Semantics of Advanced Data Types

Patricia Johann
Appalachian State University

June 14, 2021

Why Study Semantics of Advanced Data Types?

- A language's type system allows us to express correctness properties of its programs. ("Well-typed programs don't go wrong.")
- Data types are an important part of any type system.
- Fancier data types let us express more sophisticated correctness properties.
- Type-checking provides guarantees of correctness with respect to these properties.

- In this course:

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

- We ask (and answer!)
  - What correctness properties can each class of data types express?
  - What models can we build to understand each class of data types?
  - What do properties of these models say about how we can compute with, and reason about, programs involving each class of data types?

Why Study Semantics of Advanced Data Types?

- A language's type system allows us to express correctness properties of its programs. ("Well-typed programs don't go wrong.")
- Data types are an important part of any type system.
- Fancier data types let us express more sophisticated correctness properties.
- Type-checking provides guarantees of correctness with respect to these properties.

- In this course:

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

- We ask (and answer!)
  - What correctness properties can each class of data types express?
  - What models can we build to understand each class of data types?
  - What do properties of these models say about how we can compute with, and reason about, programs involving each class of data types?

Why Study Semantics of Advanced Data Types?

- A language's type system allows us to express correctness properties of its programs. ("Well-typed programs don't go wrong.")
- Data types are an important part of any type system.
- Fancier data types let us express more sophisticated correctness properties.
- Type-checking provides guarantees of correctness with respect to these properties.
- In this course:

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

- We ask (and answer!)
  - What correctness properties can each class of data types express?
  - What models can we build to understand each class of data types?
  - What do properties of these models say about how we can compute with, and reason about, programs involving each class of data types?

Why Study Semantics of Advanced Data Types?

- A language's type system allows us to express correctness properties of its programs. ("Well-typed programs don't go wrong.")
- Data types are an important part of any type system.
- Fancier data types let us express more sophisticated correctness properties.
- Type-checking provides guarantees of correctness with respect to these properties.

- In this course:

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

- We ask (and answer!)
  - What correctness properties can each class of data types express?
  - What models can we build to understand each class of data types?
  - What do properties of these models say about how we can compute with, and reason about, programs involving each class of data types?

Why Study Semantics of Advanced Data Types?

- A language's type system allows us to express correctness properties of its programs. ("Well-typed programs don't go wrong.")
- Data types are an important part of any type system.
- Fancier data types let us express more sophisticated correctness properties.
- Type-checking provides guarantees of correctness with respect to these properties.

- In this course:

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

- We ask (and answer!)
  - What correctness properties can each class of data types express?
  - What models can we build to understand each class of data types?
  - What do properties of these models say about how we can compute with, and reason about, programs involving each class of data types?

Why Study Semantics of Advanced Data Types?

- A language's type system allows us to express correctness properties of its programs. ("Well-typed programs don't go wrong.")
- Data types are an important part of any type system.
- Fancier data types let us express more sophisticated correctness properties.
- Type-checking provides guarantees of correctness with respect to these properties.

- In this course:

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

- We ask (and answer!)
  - What correctness properties can each class of data types express?
  - What models can we build to understand each class of data types?
  - What do properties of these models say about how we can compute with, and reason about, programs involving each class of data types?

Why Study Semantics of Advanced Data Types?

- A language's type system allows us to express correctness properties of its programs. ("Well-typed programs don't go wrong.")
- Data types are an important part of any type system.
- Fancier data types let us express more sophisticated correctness properties.
- Type-checking provides guarantees of correctness with respect to these properties.

- In this course:

$$\boxed{\text{ADTs}} \underset{\text{generalized by}}{\overset{\text{syntactically}}{\hookrightarrow}} \boxed{\text{nested types}} \underset{\text{generalized by}}{\overset{\text{syntactically}}{\hookrightarrow}} \boxed{\text{GADTs}}$$

- We ask (and answer!)
  - What correctness properties can each class of data types express?
  - What models can we build to understand each class of data types?
  - What do properties of these models say about how we can compute with, and reason about, programs involving each class of data types?

Why Study Semantics of Advanced Data Types?

- A language's type system allows us to express correctness properties of its programs. ("Well-typed programs don't go wrong.")
- Data types are an important part of any type system.
- Fancier data types let us express more sophisticated correctness properties.
- Type-checking provides guarantees of correctness with respect to these properties.

- In this course:

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

- We ask (and answer!)
  - What correctness properties can each class of data types express?
  - What models can we build to understand each class of data types?
  - What do properties of these models say about how we can compute with, and reason about, programs involving each class of data types?

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

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

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

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 1:
Syntax and Semantics of ADTs and Nested Types

$$\boxed{\text{ADTs}} \underset{\text{generalized by}}{\xhookrightarrow{\text{syntactically}}} \boxed{\text{nested types}} \underset{\text{generalized by}}{\xhookrightarrow{\text{syntactically}}} \boxed{\text{GADTs}}$$

Assumption: Basic familiarity with categories, functors, natural transformations.

Syntax of ADTs (I)

- Booleans

```
data Bool : Set where
  false : Bool
  true  : Bool
```

- Natural numbers

```
data Nat : Set where
  zero : Nat
  suc  : Nat → Nat
```

- Lists

```
data List (A : Set) : Set where
  []    : List A
  _::_  : A → List A → List A
```

- Binary trees

```
data Tree (A : Set) (B : Set) : Set where
  leaf : A → Tree A B
  node : Tree A B → B → Tree A B → Tree A B
```

Syntax of ADTs (I)

- Booleans

$$
\begin{array}{l}
\text{data Bool : Set where} \\
\quad \text{false} \;:\; \text{Bool} \\
\quad \text{true} \;\;:\; \text{Bool}
\end{array}
$$

- Natural numbers

$$
\begin{array}{l}
\text{data Nat : Set where} \\
\quad \text{zero} \;:\; \text{Nat} \\
\quad \text{suc} \;\;:\; \text{Nat} \to \text{Nat}
\end{array}
$$

- Lists

$$
\begin{array}{l}
\text{data List (A : Set) : Set where} \\
\quad [\,] \quad\;\; :\; \text{List A} \\
\quad \_ :: \_ \;:\; \text{A} \to \text{List A} \to \text{List A}
\end{array}
$$

- Binary trees

$$
\begin{array}{l}
\text{data Tree (A : Set) (B : Set) : Set where} \\
\quad \text{leaf} \;\;:\; \text{A} \to \text{Tree A B} \\
\quad \text{node} \;:\; \text{Tree A B} \to \text{B} \to \text{Tree A B} \to \text{Tree A B}
\end{array}
$$

Syntax of ADTs (I)

- Booleans

$$\begin{array}{l} \text{data Bool : Set where} \\ \quad \text{false} \;:\; \text{Bool} \\ \quad \text{true} \;\;:\; \text{Bool} \end{array}$$

- Natural numbers

$$\begin{array}{l} \text{data Nat : Set where} \\ \quad \text{zero} \;:\; \text{Nat} \\ \quad \text{suc} \;\;:\; \text{Nat} \to \text{Nat} \end{array}$$

- Lists

$$\begin{array}{l} \text{data List (A : Set) : Set where} \\ \quad [\,] \quad\;\; :\; \text{List A} \\ \quad \_ :: \_ \;:\; \text{A} \to \text{List A} \to \text{List A} \end{array}$$

- Binary trees

$$\begin{array}{l} \text{data Tree (A : Set) (B : Set) : Set where} \\ \quad \text{leaf} \;\;:\; \text{A} \to \text{Tree A B} \\ \quad \text{node} \;:\; \text{Tree A B} \to \text{B} \to \text{Tree A B} \to \text{Tree A B} \end{array}$$

Syntax of ADTs (I)

- Booleans

$$\begin{aligned}&\mathsf{data\ Bool} : \mathsf{Set\ where}\\ &\quad \mathsf{false} \;:\; \mathsf{Bool}\\ &\quad \mathsf{true} \;:\; \mathsf{Bool}\end{aligned}$$

- Natural numbers

$$\begin{aligned}&\mathsf{data\ Nat} : \mathsf{Set\ where}\\ &\quad \mathsf{zero} \;:\; \mathsf{Nat}\\ &\quad \mathsf{suc} \;:\; \mathsf{Nat} \to \mathsf{Nat}\end{aligned}$$

- Lists

$$\begin{aligned}&\mathsf{data\ List}\,(\mathsf{A} : \mathsf{Set}) : \mathsf{Set\ where}\\ &\quad [\,] \;:\; \mathsf{List\,A}\\ &\quad \_ :: \_ \;:\; \mathsf{A} \to \mathsf{List\,A} \to \mathsf{List\,A}\end{aligned}$$

- Binary trees

$$\begin{aligned}&\mathsf{data\ Tree}\,(\mathsf{A} : \mathsf{Set})\,(\mathsf{B} : \mathsf{Set}) : \mathsf{Set\ where}\\ &\quad \mathsf{leaf} \;:\; \mathsf{A} \to \mathsf{Tree\,A\,B}\\ &\quad \mathsf{node} \;:\; \mathsf{Tree\,A\,B} \to \mathsf{B} \to \mathsf{Tree\,A\,B} \to \mathsf{Tree\,A\,B}\end{aligned}$$

Syntax of ADTs (II)

- The only instance of the data type being defined that appears in the type of a constructor for that data type is the same one being defined
- So an ADT defines a family of inductive types, one for each choice of parameters.
- The general form of an ADT is

$$\begin{array}{l} \mathsf{data}\ D\ A_1 \ldots A_n : \mathsf{Set\ where} \\ \quad c_1 : T_{11} \to \ldots \to T_{1j_1} \to D\ A_1 \ldots A_n \\ \quad \ldots \\ \quad c_k : T_{k1} \to \ldots \to T_{kj_k} \to D\ A_1 \ldots A_n \end{array}$$

- Agda also imposes a strict positivity requirement on the types of $c_1,\ldots,c_k$: Either

  - T_{ij} is not inductive and does not mention D

  or

  - T_{ij} is inductive and has the form

$$C_1 \to \ldots \to C_p \to D\ A_1 \ldots A_n$$

  where D does not occur in any C_i.

- Strict positivity
  $\Longrightarrow$ no negative occurrences of D in the argument types of its constructors
  $\Longrightarrow$ D can be interpreted as the least fixpoint of a functor.

Syntax of ADTs (II)

- The only instance of the data type being defined that appears in the type of a constructor for that data type is the same one being defined
- So an ADT defines a family of inductive types, one for each choice of parameters.
- The general form of an ADT is

$$\begin{array}{l} \mathsf{data}\ \mathsf{D}\ \mathsf{A}_1 \ldots \mathsf{A}_n : \mathsf{Set}\ \mathsf{where} \\ \quad c_1 : T_{11} \to \ldots \to T_{1j_1} \to D\ A_1 \ldots A_n \\ \quad \ldots \\ \quad c_k : T_{k1} \to \ldots \to T_{kj_k} \to D\ A_1 \ldots A_n \end{array}$$

- Agda also imposes a strict positivity requirement on the types of $c_1,\ldots,c_k$: Either

  - T_{ij} is not inductive and does not mention D

  or

  - T_{ij} is inductive and has the form

$$C_1 \to \ldots \to C_p \to D\ A_1 \ldots A_n$$

  where D does not occur in any C_i.

- Strict positivity
  $\Longrightarrow$ no negative occurrences of D in the argument types of its constructors
  $\Longrightarrow$ D can be interpreted as the least fixpoint of a functor.

Syntax of ADTs (II)

- The only instance of the data type being defined that appears in the type of a constructor for that data type is the same one being defined
- So an ADT defines a family of inductive types, one for each choice of parameters.
- The general form of an ADT is

$$
\begin{array}{l}
\mathsf{data}\ D\, A_1 \ldots A_n : \mathsf{Set\ where} \\
\quad c_1 : T_{11} \to \ldots \to T_{1j_1} \to D\, A_1 \ldots A_n \\
\quad \ldots \\
\quad c_k : T_{k1} \to \ldots \to T_{kj_k} \to D\, A_1 \ldots A_n
\end{array}
$$

- Agda also imposes a strict positivity requirement on the types of $c_1,\ldots,c_k$: Either

  - T_{ij} is not inductive and does not mention D

  or

  - T_{ij} is inductive and has the form

$$C_1 \to \ldots \to C_p \to D\, A_1 \ldots A_n$$

  where D does not occur in any C_i.

- Strict positivity

  $\Longrightarrow$ no negative occurrences of D in the argument types of its constructors

  $\Longrightarrow$ D can be interpreted as the least fixpoint of a functor.

Syntax of ADTs (II)

- The only instance of the data type being defined that appears in the type of a constructor for that data type is the same one being defined
- So an ADT defines a family of inductive types, one for each choice of parameters.
- The general form of an ADT is

$$
\begin{array}{l}
\mathsf{data}\ D\ A_1 \ldots A_n : \mathsf{Set\ where} \\
\quad c_1 : T_{11} \to \ldots \to T_{1j_1} \to D\ A_1 \ldots A_n \\
\quad \ldots \\
\quad c_k : T_{k1} \to \ldots \to T_{kj_k} \to D\ A_1 \ldots A_n
\end{array}
$$

- Agda also imposes a strict positivity requirement on the types of $c_1, \ldots, c_k$: Either

  - T_{ij} is not inductive and does not mention D

  or

  - T_{ij} is inductive and has the form

$$C_1 \to \ldots \to C_p \to D\ A_1 \ldots A_n$$

  where D does not occur in any C_i.

- Strict positivity

  $\Longrightarrow$ no negative occurrences of D in the argument types of its constructors

  $\Longrightarrow$ D can be interpreted as the least fixpoint of a functor.

Syntax of ADTs (II)

- The only instance of the data type being defined that appears in the type of a constructor for that data type is the same one being defined
- So an ADT defines a family of inductive types, one for each choice of parameters.
- The general form of an ADT is

$$\begin{array}{l} \mathsf{data}\ D\ A_1 \ldots A_n : \mathsf{Set\ where} \\ \quad c_1 : T_{11} \to \ldots \to T_{1j_1} \to D\ A_1 \ldots A_n \\ \quad \ldots \\ \quad c_k : T_{k1} \to \ldots \to T_{kj_k} \to D\ A_1 \ldots A_n \end{array}$$

- Agda also imposes a strict positivity requirement on the types of $c_1, \ldots, c_k$: Either

  - T_{ij} is not inductive and does not mention D

  or

  - T_{ij} is inductive and has the form

$$C_1 \to \ldots \to C_p \to D\ A_1 \ldots A_n$$

  where D does not occur in any C_i.
- Strict positivity

  $\Longrightarrow$ no negative occurrences of D in the argument types of its constructors

  $\Longrightarrow$ D can be interpreted as the least fixpoint of a functor.

Category Theory Interlude (I)

- A category $\mathcal{C}$ comprises

  - a class $ob(\mathcal{C})$ of *objects*

  - for each $X, Y \in ob(\mathcal{C})$, a class $Hom_{\mathcal{C}}(X, Y)$ of *morphisms* from X to Y

  - for each $X \in ob(\mathcal{C})$, an *identity* morphism $id_X \in Hom_{\mathcal{C}}(X, X)$

  - a *composition* operator $\circ$ assigning to each pair of morphisms $f : X \to Y$ and $g : Y \to Z$, the composite morphism $g \circ f : X \to Z$

- The identity morphisms are expected to behave like identities: if $f : X \to Y$ then $f \circ id_X = f = id_Y \circ f$.

- Composition is associative.

- We write $X : \mathcal{C}$ rather than $X \in ob(\mathcal{C})$ and $f : X \to Y$ rather than $f \in Hom_{\mathcal{C}}(X, Y)$.

- We will restrict attention to the category Set for now.

Category Theory Interlude (I)

- A category $\mathcal{C}$ comprises
  - a class $ob(\mathcal{C})$ of *objects*
  - for each $X, Y \in ob(\mathcal{C})$, a class $Hom_{\mathcal{C}}(X, Y)$ of *morphisms* from X to Y
  - for each $X \in ob(\mathcal{C})$, an *identity* morphism $id_X \in Hom_{\mathcal{C}}(X, X)$
  - a *composition* operator $\circ$ assigning to each pair of morphisms $f : X \to Y$ and $g : Y \to Z$, the composite morphism $g \circ f : X \to Z$
- The identity morphisms are expected to behave like identities: if $f : X \to Y$ then $f \circ id_X = f = id_Y \circ f$.
- Composition is associative.
- We write $X : \mathcal{C}$ rather than $X \in ob(\mathcal{C})$ and $f : X \to Y$ rather than $f \in Hom_{\mathcal{C}}(X, Y)$.
- We will restrict attention to the category Set for now.

Category Theory Interlude (I)

- A category $\mathcal{C}$ comprises

  - a class $ob(\mathcal{C})$ of *objects*

  - for each $X, Y \in ob(\mathcal{C})$, a class $Hom_{\mathcal{C}}(X, Y)$ of *morphisms* from X to Y

  - for each $X \in ob(\mathcal{C})$, an *identity* morphism $id_X \in Hom_{\mathcal{C}}(X, X)$

  - a *composition* operator $\circ$ assigning to each pair of morphisms $f : X \to Y$ and $g : Y \to Z$, the composite morphism $g \circ f : X \to Z$

- The identity morphisms are expected to behave like identities: if $f : X \to Y$ then $f \circ id_X = f = id_Y \circ f$.

- Composition is associative.

- We write $X : \mathcal{C}$ rather than $X \in ob(\mathcal{C})$ and $f : X \to Y$ rather than $f \in Hom_{\mathcal{C}}(X, Y)$.

- We will restrict attention to the category Set for now.

Category Theory Interlude (I)

- A category $\mathcal{C}$ comprises

  - a class $ob(\mathcal{C})$ of *objects*

  - for each $X, Y \in ob(\mathcal{C})$, a class $Hom_{\mathcal{C}}(X, Y)$ of *morphisms* from X to Y

  - for each $X \in ob(\mathcal{C})$, an *identity* morphism $id_X \in Hom_{\mathcal{C}}(X, X)$

  - a *composition* operator $\circ$ assigning to each pair of morphisms $f : X \to Y$ and $g : Y \to Z$, the composite morphism $g \circ f : X \to Z$

- The identity morphisms are expected to behave like identities: if $f : X \to Y$ then $f \circ id_X = f = id_Y \circ f$.
- Composition is associative.
- We write $X : \mathcal{C}$ rather than $X \in ob(\mathcal{C})$ and $f : X \to Y$ rather than $f \in Hom_{\mathcal{C}}(X, Y)$.
- We will restrict attention to the category Set for now.

Category Theory Interlude (I)

- A category $\mathcal{C}$ comprises

  - a class $ob(\mathcal{C})$ of *objects*

  - for each $X, Y \in ob(\mathcal{C})$, a class $Hom_{\mathcal{C}}(X, Y)$ of *morphisms* from X to Y

  - for each $X \in ob(\mathcal{C})$, an *identity* morphism $id_X \in Hom_{\mathcal{C}}(X, X)$

  - a *composition* operator $\circ$ assigning to each pair of morphisms $f : X \to Y$ and $g : Y \to Z$, the composite morphism $g \circ f : X \to Z$
- The identity morphisms are expected to behave like identities: if $f : X \to Y$ then $f \circ id_X = f = id_Y \circ f$.
- Composition is associative.
- We write $X : \mathcal{C}$ rather than $X \in ob(\mathcal{C})$ and $f : X \to Y$ rather than $f \in Hom_{\mathcal{C}}(X, Y)$.
- We will restrict attention to the category Set for now.

Category Theory Interlude (II)

- If $\mathcal{C}$ and $\mathcal{D}$ are categories, then a functor $F : \mathcal{C} \to \mathcal{D}$ comprises
  - a function F from $ob(\mathcal{C})$ to $ob(\mathcal{D})$, together with
  - a function map_F from $Hom_{\mathcal{C}}(X, Y)$ to $Hom_{\mathcal{D}}(FX, FY)$
- A functor must preserve the fundamental structure of a category. This means that map_F must preserve identities and composition:

$$\begin{array}{lcl} map_F\, g \circ map_F\, f & = & map_F\,(g \circ f) \\ map_F\, id_X & = & id_{FX} \end{array}$$

Category Theory Interlude (II)

- If $\mathcal{C}$ and $\mathcal{D}$ are categories, then a functor $F : \mathcal{C} \to \mathcal{D}$ comprises
  - a function F from $ob(\mathcal{C})$ to $ob(\mathcal{D})$, together with
  - a function map_F from $Hom_{\mathcal{C}}(X, Y)$ to $Hom_{\mathcal{D}}(FX, FY)$
- A functor must preserve the fundamental structure of a category. This means that map_F must preserve identities and composition:

$$
\begin{array}{lcl}
map_F\, g \circ map_F\, f & = & map_F\, (g \circ f) \\
map_F\, id_X & = & id_{FX}
\end{array}
$$

Functorial Semantics for ADTs: Overview

- Each ADT has an underlying functor F because of strict positivity.
- Kelly's Transfinite Construction of Free Algebras (TFCA) constructs free (i.e., initial) algebras for these functors.
- The carrier of the initial algebra for a functor F is its least fixpoint μF.
- If the ADT D is defined by $\mathsf{D} = \mathsf{F}\,\mathsf{D}$, where F denotes the underlying functor F for D, then we interpret D as μF.

Functorial Semantics for ADTs: Overview

- Each ADT has an underlying functor F because of strict positivity.
- Kelly's Transfinite Construction of Free Algebras (TFCA) constructs free (i.e., initial) algebras for these functors.
- The carrier of the initial algebra for a functor F is its least fixpoint μF.
- If the ADT D is defined by $\mathsf{D} = \mathsf{F}\,\mathsf{D}$, where F denotes the underlying functor F for D, then we interpret D as μF.

Functorial Semantics for ADTs: Overview

- Each ADT has an underlying functor F because of strict positivity.
- Kelly's Transfinite Construction of Free Algebras (TFCA) constructs free (i.e., initial) algebras for these functors.
- The carrier of the initial algebra for a functor F is its least fixpoint μF.
- If the ADT D is defined by $\mathsf{D} = \mathsf{F}\,\mathsf{D}$, where F denotes the underlying functor F for D, then we interpret D as μF.

Functorial Semantics for ADTs: Overview

- Each ADT has an underlying functor F because of strict positivity.
- Kelly's Transfinite Construction of Free Algebras (TFCA) constructs free (i.e., initial) algebras for these functors.
- The carrier of the initial algebra for a functor F is its least fixpoint μF.
- If the ADT D is defined by $\mathsf{D} = \mathsf{F}\,\mathsf{D}$, where F denotes the underlying functor F for D, then we interpret D as μF.

Transfinite Construction of Free Algebras (Kelly'80)

- If

  $\mathcal{C}$ is a locally λ-presentable category interpreting types,

  0 is the initial object of $\mathcal{C}$,

  and

  $F : \mathcal{C} \to \mathcal{C}$ is a λ-cocontinuous functor

  then F has an initial algebra, and its carrier is the least fixpoint μF of F computed by

$$0 \hookrightarrow F\,0 \hookrightarrow F\,(F\,0)... \hookrightarrow F^n\,0... \hookrightarrow \mu F$$

- I will be deliberately vague about the requirements needed on the category interpreting types and the functors underlying data types.

- For concreteness, take $\mathcal{C}$ to be Set and F to be polynomial.

Transfinite Construction of Free Algebras (Kelly'80)

- If

  $\mathcal{C}$ is a locally λ-presentable category interpreting types,

  0 is the initial object of $\mathcal{C}$,

  and

  $F : \mathcal{C} \to \mathcal{C}$ is a λ-cocontinuous functor

  then F has an initial algebra, and its carrier is the least fixpoint μF of F computed by

$$0 \hookrightarrow F\,0 \hookrightarrow F\,(F\,0)... \hookrightarrow F^n\,0... \hookrightarrow \mu F$$

- I will be deliberately vague about the requirements needed on the category interpreting types and the functors underlying data types.

- For concreteness, take $\mathcal{C}$ to be Set and F to be polynomial.

Transfinite Construction of Free Algebras (Kelly'80)

- If

  $\mathcal{C}$ is a locally λ-presentable category interpreting types,

  0 is the initial object of $\mathcal{C}$,

  and

  $F : \mathcal{C} \to \mathcal{C}$ is a λ-cocontinuous functor

  then F has an initial algebra, and its carrier is the least fixpoint μF of F computed by

  $$0 \hookrightarrow F\,0 \hookrightarrow F\,(F\,0)... \hookrightarrow F^n\,0... \hookrightarrow \mu F$$

- I will be deliberately vague about the requirements needed on the category interpreting types and the functors underlying data types.

- For concreteness, take $\mathcal{C}$ to be Set and F to be polynomial.

Semantics of ADTs

- data Bool : Set where
  false : Bool
  true : Bool

  has $F\,X = 1 + 1$, so Bool is interpreted as μF, i.e., as $\mu X.\, 1 + 1$

- data Nat : Set where
  zero : Nat
  suc : Nat → Nat

  has $F\,X = 1 + X$, so Nat is interpreted as μF, i.e., as $\mu X.\, 1 + X$

- data List (A : Set) : Set where
  [] : List A
  _ :: _ : A → List A → List A

  has $F\,X = 1 + A \times X$, so List, A is interpreted as μF, i.e., as $\mu X.\, 1 + A \times X$

- data Tree (A : Set) (B : Set) : Set where
  leaf : A → Tree A B
  node : Tree A B → B → Tree A B → Tree A B

  has $F\,X = A + X \times B \times X$, so Tree A B is interpreted as μF, i.e., as $\mu X.\, A + X \times B \times X$

Semantics of ADTs

- $$\begin{array}{l} \mathsf{data\ Bool : Set\ where} \\ \quad \mathsf{false} \ : \ \mathsf{Bool} \\ \quad \mathsf{true} \ \ : \ \mathsf{Bool} \end{array}$$

  has $F\,X = 1 + 1$, so Bool is interpreted as μF, i.e., as $\mu X.\, 1 + 1$

- $$\begin{array}{l} \mathsf{data\ Nat : Set\ where} \\ \quad \mathsf{zero} \ : \ \mathsf{Nat} \\ \quad \mathsf{suc} \ \ \ : \ \mathsf{Nat} \to \mathsf{Nat} \end{array}$$

  has $F\,X = 1 + X$, so Nat is interpreted as μF, i.e., as $\mu X.\, 1 + X$

- $$\begin{array}{l} \mathsf{data\ List\ (A : Set) : Set\ where} \\ \quad [\,] \ \ \ \ \ : \ \mathsf{List\ A} \\ \quad \_::\_ \ : \ \mathsf{A} \to \mathsf{List\ A} \to \mathsf{List\ A} \end{array}$$

  has $F\,X = 1 + A \times X$, so List, A is interpreted as μF, i.e., as $\mu X.\, 1 + A \times X$

- $$\begin{array}{l} \mathsf{data\ Tree\ (A : Set)\ (B : Set) : Set\ where} \\ \quad \mathsf{leaf} \ \ \ : \ \mathsf{A} \to \mathsf{Tree\ A\ B} \\ \quad \mathsf{node} \ : \ \mathsf{Tree\ A\ B} \to \mathsf{B} \to \mathsf{Tree\ A\ B} \to \mathsf{Tree\ A\ B} \end{array}$$

  has $F\,X = A + X \times B \times X$, so Tree A B is interpreted as μF, i.e., as $\mu X.\, A + X \times B \times X$

Semantics of ADTs

- $$\begin{array}{l} \mathsf{data\ Bool} : \mathsf{Set\ where} \\ \quad \mathsf{false} \ : \ \mathsf{Bool} \\ \quad \mathsf{true} \ : \ \mathsf{Bool} \end{array}$$

  has $F\,X = 1 + 1$, so Bool is interpreted as μF, i.e., as $\mu X.\,1 + 1$

- $$\begin{array}{l} \mathsf{data\ Nat} : \mathsf{Set\ where} \\ \quad \mathsf{zero} \ : \ \mathsf{Nat} \\ \quad \mathsf{suc} \ : \ \mathsf{Nat} \to \mathsf{Nat} \end{array}$$

  has $F\,X = 1 + X$, so Nat is interpreted as μF, i.e., as $\mu X.\,1 + X$

- $$\begin{array}{l} \mathsf{data\ List}\,(\mathsf{A} : \mathsf{Set}) : \mathsf{Set\ where} \\ \quad [\,] \ : \ \mathsf{List\ A} \\ \quad \_ :: \_ \ : \ \mathsf{A} \to \mathsf{List\ A} \to \mathsf{List\ A} \end{array}$$

  has $F\,X = 1 + A \times X$, so List, A is interpreted as μF, i.e., as $\mu X.\,1 + A \times X$

- $$\begin{array}{l} \mathsf{data\ Tree}\,(\mathsf{A} : \mathsf{Set})\,(\mathsf{B} : \mathsf{Set}) : \mathsf{Set\ where} \\ \quad \mathsf{leaf} \ : \ \mathsf{A} \to \mathsf{Tree\ A\ B} \\ \quad \mathsf{node} \ : \ \mathsf{Tree\ A\ B} \to \mathsf{B} \to \mathsf{Tree\ A\ B} \to \mathsf{Tree\ A\ B} \end{array}$$

  has $F\,X = A + X \times B \times X$, so Tree A B is interpreted as μF, i.e., as $\mu X.\,A + X \times B \times X$

Semantics of ADTs

- $$\begin{array}{l} \mathsf{data\ Bool : Set\ where} \\ \quad \mathsf{false}\ :\ \mathsf{Bool} \\ \quad \mathsf{true}\ \ :\ \mathsf{Bool} \end{array}$$

  has $F\,X = 1 + 1$, so Bool is interpreted as μF, i.e., as $\mu X.\,1 + 1$
- $$\begin{array}{l} \mathsf{data\ Nat : Set\ where} \\ \quad \mathsf{zero}\ :\ \mathsf{Nat} \\ \quad \mathsf{suc}\ \ :\ \mathsf{Nat} \to \mathsf{Nat} \end{array}$$

  has $F\,X = 1 + X$, so Nat is interpreted as μF, i.e., as $\mu X.\,1 + X$
- $$\begin{array}{l} \mathsf{data\ List\ (A : Set) : Set\ where} \\ \quad [\,]\ \ \ \ :\ \mathsf{List\ A} \\ \quad \_ :: \_\ :\ \mathsf{A} \to \mathsf{List\ A} \to \mathsf{List\ A} \end{array}$$

  has $F\,X = 1 + A \times X$, so List, A is interpreted as μF, i.e., as $\mu X.\,1 + A \times X$
- $$\begin{array}{l} \mathsf{data\ Tree\ (A : Set)\ (B : Set) : Set\ where} \\ \quad \mathsf{leaf}\ \ :\ \mathsf{A} \to \mathsf{Tree\ A\ B} \\ \quad \mathsf{node}\ :\ \mathsf{Tree\ A\ B} \to \mathsf{B} \to \mathsf{Tree\ A\ B} \to \mathsf{Tree\ A\ B} \end{array}$$

  has $F\,X = A + X \times B \times X$, so Tree A B is interpreted as μF, i.e., as $\mu X.\,A + X \times B \times X$

Syntax of Nested Types (I)

Nested types encode stronger properties, leading to stronger correctness guarantees.

- Perfect trees

$$\begin{array}{l} \mathsf{data\ PTree} : \mathsf{Set} \to \mathsf{Set\ where} \\ \quad \mathsf{pleaf}\ \ : \forall\{\mathsf{A} : \mathsf{Set}\} \to \mathsf{A} \to \mathsf{PTree\,A} \\ \quad \mathsf{pnode} : \forall\{\mathsf{A} : \mathsf{Set}\} \to \mathsf{PTree}(\mathsf{A} \times \mathsf{A}) \to \mathsf{PTree\,A} \end{array}$$

  PTree A encodes the constraint that a datum is a list of elements of type A whose length is a power of 2.

- Bushes

$$\begin{array}{l} \mathsf{data\ Bush} : \mathsf{Set} \to \mathsf{Set\ where} \\ \quad \mathsf{bnil}\ \ \ \ : \forall\{\mathsf{A} : \mathsf{Set}\} \to \mathsf{Bush\,A} \\ \quad \mathsf{bnode} : \forall\{\mathsf{A} : \mathsf{Set}\} \to \mathsf{A} \to \mathsf{Bush\,(Bush\,A)} \to \mathsf{Bush\,A} \end{array}$$

  Bush A encodes the constraint that a datum is... a bush of elements of type A.

- Constructors can have input types involving instances of the data type being defined other than the one being defined.
- For truly nested types like Bush A these instances can even involve themselves!
- The return type of every constructor is still the same (variable) instance of the data type as the one being defined.
- A nested type defines an inductive family of types (not a family of inductive types).

Syntax of Nested Types (I)

Nested types encode stronger properties, leading to stronger correctness guarantees.

- Perfect trees

$$\begin{array}{l} \mathsf{data\ PTree} : \mathsf{Set} \to \mathsf{Set\ where} \\ \quad \mathsf{pleaf} \;\; : \forall\{\mathsf{A} : \mathsf{Set}\} \to \mathsf{A} \to \mathsf{PTree\,A} \\ \quad \mathsf{pnode} : \forall\{\mathsf{A} : \mathsf{Set}\} \to \mathsf{PTree}(\mathsf{A} \times \mathsf{A}) \to \mathsf{PTree\,A} \end{array}$$

  PTree A encodes the constraint that a datum is a list of elements of type A whose length is a power of 2.

- Bushes

$$\begin{array}{l} \mathsf{data\ Bush} : \mathsf{Set} \to \mathsf{Set\ where} \\ \quad \mathsf{bnil} \quad : \forall\{\mathsf{A} : \mathsf{Set}\} \to \mathsf{Bush\,A} \\ \quad \mathsf{bnode} : \forall\{\mathsf{A} : \mathsf{Set}\} \to \mathsf{A} \to \mathsf{Bush\,(Bush\,A)} \to \mathsf{Bush\,A} \end{array}$$

  Bush A encodes the constraint that a datum is... a bush of elements of type A.

- Constructors can have input types involving instances of the data type being defined other than the one being defined.
- For truly nested types like Bush A these instances can even involve themselves!
- The return type of every constructor is still the same (variable) instance of the data type as the one being defined.
- A nested type defines an inductive family of types (not a family of inductive types).

Syntax of Nested Types (I)

Nested types encode stronger properties, leading to stronger correctness guarantees.

- Perfect trees

$$\begin{array}{l} \mathsf{data\ PTree : Set \to Set\ where} \\ \quad \mathsf{pleaf\ \ : \forall\{A : Set\} \to A \to PTree\,A} \\ \quad \mathsf{pnode : \forall\{A : Set\} \to PTree(A \times A) \to PTree\,A} \end{array}$$

  PTree A encodes the constraint that a datum is a list of elements of type A whose length is a power of 2.

- Bushes

$$\begin{array}{l} \mathsf{data\ Bush : Set \to Set\ where} \\ \quad \mathsf{bnil\ \ \ \ : \forall\{A : Set\} \to Bush\,A} \\ \quad \mathsf{bnode : \forall\{A : Set\} \to A \to\ Bush\,(Bush\,A) \to Bush\,A} \end{array}$$

  Bush A encodes the constraint that a datum is... a bush of elements of type A.

- Constructors can have input types involving instances of the data type being defined other than the one being defined.
- For truly nested types like Bush A these instances can even involve themselves!
- The return type of every constructor is still the same (variable) instance of the data type as the one being defined.
- A nested type defines an inductive family of types (not a family of inductive types).

Syntax of Nested Types (I)

Nested types encode stronger properties, leading to stronger correctness guarantees.

- Perfect trees

$$\begin{array}{l} \mathsf{data\ PTree : Set \to Set\ where} \\ \quad \mathsf{pleaf\ \ : \forall\{A : Set\} \to A \to PTree\,A} \\ \quad \mathsf{pnode : \forall\{A : Set\} \to PTree(A \times A) \to PTree\,A} \end{array}$$

  PTree A encodes the constraint that a datum is a list of elements of type A whose length is a power of 2.

- Bushes

$$\begin{array}{l} \mathsf{data\ Bush : Set \to Set\ where} \\ \quad \mathsf{bnil\ \ \ : \forall\{A : Set\} \to Bush\,A} \\ \quad \mathsf{bnode : \forall\{A : Set\} \to A \to Bush\,(Bush\,A) \to Bush\,A} \end{array}$$

  Bush A encodes the constraint that a datum is... a bush of elements of type A.

- Constructors can have input types involving instances of the data type being defined other than the one being defined.
- For truly nested types like Bush A these instances can even involve themselves!
- The return type of every constructor is still the same (variable) instance of the data type as the one being defined.
- A nested type defines an inductive family of types (not a family of inductive types).

Syntax of Nested Types (I)

Nested types encode stronger properties, leading to stronger correctness guarantees.

- Perfect trees

$$
\begin{array}{l}
\mathsf{data\ PTree : Set \to Set\ where} \\
\quad \mathsf{pleaf}\ \ : \forall\{\mathsf{A : Set}\} \to \mathsf{A} \to \mathsf{PTree\,A} \\
\quad \mathsf{pnode} : \forall\{\mathsf{A : Set}\} \to \mathsf{PTree(A \times A)} \to \mathsf{PTree\,A}
\end{array}
$$

  PTree A encodes the constraint that a datum is a list of elements of type A whose length is a power of 2.

- Bushes

$$
\begin{array}{l}
\mathsf{data\ Bush : Set \to Set\ where} \\
\quad \mathsf{bnil}\ \ \ \ : \forall\{\mathsf{A : Set}\} \to \mathsf{Bush\,A} \\
\quad \mathsf{bnode} : \forall\{\mathsf{A : Set}\} \to \mathsf{A} \to \mathsf{Bush\,(Bush\,A)} \to \mathsf{Bush\,A}
\end{array}
$$

  Bush A encodes the constraint that a datum is... a bush of elements of type A.

- Constructors can have input types involving instances of the data type being defined other than the one being defined.
- For truly nested types like Bush A these instances can even involve themselves!
- The return type of every constructor is still the same (variable) instance of the data type as the one being defined.
- A nested type defines an inductive family of types (not a family of inductive types).

Syntax of Nested Types (I)

Nested types encode stronger properties, leading to stronger correctness guarantees.

- Perfect trees

$$\begin{array}{l} \mathsf{data\ PTree} : \mathsf{Set} \to \mathsf{Set\ where} \\ \quad \mathsf{pleaf} \;\;\, : \forall\{\mathsf{A} : \mathsf{Set}\} \to \mathsf{A} \to \mathsf{PTree\,A} \\ \quad \mathsf{pnode} : \forall\{\mathsf{A} : \mathsf{Set}\} \to \mathsf{PTree}(\mathsf{A} \times \mathsf{A}) \to \mathsf{PTree\,A} \end{array}$$

  PTree A encodes the constraint that a datum is a list of elements of type A whose length is a power of 2.

- Bushes

$$\begin{array}{l} \mathsf{data\ Bush} : \mathsf{Set} \to \mathsf{Set\ where} \\ \quad \mathsf{bnil} \quad\;\; : \forall\{\mathsf{A} : \mathsf{Set}\} \to \mathsf{Bush\,A} \\ \quad \mathsf{bnode} : \forall\{\mathsf{A} : \mathsf{Set}\} \to \mathsf{A} \to \mathsf{Bush\,(Bush\,A)} \to \mathsf{Bush\,A} \end{array}$$

  Bush A encodes the constraint that a datum is... a bush of elements of type A.

- Constructors can have input types involving instances of the data type being defined other than the one being defined.
- For truly nested types like Bush A these instances can even involve themselves!
- The return type of every constructor is still the same (variable) instance of the data type as the one being defined.
- A nested type defines an inductive family of types (not a family of inductive types).

Syntax of Nested Types (II)

- The general form of a nested type is

$$\begin{array}{l} \mathsf{data}\ D\, A_1 \ldots A_n : B_1 \to \ldots \to B_m \to \mathsf{Set}\ \mathsf{where} \\ \quad c_1 : \forall\{A_1 \ldots A_n\, B_1 \ldots B_m\} \to T_{11} \to \ldots \to T_{1j_1} \to D\, A_1 \ldots A_n\, B_1 \ldots B_m \\ \quad \ldots \\ \quad c_k : \forall\{A_1 \ldots A_n\, B_1 \ldots B_m\} \to T_{k1} \to \ldots \to T_{kj_k} \to D\, A_1 \ldots A_n\, B_1 \ldots B_m \end{array}$$

  where either

  T_{ij} is not inductive and does not mention D

  or

  T_{ij} is inductive and has the form

$$C_1 \to \ldots \to C_p \to D\, A_1 \ldots A_n\, V_1 \ldots V_m$$

  where D does not occur in any C_i or any V_i, and each V_i is functorial in $B_1, \ldots B_m$

- Strict positivity

  $\Longrightarrow$ no negative occurrences of D in argument types of constructors

  $\Longrightarrow$ D can be interpreted as the least fixpoint of a functor

Syntax of Nested Types (II)

- The general form of a nested type is

$$
\begin{array}{l}
\mathsf{data}\ D\, A_1 \ldots A_n : B_1 \to \ldots \to B_m \to \mathsf{Set}\ \mathsf{where} \\
\quad c_1 : \forall\{A_1 \ldots A_n\, B_1 \ldots B_m\} \to T_{11} \to \ldots \to T_{1j_1} \to D\, A_1 \ldots A_n\, B_1 \ldots B_m \\
\quad \ldots \\
\quad c_k : \forall\{A_1 \ldots A_n\, B_1 \ldots B_m\} \to T_{k1} \to \ldots \to T_{kj_k} \to D\, A_1 \ldots A_n\, B_1 \ldots B_m
\end{array}
$$

  where either

  T_{ij} is not inductive and does not mention D

  or

  T_{ij} is inductive and has the form

$$C_1 \to \ldots \to C_p \to D\, A_1 \ldots A_n\, V_1 \ldots V_m$$

  where D does not occur in any C_i or any V_i, and each V_i is functorial in $B_1, \ldots B_m$

- Strict positivity

  $\Longrightarrow$ no negative occurrences of D in argument types of constructors

  $\Longrightarrow$ D can be interpreted as the least fixpoint of a functor

Semantics of Nested Types (III)

- Like ADTs, nested types can be interpreted as fixpoints of functors...

  ...but now the functors must be higher-order!

- If $\mathcal{C}$ and $\mathcal{D}$ are categories, then the collection of functors from $\mathcal{C}$ to $\mathcal{D}$ also form a category. Its objects are functors from $\mathcal{C}$ to $\mathcal{D}$ and its morphisms are natural transformations between such functors.

- A natural transformation $\eta : F \to G$ is a collection $\{\eta_X : F\,X \to G\,X\}_{X:\mathcal{C}}$ such that if $f : X \to Y$ in $\mathcal{C}$ then $\eta_Y \circ map_F\, f = map_G\, f \circ \eta_X$

$$\begin{tikzcd}
F\,X \arrow[r, "\eta_X"] \arrow[d, "map_F\, f"'] & G\,X \arrow[d, "map_G\, f"] \\
F\,Y \arrow[r, "\eta_Y"] & G\,Y
\end{tikzcd}$$

- The identity on F is the identity natural transformation id_F from F to F.

- Composition of natural transformations is componentwise, i.e., if $X : \mathcal{C}$ then

$$(\eta \circ \mu)_X = \eta_X \circ \mu_X$$

Semantics of Nested Types (III)

- Like ADTs, nested types can be interpreted as fixpoints of functors...

  ...but now the functors must be higher-order!

- If $\mathcal{C}$ and $\mathcal{D}$ are categories, then the collection of functors from $\mathcal{C}$ to $\mathcal{D}$ also form a category. Its objects are functors from $\mathcal{C}$ to $\mathcal{D}$ and its morphisms are natural transformations between such functors.
- A natural transformation $\eta : F \to G$ is a collection $\{\eta_X : F\,X \to G\,X\}_{X:\mathcal{C}}$ such that if $f : X \to Y$ in $\mathcal{C}$ then $\eta_Y \circ map_F\, f = map_G\, f \circ \eta_X$

$$\begin{tikzcd} F\,X \arrow[r, "\eta_X"] \arrow[d, "map_F\ f"'] & G\,X \arrow[d, "map_G\ f"] \\ F\,Y \arrow[r, "\eta_Y"] & G\,Y \end{tikzcd}$$

- The identity on F is the identity natural transformation id_F from F to F.
- Composition of natural transformations is componentwise, i.e., if $X : \mathcal{C}$ then

$$(\eta \circ \mu)_X = \eta_X \circ \mu_X$$

Semantics of Nested Types (III)

- Like ADTs, nested types can be interpreted as fixpoints of functors...

  ...but now the functors must be higher-order!

- If $\mathcal{C}$ and $\mathcal{D}$ are categories, then the collection of functors from $\mathcal{C}$ to $\mathcal{D}$ also form a category. Its objects are functors from $\mathcal{C}$ to $\mathcal{D}$ and its morphisms are natural transformations between such functors.

- A natural transformation $\eta : F \to G$ is a collection $\{\eta_X : F\,X \to G\,X\}_{X:\mathcal{C}}$ such that if $f : X \to Y$ in $\mathcal{C}$ then $\eta_Y \circ map_F\, f = map_G\, f \circ \eta_X$

$$\begin{tikzcd}
F\,X \arrow[r, "\eta_X"] \arrow[d, "map_F\, f"'] & G\,X \arrow[d, "map_G\, f"] \\
F\,Y \arrow[r, "\eta_Y"] & G\,Y
\end{tikzcd}$$

- The identity on F is the identity natural transformation id_F from F to F.

- Composition of natural transformations is componentwise, i.e., if $X : \mathcal{C}$ then

$$(\eta \circ \mu)_X = \eta_X \circ \mu_X$$

Semantics of Nested Types (III)

- Like ADTs, nested types can be interpreted as fixpoints of functors...

  ...but now the functors must be higher-order!
- If $\mathcal{C}$ and $\mathcal{D}$ are categories, then the collection of functors from $\mathcal{C}$ to $\mathcal{D}$ also form a category. Its objects are functors from $\mathcal{C}$ to $\mathcal{D}$ and its morphisms are natural transformations between such functors.
- A natural transformation $\eta : F \to G$ is a collection $\{\eta_X : F\,X \to G\,X\}_{X:\mathcal{C}}$ such that if $f : X \to Y$ in $\mathcal{C}$ then $\eta_Y \circ map_F\, f = map_G\, f \circ \eta_X$

$$\begin{tikzcd}
F\,X \arrow[r, "\eta_X"] \arrow[d, "map_F\, f"'] & G\,X \arrow[d, "map_G\, f"] \\
F\,Y \arrow[r, "\eta_Y"] & G\,Y
\end{tikzcd}$$

- The identity on F is the identity natural transformation id_F from F to F.
- Composition of natural transformations is componentwise, i.e., if $X : \mathcal{C}$ then

$$(\eta \circ \mu)_X = \eta_X \circ \mu_X$$

Semantics of Nested Types (III)

- Like ADTs, nested types can be interpreted as fixpoints of functors...

  ...but now the functors must be higher-order!

- If $\mathcal{C}$ and $\mathcal{D}$ are categories, then the collection of functors from $\mathcal{C}$ to $\mathcal{D}$ also form a category. Its objects are functors from $\mathcal{C}$ to $\mathcal{D}$ and its morphisms are natural transformations between such functors.
- A natural transformation $\eta : F \to G$ is a collection $\{\eta_X : F\,X \to G\,X\}_{X:\mathcal{C}}$ such that if $f : X \to Y$ in $\mathcal{C}$ then $\eta_Y \circ map_F\, f = map_G\, f \circ \eta_X$

$$\begin{tikzcd}
F\,X \arrow[r, "\eta_X"] \arrow[d, "map_F\, f"'] & G\,X \arrow[d, "map_G\, f"] \\
F\,Y \arrow[r, "\eta_Y"] & G\,Y
\end{tikzcd}$$

- The identity on F is the identity natural transformation id_F from F to F.
- Composition of natural transformations is componentwise, i.e., if $X : \mathcal{C}$ then

$$(\eta \circ \mu)_X = \eta_X \circ \mu_X$$

Semantics of Nested Types (III)

- Like ADTs, nested types can be interpreted as fixpoints of functors...

  ...but now the functors must be higher-order!

- If $\mathcal{C}$ and $\mathcal{D}$ are categories, then the collection of functors from $\mathcal{C}$ to $\mathcal{D}$ also form a category. Its objects are functors from $\mathcal{C}$ to $\mathcal{D}$ and its morphisms are natural transformations between such functors.
- A natural transformation $\eta : F \to G$ is a collection $\{\eta_X : F\,X \to G\,X\}_{X:\mathcal{C}}$ such that if $f : X \to Y$ in $\mathcal{C}$ then $\eta_Y \circ map_F\, f = map_G\, f \circ \eta_X$

$$\begin{tikzcd}
F\,X \arrow[r, "\eta_X"] \arrow[d, "map_F\, f"'] & G\,X \arrow[d, "map_G\, f"] \\
F\,Y \arrow[r, "\eta_Y"] & G\,Y
\end{tikzcd}$$

- The identity on F is the identity natural transformation id_F from F to F.
- Composition of natural transformations is componentwise, i.e., if $X : \mathcal{C}$ then

$$(\eta \circ \mu)_X = \eta_X \circ \mu_X$$

Semantics of Nested Types (IV)

- A higher-order functor H is a functor (on a functor category) so it has an action on objects (functors) and on morphisms (natural transformations) of that category.
- If $F : \mathcal{C} \to \mathcal{D}$ is a functor, then HF is also a functor from $\mathcal{C}$ to $\mathcal{D}$
  - if $X : \mathcal{C}$ then $HFX : \mathcal{D}$
  - if $f : X \to Y$ in $\mathcal{C}$ then $HFf : HFX \to HFY$ in $\mathcal{D}$.
  - if $\eta : F \to G$ then $map_H\,\eta : HF \to HG$
- map_H must preserve identities and composition (now for natural transformations).
- To give an initial algebra semantics for nested types we must compute fixpoints of higher-order functors.

Semantics of Nested Types (IV)

- A higher-order functor H is a functor (on a functor category) so it has an action on objects (functors) and on morphisms (natural transformations) of that category.
- If $F : \mathcal{C} \to \mathcal{D}$ is a functor, then HF is also a functor from $\mathcal{C}$ to $\mathcal{D}$
  - if $X : \mathcal{C}$ then $HFX : \mathcal{D}$
  - if $f : X \to Y$ in $\mathcal{C}$ then $HFf : HFX \to HFY$ in $\mathcal{D}$.
  - if $\eta : F \to G$ then $map_H\, \eta : HF \to HG$
- map_H must preserve identities and composition (now for natural transformations).
- To give an initial algebra semantics for nested types we must compute fixpoints of higher-order functors.

Semantics of Nested Types (IV)

- A higher-order functor H is a functor (on a functor category) so it has an action on objects (functors) and on morphisms (natural transformations) of that category.
- If $F : \mathcal{C} \to \mathcal{D}$ is a functor, then HF is also a functor from $\mathcal{C}$ to $\mathcal{D}$
  - if $X : \mathcal{C}$ then $HFX : \mathcal{D}$
  - if $f : X \to Y$ in $\mathcal{C}$ then $HFf : HFX \to HFY$ in $\mathcal{D}$.
  - if $\eta : F \to G$ then $map_H\, \eta : HF \to HG$
- map_H must preserve identities and composition (now for natural transformations).

- To give an initial algebra semantics for nested types we must compute fixpoints of higher-order functors.

Semantics of Nested Types (IV)

- A higher-order functor H is a functor (on a functor category) so it has an action on objects (functors) and on morphisms (natural transformations) of that category.
- If $F : \mathcal{C} \to \mathcal{D}$ is a functor, then HF is also a functor from $\mathcal{C}$ to $\mathcal{D}$
  - if $X : \mathcal{C}$ then $HFX : \mathcal{D}$
  - if $f : X \to Y$ in $\mathcal{C}$ then $HFf : HFX \to HFY$ in $\mathcal{D}$.
  - if $\eta : F \to G$ then $map_H\, \eta : HF \to HG$
- map_H must preserve identities and composition (now for natural transformations).

- To give an initial algebra semantics for nested types we must compute fixpoints of higher-order functors.

Semantics of Nested Types (V)

- $$\begin{array}{l} \mathsf{data\ PTree : Set \to Set\ where} \\ \quad \mathsf{pleaf\ \ \, : \forall\{A : Set\} \to A \to PTree\,A} \\ \quad \mathsf{pnode : \forall\{A : Set\} \to PTree(A \times A) \to PTree\,A} \end{array}$$

  has $H\,F\,X = X + F\,(X \times X)$, so PTree is interpreted as μH, i.e., as $\mu F.\,\lambda X.\,X + F\,(X \times X)$

- $$\begin{array}{l} \mathsf{data\ Bush : Set \to Set\ where} \\ \quad \mathsf{bnil\ \ \ \ : \forall\{A : Set\} \to Bush\,A} \\ \quad \mathsf{bnode : \forall\{A : Set\} \to A \to\ Bush\,(Bush\,A) \to Bush\,A} \end{array}$$

  has $H\,F\,X = 1 + X \times F\,(F\,X)$, so Bush is interpreted as μH, i.e., as $\mu F.\,\lambda X.\,1 + X \times F\,(F\,X)$

Semantics of Nested Types (V)

- $$\begin{array}{l} \mathsf{data\ PTree : Set \to Set\ where} \\ \quad \mathsf{pleaf} \;\; : \forall\{\mathsf{A : Set}\} \to \mathsf{A} \to \mathsf{PTree\,A} \\ \quad \mathsf{pnode} : \forall\{\mathsf{A : Set}\} \to \mathsf{PTree(A \times A)} \to \mathsf{PTree\,A} \end{array}$$

  has $H\,F\,X = X + F\,(X \times X)$, so PTree is interpreted as μH,
  i.e., as $\mu F.\,\lambda X.\,X + F\,(X \times X)$

- $$\begin{array}{l} \mathsf{data\ Bush : Set \to Set\ where} \\ \quad \mathsf{bnil} \quad\; : \forall\{\mathsf{A : Set}\} \to \mathsf{Bush\,A} \\ \quad \mathsf{bnode} : \forall\{\mathsf{A : Set}\} \to \mathsf{A} \to \mathsf{Bush\,(Bush\,A)} \to \mathsf{Bush\,A} \end{array}$$

  has $H\,F\,X = 1 + X \times F\,(F\,X)$, so Bush is interpreted as μH,
  i.e., as $\mu F.\,\lambda X.\,1 + X \times F\,(F\,X)$

Higher-Order Functorial Semantics of ADTs

- ADTs are uniform in their type parameters, so they also define inductive families.
- That is, we can interpret ADTs as fixpoints of higher-order functors too.
-

```
data List (A : Set) : Set where
  []    : List A
  _::_  : A → List A → List A
```

is also

```
data List : Set → Set where
  []    : ∀{A : Set} → List A
  _::_  : ∀{A : Set} → A → List A → List A
```

which has $H\,F\,X = 1 + X \times F\,X$, so List is interpreted as μH,
i.e., as $\mu F.\,\lambda X.\,1 + X \times F\,X$

Higher-Order Functorial Semantics of ADTs

- ADTs are uniform in their type parameters, so they also define inductive families.
- That is, we can interpret ADTs as fixpoints of higher-order functors too.
-

```
data List (A : Set) : Set where
  []    : List A
  _::_  : A → List A → List A
```

is also

```
data List : Set → Set where
  []    : ∀{A : Set} → List A
  _::_  : ∀{A : Set} → A → List A → List A
```

which has $H\,F\,X = 1 + X \times F\,X$, so List is interpreted as μH, i.e., as $\mu F.\, \lambda X.\, 1 + X \times F\,X$

Higher-Order Functorial Semantics of ADTs

- ADTs are uniform in their type parameters, so they also define inductive families.
- That is, we can interpret ADTs as fixpoints of higher-order functors too.
-

$$\begin{array}{l} \mathsf{data\ List}\,(\mathsf{A} : \mathsf{Set}) : \mathsf{Set\ where} \\ \quad [\,] \quad\ \ : \ \mathsf{List\,A} \\ \quad \_ :: \_ \ : \ \mathsf{A} \to \mathsf{List\,A} \to \mathsf{List\,A} \end{array}$$

is also

$$\begin{array}{l} \mathsf{data\ List} : \mathsf{Set} \to \mathsf{Set\ where} \\ \quad [\,] \quad\ \ : \forall\{\mathsf{A} : \mathsf{Set}\} \to \mathsf{List\,A} \\ \quad \_ :: \_ \ : \forall\{\mathsf{A} : \mathsf{Set}\} \to \ \mathsf{A} \to \mathsf{List\,A} \to \mathsf{List\,A} \end{array}$$

which has $H\,F\,X = 1 + X \times F\,X$, so List is interpreted as μH,
i.e., as $\mu F.\,\lambda X.\,1 + X \times F\,X$

maps for ADTs and Nested Types

- If the ADT or nested type D is defined by D = HD, where H denotes the higher-order functor H for D, then we interpret D as μH.
- Because μH is itself a functor, and thus has a corresponding $map_{\mu H}$, D has a corresponding function $\mathsf{map}_{\mathsf{D}}$ that is just the reflection back into syntax of $map_{\mu H}$.
-

$$\begin{aligned}
&\mathsf{map}_{\mathsf{List}} :: (\mathsf{A} \to \mathsf{B}) \to \mathsf{List\,A} \to \mathsf{List\,B}\\
&\mathsf{map}_{\mathsf{List}}\,\mathsf{f}\,[\,] = [\,]\\
&\mathsf{map}_{\mathsf{List}}\,\mathsf{f}\,(\mathsf{x} :: \mathsf{xs}) = (\mathsf{f\,x}) :: \mathsf{map}_{\mathsf{List}}\,\mathsf{f\,xs}
\end{aligned}$$

-

$$\begin{aligned}
&\mathsf{map}_{\mathsf{PTree}} :: (\mathsf{A} \to \mathsf{B}) \to \mathsf{PTree\,A} \to \mathsf{PTree\,B}\\
&\mathsf{map}_{\mathsf{PTree}}\,\mathsf{f}\,(\mathsf{pleaf\,x}) = \mathsf{pleaf}\,(\mathsf{f\,x})\\
&\mathsf{map}_{\mathsf{PTree}}\,\mathsf{f}\,(\mathsf{pnode\,ts}) = \mathsf{pnode}\,(\mathsf{map}_{\mathsf{PTree}}\,(\mathsf{f} \times \mathsf{f})\,\mathsf{ts})
\end{aligned}$$

-

$$\begin{aligned}
&\mathsf{map}_{\mathsf{Bush}} :: (\mathsf{A} \to \mathsf{B}) \to \mathsf{Bush\,A} \to \mathsf{Bush\,B}\\
&\mathsf{map}_{\mathsf{Bush}}\,\mathsf{f\,bnil} = \mathsf{bnil}\\
&\mathsf{map}_{\mathsf{Bush}}\,\mathsf{f}\,(\mathsf{bnode\,a\,bb}) = \mathsf{bnode}\,(\mathsf{f\,a})\,(\mathsf{map}_{\mathsf{Bush}}\,(\mathsf{map}_{\mathsf{Bush}}\,\mathsf{f})\,\mathsf{bb})
\end{aligned}$$

- $\mathsf{map}_{\mathsf{D}}$ preserves the shape of a D-structure but (potentially) changes its contents.

maps for ADTs and Nested Types

- If the ADT or nested type D is defined by $\mathsf{D} = \mathsf{HD}$, where H denotes the higher-order functor H for D, then we interpret D as μH.
- Because μH is itself a functor, and thus has a corresponding $map_{\mu H}$, D has a corresponding function $\mathsf{map}_{\mathsf{D}}$ that is just the reflection back into syntax of $map_{\mu H}$.
- $$\begin{aligned} &\mathsf{map}_{\mathsf{List}} :: (\mathsf{A} \to \mathsf{B}) \to \ \mathsf{List}\,\mathsf{A} \to \mathsf{List}\,\mathsf{B} \\ &\mathsf{map}_{\mathsf{List}}\,\mathsf{f}\,[\,] \qquad\quad = [\,] \\ &\mathsf{map}_{\mathsf{List}}\,\mathsf{f}\,(\mathsf{x} :: \mathsf{xs}) = (\mathsf{f}\,\mathsf{x}) :: \mathsf{map}_{\mathsf{List}}\,\mathsf{f}\,\mathsf{xs} \end{aligned}$$
- $$\begin{aligned} &\mathsf{map}_{\mathsf{PTree}} :: (\mathsf{A} \to \mathsf{B}) \to \ \mathsf{PTree}\,\mathsf{A} \to \mathsf{PTree}\,\mathsf{B} \\ &\mathsf{map}_{\mathsf{PTree}}\,\mathsf{f}\,(\mathsf{pleaf}\,\mathsf{x}) \quad = \mathsf{pleaf}\,(\mathsf{f}\,\mathsf{x}) \\ &\mathsf{map}_{\mathsf{PTree}}\,\mathsf{f}\,(\mathsf{pnode}\,\mathsf{ts}) = \mathsf{pnode}\,(\mathsf{map}_{\mathsf{PTree}}\,(\mathsf{f} \times \mathsf{f})\,\mathsf{ts}) \end{aligned}$$
- $$\begin{aligned} &\mathsf{map}_{\mathsf{Bush}} :: (\mathsf{A} \to \mathsf{B}) \to \ \mathsf{Bush}\,\mathsf{A} \to \mathsf{Bush}\,\mathsf{B} \\ &\mathsf{map}_{\mathsf{Bush}}\,\mathsf{f}\,\mathsf{bnil} \qquad\quad\;\; = \mathsf{bnil} \\ &\mathsf{map}_{\mathsf{Bush}}\,\mathsf{f}\,(\mathsf{bnode}\,\mathsf{a}\,\mathsf{bb}) = \mathsf{bnode}\,(\mathsf{f}\,\mathsf{a})\,(\mathsf{map}_{\mathsf{Bush}}\,(\mathsf{map}_{\mathsf{Bush}}\,\mathsf{f})\,\mathsf{bb}) \end{aligned}$$
- $\mathsf{map}_{\mathsf{D}}$ preserves the shape of a D-structure but (potentially) changes its contents.

maps for ADTs and Nested Types

- If the ADT or nested type D is defined by D = HD, where H denotes the higher-order functor H for D, then we interpret D as μH.
- Because μH is itself a functor, and thus has a corresponding $map_{\mu H}$, D has a corresponding function $\mathsf{map}_{\mathsf{D}}$ that is just the reflection back into syntax of $map_{\mu H}$.
- $$\begin{array}{l} \mathsf{map}_{\mathsf{List}} :: (\mathsf{A} \to \mathsf{B}) \to \ \mathsf{List}\,\mathsf{A} \to \mathsf{List}\,\mathsf{B} \\ \mathsf{map}_{\mathsf{List}}\, \mathsf{f}\, [\,] \qquad\quad = [\,] \\ \mathsf{map}_{\mathsf{List}}\, \mathsf{f}\, (\mathsf{x} :: \mathsf{xs}) = (\mathsf{f}\,\mathsf{x}) :: \mathsf{map}_{\mathsf{List}}\, \mathsf{f}\, \mathsf{xs} \end{array}$$
- $$\begin{array}{l} \mathsf{map}_{\mathsf{PTree}} :: (\mathsf{A} \to \mathsf{B}) \to \ \mathsf{PTree}\,\mathsf{A} \to \mathsf{PTree}\,\mathsf{B} \\ \mathsf{map}_{\mathsf{PTree}}\, \mathsf{f}\, (\mathsf{pleaf}\,\mathsf{x}) \quad = \mathsf{pleaf}\,(\mathsf{f}\,\mathsf{x}) \\ \mathsf{map}_{\mathsf{PTree}}\, \mathsf{f}\, (\mathsf{pnode}\,\mathsf{ts}) = \mathsf{pnode}\,(\mathsf{map}_{\mathsf{PTree}}\,(\mathsf{f} \times \mathsf{f})\,\mathsf{ts}) \end{array}$$
- $$\begin{array}{l} \mathsf{map}_{\mathsf{Bush}} :: (\mathsf{A} \to \mathsf{B}) \to \ \mathsf{Bush}\,\mathsf{A} \to \mathsf{Bush}\,\mathsf{B} \\ \mathsf{map}_{\mathsf{Bush}}\, \mathsf{f}\, \mathsf{bnil} \qquad\qquad = \mathsf{bnil} \\ \mathsf{map}_{\mathsf{Bush}}\, \mathsf{f}\, (\mathsf{bnode}\,\mathsf{a}\,\mathsf{bb}) = \mathsf{bnode}\,(\mathsf{f}\,\mathsf{a})\,(\mathsf{map}_{\mathsf{Bush}}\,(\mathsf{map}_{\mathsf{Bush}}\,\mathsf{f})\,\mathsf{bb}) \end{array}$$
- $\mathsf{map}_{\mathsf{D}}$ preserves the shape of a D-structure but (potentially) changes its contents.

maps for ADTs and Nested Types

- If the ADT or nested type D is defined by D = HD, where H denotes the higher-order functor H for D, then we interpret D as μH.
- Because μH is itself a functor, and thus has a corresponding $map_{\mu H}$, D has a corresponding function map_D that is just the reflection back into syntax of $map_{\mu H}$.
- $$\begin{aligned}
&\mathsf{map}_{\mathsf{List}} :: (\mathsf{A} \to \mathsf{B}) \to \ \mathsf{List\,A} \to \mathsf{List\,B} \\
&\mathsf{map}_{\mathsf{List}}\ \mathsf{f}\ [\,] \qquad\quad\ = [\,] \\
&\mathsf{map}_{\mathsf{List}}\ \mathsf{f}\ (\mathsf{x} :: \mathsf{xs}) = (\mathsf{f\,x}) :: \mathsf{map}_{\mathsf{List}}\ \mathsf{f\,xs}
\end{aligned}$$
- $$\begin{aligned}
&\mathsf{map}_{\mathsf{PTree}} :: (\mathsf{A} \to \mathsf{B}) \to \ \mathsf{PTree\,A} \to \mathsf{PTree\,B} \\
&\mathsf{map}_{\mathsf{PTree}}\ \mathsf{f}\ (\mathsf{pleaf\,x}) \quad\ = \mathsf{pleaf}\ (\mathsf{f\,x}) \\
&\mathsf{map}_{\mathsf{PTree}}\ \mathsf{f}\ (\mathsf{pnode\,ts}) = \mathsf{pnode}\ (\mathsf{map}_{\mathsf{PTree}}\ (\mathsf{f} \times \mathsf{f})\ \mathsf{ts})
\end{aligned}$$
- $$\begin{aligned}
&\mathsf{map}_{\mathsf{Bush}} :: (\mathsf{A} \to \mathsf{B}) \to \ \mathsf{Bush\,A} \to \mathsf{Bush\,B} \\
&\mathsf{map}_{\mathsf{Bush}}\ \mathsf{f\,bnil} \qquad\qquad = \mathsf{bnil} \\
&\mathsf{map}_{\mathsf{Bush}}\ \mathsf{f}\ (\mathsf{bnode\,a\,bb}) = \mathsf{bnode}\ (\mathsf{f\,a})\ (\mathsf{map}_{\mathsf{Bush}}\ (\mathsf{map}_{\mathsf{Bush}}\ \mathsf{f})\ \mathsf{bb})
\end{aligned}$$
- map_D preserves the shape of a D-structure but (potentially) changes its contents.

maps for ADTs and Nested Types

- If the ADT or nested type D is defined by D = HD, where H denotes the higher-order functor H for D, then we interpret D as μH.
- Because μH is itself a functor, and thus has a corresponding $map_{\mu H}$, D has a corresponding function map_D that is just the reflection back into syntax of $map_{\mu H}$.
- $$\begin{array}{l}
\mathsf{map}_{\mathsf{List}} :: (\mathsf{A} \to \mathsf{B}) \to \ \mathsf{List\,A} \to \mathsf{List\,B}\\
\mathsf{map}_{\mathsf{List}}\ \mathsf{f}\ [\,] \qquad\quad = [\,]\\
\mathsf{map}_{\mathsf{List}}\ \mathsf{f}\ (\mathsf{x} :: \mathsf{xs}) = (\mathsf{f\,x}) :: \mathsf{map}_{\mathsf{List}}\ \mathsf{f\,xs}
\end{array}$$
- $$\begin{array}{l}
\mathsf{map}_{\mathsf{PTree}} :: (\mathsf{A} \to \mathsf{B}) \to \ \mathsf{PTree\,A} \to \mathsf{PTree\,B}\\
\mathsf{map}_{\mathsf{PTree}}\ \mathsf{f}\ (\mathsf{pleaf\,x}) \quad = \mathsf{pleaf}\ (\mathsf{f\,x})\\
\mathsf{map}_{\mathsf{PTree}}\ \mathsf{f}\ (\mathsf{pnode\,ts}) = \mathsf{pnode}\ (\mathsf{map}_{\mathsf{PTree}}\ (\mathsf{f} \times \mathsf{f})\ \mathsf{ts})
\end{array}$$
- $$\begin{array}{l}
\mathsf{map}_{\mathsf{Bush}} :: (\mathsf{A} \to \mathsf{B}) \to \ \mathsf{Bush\,A} \to \mathsf{Bush\,B}\\
\mathsf{map}_{\mathsf{Bush}}\ \mathsf{f\ bnil} \qquad\quad = \mathsf{bnil}\\
\mathsf{map}_{\mathsf{Bush}}\ \mathsf{f}\ (\mathsf{bnode\,a\,bb}) = \mathsf{bnode}\ (\mathsf{f\,a})\ (\mathsf{map}_{\mathsf{Bush}}\ (\mathsf{map}_{\mathsf{Bush}}\ \mathsf{f})\ \mathsf{bb})
\end{array}$$
- map_D preserves the shape of a D-structure but (potentially) changes its contents.

maps for ADTs and Nested Types

- If the ADT or nested type D is defined by $\mathsf{D} = \mathsf{HD}$, where H denotes the higher-order functor H for D, then we interpret D as μH.
- Because μH is itself a functor, and thus has a corresponding $map_{\mu H}$, D has a corresponding function map_D that is just the reflection back into syntax of $map_{\mu H}$.
- $$\begin{array}{l} \mathsf{map}_{\mathsf{List}} :: (\mathsf{A} \to \mathsf{B}) \to \ \mathsf{List}\,\mathsf{A} \to \mathsf{List}\,\mathsf{B} \\ \mathsf{map}_{\mathsf{List}}\,\mathsf{f}\,[\,] \qquad\quad = [\,] \\ \mathsf{map}_{\mathsf{List}}\,\mathsf{f}\,(\mathsf{x} :: \mathsf{xs}) = (\mathsf{f}\,\mathsf{x}) :: \mathsf{map}_{\mathsf{List}}\,\mathsf{f}\,\mathsf{xs} \end{array}$$
- $$\begin{array}{l} \mathsf{map}_{\mathsf{PTree}} :: (\mathsf{A} \to \mathsf{B}) \to \ \mathsf{PTree}\,\mathsf{A} \to \mathsf{PTree}\,\mathsf{B} \\ \mathsf{map}_{\mathsf{PTree}}\,\mathsf{f}\,(\mathsf{pleaf}\,\mathsf{x}) \quad\ = \mathsf{pleaf}\,(\mathsf{f}\,\mathsf{x}) \\ \mathsf{map}_{\mathsf{PTree}}\,\mathsf{f}\,(\mathsf{pnode}\,\mathsf{ts}) = \mathsf{pnode}\,(\mathsf{map}_{\mathsf{PTree}}\,(\mathsf{f} \times \mathsf{f})\,\mathsf{ts}) \end{array}$$
- $$\begin{array}{l} \mathsf{map}_{\mathsf{Bush}} :: (\mathsf{A} \to \mathsf{B}) \to \ \mathsf{Bush}\,\mathsf{A} \to \mathsf{Bush}\,\mathsf{B} \\ \mathsf{map}_{\mathsf{Bush}}\,\mathsf{f}\,\mathsf{bnil} \qquad\qquad = \mathsf{bnil} \\ \mathsf{map}_{\mathsf{Bush}}\,\mathsf{f}\,(\mathsf{bnode}\,\mathsf{a}\,\mathsf{bb}) = \mathsf{bnode}\,(\mathsf{f}\,\mathsf{a})\,(\mathsf{map}_{\mathsf{Bush}}\,(\mathsf{map}_{\mathsf{Bush}}\,\mathsf{f})\,\mathsf{bb}) \end{array}$$
- map_D preserves the shape of a D-structure but (potentially) changes its contents.

Naturality Results for ADTs and Nested Types (I)

- Just as their interpretations as fixpoints of higher-order functors give map functions for ADTs and nested types, these interpretations also give naturality results.
- A natural transformation $\eta : \mu H \to \mu H'$ gives commuting squares: if $f : X \to Y$, then

$$\begin{tikzcd}
(\mu H)\,X \arrow[r, "\eta_X"] \arrow[d, "map_{\mu H}\, f"'] & (\mu H')\,X \arrow[d, "map_{\mu H'}\, f"] \\
(\mu H)\,Y \arrow[r, "\eta_Y"] & (\mu H')\,Y
\end{tikzcd}$$

- Computationally (i.e., reflecting back into syntax), we can think of natural transformations as polymorphic functions between data types whose constructors are interpreted as μH and $\mu H'$.
- A polymorphic function (natural transformation) between (interpretations of) data types alters the shapes of data structures without changing their data elements.
- So natural transformations do the "opposite" of map functions, which act on data elements without changing the shape of the data structure in which they reside.

Naturality Results for ADTs and Nested Types (I)

- Just as their interpretations as fixpoints of higher-order functors give map functions for ADTs and nested types, these interpretations also give naturality results.
- A natural transformation $\eta : \mu H \to \mu H'$ gives commuting squares: if $f : X \to Y$, then

$$\begin{tikzcd}
(\mu H)\, X \arrow[r, "\eta_X"] \arrow[d, "map_{\mu H}\ f"'] & (\mu H')\, X \arrow[d, "map_{\mu H'}\ f"] \\
(\mu H)\, Y \arrow[r, "\eta_Y"] & (\mu H')\, Y
\end{tikzcd}$$

- Computationally (i.e., reflecting back into syntax), we can think of natural transformations as polymorphic functions between data types whose constructors are interpreted as μH and $\mu H'$.
- A polymorphic function (natural transformation) between (interpretations of) data types alters the shapes of data structures without changing their data elements.
- So natural transformations do the "opposite" of map functions, which act on data elements without changing the shape of the data structure in which they reside.

Naturality Results for ADTs and Nested Types (I)

- Just as their interpretations as fixpoints of higher-order functors give map functions for ADTs and nested types, these interpretations also give naturality results.
- A natural transformation $\eta : \mu H \to \mu H'$ gives commuting squares: if $f : X \to Y$, then

$$\begin{tikzcd}
(\mu H)\, X \arrow[r, "\eta_X"] \arrow[d, "map_{\mu H}\ f"'] & (\mu H')\, X \arrow[d, "map_{\mu H'}\ f"] \\
(\mu H)\, Y \arrow[r, "\eta_Y"] & (\mu H')\, Y
\end{tikzcd}$$

- Computationally (i.e., reflecting back into syntax), we can think of natural transformations as polymorphic functions between data types whose constructors are interpreted as μH and $\mu H'$.
- A polymorphic function (natural transformation) between (interpretations of) data types alters the shapes of data structures without changing their data elements.
- So natural transformations do the "opposite" of map functions, which act on data elements without changing the shape of the data structure in which they reside.

Naturality Results for ADTs and Nested Types (I)

- Just as their interpretations as fixpoints of higher-order functors give map functions for ADTs and nested types, these interpretations also give naturality results.
- A natural transformation $\eta : \mu H \to \mu H'$ gives commuting squares: if $f : X \to Y$, then

$$\begin{tikzcd}
(\mu H)\, X \arrow[r, "\eta_X"] \arrow[d, "map_{\mu H}\ f"'] & (\mu H')\, X \arrow[d, "map_{\mu H'}\ f"] \\
(\mu H)\, Y \arrow[r, "\eta_Y"] & (\mu H')\, Y
\end{tikzcd}$$

- Computationally (i.e., reflecting back into syntax), we can think of natural transformations as polymorphic functions between data types whose constructors are interpreted as μH and $\mu H'$.
- A polymorphic function (natural transformation) between (interpretations of) data types alters the shapes of data structures without changing their data elements.
- So natural transformations do the "opposite" of map functions, which act on data elements without changing the shape of the data structure in which they reside.

Naturality Results for ADTs and Nested Types (I)

- Just as their interpretations as fixpoints of higher-order functors give map functions for ADTs and nested types, these interpretations also give naturality results.
- A natural transformation $\eta : \mu H \to \mu H'$ gives commuting squares: if $f : X \to Y$, then

$$\begin{array}{ccc} (\mu H)\,X & \xrightarrow{\eta_X} & (\mu H')\,X \\ \downarrow{\scriptstyle map_{\mu H}\, f} & & \downarrow{\scriptstyle map_{\mu H'}\, f} \\ (\mu H)\,Y & \xrightarrow{\eta_Y} & (\mu H')\,Y \end{array}$$

- Computationally (i.e., reflecting back into syntax), we can think of natural transformations as polymorphic functions between data types whose constructors are interpreted as μH and $\mu H'$.
- A polymorphic function (natural transformation) between (interpretations of) data types alters the shapes of data structures without changing their data elements.
- So natural transformations do the "opposite" of map functions, which act on data elements without changing the shape of the data structure in which they reside.

Naturality Results for ADTs and Nested Types (II)

- The naturality square for (the interpretation of) a polymorphic function says that it doesn't matter in which order we apply the function and the map operations.
- If the polymorphic function $\mathsf{flatten} : \forall\{A : \mathsf{Set}\} \to \mathsf{PTree}\,A \to \mathsf{List}\,A$ acts like this

$$\begin{aligned}&\mathsf{flatten}\,(((a_{111}, a_{112}), (a_{121}, a_{122})), ((a_{211}, a_{212}), (a_{221}, a_{222}))) = \\ &\quad (a_{111}, a_{112}, a_{121}, a_{122}, a_{211}, a_{212}, a_{221}, a_{222})\end{aligned}$$

  then

$$\begin{tikzcd}
\mathsf{PTree}\,A \arrow[r, "\mathsf{flatten}\{A\}"] \arrow[d, "\mathsf{map}_{\mathsf{PTree}}\,f"'] & \mathsf{List}\,A \arrow[d, "\mathsf{map}_{\mathsf{List}}\,f"] \\
\mathsf{PTree}\,B \arrow[r, "\mathsf{flatten}\{B\}"] & \mathsf{List}\,B
\end{tikzcd}$$

$$\begin{tikzcd}
((a_{11}, a_{12}), (a_{21}, a_{22})) \arrow[r, mapsto, "\mathsf{flatten}\{A\}"] \arrow[d, mapsto, "\mathsf{map}_{\mathsf{PTree}}\,f"'] & (a_{11}, a_{12}, a_{21}, a_{22}) \arrow[d, mapsto, "\mathsf{map}_{\mathsf{List}}\,f"] \\
((fa_{11}, fa_{12}), (fa_{21}, fa_{22})) \arrow[r, mapsto, "\mathsf{flatten}\{B\}"] & (fa_{11}, fa_{12}, fa_{21}, fa_{22})
\end{tikzcd}$$

- This can be proved as a consequence of parametricity, but it really derives from the interpretation of ADTs and nested types as fixpoints of higher-order functors.

Naturality Results for ADTs and Nested Types (II)

- The naturality square for (the interpretation of) a polymorphic function says that it doesn't matter in which order we apply the function and the map operations.
- If the polymorphic function flatten : $\forall\{A : \mathsf{Set}\} \to \mathsf{PTree}\,A \to \mathsf{List}\,A$ acts like this

$$\begin{aligned}&\mathsf{flatten}\,(((a_{111}, a_{112}), (a_{121}, a_{122})), ((a_{211}, a_{212}), (a_{221}, a_{222}))) = \\ &\quad (a_{111}, a_{112}, a_{121}, a_{122}, a_{211}, a_{212}, a_{221}, a_{222})\end{aligned}$$

then

$$\begin{tikzcd}
\mathsf{PTree}\,A \arrow[r, "\mathsf{flatten}\{A\}"] \arrow[d, "\mathsf{map}_{\mathsf{PTree}}\,f"'] & \mathsf{List}\,A \arrow[d, "\mathsf{map}_{\mathsf{List}}\,f"] \\
\mathsf{PTree}\,B \arrow[r, "\mathsf{flatten}\{B\}"] & \mathsf{List}\,B
\end{tikzcd}$$

$$\begin{tikzcd}
((a_{11}, a_{12}), (a_{21}, a_{22})) \arrow[r, mapsto, "\mathsf{flatten}\{A\}"] \arrow[d, mapsto, "\mathsf{map}_{\mathsf{PTree}}\,f"'] & (a_{11}, a_{12}, a_{21}, a_{22}) \arrow[d, mapsto, "\mathsf{map}_{\mathsf{List}}\,f"] \\
((f a_{11}, f a_{12}), (f a_{21}, f a_{22})) \arrow[r, mapsto, "\mathsf{flatten}\{B\}"] & (f a_{11}, f a_{12}, f a_{21}, f a_{22})
\end{tikzcd}$$

- This can be proved as a consequence of parametricity, but it really derives from the interpretation of ADTs and nested types as fixpoints of higher-order functors.

Naturality Results for ADTs and Nested Types (II)

- The naturality square for (the interpretation of) a polymorphic function says that it doesn't matter in which order we apply the function and the map operations.
- If the polymorphic function $\mathsf{flatten} : \forall\{A : \mathsf{Set}\} \to \mathsf{PTree}\,A \to \mathsf{List}\,A$ acts like this

$$\begin{aligned}&\mathsf{flatten}\,(((a_{111}, a_{112}), (a_{121}, a_{122})), ((a_{211}, a_{212}), (a_{221}, a_{222}))) = \\ &\quad (a_{111}, a_{112}, a_{121}, a_{122}, a_{211}, a_{212}, a_{221}, a_{222})\end{aligned}$$

  then

$$\begin{tikzcd} \mathsf{PTree}\,A \arrow[r, "\mathsf{flatten}\{A\}"] \arrow[d, "\mathsf{map}_{\mathsf{PTree}}\,f"'] & \mathsf{List}\,A \arrow[d, "\mathsf{map}_{\mathsf{List}}\,f"] \\ \mathsf{PTree}\,B \arrow[r, "\mathsf{flatten}\{B\}"] & \mathsf{List}\,B \end{tikzcd}$$

$$\begin{tikzcd} ((a_{11}, a_{12}), (a_{21}, a_{22})) \arrow[r, mapsto, "\mathsf{flatten}\{A\}"] \arrow[d, mapsto, "\mathsf{map}_{\mathsf{PTree}}\,f"'] & (a_{11}, a_{12}, a_{21}, a_{22}) \arrow[d, mapsto, "\mathsf{map}_{\mathsf{List}}\,f"] \\ ((fa_{11}, fa_{12}), (fa_{21}, fa_{22})) \arrow[r, mapsto, "\mathsf{flatten}\{B\}"] & (fa_{11}, fa_{12}, fa_{21}, fa_{22}) \end{tikzcd}$$

- This can be proved as a consequence of parametricity, but it really derives from the interpretation of ADTs and nested types as fixpoints of higher-order functors.

Naturality Results for ADTs and Nested Types (II)

- The naturality square for (the interpretation of) a polymorphic function says that it doesn't matter in which order we apply the function and the map operations.
- If the polymorphic function flatten : $\forall\{A : \mathsf{Set}\} \to \mathsf{PTree}\,A \to \mathsf{List}\,A$ acts like this

$$\begin{aligned}&\mathsf{flatten}\,(((a_{111}, a_{112}), (a_{121}, a_{122})), ((a_{211}, a_{212}), (a_{221}, a_{222}))) = \\ &\quad (a_{111}, a_{112}, a_{121}, a_{122}, a_{211}, a_{212}, a_{221}, a_{222})\end{aligned}$$

  then

$$\begin{tikzcd}
\mathsf{PTree}\,A \arrow[r, "\mathsf{flatten}\{A\}"] \arrow[d, "\mathsf{map}_{\mathsf{PTree}}\,f"'] & \mathsf{List}\,A \arrow[d, "\mathsf{map}_{\mathsf{List}}\,f"] \\
\mathsf{PTree}\,B \arrow[r, "\mathsf{flatten}\{B\}"] & \mathsf{List}\,B
\end{tikzcd}$$

$$\begin{tikzcd}
((a_{11}, a_{12}), (a_{21}, a_{22})) \arrow[r, mapsto, "\mathsf{flatten}\{A\}"] \arrow[d, mapsto, "\mathsf{map}_{\mathsf{PTree}}\,f"'] & (a_{11}, a_{12}, a_{21}, a_{22}) \arrow[d, mapsto, "\mathsf{map}_{\mathsf{List}}\,f"] \\
((fa_{11}, fa_{12}), (fa_{21}, fa_{22})) \arrow[r, mapsto, "\mathsf{flatten}\{B\}"] & (fa_{11}, fa_{12}, fa_{21}, fa_{22})
\end{tikzcd}$$

- This can be proved as a consequence of parametricity, but it really derives from the interpretation of ADTs and nested types as fixpoints of higher-order functors.

Naturality Results for ADTs and Nested Types (II)

- The naturality square for (the interpretation of) a polymorphic function says that it doesn't matter in which order we apply the function and the map operations.
- If the polymorphic function flatten : $\forall\{A : \mathsf{Set}\} \to \mathsf{PTree}\ A \to \mathsf{List}\ A$ acts like this

$$\begin{aligned}&\mathsf{flatten}\,(((a_{111}, a_{112}), (a_{121}, a_{122})), ((a_{211}, a_{212}), (a_{221}, a_{222}))) =\\ &\quad (a_{111}, a_{112}, a_{121}, a_{122}, a_{211}, a_{212}, a_{221}, a_{222})\end{aligned}$$

  then

$$\begin{tikzcd} \mathsf{PTree}\ A \arrow[r, "\mathsf{flatten}\{A\}"] \arrow[d, "\mathsf{map}_{\mathsf{PTree}}\ f"'] & \mathsf{List}\ A \arrow[d, "\mathsf{map}_{\mathsf{List}}\ f"] \\ \mathsf{PTree}\ B \arrow[r, "\mathsf{flatten}\{B\}"] & \mathsf{List}\ B \end{tikzcd}$$

$$\begin{tikzcd} ((a_{11}, a_{12}), (a_{21}, a_{22})) \arrow[r, mapsto, "\mathsf{flatten}\{A\}"] \arrow[d, mapsto, "\mathsf{map}_{\mathsf{PTree}}\ f"'] & (a_{11}, a_{12}, a_{21}, a_{22}) \arrow[d, mapsto, "\mathsf{map}_{\mathsf{List}}\ f"] \\ ((fa_{11}, fa_{12}), (fa_{21}, fa_{22})) \arrow[r, mapsto, "\mathsf{flatten}\{B\}"] & (fa_{11}, fa_{12}, fa_{21}, fa_{22}) \end{tikzcd}$$

- This can be proved as a consequence of parametricity, but it really derives from the interpretation of ADTs and nested types as fixpoints of higher-order functors.

Summary

- Initial algebra semantics gives all of the above gives programming kit — maps, computation rules for polymorphic functions, folds (stylized recursion operators) — that we can use to program with, and reason about, ADTs and nested types.
- Next time we'll introduce GADTs and their semantics, and we'll see that this is where things start getting trickier (but also more enlightening!)

Summary

- Initial algebra semantics gives all of the above gives programming kit — maps, computation rules for polymorphic functions, folds (stylized recursion operators) — that we can use to program with, and reason about, ADTs and nested types.
- Next time we'll introduce GADTs and their semantics, and we'll see that this is where things start getting trickier (but also more enlightening!)