Lecture 25: Recursion

Lecturer: Max S. New
Scribe: Conner Rose

November 24, 2025

1 Recap

We've talked about inductive and co-inductive data types twice already:

- Lawvere's fixed point theorem tells us the limitations of recursive types. For a surjection $A \to B^A$ in a cartesian closed category, if it is a retract (has a section), then every endomorphism $f : B \to B$ has a fixed point. This is part of how we proved that set-theoretic semantics of lambda calculus is not complete, which we'll continue to talk about. This tells us we need to move beyond set theory in a certain sense.

- Initial algebra/terminal co-algebra semantics of types. We want to keep our nice set-theoretic semantics. We can have recursive definitions but only "well-founded" ones, for instance a tree.

2 General Recursion

We need to model two things: recursive programs/functions and recursive types. Some examples:

- Recursive programs: while loops, arbitrary recursive function definitions

- Recursive types: trees, circular definitions ($D = D \to D$)

We'll focus on recursive programs today.

2.1 Trace Semantics

A "trace" on a Cartesian category $\mathcal{C}$ is effectively an operation that allows us to define morphisms that are recursive. We'll denote this operation by †. Formally, we have $(-)^\dagger : \mathcal{C}(A \times X, X) \to \mathcal{C}(A, X)$. We have a few equations:

1. Naturality in A: If we have $g : B \to A, f : A \times X \to X$, then

$$(f \circ (g \circ X))^\dagger = f^\dagger \circ g$$

2. "Dinaturality" in X: If we have $g : A \times Y \to X, f : A \times X \to Y$, then we want to get a fixed point where we get out a morphism $A \to X$. We have

$$(g \circ (\pi_1, f))^\dagger = (g \circ (\pi_1, f \circ (\pi_1, g)))^\dagger$$

3. "Diagonal" property: If we have $f : (A \times X) \times X \to X$, then

$$(f^\dagger)^\dagger = (f \circ (\pi_1, \pi_2, \pi_2))^\dagger$$

2.1.1 Recursive Computations

In call by push value, we can add recursive computation types. The A in this case is the context Γ and the X is Thunk$B$ for some computation B. To define a recursive closure, we have

$$\frac{\Gamma, t : \mathrm{Thunk}B \vdash V : \mathrm{Thunk}B}{\Gamma \vdash \mathrm{fix}t.V : \mathrm{Thunk}B}$$

To define a recursive computation of type B, then we have

$$\frac{\Gamma, t : \mathrm{Thunk}B \vdash M : B}{\Gamma \vdash \mathrm{fix}t.M : B}$$

We can further simplify by saying,

$$\frac{\Gamma \vdash M : \mathrm{Thunk}B \to B}{\Gamma \vdash \mathrm{fix}M : B}$$

In this syntax, we have the following rules:

1. Substitution: $(\mathrm{fix}M)[\gamma] = \mathrm{fix}(M[\gamma])$

2. A kinda-β-rule: If we have $\Gamma \vdash M : \mathrm{Thunk}B \to B'$ and $\Gamma \vdash N : \mathrm{Thunk}B' \to B$, then we can unfold a fixed point, i.e.,

$$\mathrm{fix}(\lambda t = \mathrm{Thunk}B.N\{Mt\}) = N\{\mathrm{fix}\lambda t'.M\{Nt\}\}$$

   As a special case, we have $\mathrm{fix}M = M\{\mathrm{fix}M\}$.

3. A kind-η-rule: For $M = \mathrm{Thunk}B \to ThunkB \to B$, we have

$$\mathrm{fix}\lambda t_1 = \mathrm{Thunk}B.\mathrm{fix}\lambda t_2 = \mathrm{Thunk}B.Mt_1t_2 = \mathrm{fix}\lambda t.Mtt$$

If instead we take the opposite of our Kleisli category, we have

$$\mathrm{Kl}(X, A + X) \to \mathrm{Kl}(X, A)$$

Consider X and A to be value types. We have

$$\frac{\Gamma \vdash M : X \to \mathrm{Ret}(A + X)}{\Gamma \vdash \mathrm{while}M : X \to \mathrm{Ret}A}$$

We have rules

1. Naturality in Γ: $(\text{while}M)[\gamma] = \text{while}(M[\gamma])$
2. Naturality in A: If we have $M : X \to \text{Ret}(A + X)$ and $N : A \to \text{Ret}A'$, then

$$\big(a \leftarrow \text{while}M; Na\big) = \big(\text{while}(\lambda x.s \leftarrow ; \text{cases}\{\sigma_1 a \to s' \leftarrow Na; \text{ret}(\sigma_1 a), \sigma_2 x \to \text{ret}(\sigma_2 x)\}\big)$$

3. Dinaturality: For $M : Y \to \text{Ret}(A + X)$ and $N : X \to \text{Ret}(A + Y)$. We have

$$\begin{aligned} &\text{while}(\lambda x.s \leftarrow Nx; \text{cases}\{\sigma_1 a \to \text{ret}(\sigma_1 a), \sigma_2 y \to My\}) \\ =\ &\lambda x.s \leftarrow Nx; \text{cases}\{\sigma_1 a \to \text{ret}a, \sigma_2 y \to \text{while}(\\ &\quad \lambda y.s \leftarrow My; \text{cases}\{\sigma_1 a \to \text{ret}(\sigma_1 a), \sigma_2 x' \to Nx'\})\} \end{aligned}$$

This is basically a do-while loop.

Additionally, we can flatten nested loops. With $M : X \to \text{Ret}((A + X) + X)$, then

$$\begin{aligned} \text{while}(\text{while}M) = \text{while}(\lambda x.s \leftarrow Mx; \text{cases}\{ & \\ & \sigma_2 x' \to \text{ret}(\sigma_2 x'), \\ & \sigma_1(\sigma_2 x') \to \text{ret}(\sigma_2 x'), \\ & \sigma_1(\sigma_1 a) \to \text{ret}(\sigma_1 a) \\ \}) & \end{aligned}$$

2.2 Semantics

We can model while loops using only sets and pointed sets. We have

$$\frac{M : X \to \text{Ret}(A + X)}{\text{while}M : X \to \text{Ret}A}$$

Then, we have a partial function $[\![M]\!] : X \rightharpoonup A + X$. It follows that $[\![\text{while}M]\!] : X \rightharpoonup A$. We then say that $[\![\text{while}M]\!]x = a$ if there exists some n such that $\text{loop}^n(\sigma_2 x) = \sigma_1 a$, where $\text{loop} : A + X \rightharpoonup A + X$ is defined as $\text{loop}(\sigma_1 a) = \sigma_1 a$ and $\text{loop}(\sigma_2 x) = [\![M]\!]x$.

Why we can't model recursion in this model, as we don't always have fixed points for arbitrary functions, e.g., swapping elements of a two-element set.

2.2.1 Domain Theory

A directed-complete partial order (DCPO) is a poset that has all joins of directed subsets. A subset $D \subseteq P$ is directed if for all $d_1, d_2 \in D$, there exists some $d_3 \in D$ such that $d_1 \leq d_3$ and $d_2 \leq d_3$. All of our value types will be modeled by DCPOs. Our function types will be modeled by continuous functions between DCPOs. $f : P \to Q$ is continuous if it preserves joins of directed subsets, i.e., if $x \leq y$ then $f(x) \leq f(y)$. More broadly,

$$f\left(\bigvee_{x \in D} x\right) = \bigvee_{x \in D} (f(x))$$