Lecture 4: Inference Rules

Steven Holtzen
s.holtzen@northeastern.edu

CS4400/5400 Fall 2024

1 Inference Rules

- Inference rules are powerful notation for describing recursive algorithms and indutively-defined data-structures.
- We will use them to describe interpreters, type systems, and prove things about programs.
- We'll start by revisiting a recursive function we saw earlier, the factorial function:

$$\texttt{fact}(n) = \begin{cases} 1 & \text{if } x = 0 \\ n \times \texttt{fact}(n-1) & \text{otherwise.} \end{cases} \tag{1}$$

- The above uses standard "math notation" you may have seen before in another class. Here is another way of writing the same program using Racket:

```
> (define (fact n)
    (if (equal? n 0) 1 (× n (fact (- n 1)))))
> (fact 4)
24
```

- There is a *third* style of notation for describing the `fact` function called **inference rules**, which look like this:

$$\frac{}{\texttt{fact}(0) = 1}\ (\textsc{FactBase}) \qquad \frac{\texttt{fact}(n-1) = v}{\texttt{fact}(n) = n \times v}\ (\textsc{FactInd})$$

- Inference rules consist of two pieces: **premises** and **conclusions**. The premises go above the horizontal line, and the conclusions go below the horizontal line.
- An inference rule with no premises is called an **axiom**. The FACTBASE rule above is an axiom. Axioms are always true; these are the base cases of your induction.
- The FACTIND rule has both premises and conclusions. We read it as: "if $\texttt{fact}(n-1) = v$ for some value v, then $\texttt{fact}(n) = n \times v$."

2 Proving things using inference rules

- We can use these inference rules prove that `fact(3) = 6` by drawing a **derivation tree**.
- The process works like this: first, we copy and paste what we want to prove as a conclusion to some to-be-determined inference rule:

$$\frac{???}{\texttt{fact(3) = 6}} \tag{2}$$

- Now we ask: *which inference rule can we apply to show this desired conclusion?* There is only one possible rule, `FactInd`:

$$\frac{\texttt{fact(2)} = v}{\texttt{fact(3)} = 3 \times v}\ (\textsc{FactInd}) \tag{3}$$

- However, we are not done: we haven't yet shown what `fact(2)` is equal to. We can only terminate our proof when we hit a base case (i.e., an axiom). So, we keep going and apply the `FactInd` rule again:

$$\dfrac{\dfrac{\texttt{fact(1)} = v_1}{\texttt{fact(2)} = 2 \times v_2}\ (\textsc{FactInd})}{\texttt{fact(3)} = 3 \times v}\ (\textsc{FactInd}) \tag{4}$$

- We are still not done, there is one more step of derivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\texttt{fact(0) = 1}}\ (\textsc{FactBase})}{\texttt{fact(1)} = 1 \times 1}\ (\textsc{FactInd})}{\texttt{fact(2)} = 2 \times 1}\ (\textsc{FactInd})}{\texttt{fact(3)} = 3 \times 2 \times 1}\ (\textsc{FactInd}) \tag{5}$$

- Now we are done. The above tree is called a derivation tree, and it is typically drawn bottom-up. Each leaf (top of the tree) must be an axiom.
- Drawing this derivation tree is essentially *running a factorial program by hand*. Each recursive case (an instance of FACTIND) requires performing a recursive call, which is either another inductive case or a base case. We label the result of the recursive call with a variable (like v). Once we hit a base case, we can replace the variables with concrete comptued values. This is why in Equation (5) we substituted computed quantities (numbers) for variables ($v, v_1, ...$).
- Once again we are in a host semantics situation: how do we interpret the meaning of numbers and operations like $\times$ in derivation trees like those in Equation (5)? Once again we will use Racket as our host semantics: we will assume that these numbers and operations represent Racket numbers and Racket operations.

3 Inference Rules for a Calculator Interpreter

- Now let's use our new inference rule notation to describe a calculator interpreter
- We will define a function `eval` that evaluates each syntactic calculator term to a number
- Recall the syntax for a calculator language:

```
(struct (enum n))
(struct (eadd e1 e2))
(struct (emul e1 e2))
```

- We can define inference rules that describe the semantics of our calculator language:

$$\frac{}{\texttt{(eval (enum n))} = n}\ (\text{E-N}{\small\text{UM}}) \qquad \frac{\texttt{(eval e1)} = v_1 \qquad \texttt{(eval e2)} = v_2}{\texttt{(eval (eadd e1 e2))} = (+\, v_1\, v_2)}\ (\text{E-A}{\small\text{DD}})$$

$$\frac{\texttt{(eval e1)} = v_1 \qquad \texttt{(eval e2)} = v_2}{\texttt{(eval (emul e1 e2))} = (*\, v_1\, v_2)}\ (\text{E-M}{\small\text{UL}})$$

- Now we can draw derivation trees that describe evaluating our interpreter. For example, we can run the program `(+ (+ 1 2) 3)`:

$$\frac{\dfrac{\dfrac{}{\texttt{(eval 1) = 1}} \qquad \dfrac{}{\texttt{(eval 2) = 1}}}{\texttt{(eval (+ 1 2)) = 3}}\ (\text{E-A}{\small\text{DD}}) \qquad \dfrac{}{\texttt{(eval (3)) = 3}}\ (\text{E-N}{\small\text{UM}})}{\texttt{(eval (+ (+ 1 2) 3)) = 6}}\ (\text{E-A}{\small\text{DD}})$$

  Note: we will sometimes use surface syntax instead of abstract syntax, and elide the name of rules, to make drawing derivation trees more concise.

- Notice how these rules visualize running the interpreter. For this reason, we will sometimes refer to drawing a derivation tree for a program as **running an interpreter by hand**.

4 Substitution Semantics for Let Language

- The "evaluates to" inductive definition is so common that we will introduce special notation to describe it: the relation $e \Downarrow v$ says e evaluates to v. This means exactly the same thing as `eval(`e`) =` v; it's just easier (and common convention) to write it this way. This is sometimes called the *natural semantics* of a programming language.
- Let's use this notation to describe the semantics for the `let` language from last lecture:

$$\frac{}{\texttt{(enum n)} \Downarrow n}\ (\text{E-Num}) \qquad \frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2}{(\texttt{eadd}\, e_1\ e_2) \Downarrow (+\ v_1\ v_2)}\ (\text{E-Add})$$

$$\frac{e_1 \Downarrow v_1 \qquad e_2[x \mapsto v_1] \Downarrow v_2}{\texttt{(let x e}_1\ \texttt{e}_2\texttt{)} \Downarrow v_2}\ (\text{E-Let})$$

- Recall that the syntax $e_2[x \mapsto v_1]$ substitutes the value v_1 for x in e_2.
- Let's look at the rule for E-Let closely. What it says is: if e_1 evaluates to a value v_1 and $e_2[x \mapsto v_1]$ evaluates to v_2, then the expression `(let x e`$_1$ `e`$_2$`)` evaluates to v_2. Notice how there is a data dependence between the two premises: the value v_1 is referred to by both the rule for evaluating e_1 and e_2. This is permitted: we just need to ensure that this value v_1 is *the same* in both locations.
- Now we can run a `let` program by hand. Let's run the program `(let x 10 (+ x 5))`:

$$\dfrac{\dfrac{}{\texttt{10} \Downarrow 10}\ (\text{E-Num}) \qquad \dfrac{\dfrac{}{\texttt{10} \Downarrow 10} \qquad \dfrac{}{\texttt{5} \Downarrow 5}}{\texttt{(+ x 5)}[x \mapsto 10] \Downarrow 15}\ (\text{E-Add})}{\texttt{(let x 10 (+ x 5))} \Downarrow 15}\ (\text{E-Let})$$

- We will handle errors differently in inference rule notation than in Racket. In our inference rule notation, it is considered an error if we get "stuck" (meaning, there is no rule to apply to make progress).
- For example, if we try to run the program that consists solely of an identifier `x`, then we get stuck immediately because there is no rule to evaluate an identifier.

5 Environment-passing Semantics for Let Language

- The problem with substitution is that it is inefficient: we wouldn't want to actually use substitution to implement a programming language.
- A more practical solution to implementing the `let` language is to use an **environment** that maps identifiers to values. The environment will store the current value for a particular variable, and when an identifier is encountered during execution its value can be looked up in the environment.
- We will use an immutable hash table that maps strings (identifiers) to numbers as our environment. Immutable hash tables have the follow usage in Racket:

```
> (define my-tbl (hash)) ; declare an immutable empty hash–table
> (define my-tbl-with-x (hash-set my-tbl "x" 10)) ; add ”x” with value 10
> (hash-has-key? my-tbl-with-x "x")
#t
> (hash-has-key? my-tbl "x") ; check if my–tbl contains ”x”
#f
> (hash-ref my-tbl-with-x "x") ; look up ”x” in my–tbl–with–x
10
```

- Here is an example implementation in Racket of an environment-passing interpreter:

```
;;; type expr =
;;;   | add of expr × expr
;;;   | mul of expr × expr
;;;   | num of number
;;;   | elet of string × expr × expr
(struct eadd (e1 e2) #:transparent)
(struct enum (n) #:transparent)
(struct elet (id assignment body) #:transparent)
(struct eident (id) #:transparent)

;;; eval : expr → (string, number) hashtable → number
;;; evaluates an expression e in environment env to a number
(define (eval e env)
  (match e
    [(enum n) n]
    [(eadd e1 e2) (+ (eval e1 env) (eval e2 env))]
    [(elet id assignment body)
     (let× [(v-assgn (eval assignment env))
            (new-env (hash-set env id v-assgn))]
       (eval body new-env))]
    [(eident id)
     (if (hash-has-key? env id)
         (hash-ref env id)
         (error "unbound identifier"))]))

(check-equal? (eval (elet "x" (enum 10) (eident "x")) (hash)) 10)
(check-equal? (eval (elet "x" (enum 10)
                          (elet "x" (enum 20) (eident "x"))) (hash)) 20)
```

- **Check**: Why does the above implementation implement lexical scope? What property of the hash tables ensures this?

6 Inference Rules for Environment-passing Semantics of Let Language

- Let's get more practice with inference rules by using them to to describe the environment-passing interpreter for the `let` language
- We will use the symbol ρ (pronounced "rho") for our environment. Looking up an identifier x in ρ is written $\rho[x]$ (this corresponds to `hash-ref`). Immutably updating the table ρ to set x equal to v is denoted $\rho[x \mapsto v]$ (this corresponds to `hash-set`). Checking if ρ contains x is denoted $x \in \rho$ (this corresponds with `hash-has-key?`).
- Now we can write down the environment-passing semantics of the `let` language in terms of inference rules. These rules will define a function $\langle e, \rho\rangle \Downarrow v$, which is read "expression e with environment ρ evaluates to value v":

$$\frac{}{\langle \texttt{(enum n)}, \rho\rangle \Downarrow n}\ (\text{E-N}\textsc{um}) \qquad \frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2}{\texttt{(eadd}\, e_1\; e_2\texttt{)} \Downarrow (+\; v_1\; v_2)}\ (\text{E-A}\textsc{dd})$$

$$\frac{x \in \rho \qquad \rho[x] = v}{\langle x, \rho\rangle \Downarrow v}\ (\text{E-I}\textsc{dent})$$

$$\frac{\langle e_1, \rho\rangle \Downarrow v_1 \qquad \langle e_2, \rho[x \mapsto v_1]\rangle \Downarrow v_2}{\langle \texttt{(let x e}_1\texttt{ e}_2\texttt{)}, \rho\rangle \Downarrow v_2}\ (\text{E-L}\textsc{et})$$

- **Notice**: the above rules differ quite substantially from the substitution-based semantics for `let`. One big difference is that we now have a rule for evaluating identifiers.
- Now we can draw derivation trees that visualize running our environment-passing interpreter.
- **Example**: Draw the derivation tree that corresponds to evaluating `(let "x" 10 x)` with the environment initially empty ($\rho = \{\}$):

$$\frac{\dfrac{}{\texttt{10} \Downarrow 10}\ (\text{E-N}\textsc{um}) \qquad \dfrac{x \in \{x \mapsto 10\} \qquad \{x \mapsto 10\}[x] = 10}{\langle x, \{x \mapsto 10\}\rangle \Downarrow 10}\ (\text{E-I}\textsc{dent})}{\langle \texttt{(let "x" 10 x)}, \{\}\rangle \Downarrow 10}\ (\text{E-L}\textsc{et})$$