CS 476 – Programming Language Design

William Mansky

Questions

Nobody has responded yet.

Hang tight! Responses are coming in.

Structure of a language

- Syntax
  - Concrete: what do programs look like?
  - Abstract: what are the pieces of a program?
- Semantics
  - Static: which programs make sense? what can we expect from them?
  - Dynamic: what do programs do when we run them?
- Pragmatics
  - Implementation: how can we actually make the semantics happen?
  - IDE, tool support, etc.

Static Semantics: Types

- Exercise: what are types for? Why is it useful to have a type system in a programming language?

Static Semantics: Types

Type systems do several things:

- Help programmers tell different kinds of program expressions apart
- Stop bad programs from running/catch errors early
- Tell us what kind of value to expect from a piece of code
- Give useful information to the compiler/interpreter

Static Semantics: Types

- A type system is a relation between terms and types
- We write $t : \tau$ for "term t has type τ" ← "tau"
- We define type systems using *inference rules*:

$$\frac{P_1 \ \ldots \ P_n}{t : \tau} \qquad \frac{e_1 : \text{int} \quad e_2 : \text{int}}{e_1 + e_2 : \text{int}}$$

- Often *sound*: $t : \tau$ implies that t evaluates to a value of type τ
- Often *conservative*: not all programs that evaluate are well-typed

Static Semantics: Inference Rules

$$\frac{e_1 : \mathbf{int} \quad e_2 : \mathbf{int}}{e_1 \mathbf{+} e_2 : \mathbf{int}}$$

- Above the line: *premises* that must be true for the rule to apply
- Below the line: *conclusion* that we learn when the rule applies
- Contain both fixed symbols ($\mathbf{+}, \mathbf{int}$) and *metavariables* (e_1, e_2)
  - Rule applies for any way of filling in metavariables (e.g., for any expressions e_1, e_2)

Expressions: Types

- Types: int, bool
- Rules:

$$\frac{(n \text{ is a number literal})}{n : \text{int}} \qquad \frac{e_1 : \text{int} \quad e_2 : \text{int}}{e_1 \textbf{ + } e_2 : \text{int}}$$

$$\frac{(b \text{ is a boolean literal})}{b : \text{bool}} \qquad \frac{e_1 : \text{bool} \quad e_2 : \text{bool}}{e_1 \textbf{ and } e_2 : \text{bool}}$$

$$\frac{e_1 : \tau \quad e_2 : \tau}{e_1 \textbf{ = } e_2 : \text{bool}}$$

Questions

Nobody has responded yet.

Hang tight! Responses are coming in.

Expressions: Types

- Types: int, bool
- Rules:

$$\frac{(n \text{ is a number literal})}{n : \text{int}} \qquad \frac{e_1 : \text{int} \quad e_2 : \text{int}}{e_1 \textbf{ + } e_2 : \text{int}}$$

$$\frac{(b \text{ is a boolean literal})}{b : \text{bool}} \qquad \frac{e_1 : \text{bool} \quad e_2 : \text{bool}}{e_1 \textbf{ and } e_2 : \text{bool}}$$

$$\frac{e_1 : \tau_1 \quad e_2 : \tau_2}{e_1 \textbf{ = } e_2 : \text{bool}}$$

Expressions: Types

- Types: int, bool
- Rules:

$$\frac{(n \text{ is a number literal})}{n : \text{int}} \qquad \frac{e_1 : \text{int} \quad e_2 : \text{int}}{e_1 \texttt{ + } e_2 : \text{int}}$$

$$\frac{(b \text{ is a boolean literal})}{b : \text{bool}} \qquad \frac{e_1 : \text{bool} \quad e_2 : \text{bool}}{e_1 \texttt{ and } e_2 : \text{bool}}$$

$$\frac{e_1 : \tau \quad e_2 : \tau}{e_1 \texttt{ = } e_2 : \text{bool}} \qquad \frac{e : \text{bool} \quad e_1 : \tau \quad e_2 : \tau}{\texttt{if } e \texttt{ then } e_1 \texttt{ else } e_2 : \tau}$$

Proof Trees

$$\frac{e : \text{bool} \quad e_1 : \tau \quad e_2 : \tau}{\texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \tau}$$

$$\texttt{if 3 = 5 then 1 else 2} : \text{int}$$

Proof Trees

$$\frac{e : \text{bool} \quad e_1 : \tau \quad e_2 : \tau}{\texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \tau}$$

$$\frac{\texttt{3 = 5} : \text{bool} \quad \texttt{1} : \text{int} \quad \texttt{2} : \text{int}}{\texttt{if 3 = 5 then 1 else 2} : \text{int}}$$

Proof Trees

$$\frac{e_1 : \tau \quad e_2 : \tau}{e_1 = e_2 : \text{bool}}$$

$$\frac{3 = 5 : \text{bool} \quad 1 : \text{int} \quad 2 : \text{int}}{\texttt{if 3 = 5 then 1 else 2} : \text{int}}$$

Proof Trees

$$\frac{e_1 : \tau \quad e_2 : \tau}{e_1 = e_2 : \text{bool}}$$

$$\frac{\dfrac{3 : \text{int} \quad 5 : \text{int}}{3 = 5 : \text{bool}} \qquad 1 : \text{int} \quad 2 : \text{int}}{\texttt{if 3 = 5 then 1 else 2} : \text{int}}$$

Proof Trees

$$\frac{(n \text{ is a number literal})}{n : \text{int}}$$

$$\frac{\dfrac{\texttt{3} : \text{int} \quad \texttt{5} : \text{int}}{\texttt{3 = 5} : \text{bool}} \quad \texttt{1} : \text{int} \quad \texttt{2} : \text{int}}{\texttt{if 3 = 5 then 1 else 2} : \text{int}}$$

Proof Trees

$$\frac{(n \text{ is a number literal})}{n : \text{int}}$$

$$\dfrac{\dfrac{\dfrac{(3 \text{ is a number literal})}{3 : \text{int}} \quad 5 : \text{int}}{3 = 5 : \text{bool}} \quad 1 : \text{int} \quad 2 : \text{int}}{\texttt{if 3 = 5 then 1 else 2} : \text{int}}$$

Proof Trees

$$\frac{(n \text{ is a number literal})}{n : \text{int}}$$

$$\frac{\dfrac{\dfrac{}{\texttt{3} : \text{int}} \quad \dfrac{}{\texttt{5} : \text{int}}}{\texttt{3 = 5} : \text{bool}} \quad \dfrac{}{\texttt{1} : \text{int}} \quad \dfrac{}{\texttt{2} : \text{int}}}{\texttt{if 3 = 5 then 1 else 2} : \text{int}}$$

Proof Trees

$$\frac{e : \text{bool} \quad e_1 : \tau \quad e_2 : \tau}{\texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \tau}$$

`if 3 = 5 then 1 else false` : int

Proof Trees

$$\frac{e : \text{bool} \quad e_1 : \tau \quad e_2 : \tau}{\texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \tau}$$

$$\frac{\texttt{3 = 5} : \text{bool} \quad \texttt{1} : \text{int} \quad \texttt{false} : \text{int}}{\texttt{if 3 = 5 then 1 else false} : \text{int}}$$

Questions

Nobody has responded yet.

Hang tight! Responses are coming in.

Expressions: Types

- Types: int, bool
- Rules:

$$\frac{(n \text{ is a number literal})}{n : \text{int}} \qquad \frac{e_1 : \text{int} \quad e_2 : \text{int}}{e_1 \texttt{ + } e_2 : \text{int}}$$

$$\frac{(b \text{ is a boolean literal})}{b : \text{bool}} \qquad \frac{e_1 : \text{bool} \quad e_2 : \text{bool}}{e_1 \texttt{ and } e_2 : \text{bool}}$$

$$\frac{e_1 : \tau \quad e_2 : \tau}{e_1 \texttt{ = } e_2 : \text{bool}} \qquad \frac{e : \text{bool} \quad e_1 : \tau \quad e_2 : \tau}{\texttt{if } e \texttt{ then } e_1 \texttt{ else } e_2 : \tau}$$

Expressions: Type Checking

- Every term computes to a value, either int or bool
- Types: int, bool

type etype = IntTy | BoolTy

  - These are *object-level* (expression) types, not *meta-level* (OCaml) types `int`, `bool`

Literals, Values, Types

- We have three different kinds of int/bool constructors now:

type exp = Num of int | ... | Bool of bool | ... | If of exp * exp * exp
type retval = IntVal of int | BoolVal of bool
type etype = IntTy | BoolTy

- An *expression* is a piece of program code
- A *retval* is the result when we run a program
- An etype is information about an expression
  - If we get it right, the type of an expression tells us which kind of value it will produce!

Expressions: Type Checking

- Every term computes to a *value*, either int or bool
- Types: int, bool

type etype = IntTy | BoolTy

  - These are "object-level" (expression) types, not "meta-level" (OCaml) types (`int`, `bool`)

- A *type checker* takes an expression and a type, and determines whether the expression has that type

Expressions: Type Checking

type etype = IntTy | BoolTy

— These are “object-level” (expression) types, not “meta-level” (OCaml) types!

```
let rec typecheck (e : exp) (t : etype) : bool =   (* true when e : t *)
        match e with
        | Num i ->
        | Bool b ->
        | Add (e1, e2) ->
        …
```

$$\frac{(n \text{ is a number literal})}{n : \text{int}}$$

Expressions: Type Checking

type etype = IntTy | BoolTy

— These are “object-level” (expression) types, not “meta-level” (OCaml) types!

```
let rec typecheck (e : exp) (t : etype) : bool =   (* true when e : t *)
        match e with
        | Num i -> t = IntTy
        | Bool b ->
        | Add (e1, e2) ->
        …
```

$$\frac{(n \text{ is a number literal})}{n : \text{int}}$$

$$\frac{}{\text{Int}\, n : \text{int}}$$

Expressions: Type Checking

type etype = IntTy | BoolTy

— These are “object-level” (expression) types, not “meta-level” (OCaml) types!

```
let rec typecheck (e : exp) (t : etype) : bool =   (* true when e : t *)
        match e with
        | Num _ -> t = IntTy
        | Bool b ->
        | Add (e1, e2) ->
        …
```

$$\frac{(n \text{ is a number literal})}{n : \text{int}}$$

$$\frac{}{\text{Num } n : \text{int}}$$

Expressions: Type Checking

type etype = IntTy | BoolTy

— These are “object-level” (expression) types, not “meta-level” (OCaml) types!

```
let rec typecheck (e : exp) (t : etype) : bool =   (* true when e : t *)
        match e with
        | Num _ -> t = IntTy
        | Bool _ -> t = BoolTy
        | Add (e1, e2) ->
        …
```

$$\frac{(b \text{ is a boolean literal})}{b : \text{bool}}$$

Expressions: Type Checking

type etype = IntTy | BoolTy

— These are “object-level” (expression) types, not “meta-level” (OCaml) types!

```
let rec typecheck (e : exp) (t : etype) : bool =   (* true when e : t *)
        match e with
        | Num _ -> t = IntTy
        | Bool _ -> t = BoolTy
        | Add (e1, e2) ->
```

$$\frac{e_1 : \text{int} \quad e_2 : \text{int}}{e_1 + e_2 : \text{int}}$$

~~Exercise: Write OCaml code that implements the type rule for addition.~~

Expressions: Type Checking

type etype = IntTy | BoolTy

— These are “object-level” (expression) types, not “meta-level” (OCaml) types!

```
let rec typecheck (e : exp) (t : etype) : bool =   (* true when e : t *)
        match e with
        | Num _ -> t = IntTy
        | Bool _ -> t = BoolTy
        | Add (e1, e2) ->
        …
```

$$\frac{e_1 : \text{int} \quad e_2 : \text{int}}{e_1 + e_2 : \text{int}}$$

Expressions: Type Checking

type etype = IntTy | BoolTy

— These are "object-level" (expression) types, not "meta-level" (OCaml) types!

```
let rec typecheck (e : exp) (t : etype) : bool =   (* true when e : t *)
        match e with
        | Num _ -> t = IntTy
        | Bool _ -> t = BoolTy
        | Add (e1, e2) -> typecheck e1 IntTy && typecheck e2 IntTy
        …
```

$$\frac{e_1 : \text{int} \quad e_2 : \text{int}}{e_1 + e_2 : \text{int}}$$

Expressions: Type Checking

type etype = IntTy | BoolTy

— These are “object-level” (expression) types, not “meta-level” (OCaml) types!

```
let rec typecheck (e : exp) (t : etype) : bool =   (* true when e : t *)
    match e with
    | Num _ -> t = IntTy
    | Bool _ -> t = BoolTy
    | Add (e1, e2) -> typecheck e1 IntTy && typecheck e2 IntTy
    …
```

$$\frac{e_1 : \text{int} \quad e_2 : \text{int}}{e_1 + e_2 : \text{int}}$$

Expressions: Type Checking

type etype = IntTy | BoolTy

— These are “object-level” (expression) types, not “meta-level” (OCaml) types!

```
let rec typecheck (e : exp) (t : etype) : bool =   (* true when e : t *)
        match e with
        | Num _ -> t = IntTy
        | Bool _ -> t = BoolTy
        | Add (e1, e2) -> typecheck e1 IntTy && typecheck e2 IntTy
                          && t = IntTy
```

$$\frac{e_1 : \text{int} \quad e_2 : \text{int}}{e_1 + e_2 : \text{int}}$$

Questions

Nobody has responded yet.

Hang tight! Responses are coming in.