Haskell uses a non-strict evaluation strategy:

- in general, one uses a leftmost-outermost strategy
- pre-defined arithmetic operators (+, -, ...) and
  comparison operators (>, <, ...) require strict evaluation of their arguments
- for pattern matching, arguments are evaluated a few steps until one can decide whether pattern matches

\[
\begin{align*}
\text{infinity} &:: \text{Int} \\
\text{infinity} &= \text{infinity} + 1 \\
\text{mult} &:: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \\
\text{mult 0} &= 0 \\
\text{mult x} &= x \times y \\
\text{mult 0 infinity} &= \text{terminates and results in 0} \\
0 \times \text{infinity} &= \text{does not terminate}
\end{align*}
\]

Haskell's evaluation strategy can be used for programs that operate on infinite data objects.

\[
\begin{align*}
\text{from} &:: \text{Num a} \Rightarrow a \rightarrow [a] \\
\text{from x} &= \text{from (x+1)} \\
\text{take} &:: \text{Int} \rightarrow [a] \rightarrow [a] \\
\text{take n} &= [x_0, x_1, x_2, \ldots] = [x_0, x_1, x_2, \ldots] \\
\text{take n} &= [x_0, \ldots, x_n, x_{n+1}, \ldots] = [x_0, \ldots, x_n, x_{n+1}, \ldots] \\
\text{take 2 (from 5)} &= \text{we start evaluating from 5 to find}\n\end{align*}
\]

\[
\text{take 2 (from 5)}
\]

\[
\text{take 2 (from 5)}
\]
take 2 (from 5)  
\[= \text{take 2 (5 : from 6)}\]  
\[= 5 : \text{take 1 (from 6)}\]  
\[= 5 : \text{take 1 (6 : from 7)}\]  
\[= 5 : 6 \text{ : take 0 (from 7)}\]  
\[= 5 : 6 : [\_] \]  
\[= [\_ , 5 , 6 ] \]  

**Practical example for programming with infinite lists:**  
**Computing prime numbers (by a variant of the "Sieve of Eratosthenes")**

**General strategy for programming with infinite lists:**

- Generate a (potentially infinite) list with approximations to the solution.
- Then repeatedly filter out the real solutions from this list.

**Example:** Compute the infinite list of all prime numbers.

1. Compute the list of all natural numbers starting with 2.
2. Mark the first unmarked number in the list.
3. Remove all multiples of the just-marked number.
4. Go back to step 2.

\[
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, \ldots]
\]

\[
\text{dropMult} : \text{Int} \to \text{List[Integer]} \to \text{List[Integer]}
\]

\[
\text{dropMult} \times \text{xs} = \{ y \mid y \leftarrow \text{xs}, y \mod x = 0 \}
\]
Thus: \( \text{drop\_mult} 2 \ [3, \ldots] = [3, 5, 7, \ldots] \)

\( \text{drop\_all} \) keeps the first element, but removes all multiples of it.

Then: it keeps the first element of the remaining list, but removes all multiples of it.

etc.

\( \text{drop\_all} :: [\text{Int}] \rightarrow [\text{Int}] \)

\( \text{drop\_all} \ (x : xs) = x : \text{drop\_all} \ (\text{drop\_mult} x \ xs) \)

\( \text{primes} :: [\text{Int}] \)

\( \text{primes} = \text{drop\_all} \ [2, \ldots] \)

**Circular Data Objects**

Some infinite objects can be represented in a "circular" way:

→ very efficient representation in terms of space

→ very efficient algorithms

\( \text{ones} :: [\text{Int}] \)

\( \text{ones} = 1 : \text{ones} \)

If a constant appears in the right-hand side of its defining equation, then it is represented as a circular object:

Example for efficient computation with circular objects:
Hamming - Problem

Goal: Compute a list with the following properties:
1. The list is sorted in ascending order and it has no duplicates.
2. The list starts with 1.
3. If the list contains a number \( x \), then it also contains \( 2x, 3x, 5x \).
4. Apart from that, the list does not contain any other numbers.

\[ 1, 2, 3, 4, 5, 6, 8, 9, 10, \ldots \]

We need an auxiliary function to merge sorted (possibly infinite) lists:

\[
\text{mer} :: \text{Ord} a \Rightarrow \text{[[a]]} \rightarrow \text{[[a]]} \rightarrow \text{[[a]]}
\]

\[
\text{mer} (x : xs) (y : ys) \mid x < y = x : \text{mer} xs (y : ys)
\]

\[
\text{mer} (x : xs) (y : ys) \mid x = y = x : \text{mer} xs ys
\]

\[
\text{mer} (x : xs) (y : ys) \mid \text{otherwise} = y : \text{mer} (x : xs) ys
\]

\[
\text{hamming} :: \text{[[Int]]}
\]

\[
\text{hamming} = 1 : \text{mer} \ (\ \text{map} \ (2 \cdot) \ \text{hamming}) \ ((\ \text{mer} \ (\ \text{map} \ (3 \cdot) \ \text{hamming}) \ ((\ \text{map} \ (5 \cdot) \ \text{hamming}))))
\]

Computing \( n \) hamming takes \( O(n) \) steps.