CS2030S

Programming Methodology II

Lab 07

Checkpoint

Checkpoint

Checkpoint

We will provide a minimal implementation for the following two classes that does not follow Lab 5 and Lab 6 requirement:

- `Actually<T>`: A container for value which may or may not be an error
- `Memo<T>`: A container for value that are lazily-evaluated-and-memoized

Checkpoint

Actually

Fields

- `Exception err`: for *failure*
- `T val`: for *success*

Convention

- If `err` is `null` then it is *Failure*
  - `val` has no meaning
- If `err` is not `null` then it is *Success*
  - `val` has meaning

Checkpoint

Actually

Factory Methods

- Static method `err()`: creates a common failure for each of use
  - You should use this if you do not care for the specific error

- Static method `err(Exception)`: creates a failure with the given exception
- Static method `ok(T)`: creates a success with the given value

> You are guaranteed that our test cases will not have error so you do not have to explicitly use `err(Exception)`

Checkpoint

Actually

Value Retrieval

- `except(Constant c)`:
  - if *failure* then uses `c` to initialised some value
  - if *success* then simply returns `val`

- `unless(U u)`:
  - if *failure* then returns `u`
  - if *success* then simply returns `val`

> There is no `unwrap`. But you may add your own if you want *(you can solve Lab 7 and Lab 8 without `unwrap`!)*.

Checkpoint

Actually

Action

- `finish(Action act)`:
    - if *failure* then do nothing
    - if *success* then perform `act.call` with `val`

Checkpoint

Actually

Transformation

- `transform(Immutator<R,T> f)`:
    - if *failure* then propagate error
    - if *success* then returns a new *success* with content transformed into `f.invoke(val)`
- `next(Immutator<Actually<R>,T> f)`:
    - like `transform` but we do not need to wrap it into `Actually<T>` ourselves
    - `f` already wraps this for us!

Checkpoint

Actually

Overridden Methods

- `toString`:
  - if *failure* then returns `"<>"`
  - if *success* then returns the string representation of `val` enclosed within `"<>"`
- `equals`:
  - two *failures* are treated as equals regardless of the exception
  - two *successes* are equal if:
    - both contents are `null`
    - both contents are equal

Checkpoint

Checkpoint
Actually
- Convention
- Factory
- Retrieval
- Action
- Transformation
- Overridden
- Extra!

Actually

Extra Method

- `check(Immutator<Boolean, ? super T> pred)`:
  - this is intended to be used for `InfiniteList` in Lab 8
  - if *failure* then propagate error
  - if *success* then check if predicate `pred`
    - if `true` then we keep the *success*
    - if `false` then we change to *failure*

Checkpoint

Memo

Fields

- `Constant<? extends T> com`: for *unevaluated*
- `Actually<T> val`: for *evaluated*

Convention

- If `com` is `null` then it is *Evaluated*
  - `val` has meaning
- If `com` is not `null` then it is *Unevaluated*
  - `val` has no meaning

Checkpoint

Memo

Factory Methods

- Static method `from(T val)`: creates an *evaluated* value
- Static method `from(Constant<? extends T> com)`: creates an *unevaluated* value

> We only have a single constructor, so we must ensure that one of the argument must be `null`! For best result, you should use `from` and not add other ways to access the private constructor.

Checkpoint

Memo

Value Retrieval

- `get()`:
  - first, force an evaluation by calling `eval()`
  - then return the content of `Actually<T>`
    - however, this may actually be a *failure*
    - in such cases, we simply return `null` *(but it shouldn't happen, you should not try to retrieve a value from an error in the first place!)*

> No other way to retrieve value *(e.g., `except`, `unless`, etc)*.

Checkpoint

Memo

Transformation

- `transform(Immutator<R,T> f)`:
  - still keep the result *unevaluated*
- `next(Immutator<Memo<R>,T> f)`:
  - like `transform` but we do not need to wrap it into `Memo<T>` ourselves
  - `f` already wraps this for us!
- `combine(Memo<S> snd, Combiner<R,T,S> f)`:
  - still keep the result *unevaluated*

Memo

Overridden Methods

- `toString`:
  - if *unevaluated* then returns `?"`
  - if *evaluated* then returns the string representation of the value contained inside `val` *(not `val` itself, but the value inside!)*
- `equals`:
  - we force evaluation of both objects before comparing!
  - no choice, otherwise how would we know they are indeed equal?

Checkpoint

Memo

Extra Method

> No extra method because `check` is really just `transform` in disguise for `Memo`. You can add `check` yourself, and simply call `transform`.

Checkpoint

Note

> This implementation is to avoid errors from being carried forward. But it will still keep the solution somewhat a secret since this will not satisfy many of the requirements of Lab 5 and Lab 6 *(e.g., using inner classes, extending `Lazy`, etc)*.

- You may choose to use this implementation *(in which case, nothing to do on your end)*
- You may choose to use ***your own implementation***
  - In which case, you need to copy the implementation of `Lazy<T>` into the file `Memo.java`
  - Otherwise, `Lab7.h` will not look for `Lazy.java` and your CodeCrunch submission will fail

List

EagerList

Note

- This `EagerList` is ***different*** from the one introduced in lecture
  A. This simply wraps `List` instead of actually having a recursive structure
  B. `generate` is practically `iterate`
     - Because generating a list of constant value is not interesting!

  C. There is no `map` or `filter`.
  D. There is `get(i)` and `indexOf(v)`
- You should read up on Java `List` to get yourself more familiar with this

List

MemoList

At the Start

- Currently `MemoList` is really a copy of `EagerList` but with all the type changed from `MemoList` to `EagerList`
  - In fact, this is done using search-and-replace method!
- We want to make this *memoized*
  - Values are unevaluated unless you need the value
  - Once evaluated, you should not evaluate again

List

MemoList

When do you Need the Value?

- `get(i)`: you need the value when you are requesting for a specific value
    - this may cause a *cascade* of evaluation if created using `generate`
    - but may not cause *cascade* of evaluation if created using `map` or `flatMap` *(more on this later)*
- `indexOf(v)`: you need the value when you are searching for it
    - really actually caused by `equals` in `Memo`
    - will force evaluation from left-most element until the element is found *(or no more element in the list)*

MemoList

What to Do?

Make it Lazily-Evaluated-and-Memoized

- You need to use `Memo`, but where?
    - Is it `Memo<List<T>> list`?
    - Is it `List<Memo<T>> list`?

List

MemoList

What to Do?

Generate with Immutator

- Simply convert the `generate(int n, T seed, Immutator<T, T> f)` from `EagerList` to make it lazily-evaluated-and-memoized

> If `seed = x`, then evaluation is:
> `[x, f(x), f(f(x)), f(f(f(x))), ...]`

##Can be used to easily generate the sequence of natural number!
#But not so easy to generate the Fibonacci sequecne.

List

MemoList

What to Do?

Generate with Combiner

- Now create `generate(int n, T fst, T snd, Combiner<_, _, _> f)`
  - What should be the type of `f` that is more general?
  - Which are *producer* *(producer extends)* and which are *consumer* *(consumer super)*

> If `fst = x` and `snd = y`, then evaluation is:
> `[x, y, f(x, y), f(y, f(x, y)), f(f(x, y), f(y, f(x, y))), ...]`

#Can be used to easily generate Fibonacci sequence!

List

Eager
Memo
- Start
- Needs
- To Do

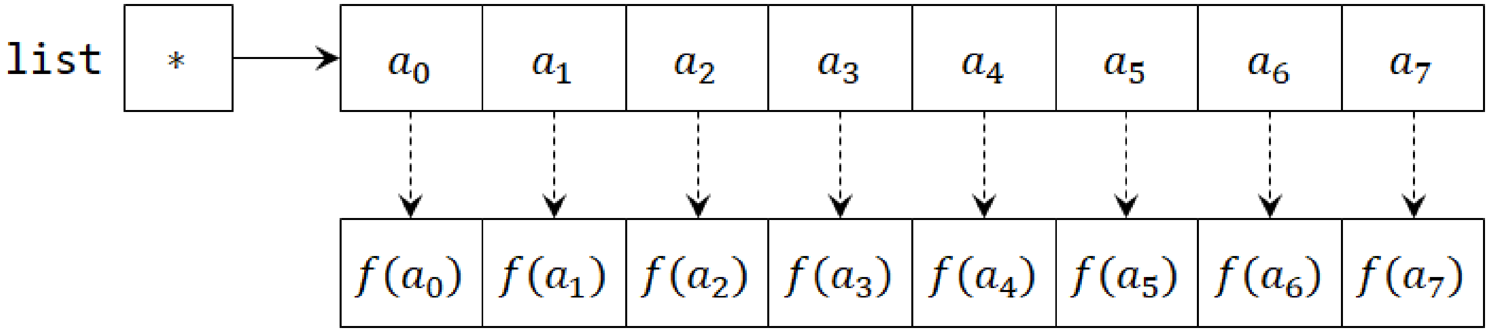
MemoList

What to Do?

Behaviour!

- Simply *invoke* `f` in each element in the list!
  - Do it lazily *(and memoized, of course)*!
  - Which `Memo` transformation is useful for this?
- To retrieve element from list at index `i`, use `get(i)`
- To insert `v` into the end of the list, use `add(v)`

Map

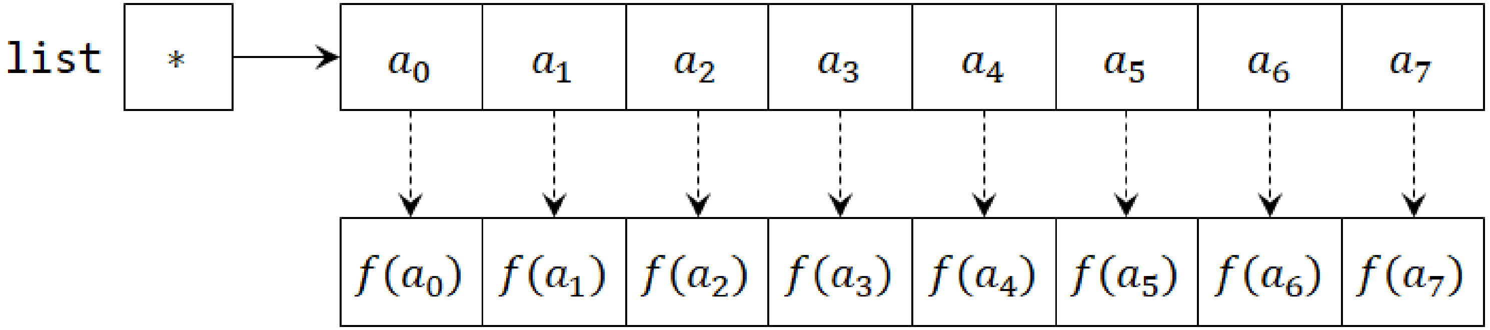


List

MemoList

What to Do?

Map



Important!

- Before we move to `flatMap`, let's take a look at a weird scenario
- Imagine if `f` returns a `MemoList`!
  - What do you think the result looks like?

List

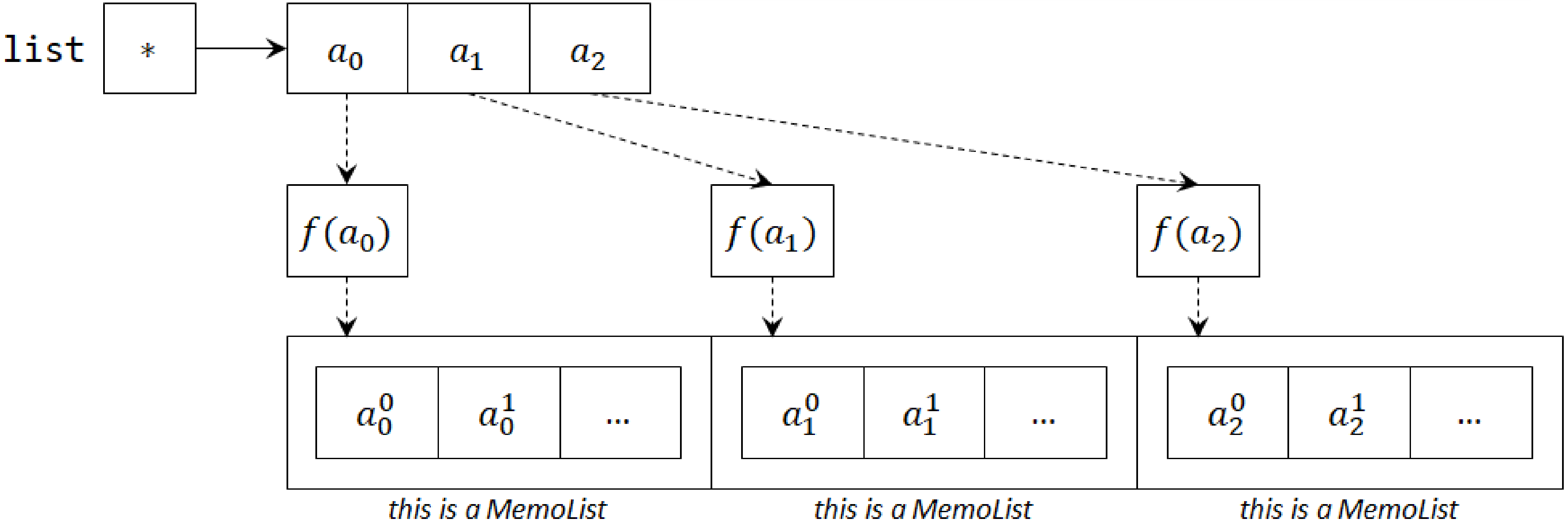
MemoList

What to Do?

Map

Important!

- Before we move to `flatMap`, let's take a look at a weird scenario
- Imagine if `f` returns a `MemoList`!
  - What do you think the result looks like?
  - A ***nested*** ***`MemoList`!***
    - FlatMap is similar but will *flatten* the list so there is nested `MemoList`



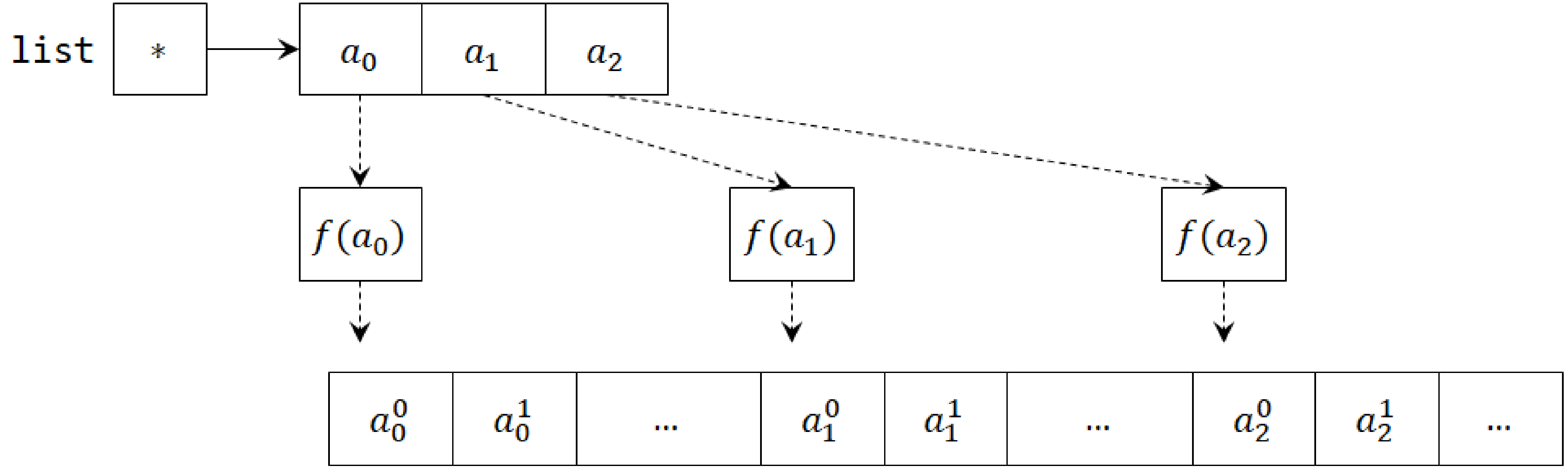
List

Eager
Memo
- Start
- Needs
- To Do

MemoList

What to Do?

FlatMap



no longer nested

What FlatMap Do?

1. Retrieve an element
2. Invoke `f` *(get a `MemoList`)*
3. ???
4. No more nested `MemoList`!

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