Semantics of Advanced Data Types

Patricia Johann Appalachian State University

June 14, 2021

- 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:



- 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?

- 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:



- 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?

- 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:



- 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?

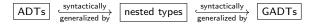
- A language's type system allows us to express correctness properties of its programs. ("Well-typed programs don't go wrong.")
- Data types are an important part of any type system.
- Fancier data types let us express more sophisticated correctness properties.
- Type-checking provides guarantees of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

In this course:



- 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?

- 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:



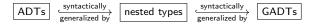
- We ask (and answer!)
 - What correctness properties can each class of data types express?
 - What models can we build to understand each class of data types?
 - What do properties of these models say about how we can compute with, and reason about, programs involving each class of data types?

- 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:



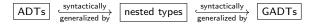
- We ask (and answer!)
 - What correctness properties can each class of data types express?
 - What models can we build to understand each class of data types?
 - What do properties of these models say about how we can compute with, and reason about, programs involving each class of data types?

- 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:



- 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?

- 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:



- 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?

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

Lecture 2: Syntax and semantics of GADTs

Lecture 3: Parametricity for ADTs and nested types

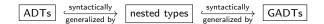
Lecture 4: Parametricity for GADTs

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



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

Booleans

data Bool : Set where false : Bool true : Bool

Natural numbers

data Nat : Set where zero : Nat suc : Nat \rightarrow Nat

Lists

data List (A : Set) : Set where [] : List A \therefore :: A \rightarrow List A \rightarrow List A

Binary trees

data Tree (A : Set) (B : Set) : Set where leaf : A \rightarrow Tree A B node : Tree A B \rightarrow B \rightarrow Tree A B \rightarrow Tree A B

Booleans

data Bool : Set where false : Bool true : Bool

• Natural numbers

data Nat : Set where zero : Nat suc : Nat \rightarrow Nat

Lists

data List (A : Set) : Set where [] : List A $_$::_ : A \rightarrow List A \rightarrow List A

Binary trees

data Tree (A : Set) (B : Set) : Set where leaf : A \rightarrow Tree A B node : Tree A B \rightarrow B \rightarrow Tree A B \rightarrow Tree A B

Booleans

data Bool : Set where false : Bool true : Bool

• Natural numbers

data Nat : Set where zero : Nat suc : Nat \rightarrow Nat

Lists

data List (A : Set) : Set where [] : List A _::_: : A \rightarrow List A \rightarrow List A

Binary trees

data Tree (A : Set) (B : Set) : Set where leaf : A \rightarrow Tree A B node : Tree A B \rightarrow B \rightarrow Tree A B \rightarrow Tree A B

Booleans

data Bool : Set where false : Bool true : Bool

Natural numbers

data Nat : Set where zero : Nat suc : Nat \rightarrow Nat

Lists

data List (A : Set) : Set where [] : List A _::_: A \rightarrow List A \rightarrow List A

Binary trees

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

- 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

• Agda also imposes a strict positivity requirement on the types of $c_1,...,c_k\colon$ Either - T_{ij} is not inductive and does not mention D

or

- T_{ii} is inductive and has the form

$$\mathsf{C}_1 \to ... \to \mathsf{C}_\mathsf{p} \to \mathsf{D}\,\mathsf{A}_1 ... \,\mathsf{A}_\mathsf{n}$$

where D does not occur in any C_i .

Strict positivity

- 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} \text{data } D A_1 \ldots A_n : \text{Set where} \\ c_1 : T_{11} \rightarrow \ldots \rightarrow T_{1j_1} \rightarrow D A_1 \ldots A_n \\ \ldots \\ c_k : T_{k1} \rightarrow \ldots \rightarrow T_{kj_k} \rightarrow D A_1 \ldots A_n \end{array}
```

• Agda also imposes a strict positivity requirement on the types of $c_1,...,c_k\colon$ Either - T_{ij} is not inductive and does not mention D

or

- T_{ii} is inductive and has the form

$$\mathsf{C}_1 \to ... \to \mathsf{C}_\mathsf{p} \to \mathsf{D}\,\mathsf{A}_1 ... \,\mathsf{A}_\mathsf{n}$$

where D does not occur in any C_i .

Strict positivity

- 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

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

• Agda also imposes a strict positivity requirement on the types of $c_1,...,c_k\colon$ Either - T_{ij} is not inductive and does not mention D

or

- T_{ii} is inductive and has the form

$$C_1 \rightarrow ... \rightarrow C_p \rightarrow D A_1 ... A_n$$

where D does not occur in any C_i .

Strict positivity

- 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} \mbox{data } D \mbox{ } A_1 \ldots A_n : \mbox{Set where} \\ c_1 : T_{11} \rightarrow \ldots \rightarrow T_{1j_1} \rightarrow D \mbox{ } A_1 \ldots A_n \\ \ldots \\ c_k : T_{k1} \rightarrow \ldots \rightarrow T_{kj_k} \rightarrow D \mbox{ } A_1 \ldots A_n \end{array}
```

• Agda also imposes a strict positivity requirement on the types of $c_1,...,c_k\colon$ Either - T_{ij} is not inductive and does not mention D

or

- T_{ij} is inductive and has the form

$$\mathsf{C}_1 \to ... \to \mathsf{C}_p \to \mathsf{D}\,\mathsf{A}_1 ... \,\mathsf{A}_n$$

where D does not occur in any C_i .

Strict posit

- 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} \mbox{data } D \mbox{ } A_1 \ldots A_n : \mbox{Set where} \\ c_1 : T_{11} \rightarrow \ldots \rightarrow T_{1j_1} \rightarrow D \mbox{ } A_1 \ldots A_n \\ \ldots \\ c_k : T_{k1} \rightarrow \ldots \rightarrow T_{kj_k} \rightarrow D \mbox{ } A_1 \ldots A_n \end{array}
```

• Agda also imposes a strict positivity requirement on the types of $c_1,...,c_k\colon$ Either - T_{ij} is not inductive and does not mention D

or

- T_{ij} is inductive and has the form

$$\mathsf{C}_1 \to ... \to \mathsf{C}_\mathsf{p} \to \mathsf{D}\,\mathsf{A}_1 ... \,\mathsf{A}_\mathsf{n}$$

where D does not occur in any C_i .

Strict positivity

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

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

- A category ${\mathcal C}$ comprises
 - a class $\mathit{ob}(\mathcal{C})$ 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 $id_X \in Hom_{\mathcal{C}}(X, X)$

- 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.

- 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 $id_X \in Hom_{\mathcal{C}}(X, X)$

- 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.

- A category ${\mathcal C}$ comprises
 - a class $\mathit{ob}(\mathcal{C})$ 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 $id_X \in Hom_{\mathcal{C}}(X, X)$

- 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.

- A category ${\mathcal C}$ comprises
 - a class $\mathit{ob}(\mathcal{C})$ 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 : C rather than $X \in ob(C)$ and $f : X \to Y$ rather than $f \in Hom_{\mathcal{C}}(X, Y)$.

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

- A category ${\mathcal C}$ comprises
 - a class $\mathit{ob}(\mathcal{C})$ 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 $id_X \in Hom_{\mathcal{C}}(X, X)$

- 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 : C rather than $X \in ob(C)$ and $f : X \to Y$ rather than $f \in Hom_{\mathcal{C}}(X, Y)$.
- We will restrict attention to the category Set for now.

• 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:

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

• 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:

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

• 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 D = FD, where F denotes the underlying functor F for D, then we interpret D as μF .

- 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 D = FD, where F denotes the underlying functor F for D, then we interpret D as μF .

- 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 D = FD, where F denotes the underlying functor F for D, then we interpret D as μF .

- 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 D = FD, where F denotes the underlying functor F for D, then we interpret D as μF .

Transfinite Construction of Free Algebras (Kelly'80)

• If

C is a locally λ -presentable category interpreting types, 0 is the initial object of C,

and

 $F:\mathcal{C}\rightarrow\mathcal{C}$ is a $\lambda\text{-}\mathrm{cocontinuous}$ functor

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

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

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

For concreteness, take C to be Set and F to be polynomial.

Transfinite Construction of Free Algebras (Kelly'80)

• If

C is a locally λ -presentable category interpreting types, 0 is the initial object of C.

and

 $F:\mathcal{C}\rightarrow\mathcal{C}$ is a $\lambda\text{-}\mathrm{cocontinuous}$ functor

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

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

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

For concreteness, take C to be Set and F to be polynomial.

Transfinite Construction of Free Algebras (Kelly'80)

• If

C is a locally λ -presentable category interpreting types, 0 is the initial object of C.

and

 $F:\mathcal{C}\rightarrow\mathcal{C}$ is a $\lambda\text{-}\mathrm{cocontinuous}$ functor

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

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

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

For concreteness, take C to be Set and F to be polynomial.

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

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 \rightarrow 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 \therefore $A \rightarrow I$ ist $A \rightarrow I$ ist A has $FX = 1 + A \times X$, so List, A is interpreted as μF , i.e., as $\mu X.1 + A \times X$

data Bool : Set where false : Bool true : Bool has FX = 1 + 1, so Bool is interpreted as μF , i.e., as $\mu X.1 + 1$ data Nat : Set where zero : Nat suc : Nat \rightarrow 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 \therefore $A \rightarrow I$ ist $A \rightarrow I$ ist A has $FX = 1 + A \times X$, so List, A is interpreted as μF , i.e., as $\mu X.1 + A \times X$ data Tree (A : Set) (B : Set) : Set where $\mathsf{leaf} \ : \mathsf{A} \to \mathsf{Tree}\,\mathsf{A}\,\mathsf{B}$ node : Tree A B \rightarrow B \rightarrow Tree A B \rightarrow Tree A B has $F X = A + X \times B \times X$, so Tree A B is interpreted as μF , i.e., as $\mu X.A + X \times B \times X$

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

Perfect trees

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

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

Bushes

data Bush : Set \rightarrow Set where bnil : \forall {A : Set} \rightarrow Bush A bnode : \forall {A : Set} \rightarrow A \rightarrow Bush (Bush A) \rightarrow Bush 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).

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

Perfect trees

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

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

Bushes

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

- 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).

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

Perfect trees

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

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

Bushes

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

- 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).

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

Perfect trees

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

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

- 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).

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

Perfect trees

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

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

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).

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

Perfect trees

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

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

Bushes

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

- 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).

The general form of a nested type is

$$\begin{array}{l} \mathsf{data} \ \mathsf{D} \ \mathsf{A}_1 \ldots \mathsf{A}_n : \mathsf{B}_1 \rightarrow \ldots \rightarrow \mathsf{B}_m \rightarrow \mathsf{Set} \ \mathsf{where} \\ \mathsf{c}_1 : \forall \{\mathsf{A}_1 \ldots \mathsf{A}_n \ \mathsf{B}_1 \ldots \mathsf{B}_m\} \rightarrow \mathsf{T}_{11} \rightarrow \ldots \rightarrow \mathsf{T}_{1j_1} \rightarrow \mathsf{D} \ \mathsf{A}_1 \ldots \mathsf{A}_n \ \mathsf{B}_1 \ldots \mathsf{B}_m \\ \\ \ldots \\ \mathsf{c}_k : \forall \{\mathsf{A}_1 \ldots \mathsf{A}_n \ \mathsf{B}_1 \ldots \mathsf{B}_m\} \rightarrow \mathsf{T}_{k1} \rightarrow \ldots \rightarrow \mathsf{T}_{kj_k} \rightarrow \mathsf{D} \ \mathsf{A}_1 \ldots \mathsf{A}_n \ \mathsf{B}_1 \ldots \mathsf{B}_m \end{array}$$

where either

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

or

 T_{ii} is inductive and has the form

$$C_1 \rightarrow ... \rightarrow C_p \rightarrow D\,A_1 ... A_n\,V_1 ... V_m$$

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

Strict positivity

 \implies no negative occurrences of D in argument types of constructors

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

The general form of a nested type is

$$\begin{array}{l} \mathsf{data} \ \mathsf{D} \ \mathsf{A}_1 \ldots \mathsf{A}_n : \mathsf{B}_1 \rightarrow \ldots \rightarrow \mathsf{B}_m \rightarrow \mathsf{Set} \ \mathsf{where} \\ \mathsf{c}_1 : \forall \{\mathsf{A}_1 \ldots \mathsf{A}_n \ \mathsf{B}_1 \ldots \mathsf{B}_m\} \rightarrow \mathsf{T}_{11} \rightarrow \ldots \rightarrow \mathsf{T}_{1j_1} \rightarrow \mathsf{D} \ \mathsf{A}_1 \ldots \mathsf{A}_n \ \mathsf{B}_1 \ldots \mathsf{B}_m \\ \\ \ldots \\ \mathsf{c}_k : \forall \{\mathsf{A}_1 \ldots \mathsf{A}_n \ \mathsf{B}_1 \ldots \mathsf{B}_m\} \rightarrow \mathsf{T}_{k1} \rightarrow \ldots \rightarrow \mathsf{T}_{kj_k} \rightarrow \mathsf{D} \ \mathsf{A}_1 \ldots \mathsf{A}_n \ \mathsf{B}_1 \ldots \mathsf{B}_m \end{array}$$

where either

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

or

 T_{ij} is inductive and has the form

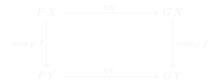
$$\mathsf{C}_1 \to ... \to \mathsf{C}_\mathsf{p} \to \mathsf{D}\,\mathsf{A}_1 \ldots \mathsf{A}_\mathsf{n}\,\mathsf{V}_1 \ldots \mathsf{V}_\mathsf{m}$$

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

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

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

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



- 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

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

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

- If C and D are categories, then the collection of functors from C to D also form a category. Its objects are functors from C to 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 $\eta_Y \circ map_F f = map_G f \circ \eta_X$



- 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

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

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

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

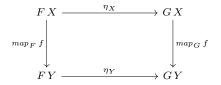


- The identity on F is the identity natural transformation id_F from F to F.
- Composition of natural transformations is componentwise, i.e., if $X : \mathcal{C}$ then

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

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

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



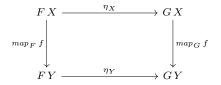
- The identity on F is the identity natural transformation id_F from F to F.
- Composition of natural transformations is componentwise, i.e., if X : C then

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

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

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

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

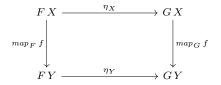


• The identity on F is the identity natural transformation id_F from F to F.

• Composition of natural transformations is componentwise, i.e., if $X:\mathcal{C}$ then $(\eta\circ\mu)_X=\eta_X\circ\mu_X$

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

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



- 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

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

- 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.

- 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.

- 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.

- 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 $\operatorname{map}_H \eta: HF \to HG$
- *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.

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

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

data Bush : Set \rightarrow Set where bnil : $\forall \{A : Set\} \rightarrow Bush A$ bnode : $\forall \{A : Set\} \rightarrow A \rightarrow Bush (Bush A) \rightarrow Bush A$

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

```
\begin{array}{l} \mathsf{data}\;\mathsf{PTree}:\mathsf{Set}\to\mathsf{Set}\;\mathsf{where}\\ \mathsf{pleaf}\;:\forall\{\mathsf{A}:\mathsf{Set}\}\to\mathsf{A}\to\mathsf{PTree}\;\mathsf{A}\\ \mathsf{pnode}:\forall\{\mathsf{A}:\mathsf{Set}\}\to\mathsf{PTree}(\mathsf{A}\times\mathsf{A})\to\mathsf{PTree}\;\mathsf{A}\\ \mathsf{has}\;H\,F\,X=X+F\,(X\times X),\;\mathsf{so}\;\mathsf{PTree}\;\mathsf{is}\;\mathsf{interpreted}\;\mathsf{as}\;\mu H,\\ \mathsf{i.e.,}\;\mathsf{as}\;\mu F.\,\lambda X.\;X+F\,(X\times X)\\ \mathsf{data}\;\mathsf{Bush}:\mathsf{Set}\to\mathsf{Set}\;\mathsf{where}\\ \mathsf{bnil}\;\;:\forall\{\mathsf{A}:\mathsf{Set}\}\to\mathsf{Bush}\;\mathsf{A}\\ \mathsf{bnode}:\forall\{\mathsf{A}:\mathsf{Set}\}\to\mathsf{A}\to\mathsf{Bush}\;(\mathsf{Bush}\;\mathsf{A})\to\mathsf{Bush}\;\mathsf{A}\\ \mathsf{bnode}:\forall\{\mathsf{A}:\mathsf{Set}\}\to\mathsf{A}\to\mathsf{Bush}\;(\mathsf{Bush}\;\mathsf{A})\to\mathsf{Bush}\;\mathsf{A}\\ \mathsf{bnode}:\forall\{\mathsf{A}:\mathsf{Set}\}\to\mathsf{A}\to\mathsf{Bush}\;(\mathsf{Bush}\;\mathsf{A})\to\mathsf{Bush}\;\mathsf{A}\\ \mathsf{has}\;H\,F\,X=1+X\times F\,(F\,X),\;\mathsf{so}\;\mathsf{Bush}\;\mathsf{is}\;\mathsf{interpreted}\;\mathsf{as}\;\mu H,\\ \mathsf{i.e.,}\;\mathsf{as}\;\mu F.\,\lambda X.\;1+X\times F\,(F\,X)\\ \end{array}
```

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 \rightarrow List A \rightarrow List A

is also

data List : Set
$$\rightarrow$$
 Set where
[] : $\forall \{A : Set\} \rightarrow List A$
:: : $\forall \{A : Set\} \rightarrow A \rightarrow List A \rightarrow List A$

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

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

\therefore :: A \rightarrow List A \rightarrow List A
```

is also

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

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

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 \rightarrow List A \rightarrow List A

is also

•

$$\begin{array}{ll} \mathsf{data}\ \mathsf{List}\ \colon \mathsf{Set}\ \to\ \mathsf{Set}\ \mathsf{where}\\ [] & :\ \forall\{\mathsf{A}:\mathsf{Set}\}\ \to\ \mathsf{List}\ \mathsf{A}\\ _\ \colon \ \colon\ \forall\{\mathsf{A}:\mathsf{Set}\}\ \to\ \mathsf{A}\ \to\ \mathsf{List}\ \mathsf{A}\ \to\ \mathsf{List}\ \mathsf{A} \end{array}$$

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

- 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_{µH}. D has a corresponding function map_D that is just the reflection back into syntax of map_{µH}.

```
\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}\,[] &= []\\ \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}) &= \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}{ll} \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} &=\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}
```

- 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 $map_{\mu H}$.

- 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}{ll} \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}\,[]&=[]\\ \mathsf{map}_{\mathsf{List}}\,\mathsf{f}\,(x::xs)=(\mathsf{f}\,x)::\mathsf{map}_{\mathsf{List}}\,\mathsf{f}\,xs \end{array}
```

```
\begin{split} \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}) &= \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{split}
```

```
\begin{array}{ll} \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}&=\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}
```

- 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 $map_{\mu H}$.

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

```
\begin{array}{ll} \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}) &= \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}{ll} \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} &=\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}
```

- 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 $map_{\mu H}$.

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

```
\begin{array}{ll} \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}) &= \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}{ll} \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} &= \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{f}) \ \mathsf{bb}) \end{array}
```

- 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 $map_{\mu H}$.

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

```
\begin{array}{ll} \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}) &= \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}
```

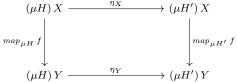
```
\begin{array}{ll} \mathsf{map}_{\mathsf{Bush}} :: (\mathsf{A} \to \mathsf{B}) \to \ \mathsf{Bush} \ \mathsf{A} \to \mathsf{Bush} \ \mathsf{B} \\ \mathsf{map}_{\mathsf{Bush}} \ \mathsf{f} \ \mathsf{bnil} & = \mathsf{bnil} \\ \mathsf{map}_{\mathsf{Bush}} \ \mathsf{f} \ (\mathsf{bnode} \ \mathsf{a} \ \mathsf{bb}) & = \mathsf{bnode} \ (\mathsf{f} \ \mathsf{a}) \ (\mathsf{map}_{\mathsf{Bush}} \ \mathsf{f} \ \mathsf{map}_{\mathsf{Bush}} \ \mathsf{f}) \ \mathsf{bb}) \end{array}
```

- 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



- 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.

- 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



• 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 $\mu H'$.

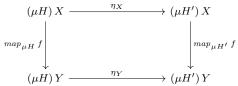
- A polymorphic function (natural transformation) between (interpretations of) data types alters the shapes of data structures without changing their data elements.
- So natural transformations do the "opposite" of map functions, which act on data elements without changing the shape of the data structure in which they reside.

- 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



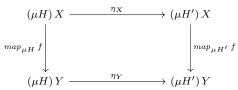
- Computationally (i.e., reflecting back into syntax), we can think of natural transformations as polymorphic functions between data types whose constructors are interpreted as μH and $\mu H'$.
- A polymorphic function (natural transformation) between (interpretations of) data types alters the shapes 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.

- 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



- 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.

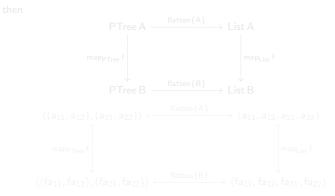
- Just as their interpretations as fixpoints of higher-order functors give map functions for ADTs and nested types, these interpretations also give naturality results.
- A natural transformation $\eta: \mu H \to \mu H'$ gives commuting squares: if $f: X \to Y,$ then



- 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.

- 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 : $\forall \{A : Set\} \rightarrow \mathsf{PTree} A \rightarrow \mathsf{List} A$ acts like this

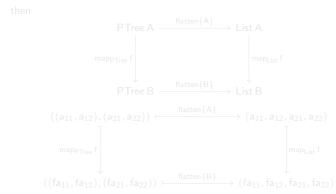
 $\begin{array}{l} {\rm flatten}\left(((a_{111},a_{112}),(a_{121},a_{122})),((a_{211},a_{212}),(a_{221},a_{222}))\right)=\\ (a_{111},a_{112},a_{121},a_{122},a_{211},a_{212},a_{221},a_{222}) \end{array}$



 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.

- 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{array}{l} \text{flatten} \left( ((a_{111}, a_{112}), (a_{121}, a_{122})), ((a_{211}, a_{212}), (a_{221}, a_{222})) \right) = \\ (a_{111}, a_{112}, a_{121}, a_{122}, a_{211}, a_{212}, a_{221}, a_{222}) \end{array}
```

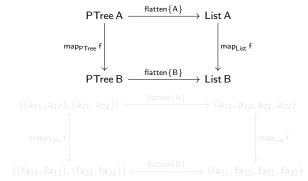


• This can be proved as a consequence of parametricity, but it really derives from the interpretation of ADTs and nested types as fixpoints of higher-order functors.

- 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

$$\begin{array}{l} \mathsf{flatten}\left(((\mathsf{a}_{111},\mathsf{a}_{112}),(\mathsf{a}_{121},\mathsf{a}_{122})),((\mathsf{a}_{211},\mathsf{a}_{212}),(\mathsf{a}_{221},\mathsf{a}_{222}))\right) = \\ (\mathsf{a}_{111},\mathsf{a}_{112},\mathsf{a}_{121},\mathsf{a}_{122},\mathsf{a}_{211},\mathsf{a}_{212},\mathsf{a}_{221},\mathsf{a}_{222}) \end{array}$$

then

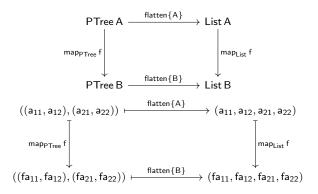


• This can be proved as a consequence of parametricity, but it really derives from the interpretation of ADTs and nested types as fixpoints of higher-order functors.

- 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

$$\begin{array}{l} {\sf flatten}\left((({\sf a}_{111},{\sf a}_{112}),({\sf a}_{121},{\sf a}_{122})),(({\sf a}_{211},{\sf a}_{212}),({\sf a}_{221},{\sf a}_{222}))\right)=\\ ({\sf a}_{111},{\sf a}_{112},{\sf a}_{121},{\sf a}_{122},{\sf a}_{211},{\sf a}_{212},{\sf a}_{221},{\sf a}_{222}) \end{array}$$

then

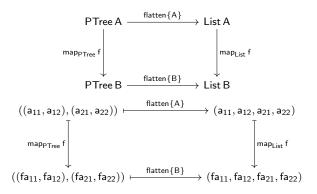


 This can be proved as a consequence of parametricity, but it really derives from the interpretation of ADTs and nested types as fixpoints of higher-order functors.

- 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

$$\begin{array}{l} {\sf flatten}\left((({\sf a}_{111},{\sf a}_{112}),({\sf a}_{121},{\sf a}_{122})),(({\sf a}_{211},{\sf a}_{212}),({\sf a}_{221},{\sf a}_{222}))\right) = \\ ({\sf a}_{111},{\sf a}_{112},{\sf a}_{121},{\sf a}_{122},{\sf a}_{211},{\sf a}_{212},{\sf a}_{221},{\sf a}_{222}) \end{array}$$

then



 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!)