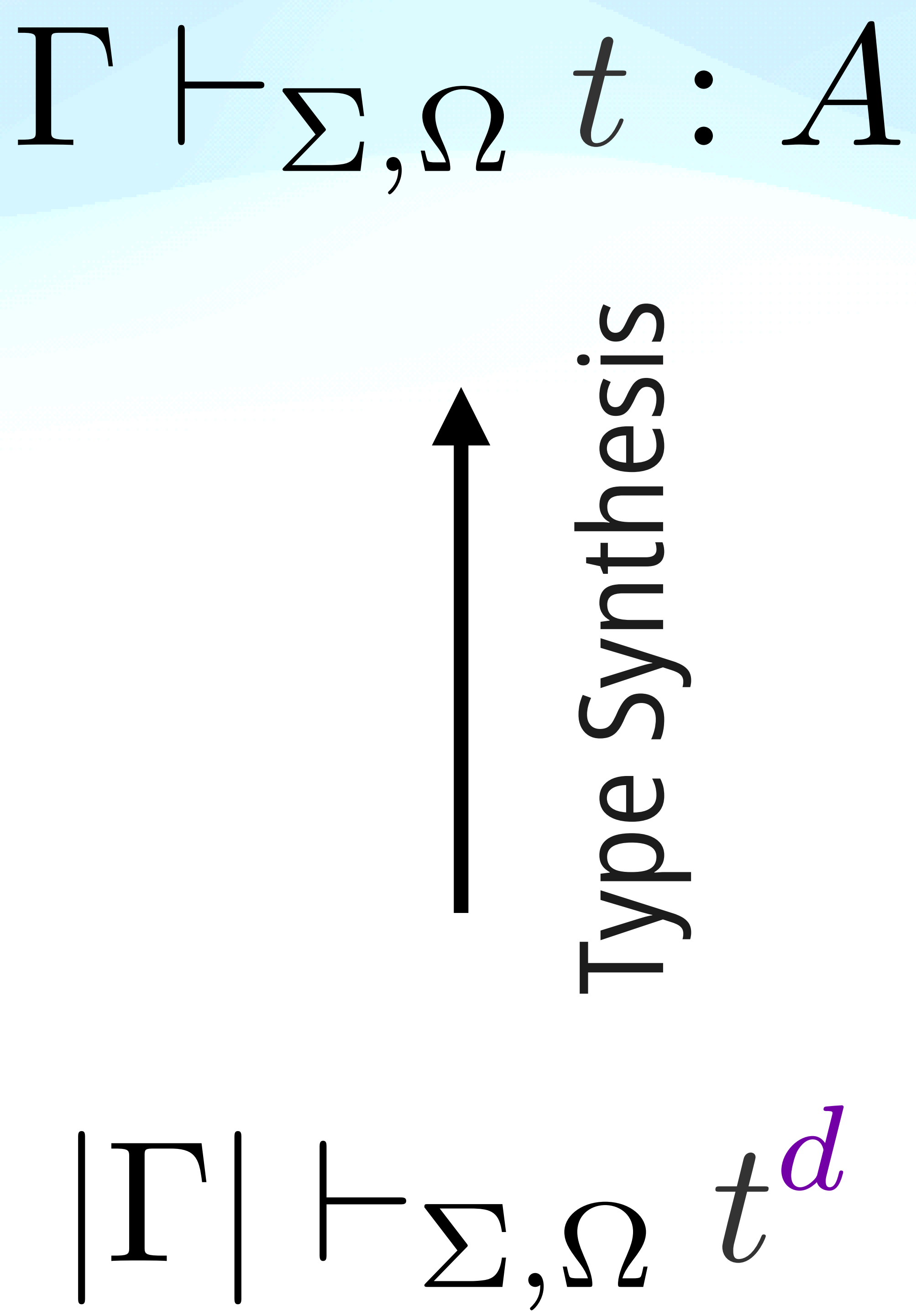
Completeness ↓ $|\Gamma| \vdash_{\Sigma,\Omega} t^{d}$

$$\Gamma \nvdash_{\Sigma,\Omega} t : A$$

# Decidable Generic Bidirectional Type Synthesis

## A formal treatment of bidirectional typing (Chen & Ko, 2024)

- Mode decoration is decidable: for every raw term $t$ under $V$, either $V \vdash t^d$ or $V \nvdash t^d$
- Every ***mode-correct*** bidirectional type system $(\Sigma, \Omega)$ has a decidable type synthesis: every $t$ and $d$: $|\Gamma| \vdash t^{\Rightarrow}$,
  - either $\Gamma \vdash t \Rightarrow A$ for some $A$
  - or $\Gamma \nvdash t \Rightarrow A$ for any $A$
- For a mode-correct $(\Sigma, \Omega)$, exactly one of the following holds:
  1. $|\Gamma| \nvdash t^{\Rightarrow}$
  2. $|\Gamma| \vdash t^{\Rightarrow}$ but $\Gamma \nvdash t : A$ for any $A$
  3. $|\Gamma| \vdash t^{\Rightarrow}$ and $\Gamma \vdash t : A$ for some $A$



# One-Hole Context (WIP)

# One-Hole Contexts for Data Types

## Differentiation of a polynomial

- Polynomials $I \xleftarrow{s} P \xrightarrow{f} S \xrightarrow{t} J$ are closed under products, sums, composition, and ***differentiation*** (Kock, unpublished), i.e.
  - $$\partial_i \sum_{s:S_j} \prod_{p:P_s} X_{r(p)} = \sum_{s:S_j} \sum_{l \in P_s, s(l)=i} \prod_{p \in P_s - l} X_{r(p)}$$
  - Differentiating a non-indexed data type (in the sense of polynomials) gives us a type of one-hole contexts and zipper (Huet 1997; McBride, 2001; Abbott et al., 2005).
- What is the differentiation of simply typed $\lambda$-calculus (Hamana & Fiore, 2011; Fiore, 2012)?

# One-Hole Contexts for Languages

## Contexts (in the sense of observational equivalence) as dependent zipper

```
data Λ : ℕ → 𝓤 where
  `_  : Fin n       → Λ n
  _·_ : Λ n → Λ n → Λ n
  ƛ_  : Λ (suc n) → Λ n
```

```
data ∂Λ : ℕ → ℕ → 𝓤 where
  hole :                          ∂Λ n n
  app₁ : ∂Λ m n → Λ n         → ∂Λ m n
  app₂ : Λ n     → ∂Λ m n     → ∂Λ m n
  ƛ_   : ∂Λ (suc m) (suc n) → ∂Λ (suc m) n
```

Are ***one-hole contexts*** of a language equivalent to the ***dependent zipper***?

$$F_\Lambda : \mathbf{Set}^{\mathbb{N}} \to \mathbf{Set}^{\mathbb{N}}$$

$$X_n \mapsto \mathsf{Fin}(n) + X_n \times X_n + X_{n+1}$$

$$\partial F_\Lambda : \mathbf{Set}^{\mathbb{N}} \to \mathbf{Set}^{\mathbb{N}\times\mathbb{N}}$$

$$\partial_i X_n \mapsto (i \equiv n) \times (X_n + X_n) + (i \equiv n + 1)$$

# Epilogue

# First-Class Datatype?

- Our previous work (Ko et al., 2022) is a *technical preview* for first-class data types.
  - Macros need to be *invoked explicitly* to reflect data types and reify descriptions.
  - Every instantiated program needs to be *tagged manually*.
- Chapman et al. (2010) describe a type theory with internalised descriptions.
  - Data type declaration becomes just a syntax sugar.
- Unfortunately, the described theory assumes $\mathscr{U} : \mathscr{U}$ and has not yet been implemented in any language.

# Language Genericity

**Thank you for your attention!**

- Viewing languages as data types allows us to apply generic programming techniques.
  - Ornaments for polynomial functors with equations and QIIT?
- First-class data types (should) enable us to use DGP naturally.
- Developing ***meta-meta***-theories of languages is advocated by Allais et al. (2021)
- Language-generic programming is a way to achieve a constructive meta-meta-theory.
- So, what else can we develop on the *meta-meta* level?