Programming Languages: Functional Programming
8. Streams and Codata

Shin-Cheng Mu

Augumn 2025

1 Data vs. Codata

Most materials in this handout are from Hinze [Hin08, Hin09], which I recommended very much. See also [GJ98, GH05, Gib21].

Recap: Lists

- Recall List, an inductive type given by:

$$\textbf{data}\ \text{List}\ a = [\,] \mid a : \text{List}\ a\ \ .$$

- An inductive type is defined by its constructors.
  - Values of such a type are constructed by finite applications of its constructors.
- A function consuming an inductive type is defined by pattern matching, that is, defining how it reacts to each constructor:

$$\begin{aligned} f\ [\,] &= \ldots \\ f\ (x : xs) &= \ldots\ \ . \end{aligned}$$

Coindutive Stream

- There is not a universally agreed way to present coinductive type (codata). For best contrast with inductive types, we might imagine a keyword **codata** used in the following manner.
- The type Stream represents infinite sequences.

$$\begin{aligned} &\textbf{codata}\ \text{Stream}\ a\ \textbf{where} \\ &\quad head :: \text{Stream}\ a \to a \\ &\quad tail\ \ :: \text{Stream}\ a \to \text{Stream}\ a\ \ . \end{aligned}$$

- Observe that a *coinductive* type is defined by its *deconstructors*.
  - The definition above says that a Stream a can be observed by $head$ and $tail$, which respectively yield an element a and another Stream a.
  - Potentially $tail$ can be applied to a stream infinitely many times. Therefore Stream represents sequences that are infinitely long.
  - Each call to $tail$, in the way we define them, returns results in finite time.

Defining Coinductive Values

- An infinite stream of 1's:

$$\begin{aligned} &one :: \text{Stream Int} \\ &head\ one = 1 \\ &tail\ \ one = one\ \ . \end{aligned}$$

- The stream $[n, n+1, n+2 \ldots]$:

$$\begin{aligned} &from :: \text{Int} \to \text{Stream Int} \\ &head\ (from\ n) = n \\ &tail\ \ (from\ n) = from\ (n+1)\ \ . \end{aligned}$$

- map for Stream:

$$\begin{aligned} &map :: (a \to b) \to \text{Stream}\ a \to \text{Stream}\ b \\ &head\ (map\ f\ xs) = f\ (head\ xs) \\ &tail\ \ (map\ f\ xs) = map\ f\ (tail\ xs)\ \ . \end{aligned}$$

- Coinductive values, or functions *producing* coinductive values, are defined by such *copatterns*, which specifies how it interacts with *deconstructors*.

Termination

- Inductive types are finite in size. Properly defined inductive programs terminate in finite time.

- Coinductive types represent potentially infinite data. However,
  - By restricting forms of coinductive programs, each call to deconstructors terminate in finite time.
  - There is no mechanism that allows programmers to call deconstructors an infinite number of times.
- Therefore, in a programming language having both, but separated, inductive and coinductive types, allowing only properly defined inductive and coinductive programs, every program still terminates.
- Non-termination happens when we unify inductive and coinductive types (e.g. allowing an inductively defined program to consume a coinductively defined value) and allow general recursion, as in Haskell.

Comparison

Data / Inductive Datatypes:

- Least prefix points, (carrier) of initial algebra.
- Types defined by constructors.
- Values consumed by patterns.
- Properties of them are established by inductive proofs.
- Relatively well-studied and understood.

Codata / Coinductive Datatypes

- Greatest postfix points, (carrier) of final coalgebra.
- Typed defined by deconstructors.
- Values produced by copatterns.
- Various methods have been proposed to prove their properties, each having its strength/weakness:
  - fixed-point induction, coinduction (bisimilarity), approximation lemma, unique fixed point...
- Relatively less studied, a lot yet to be explored and answered.

2 Preparations

Copatterns

Advantages of copatterns:

- Nice contrast to patterns.
- Consistent with the "match, substitute" style of evaluation — for example, anywhere we wee $tail\ (map\ f\ xs)$ in an expression, we may substitute for it the right-hand side $map\ f\ (tail\ xs)$, and vice versa.

Disadvantages of copatterns:

- The syntax is rather long.
- Cumbersome when the definition need to specify, say, the tail of the tail of a stream.
- Each stream must have a name — similar to the situation not having λ expressions.
- No direct counterpart in Haskell.

Cons-Based Syntax

- To remedy the situation, we use an alternative syntax.
- The definition:

  $$head\ xs = h$$
  $$tail\ xs = t$$

  is abbreviated to

  $$xs = h : t \ .$$

- Substituting $h : t$ for xs, we get that $h : t$ is a stream such that $head\ (h : t) = h$ and $tail\ (h : t) = t$.
- Stream is simulated by List in Haskell.

Examples

- The following respectively yield the streams of 1's, natural numbers, factorials, and the function map for streams:

  $$\begin{aligned} one &= 1 : one \ , \\ nat &= 0 : map\ (1+)\ nat \ , \\ fact &= 1 : tail\ nat \times fact \ , \\ map\ f\ xs &= f\ (head\ xs) : map\ f\ (tail\ xs) \ . \end{aligned}$$

- As a convenient abbreviation, we denote the tail of a stream s by s'. For example, while $nat = [0, 1, 2\,..]$, we have $nat' = tail\ nat = [1, 2, 3\,..]$.

Combinators

Besides map, we may define some other combinators that generate streams:

- $repeat\ x$ creates a stream if x's:

$$\begin{aligned} &repeat :: a \to \mathsf{Stream}\ a \\ &repeat\ x = x : repeat\ x\ . \end{aligned}$$

- $iterate\ f$ generates a streams from a "seed":

$$iterate\ f\ x = x : iterate\ f\ (f\ x)\ .$$

That gives us another way to define nat.

- $zipWith\ f$ combines two streams:

$$\begin{aligned} &zipWith :: (a \to b \to c) \\ &\qquad \to \mathsf{Stream}\ a \to \mathsf{Stream}\ b \to \mathsf{Stream}\ c \\ &zipWith\ f\ xs\ ys = f\ (head\ xs)\ (head\ ys) : \\ &\qquad\qquad zipWith\ (tail\ xs)\ (tail\ ys)\ . \end{aligned}$$

Streams as Numbers

To further encourage us to think of a streams as a whole, the following definition allows to see a stream as a number.

$$\begin{aligned} &\textbf{instance}\ \mathsf{Num}\ a \Rightarrow \mathsf{Num}\ (\mathsf{Stream}\ a)\ \textbf{where} \\ &\quad (+) = zipWith\ (+) \\ &\quad (-) = zipWith\ (-) \\ &\quad (\times) = zipWith\ (\times) \\ &\quad ... \\ &\quad fromInteger\ n = repeat\ (fromInteger\ n) \end{aligned}$$

Therefore,

- $3 :: \mathsf{Stream\ Int}$ expands to $repeat\ 3$, that is $[3, 3\,..]$,
- $1 + nat$ expands to $zipWith\ (+)\ [1, 1\,..]\ nat$.

Note: arithmetic laws (e.g. distributivity, commutativity) still hold after being lifted to Stream!

That allows us to write (recalling that $nat' = tail\ nat$):

$$\begin{aligned} nat &= 0 : 1 + nat\ , \\ fact &= 1 : nat' \times fact\ . \end{aligned}$$

Precedence: as before, $(:)$ binds looser than arithmetic operators. Therefore $0 : 1 + nat$ is parsed as $0 : (1 + nat)$.

More Examples

With the definition:

$$signs = (-1)\hat{}\ nats\ ,$$

we get $signs = [1, -1, 1, -1\,..]$.

The Fibonacci sequence can be defined by:

$$\begin{aligned} fib &= 0 : fib' \\ fib' &= 1 : fib'' \\ fib'' &= fib + fib'\ . \end{aligned}$$

We have $fib = [0, 1, 1, 2, 3, 5, 8, 13\,..]$.

3 More Combinators

Applicative

The $(\langle * \rangle)$ operator (pronounced "ap") is defined by:

$$\begin{aligned} &(\langle * \rangle) :: \mathsf{Stream}\ (a \to b) \to \mathsf{Stream}\ a \to \mathsf{Stream}\ b \\ &fx\ \langle * \rangle\ xs = head\ fs\ (head\ xs) : tail\ fs\ \langle * \rangle\ tail\ xs\ . \end{aligned}$$

With that, map and $zipWith$ can be redefined by:

$$\begin{aligned} &map :: (a \to b) \to \mathsf{Stream}\ a \to \mathsf{Stream}\ b \\ &map\ f\ xs = repeat\ f\ \langle * \rangle\ xs\ , \end{aligned}$$

$$zipWith\ f\ xs\ ys = (repeat\ f\ \langle * \rangle\ xs)\ \langle * \rangle\ ys\ .$$

(The parenthesis in $zipWith$ are not necessary.)

Note: $(\langle * \rangle)$ is part of an *applicative functor*, a useful concept which we unfortunately cannot cover in this term.

Interleaving

$$\begin{aligned} &(\curlyvee) :: \mathsf{Stream}\ a \to \mathsf{Stream}\ a \to \mathsf{Stream}\ a \\ &xs \curlyvee ys = head\ xs : ys \curlyvee tail\ xs\ . \end{aligned}$$

Note that $(\curlyvee)$ is neither commutative nor associative.

Another way to generate all natural numbers:

$$bin = 0 : 2 \times bin + 1 \curlyvee 2 \times bin + 2$$

Precedence: $(\curlyvee)$ binds looser than arithmetic operators, but tighter than $(:)$. Therefore the RHS of bin is parsed as: $0 : ((2 \times bin + 1) \curlyvee (2 \times bin + 2))$.

Properties of Interleaving

- $repeat\ x \curlyvee repeat\ x = repeat\ x$.
- $(fs\ \langle * \rangle\ xs) \curlyvee (gs\ \langle * \rangle\ ys) = (fs \curlyvee gs)\ \langle * \rangle\ (xs \curlyvee ys)$ — also called the *abide law*.

4 Proving Equalities

- How do we prove properties of coinductive values?
- Recall that induction works because inductive types are least prefix points.
  - E.g. Nat is the least prefix point of $\mathsf{F}\ \mathsf{X} = \{0\} \cup \{\mathbf{1}_+\ n \mid n \leftarrow \mathsf{X}\}$.
  - "Proof by induction" shows that the set of all values satisfying P is also a prefix point of F.
  - Thus we have $\mathsf{Nat} \subseteq \mathsf{P}$.
  - That is, all natural numbers satisfy P.
- It does not work for coinductive values, which are greatest postfix points.
  - Proving $\mathsf{P} \subseteq \mathsf{Stream}\ a$ does not establish anything useful.
- Various methods for proving properties (mainly *equalities*) of coinductive values were suggested.
- In this course we adopt that of Hinze [Hin08, Hin09], based on unique fixed points.

Admissible Definitions

A constant definition is *admissible* if it has the form:

$$xs = h : t \ \ , where$$

- h is a constant expression,
- t may refer to xs, and
- neither h nor t contain $head$ or $tail$ applied to xs.

Admissible Definitions

A function yielding a stream is *admissible* if it has the form:

$$f\ x = h : t \ \ , where$$

- h and t may use $head$ and $tail$ on x (argument to f);
- $head$ or $tail$ are not applied to result of f.

That extends to functions taking multiple arguments, e.g.

$$f\ x\ y = h : t \ .$$

Unique Solutions

- Admissible equations have unique solutions.
- Therefore, if $xs = \mathsf{F}\ xs$, and $ys = \mathsf{F}\ ys$, where F captures the same admissible equation, it must be the case that $xs = ys$.
- For functions, if $f\ x = \mathsf{F}\ (f\ x')$ and $g\ x = \mathsf{F}\ (g\ x')$, where F captures the same admissible equation, it must be the case that $f = g$. (Note that the arguments to recursive calls can be updated.)

Proving Streams Equal

To prove $xs = ys$:

- If one of them is already in an admissible form, try to prove that the other also has the same recursion pattern.
- If neither of them are defined by an admissible expression, try to turn one of them into an admissible definition, before proceeding with the proof. You may need to pick one that looks easier to start with.

Example

Recall the definition

$$repeat\ x = x : repeat\ x \ .$$

The recursive pattern is $(x:)$.

We prove that $repeat\ f\ \langle * \rangle\ repeat\ x = repeat\ (f\ x)$.

$$\begin{aligned} & repeat\ f\ \langle * \rangle\ repeat\ x \\ = \quad & \{\ \text{definition of } repeat\ \} \\ & (f : repeat\ f)\ \langle * \rangle\ (x : repeat\ x) \\ = \quad & \{\ \text{definition of } (\langle * \rangle)\ \} \\ & f\ x : (repeat\ f\ \langle * \rangle\ repeat\ x) \ . \end{aligned}$$

It has the same pattern as $repeat\ \ (f\ \ x)$, therefore $repeat\ f\ \langle * \rangle\ repeat\ x = repeat\ (f\ x)$.

Example

Prove that $nat = 2 \times nat \curlyvee 1 + 2 \times nat$.

Recall that $nat = 0 : 1 + nat$. We reason:

$$\begin{aligned} & 2 \times nat \curlyvee 1 + 2 \times nat \\ = \quad & \{\ \text{definition of } nat\ \} \\ & 2 \times (0 : 1 + nat) \curlyvee 1 + 2 \times nat \\ = \quad & \{\ \text{definition of } (\times)\ \} \\ & (0 : 2 \times (1 + nat)) \curlyvee 1 + 2 \times nat \\ = \quad & \{\ \text{definition of } (\curlyvee)\ \} \\ & 0 : 1 + 2 \times nat \curlyvee 2 \times (1 + nat) \\ = \quad & \{\ \text{arithmetics}\ \} \\ & 0 : 1 + (2 \times nat) \curlyvee 1 + (1 + 2 \times nat) \\ = \quad & \{\ \text{distributivity}\ \} \\ & 0 : 1 + (2 \times nat \curlyvee 1 + 2 \times nat) \ . \end{aligned}$$

Both have the same recursive pattern $(0:) \cdot (1+)$, therefore $nat = 2 \times nat \curlyvee 1 + 2 \times nat$.

Example: Cassini's Identity

Prove that $fib' \hat{} 2 - fib \times fib'' = (-1) \hat{} nat$.

We start with expanding $(-1) \hat{} nat$ and try to find its recursive form:

$$\begin{array}{ll} & (-1) \hat{} nat \\ = & \quad \{ \text{ definition of } nat \} \\ & (-1) \hat{} (0 : 1 + nat) \\ = & \quad \{ \text{ definition of } (\hat{}) \text{ (as a lifted binary operator)} \} \\ & (-1) \hat{} 0 : (-1) \hat{} (1 + nat) \\ = & \quad \{ \text{ arithmetics} \} \\ & 1 : (-1) \times (-1) \hat{} nat \ . \end{array}$$

We then calculate from $fib' \hat{} 2 - fib \times fib''$. A crucial property is that, since $fib'' = fib + fib'$, we have $fib'' - fib' = fib$.

$$\begin{array}{ll} & fib' \hat{} 2 - fib \times fib'' \\ = & \quad \{ \text{ definition of } fib'' \} \\ & fib' \hat{} 2 - fib \times (fib + fib') \\ = & \quad \{ \text{ arithmetic} \} \\ & fib' \hat{} 2 - fib \hat{} 2 - fib \times fib' \\ = & \quad \{ \text{ definitions of } fib \text{ and } fib' \} \\ & (1 : fib'') \hat{} 2 - (0 : fib') \hat{} 2 - (0 : fib') \times (1 : fib'') \\ = & \quad \{ \text{ definitions of lifted binary operators} \} \\ & 1 : fib'' \hat{} 2 - fib' \hat{} 2 - fib' \times fib'' \\ = & \quad \{ \text{ arithmetic, isolating } fib' \hat{} 2 \text{ and factoring } fib'' \} \\ & 1 : fib'' \times (fib'' - fib') - fib' \hat{} 2 \\ = & \quad \{ fib'' - fib' = fib \} \\ & 1 : fib'' \times fib - fib' \hat{} 2 \\ = & \quad \{ \text{ definition of } (\times) \} \\ & 1 : (-1) \times (fib' \hat{} 2 - fib \times fib'') \ . \end{array}$$

which has the same recursion pattern as $(-1) \hat{} nat$.

5 More about Interleaving

Binary Construction of Naturals

The previous construction of natural numbers:

$$nat = 0 : 1 + nat \ ,$$

can be seen as basing on an unary view of natural numbers. If we switch to a binary view, we can also construct natural numbers by

$$bin = 0 : 1 + 2 \times bin \curlyvee 2 + 2 \times bin \ .$$

Exercise: how do we show that $nat = bin$? See the practicals.

Binary Recurrence

- Given a function h defined by:

$$\begin{array}{l} h\ 0 \qquad\quad = k \\ h\ (1 + 2 \times n) = f\ (h\ n) \\ h\ (2 + 2 \times n) = g\ (h\ n) \ , \end{array}$$

the sequence $map\ h\ bin$ can be generated by:

$$xs = k : map\ f\ xs \curlyvee map\ g\ xs \ .$$

- In other words, a recurrence sequence where $a_0 = k$, $a_{1+2n} = f\ a_n$, and $a_{2+2n} = g\ a_n$ can be generated as xs above.

Positive Numbers

Let's try to find an admissible definition for the sequence of all positive integers: $1 + bin$:

$$\begin{array}{ll} & 1 + bin \\ = & \quad \{ \text{ definition of } bin \} \\ & 1 + (0 : 1 + 2 \times bin \curlyvee 2 + 2 \times bin) \\ = & \quad \{ \text{ definition of } (+), \text{ arithmetics} \} \\ & 1 : 2 + 2 \times bin \curlyvee 3 + 2 \times bin \\ = & \quad \{ \text{ arithmetics} \} \\ & 1 : 2 \times (1 + bin) \curlyvee 1 + 2 \times (1 + bin) \ . \end{array}$$

Therefore we get all the positive integers, bin', by:

$$bin' = 1 : 2 \times bin' \curlyvee 1 + 2 \times bin' \ .$$

Binary Recurrence

- Similarly, given a function h defined by:

$$\begin{array}{l} h\ 1 \qquad\quad = k \\ h\ (\quad 2 \times n) = f\ (h\ n) \\ h\ (1 + 2 \times n) = g\ (h\ n) \ , \end{array}$$

the sequence $map\ h\ bin'$ can be generated by:

$$xs = k : map\ f\ xs \curlyvee map\ g\ xs \ .$$

- In other words, a recurrence sequence where $a_1 = k$, $a_{2n} = f\ a_n$, and $a_{1+2n} = g\ a_n$ can be generated as xs above.

Most Sigificant Bit

- The following sequence yields the most significant bit of each positive integer:

$$msb = 1 : 2 \times msb \curlyvee 2 \times msb \ .$$

- The sequence is defined using the bin' pattern. It can be understood as bin' missing the $(1+)$.
- We have

$$msb = [1, 2, 2, 4, 4, 4, 4, 8, 8, 8, 8, 8, 8, 8, 8 \ldots] \ .$$

- It is interesting observing that

$$bin' - msb = [0, 0, 1, 0, 1, 2, 3, 0, 1, 2, 3, 4, 5, 6, 7 \ldots] \ .$$

Counting 1's in Binary Rep.

- The following sequence yields the number of 1's in the binary representation of each natural number.

$$\begin{aligned} ones &= 0 : ones' \\ ones' &= 1 : ones' \curlyvee 1 + ones' \end{aligned}$$

- We have $ones = [0, 1, 1, 2, 1, 2, 2, 3, 1, \ldots]$.
- The sequence $ones'$ ($tail$ of $ones$) is defined using the pattern of bin'.
- You may try defining one using the pattern of bin, and see why it does not work so well.

Binary Carry Sequence

- Also called the Ruler Sequence. Hinze wrote that it is a sequence "every computer scientist should know."
- Defined by $carry = 0 \curlyvee 1 + carry$. The definition is not an admissible one, but if we expand it by one step we get:

$$carry = 0 : 1 + carry \curlyvee 0 \ .$$

- $carry = [0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, 4 \ldots]$.
- It is the exponent of the largest power of two's that divides bin'.
- Or, the number of trailing 0's in the binary representation of each number in bin'.
- "Heights" of the numbers resemble marks on a ruler!

References

[GH05] Jeremy Gibbons and Graham Hutton. Proof methods for corecursive programs. *Fundamenta Informaticae*, 66(4):353–366, April-May 2005.

[Gib21] Jeremy Gibbons. How to design co-programs. *Journal of Functional Programming*, 31(e15), 2021.

[GJ98] Jeremy Gibbons and Geraint Jones. The under-appreciated unfold. In Matthias Felleisen, Paul Hudak, and Christian Queinnec, editors, *International Conference on Functional Programming*, pages 273–279. ACM Press, 1998.

[Hin08] Ralf Hinze. Functional pearl: streams and unique fixed points. In James Hook and Peter Thiemann, editors, *International Conference on Functional Programming*, pages 189–200. ACM Press, 2008.

[Hin09] Ralf Hinze. Reasoning about codata. In Zoltán Horváth, Rinus Plasmeijer, and Zsók Viktória, editors, *The Third Central European Functional Programming School*, number 6299 in Lecture Notes in Computer Science, pages 42–93. Springer-Verlag, June 2009.