CS450

Structure of Higher Level Languages

Lecture 22: Encoding mutability with heaps

Tiago Cogumbreiro

Today we learn about...

- Motivating example on mutability
- Implementing shared "mutable" state
- Usage examples
- Contracts

How to implement mutation without mutable constructs?

Motivating example

- Calling function b must somehow access variable a which is defined after its creation.

```
; Env: []
(define b (lambda (x) a))
; Env: [(b . (closure ?? (lambda (x) a))]
(define a 20)
; Env: [(b . (closure ?? (lambda (x) a)) (a . 20)]
(b 1)
```

Shared "mutable" state with immutable data-structures

Why immutability?

Benefits

- A necessity if we use a language without mutation (such as Haskell)
- Parallelism: A great way to implement fast and safe data-structures in concurrent code (look up copy-on-write)
- Development: Controlled mutation improves code maintainability
- Memory management: counters the problem of circular references (notably, useful in C++ and Rust, see example)

> Encoding shared mutable state with immutable data-structures is a great skill to have.

Heap

> We want to design a data-structure that represents a ***heap*** (a shared memory buffer) that allows us to: ***allocate*** a new memory cell, ***load*** the contents of a memory cell, and ***update*** the contents of a memory cell.

Constructors

- `empty-heap` returns an empty heap
- `(heap-alloc h v)` creates a new memory cell in heap h whose contents are value v
- `(heap-put h r v)` updates the contents of memory handle r with value v in heap h

Selectors

- `(heap-get h r)` returns the contents of memory handle r in heap h

Heap usage

```
(define h empty-heap)             ; h is an empty heap
(define r (heap-alloc h "foo")) ; stores "foo" in a new memory cell
```

> What should the return value of `heap-alloc`?

- Should `heap-alloc` return a copy of h extended with "foo"? How do we access the memory cell pointing to "foo"?
- Should `heap-alloc` return a handle to the new memory cell? How can we access the new heap?

Heap usage

```
(define h empty-heap)          ; h is an empty heap
(define r (heap-alloc h "foo")) ; stores "foo" in a new memory cell
```

What should the return value of `heap-alloc`?

- Should `heap-alloc` return a copy of h extended with "foo"? How do we access the memory cell pointing to "foo"?
- Should `heap-alloc` return a handle to the new memory cell? How can we access the new heap?

Function `heap-alloc` must return a ***pair*** `eff` that contains the new heap and the memory handle.

```
(struct eff (state result) #:transparent)
```

Heap usage example

Spec

```
(define h1 empty-heap)            ; h is an empty heap
(define r (heap-alloc h1 "foo")) ; stores "foo" in a new memory cell
(define h2 (eff-state r))
(define x (eff-result r)) ;
(check-equal? "foo" (heap-get h2 x)) ; checks that "foo" is in x
(define h3 (heap-put h2 x "bar"))    ; stores "bar" in x
(check-equal? "bar" (heap-get h3 x)) ; checks that "bar" is in x
```

Handles must be unique

We want to ensure that the handles we create are **unique**, otherwise allocation could overwrite existing data, which is undesirable.

Spec

```
(define h1 empty-heap)          ; h is an empty heap
(define r1 (heap-alloc h1 "foo")) ; stores "foo" in a new memory cell
(define h2 (eff-state r1))
(define x (eff-result r1))
(define r2 (heap-alloc h2 "bar")) ; stores "foo" in a new memory cell
(define h3 (eff-state r2))
(define y (eff-result r2))
(check-not-equal? x y)  ; Ensures that x ≠ y
(check-equal? "foo" (heap-get h3 x))
(check-equal? "bar" (heap-get h3 y))
```

How can we implement

a memory handle?

A simple heap implementation

- Let a handle be an integer
- Recall that the heap only grows (no deletions)
- A handle matches the number of elements already present in the heap
- When the heap is empty, the first handle is 0, the second handle is 1, and so on.

Heap implementation

- We use a hash-table to represent the heap because it has a faster random-access than a linked-list (where lookup is linear on the size of the list).
- We wrap the hash-table in a struct, and the handle (which is a number) in a struct, for better error messages. And because it helps maintaining the code.

```
(struct heap (data) #:transparent)
(define empty-heap (heap (hash)))
(struct handle (id) #:transparent)
(struct eff (state result) #:transparent)
(define (heap-alloc h v)
  (define data (heap-data h))
  (define new-id (handle (hash-count data)))
  (define new-heap (heap (hash-set data new-id v)))
  (eff new-heap new-id))
(define (heap-get h k)
  (hash-ref (heap-data h) k))
(define (heap-put h k v)
  (define data (heap-data h))
  (cond
    [(hash-has-key? data k) (heap (hash-set data k v))]
    [else (error "Unknown handle!")]))
```

Contracts

Contracts

Adding some sanity to highly dynamic code.

- Design-by-contract: idea pioneered by Bertrand Meyer and pushed in the programming language **Eiffel**, which was recognized by ACM with the Software System Award in 2006.
- Contracts are pre- and post-conditions each unit of code must satisfy (***e.g.***, a function)
- In some languages, notably F* and Dafny, pre- and post-conditions are checked at compile time!

Bibliography

Design by Contract, in Advances in Object-Oriented Software Engineering, eds. D. Mandrioli and B. Meyer, Prentice Hall, 1991.

Contracts in Racket

> Use `define/contract` rather than `define` to test the validity of each parameter and the return value.

- The → operator takes a predicate for each argument and one predicate for the return value
  For instance: `(→ symbol? real? string?)` declares that the first parameter is a symbol, the second parameter is numeric, and the return value is a string.

Example

```
(define/contract (f x y)
  ; Defines the contract
  (-> symbol? real? string?)
  (format "(~a, ~a)"))
```

Contracts examples

Read up on Racket's manual entry on: data-structure contracts

- `real?` for numbers
- `any/c` for any value
- `list?` for a list
- `listof number?` for a list that contains numbers
- `cons?` for a pair
- `(or/c integer? boolean?)` either an integer or a boolean
- `(and/c integer? even?)` an integer that is an even number
- `(cons/c number? string?)` a pair with a number and a string
- `(hash/c symbol? number?)` a hash-table where the keys are symbols and the keys are numbers