CS450

Structure of Higher Level Languages

Lecture 12: Finite-streams, evaluating expressions

Tiago Cogumbreiro

Finite streams

Finite streams

The type `set`

> A `set` is a finite stream of strings.

```
(define-type set
  ; A function that takes 0 args and builds either:
  (->
    ; an empty set OR an element and the rest of the stream
    (U set-empty set-add)
  )
)
; set-empty denotes the end of the stream, so it has no fields
(struct set-empty ())
; set-add is akin to stream-add: holds a string and the rest of the stream
(struct set-add ([first : String] [rest : set]))
```

An example of a finite stream

Here is an example of a set $\{"a", "b", "c"\}$

```
(thunk
  (set-add "a"
    (thunk
      (set-add "b"
        (thunk
          (set-add "c" (thunk (set-empty))))))))
```

which is similar to building a list, but with `thunk`s interleaved

```
(cons "a"
  (cons "b"
    (cons "c" empty)))
```

Printing the elements of a finite-stream

```
(: print-set (-> set Void))
```

```
(define (print-set s)
  (match (s)              ; Call s to build the next element:
    [(set-empty) (void)] ; If there are no more elements, void does nothing
    [(set-add h l)       ; Otherwise,
      (displayln h)      ; print the element
      (print-set l)      ; and loop
    ]
  )
)
```

UMass Boston

Copying a finite stream

```
(: copy (→ set set))
```

Finite streams

```
(define (copy l)
  (thunk
    (match (l)
      [(set-empty) (set-empty)]
      [(set-add h l)
        (set-add h (copy l))
      ]
    )
  )
)
```

Lists

```
(define (copy l)

  (match l
    [(list) (list)]
    [(cons h l)
      (cons h (copy l))
    ]
  )

)
```

> Similarly to infinite streams, when building a stream from another, we must `thunk` BEFORE we `match`.

Faulty copy implementation

```
(: faulty-copy-set (-> set set))
(define (faulty-copy-set s)
  (match (s)
    [(set-empty) (thunk (set-empty))]
    [(set-add h s)
      (thunk (set-add h (faulty-copy-set s)))]))
```

- Calling `(faulty-copy-set s)` asks `s` to build the first element *before* building its copy.
- The problem is that when you ask `(faulty-copy-set s)` to build its *first* element, you are actually requesting `s` to build its *second* element.
- Solution: a `(thunk ...)` **must** surround any `(match (s) ...)`

Understanding stream-fold

Understanding `stream-fold`

Stream-fold behaves like the following function, which returns a list of the updates to the accumulator. Use this function as the blueprint to your solution.

```
(define (fold2 f a l)
  (match l
    [(list) (list)]
    [(cons h l)
      ; h = e1
      ; l = [e2, e3]
      ; a1 = f e1 a
      (define a1 (f h a))
      (displayln a1)
      (define result (fold2 f a1 l))
      ; result = [a1, a2, a3]
      (cons a result)
      ; expected output: [a, f e1 a, f e2 (f e1 a), f e3 (f e2 (f e1 a))]
    ]))
```

Evaluating expressions

Evaluating expressions

> Our goal is to implement an evaluation function that takes an expression and yields a value.

```
expression = value | variable | function-call
value = number
function-call = ( expression+ )
```

UMass Boston

How do we evaluate an expression

What is an expression?

```
expression = value | variable | function-call
```

> How do we evaluate a value? **The evaluation of a value `v` is `v` itself.**

```
(check-equal? 10 (eval-exp (r:number 10)))
```

> How do we evaluate a function call? **The evaluation of a function call evaluates each expression from left to right and then it applies the function to the arguments.**

UMass Boston

Example

> How do we evaluate a function call? **The evaluation of a function call evaluates each expression from left to right and then it applies the function to the arguments.**

```
  (eval-exp
    '(-
       (+ 3 2)
       (* 5 2)) )
```

①
← evaluate '-
← evaluate '(+ 3 2)
← evaluate '(* 5 2)

```
= ((eval-exp '-)
   (eval-exp '(+ 3 2))
   (eval-exp '(* 5 2)))
```

②
← evaluate '+, evaluate 3, evaluate 2
← evaluate '*, evaluate 5, evaluate 2

```
= ((eval-exp '-)
   ((eval-exp '+) 3 2)
   ((eval-exp '*) 5 2))
```

③
← numbers are values, so just return those
← numbers are values, so just return those

How do we evaluate arithmetic operators?

```
= ((eval-exp '-)
   ((eval-exp '+) 3 2)
   ((eval-exp '*) 5 2))
```

```
= (-
    (+ 3 2)
    (* 5 2))
```

```
← Evaluate '- as function -
← Evaluate '+ as function +
← Evaluate '* as function *
```

Evaluation of arithmetic expressions

1. When evaluating a number, just return that number
2. When evaluating an arithmetic symbol, return the respective arithmetic function
3. When evaluating a function call evaluate each expression and apply the first expression to the remaining ones

> Essentially evaluating an expression **translates** our AST nodes as a Racket expression.

UMass Boston

Implementing eval-exp...

Specifying `eval-exp`

- We use the AST we defined in Lesson 5, not datums.
- Assume function calls are binary.

```
(check-equal? (r:eval-exp (r:number 5)) 5)
(check-equal? (r:eval-exp (r:number 10)) 10)
(check-equal? (r:eval-exp (r:variable '+)) +)
(check-equal?
  (r:eval-exp
    (r:apply
      (r:variable '+)
      (list (r:number 10) (r:number 5))))
  15)
```

Implementing `eval-exp`

We are using the AST as structs, not datums. Assume function calls are binary.

```
(: r:eval-exp (-> r:expression Number))
(define (r:eval-exp exp)
  (match exp
    ; If it's a number, return that number
    [(r:number v) v]
    ; If it's a function with 2 arguments
    [(r:apply (r:variable f) (list arg1 arg2))
      (define func (r:eval-builtin f))
      (func (r:eval-exp arg1) (r:eval-exp arg2))
    ]
  )
)
```

UMass Boston

Implementing `r:eval-builtin`

Spec

```
(check-equal? (r:eval-builtin '+) +)
(check-equal? (r:eval-builtin '-) -)
```

Solution

```
(define (r:eval-builtin sym)
  (cond [(equal? sym '+) +]
        [(equal? sym '-) -]
        [(equal? sym '*) *]))
```

UMass Boston

Handling functions with an arbitrary number of parameters

(required for Homework 4)

Function `apply`

Function `(apply f args)` applies function `f` to the list of arguments `args`.

Examples

```
(check-equal? (apply + (list 1 2 3 4)) 10)
```

> Example: implement `(sum l)` that returns the summation of all members in `l` using `apply`.

Spec

```
(check-equal? (sum (list)) 0)
(check-equal? (sum (list 1 2 3 4)) 10)
```

Solution

```
(define (sum l) (apply + l))
```

Handling multiple-args without apply

> Some multi-arg operations can be implemented without the need of `apply`.

Implement `(sum l)` without using `apply`.

Spec

```
(check-equal? (sum (list)) 0)
(check-equal? (sum (list 1 2 3 4)) 10)
```

Solution

```
(define (sum l)
  (cond
    [(empty? l) 0]
    [else (+ (first l) (sum (rest l)))]))
```

Implementing functions with multi-args

> How could we implement a function with multiple parameters, similar to `+`? **Use the `.` notation.**

The dot `.` notation declares that the next variable represents a list of zero or more parameters.

Examples

```
(define (map-ex f . args)
  (map f args))

(check-equal? (list 2 3 4) (map-ex (curry + 1) 1 2 3))


(define (sum . l) (foldl + 0 l))
(check-equal? 6 (sum 1 2 3))
```