Semantics of Advanced Data Types

Patricia Johann
Appalachian State University

June 17, 2021

Course Outline

Lecture 1: Syntax and semantics of ADTs and nested types ✓

Lecture 2: Syntax and semantics of GADTs

Lecture 3: Parametricity for ADTs and nested types

Lecture 4: Parametricity for GADTs

Lecture 2:
Syntax and Semantics of GADTs

ADTs $\xhookrightarrow[\text{generalized by}]{\text{syntactically}}$ nested types $\xhookrightarrow[\text{generalized by}]{\text{syntactically}}$ GADTs

✓ ✓

What is a GADT?

- The shape of a GADT structure can depend on the data it contains.
- GADT data constructors can have both input types *and return types* involving instances of the data type being defined other than the one being defined.
- Fancier constructor types mean that GADTs can encode more sophisticated correctness properties.
- Sequences

```
data Seq : Set → Set where
  const : ∀{A : Set} → A → Seq A
  spair  : ∀{A B : Set} → Seq A × Seq B → Seq (A × B)
```

  Note that spair only constructs sequences of pair types.

- Polynomial expressions with variables of type A and coefficients of type B

```
data Expr : Set → Set → Set where
  var      : ∀{A B : Set} → A → Expr A B
  iconst   : ∀{A : Set} → Int → Expr A Int
  fconst   : ∀{A : Set} → Float → Expr A Float
  prod     : ∀{A B : Set} → Expr A B → Expr A B → Expr A B
  iscmult : ∀{A B : Set} → Expr A B → Int → Expr A B
  fscmult : ∀{A B : Set} → Expr A B → Float → Expr A Float
```

  Note that iconst, fconst, and fscmult again construct expressions at instances of certain forms of types only.

What is a GADT?

- The shape of a GADT structure can depend on the data it contains.
- GADT data constructors can have both input types *and return types* involving instances of the data type being defined other than the one being defined.
- Fancier constructor types mean that GADTs can encode more sophisticated correctness properties.
- Sequences

$$
\begin{array}{l}
\text{data Seq} : \text{Set} \to \text{Set where} \\
\quad \text{const} : \forall\{A : \text{Set}\} \to A \to \text{Seq } A \\
\quad \text{spair} \;\; : \forall\{A\, B : \text{Set}\} \to \text{Seq } A \times \text{Seq } B \to \text{Seq } (A \times B)
\end{array}
$$

  Note that spair only constructs sequences of pair types.

- Polynomial expressions with variables of type A and coefficients of type B

$$
\begin{array}{l}
\text{data Expr} : \text{Set} \to \text{Set} \to \text{Set where} \\
\quad \text{var} \quad\;\; : \forall\{A\, B : \text{Set}\} \to A \to \text{Expr } A\, B \\
\quad \text{iconst} \;\; : \forall\{A : \text{Set}\} \to \text{Int} \to \text{Expr } A\, \text{Int} \\
\quad \text{fconst} \;\, : \forall\{A : \text{Set}\} \to \text{Float} \to \text{Expr } A\, \text{Float} \\
\quad \text{prod} \quad : \forall\{A\, B : \text{Set}\} \to \text{Expr } A\, B \to \text{Expr } A\, B \to \text{Expr } A\, B \\
\quad \text{iscmult} : \forall\{A\, B : \text{Set}\} \to \text{Expr } A\, B \to \text{Int} \to \text{Expr } A\, B \\
\quad \text{fscmult} : \forall\{A\, B : \text{Set}\} \to \text{Expr } A\, B \to \text{Float} \to \text{Expr } A\, \text{Float}
\end{array}
$$

  Note that iconst, fconst, and fscmult again construct expressions at instances of certain forms of types only.

What is a GADT?

- The shape of a GADT structure can depend on the data it contains.
- GADT data constructors can have both input types *and return types* involving instances of the data type being defined other than the one being defined.
- Fancier constructor types mean that GADTs can encode more sophisticated correctness properties.
- Sequences

$$\begin{array}{l} \text{data Seq} : \text{Set} \rightarrow \text{Set where} \\ \quad \text{const} : \forall\{A : \text{Set}\} \rightarrow A \rightarrow \text{Seq } A \\ \quad \text{spair} \;\; : \forall\{A\, B : \text{Set}\} \rightarrow \text{Seq } A \times \text{Seq } B \rightarrow \text{Seq } (A \times B) \end{array}$$

  Note that spair only constructs sequences of pair types.
- Polynomial expressions with variables of type A and coefficients of type B

$$\begin{array}{l} \text{data Expr} : \text{Set} \rightarrow \text{Set} \rightarrow \text{Set where} \\ \quad \text{var} \quad\;\; : \forall\{A\, B : \text{Set}\} \rightarrow A \rightarrow \text{Expr } A\, B \\ \quad \text{iconst} \;\; : \forall\{A : \text{Set}\} \rightarrow \text{Int} \rightarrow \text{Expr } A \text{ Int} \\ \quad \text{fconst} \;\; : \forall\{A : \text{Set}\} \rightarrow \text{Float} \rightarrow \text{Expr } A \text{ Float} \\ \quad \text{prod} \quad : \forall\{A\, B : \text{Set}\} \rightarrow \text{Expr } A\, B \rightarrow \text{Expr } A\, B \rightarrow \text{Expr } A\, B \\ \quad \text{iscmult} : \forall\{A\, B : \text{Set}\} \rightarrow \text{Expr } A\, B \rightarrow \text{Int} \rightarrow \text{Expr } A\, B \\ \quad \text{fscmult} : \forall\{A\, B : \text{Set}\} \rightarrow \text{Expr } A\, B \rightarrow \text{Float} \rightarrow \text{Expr } A \text{ Float} \end{array}$$

  Note that iconst, fconst, and fscmult again construct expressions at instances of certain forms of types only.

What is a GADT?

- The shape of a GADT structure can depend on the data it contains.
- GADT data constructors can have both input types *and return types* involving instances of the data type being defined other than the one being defined.
- Fancier constructor types mean that GADTs can encode more sophisticated correctness properties.
- Sequences

$$
\begin{array}{l}
\mathsf{data\ Seq : Set \to Set\ where} \\
\quad \mathsf{const : \forall\{A : Set\} \to A \to Seq\ A} \\
\quad \mathsf{spair\ \ : \forall\{A\ B : Set\} \to Seq\ A \times Seq\ B \to Seq\ (A \times B)}
\end{array}
$$

  Note that spair only constructs sequences of pair types.

- Polynomial expressions with variables of type A and coefficients of type B

$$
\begin{array}{l}
\mathsf{data\ Expr : Set \to Set \to Set\ where} \\
\quad \mathsf{var\ \ \ \ \ : \forall\{A\ B : Set\} \to A \to Expr\ A\ B} \\
\quad \mathsf{iconst\ \ : \forall\{A : Set\} \to Int \to Expr\ A\ Int} \\
\quad \mathsf{fconst\ \ : \forall\{A : Set\} \to Float \to Expr\ A\ Float} \\
\quad \mathsf{prod\ \ \ \ : \forall\{A\ B : Set\} \to Expr\ A\ B \to Expr\ A\ B \to Expr\ A\ B} \\
\quad \mathsf{iscmult : \forall\{A\ B : Set\} \to Expr\ A\ B \to Int \to Expr\ A\ B} \\
\quad \mathsf{fscmult : \forall\{A\ B : Set\} \to Expr\ A\ B \to Float \to Expr\ A\ Float}
\end{array}
$$

  Note that iconst, fconst, and fscmult again construct expressions at instances of certain forms of types only.

What is a GADT?

- The shape of a GADT structure can depend on the data it contains.
- GADT data constructors can have both input types *and return types* involving instances of the data type being defined other than the one being defined.
- Fancier constructor types mean that GADTs can encode more sophisticated correctness properties.
- Sequences

$$
\begin{array}{l}
\mathsf{data\ Seq} : \mathsf{Set} \to \mathsf{Set\ where} \\
\quad \mathsf{const} : \forall\{\mathsf{A} : \mathsf{Set}\} \to \mathsf{A} \to \mathsf{Seq\ A} \\
\quad \mathsf{spair}\ \ : \forall\{\mathsf{A\ B} : \mathsf{Set}\} \to \mathsf{Seq\ A} \times \mathsf{Seq\ B} \to \mathsf{Seq\ (A} \times \mathsf{B)}
\end{array}
$$

  Note that spair only constructs sequences of pair types.
- Polynomial expressions with variables of type A and coefficients of type B

$$
\begin{array}{l}
\mathsf{data\ Expr} : \mathsf{Set} \to \mathsf{Set} \to \mathsf{Set\ where} \\
\quad \mathsf{var} \quad\quad : \forall\{\mathsf{A\ B} : \mathsf{Set}\} \to \mathsf{A} \to \mathsf{Expr\ A\ B} \\
\quad \mathsf{iconst} \ \ : \forall\{\mathsf{A} : \mathsf{Set}\} \to \mathsf{Int} \to \mathsf{Expr\ A\ Int} \\
\quad \mathsf{fconst} \ \ : \forall\{\mathsf{A} : \mathsf{Set}\} \to \mathsf{Float} \to \mathsf{Expr\ A\ Float} \\
\quad \mathsf{prod} \quad : \forall\{\mathsf{A\ B} : \mathsf{Set}\} \to \mathsf{Expr\ A\ B} \to \mathsf{Expr\ A\ B} \to \mathsf{Expr\ A\ B} \\
\quad \mathsf{iscmult} : \forall\{\mathsf{A\ B} : \mathsf{Set}\} \to \mathsf{Expr\ A\ B} \to \mathsf{Int} \to \mathsf{Expr\ A\ B} \\
\quad \mathsf{fscmult} : \forall\{\mathsf{A\ B} : \mathsf{Set}\} \to \mathsf{Expr\ A\ B} \to \mathsf{Float} \to \mathsf{Expr\ A\ Float}
\end{array}
$$

  Note that iconst, fconst, and fscmult again construct expressions at instances of certain forms of types only.

GADTs Are Not Functorial

- GADTs were functorial, they'd have shape-preserving, data-changing map functions.
- Consider $\text{map}_{\text{Seq}} : (A \to B) \to \text{Seq}\, A \to \text{Seq}\, B$
- The clause of map for const should have

$$\text{map}_{\text{Seq}}\ f\ (\text{const}\ x) = \text{const}\ (f\ x)$$

- What should the clause of map for spair be? If $f : C \times D \to E$ then

$$\text{map}_{\text{Seq}}\ f\ (\text{spair}\ s_1\ s_2) = \text{spair}\ ?\ ?$$

- What if $E \neq U \times V$?
- What if $E = U \times V$ but $f \neq (g : C \to U) \times (h : D \to V)$?
- Similarly, we can't construct the clause of map_{Expr} for iconst, fconst, or fscmult.
- GADTs do not support map functions because they are not data types in the usual container-y sense.
- Question: How do we give initial algebra semantics to GADTs?

GADTs Are Not Functorial

- GADTs were functorial, they'd have shape-preserving, data-changing map functions.
- Consider $\text{map}_{\text{Seq}} : (A \to B) \to \text{Seq}\,A \to \text{Seq}\,B$
- The clause of map for const should have

$$\text{map}_{\text{Seq}}\ f\ (\text{const}\ x) = \text{const}\ (f\ x)$$

- What should the clause of map for spair be? If $f : C \times D \to E$ then

$$\text{map}_{\text{Seq}}\ f\ (\text{spair}\ s_1\ s_2) = \text{spair}\ ?\ ?$$

- What if $E \neq U \times V$?
- What if $E = U \times V$ but $f \neq (g : C \to U) \times (h : D \to V)$?
- Similarly, we can't construct the clause of map_{Expr} for iconst, fconst, or fscmult.
- GADTs do not support map functions because they are not data types in the usual container-y sense.
- Question: How do we give initial algebra semantics to GADTs?

GADTs Are Not Functorial

- GADTs were functorial, they'd have shape-preserving, data-changing map functions.
- Consider $\mathsf{map}_{\mathsf{Seq}} : (A \to B) \to \mathsf{Seq}\,A \to \mathsf{Seq}\,B$
- The clause of map for const should have

$$\mathsf{map}_{\mathsf{Seq}}\ f\ (\mathsf{const}\ x) = \mathsf{const}\ (f\ x)$$

- What should the clause of map for spair be? If $f : C \times D \to E$ then

$$\mathsf{map}_{\mathsf{Seq}}\ f\ (\mathsf{spair}\ s_1\ s_2) = \mathsf{spair}\ ?\ ?$$

- What if $E \neq U \times V$?
- What if $E = U \times V$ but $f \neq (g : C \to U) \times (h : D \to V)$?
- Similarly, we can't construct the clause of $\mathsf{map}_{\mathsf{Expr}}$ for iconst, fconst, or fscmult.
- GADTs do not support map functions because they are not data types in the usual container-y sense.
- Question: How do we give initial algebra semantics to GADTs?

GADTs Are Not Functorial

- GADTs were functorial, they'd have shape-preserving, data-changing map functions.
- Consider $\text{map}_{\text{Seq}} : (A \to B) \to \text{Seq}\,A \to \text{Seq}\,B$
- The clause of map for const should have

$$\text{map}_{\text{Seq}}\ f\ (\text{const}\ x) = \text{const}\ (f\ x)$$

- What should the clause of map for spair be? If $f : C \times D \to E$ then

$$\text{map}_{\text{Seq}}\ f\ (\text{spair}\ s_1\ s_2) = \text{spair}\ ?\ ?$$

- What if $E \neq U \times V$?
- What if $E = U \times V$ but $f \neq (g : C \to U) \times (h : D \to V)$?
- Similarly, we can't construct the clause of map_{Expr} for iconst, fconst, or fscmult.
- GADTs do not support map functions because they are not data types in the usual container-y sense.
- Question: How do we give initial algebra semantics to GADTs?

GADTs Are Not Functorial

- GADTs were functorial, they'd have shape-preserving, data-changing map functions.
- Consider $\mathsf{map}_{\mathsf{Seq}} : (A \to B) \to \mathsf{Seq}\,A \to \mathsf{Seq}\,B$
- The clause of map for const should have

$$\mathsf{map}_{\mathsf{Seq}}\ f\ (\mathsf{const}\ x) = \mathsf{const}\ (f\ x)$$

- What should the clause of map for spair be? If $f : C \times D \to E$ then

$$\mathsf{map}_{\mathsf{Seq}}\ f\ (\mathsf{spair}\ s_1\ s_2) = \mathsf{spair}\ ?\ ?$$

- What if $E \neq U \times V$?
- What if $E = U \times V$ but $f \neq (g : C \to U) \times (h : D \to V)$?
- Similarly, we can't construct the clause of $\mathsf{map}_{\mathsf{Expr}}$ for iconst, fconst, or fscmult.
- GADTs do not support map functions because they are not data types in the usual container-y sense.
- Question: How do we give initial algebra semantics to GADTs?

GADTs Are Not Functorial

- GADTs were functorial, they'd have shape-preserving, data-changing map functions.
- Consider $\mathsf{map_{Seq}} : (A \to B) \to \mathsf{Seq}\, A \to \mathsf{Seq}\, B$
- The clause of map for const should have

$$\mathsf{map_{Seq}}\ f\ (\mathsf{const}\ x) = \mathsf{const}\ (f\ x)$$

- What should the clause of map for spair be? If $f : C \times D \to E$ then

$$\mathsf{map_{Seq}}\ f\ (\mathsf{spair}\ s_1\ s_2) = \mathsf{spair}\ ?\ ?$$

- What if $E \neq U \times V$?
- What if $E = U \times V$ but $f \neq (g : C \to U) \times (h : D \to V)$?
- Similarly, we can't construct the clause of $\mathsf{map_{Expr}}$ for iconst, fconst, or fscmult.
- GADTs do not support map functions because they are not data types in the usual container-y sense.
- Question: How do we give initial algebra semantics to GADTs?

GADTs Are Not Functorial

- GADTs were functorial, they'd have shape-preserving, data-changing map functions.
- Consider $\mathsf{map}_{\mathsf{Seq}} : (\mathsf{A} \to \mathsf{B}) \to \mathsf{Seq}\,\mathsf{A} \to \mathsf{Seq}\,\mathsf{B}$
- The clause of map for const should have

$$\mathsf{map}_{\mathsf{Seq}}\ \mathsf{f}\ (\mathsf{const}\ \mathsf{x}) = \mathsf{const}\ (\mathsf{f}\ \mathsf{x})$$

- What should the clause of map for spair be? If $\mathsf{f} : \mathsf{C} \times \mathsf{D} \to \mathsf{E}$ then

$$\mathsf{map}_{\mathsf{Seq}}\ \mathsf{f}\ (\mathsf{spair}\ \mathsf{s}_1\ \mathsf{s}_2) = \mathsf{spair}\ ?\ ?$$

- What if $\mathsf{E} \neq \mathsf{U} \times \mathsf{V}$?
- What if $\mathsf{E} = \mathsf{U} \times \mathsf{V}$ but $\mathsf{f} \neq (\mathsf{g} : \mathsf{C} \to \mathsf{U}) \times (\mathsf{h} : \mathsf{D} \to \mathsf{V})$?
- Similarly, we can't construct the clause of $\mathsf{map}_{\mathsf{Expr}}$ for iconst, fconst, or fscmult.
- GADTs do not support map functions because they are not data types in the usual container-y sense.
- Question: How do we give initial algebra semantics to GADTs?

GADTs Are Not Functorial

- GADTs were functorial, they'd have shape-preserving, data-changing map functions.
- Consider $\mathsf{map}_{\mathsf{Seq}} : (\mathsf{A} \to \mathsf{B}) \to \mathsf{Seq}\,\mathsf{A} \to \mathsf{Seq}\,\mathsf{B}$
- The clause of map for const should have

$$\mathsf{map}_{\mathsf{Seq}}\ \mathsf{f}\ (\mathsf{const}\ \mathsf{x}) = \mathsf{const}\ (\mathsf{f}\ \mathsf{x})$$

- What should the clause of map for spair be? If $\mathsf{f} : \mathsf{C} \times \mathsf{D} \to \mathsf{E}$ then

$$\mathsf{map}_{\mathsf{Seq}}\ \mathsf{f}\ (\mathsf{spair}\ \mathsf{s}_1\ \mathsf{s}_2) = \mathsf{spair}\ ?\,?$$

- What if $\mathsf{E} \neq \mathsf{U} \times \mathsf{V}$?
- What if $\mathsf{E} = \mathsf{U} \times \mathsf{V}$ but $\mathsf{f} \neq (\mathsf{g} : \mathsf{C} \to \mathsf{U}) \times (\mathsf{h} : \mathsf{D} \to \mathsf{V})$?
- Similarly, we can't construct the clause of $\mathsf{map}_{\mathsf{Expr}}$ for iconst, fconst, or fscmult.
- GADTs do not support map functions because they are not data types in the usual container-y sense.
- Question: How do we give initial algebra semantics to GADTs?

GADTs Are Not Functorial

- GADTs were functorial, they'd have shape-preserving, data-changing map functions.
- Consider $\mathsf{map}_{\mathsf{Seq}} : (\mathsf{A} \to \mathsf{B}) \to \mathsf{Seq}\,\mathsf{A} \to \mathsf{Seq}\,\mathsf{B}$
- The clause of map for const should have

$$\mathsf{map}_{\mathsf{Seq}}\ \mathsf{f}\ (\mathsf{const}\ \mathsf{x}) = \mathsf{const}\ (\mathsf{f}\ \mathsf{x})$$

- What should the clause of map for spair be? If $\mathsf{f} : \mathsf{C} \times \mathsf{D} \to \mathsf{E}$ then

$$\mathsf{map}_{\mathsf{Seq}}\ \mathsf{f}\ (\mathsf{spair}\ \mathsf{s}_1\ \mathsf{s}_2) = \mathsf{spair}\ ?\ ?$$

- What if $\mathsf{E} \neq \mathsf{U} \times \mathsf{V}$?
- What if $\mathsf{E} = \mathsf{U} \times \mathsf{V}$ but $\mathsf{f} \neq (\mathsf{g} : \mathsf{C} \to \mathsf{U}) \times (\mathsf{h} : \mathsf{D} \to \mathsf{V})$?
- Similarly, we can't construct the clause of $\mathsf{map}_{\mathsf{Expr}}$ for iconst, fconst, or fscmult.
- GADTs do not support map functions because they are not data types in the usual container-y sense.
- Question: How do we give initial algebra semantics to GADTs?

Recovering Functoriality

- There are two ways to recover functoriality. Both can be described in terms of left Kan extensions.
- The left Kan extension of $F : \mathcal{C} \to \mathcal{D}$ along $K : \mathcal{C} \to \mathcal{E}$ — denoted $Lan_K F$ — gives the "best functorial approximation" to F that factors through K.
- Intuitively, this means that $Lan_K F$ is the smallest functor that both extends the image of K to $\mathcal{D}$ and is such that the extension $Lan_K F \circ K$ agrees with F on $\mathcal{C}$, in the sense that there is a natural transformation η from F to $Lan_K F \circ K$.
- "Smallest" means that, for any other such extension G, there is a unique natural transformation δ from $Lan_K F$ to G such that the two natural transformations η and γ out of F are related nicely.

Recovering Functoriality

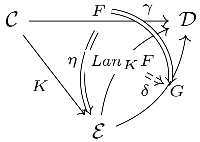
- There are two ways to recover functoriality. Both can be described in terms of left Kan extensions.
- The left Kan extension of $F : \mathcal{C} \to \mathcal{D}$ along $K : \mathcal{C} \to \mathcal{E}$ — denoted $Lan_K F$ — gives the "best functorial approximation" to F that factors through K.
- Intuitively, this means that $Lan_K F$ is the smallest functor that both extends the image of K to $\mathcal{D}$ and is such that the extension $Lan_K F \circ K$ agrees with F on $\mathcal{C}$, in the sense that there is a natural transformation η from F to $Lan_K F \circ K$.
- "Smallest" means that, for any other such extension G, there is a unique natural transformation δ from $Lan_K F$ to G such that the two natural transformations η and γ out of F are related nicely.

Recovering Functoriality

- There are two ways to recover functoriality. Both can be described in terms of left Kan extensions.
- The left Kan extension of $F : \mathcal{C} \to \mathcal{D}$ along $K : \mathcal{C} \to \mathcal{E}$ — denoted $Lan_K F$ — gives the "best functorial approximation" to F that factors through K.
- Intuitively, this means that $Lan_K F$ is the smallest functor that both extends the image of K to $\mathcal{D}$ and is such that the extension $Lan_K F \circ K$ agrees with F on $\mathcal{C}$, in the sense that there is a natural transformation η from F to $Lan_K F \circ K$.
- "Smallest" means that, for any other such extension G, there is a unique natural transformation δ from $Lan_K F$ to G such that the two natural transformations η and γ out of F are related nicely.

Recovering Functoriality

- There are two ways to recover functoriality. Both can be described in terms of left Kan extensions.
- The left Kan extension of $F : \mathcal{C} \to \mathcal{D}$ along $K : \mathcal{C} \to \mathcal{E}$ — denoted $Lan_K F$ — gives the "best functorial approximation" to F that factors through K.
- Intuitively, this means that $Lan_K F$ is the smallest functor that both extends the image of K to $\mathcal{D}$ and is such that the extension $Lan_K F \circ K$ agrees with F on $\mathcal{C}$, in the sense that there is a natural transformation η from F to $Lan_K F \circ K$.
- "Smallest" means that, for any other such extension G, there is a unique natural transformation δ from $Lan_K F$ to G such that the two natural transformations η and γ out of F are related nicely.



Left Kan Extensions

- If $F : \mathcal{C} \to \mathcal{D}$ and $K : \mathcal{C} \to \mathcal{E}$ are functors, then the *left Kan extension of F along K* is a functor $Lan_K\, F : \mathcal{E} \to \mathcal{D}$ together with a natural transformation $\eta : F \to Lan_K\, F \circ K$ such that, for every functor $G : \mathcal{E} \to \mathcal{D}$ and natural transformation $\gamma : F \to G \circ K$, there exists a unique natural transformation $\delta : Lan_K\, F \to G$ such that $(\delta K) \circ \eta = \gamma$.
- There is an isomorphism of natural transformations

$$F \to G \circ K \;\cong\; Lan_K F \to G$$

- If we add to our type system a type constructor Lan that is the syntactic reflection of the categorical Lan, then we can use (the syntactic reflection of) the above isomorphism to rewrite the syntax of our GADTs.
- This gives a "best approximation" *functorial completion* of GADT syntax that lets us rewrite GADT data constructor types in the same form as the types of data constructors for nested types.
- Functional completion lets us model GADTs as fixpoints of higher-order functors.

Left Kan Extensions

- If $F : \mathcal{C} \to \mathcal{D}$ and $K : \mathcal{C} \to \mathcal{E}$ are functors, then the *left Kan extension of F along K* is a functor $Lan_K\, F : \mathcal{E} \to \mathcal{D}$ together with a natural transformation $\eta : F \to Lan_K\, F \circ K$ such that, for every functor $G : \mathcal{E} \to \mathcal{D}$ and natural transformation $\gamma : F \to G \circ K$, there exists a unique natural transformation $\delta : Lan_K\, F \to G$ such that $(\delta K) \circ \eta = \gamma$.
- There is an isomorphism of natural transformations

$$F \to G \circ K \;\cong\; Lan_K F \to G$$

- If we add to our type system a type constructor Lan that is the syntactic reflection of the categorical Lan, then we can use (the syntactic reflection of) the above isomorphism to rewrite the syntax of our GADTs.
- This gives a "best approximation" *functorial completion* of GADT syntax that lets us rewrite GADT data constructor types in the same form as the types of data constructors for nested types.
- Functional completion lets us model GADTs as fixpoints of higher-order functors.

Left Kan Extensions

- If $F : \mathcal{C} \to \mathcal{D}$ and $K : \mathcal{C} \to \mathcal{E}$ are functors, then the *left Kan extension of F along K* is a functor $Lan_K\, F : \mathcal{E} \to \mathcal{D}$ together with a natural transformation $\eta : F \to Lan_K\, F \circ K$ such that, for every functor $G : \mathcal{E} \to \mathcal{D}$ and natural transformation $\gamma : F \to G \circ K$, there exists a unique natural transformation $\delta : Lan_K\, F \to G$ such that $(\delta K) \circ \eta = \gamma$.
- There is an isomorphism of natural transformations

$$F \to G \circ K \;\cong\; Lan_K F \to G$$

- If we add to our type system a type constructor Lan that is the syntactic reflection of the categorical Lan, then we can use (the syntactic reflection of) the above isomorphism to rewrite the syntax of our GADTs.
- This gives a "best approximation" *functorial completion* of GADT syntax that lets us rewrite GADT data constructor types in the same form as the types of data constructors for nested types.
- Functional completion lets us model GADTs as fixpoints of higher-order functors.

Left Kan Extensions

- If $F : \mathcal{C} \to \mathcal{D}$ and $K : \mathcal{C} \to \mathcal{E}$ are functors, then the *left Kan extension of* F *along* K is a functor $Lan_K\, F : \mathcal{E} \to \mathcal{D}$ together with a natural transformation $\eta : F \to Lan_K\, F \circ K$ such that, for every functor $G : \mathcal{E} \to \mathcal{D}$ and natural transformation $\gamma : F \to G \circ K$, there exists a unique natural transformation $\delta : Lan_K\, F \to G$ such that $(\delta K) \circ \eta = \gamma$.
- There is an isomorphism of natural transformations

$$F \to G \circ K \;\cong\; Lan_K F \to G$$

- If we add to our type system a type constructor Lan that is the syntactic reflection of the categorical Lan, then we can use (the syntactic reflection of) the above isomorphism to rewrite the syntax of our GADTs.
- This gives a "best approximation" *functorial completion* of GADT syntax that lets us rewrite GADT data constructor types in the same form as the types of data constructors for nested types.
- Functional completion lets us model GADTs as fixpoints of higher-order functors.

Rewriting GADT Syntax (I)

- We can rewrite Seq as follows:

$$
\begin{array}{l}
\mathsf{data\ Seq} : \mathsf{Set} \to \mathsf{Set\ where} \\
\quad \mathsf{const} : \forall\{\mathsf{A} : \mathsf{Set}\} \to \mathsf{A} \to \mathsf{Seq\,A} \\
\quad \mathsf{spair} \ : \forall\{\mathsf{A\,B} : \mathsf{Set}\} \to \underbrace{Seq\,A \times Seq\,B}_{\mathsf{F\,A\,B}} \to \underbrace{Seq}_{\mathsf{G}}\,(\underbrace{A \times B}_{\mathsf{K\,A\,B}}) \\
\quad \mathsf{spair} \ : \forall\{\mathsf{A} : \mathsf{Set}\} \to (\mathsf{Lan}_{\lambda \mathsf{A\,B}.\, \mathsf{A} \times \mathsf{B}}\, \lambda \mathsf{A\,B}.\, \mathsf{Seq\,A} \times \mathsf{Seq\,B})\,\mathsf{A} \to \mathsf{Seq\,A}
\end{array}
$$

- Then Seq can be interpreted as μH for the higher-order functor

$$H\,F\,X = X + (Lan_{\lambda XY.\,X \times Y}\, \lambda XY.\, F X \times F Y)\, X$$

Rewriting GADT Syntax (I)

- We can rewrite Seq as follows:

$$
\begin{array}{l}
\text{data Seq} : \text{Set} \to \text{Set where} \\
\quad \text{const} : \forall\{\text{A} : \text{Set}\} \to \text{A} \to \text{Seq A} \\
\quad \text{spair} \;\; : \forall\{\text{A B} : \text{Set}\} \to \underbrace{\mathit{Seq}\, A \times \mathit{Seq}\, B}_{\text{F A B}} \to \underbrace{\mathit{Seq}}_{\text{G}}\,(\underbrace{A \times B}_{\text{K A B}}) \\
\quad \text{spair} \;\; : \forall\{\text{A} : \text{Set}\} \to (\text{Lan}_{\lambda \text{A B. A} \times \text{B}}\, \lambda \text{A B. Seq A} \times \text{Seq B})\, \text{A} \to \text{Seq A}
\end{array}
$$

- Then Seq can be interpreted as μH for the higher-order functor

$$H\, F\, X = X + (Lan_{\lambda XY.\, X \times Y}\, \lambda XY.\, FX \times FY)\, X$$

Rewriting GADT Syntax (I)

- We can rewrite Seq as follows:

$$
\begin{array}{l}
\mathsf{data\ Seq : Set \to Set\ where} \\
\quad \mathsf{const} : \forall\{\mathsf{A : Set}\} \to \mathsf{A} \to \mathsf{Seq\ A} \\
\quad \mathsf{spair}\ : \forall\{\mathsf{A\,B : Set}\} \to \underbrace{Seq\,A \times Seq\,B}_{\mathsf{F\,A\,B}} \to \underbrace{Seq}_{\mathsf{G}}\,(\underbrace{A \times B}_{\mathsf{K\,A\,B}}) \\
\quad \mathsf{spair}\ : \forall\{\mathsf{A : Set}\} \to (\mathsf{Lan}_{\lambda \mathsf{A\,B.\,A \times B}}\, \lambda \mathsf{A\,B.\,Seq\,A} \times \mathsf{Seq\,B})\,\mathsf{A} \to \mathsf{Seq\,A}
\end{array}
$$

- Then Seq can be interpreted as μH for the higher-order functor

$$H\,F\,X = X + (Lan_{\lambda XY.\,X \times Y}\,\lambda XY.\,FX \times FY)\,X$$

Rewriting GADT Syntax (II)

- We can rewrite Expr as follows:

  data Expr : Set → Set → Set where
  var : ∀{A B : Set} → A → Expr A B
  iconst : ∀{A : Set} → Int → Expr A Int
  iconst : ∀{A B : Set} → $(\mathsf{Lan}_{\lambda A\,B.A \times \mathsf{Int}}\ \lambda A\,B.\ \mathsf{Int})\ A\ B$ → Expr A B
  fconst : ∀{A : Set} → Float → Expr A Float
  fconst : ∀{A B : Set} → $(\mathsf{Lan}_{\lambda A\,B.A \times \mathsf{Float}}\ \lambda A\,B.\ \mathsf{Float})\ A\ B$ → Expr A B
  prod : ∀{A B : Set} → Expr A B → Expr A B → Expr A B
  iscmult : ∀{A B : Set} → Expr A B → Int → Expr A B
  fscmult : ∀{A B : Set} → Expr A B → Float → Expr A Float
  fscmult : ∀{A B : Set} → $(\mathsf{Lan}_{\lambda A\,B.A \times \mathsf{Float}}\ \lambda A\,B.\ \mathsf{Expr}\ A\ B \times \mathsf{Float})\ A\ B$ → Expr A B

- Then Expr can be interpreted as μH for the higher-order functor

$$
\begin{aligned}
H\,F\,X \quad = \quad & \pi_1\,X \\
& + (Lan_{\lambda X\,Y.X \times Int}\ \lambda X\,Y.\,Int)\ X \\
& + (Lan_{\lambda X\,Y.X \times Float}\ \lambda X\,Y.\,Float)\ X \\
& + F\,X\,Y \times F\,X\,Y \\
& + F\,X\,Y \times Int \\
& + (Lan_{\lambda X\,Y.X \times Float}\ \lambda X\,Y.\,F\,X\,Y \times Float)\ X
\end{aligned}
$$

Rewriting GADT Syntax (II)

- We can rewrite Expr as follows:

$$
\begin{array}{l}
\text{data Expr} : \text{Set} \to \text{Set} \to \text{Set where} \\
\quad \text{var} \quad : \forall\{\text{A B} : \text{Set}\} \to \text{A} \to \text{Expr A B} \\
\quad \text{iconst} \quad : \forall\{\text{A} : \text{Set}\} \to \text{Int} \to \text{Expr A Int} \\
\quad \text{iconst} \quad : \forall\{\text{A B} : \text{Set}\} \to (\text{Lan}_{\lambda \text{A B}. \text{A} \times \text{Int}}\, \lambda \text{A B}.\, \text{Int})\ \text{A B} \to \text{Expr A B} \\
\quad \text{fconst} \quad : \forall\{\text{A} : \text{Set}\} \to \text{Float} \to \text{Expr A Float} \\
\quad \text{fconst} \quad : \forall\{\text{A B} : \text{Set}\} \to (\text{Lan}_{\lambda \text{A B}. \text{A} \times \text{Float}}\, \lambda \text{A B}.\, \text{Float})\ \text{A B} \to \text{Expr A B} \\
\quad \text{prod} \quad : \forall\{\text{A B} : \text{Set}\} \to \text{Expr A B} \to \text{Expr A B} \to \text{Expr A B} \\
\quad \text{iscmult} : \forall\{\text{A B} : \text{Set}\} \to \text{Expr A B} \to \text{Int} \to \text{Expr A B} \\
\quad \text{fscmult} : \forall\{\text{A B} : \text{Set}\} \to \text{Expr A B} \to \text{Float} \to \text{Expr A Float} \\
\quad \text{fscmult} : \forall\{\text{A B} : \text{Set}\} \to (\text{Lan}_{\lambda \text{A B}. \text{A} \times \text{Float}}\, \lambda \text{A B}.\, \text{Expr A B} \times \text{Float})\ \text{A B} \to \text{Expr A B}
\end{array}
$$

- Then Expr can be interpreted as μH for the higher-order functor

$$
\begin{array}{rcl}
H\,F\,X & = & \quad \pi_1\,X \\
 & & + \, (Lan_{\lambda X\,Y. X \times Int}\ \lambda X\,Y.\, Int)\ X \\
 & & + \, (Lan_{\lambda X\,Y. X \times Float}\ \lambda X\,Y.\, Float)\ X \\
 & & + \, F\,X\,Y \times F\,X\,Y \\
 & & + \, F\,X\,Y \times Int \\
 & & + \, (Lan_{\lambda X\,Y. X \times Float}\ \lambda X\,Y.\, F\,X\,Y \times Float)\ X
\end{array}
$$

Completion Choices

- At the level of objects, this gives (at least) the syntactic data elements for GADTs.
- But what about morphisms? What about natural transformations?
- There are two obvious choices:

  - The discrete category $|\mathcal{C}|$ — equivalently, the discrete category $\mathcal{I}$ of interpretations of types in $\mathcal{C}$.

  - The full category $\mathcal{C}$.

Completion Choices

- At the level of objects, this gives (at least) the syntactic data elements for GADTs.
- But what about morphisms? What about natural transformations?
- There are two obvious choices:

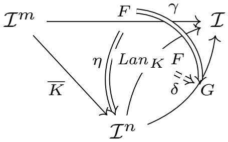
  - The discrete category $|\mathcal{C}|$ — equivalently, the discrete category $\mathcal{I}$ of interpretations of types in $\mathcal{C}$.

  - The full category $\mathcal{C}$.

Completion Choices

- At the level of objects, this gives (at least) the syntactic data elements for GADTs.
- But what about morphisms? What about natural transformations?
- There are two obvious choices:

  - The discrete category $|\mathcal{C}|$ — equivalently, the discrete category $\mathcal{I}$ of interpretations of types in $\mathcal{C}$.

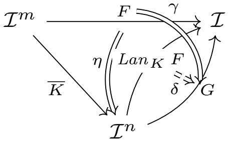


  - The full category $\mathcal{C}$.

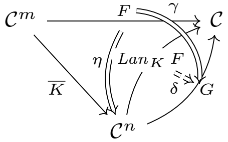
Completion Choices

- At the level of objects, this gives (at least) the syntactic data elements for GADTs.
- But what about morphisms? What about natural transformations?
- There are two obvious choices:

  - The discrete category $|\mathcal{C}|$ — equivalently, the discrete category $\mathcal{I}$ of interpretations of types in $\mathcal{C}$.



  - The full category $\mathcal{C}$.



Simplifying Assumptions

- All of the type arguments of our GADT are treated uniformly.
- All of that GADT's data constructors are treated uniformly.
- So we assume for now that a GADT of interest takes exactly one type argument (so $m = n = 1$) and has exactly one data constructor.
- That is, we assume our GADT has the form

$$\begin{aligned}&\mathsf{data\ G\ \ : Set \to Set\ where}\\ &\quad \mathsf{c} : \forall\{\mathsf{A} : \mathsf{Set}\} \to \mathsf{F\,A} \to \mathsf{G\,(K\,A)}\end{aligned}$$

- Then the interpretation G of G is μH, where $H\,J = Lan_K\,F$, i.e.,

$$G \;\cong\; \mu J.\,Lan_K\,F$$

- But how do we compute left Kan extensions, and thus semantics of GADTs?

Simplifying Assumptions

- All of the type arguments of our GADT are treated uniformly.
- All of that GADT's data constructors are treated uniformly.
- So we assume for now that a GADT of interest takes exactly one type argument (so $m = n = 1$) and has exactly one data constructor.
- That is, we assume our GADT has the form

  data G : Set → Set where
    c : ∀{A : Set} → F A → G (K A)

- Then the interpretation G of G is μH, where $H\,J = Lan_K\,F$, i.e.,

$$G \;\cong\; \mu J.\, Lan_K\, F$$

- But how do we compute left Kan extensions, and thus semantics of GADTs?

Simplifying Assumptions

- All of the type arguments of our GADT are treated uniformly.
- All of that GADT's data constructors are treated uniformly.
- So we assume for now that a GADT of interest takes exactly one type argument (so $m = n = 1$) and has exactly one data constructor.
- That is, we assume our GADT has the form

$$\begin{array}{l}\mathsf{data\ G\ \ : Set \to Set\ where} \\ \quad \mathsf{c : \forall\{A : Set\} \to F\,A \to G\,(K\,A)}\end{array}$$

- Then the interpretation G of G is μH, where $H\,J = Lan_K\,F$, i.e.,

$$G \;\cong\; \mu J.\, Lan_K\, F$$

- But how do we compute left Kan extensions, and thus semantics of GADTs?

Simplifying Assumptions

- All of the type arguments of our GADT are treated uniformly.
- All of that GADT's data constructors are treated uniformly.
- So we assume for now that a GADT of interest takes exactly one type argument (so $m = n = 1$) and has exactly one data constructor.
- That is, we assume our GADT has the form

$$\begin{array}{l} \mathsf{data\ G\ \ : Set \to Set\ where} \\ \quad \mathsf{c : \forall\{A : Set\} \to F\,A \to G\,(K\,A)} \end{array}$$

- Then the interpretation G of G is μH, where $H\,J = Lan_K\,F$, i.e.,

$$G \;\cong\; \mu J.\, Lan_K\, F$$

- But how do we compute left Kan extensions, and thus semantics of GADTs?

Simplifying Assumptions

- All of the type arguments of our GADT are treated uniformly.
- All of that GADT's data constructors are treated uniformly.
- So we assume for now that a GADT of interest takes exactly one type argument (so $m = n = 1$) and has exactly one data constructor.
- That is, we assume our GADT has the form

$$\begin{array}{l} \mathsf{data\ G\ \ : Set \to Set\ where} \\ \quad \mathsf{c : \forall\{A : Set\} \to F\,A \to G\,(K\,A)} \end{array}$$

- Then the interpretation G of G is μH, where $H\,J = Lan_K\,F$, i.e.,

$$G \;\cong\; \mu J.\, Lan_K\, F$$

- But how do we compute left Kan extensions, and thus semantics of GADTs?

Simplifying Assumptions

- All of the type arguments of our GADT are treated uniformly.
- All of that GADT's data constructors are treated uniformly.
- So we assume for now that a GADT of interest takes exactly one type argument (so $m = n = 1$) and has exactly one data constructor.
- That is, we assume our GADT has the form

```
data G  : Set → Set where
   c : ∀{A : Set} → F A → G (K A)
```

- Then the interpretation G of G is μH, where $H\,J = Lan_K\,F$, i.e.,

$$G \;\cong\; \mu J.\, Lan_K\, F$$

- But how do we compute left Kan extensions, and thus semantics of GADTs?

Computing Left Kan Extensions

- Under the same conditions needed to compute fixpoints of functors using the TFCA, we can compute left Kan extensions using the following well-known colimit formula:

  If $\mathcal{C}$ is locally λ-presentable and F and K are λ-cocontinuous functors on $\mathcal{C}$, then The left Kan extension of F along K can be computed as the colimit

  $$(Lan_K\, F)\, X \;=\; \varinjlim_{(A:\mathcal{C}_0,\, f:KA \to X)} FA$$

- $\mathcal{C}_0$ is a set of objects in $\mathcal{C}$ from which all others can be generated by colimits.
- The idea is that, under these conditions, the "large" colimit that is a left Kan extension can actually be computed as a colimit over a "small" set of support.

Computing Left Kan Extensions

- Under the same conditions needed to compute fixpoints of functors using the TFCA, we can compute left Kan extensions using the following well-known colimit formula:

  If $\mathcal{C}$ is locally λ-presentable and F and K are λ-cocontinuous functors on $\mathcal{C}$, then The left Kan extension of F along K can be computed as the colimit

$$(Lan_K\, F)\, X \;=\; \varinjlim_{(A:\mathcal{C}_0,\, f:KA\to X)} FA$$

- $\mathcal{C}_0$ is a set of objects in $\mathcal{C}$ from which all others can be generated by colimits.
- The idea is that, under these conditions, the "large" colimit that is a left Kan extension can actually be computed as a colimit over a "small" set of support.

Computing Left Kan Extensions

- Under the same conditions needed to compute fixpoints of functors using the TFCA, we can compute left Kan extensions using the following well-known colimit formula:

  If $\mathcal{C}$ is locally λ-presentable and F and K are λ-cocontinuous functors on $\mathcal{C}$, then The left Kan extension of F along K can be computed as the colimit

$$(Lan_K\, F)\, X \;=\; \varinjlim_{(A:\mathcal{C}_0,\, f:K A \to X)} F A$$

- $\mathcal{C}_0$ is a set of objects in $\mathcal{C}$ from which all others can be generated by colimits.
- The idea is that, under these conditions, the "large" colimit that is a left Kan extension can actually be computed as a colimit over a "small" set of support.

Computing Left Kan Extensions

- Under the same conditions needed to compute fixpoints of functors using the TFCA, we can compute left Kan extensions using the following well-known colimit formula:

  If $\mathcal{C}$ is locally λ-presentable and F and K are λ-cocontinuous functors on $\mathcal{C}$, then
  The left Kan extension of F along K can be computed as the colimit

$$(Lan_K\, F)\, X \;=\; \varinjlim_{(A:\mathcal{C}_0,\; f:KA \to X)} F A$$

- $\mathcal{C}_0$ is a set of objects in $\mathcal{C}$ from which all others can be generated by colimits.
- The idea is that, under these conditions, the "large" colimit that is a left Kan extension can actually be computed as a colimit over a "small" set of support.

Computing Discrete GADT Interpretations

- In Set:

$$(Lan_K\, F)\, X \;=\; \varinjlim_{A:\mathcal{I},\, f:KA\to X} FA \;=\; \Big(\bigcup_{A:\mathcal{I},\, f:KA\to X} FA\Big)/\sim$$

- Elements of the union are triples $(A : \mathcal{I}, f : KA \to X, y : FA)$ and $\sim$ is the smallest equivalence relation generated by

  $(A, f, y) \sim (A', f', y')$ iff $\exists h : A \to A'$ such that $f = Kh \circ f'$ and $y' = Fhy$

- In the discrete setting, $f = f' = h = id$, so $A = A'$ and $y = y'$, so

$$(Lan_K\, F)\, X \quad = \quad \bigcup_{A:\mathcal{I},\, KA=X} FA$$

  i.e.,

$$(Lan_K\, F)\, (KA) \quad = \quad \{y : FA \mid A : \mathcal{I}\}$$

- GADT syntax gives an element $c\, y$ of $G\, (KA)$ for every $y : FA$.
- Moreover, every element in $G\, (KA)$ is obtained in this way, and instances of G not of the form KA are not inhabited.
- Discrete GADT interpretations contain exactly that data constructed from syntax.

Computing Discrete GADT Interpretations

- In Set:

$$(Lan_K\, F)\, X \;=\; \varinjlim_{A:\mathcal{I},\, f:KA\to X} FA \;=\; \Big(\bigcup_{A:\mathcal{I},\, f:KA\to X} FA \Big) / \sim$$

- Elements of the union are triples $(A : \mathcal{I}, f : KA \to X, y : FA)$ and $\sim$ is the smallest equivalence relation generated by

$$(A, f, y) \sim (A', f', y') \text{ iff } \exists h : A \to A' \text{ such that } f = Kh \circ f' \text{ and } y' = Fhy$$

$$\begin{array}{ccc} KA & \xrightarrow{\;Kh\;} & KA' \\ & {\scriptstyle f}\searrow \quad \swarrow{\scriptstyle f'} & \\ & X & \end{array}$$

- In the discrete setting, $f = f' = h = id$, so $A = A'$ and $y = y'$, so

$$(Lan_K\, F)\, X \quad = \quad \bigcup_{A:\mathcal{I},\, KA=X} FA$$

  i.e.,

$$(Lan_K\, F)\, (KA) \quad = \quad \{y : FA \mid A : \mathcal{I}\}$$

- GADT syntax gives an element $c\,y$ of $G\,(KA)$ for every $y : FA$.
- Moreover, every element in $G\,(KA)$ is obtained in this way, and instances of G not of the form KA are not inhabited.
- Discrete GADT interpretations contain exactly that data constructed from syntax.

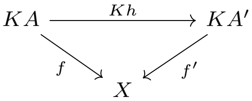
Computing Discrete GADT Interpretations

- In Set:

$$(Lan_K\, F)\, X \;=\; \varinjlim_{A:\mathcal{I},\, f:KA\to X} FA \;=\; \Big(\bigcup_{A:\mathcal{I},\, f:KA\to X} FA\Big)/\sim$$

- Elements of the union are triples $(A : \mathcal{I},\, f : KA \to X,\, y : FA)$ and $\sim$ is the smallest equivalence relation generated by

$$(A, f, y) \sim (A', f', y') \text{ iff } \exists h : A \to A' \text{ such that } f = Kh \circ f' \text{ and } y' = Fhy$$



- In the discrete setting, $f = f' = h = id$, so $A = A'$ and $y = y'$, so

$$(Lan_K\, F)\, X \quad = \quad \bigcup_{A:\mathcal{I},\, KA=X} FA$$

i.e.,

$$(Lan_K\, F)\, (KA) \quad = \quad \{y : FA \mid A : \mathcal{I}\}$$

- GADT syntax gives an element $c\,y$ of $G\,(KA)$ for every $y : FA$.
- Moreover, every element in $G\,(KA)$ is obtained in this way, and instances of G not of the form KA are not inhabited.
- Discrete GADT interpretations contain exactly that data constructed from syntax.

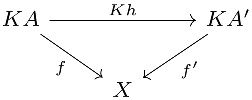
Computing Discrete GADT Interpretations

- In Set:

$$(Lan_K\, F)\, X \;=\; \varinjlim_{A:\mathcal{I},\, f:KA\to X} FA \;=\; \Big(\bigcup_{A:\mathcal{I},\, f:KA\to X} FA\Big)/\sim$$

- Elements of the union are triples $(A : \mathcal{I}, f : KA \to X, y : FA)$ and $\sim$ is the smallest equivalence relation generated by

$(A, f, y) \sim (A', f', y')$ iff $\exists h : A \to A'$ such that $f = Kh \circ f'$ and $y' = Fhy$



- In the discrete setting, $f = f' = h = id$, so $A = A'$ and $y = y'$, so

$$(Lan_K\, F)\, X \quad = \quad \bigcup_{A:\mathcal{I},\, KA=X} FA$$

i.e.,

$$(Lan_K\, F)\,(KA) \quad = \quad \{y : FA \mid A : \mathcal{I}\}$$

- GADT syntax gives an element $c\,y$ of $G\,(KA)$ for every $y : FA$.
- Moreover, every element in $G\,(KA)$ is obtained in this way, and instances of G not of the form KA are not inhabited.
- Discrete GADT interpretations contain exactly that data constructed from syntax.

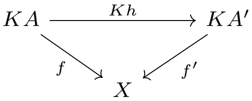
Computing Discrete GADT Interpretations

- In Set:

$$(Lan_K\, F)\, X \;=\; \varinjlim_{A:\mathcal{I},\, f:KA\to X} FA \;=\; \Big(\bigcup_{A:\mathcal{I},\, f:KA\to X} FA\Big)/\sim$$

- Elements of the union are triples $(A : \mathcal{I}, f : KA \to X,\, y : FA)$ and $\sim$ is the smallest equivalence relation generated by

$(A, f, y) \sim (A', f', y')$ iff $\exists h : A \to A'$ such that $f = Kh \circ f'$ and $y' = Fhy$



- In the discrete setting, $f = f' = h = id$, so $A = A'$ and $y = y'$, so

$$(Lan_K\, F)\, X \quad = \quad \bigcup_{A:\mathcal{I},\, KA=X} FA$$

i.e.,

$$(Lan_K\, F)\, (KA) \quad = \quad \{y : FA \mid A : \mathcal{I}\}$$

- GADT syntax gives an element $c\,y$ of $G\,(KA)$ for every $y : FA$.
- Moreover, every element in $G\,(KA)$ is obtained in this way, and instances of G not of the form KA are not inhabited.
- Discrete GADT interpretations contain exactly that data constructed from syntax.

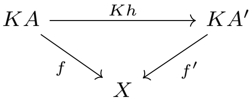
Computing Discrete GADT Interpretations

- In Set:

$$(Lan_K\, F)\, X \;=\; \varinjlim_{A:\mathcal{I},\, f:KA\to X} FA \;=\; \Big(\bigcup_{A:\mathcal{I},\, f:KA\to X} FA\Big)/\sim$$

- Elements of the union are triples $(A : \mathcal{I}, f : KA \to X, y : FA)$ and $\sim$ is the smallest equivalence relation generated by

$(A, f, y) \sim (A', f', y')$ iff $\exists h : A \to A'$ such that $f = Kh \circ f'$ and $y' = Fhy$



- In the discrete setting, $f = f' = h = id$, so $A = A'$ and $y = y'$, so

$$(Lan_K\, F)\, X \quad = \quad \bigcup_{A:\mathcal{I},\, KA=X} FA$$

i.e.,

$$(Lan_K\, F)\, (KA) \quad = \quad \{y : FA \mid A : \mathcal{I}\}$$

- GADT syntax gives an element $c\, y$ of $G\,(KA)$ for every $y : FA$.
- Moreover, every element in $G\,(KA)$ is obtained in this way, and instances of G not of the form KA are not inhabited.
- Discrete GADT interpretations contain exactly that data constructed from syntax.

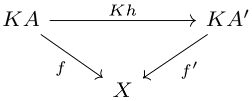
Computing Discrete GADT Interpretations

- In Set:

$$(Lan_K\, F)\, X \;=\; \varinjlim_{A:\mathcal{I},\, f:KA\to X} FA \;=\; \Big(\bigcup_{A:\mathcal{I},\, f:KA\to X} FA\Big)/\sim$$

- Elements of the union are triples $(A:\mathcal{I}, f : KA \to X,\, y : FA)$ and $\sim$ is the smallest equivalence relation generated by

$(A, f, y) \sim (A', f', y')$ iff $\exists h : A \to A'$ such that $f = Kh \circ f'$ and $y' = Fhy$



- In the discrete setting, $f = f' = h = id$, so $A = A'$ and $y = y'$, so

$$(Lan_K\, F)\, X \;=\; \bigcup_{A:\mathcal{I},\, KA=X} FA$$

i.e.,

$$(Lan_K\, F)\, (KA) \;=\; \{y : FA \mid A : \mathcal{I}\}$$

- GADT syntax gives an element $c\, y$ of $G\,(KA)$ for every $y : FA$.
- Moreover, every element in $G\,(KA)$ is obtained in this way, and instances of G not of the form KA are not inhabited.
- Discrete GADT interpretations contain exactly that data constructed from syntax.

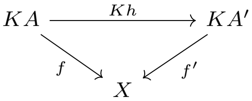
Computing Discrete GADT Interpretations

- In Set:

$$(Lan_K\, F)\, X \;=\; \varinjlim_{A:\mathcal{I},\, f:KA\to X} FA \;=\; \Big(\bigcup_{A:\mathcal{I},\, f:KA\to X} FA\Big)/\sim$$

- Elements of the union are triples $(A:\mathcal{I}, f:KA\to X, y:FA)$ and $\sim$ is the smallest equivalence relation generated by

$(A,f,y)\sim(A',f',y')$ iff $\exists h: A\to A'$ such that $f = Kh\circ f'$ and $y' = Fhy$



- In the discrete setting, $f=f'=h=id$, so $A=A'$ and $y=y'$, so

$$(Lan_K\, F)\, X \quad = \quad \bigcup_{A:\mathcal{I},\, KA=X} FA$$

i.e.,

$$(Lan_K\, F)\,(KA) \quad = \quad \{y : FA \mid A:\mathcal{I}\}$$

- GADT syntax gives an element $c\,y$ of $G\,(KA)$ for every $y:FA$.
- Moreover, every element in $G\,(KA)$ is obtained in this way, and instances of G not of the form KA are not inhabited.
- Discrete GADT interpretations contain exactly that data constructed from syntax.

Something Amiss?

- This clearly "works".
- But ADTs and nested types don't need to invoke discreteness to get functoriality.
- So something seems amiss.
- Question: Can we see GADTs as fixpoints of non-discrete functors? That is, can we see GADTs as data types in the "normal", container-y sense, with proper map functions?

Something Amiss?

- This clearly “works”.
- But ADTs and nested types don’t need to invoke discreteness to get functoriality.
- So something seems amiss.
- Question: Can we see GADTs as fixpoints of non-discrete functors? That is, can we see GADTs as data types in the “normal”, container-y sense, with proper map functions?

Something Amiss?

- This clearly "works".
- But ADTs and nested types don't need to invoke discreteness to get functoriality.
- So something seems amiss.
- Question: Can we see GADTs as fixpoints of non-discrete functors? That is, can we see GADTs as data types in the "normal", container-y sense, with proper map functions?

Something Amiss?

- This clearly "works".
- But ADTs and nested types don't need to invoke discreteness to get functoriality.
- So something seems amiss.
- Question: Can we see GADTs as fixpoints of non-discrete functors? That is, can we see GADTs as data types in the "normal", container-y sense, with proper map functions?

Computing Fully Functorial Interpretations of GADTs

- In Set:

$$(Lan_K\, F)\, X \;=\; \varinjlim_{A:\mathcal{C}_0,\, f:KA\to X} FA \;=\; \Big(\bigcup_{A:\mathcal{C}_0,\, f:KA\to X} FA\Big)/\sim$$

- It is now harder to compute and mod out by the equivalence generated by

  $(A, f, y) \sim (A', f', y')$ iff $\exists h : A \to A'$ such that $f = Kh \circ f'$ and $y' = Fhy$

- Need restrictions on syntax to ensure functoriality of interpretation G of G:
  - Assume GADTs are (hereditarily) polynomial
  - Require strict positivity
  - No truly nested GADTs (no nested Gs in constructor domains or codomains)
- If F and K are higher-order functors then so is $Lan_K\, F$. So $G = \mu J.Lan_K\, F$ is a functor and thus has an associated function map_G.
- Each triple $(A : \mathcal{C}_0, f : KA \to X, y : FA)$ gives an element $map_G\, f\, (c\, y)$ of GX.
- This is cannot possibly be the interpretation of any term constructed from G's syntax unless $X = KB$ for some B.

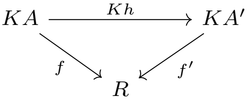
Computing Fully Functorial Interpretations of GADTs

- In Set:

$$(Lan_K\, F)\, X \;=\; \varinjlim_{A:\mathcal{C}_0,\; f:KA\to X} FA \;=\; \Big(\bigcup_{A:\mathcal{C}_0,\; f:KA\to X} FA\Big)/\sim$$

- It is now harder to compute and mod out by the equivalence generated by

$(A, f, y) \sim (A', f', y')$ iff $\exists h : A \to A'$ such that $f = Kh \circ f'$ and $y' = Fhy$



- Need restrictions on syntax to ensure functoriality of interpretation G of G:
  - Assume GADTs are (hereditarily) polynomial
  - Require strict positivity
  - No truly nested GADTs (no nested Gs in constructor domains or codomains)
- If F and K are higher-order functors then so is $Lan_K\, F$. So $G = \mu J.Lan_K\, F$ is a functor and thus has an associated function map_G.
- Each triple $(A : \mathcal{C}_0, f : KA \to X, y : FA)$ gives an element $map_G\, f\, (c\, y)$ of GX.
- This is cannot possibly be the interpretation of any term constructed from G's syntax unless $X = KB$ for some B.

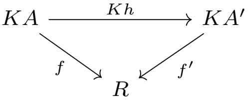
Computing Fully Functorial Interpretations of GADTs

- In Set:

$$(Lan_K\, F)\, X \;=\; \varinjlim_{A:\mathcal{C}_0,\, f:KA\to X} FA \;=\; \Big(\bigcup_{A:\mathcal{C}_0,\, f:KA\to X} FA\Big)/\sim$$

- It is now harder to compute and mod out by the equivalence generated by

$(A, f, y) \sim (A', f', y')$ iff $\exists h : A \to A'$ such that $f = Kh \circ f'$ and $y' = Fhy$



- Need restrictions on syntax to ensure functoriality of interpretation G of G:
  - Assume GADTs are (hereditarily) polynomial
  - Require strict positivity
  - No truly nested GADTs (no nested Gs in constructor domains or codomains)
- If F and K are higher-order functors then so is $Lan_K\, F$. So $G = \mu J.Lan_K\, F$ is a functor and thus has an associated function map_G.
- Each triple $(A : \mathcal{C}_0, f : KA \to X, y : FA)$ gives an element $map_G\, f\,(c\, y)$ of GX.
- This is cannot possibly be the interpretation of any term constructed from G's syntax unless $X = KB$ for some B.

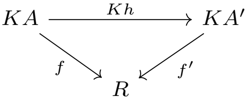
Computing Fully Functorial Interpretations of GADTs

- In Set:

$$(Lan_K\, F)\, X \;=\; \varinjlim_{A:\mathcal{C}_0,\, f:KA\to X} FA \;=\; \Big(\bigcup_{A:\mathcal{C}_0,\, f:KA\to X} FA\Big)/\sim$$

- It is now harder to compute and mod out by the equivalence generated by

$(A, f, y) \sim (A', f', y')$ iff $\exists h : A \to A'$ such that $f = Kh \circ f'$ and $y' = Fhy$



- Need restrictions on syntax to ensure functoriality of interpretation G of G:
  - Assume GADTs are (hereditarily) polynomial
  - Require strict positivity
  - No truly nested GADTs (no nested Gs in constructor domains or codomains)
- If F and K are higher-order functors then so is $Lan_K\, F$. So $G = \mu J.Lan_K\, F$ is a functor and thus has an associated function map_G.
- Each triple $(A : \mathcal{C}_0, f : KA \to X, y : FA)$ gives an element $map_G\, f\, (c\, y)$ of GX.
- This is cannot possibly be the interpretation of any term constructed from G's syntax unless $X = KB$ for some B.

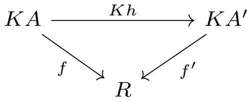
Computing Fully Functorial Interpretations of GADTs

- In Set:

$$(Lan_K\, F)\, X \;=\; \varinjlim_{A:\mathcal{C}_0,\; f:KA\to X} FA \;=\; \Big(\bigcup_{A:\mathcal{C}_0,\; f:KA\to X} FA\Big)/\sim$$

- It is now harder to compute and mod out by the equivalence generated by

$$(A, f, y) \sim (A', f', y') \text{ iff } \exists h : A \to A' \text{ such that } f = Kh \circ f' \text{ and } y' = Fhy$$



- Need restrictions on syntax to ensure functoriality of interpretation G of G:
  - Assume GADTs are (hereditarily) polynomial
  - Require strict positivity
  - No truly nested GADTs (no nested Gs in constructor domains or codomains)
- If F and K are higher-order functors then so is $Lan_K\, F$. So $G = \mu J.Lan_K\, F$ is a functor and thus has an associated function map_G.
- Each triple $(A : \mathcal{C}_0, f : KA \to X, y : FA)$ gives an element $map_G\, f\, (c\, y)$ of GX.
- This is cannot possibly be the interpretation of any term constructed from G's syntax unless $X = KB$ for some B.

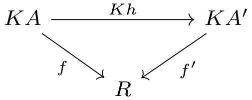
Computing Fully Functorial Interpretations of GADTs

- In Set:

$$(Lan_K\, F)\, X \;=\; \varinjlim_{A:\mathcal{C}_0,\, f:KA\to X} FA \;=\; \Big(\bigcup_{A:\mathcal{C}_0,\, f:KA\to X} FA\Big)/\sim$$

- It is now harder to compute and mod out by the equivalence generated by

$(A, f, y) \sim (A', f', y')$ iff $\exists h : A \to A'$ such that $f = Kh \circ f'$ and $y' = Fhy$



- Need restrictions on syntax to ensure functoriality of interpretation G of G:
  - Assume GADTs are (hereditarily) polynomial
  - Require strict positivity
  - No truly nested GADTs (no nested Gs in constructor domains or codomains)
- If F and K are higher-order functors then so is $Lan_K\, F$. So $G = \mu J.Lan_K\, F$ is a functor and thus has an associated function map_G.
- Each triple $(A : \mathcal{C}_0, f : KA \to X, y : FA)$ gives an element $map_G\, f\,(c\, y)$ of GX.
- This is cannot possibly be the interpretation of any term constructed from G's syntax unless $X = KB$ for some B.

The Functorial Interpretation of Seq

- We interpret spair : $\forall\{\mathsf{A}\,\mathsf{B} : \mathsf{Set}\} \to \mathsf{Seq}\,\mathsf{A} \to \mathsf{Seq}\,\mathsf{B} \to \mathsf{Seq}(\mathsf{A} \times \mathsf{B})$ as a morphism

$$\varinjlim_{(A,B):\mathcal{C}_0 \times \mathcal{C}_0,\, f:A\times B \to X} Seq\,A \times Seq\,B \;\to\; Seq\,X$$

- It is no coincidence that the morphism that $f : A \times B \to X$ that was missing from Seq's map, and thus motivated its discrete semantics, appears in this colimit!
- Thus

$$map_{Seq}\,f\;(spair\;t_1\;t_2)$$

  is in $Seq\,X$ but is not the interpretation of any term constructed from Seq's syntax.
- The properly functorial interpretation of Seq thus contains data elements not constructed from its syntax!
- For ADTs and nested types, map-closure adds no new data elements because the interpretation of the data type is already a proper functor.

  (Technically: $K = Id$, and a left Kan extension along an identity is an identity, i.e., $Lan_{Id}F = F$.)

The Functorial Interpretation of Seq

- We interpret spair : $\forall\{\mathsf{A}\,\mathsf{B} : \mathsf{Set}\} \to \mathsf{Seq}\,\mathsf{A} \to \mathsf{Seq}\,\mathsf{B} \to \mathsf{Seq}(\mathsf{A} \times \mathsf{B})$ as a morphism

$$\varinjlim_{(A,B):\mathcal{C}_0\times\mathcal{C}_0,\, f:A\times B\to X} Seq\,A \times Seq\,B \;\to\; Seq\,X$$

- It is no coincidence that the morphism that $f : A \times B \to X$ that was missing from Seq's map, and thus motivated its discrete semantics, appears in this colimit!
- Thus

$$map_{Seq}\,f\;(spair\;t_1\;t_2)$$

is in $Seq\,X$ but is not the interpretation of any term constructed from Seq's syntax.
- The properly functorial interpretation of Seq thus contains data elements not constructed from its syntax!
- For ADTs and nested types, map-closure adds no new data elements because the interpretation of the data type is already a proper functor.

(Technically: $K = Id$, and a left Kan extension along an identity is an identity, i.e., $Lan_{Id}F = F$.)

The Functorial Interpretation of Seq

- We interpret spair : ∀{A B : Set} → Seq A → Seq B → Seq(A × B) as a morphism

$$\varinjlim_{(A,B):\mathcal{C}_0\times\mathcal{C}_0,\, f:A\times B\to X} Seq\,A \times Seq\,B \;\to\; Seq\,X$$

- It is no coincidence that the morphism that $f : A \times B \to X$ that was missing from Seq's map, and thus motivated its discrete semantics, appears in this colimit!
- Thus

$$map_{Seq}\, f\; (spair\; t_1\; t_2)$$

  is in $Seq\,X$ but is not the interpretation of any term constructed from Seq's syntax.
- The properly functorial interpretation of Seq thus contains data elements not constructed from its syntax!
- For ADTs and nested types, map-closure adds no new data elements because the interpretation of the data type is already a proper functor.

  (Technically: $K = Id$, and a left Kan extension along an identity is an identity, i.e., $Lan_{Id}F = F$.)

The Functorial Interpretation of Seq

- We interpret spair : $\forall\{\mathsf{A}\,\mathsf{B} : \mathsf{Set}\} \to \mathsf{Seq}\,\mathsf{A} \to \mathsf{Seq}\,\mathsf{B} \to \mathsf{Seq}(\mathsf{A} \times \mathsf{B})$ as a morphism

$$\varinjlim_{(A,B):\mathcal{C}_0 \times \mathcal{C}_0,\, f:A \times B \to X} Seq\,A \times Seq\,B \;\to\; Seq\,X$$

- It is no coincidence that the morphism that $f : A \times B \to X$ that was missing from Seq's map, and thus motivated its discrete semantics, appears in this colimit!
- Thus

$$map_{Seq}\,f\;(spair\;t_1\;t_2)$$

  is in $Seq\,X$ but is not the interpretation of any term constructed from Seq's syntax.
- The properly functorial interpretation of Seq thus contains data elements not constructed from its syntax!
- For ADTs and nested types, map-closure adds no new data elements because the interpretation of the data type is already a proper functor.

  (Technically: $K = Id$, and a left Kan extension along an identity is an identity, i.e., $Lan_{Id}F = F$.)

The Functorial Interpretation of Seq

- We interpret spair : $\forall \{A\,B : \mathsf{Set}\} \to \mathsf{Seq}\,A \to \mathsf{Seq}\,B \to \mathsf{Seq}(A \times B)$ as a morphism

$$\varinjlim_{(A,B):\mathcal{C}_0 \times \mathcal{C}_0,\ f:A\times B\to X} Seq\,A \times Seq\,B \ \to \ Seq\,X$$

- It is no coincidence that the morphism that $f : A \times B \to X$ that was missing from Seq's map, and thus motivated its discrete semantics, appears in this colimit!
- Thus

$$map_{Seq}\,f\;(spair\;t_1\;t_2)$$

  is in $Seq\,X$ but is not the interpretation of any term constructed from Seq's syntax.
- The properly functorial interpretation of Seq thus contains data elements not constructed from its syntax!
- For ADTs and nested types, map-closure adds no new data elements because the interpretation of the data type is already a proper functor.

  (Technically: $K = Id$, and a left Kan extension along an identity is an identity, i.e., $Lan_{Id}F = F$.)

Summary

- We have seen that the discrete and fully functorial interpretations of GADTs can be very different.
- This differs from the discrete and functorial interpretations of ADTs and nested types, which always contain exactly the same data elements.
- For ADTs and nested types there is only one natural semantics. For GADTs there are two, and they don't even contain the same data elements.

  (By design, they don't contain the same morphisms.)
- Question: What practical difference could the difference in semantics possibly have? We'll see next time.

Summary

- We have seen that the discrete and fully functorial interpretations of GADTs can be very different.
- This differs from the discrete and functorial interpretations of ADTs and nested types, which always contain exactly the same data elements.
- For ADTs and nested types there is only one natural semantics. For GADTs there are two, and they don't even contain the same data elements.

  (By design, they don't contain the same morphisms.)
- Question: What practical difference could the difference in semantics possibly have? We'll see next time.

Summary

- We have seen that the discrete and fully functorial interpretations of GADTs can be very different.
- This differs from the discrete and functorial interpretations of ADTs and nested types, which always contain exactly the same data elements.
- For ADTs and nested types there is only one natural semantics. For GADTs there are two, and they don't even contain the same data elements.

  (By design, they don't contain the same morphisms.)
- Question: What practical difference could the difference in semantics possibly have? We'll see next time.

Summary

- We have seen that the discrete and fully functorial interpretations of GADTs can be very different.
- This differs from the discrete and functorial interpretations of ADTs and nested types, which always contain exactly the same data elements.
- For ADTs and nested types there is only one natural semantics. For GADTs there are two, and they don't even contain the same data elements.

  (By design, they don't contain the same morphisms.)
- Question: What practical difference could the difference in semantics possibly have? We'll see next time.

Summary

- We have seen that the discrete and fully functorial interpretations of GADTs can be very different.
- This differs from the discrete and functorial interpretations of ADTs and nested types, which always contain exactly the same data elements.
- For ADTs and nested types there is only one natural semantics. For GADTs there are two, and they don't even contain the same data elements.

  (By design, they don't contain the same morphisms.)
- Question: What practical difference could the difference in semantics possibly have? We'll see next time.