CS450

Structure of Higher Level Languages

Lecture 14: Implementing definitions

Tiago Cogumbreiro

Introducing the λ_D

Language λ_D: Terms

We highlight in red an operation that produces a side effect: ***mutating an environment***.

$$\frac{e \Downarrow_E v \qquad E \leftarrow [x := v]}{(\texttt{define}\ x\ e) \Downarrow_E \texttt{void}}\ (\texttt{E-def})$$

$$\frac{t_1 \Downarrow_E v_1 \qquad t_2 \Downarrow_E v_2}{t_1; t_2 \Downarrow_E v_2}\ (\texttt{E-seq})$$

Language λ_D: Expressions

> Because we have side-effects, the order in which we evaluate each sub-expression is important.

$$v \Downarrow_E v \qquad (\texttt{E-val})$$

$$x \Downarrow_E E(x) \qquad (\texttt{E-var})$$

$$\lambda x.t \Downarrow_E (E, \lambda x.t) \qquad (\texttt{E-lam})$$

$$\frac{e_f \Downarrow_E (E_f, \lambda x.t_b) \qquad e_a \Downarrow_E v_a \qquad E_b \leftarrow E_f + [x := v_a] \qquad t_b \Downarrow_{E_b} v_b}{(e_f\ e_a) \Downarrow_E v_b} \ (\texttt{E-app})$$

> Can you explain why the order is important?

Language λ_D: Expressions

> Because we have side-effects, the order in which we evaluate each sub-expression is important.

$$v \Downarrow_E v \qquad (\texttt{E-val})$$

$$x \Downarrow_E E(x) \qquad (\texttt{E-var})$$

$$\lambda x.t \Downarrow_E (E, \lambda x.t) \qquad (\texttt{E-lam})$$

$$\frac{e_f \Downarrow_E (E_f, \lambda x.t_b) \qquad e_a \Downarrow_E v_a \qquad E_b \leftarrow E_f + [x := v_a] \qquad t_b \Downarrow_{E_b} v_b}{(e_f\ e_a) \Downarrow_E v_b} \ (\texttt{E-app})$$

> Can you explain why the order is important? Otherwise, we might evaluate the body of the function e_b without observing the assignment $x := v_a$ in E_b.

Mutable operations on environments

Mutable operations on environments

Put

$E \leftarrow [x := v]$

Take a reference to an environment E and mutate its contents, by adding a new binding.

Push

$E \leftarrow E' + [x := v]$

Create a new environment referenced by E which copies the elements of E' and also adds a new binding.

Making side-effects explicit

Mutation as a side-effect

Let us use a triangle ▶ to represent the order of side-effects.

$$\frac{e \Downarrow_E v \quad \blacktriangleright \quad E \leftarrow [x := v]}{(\texttt{define}\ x\ e) \Downarrow_E \texttt{void}}\ (\texttt{E-def})$$

$$\frac{t_1 \Downarrow_E v_1 \quad \blacktriangleright \quad t_2 \Downarrow_E v_2}{t_1 ; t_2 \Downarrow_E v_2}\ (\texttt{E-seq})$$

$$\frac{e_f \Downarrow_E (E_f, \lambda x.t_b) \ \blacktriangleright\ e_a \Downarrow_E v_a \ \blacktriangleright\ E_b \leftarrow E_f + [x := v_a] \ \blacktriangleright\ t_b \Downarrow_{E_b} v_b}{(e_f\ e_a) \Downarrow_E v_b}\ (\texttt{E-app})$$

Implementing side-effect mutation

Making the heap explicit

> We can annotate each triangle with a heap, to make explicit which how the global heap should be passed from one operation to the next. In this example, defining a variable takes an input global heap H and produces an output global heap H_2.

$$\frac{\blacktriangleright_H \quad e \Downarrow_E v \quad \blacktriangleright_{H_1} \quad E \leftarrow [x := v] \quad \blacktriangleright_{H_2}}{\blacktriangleright_H \; (\texttt{define}\ x\ e) \Downarrow_E \texttt{void} \blacktriangleright_{H_2}}\ (\texttt{E-def})$$

UMass Boston

Let us use our rule sheet!

$$\frac{e \Downarrow_E v \quad \blacktriangleright \quad E \leftarrow [x := v]}{(\texttt{define}\ x\ e) \Downarrow_E \texttt{void}}\ \texttt{(E-def)}$$

$$\frac{t_1 \Downarrow_E v_1 \quad \blacktriangleright \quad t_2 \Downarrow_E v_2}{t_1 ; t_2 \Downarrow_E v_2}\ \texttt{(E-seq)}$$

$$\frac{e_f \Downarrow_E (E_f, \lambda x.t_b) \quad \blacktriangleright \quad e_a \Downarrow_E v_a \quad \blacktriangleright \quad E_b \leftarrow E_f + [x := v_a] \quad \blacktriangleright \quad t_b \Downarrow_{E_b} v_b}{(e_f\ e_a) \Downarrow_E v_b}\ \texttt{(E-app)}$$

$$v \Downarrow_E v \qquad \texttt{(E-val)}$$

$$x \Downarrow_E E(x) \qquad \texttt{(E-var)}$$

$$\lambda x.t \Downarrow_E (E, \lambda x.t) \qquad \texttt{(E-lam)}$$

Examples

Evaluating Example 2

```
(define b (lambda (x) a))
(define a 20)
(b 1)
```

Input

```
E0: []
---
Env: E0
Term: (define b (lambda (y) a))
```

Evaluating Example 2

```
(define b (lambda (x) a))
(define a 20)
(b 1)
```

Input

```
E0: []
---
Env: E0
Term: (define b (lambda (y) a))
```

Output

```
E0: [
  (b . (closure E0 (lambda (y) a)))
]
Value: #<void>
```

$$\dfrac{\dfrac{}{\lambda y.a \Downarrow_{E_0} (E_0, \lambda y.a)} \quad \blacktriangleright \quad \dfrac{}{E_0 \leftarrow [b := (E_0, \lambda y.a)]}}{(\texttt{define}\ b\ \lambda y.a) \Downarrow_{E_0} \texttt{void}}$$

Example 2: step 2

Input

```
E0: [
  (b . (closure E0 (lambda (y) a)))
]
---
Env: E0
Term: (define a 20)
```

Example 2: step 2

Input

```
E0: [
  (b . (closure E0 (lambda (y) a)))
]
---
Env: E0
Term: (define a 20)
```

Output

```
E0: [
  (a . 20)
  (b . (closure E0 (lambda (y) a)))
]
Value: #<void>
```

$$\frac{\dfrac{}{20 \Downarrow_{E_0} 20} \quad \blacktriangleright \quad \dfrac{}{E_0 \leftarrow [a := 20]}}{(\texttt{define}\ a\ 20) \Downarrow_{E_0} \texttt{void}}$$

Example 2: step 3

Input

```
E0: [
  (a . 20)
  (b . (closure E0 (lambda (y) a)))
]
---
Env: E0
Term: (b 1)
```

Example 2: step 3

Input

```
E0: [
  (a . 20)
  (b . (closure E0 (lambda (y) a)))
]
---
Env: E0
Term: (b 1)
```

Output

```
E0: [
  (a . 20)
  (b . (closure E0 (lambda (y) a)))
]
E1: [ E0
  (y . 1)
]
Value: 20
```

$$\frac{b \Downarrow_{E_0} (E_0, \lambda y.a) \;\blacktriangleright\; 1 \Downarrow_{E_0} 1 \;\blacktriangleright\; E_1 \leftarrow E_0 + [y := 1] \;\blacktriangleright\; a \Downarrow_{E_1} 20}{(b\ 1) \Downarrow_{E_0} 20}$$

Example 3

```
(define (f x) (lambda (y) x))
(f 10)
```

Input

```
E0: []
---
Env: E0
Term: (define (f x) (lambda (y) x))
```

Example 3

```
(define (f x) (lambda (y) x))
(f 10)
```

Input

```
E0: []
---
Env: E0
Term: (define (f x) (lambda (y) x))
```

Output

```
E0: [
  (f . (closure E0
          (lambda (x) (lambda (y) x))))
]
Value: void
```

Example 3

```
(define (f x) (lambda (y) x))
(f 10)
```

Input

```
E0: []
---
Env: E0
Term: (define (f x) (lambda (y) x))
```

Output

```
E0: [
  (f . (closure E0
         (lambda (x) (lambda (y) x))))
]
Value: void
```

$$\frac{\lambda x.\lambda y.x \Downarrow_{E_0} (E_0, \lambda x.\lambda y.x)}{(\mathtt{define}\ f\ \lambda x.\lambda y.x) \Downarrow_{E_0} \mathtt{void}}$$

Example 3

```
(define (f x) (lambda (y) x))
(f 10)
```

Input

```
E0: []
---
Env: E0
Term: (define (f x) (lambda (y) x))
```

Output

```
E0: [
  (f . (closure E0
           (lambda (x) (lambda (y) x))))
]
Value: void
```

$$\frac{\lambda x.\lambda y.x \Downarrow_{E_0} (E_0, \lambda x.\lambda y.x) \quad \blacktriangleright \quad E_0 \leftarrow [f := (E_0, \lambda x.\lambda y.x)]}{(\mathtt{define}\ f\ \lambda x.\lambda y.x) \Downarrow_{E_0} \mathtt{void}}$$

Example 3

Input

```
E0: [
  (f . (closure E0
          (lambda (x) (lambda (y) x))))
]
---
Env: E0
Term: (f 10)
```

Example 3

Input

```
E0: [
  (f . (closure E0
          (lambda (x) (lambda (y) x))))
]
---
Env: E0
Term: (f 10)
```

Output

```
E0: [
  (f . (closure E0
          (lambda (x) (lambda (y) x))))
]
E1: [ E0 (x . 10) ]
Value: (closure E1 (lambda (y) x))
```

Example 3

Input

```
E0: [
  (f . (closure E0
          (lambda (x) (lambda (y) x))))
]
---
Env: E0
Term: (f 10)
```

Output

```
E0: [
  (f . (closure E0
          (lambda (x) (lambda (y) x))))
]
E1: [ E0 (x . 10) ]
Value: (closure E1 (lambda (y) x))
```

$$\dfrac{\dfrac{E_0(f) = (E_0, \lambda x.\lambda y.x)}{f \Downarrow_{E_0} (E_0, \lambda x.\lambda y.x)} \quad \dfrac{}{10 \Downarrow_{E_0} 10} \quad \dfrac{}{E_1 \leftarrow E_0 + [x := 10]} \quad \dfrac{}{\lambda y.x \Downarrow_{E_1} (E_1, \lambda y.x)}}{(f\ 10) \Downarrow_{E_0} (E_1, \lambda y.x)}$$

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