# The Essence of Generalized Algebraic Data Types

Filip Sieczkowski [1] **Sergei Stepanenko** [2] Jonathan Sterling [3] Lars Birkedal [2]

[1]Heriot-Watt University

[2]Aarhus University

[3]University of Cambridge

June 9, 2024

# Recalling GADTs

GADTs allow us to express stronger invariants.

E.g., vectors without dependent types.

```
data Zero :: *
data Succ :: * -> *

data VecNat :: * -> * where
  Nil :: VecNat Zero
  Cons :: forall n. Nat -> VecNat n -> VecNat (Succ n)
```

Types are used as indices
(which means that we need to reason about equalities of types).

# This work

- Extend relational reasoning techniques to languages with GADTs, to be able to show representation independence results.
  - Calculus for GADTs: $\mathsf{F}_{\omega\mu}^{=i}$.
  - Semantic models for $\mathsf{F}_{\omega\mu}^{=i}$.
    - Unary model for semantic type safety.
    - Binary model for reasoning about contextual equivalences.

# $\mathsf{F}_{\omega\mu}^{=i}$

| | | | |
|---|---|---|---|
| kinds | $\kappa$ | $::=$ | $* \mid \kappa \Rightarrow \kappa$ |
| constructors | $c$ | $::=$ | $\forall_\kappa \mid \exists_\kappa \mid \mu_\kappa \mid\!\to\!\mid \times \mid + \mid \mathsf{unit} \mid \mathsf{void}$ |
| constraints | $\chi$ | $::=$ | $\sigma \equiv_\kappa \tau$ |
| types | $\tau, \sigma$ | $::=$ | $\alpha \mid \lambda\alpha :: \kappa.\ \tau \mid \sigma\ \tau \mid c \mid \chi \to \tau \mid \chi \times \tau$ |

# $\mathsf{F}_{\omega\mu}^{=i}$

| | | | |
|---|---|---|---|
| kinds | $\kappa$ | $::=$ | $* \mid \kappa \Rightarrow \kappa$ |
| constructors | $c$ | $::=$ | $\forall_\kappa \mid \exists_\kappa \mid \mu_\kappa \mid \rightarrow \mid \times \mid + \mid \mathsf{unit} \mid \mathsf{void}$ |
| constraints | $\chi$ | $::=$ | $\sigma \equiv_\kappa \tau$ |
| types | $\tau, \sigma$ | $::=$ | $\alpha \mid \lambda\alpha :: \kappa.\ \tau \mid \sigma\ \tau \mid c \mid \chi \rightarrow \tau \mid \chi \times \tau$ |

| | | | |
|---|---|---|---|
| values | $v$ | $::=$ | $\ldots$ |
| | | | $\mid \lambda\bullet.\ e \mid \langle \bullet, v \rangle$ |
| expressions | $e$ | $::=$ | $\ldots$ |
| | | | $\mid \mathsf{abort}\ \bullet \mid v\ \bullet$ |
| | | | $\mid \mathsf{let}\ (\bullet, x) = v\ \mathsf{in}\ e$ |

- Type constructors are built-in functions on types.
- Constraint types are 'assert''s and 'assume''s for type equalities.
- Constraints are 'proof-irrelevant'.

# Reasoning about equalities I

## Provability

Computational rules ($\beta, \eta$ for types), injectivity, congruence.

$$\frac{c :: (\kappa_i \Rightarrow)_i\, \kappa \qquad \Delta \mid \Phi \Vdash c\ (\sigma_i)_i \equiv_\kappa c\ (\tau_i)_i}{(\Delta \mid \Phi \Vdash \sigma_i \equiv_{\kappa_i} \tau_i)_i}$$

$$\frac{\Delta \mid \Phi \Vdash \sigma_1 \times \tau_1 \equiv_* \sigma_2 \times \tau_2}{\Delta \mid \Phi \Vdash \tau_1 \equiv_* \tau_2}$$

# Reasoning about equalities I

## Provability

Computational rules ($\beta, \eta$ for types), injectivity, congruence.

$$\frac{c :: (\kappa_i \Rightarrow)_i\, \kappa \qquad \Delta \mid \Phi \Vdash c\ (\sigma_i)_i \equiv_\kappa c\ (\tau_i)_i}{(\Delta \mid \Phi \Vdash \sigma_i \equiv_{\kappa_i} \tau_i)_i}$$

$$\frac{\Delta \mid \Phi \Vdash \sigma_1 \times \tau_1 \equiv_* \sigma_2 \times \tau_2}{\Delta \mid \Phi \Vdash \tau_1 \equiv_* \tau_2}$$

## Discriminability

For impossible case elimination it is enough to look at the head symbols.

$$\frac{c_1 \neq c_2 \qquad (\Delta \vdash c_i\ \overline{\tau}_i :: \kappa)_{i \in \{1,2\}}}{\Delta \Vdash c_1\ \overline{\tau}_1 \#_\kappa c_2\ \overline{\tau}_2}$$

$$\frac{\Delta \vdash \tau_1 :: * \qquad \Delta \vdash \tau_2 :: * \qquad \Delta \vdash \sigma_1 :: * \qquad \Delta \vdash \sigma_2 :: *}{\Delta \Vdash \tau_1 + \sigma_1 \#_* \tau_2 \times \sigma_2}$$

# Reasoning about equalities II

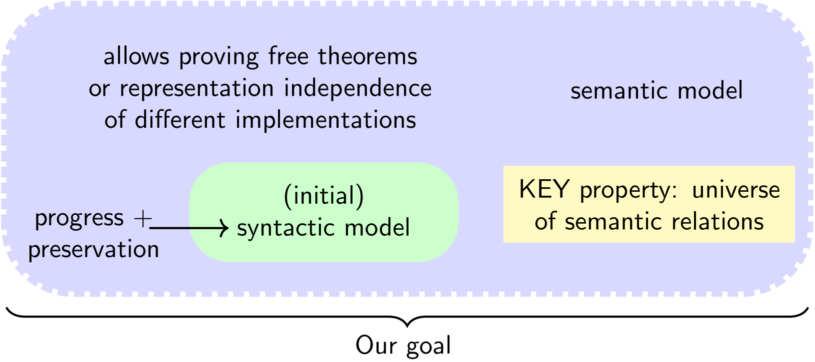
**Elimination of impossible equalities.**

$$\frac{\Delta \mid \Phi \Vdash \sigma_1 \equiv_\kappa \sigma_2 \qquad \Delta \Vdash \sigma_1 \#_\kappa \sigma_2 \qquad \Delta \vdash \tau :: *}{\Delta \mid \Phi \mid \Gamma \vdash \mathsf{abort}\,\bullet : \tau}$$

# $\mathsf{F}_{\omega\mu}^{=i}$ can represent GADTs

$$\mathsf{natvec} :: * \Rightarrow *$$
$$\mathsf{natvec} \triangleq$$
$$\mu\varphi :: * \Rightarrow *.\ \lambda\alpha :: *.$$
$$((\alpha \equiv_* \mathsf{void}) \times \mathsf{unit})$$
$$+(\mathbb{N} \times \exists\beta :: *.\ (\alpha \equiv_* (\beta + \mathsf{unit})) \times (\varphi\ \beta))$$

- natvec is either unit (and has void as its index)
- or not unit (and the tail has a smaller index).


allows proving free theorems
or representation independence
of different implementations
semantic model
progress + preservation
(initial) syntactic model
KEY property: universe of semantic relations
Our goal

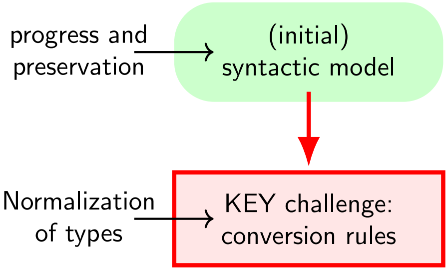
# Naïve approach

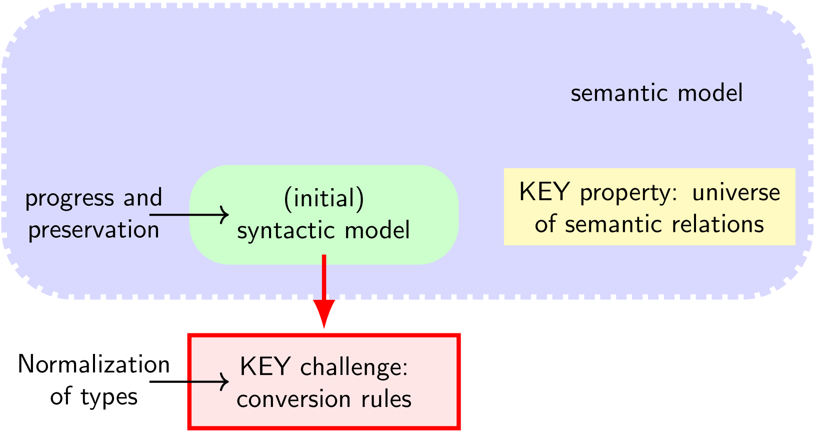
- Types are interpreted as sets of values. Constraints are interpreted as equalities of these sets.
- We can't validate injectivity rules, e.g., consider this instance:

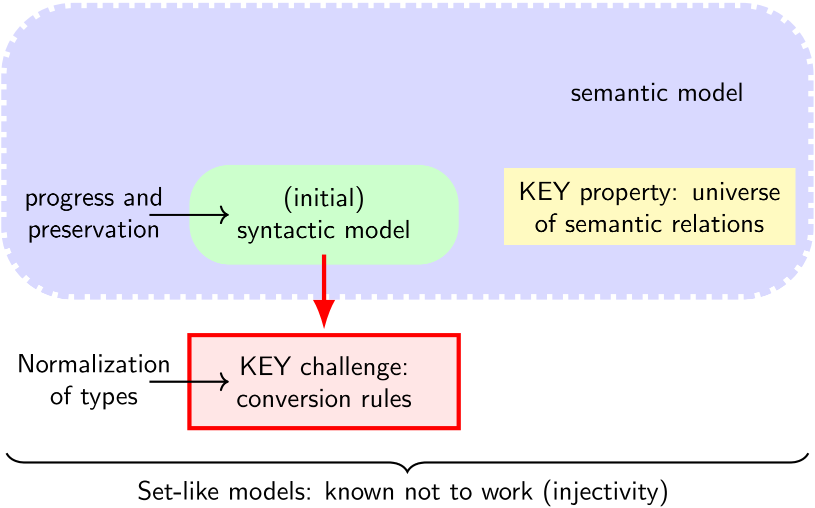
$$\frac{\Delta \mid \Phi \Vdash \mathsf{void} \times \tau_1 \equiv_* \mathsf{void} \times \tau_2}{\Delta \mid \Phi \Vdash \tau_1 \equiv_* \tau_2}$$

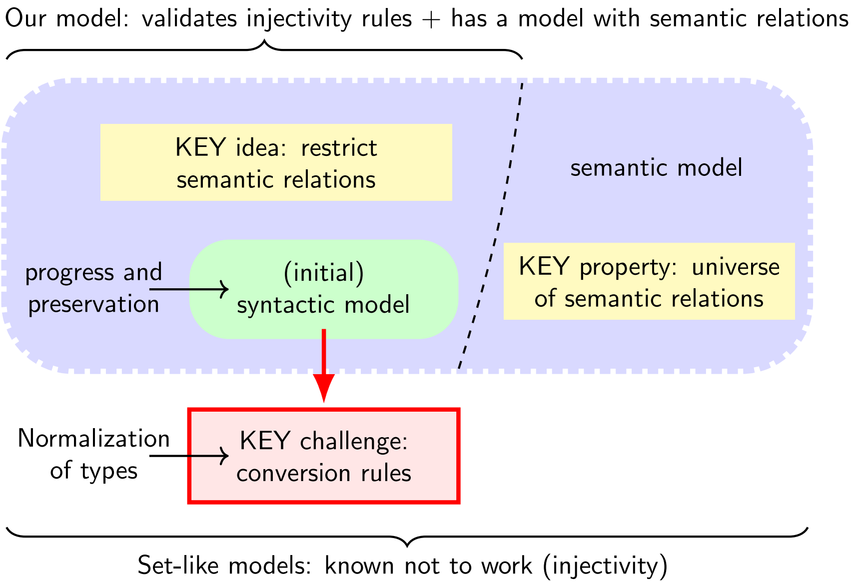
- If $\emptyset \times A = \emptyset \times B$, then it isn't necessarily true that $A = B$.

progress and preservation $\longrightarrow$ (initial) syntactic model


progress and preservation
(initial) syntactic model
Normalization of types
KEY challenge: conversion rules


semantic model
progress and preservation
(initial) syntactic model
KEY property: universe of semantic relations
Normalization of types
KEY challenge: conversion rules


semantic model
progress and preservation
(initial) syntactic model
KEY property: universe of semantic relations
Normalization of types
KEY challenge: conversion rules
Set-like models: known not to work (injectivity)


Our model: validates injectivity rules + has a model with semantic relations
KEY idea: restrict semantic relations
semantic model
progress and preservation
(initial) syntactic model
KEY property: universe of semantic relations
Normalization of types
KEY challenge: conversion rules
Set-like models: known not to work (injectivity)

# High-level view of model construction

Idea: two stages.

- The first stage helps to reason about equalities.
- The second stage is for sets of values.

# First stage

- *Normal forms* of types.
- Normalization (*NbE*).
- Syntactic equality of normal forms validates reduction rules for types.

# Modeling type equalities

$$(\Delta \vdash \tau \equiv_\kappa \sigma\ \mathsf{constr})\,\mathrm{true} \triangleq \mathsf{normal\ form\ of}\ \tau = \mathsf{normal\ form\ of}\ \sigma$$

# Second stage (unary model)

- We cannot use purely semantic predicates in $\forall$.
- Guarded recursion not only in case of recursive types, but also in $\forall$.
- We can interpret normal forms now, instead of arbitrary types.
- Syntactic equality of normal forms for constraints.

$$\mathcal{R}(\forall\alpha :: *.\ \tau)(v) \triangleq \exists e.\ v = \Lambda.\ e \wedge \forall\mu \in \mathrm{Neu}^{\cdot}_{*}. \cdots \rightarrow\ \triangleright\mathrm{wp}(\mathcal{R}(\mathrm{eval}(\tau[\alpha \mapsto \mu])))(e)$$
$$\mathcal{R}(\chi \times \nu)(v) \triangleq \exists v'.v = \langle \bullet, v' \rangle \wedge \chi\,\mathrm{true} \wedge \mathcal{R}(\nu)(v')$$

# Key observation

We can extend the syntax of normal forms at the base kind.

$$\frac{\varphi : X}{\varphi : \mathrm{Neu}_*^{\Delta}}$$

# Key observation

We can extend the syntax of normal forms at the base kind.

$$\frac{\varphi : X}{\varphi : \mathrm{Neu}_{*}^{\Delta}}$$

If $X$ is instantiated with relations on syntactic values, we can prove relational properties. This allows us to *combine* syntactic reasoning (via normal forms for types) and semantic reasoning.

# Key observation

We can extend the syntax of normal forms at the base kind.

$$\frac{\varphi : X}{\varphi : \mathrm{Neu}^{\Delta}_{*}}$$

If $X$ is instantiated with relations on syntactic values, we can prove relational properties. This allows us to *combine* syntactic reasoning (via normal forms for types) and semantic reasoning.

Our model walks on a thin line in-between being too *syntactical* (no relational reasoning) and being too semantical (invalid):

- If the interpretation of equalities is too *semantical*, we cannot validate injectivity rules.
- If we use equalities of normal forms to interpret equalities, but use just syntactical normal forms, we cannot validate the conversion rules.

**Contributions:**

- Calculus for studies of GADTs.
- Novel approach to study semantics of feature-rich languages with syntactic constraints for types.
- Semantical models of a language that allows us to express GADTs:
  - Unary model that validates potential extensions for languages with GADTs.
  - Binary model that allows reasoning about representation independence.
- Coq mechanization.

**and future:**

- Extensions (general effects).
- Relational interpretation of $\forall$ quantified at higher kinds.

# Placeholder before backup slides

# NbE

$$[\![*]\!] \triangleq \mathrm{Neu}_*$$
$$[\![\kappa_a \Rightarrow \kappa_r]\!] \triangleq [\![\kappa_a]\!] \Rightarrow [\![\kappa_r]\!]$$
$$[\![\Delta]\!] \triangleq \prod_{\alpha::\kappa\in\Delta} [\![\kappa]\!]$$

$$\mathrm{reify} : [\![\kappa]\!] \Rightarrow \mathrm{Nf}_\kappa$$
$$\mathrm{reflect} : \mathrm{Neu}_\kappa \Rightarrow [\![\kappa]\!]$$
$$\mathrm{eval} : \mathrm{Ty}_\kappa^\Delta \to ([\![\Delta]\!] \Rightarrow [\![\kappa]\!])$$

# Type-indexed vectors

The head function is now total! (We can eliminate the impossible case.)

$$
\begin{array}{l}
\mathsf{vhead} : \mathsf{nenatvec} \to \mathbb{N} \\
\mathsf{vhead}\; xs \triangleq \\
\quad \mathsf{let}\, (*, ys) = xs\, \mathsf{in} \\
\quad \mathsf{case}\ \mathsf{unroll}\, ys \\
\quad \mid \mathsf{inj}_1\, (\bullet, w).\ \mathsf{abort}\, \bullet \\
\quad \mid \mathsf{inj}_2\, \langle y, \_ \rangle.\ y
\end{array}
$$

# Setup for the second stage

We used step-indexed logic for this version of the calculus.
Language features (e.g., state) might require additional gadgets.

$$\tau ::= T \mid \mathit{Val} \mid \mathit{Expr} \mid \mathrm{Prop} \mid 1 \mid \tau + \tau \mid \tau \times \tau \mid \tau \to \tau$$

$$\begin{aligned} t, P ::= {} & x \mid v \mid e \mid F(t_1, \ldots, t_n) \mid \\ & () \mid (t, t) \mid \pi_i\ t \mid \lambda x : \tau.\, t \mid t(t) \mid \\ & \mathrm{inl}\ t \mid \mathrm{inr}\ t \mid \mathrm{case}(t, x.t, y.t) \mid \\ & \mathrm{False} \mid \mathrm{True} \mid t = t \mid P \Rightarrow P \mid P \wedge P \mid P \vee P \mid \\ & \exists x : \tau.\, P \mid \forall x : \tau.\, P \mid \triangleright P \mid \mu\, x : \tau.\, t \mid \ldots \end{aligned}$$

$$\frac{\Gamma, x : \tau \vdash t : \tau \qquad x \text{ is guarded in } t}{\Gamma \vdash \mu\, x : \tau.\, t : \tau}$$

# Logical relation

$$
\begin{aligned}
[\![\Phi]\!]_\eta\,\mathrm{true} &\triangleq \forall \varphi \in \Phi.\ [\![\varphi]\!]_\eta\,\mathrm{true}\\
[\![\Gamma]\!]_\eta &\triangleq \{\gamma \in \mathrm{dom}(\Gamma) \to \mathrm{Val} \mid \forall x \in \mathrm{dom}(\Gamma).\ \mathcal{R}(\mathrm{eval}(\Gamma(x))(\eta))(\gamma(x))\}\\
\Delta \mid \Phi \mid \Gamma \models e : \tau &\triangleq \forall \eta \in [\![\Delta]\!](\cdot).\ \mathrm{good}(\eta) \to [\![\Phi]\!]_\eta\,\mathrm{true} \to \forall \gamma \in [\![\Gamma]\!]_\eta \to \mathrm{wp}(\mathcal{R}(\mathrm{eval}(\tau)(\eta)))(e)
\end{aligned}
$$

# Injectivity and Cantor's paradox

Injectivity of some constructors implies false. It's a known fact, but can come up as a surprise.

For any injective constructor $c :: (* \Rightarrow *) \Rightarrow *$ and type $\alpha :: *$ it is possible to derive a value of type void in System $\mathsf{F}_{\omega}^{=i}$.

# Non-termination

For any injective constructor $c :: (* \Rightarrow *) \Rightarrow *$ and type $\alpha :: *$ it is possible to derive a value of type void in System $\mathsf{F}_\omega^{=i}$.

- $\tau_c^{\mathrm{loop}} \triangleq \exists \beta :: * \Rightarrow *.\ (c\ \beta \equiv_* \alpha) \times (\beta\ \alpha \to \mathsf{void})$
- $v^{\mathrm{loop}} \triangleq \lambda x.\ \mathsf{let}\,(*, (\bullet, y)) = x\,\mathsf{in}\,y\ (\mathsf{pack}\,\langle \bullet, y \rangle)$
- $\vdash v^{\mathrm{loop}} : \tau_c^{\mathrm{loop}}[(c\ (\lambda\alpha :: *.\ \tau_c^{\mathrm{loop}}))/\alpha] \to \mathsf{void},$
- $\vdash v^{\mathrm{loop}}\ (\mathsf{pack}\,\langle \bullet, v^{\mathrm{loop}} \rangle) : \mathsf{void}$

# Syntactic type-safety via NbE

**Lemma (Consistency)**

*A discriminable constraint is not provable in an empty context: in other words,* $\emptyset \mid \emptyset \Vdash \tau_1 \equiv_\kappa \tau_2$ *and* $\emptyset \Vdash \tau_1 \#_\kappa \tau_2$ *are contradictory.*

- Consequence of the injectivity of reify.
- Allows to discharge impossible cases.

**Lemma (Canonical form for arrows)**

*If* $v$ *is a closed value of type* $\tau$ *and* $\tau$ *is provably equal to some arrow type in an empty context, then* $v$ *is a lambda-abstraction with a well-typed body.*

$$(\emptyset \mid \emptyset \Vdash \tau \equiv_* (\tau_1 \to \tau_2)) \wedge (\emptyset \mid \emptyset \mid \Gamma \vdash v : \tau)$$
$$\implies (\exists xe.\ v = \lambda x.\ e \wedge \emptyset \mid \emptyset \mid \Gamma, x : \tau_1 \vdash e : \tau_2)$$

# Orthogonal extensions

References and concurrency.

$$
\begin{array}{llcl}
\text{constructors} & c & ::= & \cdots \mid \mathsf{ref} \\
\text{references} & \ell & ::= & \mathbb{N} \\
\text{values} & v & ::= & \cdots \mid \ell \\
\text{expressions} & e & ::= & \cdots \mid \mathsf{fork}\, e \mid \mathsf{alloc}\, v \mid v := v \mid !\, v
\end{array}
$$

The first stage stays the same, and the rest depends only on the logic used for defining $\mathcal{R}$.

The only requirements are that new effects should be expressed by type constructors, and that the ambient logic can express them.

# Type-safe red-black trees

```
data Red
data Black
data Tree a where
  Tree :: Node Black n a -> Tree a

data Node t n a where
  Nil :: Node Black Zero a
  BlackNode :: NodeH t0 t1 n a -> Node Black (Succ n) a
  RedNode :: NodeH Black Black n a -> Node Red n a
data NodeH l r n a = NodeH (Node l n a) a (Node r n a)
```

Stronger type invariants.

# Well-typed lambda terms

$$
\begin{aligned}
&\mathsf{Tm} :: * \Rightarrow * \\
&\mathsf{Tm} \triangleq \\
&\quad \mu\varphi :: * \Rightarrow *.\ \lambda\alpha :: *. \\
&\quad \alpha + (\exists\beta, \gamma :: *.\ (\alpha \equiv_* (\beta \to \gamma)) \times (\beta \to \varphi\ \gamma)) \\
&\quad\quad + (\exists\beta :: *.\ \varphi\ (\beta \to \alpha) \times \varphi\ \beta)
\end{aligned}
$$

# Well-typed lambda terms

$$
\begin{aligned}
&\mathsf{eval} : \forall\alpha :: *.\ \mathsf{Tm}\ \alpha \to \alpha \\
&\mathsf{eval} \triangleq \\
&\quad \mathsf{fix}\,\lambda f.\ \Lambda.\ \lambda x. \\
&\quad \mathsf{case\ unroll}\, x \\
&\quad \mid \mathsf{inj}_1\, y.\ y \\
&\quad \mid \mathsf{inj}_2\, y.\ \mathsf{case}\ y \\
&\qquad \mid \mathsf{inj}_1\,(*, (*, (\bullet, g))).\ \lambda z.\ f * (g\ z) \\
&\qquad \mid \mathsf{inj}_2\,(*, \langle g, x\rangle).\ (f * g)\ (f * x)
\end{aligned}
$$

# Logical relation for denotations

For any two bigger related contexts and arguments in this extended contexts, results are related after extension.

$$\eta \mid \nu_1 \approx_* \nu_2 \triangleq [\![\nu_1]\!]_\eta = \nu_2$$

$$\eta \mid \varphi_1 \approx_{\kappa_a \Rightarrow \kappa_r} \varphi_2 \triangleq \forall \Delta_1', \Delta_2', (\delta_1 : \mathsf{hom}_{\mathcal{K}}(\Delta_1', \Delta_1), \delta_2 : \mathsf{hom}_{\mathcal{K}}(\Delta_2', \Delta_2)), (\eta' : [\![\Delta_1']\!]^{\Delta_2'}), \mu_1, \mu_2.$$
$$(\delta_2^* \eta = \lambda x.\ \eta'(\delta_1(x))) \to (\eta' \mid \mu_1 \approx_{\kappa_a} \mu_2) \to (\eta' \mid \varphi_1(\delta_1, \mu_1) \approx_{\kappa_r} \varphi_2(\delta_2, \mu_2))$$

## Lemma

*If* $\eta \mid \mu_1 \approx \mu_2$, *then* $[\![\mathrm{reify}(\mu_1)]\!]_\eta = \mu_2$.
*If* $\eta \mid \eta_1 \approx \eta_2$, *then* $\eta \mid [\![\tau]\!]_{\eta_1} \approx [\![\tau]\!]_{\eta_2}$.

$$\imath \mid \nu \approx_\kappa \nu$$