Lecture 10: Introduction to Types

Steven Holtzen
s.holtzen@northeastern.edu

CS4400/5400 Fall 2024

1 What are types

- **Types** are collections of values.
- For example, we could define `int` as the collection of all integers (numbers `int` $= \{\cdots, -2, -1, 0, 1, 2, \cdots\}$). Or, we can define `bool` to be the collection truth-values `bool` $= \{T, F\}$.
- Types are quite commonly seen in programming languages. You've likely programmed with types before. For example, in C, you might have the following program:

```
int main() {
  int i = 1;
  i = i + 1;
  printf("%d", i);
  return 0;
}
```

- In programs we often associate types with terms. For example, in the above program, we declared a local variable `i` and associated it with the type `int`. This tells the C compiler (and other programmers) that the name `i` must contain a value from the type `int`. Expressions in C also have types: the expression `i + 1` also has type `int`.
- The process of associating types with terms in a programming language is called a **type system**. This is an intentionally broad definition; we will see many examples of type systems in this class.
- So far we have been programming in Racket using `#lang racket`. Racket running in this mode does not associate types with terms, so we call it an **untyped language**.
- Increasingly, types are being added to today's programming languages. For example, TypeScript adds types to JavaScript. Here is an example of programming with types in TypeScript:

```
var hello : string = "Hello World"
console.log(hello)
```

- The syntax `var hello : string` declares that the variable `hello` has type `string`.

2 Why Types?

- Broadly, there are two motivations for wanting to have types in your programming languages:
  - *Physical reasons*. In order to run programs we need to run it on a low-level CPU that doesn't know about strings, Booleans, etc. Types tell the compiler what shape data has so it can know how to map it into something the CPU understands. This can make writing compilers much easier, and it was the initial motivation for introducing types into programs.
  - *Logical reasons*. Types convey a lot of information about how programs behave to the compiler and to other programmers beyond how that information is represented physically on the computer. Types are used as a means to (1) find and prevent certain kinds of bugs, (2) provide checked specifications for how programs behave, (3) aid in the creation of developer tools, and more.
- Type systems prevent you from writing certain kinds of programs. For example, the TypeScript type system rejects the following program because you are attempting to call the function `add` with values that are not of the correct type:

```
function add(x : number, y : number) {
  return x + y;
}

var hello : string = "Hello World"
var anum : number = 0;

add(hello, anum)
```

- The above program has a **type error** because the type of `hello` is not equal to `number`. Intuitively, this kind of type error occurs when a value is being used in a way it's not meant to be used. In general, a type error occurs whenever it is not possible to find a valid type for any part of the program.
- A program that is free of type errors is called **well-typed**. The process of determining if a program is well-typed is called **type checking**. We often say that a well-typed program "type checks".
- A classic saying that motivates the logical perspective on types is that "well-typed programs do not go wrong", for various definitions of "wrong" [1]. We will explore several notions of being "wrong" as we design type systems throughout this course.
- For the next few weeks we will be designing our own type systems that assign types to programs in order to learn more about how they work and are implemented.
- What are the high-level design goals for a type system for a programming language? Here is an incomplete list:
  - **Soundness**. If a program is well-typed, then it should be free of whatever kind of error the type system is trying to prevent.
  - **Expressivity**. It should be possible to write interesting well-typed programs.
  - **Usability**. A programmer should be able to write well-typed programs without too much trouble.
  - **Efficiency**. It should be faster to run the type-checker than it is to run the program. This is partially motivated by the typical use-case of type systems: we want to run our type-checking algorithm every time the user changes their code, and programs that users write might take a very long time to run.

3 A Type-system for calculators

- Let's design a type-system for a tiny calculator language with the following abstract syntax and semantics (given as an interpreter):

```
;;; type expr =
;;;   | eadd of expr * expr
;;;   | eand of expr * expr
;;;   | elessthan of expr * expr
;;;   | enum of number
;;;   | ebool of bool
(struct eadd (e1 e2) #:transparent)
(struct eand (e1 e2) #:transparent)
(struct elessthan (e1 e2) #:transparent)
(struct enum (n) #:transparent)
(struct ebool (b) #:transparent)

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

;;; value->number : value -> number
(define (value->number v)
  (match v
    [(vnum n) n]
    [_ (error "not a number")]))


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

;;; interp : expr -> value
;;; evaluates an expression to a number
(define (interp e)
  (match e
    [(eadd e1 e2)
     (vnum (+ (value->number (interp e1)) ((value->number (interp e2)))))]
    [(enum n) (vnum n)]
    [(ebool b) (vbool b)]
    [(elessthan e1 e2)
     (vbool (< (value->number (interp e1)) (value->number (interp e2))))]
    [(eand e1 e2)
     (vbool (and (value->bool (interp e1)) (value->bool (interp e2))))]))
```

4 Designing a Type-system for the Calculator Language, Try 1

- To design a type-system, we must first define what it means for our calculator programs to "go wrong".
- We will define "going wrong" as *raising a runtime error in the Racket host language*. For example, the following program goes wrong:

```
> (interp (eadd (enum 1) (ebool #t)))
not a number
```

- So, our design goal is to write a program that rejects all calculator programs that raise Racket runtime errors. Here is a program that does that:

```
;;; type-check-1 : expr -> bool
;;; returns #t if a program is well-typed, #f otherwise
(define (type-check-1 e)
  #f)
```

- Is this a good type-checker? Clearly `type-check-1` is *sound*: all programs are rejected, so it trivially satisfies the requirement that all bad programs are rejected. It is also efficient, since it runs instantly. But, it's not expressive: it rejects too many programs, including many that are perfectly valid calculator programs. So, it is *not* a very good type checker, because it fails one of our important evaluation criteria.

5 A Second Attempt at a Type-system for the Calculator Language

- Let's try again: we need a more expressive type-checker that accepts more valid calculator programs. We want a typechecker that (1) rejects all calculator programs that raise Racket runtime errors when they are evaluated, and (2) accepts the rest.
- We start by asking ourselves: what kind of bad programs do we want to eliminate during type-checking? In short, any programs that try to do the following:
  - Add together non-numbers
  - And together non-Booleans.
- So, how do we know when we are trying to add a non-number or and a non-Boolean? We need a way of knowing *the type of value a program will evaluate to*.
- First, *what are the types*? Well, recall that types are collections of values, and we have two types of values in our calculator language, `vbool` and `vnum`. So, we can define two kinds of types, `tbool` and `tnum`:

```
;;; type calctype =
;;;    | tbool
;;;    | tnum
(struct tbool () #:transparent)
(struct tnum () #:transparent)
```

- Note the difference between the value struct `vbool` and the type struct `tbool`: the type struct does not keep track of the value.
- Now we can write a `type-of?` function that attempts to compute the type of a `calc` program, and raises a `type-error` if it is not possible to compute a valid type:
- We will take a small excursion into Racket's exception-handling and defining facilities. We want to define a new kind of error, a type error, for your `type-of?` function to raise in case it encounters an ill-typed term. We define this in Racket as:

```
;;; declare a type exception that is a subtype of exn:fail
(struct exn:type exn:fail ())

;;; raise a type error exception when called
(define (raise-type-error)
  (raise (exn:type
          "type error"
          (current-continuation-marks))))
```

- Now we can define a function `type-check?` that calls our `type-check?` function and returns `#t` if it successfully completes and `#f` if it raises a type exception (it's not super important right now that you understand this code):

```
;;; type-check? : expr -> bool
;;; returns #t if e is well-typed, #f otherwise
(define (type-check? e)
  (with-handlers ([exn:type? (lambda _ #f)])
    (let [(_ (type-of? e))]
      #t)))
```

- Now, on the next page, we can define our `type-of?` function that makes use of exceptions to terminate early in the event that an ill-typed expression is found.

```
;;; type-of? : expr -> calctype
;;; computes the type of an expression or raises an error if there is no valid
;;; type.
(define (type-of? e)
  (match e
    [(ebool _) (tbool)]
    [(enum _) (tnum)]
    [(eadd e1 e2)
     ; check if the type of e1 and e2 are both tnum.
     ; if they are, return tnum. otherwise, raise a type error.
     (match (list (type-of? e1) (type-of? e2))
       [(list (tnum) (tnum))
        (tnum)]
       [_ (raise-type-error)])]
    [(eand e1 e2)
     ; check if the type of e1 and e2 are both tbool.
     ; if they are, return tbool. otherwise, raise a type error.
     (match (list (type-of? e1) (type-of? e2))
       [(list (tbool) (tbool))
        (tbool)]
       [_ (raise-type-error)])]
    [(elessthan e1 e2)
     ; check if the type of e1 and e2 are both tnum.
     ; if they are, return tbool. otherwise, raise a type error.
     (match (list (type-of? e1) (type-of? e2))
       [(list (tnum) (tnum))
        (tbool)]
       [_ (raise-type-error)])]))
```

Listing 1: An implementation of the type-of? function

6 Testing Our Type-checker

- First, we check soundness: are there any calculator programs that cause runtime errors in Racket that our type-checker accepts as well-typed? We can check for this by checking that our type-checker rejects such programs:

  ```
  ;;; soundness checks
  (check-equal? (type-check? (eadd (enum 1) (ebool #t))) #f)
  (check-equal? (type-check? (eand (enum 1) (ebool #t))) #f)
  (check-equal? (type-check? (eand (eadd (enum 10) (enum 20)) (ebool #t))) #f)
  (check-equal? (type-check? (elessthan (enum 1) (ebool #t))) #f)
  ```

- Next, let's test for expressivity by checking that well-behaved programs do indeed typecheck:

  ```
  ;;; expressivity checks
  (check-equal? (type-check? (eadd (enum 1) (enum 10))) #t)
  (check-equal? (type-check? (eand (ebool #t) (ebool #f))) #t)
  (check-equal? (type-check? (elessthan (enum 10) (enum 20))) #t)
  (check-equal? (type-check? (eand (elessthan (enum 10) (enum 20)) (ebool #t))) #t)
  ```

7 Types and Inference Rules

- Our `type-of?` function looks a lot like an interpreter. This means we can use our inference rule machinery to describe how it works, just like we did for semantics.
- Here are the inference rules, written in full Racket syntax. Similar to the case for interpreters, if there is no valid inference rule to apply, then the term is assumed to be not well-typed:

$$\frac{}{\texttt{(type-of? (enum n)) = (tnum)}}\ \text{(T-\textsc{Num})}$$

$$\frac{}{\texttt{(type-of? (ebool b)) = (tbool)}}\ \text{(T-\textsc{Bool})}$$

$$\frac{\texttt{(type-of? e1) = (tint)} \quad \texttt{(type-of? e2) = (tint)}}{\texttt{(type-of? (eand e1 e2)) = (tint)}}\ \text{(T-\textsc{And})}$$

$$\frac{\texttt{(type-of? e1) = (tint)} \quad \texttt{(type-of? e2) = (tint)}}{\texttt{(type-of? (elessthan e1 e2)) = (tbool)}}\ \text{(T-\textsc{LessThan})}$$

$$\frac{\texttt{(type-of? e1) = (tbool)} \quad \texttt{(type-of? e2) = (tbool)}}{\texttt{(type-of? (eand e1 e2)) = (tbool)}}\ \text{(T-\textsc{And})}$$

- Once again, just like for semantics, we define some short-hand to make it easier to draw and visualize derivations involving types. We will say `e:t` to denote the fact that term `(type-of?  e)` equals `t`. We will also drop the s-expressions to make it easier to read:

$$\frac{}{\texttt{(enum n):tnum}}\ \text{(T-\textsc{Num})} \qquad \frac{}{\texttt{(ebool b):tbool}}\ \text{(T-\textsc{Bool})}$$

$$\frac{\texttt{e1:tbool} \quad \texttt{e2:tbool}}{\texttt{(eand e1 e2):tbool}}\ \text{(T-\textsc{Bool})} \qquad \frac{\texttt{e1:tnum} \quad \texttt{e2:tnum}}{\texttt{(elessthan e1 e2):tbool}}\ \text{(T-\textsc{LessThan})}$$

$$\frac{\texttt{e1:tbool} \quad \texttt{e2:tbool}}{\texttt{(eand e1 e2):tbool}}\ \text{(T-\textsc{And})}$$

8 Typing Derivations

- Now we can draw some derivation trees for computing the types of terms, just like we did for describing running programs using big-step semantics.
- For example, here is a derivation tree that shows that the term `(eadd (enum 10) (enum 20))` has type `(T-Add)`:

$$\dfrac{\dfrac{}{\texttt{(enum 10):tnum}}\;(\textsc{T-Num}) \qquad \dfrac{}{\texttt{(enum 20):tnum}}\;(\textsc{T-Num})}{\texttt{(eadd (enum 10) (enum 20)) : tnum}}\;(\textsc{T-Add})$$

- Hopefully you can see how similar drawing these trees is to running your interpreter. It might even seem that they're basically identical! But, we will see that as our type-systems and languages get more intricate, the behavior of the type checker and interpreter will begin to diverge.

References

[1] Robin Milner. A theory of type polymorphism in programming. *Journal of computer and system sciences*, 17(3):348–375, 1978.