Lecture 17: Inductive Types

Lecturer: Max S. New
Scribe: Kevin Wang

March 15, 2023

1 Inductive data type examples

1.1 Natural numbers $\mathcal{N}$

Recall that at the end of last lecture we introduced the concept of inductive types in STT and category theory using natural numbers. We created what we called a *natural number object* (NNO) in a category C with terminal object 1, defined as the following structure:

1. An object $N \in C$
2. Two functions `zero` and `succ` (short for successor) such that $1 \xrightarrow{\texttt{zero}} N \xleftarrow{\texttt{succ}} N$
3. defined such that $\forall A \in C. 1 \xrightarrow{\texttt{z}} A \xleftarrow{\texttt{s}} A$, $\exists! N \xrightarrow{\texttt{rec(z,s)}} A$ the following diagram commutes:

$$\begin{tikzcd}
1 \arrow[r, "zero"] \arrow[d, "id_1"] & N \arrow[d, "rec(z,s)"] & N \arrow[l, "succ"'] \arrow[d, "rec(z,s)"] \\
1 \arrow[r, "z"'] & A & A \arrow[l, "s"]
\end{tikzcd}$$

`succ` can be thought of as a way to define recursive functions on $\mathcal{N}$ that are guaranteed to terminate. We will expand on this today to extend to other inductive types.

1.2 Lists

Let $A \in C$ where C is a cartesian category (i.e. has products and unit 1). We define an *A-list object* to consist of:

1. An object $L \in C$
2. Two functions `nil` and `cons` (short for constructor) such that $1 \xrightarrow{\texttt{nil}} L \xleftarrow{\texttt{cons}} A \times L$, where `nil` gives an "empty" list and `cons` can be thought of as giving or *constructing* an element of type A and adding it to the list;
3. defined such that $\forall B \in C.1 \xrightarrow{\texttt{n}} B \xleftarrow{\texttt{c}} A \times B$, $\exists! L \xrightarrow{\texttt{fold}} B$ the following diagram commutes:

$$\begin{tikzcd}
1 \arrow[r, "nil"] \arrow[d, "id_1"] & L \arrow[d, "fold"] & A \times L \arrow[l, "cons"'] \arrow[d, "id_A \times fold"] \\
1 \arrow[r, "n"'] & B & A \times B \arrow[l, "c"]
\end{tikzcd}$$

where the rightmost morphism $id_A \times fold = (\pi_1, fold \circ \pi_2) : A \times L \to A \times B$. This is simply preserving A by identity, then applying `fold` to L to get B.

Let's talk more about `fold` and what exactly it means[1]. We can more formally define `fold` as a morphism which, given an element of type B and a function $A \times B \to B$, converts to a list of type A and then returns an element of type B. Mathematically,

$$\texttt{fold} : B \to (A \times B \to B) \to \texttt{List}A \to B$$

Note that based on diagram commutivity, $\texttt{fold} \circ \texttt{nil} = n$ and $\texttt{fold}(\texttt{cons}(a, l)) = c(a, \texttt{fold}(l))$. This provides a formal definition of `fold`. Note as well that by the unique existence qualifier, if we have $f : \texttt{List}A \to B$ such that $f(nil) = n$ and $\forall a, l. f(a, l) = c(a, f(l))$, then $f = fold$.

1.3 Binary trees

Let $D_l, D_n \in C$ be arbitrary objects. A *binary tree* with D_l data at leaves and D_n data at nodes is defined as

1. An object $T \in C$
2. Two functions `leaf` and `node` such that $D_L \xrightarrow{\texttt{leaf}} T \xleftarrow{\texttt{node}} D_n \times T \times T$
3. defined such that $\forall B \in C.D_L \xrightarrow{\texttt{l}} B \xleftarrow{\texttt{n}} D_n \times B \times B$, $\exists! T \xrightarrow{\texttt{fold}} B$ the following diagram commutes:

[1]Note that in most functional programming languages this is called `foldr`, as we are "folding" from right hand side to left hand side. See Haskell documentation here: https://zvon.org/other/haskell/Outputprelude/foldr_f.html There is also a similar `foldl` that works left-to-right.

$$\begin{tikzcd}
D_L \arrow[r, "leaf"] \arrow[d, "id_{D_L}"] & T \arrow[d, "fold"] & D_n \times T \times L \arrow[l, "node"'] \arrow[d, "id_{D_n} \times fold \times fold"] \\
D_L \arrow[r, "l"'] & B & D_n \times B \times B \arrow[l, "n"]
\end{tikzcd}$$

Here, fold must be applied twice in the rightmost morphism on both the second and third elements of the product. The ideas here are otherwise similar to the ones in List.

2 Connections to category theory

We would like to abstract over the commonality here to get a general notion of "inductive data object" as a universal property in a category. It will be most convenient to do this when C in addition to products has all coproducts. The reason is that the top of each diagram above can be expressed through a single morphism from a sum of two types to the non-unit type. For $\mathcal{N}$ this is some $f : 1 + N \to N$; for lists this is some $g : 1 + (A \times L) \to L$; for binary trees this is some $h : D_L + (D_n \times T \times T) \to T$.

Crucially, every type on the left-hand side is some expression of the type on the right-hand side. Rewriting the previous definitions we can say each left-hand side expression is some $F(B)$ for type B on the right-hand side, where for $\mathcal{N}$ $F(B) = 1+B$, for lists $F(B) = 1 + A + B$ and for binary trees $F(B) = D_L + (D_n \times B \times B)$. We can show that all of these expressions are functorial in B; in other words, each of these uses an *endofunctor*[2] $F : C \to C$ which essentially shows the recursive structure of the inductive datatype $B \in C$.

$$F_{nat}(X) = 1 + X$$
$$F_{list}(X) = 1 + A \times X$$
$$F_{tree}(X) = D_l + D_n \times X \times X$$

In fact, notice that all of these are essentially *polynomials* in X. Any such polynomial expression can be proven to be functorial since coproducts, products and constant functors are functorial.

2.1 F-algebras

Given an endofunctor F, an F-algebra is a morphism $\alpha : FB \to B$ for some object $B \in C$. F, as described above, is a functor describing the recursive structure of an

[2]An endofunctor is simply a functor from a category into itself.

inductive datatype in the F-algebra.

We then want to construct a *category* of F-algebras, such that the universal property of the inductive datatypes above is given by being an *initial object* in the category $F - Alg$ of F-algebras, often called an *initial algebra*.

Let $\alpha : FB \to B$ and $\beta : FC \to C$. A homomorphism between F-algebras α to β consists of $\psi : B \to C$ such that the following diagram commutes:

$$\begin{tikzcd}
FB \arrow[r, "F\psi"] \arrow[d, "\alpha"'] & FC \arrow[d, "\beta"] \\
B \arrow[r, "\psi"'] & C
\end{tikzcd}$$

Let's consider what this diagram looks like in the context of lists. In the following diagram, note that $F(B) = 1 + (A \times B)$ for a type B.

$$\begin{tikzcd}
1 + (A \times L_A) \arrow[r, "id_1 + (id_A \times fold_\beta)"] \arrow[d, "cons"'] & 1 + (A \times B) \arrow[d, "\beta"] \\
L_A \arrow[r, "fold_\beta"'] & B
\end{tikzcd}$$

The β function should be interpreted as follows: if the input type is of type 1, then $\beta \circ 1 + (A \times (fold_\beta))$ will simply be the base case of fold for type B. For an input type of type $A \times L_A$, $\beta \circ 1 + (A \times (fold_\beta))$ will *recursively* call fold.

Note that the path from $1 + (A \times L_A) \to 1 + (A \times B) \to B$ can be broken down into the following diagram, which more clearly illustrates how we can arrive at B from type 1 or from type $(A \times B)$:

$$\begin{tikzcd}
 & 1 + (A \times L_A) \arrow[d, "id_1 + (id_A \times fold_\beta)"] & \\
 & 1 + (A \times B) \arrow[d, "\beta"'] & \\
1 \arrow[uur, "i_1"] \arrow[r, dashed, "\beta \circ i_1(...)"'] & B & A \times B \arrow[uul, "i_2"'] \arrow[l, dashed, "\beta \circ i_2(...)"]
\end{tikzcd}$$

Similar to before, unique existence allows us to say $f = fold$ for any f that makes the diagram commute.

$$\begin{tikzcd}
1 + (A \times L_A) \arrow[r, "id_1 + (id_A \times f)"] \arrow[d, "cons"'] & 1 + (A \times B) \arrow[d, "\beta"] \\
L_A \arrow[r, "f"'] & B
\end{tikzcd}$$

We will next show how to implement cases in terms of fold, starting with an arbitrary functor F and the corresponding F-algebra. This will be a proof of *Lambek's Lemma*, which says that the `cons` morphism we have been using to create objects of a given type is an isomorphism.

2.2 Lambek's Lemma

Fix $F : C \to C$ for some category C. Let $FT \xrightarrow{cons} T$ be the initial F-algebra. Our goal is to show we can "go backwards" in recursive cases to reach base cases. As such, our goal is to show that **`cons` is an isomorphism**. The inverse allows us to "unwrap" one layer of the recursive structure. This shows that adding in such a function is a definitional extension to a language with fold, and so we can add it without changing any extensional properties of the programming language.

So we want to construct a morphism `uncons`: $T \to FT$ that is a left and right inverse of `cons`. To do this we can use the universal property: to construct such a map we can use `fold` on an algebra structure for FT:

$$\begin{tikzcd}
FT \arrow[r, "F(fold_{F(cons)})"] \arrow[d, "cons"'] & FFT \arrow[d, "F(cons)"] \\
T \arrow[r, "fold_{F(cons)}"'] & FT
\end{tikzcd}$$

To show that uncons and cons compose to the identity, we can extend the square from above (which we know commutes) to another diagram which we know commutes:

$$\begin{tikzcd}
FT \arrow[r, "F(uncons)"] \arrow[d, "cons"'] \arrow[rr, bend left, "F(cons \circ uncons)"] & FFT \arrow[r, "F(cons)"] \arrow[d, "F(cons)"] & FT \arrow[d, "cons"] \\
T \arrow[r, "uncons"'] & FT \arrow[r, "cons"'] & T
\end{tikzcd}$$

Note that we get this square by using already established morphisms ($F(cons) : FFT \to FT$ and $cons : FT \to T$). Because functors preserve compositionality and $F(cons \circ uncons) : FT \to FT$, $cons \circ uncons$ is the identity.

We can zoom in on the left subsquare to show identity in the other direction, $uncons \circ cons$:

$$\begin{tikzcd}
FT \arrow[r, "F(uncons)"] \arrow[d, "cons"'] \arrow[dr, "F(cons \circ uncons) = F(id_T)" description] & FFT \arrow[d, "F(cons)"] \\
T \arrow[r, "uncons"'] & FT
\end{tikzcd}$$

We have already showed that $cons \circ uncons$ composes to the identity, so we can draw the additional arrow $F(id_T) : FT \to FT$. Note that functors preserve identity, so $F(id_T) = id_T$, meaning that $uncons \circ cons = id_T$ by diagram commutivity.

2.3 List example of uncons

When considering lists, we can define the uncons morphism as $uncons : ListA \to 1 + (A \times ListA)$. By simply unraveling what this should be according to the general definition in the previous subsection, this translates to $fold(i_0())(\lambda(x, \alpha) \to L(x, cons(t)))$, where $\lambda(x, \alpha) \to L(x, cons(t)) : A \times (1 + (A \times ListA)) \to 1 + (A \times ListA)$. The first part of this definition, the $(i_0())$, says to construct an empty list if the input is of type 1. If the input is of type $A \times List_A$, then we instead apply the λ function which takes in an existing list and tail element, then constructs the tail and appends it to the list.

By unraveling the definition using other inductive datatypes, we can come up with similar interpretations of uncons using those datatypes.

3 Coinductive datatypes

Just as inductive datatypes such as numbers, lists and trees can be defined as initial algebras, *co*inductive datatypes can be defined as initial algebras on the opposite category. Unraveling the dualities, the direct definition is that of a *final coalgebras*, i.e., a terminal (aka final) object in a category of coalgebras, which are morphisms $B \to FB$, whose morphisms are analogous to algebra homomorphisms.

Examples of coinductive datatypes in Set are similar to inductive datatypes but allowing for infinitely large structures:

- Extended natural numbers $\mathbb{N}_\infty$ are the final coalgebra for $F(X) = X + 1$ (i.e., the naturals as well as an infinite number ω that satisfies $\text{succ}(\omega) = \omega$).
- Possibly infinite lists are final for $F(X) = 1 + A \times X$

- Infinite streams[3] are final for $F(X) = A \times X$
- Possibly infinite binary trees for $F(X) = D_l + D_n \times X \times X$.

[3]the initial algebra in this case is trivial: $\emptyset$