Discussion 09

Tail Calls, Interpreters

Aditya Balasubramanian
aditbala [at] berkeley [dot] edu

Slides available at `teaching.aditbala.com`

Announcements 📣

- Ants due Thursday (7/28)
  - Turn in on Wednesday (7/27) for EC
- HW05 due Thursday (7/28)
- Scheme Project starts Thursday (7/28)
  - Two checkpoints instead of one

Scope 

Dynamic Scope vs Lexical Scope

- Lexical Scope
  - Parent of function is where function is **defined**
  - All Python functions and `lambda` functions in Scheme
- Dynamic Scope
  - Parent of function is where function is **called**
  - Occurs in `mu` procedures
  - Will implement this yourself in project!

Worksheet

Tail Recursion 

Tail Recursion

- What is Tail Recursion?
  - Recursion while keeping track of our result, instead of accumulating back up at the end
- How do we make our function Tail Recursive?
  - Usually have a helper function to keep track of information from previous frames

Tail Calls

- What are Tail Calls?
  - The recursive calls in Tail recursion
- When do they occur?
  - When a function calls another function as the final action of the current frame
  - This also means that the current frame can be discarded (different than normal recursion)
- Why Tail Recursion?
  - More efficient
  - Less Space

Consider this implementation of `factorial` that is NOT tail recursive:

```
(define (factorial n)
  (if (= n 0)
      1
      (* n (factorial (- n 1)))))
```

- Recursive call is on the last line, but is not the last expression to be evaluated.
- Must accumulate the result of `n * recursive calls` at the end

Visualization of NON tail-recursive `factorial`

```
(factorial 6)
(* 6 (factorial 5))
(* 6 (* 5 (factorial 4)))
(* 6 (* 5 (* 4 (factorial 3))))
(* 6 (* 5 (* 4 (* 3 (factorial 2)))))
(* 6 (* 5 (* 4 (* 3 (* 2 (factorial 1))))))
(* 6 (* 5 (* 4 (* 3 (* 2 1)))))
(* 6 (* 5 (* 4 (* 3 2))))
(* 6 (* 5 (* 4 6)))
(* 6 (* 5 24))
(* 6 120)
720
```

Tail recursive `factorial`:

```
(define (factorial n)
  (define (fact-tail n result)
    (if (= n 0)
        result
        (fact-tail (- n 1) (* n result))))
  (fact-tail n 1))
```

- `fact-tail` makes a single recursive call to `fact-tail`, and that recursive call is the last expression to be evaluated

Visualization of tail-recursive `factorial`

```
(factorial 6)
(fact-tail 6 1)
(fact-tail 5 6)
(fact-tail 4 30)
(fact-tail 3 120)
(fact-tail 2 360)
(fact-tail 1 720)
(fact-tail 0 720)
720
```

Tail Context

- To determine if a function call is a Tail Call, we have to look for Tail Contexts
- Following expressions are Tail Contexts
  1. the second or third operand in an `if` expression
  2. any of the non-predicate sub-expressions in a `cond` expression (i.e. the second expression of each clause)
  3. the last operand in an `and` or an `or` expression
  4. the last operand in a `begin` expression's body
  5. the last operand in a `let` expression's body
- What is the tail call in `(begin (+ 2 3) (- 2 3) (* 2 3))`
- `(* 2 3)` is a tail call because it is the last operand expression to be evaluated

Worksheet

Interpreters 

Interpreters

- What is an Interpreter?
  - A program used to understand other programs
- Will be using the `Calculator` language to look at Scheme Syntax
  - Simplified to have a few operations
  - `+`, `-`, `*`, `/`

Syntax

- Numbers (Numbers) -> `4`, `7`
- Arithmetic Procedures (Strings) -> `"+"`
- Call expressions are similar to Scheme Lists
  - Going to use `Pair` Class to represent `Calculator` expressions
  - `(+ 2 3)` -> `Pair("+", Pair(2, Pair(3, nil)))`

`Pair` Class

- What is the `Pair` Class?
  - How we represent expressions
- What is in the class?
  - Has a `first` and `rest` attribute
  - Must include `nil` at the end of expression
- If `p` is a `Pair`
  - `p.first` is the operator
  - `p.rest` is the list of operands
  - `p.rest.first` is the first operand

```
class Pair:
    """Represents the built-in pair data structure in Scheme."""
    def __init__(self, first, rest):
        self.first = first
        if not scheme_valid_cdrp(rest):
            raise SchemeError("cdr can only be a pair, nil, or a pro
        self.rest = rest

    def map(self, fn):
        """Maps fn to every element in a list, returning a new
        Pair.

        >>> Pair(1, Pair(2, Pair(3, nil))).map(lambda x: x * x)
        Pair(1, Pair(4, Pair(9, nil)))
        """
        assert isinstance(self.rest, Pair) or self.rest is nil, \
            "rest element in pair must be another pair or nil"
        return Pair(fn(self.first), self.rest.map(fn))
```

Q10

How do we actually interpret?

- Evaluation
  - Interpreter determines type of expression and follows rules to evaluate
  - Know what we are inputting
- Applying
  - Once evaluation is done, can apply procedure to operands
  - Simple for `Calculator`, apply function to operands

Evaluation Rules

- Numbers
  - Self Evaluating
- Names
  - Look up in `OPERATORS` Dictionary
  - `"+"` maps to a Python function that does the operation
- Call Expression
  - Evaluate operator, evaluate operands, apply operator to operands
- Evaluation is recursive!

creds to @Poggenkemper

Apply Rules

- Apply once Evaluation is done
  - Simple call for `Calculator` to apply function to operands
  - More complicated for Scheme since users can define their own procedures

creds to @Poggenkemper

Code for Evaluating and Applying

```
def calc_eval(exp):
    if isinstance(exp, Pair): # Call expressions
        return calc_apply(calc_eval(exp.first), exp.rest.map(calc_
    elif exp in OPERATORS:       # Names
        return OPERATORS[exp]
    else:                        # Numbers
        return exp

def calc_apply(fn, args):
"""Applies a calculator expression to a list of numbers"""
    return fn(args)
```

`Q9`, `Q6`, `Q7`

Thank you!!!

Attendance Form -> https://tinyurl.com/adit-disc09

Anon Feedback -> https://tinyurl.com/adit-anon