PAUL G. ALLEN SCHOOL
OF COMPUTER SCIENCE & ENGINEERING

CSE 341
Section 4

HW1 Debrief, Higher-Order Functions, Closures
Spring 2020

Learning Objectives

- HW1 de-brief (~5 minutes)
- Higher-Order Functions (~35 min)
    - Become familiar with anonymous functions
    - Understand higher order functions and their expressiveness
- Currying and partial application (rest of section)
    - Identify the relationship between currying and partial application

Homework 1 Recap

`is_older` was quite subtle.

(Switch to Emacs)

Homework 1 Recap

Think about what makes a date *d1* earlier than another date *d2*:

1. If the year of *d1* is before the year of *d2* (March 1, 1970 is older than Feb 6, 2010)
2. Or, if the years are equal, then if month of *d1* is earlier (March 1, 1970 is older than April 1, 1970)
3. Or, if both the year and month are equal, then if the day is earlier (March 1, 1970 is older than March 2, 1970)

Key Concepts Review

- Higher-order functions
  - Pass functions around like any data
  - Closures: functions *capture* references to their environment
  - Lexical scoping: variables are captured at time of creation
- Higher-order function idioms:
  - foldl, map, etc.
- Polymorphic functions
  - Functions that are *generic* over the type of arguments

Polymorphic Datatypes

Q3: Consider the following datatype binding that represents a binary tree:

```
datatype ('a, 'b) tree =
  Leaf of 'a | Node of 'b * ('a, 'b) tree
                          * ('a, 'b) tree
```

What expressions could this datatype support, and what are their types?

Anonymous Functions

An Anonymous Function

```
fn pattern => expression
```

- An expression that creates a new function with no name.
- Usually used as an argument to a higher-order function.
- Almost equivalent to the following:

```
let fun name pattern = expression in name end
```

What's the difference? What can you do with one that you can't do with the other?

- The difference is that anonymous functions cannot be recursive!!!

Let's practice! (Q1 and Q2 on Worksheet)

Unnecessary Function Wrapping

What's the difference between the following two expressions?

`(fn xs => tl xs)` vs. `tl`

STYLE POINTS!

- Other than style, these two expressions result in the exact same thing.
- However, one creates an unnecessary function to wrap `tl`.
- This is very similar to this style issue:

`(if ex then true else false)` vs. `ex`

Higher-Order Functions

Functions that are no different from any program data.

An extremely powerful feature! The “defining feature” of functional programming.*

* debatable

fold

- **`fold : 'b list * ('a * 'b -> 'a) * 'a -> 'a`**
  - Returns a “thing” that is the accumulation of the first argument applied to the third arguments elements stored in the second argument.
  - Processes the list in reverse order!
  - Example:

    ```
    fold([1,2,3], (fn (a,b) => a + b), 0) === 6
    ```

Higher-Order Functions

Worksheet Q4! (~5mins)

Higher-Order Functions

What is the type of `fold`?

In what order does `fold` process its elements?

Is there the *one true type* for a `fold` function? Why/Why not?

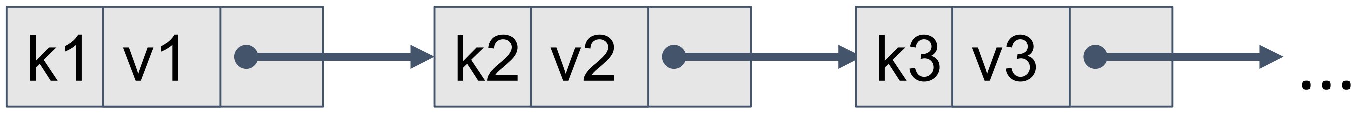
Higher-Order Functions

- More practice (select problems of Q4 of worksheet)

Higher-Order Functions

Let’s look at an association list representation of a map and some operations (Emacs)

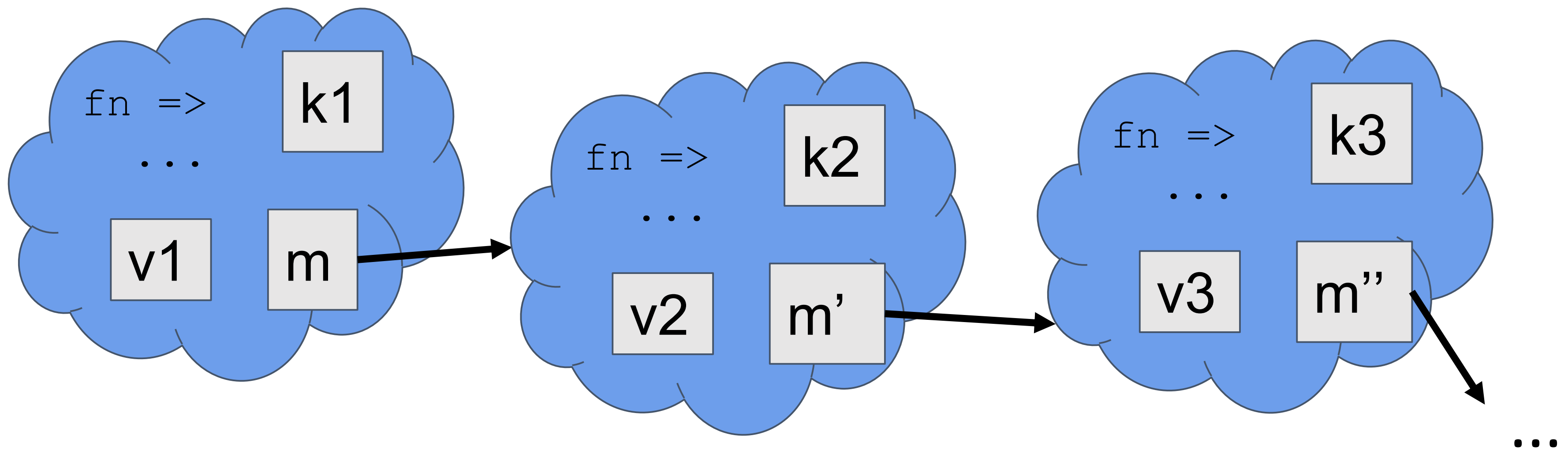
Association Lists



Closure-Based Representation

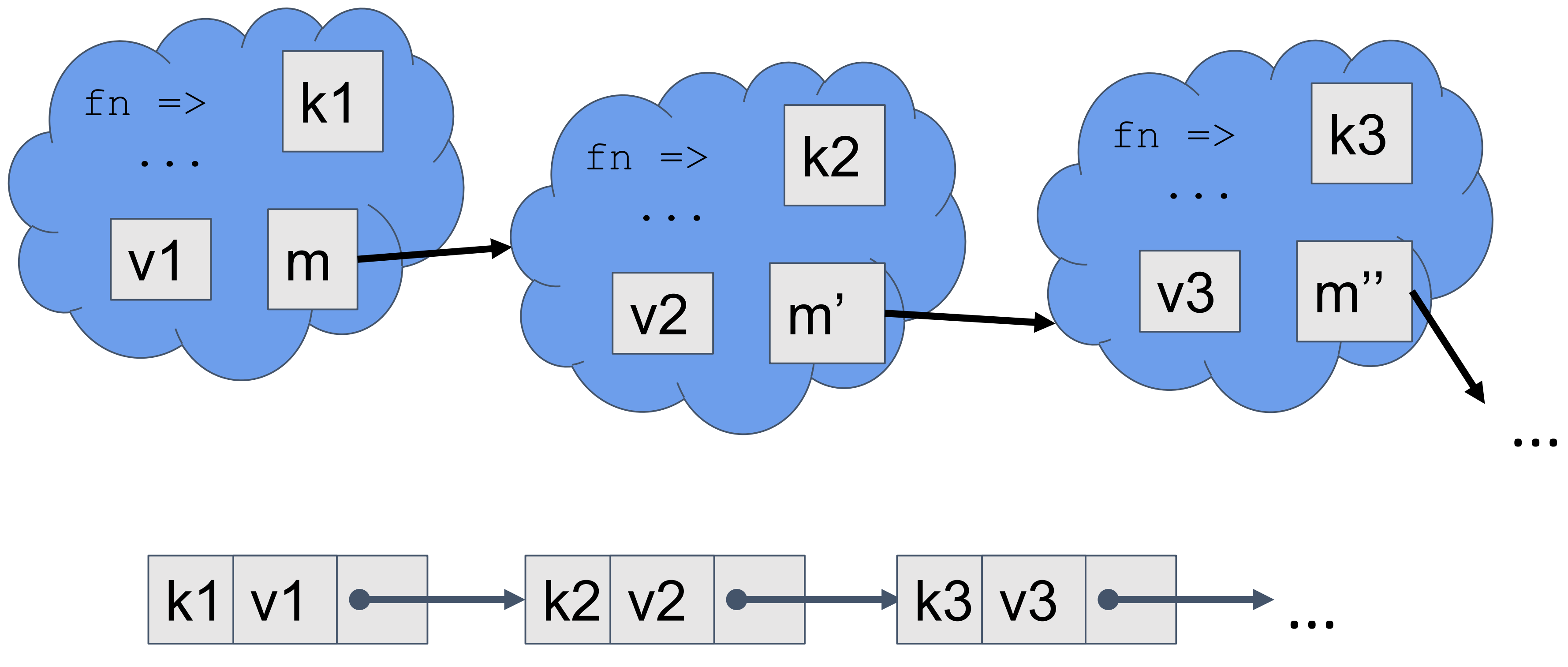
- The function (map!) returned by add captures:
  - the inserted key (k)
  - the inserted value (v)
  - the original map (m)

Closure-Based Representation



Does this look familiar?

Closure-Based Representation



Benefits of this representation

- Remove is O(1)
- Map is O(1) (kinda!)
    - Only ends up transforming values accessed later (emacs)
    - Although the result can be more expensive computationally (why?)