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:

- 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?

ADTs ➔ syntactically generalized by nested types ➔ syntactically generalized by GADTs
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:

- ADTs syntactically generalized by nested types syntactically generalized by 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:

- 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:

  ADTs (syntactically generalized by) nested types (syntactically generalized by) 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:

  ADTs ← syntactically generalized by nested types ← syntactically generalized by 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:

- 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:

  \[
  \begin{array}{c}
  \text{ADTs} \xrightarrow{\text{syntactically}} \text{nested types} \xrightarrow{\text{syntactically}} \text{GADTs}
  \end{array}
  \]

- 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:
  
  \[
  \text{ADTs} \quad \xrightarrow{\text{syntactically generalized by}} \quad \text{nested types} \quad \xrightarrow{\text{syntactically generalized by}} \quad \text{GADTs}
  \]

• 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

Assumption: Basic familiarity with categories, functors, natural transformations.
Syntax of ADTs (I)

- **Booleans**
  
  ```haskell
data Bool : Set where
    false : Bool
    true : Bool
  ```

- **Natural numbers**
  
  ```haskell
data Nat : Set where
    zero : Nat
    suc : Nat → Nat
  ```

- **Lists**
  
  ```haskell
data List (A : Set) : Set where
    [] : List A
    _ :: _ : A → List A → List A
  ```

- **Binary trees**
  
  ```haskell
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

\[
data \text{ Bool} : \text{ Set where}
\]
\[
\text{false} : \text{ Bool} \\
\text{true} : \text{ Bool}
\]

• Natural numbers

\[
data \text{ Nat} : \text{ Set where}
\]
\[
\text{zero} : \text{ Nat} \\
\text{suc} : \text{ Nat} \rightarrow \text{ Nat}
\]

• Lists

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

• Binary trees

\[
data \text{ Tree} (A : \text{ Set}) (B : \text{ Set}) : \text{ Set where}
\]
\[
\text{leaf} : A \rightarrow \text{ Tree } A B \\
\text{node} : \text{ Tree } A B \rightarrow B \rightarrow \text{ Tree } A B \rightarrow \text{ Tree } A B
\]
Syntax of ADTs (I)

• Booleans

\[
data \text{Bool} : \text{Set} \text{ where } \\
\text{false} : \text{Bool} \\
\text{true} : \text{Bool}
\]

• Natural numbers

\[
data \text{Nat} : \text{Set} \text{ where } \\
\text{zero} : \text{Nat} \\
\text{suc} : \text{Nat} \rightarrow \text{Nat}
\]

• Lists

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

• Binary trees

\[
data \text{Tree} (\text{A} : \text{Set}) (\text{B} : \text{Set}) : \text{Set} \text{ where } \\
\text{leaf} : \text{A} \rightarrow \text{Tree A B} \\
\text{node} : \text{Tree A B} \rightarrow \text{B} \rightarrow \text{Tree A B} \rightarrow \text{Tree A B}
\]
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 (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

\[
\text{data } D\ A_1 \ldots A_n : \text{Set where}
\]
\[
c_1 : T_{11} \rightarrow \ldots \rightarrow T_{1j_1} \rightarrow D\ A_1 \ldots A_n \\
\ldots
\]
\[
c_k : T_{k1} \rightarrow \ldots \rightarrow T_{kj_k} \rightarrow D\ A_1 \ldots A_n
\]

• 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

\[
C_1 \rightarrow \ldots \rightarrow C_p \rightarrow D\ A_1 \ldots A_n
\]

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.
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

  \[
  \text{data } D \ A_1 \ldots A_n : \text{Set where}
  \begin{align*}
  c_1 & : T_{11} \to \ldots \to T_{1j_1} \to D \ A_1 \ldots A_n \\
  \vdots & \\
  c_k & : T_{k1} \to \ldots \to T_{kj_k} \to D \ A_1 \ldots A_n
  \end{align*}
  \]

- 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
  \[\implies\] no negative occurrences of \(D\) in the argument types of its constructors
  \[\implies\] \(D\) can be interpreted as the least fixpoint of a functor.
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

\[
\text{data } D A_1 \ldots A_n : \text{Set where}
\]
\[
c_1 : T_{11} \rightarrow \ldots \rightarrow T_{1j_1} \rightarrow D A_1 \ldots A_n
\]
\[
\ldots
\]
\[
c_k : T_{k1} \rightarrow \ldots \rightarrow T_{kj_k} \rightarrow D A_1 \ldots A_n
\]

• 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

    \[
    C_1 \rightarrow \ldots \rightarrow C_p \rightarrow D A_1 \ldots A_n
    \]

    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.
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

\[
data D A_1 ... A_n : \text{Set where} \\
c_1 : T_{11} \to ... \to T_{1j_1} \to D A_1 ... A_n \\
... \\
c_k : T_{k1} \to ... \to T_{kj_k} \to D A_1 ... A_n
\]

- Agda also imposes a strict positivity requirement on the types of \(c_1, ..., c_k\): Either
  - \(T_{ij}\) is not inductive and does not mention \(D\)
  or
  - \(T_{ij}\) is inductive and has the form

\[
C_1 \to ... \to C_p \to D A_1 ... A_n
\]

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.
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

```
data D A_1 ... A_n : Set where
  c_1 : T_{11} → ... → T_{1j_1} → D A_1 ... A_n
  ...
  c_k : T_{k1} → ... → T_{kj_k} → D A_1 ... A_n
```

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

```
C_1 → ... → C_p → D A_1 ... A_n
```

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

⇒⇒ no negative occurrences of D in the argument types of its constructors
⇒⇒ D can be interpreted as the least fixpoint of a functor.
A category $C$ comprises
- a class $ob(C)$ of objects
- for each $X, Y \in ob(C)$, a class $Hom_C(X, Y)$ of morphisms from $X$ to $Y$
- for each $X \in ob(C)$, an identity morphism $id_X \in Hom_C(X, X)$
- 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 $f \in Hom_C(X, Y)$.

We will restrict attention to the category $Set$ for now.
A category $C$ comprises
- a class $ob(C)$ of objects
- for each $X, Y \in ob(C)$, a class $Hom_C(X, Y)$ of morphisms from $X$ to $Y$
- for each $X \in ob(C)$, an identity morphism $id_X \in Hom_C(X, X)$
- 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 $f \in Hom_C(X, Y)$.

We will restrict attention to the category $Set$ for now.
Category Theory Interlude (I)

• A category $C$ comprises
  - a class $ob(C)$ of objects
  - for each $X, Y \in ob(C)$, a class $Hom_C(X, Y)$ of morphisms from $X$ to $Y$
  - for each $X \in ob(C)$, an identity morphism $id_X \in Hom_C(X, X)$
  - a composition operator $\circ$ assigning to each pair of morphisms $f : X \rightarrow Y$ and $g : Y \rightarrow Z$, the composite morphism $g \circ f : X \rightarrow Z$

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

• Composition is associative.
  • We write $X : C$ rather than $X \in ob(C)$ and $f : X \rightarrow Y$ rather than $f \in Hom_C(X, Y)$.
  • We will restrict attention to the category $Set$ for now.
A category $C$ comprises
- a class $ob(C)$ of objects
- for each $X, Y \in ob(C)$, a class $Hom_C(X, Y)$ of morphisms from $X$ to $Y$
- for each $X \in ob(C)$, an identity morphism $id_X \in Hom_C(X, X)$
- 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 $f \in Hom_C(X, Y)$.

We will restrict attention to the category $Set$ for now.
A category \( C \) comprises
- a class \( \text{ob}(C) \) of objects
- for each \( X, Y \in \text{ob}(C) \), a class \( \text{Hom}_C(X,Y) \) of morphisms from \( X \) to \( Y \)
- for each \( X \in \text{ob}(C) \), an identity morphism \( \text{id}_X \in \text{Hom}_C(X,X) \)
- 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 \( f \circ \text{id}_X = f = \text{id}_Y \circ f \).
- Composition is associative.
- We write \( X : C \) rather than \( X \in \text{ob}(C) \) and \( f : X \to Y \) rather than \( f \in \text{Hom}_C(X,Y) \).
- We will restrict attention to the category \( \text{Set} \) for now.
If $\mathcal{C}$ and $\mathcal{D}$ are categories, then a functor $F : \mathcal{C} \to \mathcal{D}$ comprises
- a function $F$ from $\text{ob}(\mathcal{C})$ to $\text{ob}(\mathcal{D})$, together with
- a function $map_F$ from $\text{Hom}_\mathcal{C}(X, Y)$ to $\text{Hom}_\mathcal{D}(FX, FY)$

A functor must preserve the fundamental structure of a category. This means that $map_F$ must preserve identities and composition:

$$map_F g \circ map_F f = map_F (g \circ f)$$
$$map_F id_X = id_{FX}$$
Category Theory Interlude (II)

• If $C$ and $D$ are categories, then a functor $F : C \to D$ comprises
  - a function $F$ from $ob(C)$ to $ob(D)$, together with
  - a function $map_F$ from $Hom_C(X, Y)$ to $Hom_D(FX, FY)$

• A functor must preserve the fundamental structure of a category. This means that
  $map_F$ must preserve identities and composition:

  \[
  \begin{align*}
  map_F g \circ map_F f & = map_F (g \circ f) \\
  map_F id_X & = id_{FX}
  \end{align*}
  \]
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 $\mu F$.
- If the ADT $D$ is defined by $D = F D$, where $F$ denotes the underlying functor $F$ for $D$, then we interpret $D$ as $\mu F$. 
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 $\mu F$.
- If the ADT $D$ is defined by $D = F D$, where $F$ denotes the underlying functor $F$ for $D$, then we interpret $D$ as $\mu F$. 
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 $\mu F$.
- If the ADT $D$ is defined by $D = F D$, where $F$ denotes the underlying functor $F$ for $D$, then we interpret $D$ as $\mu F$. 
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 $\mu F$.
- If the ADT $D$ is defined by $D = F D$, where $F$ denotes the underlying functor $F$ for $D$, then we interpret $D$ as $\mu F$. 
If

\[ C \text{ is a locally } \lambda\text{-presentable category interpreting types,} \]
\[ 0 \text{ is the initial object of } C, \]

and

\[ F : C \to C \text{ is a } \lambda\text{-cocontinuous functor} \]

then \( F \) has an initial algebra, and its carrier is the least fixpoint \( \mu F \) of \( F \)
computed by

\[ 0 \hookrightarrow F 0 \hookrightarrow F (F 0) \ldots \hookrightarrow F^n 0 \ldots \hookrightarrow \mu F \]

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 \( \text{Set} \) and \( F \) to be polynomial.
Transfinite Construction of Free Algebras (Kelly’80)

- If

  \( C \) is a locally \( \lambda \)-presentable category interpreting types,
  
  \( 0 \) is the initial object of \( C \),

  and

  \( F : C \to C \) is a \( \lambda \)-cocontinuous functor

  then \( F \) has an initial algebra, and its carrier is the least fixpoint \( \mu F \) 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 \( C \) to be \( Set \) and \( F \) to be polynomial.
Transfinite Construction of Free Algebras (Kelly’80)

• If

   \[ C \text{ is a locally } \lambda\text{-presentable category interpreting types,} \]
   \[ 0 \text{ is the initial object of } C, \]
   \[ \text{and} \]
   \[ F : C \to C \text{ is a } \lambda\text{-cocontinuous functor} \]

then \( F \) has an initial algebra, and its carrier is the least fixpoint \( \mu F \) of \( F \) computed by

\[
0 \hookrightarrow F 0 \hookrightarrow F (F 0) \ldots \hookrightarrow F^n 0 \ldots \hookrightarrow \mu F
\]

• 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.
Semantics of ADTs

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

  has $F X = 1 + 1$, so Bool is interpreted as $\mu F$, 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 $\mu F$, 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 $\mu F$, 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 $\mu F$, i.e., as $\mu X. A + X \times B \times X$
Semantics of ADTs

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

  has $F X = 1 + 1$, so Bool is interpreted as $\mu F$, 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 $\mu F$, 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 $\mu F$, 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 $\mu F$, i.e., as $\mu X. A + X \times B \times X$
Semantics of ADTs

\[ F(X) = 1 + 1 \]

1. **data Bool : Set where**
   
   \[
   \begin{align*}
   \text{false} & : \text{Bool} \\
   \text{true} & : \text{Bool}
   \end{align*}
   \]
   
   has \( F(X) = 1 + 1 \), so Bool is interpreted as \( \mu F \), i.e., as \( \mu X. 1 + 1 \)

2. **data Nat : Set where**
   
   \[
   \begin{align*}
   \text{zero} & : \text{Nat} \\
   \text{suc} & : \text{Nat} \rightarrow \text{Nat}
   \end{align*}
   \]
   
   has \( F(X) = 1 + X \), so Nat is interpreted as \( \mu F \), i.e., as \( \mu X. 1 + X \)

3. **data List (A : Set) : Set where**
   
   \[
   \begin{align*}
   [\ ] & : \text{List A} \\
   _ :: _ & : A \rightarrow \text{List A} \rightarrow \text{List A}
   \end{align*}
   \]
   
   has \( F(X) = 1 + A \times X \), so List, A is interpreted as \( \mu F \), i.e., as \( \mu X. 1 + A \times X \)

4. **data Tree (A : Set) (B : Set) : Set where**
   
   \[
   \begin{align*}
   \text{leaf} & : A \rightarrow \text{Tree A B} \\
   \text{node} & : \text{Tree A B} \rightarrow B \rightarrow \text{Tree A B} \rightarrow \text{Tree A B}
   \end{align*}
   \]
   
   has \( F(X) = A + X \times B \times X \), so Tree A B is interpreted as \( \mu F \), i.e., as \( \mu X. A + X \times B \times X \)
Semantics of ADTs

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

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

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

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

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

  has $FX = 1 + A \times X$, so List, A is interpreted as $\mu F$, 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 $FX = A + X \times B \times X$, so Tree A B is interpreted as $\mu F$, 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**

  \[
  \text{data } \text{PTree} : \text{Set} \rightarrow \text{Set} \text{ where} \\
  \text{pleaf} : \forall \{A : \text{Set}\} \rightarrow A \rightarrow \text{PTree } A \\
  \text{pnode} : \forall \{A : \text{Set}\} \rightarrow \text{PTree}(A \times A) \rightarrow \text{PTree } A
  \]

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

- **Bushes**

  \[
  \text{data } \text{Bush} : \text{Set} \rightarrow \text{Set} \text{ where} \\
  \text{bnil} : \forall \{A : \text{Set}\} \rightarrow \text{Bush } A \\
  \text{bnode} : \forall \{A : \text{Set}\} \rightarrow A \rightarrow \text{Bush } (\text{Bush } A) \rightarrow \text{Bush } A
  \]

  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

  \[
  \text{data PTree : Set} \rightarrow \text{Set where}
  \]

  \[
  \begin{align*}
  \text{pleaf} & : \forall \{A : \text{Set}\} \rightarrow A \rightarrow \text{PTree } A \\
  \text{pnode} & : \forall \{A : \text{Set}\} \rightarrow \text{PTree } (A \times A) \rightarrow \text{PTree } A
  \end{align*}
  \]

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

- Bushes

  \[
  \text{data Bush : Set} \rightarrow \text{Set where}
  \]

  \[
  \begin{align*}
  \text{bnil} & : \forall \{A : \text{Set}\} \rightarrow \text{Bush } A \\
  \text{bnode} & : \forall \{A : \text{Set}\} \rightarrow A \rightarrow \text{Bush } (\text{Bush } A) \rightarrow \text{Bush } A
  \end{align*}
  \]

  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**

  
  \[
  \text{data PTree : Set} \rightarrow \text{Set where} \\
  \text{pleaf : } \forall \{ \text{A : Set} \} \rightarrow \text{A} \rightarrow \text{PTree A} \\
  \text{pnode : } \forall \{ \text{A : Set} \} \rightarrow \text{PTree}(\text{A} \times \text{A}) \rightarrow \text{PTree A}
  \]

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

- **Bushes**

  
  \[
  \text{data Bush : Set} \rightarrow \text{Set where} \\
  \text{bnil : } \forall \{ \text{A : Set} \} \rightarrow \text{Bush A} \\
  \text{bnode : } \forall \{ \text{A : Set} \} \rightarrow \text{A} \rightarrow \text{Bush (Bush A)} \rightarrow \text{Bush A}
  \]

  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

```haskell
data PTree : Set → Set where
  pleaf : ∀{A : Set} → A → PTree A
  pnode : ∀{A : Set} → PTree (A × A) → PTree A
```

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

• Bushes

```haskell
data Bush : Set → Set where
  bnil : ∀{A : Set} → Bush A
  bnode : ∀{A : Set} → A → Bush (Bush A) → Bush A
```

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**

  ```haskell
  data PTree : Set → Set where
  pleaf : ∀{A : Set} → A → PTree A
  pnode : ∀{A : Set} → PTree(A × A) → PTree A
  ```

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

- **Bushes**

  ```haskell
  data Bush : Set → Set where
  bnil : ∀{A : Set} → Bush A
  bnode : ∀{A : Set} → A → Bush (Bush A) → Bush A
  ```

  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

\[
\text{data } \text{PTree} : \text{Set } \rightarrow \text{Set where}
\]
\[
\quad \text{pleaf} : \forall \{A : \text{Set}\} \rightarrow A \rightarrow \text{PTree } A
\]
\[
\quad \text{pnode} : \forall \{A : \text{Set}\} \rightarrow \text{PTree } (A \times A) \rightarrow \text{PTree } A
\]

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

- Bushes

\[
\text{data } \text{Bush} : \text{Set } \rightarrow \text{Set where}
\]
\[
\quad \text{bnil} : \forall \{A : \text{Set}\} \rightarrow \text{Bush } A
\]
\[
\quad \text{bnode} : \forall \{A : \text{Set}\} \rightarrow A \rightarrow \text{Bush } (\text{Bush } A) \rightarrow \text{Bush } A
\]

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

\[
\text{data } D \ A_1 \ldots A_n : B_1 \to \ldots \to B_m \to \text{Set where}
\]

\[
c_1 : \forall \{A_1 \ldots A_n B_1 \ldots B_m\} \to T_{11} \to \ldots \to T_{1j_1} \to D \ A_1 \ldots A_n B_1 \ldots B_m
\]

\[
\ldots
\]

\[
c_k : \forall \{A_1 \ldots A_n B_1 \ldots B_m\} \to T_{k1} \to \ldots \to T_{k_j} \to D \ A_1 \ldots A_n B_1 \ldots B_m
\]

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

\[\implies\] no negative occurrences of \(D\) in argument types of constructors

\[\implies\] \(D\) can be interpreted as the least fixpoint of a functor
Syntax of Nested Types (II)

- The general form of a nested type is

\[
\text{data } D \ A_1 \ldots A_n : B_1 \rightarrow \ldots \rightarrow B_m \rightarrow \text{Set where} \\
\quad c_1 : \forall \{A_1 \ldots A_n B_1 \ldots B_m\} \rightarrow T_{11} \rightarrow \ldots \rightarrow T_{1j_1} \rightarrow D \ A_1 \ldots A_n B_1 \ldots B_m \\
\ldots \\
\quad c_k : \forall \{A_1 \ldots A_n B_1 \ldots B_m\} \rightarrow T_{k1} \rightarrow \ldots \rightarrow T_{k_j k} \rightarrow D \ A_1 \ldots A_n B_1 \ldots B_m
\]

where either

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

or

- \(T_{ij}\) is inductive and has the form

\[
C_1 \rightarrow \ldots \rightarrow C_p \rightarrow D \ A_1 \ldots A_n V_1 \ldots V_m
\]

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
  \[\Rightarrow\] no negative occurrences of \(D\) in argument types of constructors
  \[\Rightarrow\] \(D\) 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 $C$ 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 $\{\eta_X : FX \to GX\}_{X : C}$ such that if $f : X \to Y$ in $C$ then $\eta_Y \circ map_F f = map_G f \circ \eta_X$.

\[
\begin{array}{ccc}
FX & \xrightarrow{\eta_X} & GX \\
\downarrow{\text{map}_F f} & & \downarrow{\text{map}_G f} \\
FY & \xrightarrow{\eta_Y} & GY \\
\end{array}
\]

• 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 $(\eta \circ \mu)_X = \eta_X \circ \mu_X$.
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 $C$ 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 $\{\eta_X : FX \to GX\}_{X:C}$ such that if $f : X \to Y$ in $C$ then $\eta_Y \circ \text{map}_F f = \text{map}_G f \circ \eta_X$.

$$
\begin{array}{ccc}
FX & \xrightarrow{\eta_X} & GX \\
\downarrow \text{map}_F f & & \downarrow \text{map}_G f \\
FY & \xrightarrow{\eta_Y} & GY
\end{array}
$$

- 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 $(\eta \circ \mu)_X = \eta_X \circ \mu_X$.
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 $C$ 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 $\{\eta_X : FX \to GX\}_{X : C}$ such that if $f : X \to Y$ in $C$ then $\eta_Y \circ \text{map}_F f = \text{map}_G f \circ \eta_X$

\[\begin{array}{c}
FX \xrightarrow{\eta_X} GX \\
\downarrow \text{map}_F f \downarrow \downarrow \text{map}_G f \\
FY \xrightarrow{\eta_Y} GY
\end{array}\]

• The identity on $F$ is the identity natural transformation $\text{id}_F$ from $F$ to $F$.

• Composition of natural transformations is componentwise, i.e., if $X : C$ then $(\eta \circ \mu)_X = \eta_X \circ \mu_X$
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 $C$ 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 $\{\eta_X : FX \to GX\}_{X : C}$ such that if $f : X \to Y$ in $C$ then $\eta_Y \circ map_F f = map_G f \circ \eta_X$

  \[
  \begin{array}{ccc}
  FX & \xrightarrow{\eta_X} & GX \\
  \downarrow{map_F f} & & \downarrow{map_G f} \\
  FY & \xrightarrow{\eta_Y} & GY
  \end{array}
  \]

• 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 $\mathcal{C}$ 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 $\{\eta_X : FX \to GX\}_{X:\mathcal{C}}$ 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

$$(\eta \circ \mu)_X = \eta_X \circ \mu_X$$
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 $\{\eta_X : FX \to GX\}_{X: \mathcal{C}}$ such that if $f : X \to Y$ in $\mathcal{C}$ then $\eta_Y \circ map_F f = map_G f \circ \eta_X$

\[FX \xrightarrow{\eta_X} GX\]
\[FY \xrightarrow{\eta_Y} GY\]
\[map_F f \downarrow \quad \quad \quad map_G f \downarrow\]

• 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 $F : C \to D$ is a functor, then $HF$ is also a functor from $C$ to $D$
  - if $X : C$ then $HFX : D$
  - if $f : X \to Y$ in $C$ then $HFf : HFX \to HFY$ in $D$.
  - if $\eta : F \to G$ then $\text{map}_H \eta : HF \to HG$

$\text{map}_H$ 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 $F : C \to D$ is a functor, then $HF$ is also a functor from $C$ to $D$

- if $X : C$ then $HF X : D$
- if $f : X \to Y$ in $C$ then $HF f : HFX \to HFY$ in $D$.
- 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 $F : C \to D$ is a functor, then $HF$ is also a functor from $C$ to $D$

- if $X : C$ then $HFX : D$
- if $f : X \to Y$ in $C$ then $HFf : HFX \to HFY$ in $D$.
- 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 $F : C \rightarrow D$ is a functor, then $HF$ is also a functor from $C$ to $D$
  - if $X : C$ then $HFX : D$
  - if $f : X \rightarrow Y$ in $C$ then $HFf : HFX \rightarrow HFY$ in $D$.
  - if $\eta : F \rightarrow G$ then $map_H \eta : HF \rightarrow HG$

• $map_H$ 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)

- data PTree : Set → Set where
  - pleaf : ∀{A : Set} → A → PTree A
  - pnode : ∀{A : Set} → PTree (A × A) → PTree A

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

- data Bush : Set → Set where
  - bnil : ∀{A : Set} → Bush A
  - bnode : ∀{A : Set} → A → Bush (Bush A) → Bush A

  has \( HF X = 1 + X \times F (F X) \), so Bush is interpreted as \( \mu H \),
i.e., as \( \mu F. \lambda X. 1 + X \times F (F X) \)
Semantics of Nested Types (V)

- data PTree : Set → Set where
  pleaf : ∀{A : Set} → A → PTree A
  pnode : ∀{A : Set} → PTree(A × A) → PTree A

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

- data Bush : Set → Set where
  bnil : ∀{A : Set} → Bush A
  bnode : ∀{A : Set} → A → Bush (Bush A) → Bush A

  has \( HF X = 1 + X \times F(F X) \), so Bush is interpreted as \( \mu H \),
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 $HFX = 1 + X \times FX$, so List is interpreted as $\mu H$, 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.

\[
\text{data List (A : Set) : Set where}
\]
\[
\begin{array}{c}
\text{[ ] : List A} \\
\text{- :: - : A → List A → List A}
\end{array}
\]

is also

\[
\text{data List : Set → Set where}
\]
\[
\begin{array}{c}
\text{[ ] : ∀\{A : Set\} → List A} \\
\text{- :: - : ∀\{A : Set\} → A → List A → List A}
\end{array}
\]

which has \( H F X = 1 + X \times FX \), so List is interpreted as \( \mu H \),
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.

\[
\text{data List } (A : \text{Set}) : \text{Set where} \\
[ ] : \text{List } A \\
\_ :: \_ : A \to \text{List } A \to \text{List } A
\]

is also

\[
\text{data List } : \text{Set } \to \text{Set where} \\
[ ] : \forall\{A : \text{Set}\} \to \text{List } A \\
\_ :: \_ : \forall\{A : \text{Set}\} \to A \to \text{List } A \to \text{List } A
\]

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

• Because $\mu H$ 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 $map_{\mu H}$.

- $map_{List} :: (A \rightarrow B) \rightarrow List A \rightarrow List B$
  $map_{List} f [] = []$
  $map_{List} f (x :: xs) = (f x) :: map_{List} f xs$

- $map_{PTree} :: (A \rightarrow B) \rightarrow PTree A \rightarrow PTree B$
  $map_{PTree} f (pleaf x) = pleaf (f x)$
  $map_{PTree} f (pnode ts) = pnode (map_{PTree} (map_{PTree} f) ts)$

- $map_{Bush} :: (A \rightarrow B) \rightarrow Bush A \rightarrow Bush B$
  $map_{Bush} f bnil = bnil$
  $map_{Bush} f (bnode a bb) = bnode (f a) (map_{Bush} (map_{Bush} f) bb)$

• $map_D$ 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 $\mu H$.

• Because $\mu H$ 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 $map_{\mu H}$.

\[
map_{List} :: (A \rightarrow B) \rightarrow List A \rightarrow List B
\]
\[
map_{List} f [] = []
\]
\[
map_{List} f (x :: xs) = (f x) :: map_{List} f xs
\]

\[
map_{PTree} :: (A \rightarrow B) \rightarrow PTree A \rightarrow PTree B
\]
\[
map_{PTree} f (pleaf x) = pleaf (f x)
\]
\[
map_{PTree} f (pnode ts) = pnode (map_{PTree} (map_{PTree} f) ts)
\]

\[
map_{Bush} :: (A \rightarrow B) \rightarrow Bush A \rightarrow Bush B
\]
\[
map_{Bush} f bnil = bnil
\]
\[
map_{Bush} f (bnode a bb) = bnode (f a) (map_{Bush} (map_{Bush} f) bb)
\]

• $map_D$ 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 $\mu H$.
- Because $\mu H$ 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 $map_{\mu H}$.

\[
\begin{align*}
map_{\text{List}} &:: (A \rightarrow B) \rightarrow \text{List } A \rightarrow \text{List } B \\
map_{\text{List}} f \; [ ] & = [ ] \\
map_{\text{List}} f \; (x :: xs) & = (f \; x) :: \map_{\text{List}} f \; xs
\end{align*}
\]

- $map_{\text{PTree}} :: (A \rightarrow B) \rightarrow \text{PTree } A \rightarrow \text{PTree } B$
- $map_{\text{PTree}} f \; (\text{pleaf } x) = \text{pleaf } (f \; x)$
- $map_{\text{PTree}} f \; (\text{pnode } ts) = \text{pnode } (map_{\text{PTree}} (map_{\text{PTree}} f) \; ts)$

- $map_{\text{Bush}} :: (A \rightarrow B) \rightarrow \text{Bush } A \rightarrow \text{Bush } B$
- $map_{\text{Bush}} f \; \text{bnil} = \text{bnil}$
- $map_{\text{Bush}} f \; (\text{bnode } a \; bb) = \text{bnode } (f \; a) \; (map_{\text{Bush}} (map_{\text{Bush}} f) \; bb)$

- $map_D$ 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 $\mu H$.

- Because $\mu H$ 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 $map_{\mu H}$.

- $map_{List} :: (A \rightarrow B) \rightarrow List A \rightarrow List B$
  
  $map_{List} f [] = []$
  
  $map_{List} f (x :: xs) = (f x) :: map_{List} f xs$

- $map_{PTree} :: (A \rightarrow B) \rightarrow PTree A \rightarrow PTree B$
  
  $map_{PTree} f (pleaf x) = pleaf (f x)$
  
  $map_{PTree} f (pnode ts) = pnode (map_{PTree} (map_{PTree} f \times f) ts)$

- $map_{Bush} :: (A \rightarrow B) \rightarrow Bush A \rightarrow Bush B$
  
  $map_{Bush} f bn = bn$
  
  $map_{Bush} f (bnode a bb) = bnode (f a) (map_{Bush} (map_{Bush} f) bb)$

- $map_D$ 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 $\mu H$.

• Because $\mu H$ 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 $map_{\mu H}$.

- $map_{List} :: (A \to B) \to List A \to List B$
  - $map_{List} f [] = []$
  - $map_{List} f (x :: xs) = (f x) :: map_{List} f xs$

- $map_{PTree} :: (A \to B) \to PTree A \to PTree B$
  - $map_{PTree} f (pleaf x) = pleaf (f x)$
  - $map_{PTree} f (pnode ts) = pnode (map_{PTree} (map_{PTree} f) ts)$

- $map_{Bush} :: (A \to B) \to Bush A \to Bush B$
  - $map_{Bush} f bnil = bnil$
  - $map_{Bush} f (bnode a bb) = bnode (f a) (map_{Bush} (map_{Bush} f) bb)$

• $map_D$ 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 \( \mu H \).

• Because \( \mu H \) 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 \( map_{\mu H} \).

  \[
  \begin{align*}
  map_{\text{List}} & :: (A \to B) \to \text{List} A \to \text{List} B \\
  map_{\text{List}} f \ [\] & = [\] \\
  map_{\text{List}} f \ (x :: xs) & = (f x) :: map_{\text{List}} f \ xs
  \end{align*}
  \]

  \[
  \begin{align*}
  map_{\text{PTree}} & :: (A \to B) \to \text{PTree} A \to \text{PTree} B \\
  map_{\text{PTree}} f \ (\text{pleaf} \ x) & = \text{pleaf} \ (f x) \\
  map_{\text{PTree}} f \ (\text{pnode} \ ts) & = \text{pnode} \ (map_{\text{PTree}} (map_{\text{PTree}} f) \ ts)
  \end{align*}
  \]

  \[
  \begin{align*}
  map_{\text{Bush}} & :: (A \to B) \to \text{Bush} A \to \text{Bush} B \\
  map_{\text{Bush}} f \ \text{bnil} & = \text{bnil} \\
  map_{\text{Bush}} f \ (\text{bnode} \ a \ \text{bb}) & = \text{bnode} \ (f a) \ (map_{\text{Bush}} (map_{\text{Bush}} f) \ \text{bb})
  \end{align*}
  \]

• \( map_D \) 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

\[
\begin{array}{ccc}
(\mu H) X & \xrightarrow{\eta_X} & (\mu H') X \\
\downarrow{\text{map}_{\mu H} f} & & \downarrow{\text{map}_{\mu H'} f} \\
(\mu H) Y & \xrightarrow{\eta_Y} & (\mu H') Y
\end{array}
\]

• Computationally (i.e., reflecting back into syntax), we can think of natural transformations as polymorphic functions between data types whose constructors are interpreted as \( \mu H \) 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 $\eta : \mu H \rightarrow \mu H'$ gives commuting squares: if $f : X \rightarrow Y$, then

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

- Computationally (i.e., reflecting back into syntax), we can think of natural transformations as polymorphic functions between data types whose constructors are interpreted as $\mu H$ 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

\[
\begin{array}{c}
\begin{array}{ccc}
(\mu H) X & \xrightarrow{\eta X} & (\mu H') X \\
\downarrow & & \downarrow \\
(\mu H) Y & \xrightarrow{\eta Y} & (\mu H') Y
\end{array}
\end{array}
\]

\[
\begin{array}{c}
\begin{array}{c}
map_{\mu H} f \\
\downarrow
\end{array}
\end{array}
\begin{array}{c}
\begin{array}{c}
map_{\mu H'} f \\
\downarrow
\end{array}
\end{array}
\]

• Computationally (i.e., reflecting back into syntax), we can think of natural transformations as polymorphic functions between data types whose constructors are interpreted as \( \mu H \) 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

\[
\begin{array}{ccc}
(\mu H) X & \xrightarrow{\eta X} & (\mu H') X \\
\downarrow_{\text{map}_{\mu H} f} & & \downarrow_{\text{map}_{\mu H'} f} \\
(\mu H) Y & \xrightarrow{\eta Y} & (\mu H') Y
\end{array}
\]

• Computationally (i.e., reflecting back into syntax), we can think of natural transformations as polymorphic functions between data types whose constructors are interpreted as \( \mu H \) 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

$$
\begin{array}{ccc}
(\mu H) X & \xrightarrow{\eta_X} & (\mu H') X \\
\downarrow_{\text{map}_{\mu H} f} & & \downarrow_{\text{map}_{\mu H'} f} \\
(\mu H) Y & \xrightarrow{\eta_Y} & (\mu H') Y 
\end{array}
$$

- Computationally (i.e., reflecting back into syntax), we can think of natural transformations as polymorphic functions between data types whose constructors are interpreted as $\mu H$ 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 flatten : ∀{A : Set} → PTree A → List A acts like this

\[
\text{flatten} \left( ((a_{11}, a_{12}), (a_{21}, a_{22})), ((a_{21}, a_{22}), (a_{221}, a_{222}))) = (a_{11}, a_{12}, a_{21}, a_{22}, a_{221}, a_{222}) \right)
\]

then

\[
\begin{align*}
\text{PTree A} & \quad \xrightarrow{\text{flatten}\{A\}} \quad \text{List A} \\
\downarrow \text{mapp}_{\text{Tree}} f & \quad \quad & \quad \downarrow \text{map}_{\text{List}} f \\
\text{PTree B} & \quad \xrightarrow{\text{flatten}\{B\}} \quad \text{List B} \\
\downarrow \text{mapp}_{\text{Tree}} f & \quad \quad & \quad \downarrow \text{map}_{\text{List}} f \\
((a_{11}, a_{12}), (a_{21}, a_{22})) & \quad \xrightarrow{\text{flatten}\{A\}} \quad (a_{11}, a_{12}, a_{21}, a_{22}) \\
\downarrow \text{mapp}_{\text{Tree}} f & \quad \quad & \quad \downarrow \text{map}_{\text{List}} f \\
((fa_{11}, fa_{12}), (fa_{21}, fa_{22})) & \quad \xrightarrow{\text{flatten}\{B\}} \quad (fa_{11}, fa_{12}, fa_{21}, fa_{22})
\end{align*}
\]

- 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 : ∀{A : Set} → PTree A → List A` acts like this
  \[\text{flatten}(((a_{111}, a_{112}), (a_{121}, a_{122})), ((a_{211}, a_{212}), (a_{221}, a_{222}))) = (a_{111}, a_{112}, a_{121}, a_{122}, a_{211}, a_{212}, a_{221}, a_{222})\]

  then

```
PTree A      flatten{A}     List A
  |                |                 |
  v                v                 v
PTree B  flatten{B}  List B
  |                  |
  |                  |
  v                  v
((a_{11}, a_{12}), (a_{21}, a_{22})) flatten{A} (a_{11}, a_{12}, a_{21}, a_{22})
  |                  |
  |                  |
  v                  v
((fa_{11}, fa_{12}), (fa_{21}, fa_{22})) flatten{B} (fa_{11}, fa_{12}, fa_{21}, fa_{22})
```

- 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 : ∀{A : Set} → PTree A → List A` acts like this

\[
\text{flatten}(((a_{111}, a_{112}), (a_{121}, a_{122})), ((a_{211}, a_{212}), (a_{221}, a_{222}))) = (a_{111}, a_{112}, a_{121}, a_{122}, a_{211}, a_{212}, a_{221}, a_{222})
\]

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.
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 : ∀{A : Set} → PTree A → List A` acts like this
  
  \[
  \text{flatten}(((a_{111}, a_{112}), (a_{121}, a_{122})), ((a_{211}, a_{212}), (a_{221}, a_{222}))) = (a_{111}, a_{112}, a_{121}, a_{122}, a_{211}, a_{212}, a_{221}, a_{222})
  \]

  then

\[
\begin{array}{ccc}
\text{PTree A} & \xrightarrow{\text{flatten}\{A\}} & \text{List A} \\
\downarrow\text{map}_{\text{Tree}} f & & \downarrow\text{map}_{\text{List}} f \\
\text{PTree B} & \xrightarrow{\text{flatten}\{B\}} & \text{List B}
\end{array}
\]

\[
((a_{11}, a_{12}), (a_{21}, a_{22})) \xleftarrow{\text{flatten}\{A\}} (a_{11}, a_{12}, a_{21}, a_{22})
\]

\[
((fa_{11}, fa_{12}), (fa_{21}, fa_{22})) \xleftarrow{\text{flatten}\{B\}} (fa_{11}, fa_{12}, fa_{21}, fa_{22})
\]

- 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 : ∀{A : Set} → PTree A → List A acts like this
  \[
  \text{flatten}
  \begin{pmatrix}
  ((a_{111}, a_{112}), (a_{121}, a_{122})),
  ((a_{211}, a_{212}), (a_{221}, a_{222}))
  \end{pmatrix}
  =
  \begin{pmatrix}
  (a_{111}, a_{112}, a_{121}, a_{122}, a_{211}, a_{212}, a_{221}, a_{222})
  \end{pmatrix}
  \]
  then
  \[
  \begin{array}{c}
  \text{PTree A} \\
  \downarrow \text{mapper}_\text{Tree f} \\
  \text{PTree B}
  \end{array}
  \xrightarrow{\text{flatten}\{A\}}
  \begin{array}{c}
  \text{List A} \\
  \downarrow \text{mapper}_\text{List f} \\
  \text{List B}
  \end{array}
  \]

- 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.
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!)