Inference Rules

Concepts of Programming Languages
Lecture 4

CAS CS 320

Outline

- » Discuss Formal Typing/Semantic Rules
- » Look at example rules for the constructs we've seen so far
- » Learn to read inference rules, i.e., translate mathematical notation to English and English to mathematical notation

Recall: Integer Addition (Informal)

$\boxed{13 + x}$

Recall: Integer Addition (Informal)

$\boxed{\texttt{13 + x}}$

Syntax: $\texttt{EXPRESSION}_1$ + $\texttt{EXPRESSION}_2$

Recall: Integer Addition (Informal)

`13 + x`

Syntax: EXPRESSION_1 + EXPRESSION_2

Typing: EXPRESSION_1 *must be an Integer;* EXPRESSION_2 *must be an Integer; then* EXPRESSION_1 + EXPRESSION_2 *will be an integer*

Recall: Integer Addition (Informal)

```
13 + x
```

Syntax: $\texttt{EXPRESSION}_1$ + $\texttt{EXPRESSION}_2$

Typing: $\texttt{EXPRESSION}_1$ *must be an Integer;* $\texttt{EXPRESSION}_2$ *must be an Integer; then* $\texttt{EXPRESSION}_1$ + $\texttt{EXPRESSION}_2$ *will be an integer*

Semantics: *Evaluate* $\texttt{EXPRESSION}_1$*; say it has value* v_1*; Evaluate* $\texttt{EXPRESSION}_2$*; say it has value* v_2*; add* v_1 *and* v_2

Let’s start with the formal syntax

Note: Production Rules and Syntax

<expr> ::= <expr> + <expr>

Note: Production Rules and Syntax

<expr> ::= <expr> + <expr>

This is called a ***production rule*** and is part of a ***BNF grammar***

Note: Production Rules and Syntax

<expr> ::= <expr> + <expr>

This is called a ***production rule*** and is part of a ***BNF grammar***

This reads as: if e_1 is a well-formed expression and e_2 is a well-formed expression, then $e_1 + e_2$ is a well-formed expression

Note: Production Rules and Syntax

<expr> ::= <expr> + <expr>

This is called a ***production rule*** and is part of a ***BNF grammar***

This reads as: if e_1 is a well-formed expression and e_2 is a well-formed expression, then $e_1 + e_2$ is a well-formed expression

We won't focus on this until the second half of the course but you should start to get comfortable with the syntax

Inference Rules

Inference Rules

$$\frac{\overset{\texttt{premise}}{P_1} \qquad \overset{\texttt{premise}}{P_2} \qquad \cdots \qquad \overset{\texttt{premise}}{P_k}}{\underset{\texttt{conclusion}}{C}} \;\; \overset{\texttt{name}}{\texttt{my-rule}}$$

Inference Rules

$$\dfrac{\overset{\text{premise}}{P_1} \qquad \overset{\text{premise}}{P_2} \qquad \cdots \qquad \overset{\text{premise}}{P_k}}{\underset{\text{conclusion}}{C}}\ \overset{\text{name}}{\texttt{my-rule}}$$

Then general form of an inference rule has a collection of **premises** and a **conclusion**

Inference Rules

$$\dfrac{\overset{\text{premise}}{P_1} \qquad \overset{\text{premise}}{P_2} \qquad \cdots \qquad \overset{\text{premise}}{P_k}}{\underset{\text{conclusion}}{C}}\ \overset{\text{name}}{\texttt{my-rule}}$$

Then general form of an inference rule has a collection of **premises** and a **conclusion**

There may be no premises, this is called an **axiom**

Inference Rules

$$\frac{\overset{\texttt{premise}}{P_1} \qquad \overset{\texttt{premise}}{P_2} \qquad \ldots \qquad \overset{\texttt{premise}}{P_k}}{\underset{\texttt{conclusion}}{C}}\ \overset{\texttt{name}}{\texttt{my-rule}}$$

We can read this as:

*If P_1 holds and P_2 holds and … P_k holds, then C holds (by **my-rule**)*

Typing Judgment and Rules

Typing Judgment and Rule

$$\underbrace{e}_{\text{expression}} : \underbrace{\tau}_{\text{type}}$$

A **typing judgment** is a formal way of representing the statement:

e has type τ

*A **typing rule** is an inference rule whose premises and conclusion are typing judgments*

Typing Rule for Addition

We will use inference rules both to express typing rules and semantics rules

$$\frac{e_1 : \mathsf{int} \qquad e_2 : \mathsf{int}}{e_1 + e_2 : \mathsf{int}} \; (\mathrm{addInt})$$

Say it in English!

*If (e_1 has type int) holds and (e_2 has type int) holds, then ($e_1 + e_2$ has type int) holds (by **addInt**)*

But this breaks down as soon as we encounter variables

Let Expressions (Informal)

body

```
let x = 2 in x + x
```

Let Expressions (Informal)

body

```
let x = 2 in x + x
```

syntax: `let VARIABLE = EXPRESSION in BODY`

Let Expressions (Informal)

body

```
let x = 2 in x + x
```

syntax: `let VARIABLE = EXPRESSION in BODY`

typing: *compute type of* EXPRESSION*; assume* VARIABLE *has that type; compute type of* BODY

Let Expressions (Informal)

body

```
let x = 2 in x + x
```

syntax: `let VARIABLE = EXPRESSION in BODY`

typing: *compute type of* `EXPRESSION`*; assume* `VARIABLE` *has that type; compute type of* `BODY`

semantics: *compute value of* `EXPRESSION`*; substitute that value for* `VARIABLE` *in* `BODY`

Let-Expressions (Syntax Rule)

<expr> ::= let <var> = <expr> in <expr>

If x is a valid variable name, and e_1 is a well-formed expression and e_2 is a well-formed expression then

$$\texttt{let}\ x = e_1\ \texttt{in}\ e_2$$

is a well-formed expression

Let's Try to Write the Typing Rule

We will use inference rules both to express typing rules and semantics rules

$$\frac{e_1 : \tau_1 \qquad e_2 : \tau_2}{\text{let } x = e_1 \text{ in } e_2 : \tau_2}\ (\text{let})$$

Say it in English!

If (e_1 has type τ_1) holds and (e_2 has type τ_2) holds, then ($\text{let } x = e_1 \text{ in } e_2$ has type τ_2) holds (by ***let****)*

Let's Try to Write the Typing Rule

We will use inference rules both to express typing rules and semantics rules

assuming x has type τ_1?

$$\frac{e_1 : \tau_1 \qquad \boxed{e_2 : \tau_2}}{\text{let } x = e_1 \text{ in } e_2 : \tau_2} \text{ (let)}$$

Say it in English!

*If (e_1 has type τ_1) holds and (e_2 has type τ_2) holds, then (let $x = e_1$ in e_2 has type τ_2) holds (by **let**)*

Example

```
let x = 2 in x
```

$$\frac{2 : \mathsf{int} \qquad x : \tau_2}{\mathsf{let}\ x = 2\ \mathsf{in}\ x : \tau_2}\ (\mathsf{let})$$

Example

```
let x = 2 in x
```

What is τ_2?

$$\frac{2 : \mathsf{int} \qquad x : \tau_2}{\mathsf{let}\ x = 2\ \mathsf{in}\ x : \tau_2}\ (\mathsf{let})$$

Example

```
let x = 2 in x
```

What is τ_2?

$$\frac{2 : \mathsf{int} \qquad x : \tau_2}{\mathsf{let}\ x = 2\ \mathsf{in}\ x : \tau_2}\ (\mathsf{let})$$

We need to carry over the information that x has type int to the second premise. How do we do that?

Contexts

$$\Gamma = \{\ \text{x : int, y : string, z : int -> string}\ \}$$

Contexts

$$\Gamma = \{\ \text{x : int, y : string, z : int -> string}\ \}$$

A **context** is a set of **variable declarations**

Contexts

$$\Gamma = \{\, \text{x} : \text{int},\ \text{y} : \text{string},\ \text{z} : \text{int -> string} \,\}$$

A **context** is a set of **variable declarations**

A variable declaration $(x : \tau)$ says: "I declare that the variable x is of type τ"

Contexts

$$\Gamma = \{\, \text{x} : \text{int},\ \text{y} : \text{string},\ \text{z} : \text{int -> string} \,\}$$

A **context** is a set of **variable declarations**

A variable declaration $(x : \tau)$ says: "I declare that the variable x is of type τ"

A context keeps track of all the types of variables in the "(static) environment” (or scope)

Typing Judgments

$$\underbrace{\Gamma}_{\text{context}} \vdash \underbrace{e}_{\text{expression}} : \underbrace{\tau}_{\text{type}}$$

A typing judgment a compact way of representing the statement:

e is of type τ in the context Γ

A **typing rule** is an inference rule whose premises and conclusion are typing judgments

Example: Using Typing Judgements

$$\{x : \mathsf{int}\} \vdash x : \mathsf{int}$$

Example: Using Typing Judgements

$$\{x : \mathsf{int}\} \vdash x : \mathsf{int}$$

In English: *Given I declare that x is an int, the expression x is an int*

Example: Using Typing Judgements

$$\{x : \mathsf{int}\} \vdash x : \mathsf{int}$$

In English: *Given I declare that x is an int, the expression x is an int*

The context allows us to determine the type of an expression *relative to the types of variables*

Example: Using Typing Judgements

$$\{\mathsf{b} : \mathsf{bool}\} \vdash \mathsf{if}\ \ \mathsf{b}\ \ \mathsf{then}\ \ 2\ \ \mathsf{else}\ \ 3 : \mathsf{int}$$

Example: Using Typing Judgements

$$\{b : \text{bool}\} \vdash \text{if } \ b \ \text{ then } \ 2 \ \text{ else } \ 3 : \text{int}$$

In English: *Given I declare that b has type bool, the expression if b then 2 else 3 has type int*

Example: Using Typing Judgements

$$\{\mathsf{b} : \mathsf{bool}\} \vdash \mathsf{if}\ \ \mathsf{b}\ \ \mathsf{then}\ \ 2\ \ \mathsf{else}\ \ 3 : \mathsf{int}$$

In English: *Given I declare that b has type bool, the expression if b then 2 else 3 has type int*

The context allows us to determine the type of an expression *relative to the types of variables*

Integer Addition Typing Rule

$$\frac{\Gamma \vdash e_1 : \mathsf{int} \qquad \Gamma \vdash e_2 : \mathsf{int}}{\Gamma \vdash e_1 + e_2 : \mathsf{int}} \text{ (addInt)}$$

If (e_1 has type int *in any context* Γ*) holds and (e_2 has type* int *in the same context* Γ *holds, then ($e_1 + e_2$ has type* int*) holds (by* ***addInt****)*

Let-Expressions (Typing Rule)

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma, x : \tau_1 \vdash e_2 : \tau}{\Gamma \vdash \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 : \tau}\ \text{(let)}$$

If e_1 is of type τ_1 in the context Γ, and e_2 is of type τ in the context Γ *with the variable declaration* $(x : \tau_1)$ *added to it*, then

let x = e_1 in e_2

is of type τ in the context Γ

Let-Expressions (Typing Rule)

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma, x : \tau_1 \vdash e_2 : \tau}{\Gamma \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2 : \tau}\ (\text{let})$$

Note: Look at how much more compact the rule is!

If e_1 is of type τ_1 in the context Γ, and e_2 is of type τ in the context Γ *with the variable declaration* $(x : \tau_1)$ *added to it*, then

`let` x `=` e_1 `in` e_2

is of type τ in the context Γ

If-Expressions (Informal)

```
if x > 0 then x else -x
```

If-Expressions (Informal)

```
if x > 0 then x else -x
```

Syntax: `if CONDITION then TRUE-CASE else FALSE-CASE`

If-Expressions (Informal)

```
if x > 0 then x else -x
```

Syntax: if CONDITION then TRUE–CASE else FALSE–CASE

Typing: CONDITION *must be a Boolean; compute the types of* TRUE–CASE *and* FALSE–CASE*; must be the same type; expression type is same as that of* TRUE–CASE *and* FALSE–CASE

If-Expressions (Informal)

```
if x > 0 then x else -x
```

Syntax: if CONDITION then TRUE–CASE else FALSE–CASE

Typing: CONDITION *must be a Boolean; compute the types of* TRUE–CASE *and* FALSE–CASE*; must be the same type; expression type is same as that of* TRUE–CASE *and* FALSE–CASE

Semantics: *If* CONDITION *evaluates to true; evaluate* TRUE–CASE*, else evaluate* FALSE–CASE

If-Expressions (Informal)

```
if x > 0 then x else -x
```

Syntax: if CONDITION then TRUE–CASE else FALSE–CASE

Typing: CONDITION *must be a Boolean; compute the types of* TRUE–CASE *and* FALSE–CASE*; must be the same type; expression type is same as that of* TRUE–CASE *and* FALSE–CASE

Semantics: *If* CONDITION *evaluates to true; evaluate* TRUE–CASE*, else evaluate* FALSE–CASE

Formal Syntax: **<expr> ::= if <expr> then <expr> else <expr>**

If-Expressions (Syntax Rule)

<expr> ::= if <expr> then <expr> else <expr>

If e_1 is a well-formed expression and e_2 is a well-formed expression and e_3 is a well-formed expression, then

if e_1 then e_2 else e_3

is a well-formed expression

If-Expressions (Typing Rule)

$$\frac{\Gamma \vdash e_1 : \textsf{bool} \qquad \Gamma \vdash e_2 : \tau \qquad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \textsf{if}\ \ e_1\ \ \textsf{then}\ \ e_2\ \ \textsf{else}\ \ e_3 : \tau}\,(\text{if})$$

If e_1 is of type bool in the context Γ and e_2 and e_3 are of type τ in the context Γ, then

$$\textsf{if}\ \ e_1\ \ \textsf{then}\ \ e_2\ \ \textsf{else}\ \ e_3$$

is of type τ in the context Γ

Note: Judgements are claims

$$\{b : \mathrm{bool}\} \vdash \mathrm{if}\ \ b\ \ \mathrm{then}\ \ 2\ \ \mathrm{else}\ \ 3 : \boxed{\mathrm{string}}$$

Note: Judgements are claims

$$\{\mathsf{b} : \mathsf{bool}\} \vdash \mathsf{if\ \ b\ \ then\ \ 2\ \ else\ \ 3} : \boxed{\mathsf{string}}$$

`A judgement is a` *`claim`* `in the same way that "there are infinitely many twin primes" or "pigs fly" is a claim`

Note: Judgements are claims

$$\{b : \mathsf{bool}\} \vdash \mathsf{if}\ \ b\ \ \mathsf{then}\ \ 2\ \ \mathsf{else}\ \ 3 : \mathsf{string}$$

A judgement is a *claim* in the same way that "there are infinitely many twin primes" or "pigs fly" is a claim

We haven't **proved** anything by writing down a typing judgment

Note: Judgements are claims

$$\{b : \text{bool}\} \vdash \text{if} \;\; b \;\; \text{then} \;\; 2 \;\; \text{else} \;\; 3 : \text{string}$$

A judgement is a *claim* in the same way that "there are infinitely many twin primes" or "pigs fly" is a claim

We haven't **proved** anything by writing down a typing judgment

Next week: We will talk about **typing derivations,** which are used to demonstrate that expressions *actually* have their desired types in our PL

Semantic Judgment and Rules

Semantic Judgements

$$\underbrace{e}_{\text{expression}} \Downarrow \underbrace{v}_{\text{value}}$$

A <u>semantic judgment</u> is a compact way of representing the statement:

The expression e evaluates to the value v

A **semantic rule** is an inference rule with semantic judgments

Example: Reading Semantic Judgments

$$\boxed{\textsf{if } 2 > 3 \textsf{ then } 2 + 2 \textsf{ else } 3 \Downarrow 3}$$

In English: The expression

`if 2 > 3 then 2 + 2 else 3`

evaluates to the value 3

Recall: Integer Addition Semantic Rule

$$\frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2}{e_1 \mathbin{\color{red}{+}} e_2 \Downarrow v_1 + v_2} \text{(evalInt)}$$

If e_1 evaluates to the (integer) v_1 and e_2 evaluates to the (integer) v_2, then $e_1 \mathbin{\color{red}{+}} e_2$ evaluates to the (integer) $v_1 + v_2$

Let-Expressions (Semantic Rule)

$$\frac{e_1 \Downarrow v_1 \qquad [v_1/x]e_2 \Downarrow v}{\texttt{let}\ x\ =\ e_1\ \texttt{in}\ e_2 \Downarrow v}\ \text{(letEval)}$$

If e_1 evaluates to v_1 and e_2 *with* v_2 *substituted for* x evaluates to v, then

$$\texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2$$

evaluates to v

Note: Values are not Expressions

$$\textcolor{red}{\textsf{if } 2 > 3 \textsf{ then } 2 + 2 \textsf{ else } 3} \Downarrow 3$$

We will draw a distinction between values and expressions (note the font and color difference)

Example. We'll use regular numbers to represented integer values, and we'll use $\top$ and $\bot$ for the true and false Boolean values

If-Expressions (Semantic Rule 1)

$$\frac{e_1 \Downarrow \top \qquad e_2 \Downarrow v_2}{\textsf{if } e_1 \textsf{ then } e_2 \textsf{ else } e_3 \Downarrow v_2}\ (\textsf{ifEvalTrue})$$

If e_1 evaluates to $\top$ and e_2 evaluates to v_2, then

$$\textsf{if } e_1 \textsf{ then } e_2 \textsf{ else } e_3$$

evaluates to v_2

If-Expressions (Semantic Rule 2)

$$\frac{e_1 \Downarrow \bot \qquad e_3 \Downarrow v_3}{\textsf{if}\ \ e_1\ \ \textsf{then}\ \ e_2\ \ \textsf{else}\ \ e_3 \Downarrow v_3}\ (\text{ifEvalFalse})$$

If e_1 evaluates to $\bot$ and e_2 evaluates to v_2, then

$$\textsf{if}\ e_1\ \textsf{then}\ e_2\ \textsf{else}\ e_3$$

evaluates to v_3

If-Expressions (Semantic Rule 2)

$$\frac{e_1 \Downarrow \bot \qquad e_3 \Downarrow v_3}{\textbf{if} \ e_1 \ \textbf{then} \ e_2 \ \textbf{else} \ e_3 \Downarrow v_3} \ (\text{ifEvalFalse})$$

Note: we never evaluate both branches

If e_1 evaluates to $\bot$ and e_2 evaluates to v_2, then

$$\textbf{if} \ e_1 \ \textbf{then} \ e_2 \ \textbf{else} \ e_3$$

evaluates to v_3

Questions?

Understanding Check: Functions and Applications

Functions (Syntax Rule)

Functions (Syntax Rule)

<expr> ::= fun <var> → <expr>

If x is a valid variable name and e is a well-formed expression, then

$$\text{fun } x \to e$$

is a well-formed expression

Functions (Typing Rule)

Functions (Typing Rule)

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \textsf{fun}\ x \rightarrow e : \tau_1 \rightarrow \tau_2}\ \text{(fun)}$$

If e has type τ_2 in the context Γ with the variable declaration $(x : \tau_1)$ added, then

$$\textsf{fun}\ x \rightarrow e$$

is of type $\tau_1 \rightarrow \tau_2$ in the context Γ

Functions (Semantic Rule)

Functions (Semantic Rule)

$$\frac{}{\texttt{fun}\ x \rightarrow e \Downarrow \lambda x . e}\ \text{(funEval)}$$

Under no premises, the expression

$$\texttt{fun}\ x \rightarrow e$$

evaluates to the *function value* $\lambda x . e$ (we'll talk more about function values later)

Application (Syntax Rule)

Application (Syntax Rule)

<expr> ::= <expr> <expr>

If e_1 is a well-formed expression and e_2 is a well-formed expression, then $e_1\ e_2$ is a well-formed expression

Application (Typing Rule)

Application (Typing Rule)

$$\frac{\Gamma \vdash e_1 : \tau_2 \rightarrow \tau \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\ e_2 : \tau} \text{(app)}$$

If e_1 has type $\tau_2 \rightarrow \tau$ under the context Γ and e_2 has type τ_2 under the context Γ, then $e_1\ e_2$ has type τ under the context Γ

Application (Semantic Rule)

Application (Semantic Rule)

$$\frac{e_1 \Downarrow \lambda\, x\, .\, e \qquad e_2 \Downarrow v_2 \qquad [v_2/x]e \Downarrow v}{e_1\ e_2 \Downarrow v}\text{(appEval)}$$

1. e_1 evaluates to a function value $\lambda x\,.\,e$
2. e_2 evaluates to v_2
3. e with v_2 substituted for x evaluates to v

It follows that $e_1\ e_2$ evaluates to v

Homework

Offline, go back to the recap slides at the beginning and compare the formal and informal descriptions...

We'll give a written reference for the rules we talk about in class

You never need to remember any rules in class

Summary

Inference rules formally describe how the typing and semantics of a programming language work

Tuples and **records** allow us to group data

Variants allow us to organize data by *possible outcomes*