Lecture 12: Growing the Type System

Steven Holtzen
s.holtzen@northeastern.edu

CS4400/5400 Fall 2024

1 Typing Judgments for STLC

- Recall our syntax for the simply-typed lambda calculus (STLC):

```
;;; type typ
(struct tnum () #:transparent)
(struct tfun (t1 t2) #:transparent)

;;; type lexpr
(struct eident (s) #:transparent)
(struct elam (id typ body) #:transparent)
(struct eapp (e1 e2) #:transparent)
(struct enum (n) #:transparent)
(struct eadd (e1 e2) #:transparent)
```

- We will be making lots of typesystems for the next few lectures, so it makes sense to establish some efficient notation for describing them, similar to how we used big-step semantics to efficiently describe the semantics of programs.
- Once again we will use inference rules. As before, we will first give the inference rules using "code notation". We will define a function `type-of env e` that takes two arguments: a type environment `env` that maps identifiers to types, and a term `e`, shown in Figure 1.
- As in the case with operational semantics, this code notation is a bit too unwieldy, so we use a more concise notation for describing the types of terms.

$$\frac{}{\texttt{(typeof env (enum n)) = tnum}}\ \text{(T-NUM)} \qquad \frac{\texttt{x} \in \texttt{env} \quad \texttt{env[x]} = t}{\texttt{(type-of env (eident x)) =}\ t}\ \text{(T-IDENT)}$$

$$\frac{\texttt{(type-of env[x} \mapsto \texttt{t1] e) = t2}}{\texttt{(type-of env (elam id t1 e)) = (tfun t1 t2)}}\ \text{(T-LAM)}$$

$$\frac{\texttt{(type-of e1 env) = (tfun t1 t2)} \quad \texttt{(type-of env e2) = t1}}{\texttt{(type-of env (e1 e2)) = t2}}\ \text{(T-APP)}$$

Figure 1: Type-system for STLC with code notation.

- First, we design some useful surface syntax. Similar to the λ-calculus, we will use $\lambda x : t \,.\, e$ to denote a lambda term with argument x of type t with body e. As surface syntax for types, instead of writing `(tnum)` we write `num`, and instead of writing `(tfun t1 t2)` we write $t_1 \to t_2$.
- For example, instead of writing `(elam x (tnum) (enum 10))`, we write $\lambda x : \texttt{num} \,.\, 10$.
- Then, similar to how we used the big-step arrow $\Downarrow$ to denote the result of an interpreter, we will introduce some extra notation to denote the result of running `(typeof env e)`.
- **Mathematical typing notation**: We will denote the fact that a term e has type t in type environment Γ as $\Gamma \vdash e : t$ (read as "in context Γ term e has type t").
- Figure 2 uses this notation to concisely describe the type-checker for STLC.
- We will denote the empty type environment as $\{\}$

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

$$\frac{\Gamma \cup \{x : t_1\} \vdash e : t_2}{\Gamma \vdash (\lambda x : t_1 . e) : t_1 \to t_2}\ \text{(T-LAM)} \qquad \frac{\Gamma \vdash e_1 : t_1 \to t_2 \qquad \Gamma \vdash e_2 : t_1}{\Gamma \vdash (e_1\ e_2) : t_2}\ \text{(T-APP)}$$

Figure 2: Type-system for STLC with mathematical notation.

2 Example Typing Derivations

- As usual it helps to see some derivation trees to understand how to use this new notation. Here are some examples. Notice how we use Γ in these rules; it is similar to how we used ρ in Lecture 4 when we were implement the let language using environments.

$$\dfrac{\dfrac{y \in \Gamma \qquad \Gamma(y) = \mathsf{num} \to \mathsf{num}}{\{y : \mathsf{num} \to \mathsf{num}\} \vdash y : \mathsf{num} \to \mathsf{num}}\ \text{(T-IDENT)}}{\{y : \mathsf{num} \to \mathsf{num}\} \vdash (\lambda x : \mathsf{num}\ .\ y) : \mathsf{num} \to (\mathsf{num} \to \mathsf{num})}\ \text{(T-LAM)}$$

$$\dfrac{\dfrac{\dfrac{x \in \Gamma \qquad \Gamma(x) = \mathsf{num}}{\{x : \mathsf{num}\} \vdash x : \mathsf{num}}\ \text{(T-IDENT)}}{\{\} \vdash (\lambda x : \mathsf{num}\ .\ x) : \mathsf{num} \to \mathsf{num}}\ \text{(T-LAM)} \qquad \dfrac{}{\{\} \vdash 10 : \mathsf{num}}\ \text{(T-NUM)}}{\{\} \vdash ((\lambda x : \mathsf{num}\ .\ x)\ 10) : \mathsf{num}}\ \text{(T-APP)}$$

3 Attempting to Typecheck Omega

- Let's try to typecheck $\Omega = ((\lambda x.(x\ x))\ (\lambda x.(x\ x)))$ in the empty type context.
- Let's start by just focusing on the term $\lambda x.(x\ x)$. What type should we give to the argument?
- Let's start by trying to fill in a derivation tree, and just leave the type of the argument as some unknown type t_1 and the return type as some unknown type t_2:

$$\dfrac{\dfrac{\dfrac{x \in \{x : t_1\} \qquad \{x : t_1\}[x] = t_1}{\{x : t_1\} \vdash x : t_1}\ \text{(T-IDENT)} \qquad \dfrac{x \in \{x : t_1\} \qquad \{x : t_1\}[x] = t_1}{\{x : t_1\} \vdash x : t_1}\ \text{(T-IDENT)}}{\{x : t_1\} \vdash (x\ x) : t_2}\ \text{(T-APP)}}{\{\} \vdash \lambda x : t_1.(x\ x) : t_1 \to t_2}\ \text{(T-LAM)}$$

- Now let's try to solve for the unknown types t_1 and t_2.
- We know from the (T-APP) rule that, when we call x, we should get something of type t_2. This means that x must have type $t_1 \to t_2$. But, there is no way this is possible! We know that x has type t_1 because we annotated it that way.
- So, there's no way to give a valid type for this term unless we have a type that satisfies the requirement that $t_1 = t_1 \to t_2$. There are no such types in our simple type system, so there is no valid type for this term.

4 Algebraic Data Types and Match

- Let's continue with our journey of slowly adding features to our language, giving them a semantics, and designing a typechecker to prevent runtime errors.
- We've been using `match` quite extensively so far in this class. Let's make an interpreter for `match` and design a typechecker to prevent type errors involving this feature.
- For our version of `match`, we will simplify things by having only two kinds of structs: `inl` and `inr` (short for "into left" and "into right").
- We can program with these structs in Racket like this:

```
;;; type left-or-right =
;;;   | inl of num
;;;   | inr of num
(struct inl (v) #:transparent)
(struct inr (v) #:transparent)
(define left-or-right (inl 10))
(match left-or-right
 [(inl _) (display "left!")]
 [(inr _) (display "right!")])
```

- The above program prints "left!" A few things are happening:
  - A value of type `left-or-right` can be **either** an `inl` or `inr`
  - The `match` **breaks apart** a value of type `left-or-right` into two **arms**: an arm that handles the `inl` case, and an arm that handles the `inr` case.
- Data types that can be one of many possible types are called **algebraic data types** (ADTs) or **sum types**. You are familiar with this idea: all of our abstract syntax trees have been ADTs.
- ADTs have been steadily added to many of today's modern programming languages like Python, Typescript, and Rust.

5 Designing a Tiny Language with Pattern Matching

- Let's make a tiny language and interpreter to better understand how pattern matching and ADTs work. Here is the initial abstract syntax for our language. To do this, we will extend the STLC with (1) the ability to introduce ADTs using `inr e` and `inl e`, and (2) the ability to decompose ADTs using `match e leftarm rightarm`. We will expect that `leftarm` and `rightarm` are lambda terms.

```
;;; type lexpr =
;;;   | eident of string
;;;   | elam of string * typ * expr
;;;   | eapp of expr * expr
;;;   | enum of expr * expr
;;;   | eadd of expr * expr
;;;   | einl of expr * type
;;;   | einr of expr * type
;;;   | ematch of expr * expr * expr
(struct eident (s) #:transparent)
(struct elam (id typ body) #:transparent)
(struct eapp (e1 e2) #:transparent)
(struct enum (n) #:transparent)
(struct eadd (e1 e2) #:transparent)
(struct einl (e) #:transparent)
(struct einr (e) #:transparent)
(struct ematch (e leftarm rightarm) #:transparent)
```

- Let's give this language a semantics, inspired by how Racket works. The semantics of `inl e` is simple because they are essentially values: their semantics is to evaluate `e` to `e2`, and store the result in `inl e2`. `inr` is similar.

  The semantics for `match e leftarm rightarm` is as follows: first, evaluate `e` to either `inl v` or `inr v`. If you evaluate to `inl v`, call `leftarm` with `v` as an argument; if you evaluate to `inr v`, call `rightarm` with `v` as an argument; otherwise, raise a runtime error.

```
;;; interp : lexpr -> lexpr
;;; runs a lambda term and produces a value
(define (interp e)
  (match e
    [(eident x) (error "unbound ident")]
    [(elam id t x) (elam id t 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 t body)
        (let* [(arg-v (interp e2))
               (subst-body (subst body id arg-v))]
          (interp subst-body))])]
    [(einl e) (einl (interp e))]
    [(einr e) (einr (interp e))]
    [(ematch e leftarm rightarm)
     (match (interp e)
       [(einl v)
        ; call leftarm with v as an argument
        (interp (eapp leftarm v))]
       [(einr v)
        ; call rightarm with v as an argument
        (interp (eapp rightarm v))])]))
```

- As usual we can also summarize these using inference rules:

$$\frac{e \Downarrow v}{\texttt{inr}\ e \Downarrow \texttt{inr}\ v}\ \text{(E-INR)} \qquad \frac{e \Downarrow v}{\texttt{inl}\ e \Downarrow \texttt{inl}\ v}\ \text{(E-INL)}$$

$$\frac{e_g \Downarrow \texttt{inl}\ v \qquad (e_l\ v) \Downarrow v'}{\texttt{match}\ e_g\ e_l\ e_r \Downarrow v'}\ \text{(E-MATCHL)} \qquad \frac{e_g \Downarrow \texttt{inr}\ v \qquad (e_r\ v) \Downarrow v'}{\texttt{match}\ e_g\ e_l\ e_r \Downarrow v'}\ \text{(E-MATCHR)}$$

6 Typechecking Match and ADTs

- Let's design a type system to prevent programs involving `match` from causing Racket runtime errors.
- In class we will walk through this in extensive detail. There are a few key points in the design of this type system that I will note here:
  - First, we need a new type `tsum t1 t2` that denotes a value that can be **either** `t1` or `t2`
  - Now, when designing our typechecker, we hit a challenge: what is the type of `inl 10`? We know that it's a sum, and we know that the left part of the sum is `tnum`, but what is the right part? A key point is that *we cannot know* just by looking at this term what its right part is, so we need to add a type annotation that tells us what type it is.

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

;;; type-of : typeenv -> expr -> typ
;;; computes the type of a term, or raises an type exception if no valid type
(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))]
    [(eadd e1 e2)
     (if (and (equal? (type-of tenv e1) (tnum))
              (equal? (type-of tenv e2) (tnum)))
         (tnum)
         (raise-type-error))]
    [(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)])]
    [(einl e (tsum l r))
     ; check that typeof e = l
     (if (equal? (type-of tenv e) l)
         (tsum l r)
         (raise-type-error))]
    [(einr e (tsum l r))
     ; check that typeof e = r
     (if (equal? (type-of tenv e r))
         (tsum l r)
         (raise-type-error))]
    [(ematch e leftarm rightarm)
     ; check that e has type (tsum l r)
     ; check that leftarm has type l -> t
     ; check that rightarm has type r -> t
     ; return type t
     (match (list (type-of tenv e) (type-of tenv leftarm) (type-of tenv rightarm))
       [(list (tsum l r) (tfun l-t1 l-t2) (tfun r-t1 r-t2))
        (if (and (equal? l l-t1)
                 (equal? r r-t1)
                 (equal? l-t2 r-t2))
            l-t1
            (raise-type-error))
        ]
       [_ (raise-type-error)])]))
```

7 Practical Programming with Types: A Taste of OCaml

- At this point, it's very educational to see an example of an existing typed language that makes use of some of the typing features we've been exploring in this class so far, and how type systems can be translated into more practical languages.
- We will explore **OCaml**, a typed language that looks and behaves a lot like the simply-typed lambda calculus we have been studying. OCaml is not very widely used, but it has been very influential on many more widely-used languages: it has directly influenced the design of TypeScript, Rust, and Scala
- Let's quickly see how the typed language features we've explored so far – typechecking functions, pairs, and algebraic data types – work in OCaml.
- You can easily use OCaml by going to the following link: `https://try.ocamlpro.com/`
- On this webpage, you will see a split window similar to Racket with a *top-level* window and a *interaction window*. Let's type in the interaction window for now.
- Here is a small OCaml program that prints the number 5:

```
> 5
- : int = 5
```

- OCaml comments begin with `(*` and end with `*)`
- OCaml is an expression-oriented language, similar to Racket, but it has very different syntax: it uses an infix-style syntax. Here are some simple OCaml programs:

```
> 5 + 5
- : int = 10

> 5 = 10
- : bool = false

> "hello" = "world" ;;
- : bool = false
```

- OCaml has if expressions, Booleans, etc., which behave in an identical manner to the type systems we've explored so far for these features. Here is how they work syntactically:

```
> if true then 5 else 10
- : int = 5
```

- OCaml has a tyepechecker, so it will reject ill-formed programs:

```
> if 10 then 2 else 5 ;;
  Line 1, characters 3-5:
  Error: This expression has type int but an expression was expected of type bool
  because it is in the condition of an if-statement
```

- OCaml makes a similar compromise to our typechecker for `if`, and rejects if expressions whose branches return different types:

```
> if true then 10 else true ;;
  Line 1, characters 21-25:
  Error: This expression has type bool but an expression was expected of type int
```

- OCaml has support for lambdas and first-class functions. Functions are called in a manner similar to the surface syntax we've been using for the lambda calculus: by leaving a space between a function and its argument. Note that, in OCaml, parenthesis are used for disambiguating binding order, which is quite different from their meaning in Racket.

```
> (fun x -> x + 1) 10 ;;
  - : int = 11
```

  Notice that we were not required to annotate the type of the argument for this function! OCaml has **type inference**, which can infer from usage context that the argument must be of type integer here.

- If you want to annotate the type of a function's argument, you can do that with the following syntax:

```
> (fun (x:int) -> x + 1) 10 ;;
  - : int = 11
```

- OCaml supports let expressions as follows:

```
> let x = 10 in x + 5 ;;
- : int = 15
```

7.1 Pairs, Algebraic Data Types, and Pattern Matching

- OCaml supports pairs with the following syntax:

```
> (true, 10) ;;
- : bool * int = (true, 10)

> fst (true, 10)
- : bool = true

> snd (true, 10)
- : int = 10
```

- OCaml supports ADTs with the following syntax (which should be familiar to you!)

```
> type left_or_right = Inl of int | Inr of int ;;
type left_or_right = Inl of int | Inr of int

> Inl 10 ;;
- : left_or_right = Inl 10
```

  Note: OCaml requires that the first letter of each ADT is capitalized.

- OCaml supports pattern matching for breaking apart ADTs:

```
>  match (Inl 10) with
  | Inl v -> v + 10
  | Inr v -> v + 20 ;;
  - : int = 20
```

- OCaml's type checker will helpfully alert you if you are not handling each case in a pattern match:

```
> match (Inl 10) with
  | Inl v -> v + 101 ;;
- : int = 20
  1 Warning : this pattern-matching is not exhaustive.
  Here is an example of a case that is not matched:
  Inr _
```