Lecture 11: Typed If-then-Else and Lambda Calculus

Steven Holtzen
s.holtzen@northeastern.edu

CS4400/5400 Fall 2024

1 Typed If-then-Else

- Let's continue the process of adding language features and trying to design type checkers that eliminate runtime errors in the Racket host semantics.
- The next feature we'll add is if-then-else. Here is its syntax and semantics:

```
;;; type expr =
;;;   | enum of number
;;;   | ebool of boolean
;;;   | eite of expr * expr * expr
(struct enum (n) #:transparent)
(struct ebool (b) #:transparent)
(struct eite (grd thn els) #:transparent)

;;; type value =
;;;    | vbool of bool
;;;    | vnum of num
(struct vbool (b) #:transparent)
(struct vnum (n) #:transparent)

;;; value->bool : value -> bool
(define (value->bool v)
  (match v
    [(vbool b) b]
    [_ (error "not a number")]))

;;; interp : expr -> value
(define (interp e)
  (match e
    [(enum n) (vnum n)]
    [(ebool b) (vbool b)]
    [(eite g thn els)
     (if (value->bool (interp g))
                      (interp thn)
                      (interp els))]))
```

2 Designing a type system for the if-then-else language

- We defined our notion of "going wrong" as causing the host semantics of Racket to raise a runtime error
- So, what programs cause this to happen? Here are some:

```
(eite (enum 10) (ebool #t) (ebool #f))
(eite (eite (ebool #t) (enum 10) (enum 30)) (ebool #t) (ebool #f))
```

- Clearly, the requirement is: the guard of an `if` must evaluate to a Boolean.
- So, let's try to design a typesystem that enforces this. Our possible types are `tbool` and `tnum`:

```
(struct tnum () #:transparent)
(struct tbool () #:transparent)
```

- Just like last lecture, we start by defining a function `type-of` that computes the type of an if-then-else term. Let's try:

```
;;; type-of : expr -> itetype
(define (type-of e)
  (match e
    [(enum n) (tnum)]
    [(ebool b) (tbool)]
    [(eite g thn els)
     (if (and (equal? (type-of g) (tbool))
              (equal? (type-of thn) (type-of els)))
         ???
         (raise-type-error))]))
```

- The base cases are easy, but we hit a wrinkle: what do we replace `???` with? It's not obvious what to do here! There are two choices that each have to make different compromises:

  1. *Evaluate the guard to see which branch is taken*. This replaces `???` with:

     ```
     (match (interp g)
       [(vbool #t) (type-of thn)]
       [(vbool #f) (type-of els)]
       [_ (raise-type-error)])
     ```

     This prioritizes expressivity at the expense of performance: computing the type of a term requires evaluating code, which might be very expensive, especially once we start to add more language features. In general, we avoid using `interp` in our type-checkers for this reason.

  2. *Require that both branches of the if return the same type, and return that type.* This replaces `???` with:

     ```
     (if (and (equal? (type-of g) (tbool))
              (equal? (type-of thn) (type-of els)))
         (type-of thn)
         (raise-type-error))
     ```

     This prioritizes performance at the expense of expressivity. With this choice of type-checking, some programs that do not cause runtime errors will not be well-typed, such as the following program:

     ```
     (ite (ebool #t) (enum 10) (ebool 30))
     ```

3 Typing rules for if-then-else

- We will go with Option #2: we will require that the two branches of if-then-else evaluate to the same type. In practice all languages make this design choice so that type-checking is efficient.
- Here are the inference rules for the type system for this language:

$$\frac{}{\texttt{(enum n)} : \texttt{num}}\ (\text{T-N}\textsc{um}) \qquad \frac{}{\texttt{(ebool b)} : \texttt{bool}}\ (\text{T-B}\textsc{ool})$$

$$\frac{g : \texttt{bool} \quad \texttt{thn} : t \quad \texttt{els} : t}{\texttt{(ite g thn els)} : t}$$

4 Towards a Typed Lambda Calculus

- Continuing along, let's now explore how to add types to the lambda calculus
- Let's start with the lambda calculus with numbers and addition. Here is the syntax and semantics, which should be quite familiar:

```
#lang racket
;;; type lexpr =
;;;   | eident of string
;;;   | elam of string * typ * lexpr
;;;   | eapp of expr * lexpr
;;;   | enum of expr * lexpr
;;;   | eadd of expr * lexpr
(struct eident (s) #:transparent)
(struct elam (id body) #:transparent)
(struct eapp (e1 e2) #:transparent)
(struct enum (n) #:transparent)
(struct eadd (e1 e2) #:transparent)

;;; subst : expr -> string -> expr -> expr
;;; performs the substitution e1[x |-> e2] with lexical scope
(define (subst e1 id e2)
  (match e1
    [(eident x)
     (if (equal? x id) e2 (eident x))]
    [(elam x body)
     (if (equal? x id)
         (elam x body) ; shadowing case; do nothing
         (elam x (subst body id e2)) ; non-shadowing case
         )]
    [(eapp f arg)
     (eapp (subst f id e2) (subst arg id e2))]
    [(eadd l r)
     (eadd (subst l id e2) (subst r id e2))]
    [(enum n) (enum n)]))

;;; interp : lexpr -> value
;;; runs a lambda term and produces a value
(define (interp e)
  (match e
    [(eident x) (error "unbound ident")]
    [(elam id x) (elam id x)]
    [(enum n) (enum n)]
    [(eadd e1 e2)
     (match (list (interp e1) (interp e2))
       [(list (enum n1) (enum n2)) (enum (+ n1 n2))])]
    [(eapp e1 e2)
     (match (interp e1)
       [(elam id body)
        (let* [(arg-v (interp e2))
               (subst-body (subst body id arg-v))]
          (interp subst-body))])]))
```

5 Attempt 1 at Designing a Type System for the Lambda Calculus

- As usual we ask: *how can things go wrong* (i.e., cause a runtime error in the Racket host semantics)? Well, the most straightforward way is calling something that is not a function. There are a few ways this can happen, such as:

  ```
  (interp (eapp (enum 10) (enum 30)))
  (interp (eapp (eapp (elam "x" (enum 10)) 20) 30))
  (interp (eapp (elam "x" (x 5)) (enum 10)))
  ```

- Let's design a type system to prevent non-functions from being called.

- First, what are our types? Well, we know that types are collections of values, and clearly there are two kinds of values: numbers and functions. So, we might try the following type of types:

  ```
  ;;; type typ =
  ;;;    tnum
  ;;;    tfun
  (struct tnum ())
  (struct tfun ())
  ```

- We can try to design a typechecker that attempts to typecheck programs using these rules:

  ```
  ;;; type-of : expr -> typ
  ;;; computes the type of a term, or raises an type exception if no valid type
  ;;; exists for that term
  (define (type-of e)
    (match e
      [(enum _) (tnum)]
      [(elam id body) (tfun)]
      [(eapp e1 e2)
       ???]
      [(eident id) !?!?]))
  ```

- What do we replace ??? with (let's forget about !?!? for now)? Well, first we should check if the type of `e1` is a function. This is easy enough:

  ```
  ;;; type-of : expr -> typ
  ;;; computes the type of a term, or raises an type exception if no valid type
  ;;; exists for that term
  (define (type-of e)
    (match e
      [(enum _) (tnum)]
      [(elam id body) (tfun)]
      [(eapp e1 e2)
       (if (equal? (type-of e1) (tfun))
           ???
           (raise-type-error))]
      [(eident id) !?!?]))
  ```

- Now what do we replace ??? with? It's not clear what the type of *evaluating* `e1` is! Remember, we want to do this *without* interpreting `e1`. Even worse, we need to make sure we're calling `e1` with the right type of argument. If we don't do that, `e1` might misuse its argument in some way.

6 Attempt 2: Simple Types

- The previous discussion showed us that we need to track the type of a function's argument and return value in our typesystem. This shouldn't be surprising: many languages that use types, like C, Java, etc. all have this requirement.
- We address this by making our types more fine-grained. For functions, we will keep track of their return and argument type:

  ```
  ;;; type typ =
  ;;;   | tnum
  ;;;   | tfun of typ * typ
  (struct tnum () #:transparent)
  (struct tfun (targ tret) #:transparent) ; targ is argument type, tret is return type
  ```

  These are called **simple types**, which means they are either (1) a non-recursive **base type**, like numbers; or (2) a **function type**.

- Now we can try to design a type checker that associates terms with simple types:

  ```
  (define (type-of e)
    (match e
      [(enum n) (tnum)]
      [(elam id body) ???]
      ...))
  ```

- We immediately hit a problem: *what is the type of a lambda*? It's not obvious! For example, what is the type of this program:

  ```
  (elam "x" (eident "x"))
  ```

- To resolve this issue, we will make a simplifying assumption: *we will annotate the arguments to lambdas with their types*. We call these **type annotations**. Again, this is familiar: many languages that have typesystems do this, and this is why. So, we will adjust our syntax of lambda terms to include a type for the argument:

  ```
  ;;; type typ =
  ;;;   | tnum
  ;;;   | tfun of typ * typ
  (struct tnum () #:transparent)
  (struct tfun (t1 t2) #:transparent)

  ;;; type lexpr =
  ;;;   | eident of string
  ;;;   | elam of string * typ * lexpr
  ;;;   | eapp of expr * lexpr
  ;;;   | enum of expr * lexpr
  (struct eident (s) #:transparent)
  (struct elam (id typ body) #:transparent)
  (struct eapp (e1 e2) #:transparent)
  (struct enum (n) #:transparent)
  ```

- For surface syntax we often abbreviate `(elam x t e)` as $\lambda x : t.e$, for instance, we may write the following:

  $$(\lambda x : \texttt{num}.\ x)\ 10$$

7 Type Environments

- Imagine we want to type check the following application:

$$(\lambda x : \texttt{num}.\ x)\ 10$$

- What do we need to do? We need to:
  - Check that the function $(\lambda x : \texttt{num}.\ x)$ takes a number as an argument. This is easy to see: we can just check its type annotation.
  - Determine what type $(\lambda x : \texttt{num}.\ x)$ returns when we call it with a number, and return that type. This is a bit trickier. We would like to determine the type of a function by computing the type of its body. But, the challenge is that the function body here consists only of an identifier x: *what is its type?*
- To solve this problem, during type checking, we will need to keep track of *the types of identifiers*. This is done using a **type environment**, which is a hash-table that will map identifiers to their types.
- Note that again we will need to be careful of scope: an identifier might be shadowed by another identifier of a different type, and our type environment will need to properly account for this.

8 A Simply-Typed Lambda Calculus Implementation

- Finally we are ready for the simply-typed lambda calculus (STLC) in all its glory:

```
;;; type typ =
;;;   | tnum
;;;   | tfun of typ * typ
(struct tnum () #:transparent)
(struct tfun (t1 t2) #:transparent)

;;; type lexpr =
;;;   | eident of string
;;;   | elam of string * typ * lexpr
;;;   | eapp of expr * lexpr
;;;   | enum of expr * lexpr
;;;   | eadd of expr * lexpr
(struct eident (s) #:transparent)
(struct elam (id typ body) #:transparent)
(struct eapp (e1 e2) #:transparent)
(struct enum (n) #:transparent)

;;; type-of : typeenv -> expr -> typ
;;; computes the type of a term, or raises an type exception if no valid type
;;; exists for that term
(define (type-of tenv e)
  (match e
    [(enum n) (tnum)]
    [(eident id)
     ; look up id in the type environment; if it's not there, raise a type error
     (hash-ref tenv id (lambda () (raise-type-error)))]
    [(elam id typ body)
     ; 1. let T be the type of body with the type environment extended with [id |-> typ]
     ; return a function typ -> T
     (let* ([extended-env (hash-set tenv id typ)]
            [T (type-of extended-env body)])
       (tfun typ T))]
    [(eapp e1 e2)
     ; let t1 -> t2 be the type of e1; if this is not a function type, then raise an error
     ; check that e2 has type t1
     ; return type t2
     (match (type-of tenv e1)
       [(tfun t1 t2)
        (if (equal? (type-of tenv e2) t1)
            t2
            (raise-type-error))]
       [_ (raise-type-error)])]))
```

9 Inference Rules for STLC

- We will likely not have time to discuss these in class today, but I will include the inference rules here in case we have time:

$$\frac{}{\Gamma \vdash \texttt{(enum n)} : \texttt{num}} \text{ (T-Num)} \qquad \frac{x \in \Gamma \qquad \Gamma(x) = t}{\Gamma \vdash x : t} \text{ (T-Ident)}$$

$$\frac{\Gamma \cup \{x \mapsto t_1\} \vdash e : t_2}{\Gamma \vdash (\lambda x : t_1 . e) : t_1 \to t_2} \text{ (T-Lam)}$$

$$\frac{\Gamma \vdash e_1 : t_1 \to t_2 \qquad \Gamma \vdash e_2 : t_1}{\Gamma \vdash (e_1\ e_2) : t_2} \text{ (T-App)}$$

10 Consequences of Simple Types

- Let's evaluate the quality of STLC on the type design axes:
  - Soundness: Our type system is indeed sound. No well-typed STLC programs can trigger a runtime error in the Racket host semantics.
  - Efficiency. It is indeed efficient to type-check programs.
  - Usability. It seems reasonable that programmers could annotate their lambda terms with types. It seems so straightforward that it could even feasibly be automated; we'll see later that this is indeed possible.
  - What about expressivity? Is it the case that there is a lambda term that is not well-typed, but does not cause a runtime error in our interpreter?
- It turns out that we lose expressivity: there are terms that run without raising a runtime error that are not well-typed according to the rules of STLC. Here is one such term:

$$\Omega = \Big((\lambda x.(x\ x))\ (\lambda x.(x\ x))\Big)$$

- *What is the type of* `x` in each of these expressions? Try for yourself and see what happens; we will discuss this more next lecture.