The Essence of Generalized Algebraic Data Types

Filip Sieczkowski ¹  Sergei Stepanenko ²  Jonathan Sterling ³
Lars Birkedal ²

¹Heriot-Watt University
²Aarhus University
³University of Cambridge

November 30, 2023
Table of Contents

1 Motivation
   • Informal description
   • Languages with GADTs
   • Examples
   • State-of-the-Art

2 Language
   • Intuition
   • Definitions
   • Example programs

3 Semantics
   • First attempt
   • Second attempt

4 Conclusion
   • Contributions and future
Type-indexed vectors

\[
data \text{ Zero} :: * \\
data \text{ Succ} :: * \rightarrow * \\
\]
\[
data \text{ Vec} :: * \rightarrow * \rightarrow * \text{ where} \\
\text{ Nil} :: \forall a. \text{ Vec } a \text{ Zero} \\
\text{ Cons} :: \forall a n. a \rightarrow \text{ Vec } a n \rightarrow \text{ Vec } a (\text{ Succ } n) \\
\]
Fixed-length lists without dependent types.
Well-typed DSLs

data Expr :: * → * where
LiftExpr :: forall a. a → Expr a
LamExpr :: forall a b. (a → Expr b) → Expr (a → b)
AppExpr :: forall a b. Expr (a → b)
    → Expr a → Expr b
FixExpr :: forall a. Expr (a → a) → Expr a
...

Can be used to embed DSLs, and fallback onto Haskell for typechecking.
E.g., Pugs, Darcs.
Existing works on GADTs

31 relevant results on dblp:
- 8: categorical semantics
- 23: syntax, type-inference, implementation, usage

- Our work can be used for compiler developers, language designers, or for verification of data structures.
- And, moreover, we target feature-rich languages.
This work allows proving free theorems or representation independence of different implementations.

Model with semantic relations

KEY property: universe of semantic relations

Calculus for GADTs.

Semantical model for this calculus.
Short description

Our calculus:
System $F_\omega$
+ recursive types
+ type equalities
+ optional additional type constructors

Can be perceived as an IR used by a compiler.
Syntax

kinds \( \kappa ::= T | \kappa \Rightarrow \kappa \)

constructors \( c ::= \forall \kappa | \exists \kappa | \mu \kappa | \rightarrow | \times | + | \text{unit} | \text{void} \)

constraints \( \chi ::= \sigma \equiv_{\kappa} \tau \)

types \( \tau, \sigma ::= \alpha | \lambda \alpha :: \kappa. \tau | \sigma \tau | c | \chi \rightarrow \tau | \chi \times \tau \)
Syntax

kinds \( \kappa ::= T \mid \kappa \Rightarrow \kappa \)

constructors \( c ::= \forall \kappa \mid \exists \kappa \mid \mu \kappa \mid \rightarrow \mid \times \mid + \mid \text{unit} \mid \text{void} \)

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 \( v ::= x \mid \langle \rangle \mid \lambda x. e \mid \langle v_1, v_2 \rangle \mid \text{inj}_1 v \mid \text{inj}_2 v \mid \Lambda. e \mid \text{pack} v \mid \text{roll} v \mid \lambda \bullet. e \mid \langle \bullet, v \rangle \)

expressions \( e ::= \text{let } x = e_1 \text{ in } e_2 \mid v_1 v_2 \mid \text{proj}_1 v \mid \text{proj}_2 v \mid \text{case } v [x. e_1 \mid y. e_2] \mid v \mid \text{abort} \bullet \mid v \bullet \mid \text{let } (\bullet, x) = v \text{ in } e \)

eval. contexts \( E ::= \square \mid \text{let } x = E \text{ in } e \)

- Type constructors are built-in functions on types.
- Constraint types are ‘assert’s and ‘assume’s for type equalities.
Constraints don’t have any interesting operational semantics, and their introduction and elimination forms just ‘guide typechecking’.

\[(\lambda \bullet. \, e) \bullet \mapsto e\]
Constraints don’t have any interesting operational semantics, and their introduction and elimination forms just ‘guide typechecking’.

\[(\lambda \bullet. \ e) \bullet \mapsto e\]

\[\text{let } (\bullet, x) = \langle \bullet, v \rangle \text{ in } e \mapsto e[v/x]\]
Dynamic semantics

Constraints don’t have any interesting operational semantics, and their introduction and elimination forms just ‘guide typechecking’.

\[(\lambda \bullet. \ e) \bullet \rightarrow e\]

\[\text{let } (\bullet, x) = \langle \bullet, v \rangle \text{ in } e \rightarrow e[v/x]\]

Informally, we can consider constraints as singleton types.
Discriminability

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

\[
\begin{align*}
  c_1 \neq c_2 & \quad (\Delta \vdash c_i \bar{T}_i :: \kappa)_{i \in \{1,2\}} \\
  \Delta \not \models c_1 \bar{T}_1 \#_\kappa c_2 \bar{T}_2
\end{align*}
\]
Discriminability

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

\[
\begin{align*}
  c_1 \neq c_2 & \quad \frac{\left( \Delta \vdash c_i \bar{\tau}_i :: \kappa \right)_{i \in \{1,2\}}}{\Delta \models c_1 \bar{\tau}_1 \#_{\kappa} c_2 \bar{\tau}_2} \\
  \Delta \vdash \tau_1 :: T \quad \Delta \vdash \tau_2 :: T \quad \Delta \vdash \sigma_1 :: T \quad \Delta \vdash \sigma_2 :: T
\end{align*}
\]

\[
\Delta \models \tau_1 + \sigma_1 \#_T \tau_2 \times \sigma_2
\]
The main crux of the system — injectivity. That rule makes it possible to actually use GADTs (e.g., in case of well-typed terms) to derive non-trivial equalities.

\[
\begin{align*}
    c :: (\kappa_i \Rightarrow) \kappa & \quad \Delta \mid \Phi \vdash c (\sigma_i)_i \equiv_{\kappa} c (\tau_i)_i \\
    (\Delta \mid \Phi \vdash \sigma_i \equiv_{\kappa, \tau_i})_i &
\end{align*}
\]
The main crux of the system — injectivity. That rule makes it possible to actually use GADTs (e.g., in case of well-typed terms) to derive non-trivial equalities.

\[
\begin{align*}
c :: (\kappa_i \Rightarrow) \kappa & \quad \Delta \mid \Phi \vdash c (\sigma_i)_i \equiv \kappa (\tau_i)_i \\
(\Delta \mid \Phi \vdash \sigma_i \equiv \kappa_i \tau_i)_i
\end{align*}
\]

\[
\begin{align*}
\Delta \mid \Phi \vdash \sigma_1 \times \tau_1 \equiv_T \sigma_2 \times \tau_2 & \quad \Delta \mid \Phi \vdash \tau_1 \equiv_T \tau_2
\end{align*}
\]
We handle constraint passing manually to simplify semantical model. In real calculi with GADTs, this is handled by the typechecker.

\[
\frac{\Delta \vdash \chi \text{ constr}}{\Delta \mid \Phi \mid \Gamma \vdash \lambda \cdot. e : \chi \rightarrow \tau}
\]

\[
\frac{\Delta \mid \Phi \vdash \chi \quad \Delta \mid \Phi \mid \Gamma \vdash v : \tau}{\Delta \mid \Phi \mid \Gamma \vdash \langle \cdot, v \rangle : \chi \times \tau}
\]

\[
\frac{\Delta \mid \Phi \vdash \sigma_1 \equiv_{\kappa} \sigma_2 \quad \Delta \vdash \sigma_1 \#_{\kappa} \sigma_2}{\Delta \mid \Phi \mid \Gamma \vdash \tau :: T}
\]

\[
\frac{\Delta \mid \Phi \vdash \tau_1 \equiv_T \tau_2 \quad \Delta \mid \Phi \mid \Gamma \vdash e : \tau_1}{\Delta \mid \Phi \mid \Gamma \vdash e : \tau_2}
\]

\[
\frac{\Delta \mid \Phi \vdash \tau \equiv_T T}{\Delta \mid \Phi \mid \Gamma \vdash \text{abort} \cdot : \tau}
\]
Some classical examples of dependent types can be expressed with just GADTs, if we use them as ‘tags’.

\[
\text{natvec} :: T \Rightarrow T \\
\text{natvec} \triangleq \mu \varphi :: T \Rightarrow T. \lambda \alpha :: T. ((\alpha \equiv_T \text{void}) \times \text{unit}) + (\mathbb{N} \times \exists \beta :: T. (\alpha \equiv_T (\beta + \text{unit})) \times (\varphi \beta))
\]

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

\[
\text{nenatvec} :: T \\
\text{nenatvec} \triangleq \exists \alpha :: T. \text{natvec} (\alpha + \text{unit})
\]
**Type-indexed vectors**

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

\[
vhead : \text{nenatvec} \rightarrow \mathbb{N}
vhead \text{ } xs \triangleq \begin{align*}
&\text{let } (\ast, \text{ } ys) = xs \text{ in} \\
&\text{case unroll } \text{ } ys \\
&| \text{ inj}_1 (\bullet, w). \text{ abort } \bullet \\
&| \text{ inj}_2 \langle y, \_ \rangle. \text{ } y
\end{align*}
\]
Enough about syntax of our calculus! Can we actually prove something about it?
Naïve approach

- Types are interpreted as sets of values. Constraint are interpreted as equalities of these sets.
- We can’t validate injectivity rules, e.g., we need to be able to validate this instance:

\[
\Delta \mid \Phi \vdash \text{void} \times \tau_1 \equiv_T \text{void} \times \tau_2
\]

\[
\Delta \mid \Phi \vdash \tau_1 \equiv_T \tau_2
\]

- If \( \emptyset \times A = \emptyset \times B \), then it isn’t necessarily true that \( A = B \).
Our model: validates injectivity rules + has a model with semantic relations

**KEY idea:** restrict semantic relations

**NbE** progress + preservation

**KEY challenge:** injectivity rules

**KEY property:** universe of semantic relations

Known not to work (injectivity)

17
Our model: validates injectivity rules + has a model with semantic relations

KEY idea: restrict semantic relations (initial) syntactic model

NbE progress + preservation

KEY challenge: injectivity rules
Our model: validates injectivity rules + has a model with semantic relations

**KEY idea:** restrict semantic relations

**(initial)** syntactic model

**NbE** progress + preservation

**KEY challenge:** injectivity rules

**model with semantic relations**

**KEY property:** universe of semantic relations
Our model: validates injectivity rules + has a model with semantic relations

KEY idea: restrict semantic relations

NbE progress + preservation

(initial) syntactic model

KEY challenge: injectivity rules

Known not to work (injectivity)

KEY property: universe of semantic relations
Our model: validates injectivity rules + has a model with semantic relations

- **KEY idea:** restrict semantic relations

- **NbE progress + preservation**

- **(initial) syntactic model**

- **KEY property:** universe of semantic relations

- **KEY challenge:** injectivity rules

Known not to work (injectivity)
Idea: two stages.

- Before we compute the actual semantics, we need to do some equalities house-keeping.
- The first stage helps to reason about equalities.
- The second stage is for sets of values.
Normal forms

- We need an inductively defined universe of ‘codes’ for types.
- We can use NbE for types, and use their normal forms as codes.
First stage

- *Normalization by evaluation* for types.
- Syntax of normal and neutral forms for types.
- Normalization is performed by composition of `reify` and `eval`.

\[
\begin{align*}
[T] & \triangleq \text{Neu}_T \\
[k_a \Rightarrow \kappa_r] & \triangleq [k_a] \Rightarrow [k_r] \\
[\Delta] & \triangleq \prod_{\alpha::\kappa \in \Delta} [\kappa] \\
\text{reify} : [\kappa] & \Rightarrow \text{Nf}_{\kappa} \\
\text{reflect} : \text{Neu}_{\kappa} & \Rightarrow [\kappa] \\
\text{eval} : \text{Ty}_{\kappa} & \Rightarrow ([\Delta] \Rightarrow [\kappa])
\end{align*}
\]
We used step-indexed logic for this version of the calculus. Language features might require additional gadgets.

\[

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 \\
inl t \mid inr t \mid \text{case}(t, x.t, y.t) \mid \\
\text{False} \mid \text{True} \mid t =_\tau t \mid P \Rightarrow P \mid P \land P \mid P \lor P \mid \\
\exists x : \tau. P \mid \forall x : \tau. P \mid \triangleright P \mid \mu x : \tau. t
\]

\[
\Gamma, x : \tau \vdash t : \tau \quad \text{\(x\) is guarded in \(t\)} \\
\hline
\Gamma \vdash \mu x : \tau. t : \tau
\]
We can interpret normal forms now, instead of arbitrary types.

Syntactic equality of normal forms for constraints.

\[(\Delta \vdash \tau \equiv_{\kappa} \sigma \text{ constr}) \text{ true} \triangleq \]
\[
\forall (\Delta' : \text{Ctx})(\eta : \llbracket \Delta \rrbracket^{\Delta'}).\text{reify}(\text{eval}(\tau)(\eta)) = \text{reify}(\text{eval}(\sigma)(\eta))
\]

\[\mathcal{R}(\chi \times \nu)(\nu) \triangleq \exists \nu'. \nu = \langle \bullet, \nu' \rangle \land \chi \text{ true} \land \mathcal{R}(\nu)(\nu')\]
But what happens when we want to go under binders in types (at this point, it’s either forall or recursive types)?

\[ R(\forall \alpha :: \kappa. \tau)(v) \triangleq \exists e. \ v = \Lambda. e \land \forall \mu \in \wp(R(\tau + \text{something about } \mu \text{ and } \alpha))(e) \]
But what happens when we want to go under binders in types (at this point, it’s either forall or recursive types)?

\[ R(\forall \alpha :: \kappa. \tau)(\nu) \triangleq \exists e. \nu = \Lambda e \land \forall \mu \in ?.wp(R(\tau + \text{something about } \mu \text{ and } \alpha))(e) \]

What if we want to validate an equality in \( \tau \) that involves \( \alpha \)?

If we try to use predicates, we end up with a function from interpretation of \( \mathbb{N} \) to some arbitrary interpretation of \( \alpha \) and a syntactic constraint.

\[ R(\forall \alpha :: T. \forall \beta :: T. (\alpha \times \beta \equiv_T \mathbb{N} \times \beta) \to \mathbb{N} \to \alpha) \]
We cannot use purely semantic predicates for $\forall$.

Guarded recursion not only in case of recursive types, but also in case of $\forall$ (syntactic substitution strikes back).

We can interpret normal forms now, instead of arbitrary types.

Syntactic equality of normal forms for constraints.

\[
\begin{align*}
\mathcal{R}(\forall \alpha :: \kappa. \tau)(v) & \triangleq \exists e. \; v = \Lambda. \; e \land \forall \mu \in [\kappa](\cdot).\triangleright \text{wp}(\mathcal{R}(\text{eval}(\tau)([\alpha \mapsto \mu])))(e) \\
\mathcal{R}(\chi \times \nu)(v) & \triangleq \exists v'. \; v = \langle \bullet, v' \rangle \land \chi \text{ true} \land \mathcal{R}(\nu)(v')
\end{align*}
\]
Dealing with equalities

\[(\Delta \vdash \tau \equiv_{\kappa} \sigma \text{ constr}) \text{ true} \equiv \]
\[
\forall (\Delta' : \text{Ctx})(\eta : \llbracket \Delta \rrbracket^{\Delta'}).\text{reify}(\text{eval}(\tau)(\eta)) = \text{reify}(\text{eval}(\sigma)(\eta))
\]

- To validate the conversion rule we need reification to be injective.
- That’s what \textit{good} stands for.

**Lemma (Injectivity of reify)**

For any types \(\tau_1, \tau_2\) of kind \(\kappa\), well-formed in \(\Delta\) and any good environment \(\eta : \llbracket \Delta \rrbracket^{\Delta'}\), if \(\text{reify}(\text{eval}(\tau_1)(\eta)) = \text{reify}(\text{eval}(\tau_2)(\eta))\), then \(\text{eval}(\tau_1)(\eta) = \text{eval}(\tau_2)(\eta)\).
Second stage**

- We need to maintain a *good* environment.
- We cannot use purely semantic predicates in $\forall$.
- Guarded recursion not only in case of recursive types, but also in $\forall$ (syntactic substitution strikes back).
- We can interpret normal forms now, instead of arbitrary types.
- Syntactic equality of normal forms for constraints.

\[
\begin{align*}
R(\forall \alpha :: \kappa. \tau)(\nu) & \triangleq \exists e. \nu = \Lambda. e \land \forall \mu \in \llbracket \kappa \rrbracket (\cdot). \text{good}(\mu) \rightarrow \text{wp}(R(\text{eval}(\tau)([\alpha \mapsto \mu])))(e) \\
R(\chi \times \nu)(\nu) & \triangleq \exists \nu'. \nu = \langle \bullet, \nu' \rangle \land \chi \text{ true} \land R(\nu)(\nu')
\end{align*}
\]
Logical relation

We avoided a few problems so far:

- If interpretation of equalities is too *semantical*, we cannot validate injectivity rules.
- If we use equalities of normal forms to interpret equalities, but do not use *good* environments, we cannot validate conversion rules.

After that, we can actually validate all the rules, given a *good* environment for types, and valid contexts for constraints and term variables.
You might have a question: it’s good, but is it usable for anything apart from proving safety in a semantical way?
You might have a question: it’s good, but is it usable for anything apart from proving safety in a semantical way?

No!
You might have a question: it’s good, but is it usable for anything apart from proving safety in a semantical way?

No!

But we can cook up something.
We can extend the syntax with arbitrary stuff at the base kind (remember, that reify and reflect are ‘inert’ for base kind).

\[
\varphi : X \\
\frac{}{\varphi : \text{Neu}^\Delta} \\
\varphi : \text{Neu}^\Delta
\]

If $X$ is instantiated with predicates (or relations), we can verify syntactically not well-typed programs (or prove interesting binary properties).
To evaluate the model we prove representation independence of type-indexed vectors and lists.

Moreover, we don’t break anything in the process (e.g., we still can prove free theorems).
Limitations

- The model is not robust enough to ensure termination of the sub-calculus with restricted injectivity and no recursive types.
- Relational reasoning is limited to types at the base kind (well, programmers don’t use types at higher kinds in any case).
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 to express GADTs:
  - Unary model that validates potential extensions for languages with GADTs.
  - Binary model that allows to reason about representation independence (and (sic!) doesn’t break anything from System $F$).

and future:

- Extensions (general effects).
- Relational interpretation of $\forall$ quantified at higher kinds. (Less restrictive interpretation of $\forall$?)
- The end goal is to provide a setup, which can be used for designing a language that supports GADTs.
Placeholder before backup slides
Logical relation

\[ [\Phi]_\eta \text{ true} \triangleq \forall \varphi \in \Phi. [\varphi]_\eta \text{ true} \]

\[ [\Gamma]_\eta \triangleq \{ \gamma \in \text{dom}(\Gamma) \to \text{Val} \mid \forall x \in \text{dom}(\Gamma). R(\text{eval}(\Gamma(x))(\eta))(\gamma(x)) \} \]

\[ \Delta \mid \Phi \mid \Gamma \models e : \tau \triangleq \forall \eta \in [\Delta](\cdot). \text{good}(\eta) \to [\Phi]_\eta \text{ true} \to \forall \gamma \in [\Gamma]_\eta \to \text{wp}(R(\text{eval}(\tau)(\eta)))(e) \]
Injectivity of some constructors implies false. It’s a known fact, but can come up as a surprise.

For any injective constructor $c :: (T \Rightarrow T) \Rightarrow T$ and type $\alpha :: T$ it is possible to derive a value of type void in System $F_{\omega}^{i}$. 
Non-termination

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

\[
\tau_{c}^{\text{loop}} \triangleq \exists \beta :: T \Rightarrow T. (c \beta \equiv_{T} \alpha) \times (\beta \alpha \rightarrow \text{void})
\]

\[
\nu^{\text{loop}} \triangleq \lambda x. \text{let } (\ast, (\bullet, y)) = x \text{ in } y (\text{pack } \langle \bullet, y \rangle)
\]

\[
\vdash \nu^{\text{loop}} : \tau_{c}^{\text{loop}}[(c (\lambda \alpha :: T. \tau_{c}^{\text{loop}}))/\alpha] \rightarrow \text{void},
\]

\[
\vdash \nu^{\text{loop}} (\text{pack } \langle \bullet, \nu^{\text{loop}} \rangle) : \text{void}
\]
Syntactic type-safety via NbE

**Lemma (Consistency)**

*A discriminable constraint is not provable in an empty context: in other words, $\emptyset | \emptyset \vdash \tau_1 \equiv_{\kappa} \tau_2$ and $\emptyset \vdash \tau_1 \neq_{\kappa} \tau_2$ 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.*

$$(\emptyset | \emptyset \vdash \tau \equiv_T (\tau_1 \rightarrow \tau_2)) \land (\emptyset | \emptyset | \Gamma \vdash v : \tau) \implies (\exists x e. v = \lambda x. e \land \emptyset | \emptyset | \Gamma, x : \tau_1 \vdash e : \tau_2)$$
Orthogonal extensions

References and concurrency.

constructors  \( c ::= \cdots | \text{ref} \)
references  \( \ell ::= \mathbb{N} \)
values  \( v ::= \cdots | \ell \)
expressions  \( e ::= \cdots | \text{fork } e | \text{alloc } v | v ::= v | ! v \)

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 \rightarrow Tree a


data Node t n a where
  Nil :: Node Black Zero a

  BlackNode :: NodeH t0 t1 n a \rightarrow Node Black (Succ n) a

  RedNode :: NodeH Black Black Black n a \rightarrow 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

\[ Tm :: T \Rightarrow T \]
\[ Tm \triangleq \]
\[ \mu \varphi :: T \Rightarrow T. \lambda \alpha :: T. \alpha + (\exists \beta, \gamma :: T. (\alpha \equiv_T (\beta \rightarrow \gamma)) \times (\beta \rightarrow \varphi \gamma)) \]
\[ + (\exists \beta :: T. \varphi (\beta \rightarrow \alpha) \times \varphi \beta) \]
Well-typed lambda terms

\[
\text{eval} : \forall \alpha :: T. \ Tm \ \alpha \rightarrow \alpha
\]

\[
\text{eval} \triangleq
\]

\[
\text{fix } \lambda f. \Lambda. \lambda x.
\]

\[
\text{case unroll } x
\]

\[
| \text{inj}_1 y \cdot y
\]

\[
| \text{inj}_2 y \cdot \text{case } y
\]

\[
| \text{inj}_1 (\ast, (\ast, (\bullet, g))). \lambda z. \ f \ast (g \ z)
\]

\[
| \text{inj}_2 (\ast, \langle g, x \rangle). \ (f \ast g) \ (f \ast x)
\]
For any two bigger related contexts and arguments in this extended contexts, results are related after extension.

\[ \eta \mid \nu_1 \approx \nu_2 \triangleq [\nu_1]_\eta = \nu_2 \]
\[ \eta \mid \phi_1 \approx_{\kappa_\Lambda} \phi_2 \triangleq \forall \Delta_1', \Delta_2', (\delta_1 : \text{hom}_K(\Delta_1', \Delta_1), \delta_2 : \text{hom}_K(\Delta_2', \Delta_2)), (\eta' : [\Delta_1']^{\Delta_2'}), \mu_1, \mu_2.
\]
\[ (\delta_2^* \eta = \lambda x. \eta'(\delta_1(x))) \rightarrow (\eta' \mid \mu_1 \approx_{\kappa_\Lambda} \mu_2) \rightarrow (\eta' \mid \phi_1(\delta_1, \mu_1) \approx_{\kappa_\Lambda} \phi_2(\delta_2, \mu_2)) \]

**Lemma**

*If* \( \eta \mid \mu_1 \approx \mu_2 \), *then* \([\text{reify}(\mu_1)]_\eta = \mu_2 \).

*If* \( \eta \mid \eta_1 \approx \eta_2 \), *then* \( \eta \mid [\tau]_{\eta_1} \approx [\tau]_{\eta_2} \).

\[ \nu \mid \nu \approx_{\kappa_\Lambda} \nu \]