CS2030S

Programming Methodology II

Lab 08

Infinite List

Infinite List

Problems

Our implementation in lecture has several issues:

1. It uses `null` to represent a missing value
    - This design prevents us from having `null` as elements in the list
    - **Solution:** Use `Actually<T>`
        - *Failure:* value is missing
        - *Success:* value is present
2. Produced values are *not* memoized
    - This design results in repeated computation of the same value
    - If the computation is *expensive*, you waste processing power
    - **Solution:** Use `Memo<T>`

Infinite List

Solution

Combining the two solutions before, we have the following fields for `InfiniteList<T>`

```
public class InfiniteList<T> {
  private Lazy<Actualy<T>> head;
  private Lazy<InfiniteList<T>> tail;
    :
}
```

Logical View

- This is the *recursive* view of a list like we discuss in Lecture

Infinite List

Restrictions

Hard Restrictions

- The two fields **CANNOT** be changed
  - Cannot change type
  - Cannot add more fields
  - Cannot remove fields *(you can simply not use it, but it will be harder)*
  - **Can** change the name

- You **CANNOT** use *raw types*
- You **CANNOT** use `java.util.stream.Stream`
  - Otherwise, it is trivial to do all these and it is not much of a practice in understanding infinite list

- You **CANNOT** use `unwrap` from `Actually<T>`

Infinite List

Restrictions

Soft Restrictions

- Use `@SuppressWarnings` responsibly
  - You can minimise the use of `@SuppressWarnings` by creating a method that handles the need for this and simply call this method whenever you need
- Where possible, use the methods provided by `Actually<T>`* to handle if a value is *missing* or *present*
  - Do **NOT** use `if-else` or `try-catch`
  - **Hint:** you do **NOT** need `if-else`/`try-catch` in `generate`, `iterate`, `head`, `tail`, `map`, `filter`, and `count`

*Remember, the idea is that *failure* ≡ *missing* and *success* ≡ *present*

Tasks

Tasks

Generate

> **Description**
>
> Generate an infinite list containing only a single value supplied by the `Constant`

Example

```
InfiniteList.generate(() -> 1)
// produce [<1> [<1> [<1> [...]]]]
//   but lazily (and memoized later)
//   so initially it is simply [? ?]
```

Tasks

Iterate

> **Description**
>
> Generate an infinite list given a function `f` and seed `x` to produce `[<x> [<f(x)> [<f(f(x))> [...]]]]`

Example

```
InfiniteList.generate(0, x -> x + 1)
// produce [<0>, [<1>, [<2>, [...]]]]
//   but lazily (and memoized later)
//   so initially it is simply [<0> ?]
//   note: <0> is already computed so
//         no need to make this lazy
```

Tasks

Head/Tail

> **Description**
>
> Write the methods **`head()`** and **`tail()`** to get the head and tail of the infinite list

Problem

- In the method **`filter`**, a value may be *missing* *(i.e., not selected)*
- This needs to be accounted for in **`head()`** and **`tail()`**
  - Not in **`filter`** because we want to make **`filter`** lazy

Tasks

Head/Tail

> **Description**
>
> Write the methods `head()` and `tail()` to get the head and tail of the infinite list

Lecture Solution

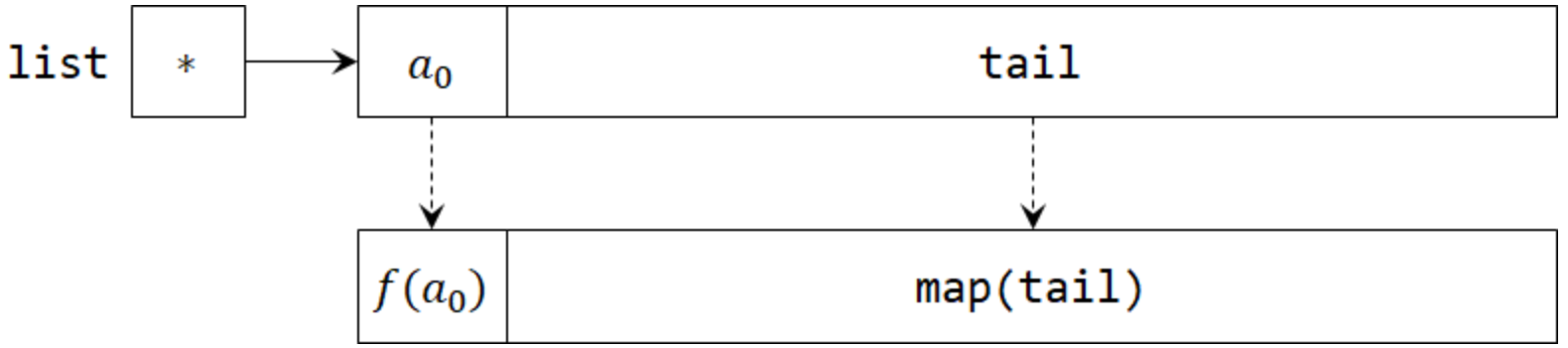
- `head()` finds the first non-null of `head` field recursively and return this value *(compute and memoize if necessary)*
- `tail()` finds the first non-null of `head` field recursively and return the corresponding `tail` value *(compute and memoize if necessary)*

> In both cases, you do **NOT** need `if-else` or `try-catch`

Tasks

Generate
Iterate
Head/Tail
Map

Map



Note

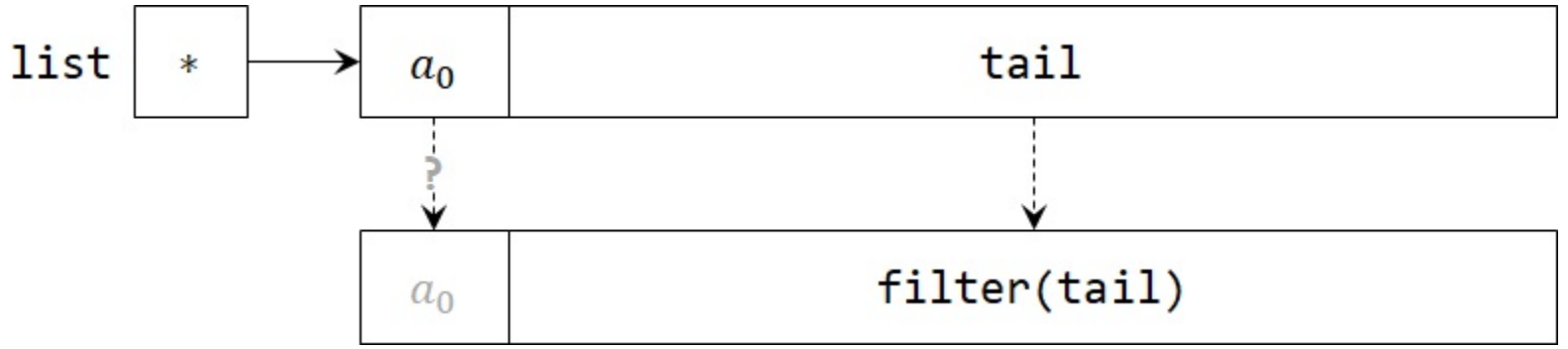
- Computing `f(a₀)` has to be done *lazily**
  - `a₀` is of type `Actually<T>`
  - Can you use a method in `Actually<T>`**?

**Remember, the idea is that *failure* ≡ *missing* and *success* ≡ *present*
*Don't forget to memoize the value too

Tasks

Generate
Iterate
Head/Tail
Map
Filter

Filter



Note

- Checking `pred(a₀)` is missing or present has to be done *lazily**
  - `a₀` is of type `Actually<T>`
  - Can you use a method in `Actually<T>`**?

**Remember, the idea is that *failure* ≡ *missing* and *success* ≡ *present*
*Don't forget to memoize the value too

Finitising Infinite List

Finitising Infinite List

The End

> ### Description
>
> We can make an infinite list *finite* by having a *special value* *(in this case, a special class to use the power of polymorphism)* to indicate the end of the infinite list*.

Note

- `End` is a subclass of `InfiniteList<T>`
- `End` is an inner class *(or nested class)*

*This can also be called `sentinel`, `marker`, `endlist`, *etc*

Finitising Infinite List

Is End?

> ### Description
>
> Write a method `isEnd()` to check if this element is an end element. You need to add in **both** `InfiniteList<T>` and `End`.

Behaviour

Method	InfiniteList< T >	End
`isEnd()`	`false`	`true`

Finitising Infinite List

Head/Tail

Description

Write a method `isEnd()` to check if this element is an end element. You need to add in **both** `InfiniteList<T>` and `End`.

Behaviour

Method	InfiniteList< T >	End
`head()`	*as before*	`java.util.NoSuchElementException`
`tail()`	*as before*	`java.util.NoSuchElementException`

Finitising Infinite List

Limit

> ### Description
>
> Write a method `limit(n)` to create a new `InfiniteList<T>` such that the element at index `n` is `End`.

Example

```
InfiniteList.generate(0, x -> x + 1).limit(2)
// produce [<0>, [<1>, [<2>, -]]]
//   but lazily (and memoized later)
//   so initially it is simply [<0> ?]
//   note: <0> is already computed so
//         no need to make this lazy
//   also, - indicates the "End"
```

Finitising Infinite List

Limit

> ### Description
>
> Write a method `limit(n)` to create a new `InfiniteList<T>` such that the element at index `n` is `End`.

Idea

> If this is **NOT** an infinite list, then the pseudo-code can be something like *(may not use while, could be recursive)*:

```
while ("this is not the n-th element"):
  go to next
// now at n-th element
current element is the "End"
```

Finitising Infinite List

To List

> ### Description
>
> Write a *terminal* method `toList()` to collect the elements in `InfiniteList<T>` into a `java.util.List<T>`.

Hint

- Use `java.util.ArrayList<T>` as it is a subclass of `java.util.List<T>` and it is not an *interface*
- Inserting into `java.util.ArrayList<T>` can be done using an *action*
  - Is there any method in `Actually<T>` we can use?

Finitising Infinite List

To List

> ### Description
>
> Write a *terminal* method **`toList()`** to collect the elements in **`InfiniteList<T>`** into a **`java.util.List<T>`**.

Idea

> If this is **NOT** an infinite list, then the pseudo-code can be something like *(may not use while, could be recursive)*:

```
create an empty ArrayList called arr
while ("the current element exists"):
  add the current element to arr
  go to next element
```

Finitising Infinite List

Take While

> ### Description
>
> Write a method **`takeWhile`** to *truncate* the infinite list and *keep* only elements that satisfies a *condition* *(a predicate)*

Hint

- May look like **`filter`** *(from lecture, but you need to modify this to the current setting)* but make sure that once the condition evaluates to **`false`**, the element returned in **`End`** and do not call the method recursively anymore

Finitising Infinite List

Take While

> ### Description
>
> Write a method **`takeWhile`** to *truncate* the infinite list and *keep* only elements that satisfies a *condition* *(a predicate)*

Idea

> If this is **NOT** an infinite list, then the pseudo-code can be something like *(may not use while, could be recursive)*:

```
while ("the element satisfies the predicate"):
  keep the element
// now is the first element that does not satisfy
current element is the "End"
```

Finitising Infinite List

Count

> ### Description
>
> Write a method **`count`** to count the number of element in the infinite list *(this should return **`long`** because it may be a big number!)*

Idea

> If this is **NOT** an infinite list, then the pseudo-code can be something like *(may not use while, could be recursive)*:

```
count = 0
while ("the element is not the end")
  count++
```

Finitising Infinite List

Count

> ### Description
>
> Write a method `count` to count the number of element in the infinite list *(this should return `long` because it may be a big number!)*

Hint

- Better design is to simply call `reduce`!!!
  - What should be the reduction process?
  - See next part for the explanation for `reduce`

Finitising Infinite List

Reduce

> ### Description
>
> Write a method `reduce` to perform reduction operation *(recap: this is similar to `foldLeft`)*

Example

- Consider a list `[1, [2, [3, [4, -]]]]` with initial result `0` and accumulator function `(x, y) -> x - y`
  - The result is: `((((0 - 1) - 2) - 3) - 4)`
  - Note the computation:
    - starts with the initial result
    - operated with the elements from left to right *(hence, `foldLeft`)*

Finitising Infinite List

Reduce

> ### Description
>
> Write a method **`reduce`** to perform reduction operation *(recap: this is similar to **`foldLeft`**)*

Idea

> If this is **NOT** an infinite list, then the pseudo-code can be something like *(may not use while, could be recursive)*:

```
res = init
while ("the element is not the end")
  res = accumulator(res, element)
```

Tips & Tricks

Tips & Tricks

Memo

Understanding Memo< T >

- You will be utilising the methods from `Memo<T>`
  - In particular, use `get` to ensure that the result will then be memoized
  - Also of interest:
    - `transform`
    - `next`
    - `check`

Tips & Tricks

Memo
Actually

Understanding Actually< T >

- You will be utilising the methods from `Actually<T>`
  - In particular, you will be using the following quite a lot:
    A. `T except(Constant<? extends T> com)`
       - If the value is *missing* then use `com` to generate the result, otherwise simply use the value that is *present*

    B. `<U extends T> T unless(U val)`
       - If the value is *missing* then use `val`, otherwise simply use the value that is *present*

  - Also of interest:
    - `transform`
    - `next`
    - `check`

Tips & Tricks

Eager First

- This is particularly relevant for methods where we have already make the infinite list to be *finite*
  - If you cannot think about what to do for the infinite list
  - Then imagine if you have an eager list *(created appropriately to use the recursive structure rather than using* ***`java.util.List`*** *like in Lab 7)*
    - Can you solve this if you have this kind of eager list?
    - Once you have solved this with this kind of eager list, then you can move to the infinite list
  - You can keep a file called `FiniteList` to capture this kind of `EagerList` *(so that you do not confuse yourself with the eager list from Lab 7)*
    - Do not need to submit `FiniteList`, this is simply useful for your working only

Tips & Tricks

Memo
Actually
Eager First
Start Early

Start Early

- Although the deadline is still ***2 weeks*** from now, there are quite a number of tasks to do
  - So start early and do not wait until last minute
  - You may not have enough time to do it last minute

```
jshell> /exit
|    Goodbye
```