EECS 490 – Lecture 14

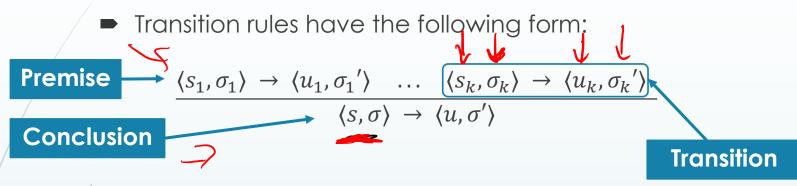
Formal Type Systems

1

Announcements

- Project 3 due Friday 10/27 at 8pm
- Midterm Tuesday 10/31 during class time
 - **■** Will be in 1109 FXB, not in this room
 - Covers lectures 1-12
 - ➤ You are allowed one 8.5x11" note sheet, double sided
 - Review session: Sunday 10/29 2-4pm in 1690 BBB
- Read §4.1 in the notes **before** Thursday's lecture

Review: Operational Semantics



- This is a conditional rule that means:
 - If s_1 computed in state σ_1 yields value u_1 and modified state σ_1 '
 - **...**
 - If s_k computed in state σ_k yields value u_k and modified state σ_k'
 - **Then** s computed in state σ yields value u and modified state σ'

In our convention, only transitions can appear in the premises or conclusion of a rule.

Review: Interpretation

Transition rules have the following form:

$$\frac{\langle s_1, \sigma_1 \rangle \to \langle u_1, \sigma_1' \rangle \dots \langle s_k, \sigma_k \rangle \to \langle u_k, \sigma_k' \rangle}{\langle s, \sigma \rangle \to \langle u, \sigma' \rangle}$$

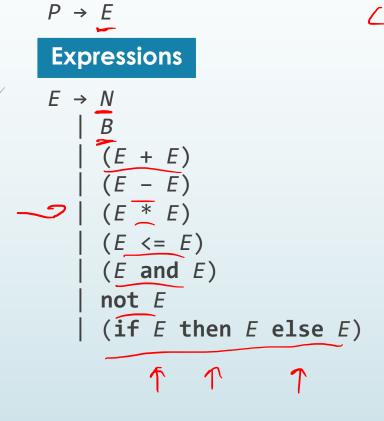
- A transition rule specifies a formula for interpreting a program fragment
 - If the interpreter sees a fragment of the form s, it can compute s by instead computing the fragments $s_1, ..., s_k$ that are in the premises, in the specified states
 - Computation terminates when no more transition rules can be applied

Type Systems

- Types play an important role in programming languages
 - Signify what data bits actually represent
 - Determine what operations are valid on a piece of data
 - Determine how to perform a particular operation
- In statically typed languages, the compiler computes types for each expression and checks that the types are used appropriately
- A type system specifies a method for computing types based on the syntactic structure of a program

Language

We will use a simple language of numbers and booleans:



Booleans

Numbers

 $N \rightarrow IntegerLiteral$

Types and Type Judgments

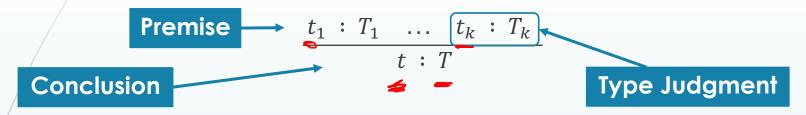
- Our language has two types: Int and Bool
- We determine the type of a **term** in the program based on its syntactic form and the types of its subterms
- A typing relation or type judgment has the form

$$t:T$$
 and it specifies that term t has type T

3: Int
$$(3+4)$$
: Int true: Bool $(3 < = 4)$: Bool $(3 < = 4)$: Bool

Typing Rule

Typing rules have the following familiar form:



- This is a conditional rule that means:
 - If t_1 has type T_1 , ..., and if t_k has type T_k
 - **Then** t has type T
- This specifies a formula for computing the type of a term in a compiler
 - If the compiler sees a term of the form t, it can compute the type of t by computing the types of $t_1, ..., t_k$ that are in the premises

Axioms

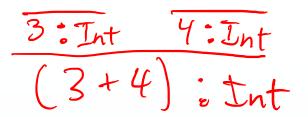
■ Literals can be typed directly with no premises:

IntegerLiteral : Int

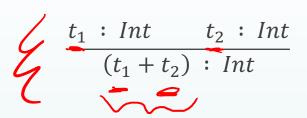
true : Bool

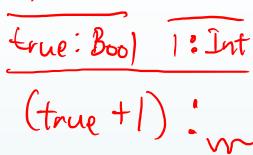
false : Bool

Addition



Rule for addition:





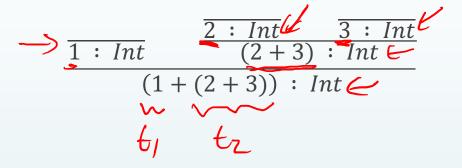
Meaning: if t_1 has type Int, and t_2 has type Int, then $(t_1 + t_2)$ also has type Int



- If either t_1 or t_2 does not have type Int, then the rule cannot be applied
 - The term $(t_1 + t_2)$ will not be typable, so it is erroneous

Type Derivations

 Typing rules lead to derivation trees, as in operational semantics



Arithmetic and Comparisons

Subtraction and multiplication rules similar to addition:

Comparisons require the operands to be *Ints* and produce a *Bool*:

 $(t_1 * t_2) : Int$

$$\frac{t_1 : Int}{(t_1 \le t_2) : Bool}$$

Conjunction and Negation

 Conjunction and negation require the operands to be Bools, produce a Bool as the result

```
t_1:Bool t_2:Bool t_1:Bool t_2:Bool t_2:Bool t_2:Bool t_2:Bool t_2:Bool
```

Conditionals (if b then Delse true)

- A conditional requires its two branches to have the same type
 - The term (if b then 0 else 1) should be typable as Int, while (if b then true else false) should be typable as Bool

$$\begin{cases} \frac{t_1 : Bool}{\text{(if } t_1 \text{ then } t_2 \text{ else } t_3)} : T \end{cases}$$

Type variable can be any type

Variables

■ Let's add variables to our language:

$$E \rightarrow ($$
 let $V = E$ in E $)$
 $V \rightarrow Identifier$

- The let construct has the semantics of replacing each occurrence of the variable in its body with the value bound to the variable
 - Example: $(let x = 3 in (x + 2)) \rightarrow 5$
- We will assume for simplicity that all variable names in a program are distinct

Type Environments

- In order to type the body of a let, we need to keep track of the mapping between the variables that are in scope and their types
- The type context or type environment, denoted by Γ, maps variables to types
- The notation $x:T\in\Gamma$ means that Γ maps the variable x to type T
- The environment Γ , x: T is the same as Γ , with the addition of the mapping x: T
- Type judgments are now in the context of an environment:

$$\Gamma \vdash t : T$$

This means that term t has type T within the context of Γ

Rules with Type Environments

True in any context, so context is / \vdash false : Boolelided

Frue: Bool $\Gamma \vdash t_1 : Bool \quad \Gamma \vdash t_2 : T \quad \Gamma \vdash t_3 : T$ $\Gamma \vdash (\mathbf{if} \ t_1 \ \mathbf{then} \ t_2 \ \mathbf{else} \ t_3) : T$

⊢ IntegerLiteral : Int

 $\Gamma \vdash t_1 : Bool \qquad \Gamma \vdash t_2 : Bool$ $\Gamma \vdash (t_1 \text{ and } t_2) : Bool$

> $\Gamma \vdash t : Bool$ $\Gamma \vdash \mathbf{not} \, t : Int$

 $\Gamma \vdash t_1 : Int \qquad \Gamma \vdash t_2 : Int$ $\Gamma \vdash (t_1 + t_2) : Int$

 $\Gamma \vdash t_1 : Int \qquad \Gamma \vdash t_2 : Int$ $\Gamma \vdash (t_1 * t_2) : Int$

 $\Gamma \vdash t_1 : Int \qquad \Gamma \vdash t_2 : Int \qquad \Gamma \vdash t_1 : Int \qquad \Gamma \vdash t_2 : Int$ $\Gamma \vdash (t_1 - t_2) : Int$ $\Gamma \vdash (t_1 \le t_2) : Bool$

Variable Typing Rule

Rule for typing a variable retrieves its mapping from the context, assuming there is a mapping:

$$\underbrace{x:T\in\Gamma}_{\Gamma\vdash x:T}$$

Rule for a let types the body in a context extended with a mapping for the variable:

Example

Type derivation for (**let** x = 3 **in** (x + 2)) in an arbitrary context:

```
x: Int \in x: Int
x: Int \mapsto x: Int
x: Int \mapsto x
```

■ We'll start again in five minutes.

Functions

■ Let's now add functions that take in a single argument:

$$E \rightarrow (\underline{lambda} \ V : T \cdot E) \leftarrow \underline{Abstraction}$$

$$| (\underline{E} \ E) \underline{Application}$$

Function parameters are explicitly typed, so we need to add types to our grammar:

Function Types

 A function takes in an argument of a specific type and produces a return value of a specific type

(lambda
$$x : Int . (x <= 0)) : Int \rightarrow Bool$$
Parameter type Return type

The type constructor → is right associative

```
(lambda x : Int . (lambda y : Int . (x <= y))) : Int \rightarrow Int \rightarrow Bool
```

Parameter type

Return type

Function Abstraction

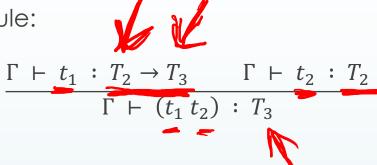
Typing rule:

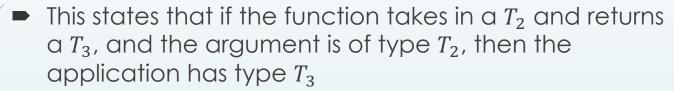
$$\frac{\Gamma, v: T_1 \vdash t_2 : T_2}{\Gamma \vdash (\mathbf{lambda} \ v: T_1 \cdot t_2) : T_1 \rightarrow T_2}$$

- This states that if the body, when assigned a type within a context that maps v to T_1 , has type T_2 , then the function has type $T_1 \rightarrow T_2$
 - lacktriangle i.e. it takes in a T_1 as an argument and returns a T_2

Function Application

Typing rule:





25

Example

■ Type derivation for $((lambda x : Int . (x \le 0)) 3)$:

```
\frac{x: Int \in x: Int}{x: Int \vdash x : Int} \xrightarrow{\vdash 0 : Int} x: Int \vdash (x <= 0) : Bool}
\vdash (lambda x : Int . (x <= 0)) : Int \rightarrow Bool \qquad \vdash 3 : Int
\vdash ((lambda x : Int . (x <= 0))) : Bool
```

Subtyping

■ Let's now add *Float* as another numerical type:

We would like to allow a call such as the following:

$$\left(\left(\mathbf{lambda} \ x : Float \ . \ (x + 1.0) \right) 3 \right)$$

- Conceptually, every integer is a floating-point number, so we'd like to allow an *Int* where a *Float* is expected
- We specify that Int is a **subtype** of Float

Subtype Relation

The subtype relation is denoted as:

- lacktriangle This means that S is a subtype of T
- The relation must be a *preorder*:
 - It is **reflexive**, so that S <: S for any type S
 - It is **transitive**, so that S <: T and T <: U imply S <: U
- In many languages, the relation is also a partial order:
 - It is **antisymmetric**, so that S <: T and T <: S imply that S = T
- In our language, we have:

Int <: *Float*

Subsumption Rule

The subsumption rule allows a term to be typed as a supertype of its actual type:

$$\frac{\Gamma \vdash \underline{s} : \underline{S} \qquad \underline{S} <: \underline{T}}{\Gamma \vdash \underline{s} : \underline{T}}$$

■ The rule encodes a notion of substitutability, allowing a subtype to be used where a supertype is expected:

$$\frac{\Gamma \vdash f : Float \rightarrow Float}{\Gamma \vdash x : Int} \frac{Int <: Float}{\Gamma \vdash x : Float}$$

$$\Gamma \vdash (f x) : Float$$

Joins

- We need to rewrite the arithmetic rules to work with both Ints and Floats
- The result type should be the least upper bound, or join, of the operand types
 - The join $T = T_1 \sqcup T_2$ is the minimal type T such that $T <: T_1$ and $T <: T_2$

$$Int = Int \sqcup Int$$

 $Float = Int \sqcup Float$
 $Float = Float \sqcup Float$

■ Rule for addition:

Require operand type to be a number

$$\frac{\Gamma \vdash t_1 : T_1 \quad \Gamma \vdash t_2 : T_2 \quad T_1 <: Float \quad T_2 <: Float \quad T = T_1 \sqcup T_2}{\Gamma \vdash (t_1 + t_2) : T}$$

The Top Type

Many languages have a Top type (also written as T), that is a supertype of every other type:

- Example: object in Python
- Adding Top to our language ensures that every pair of types has a join¹
- We can then relax the rule for conditionals:

$$\frac{\Gamma \vdash t_1 : Bool}{\Gamma \vdash (\mathbf{if} \ t_1 \ \mathbf{then} \ t_2 \ \mathbf{else} \ t_3) : T} \qquad T = T_2 \sqcup T_3$$

Contravariant Parameters

- A function that takes in a more general parameter type should be substitutable for a function that takes in a more specific parameter type
- For example, the following should be valid:

```
((\mathbf{lambda} \ f : Int \rightarrow Bool \ . \ (f \ 3)) \ (\mathbf{lambda} \ x : Float \ . \ \mathbf{true}))
```

- Thus, if $T_1 <: S_1$, then it should be that $S_1 \to U <: T_1 \to U$
- This permits a contravariant parameter type, since the direction of <: is switched between the parameter and function types

Covariant Return Types

- A function that takes returns a more specific type should be substitutable for a function returns a more general type
- For example, the following should be valid:

```
((\mathbf{lambda} f : Int \rightarrow Float . (f 3)) (\mathbf{lambda} x : Int . x))
```

- Thus, if $S_2 <: T_2$, then it should be that $U \to S_2 <: U \to T_2$
- This permits a covariant return type, since the direction of <: is the same between the return and function types

Subtyping for Functions

In general, a function is substitutable for another if the parameter types are contravariant and the return types are covariant:

$$((\mathbf{lambda} f : Int \rightarrow Float . (f 3)) (\mathbf{lambda} x : Float . 0))$$

Rule for subtyping functions:

$$\frac{T_1 <: S_1 \qquad S_2 <: T_2}{S_1 \to S_2 <: T_1 \to T_2}$$