Types + Type Checking

These slides adapted from notes by George Necula, slides from Paul Hilfinger

First let’s remember some of our key terminology

- Static

- Dynamic

Types

- What is a type?
  - The notion varies from language to language

- Consensus
  - A set of values
  - A set of operations on those values

- Classes are one instantiation of the modern notion of type

Why Do We Need Type Systems?

Consider the assembly language fragment

```
add r1, r2
```

What are the types of `r1, r2`?

Types and Operations

- Most operations are legal only for values of some types
  - It doesn't make sense to add a function pointer and an integer in C
  - It does make sense to add two integers
  - But both have the same assembly language implementation!

Type Systems

- A language's type system specifies which operations are valid for which types
- The goal of type checking is to ensure that operations are used with the correct types
  - Enforces intended interpretation of values, because nothing else will!
- Type systems provide a concise formalization of the semantic checking rules

So…what are we doing here?

- Approx quote from Andrew Appel:
  - “When the static analysis is wrong, it's the analysis’s fault. When the type checker is wrong, it's the programmer’s fault.”
    - With many apologies if I got this quote very very wrong. I Googled for it a long time…
- We’re ***intentionally putting some programs off limits for programmers***

So…what are we doing here?

- `5 + “test”`
  - Ok, we probably don’t actually want this to run, so we’re pretty happy to put this off limits
- `let returns_number_or_false`
- `let adds_ints_or_floats`
  - But we might actually want to write these in OCaml…

Type Checking Overview

- Three kinds of languages:
  - Statically typed: All or almost all checking of types is done as part of compilation (OCaml, C)
  - Dynamically typed: Almost all checking of types is done as part of program execution (Scheme)
  - Untyped: No type checking (machine code)

The Type Wars

- Competing views on static vs. dynamic typing
- Static typing proponents say:
  - Static checking catches many programming errors at compile time
  - Avoids overhead of runtime type checks
    - In our language, we get our tag bits back!!
- Dynamic typing proponents say:
  - Static type systems are restrictive
  - Rapid prototyping easier in a dynamic type system

- Type Checking is the process of checking that the program obeys the Type System

- Often involves inferring types for parts of the program
  - Can call the process Type Inference when inference is necessary

Rules of Inference

- We've seen two examples of formal notation specifying parts of a compiler
  - Regular expressions (for the tokenizer/lexer)
  - Context-free grammars (for the parser)

- The appropriate formalism for type checking is logical rules of inference

Why Rules of Inference?

- Inference rules have the form

  If Hypothesis is true, then Conclusion is true

- Type checking computes via reasoning

  If E_1 and E_2 have certain types, then E_3 has a certain type

- Rules of inference are a compact notation for “If-Then” statements

From English to an Inference Rule

- The notation can be weird for folks, but we'll just say it with English words to remind ourselves what it means

- Start with a simplified system and gradually add features

- Building blocks
  - Symbol $\wedge$ is "and"
  - Symbol $\Rightarrow$ is "if-then"
  - $x{:}T$ is "x has type T"

From English to an Inference Rule (2)

If e_1 has type Int and e_2 has type Int,

then $e_1 + e_2$ has type Int

$(e_1$ has type Int $\wedge$ e_2 has type Int$)$ $\Rightarrow$

$e_1 + e_2$ has type Int

$(e_1$: Int $\wedge$ e_2: Int$)$ $\Rightarrow$ $e_1 + e_2$: Int

From English to an Inference Rule (3)

The statement

$$(e_1: \text{Int} \wedge e_2: \text{Int}) \Rightarrow e_1 + e_2: \text{Int}$$

is a special case of

$$(\text{Hypothesis}_1 \wedge \ldots \wedge \text{Hypothesis}_n) \Rightarrow \text{Conclusion}$$

This is an inference rule

Notation for Inference Rules

- By tradition inference rules are written

$$\frac{\vdash \text{Hypothesis}_1 \quad \ldots \vdash \text{Hypothesis}_n}{\vdash \text{Conclusion}}$$

- Type rules have hypotheses and conclusions of the form:

$$\vdash e : T$$

- $\vdash$ means "we can prove that . . ."

A Couple Example Rules

$$\frac{}{\vdash i : \mathsf{Int}}\ [\mathsf{Int}] \quad (i \text{ is an integer})$$

$$\frac{\begin{array}{c}\vdash e_1 : \mathsf{Int} \\ \vdash e_2 : \mathsf{Int}\end{array}}{\vdash e_1 + e_2 : \mathsf{Int}}\ [\mathsf{Add}]$$

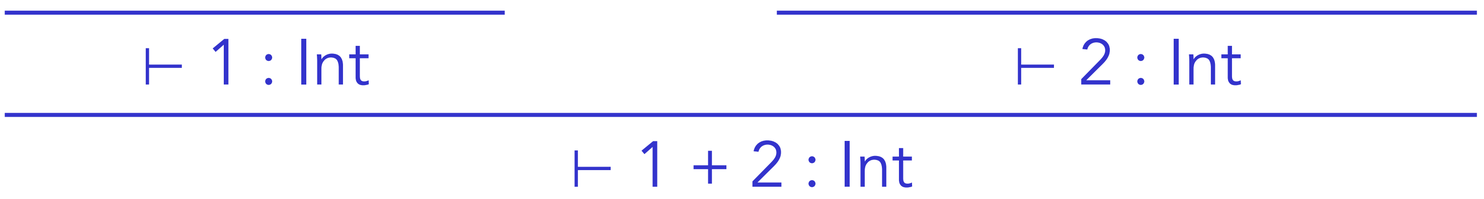
Two Rules (Cont.)

- These rules give templates describing how to type integers and addition expressions
- By filling in the templates, we can produce complete typings for expressions

How will we use these?

- We can use the same rules for a couple different purposes:
  - Type checking:
    - Say the user has annotated their program with types. Have they used types that the type system allows?
  - Type inference:
    - Say the user hasn't annotated their program with types. Can we infer types that the type system allows? (Basically fill in those type annotations for the user, even though we won't change the text.)

Example: 1 + 2



Soundness

- A type system is sound if
  - Whenever $\vdash e : T$
  - Then e evaluates to a value of type T
- We **only** want sound rules
  - But some sound rules are better than others; here's one that's not very useful:

$$\frac{}{\vdash i : \text{Any}} \quad \text{(i is an integer)}$$

Type Checking Proofs

- Type checking proves facts e : T
  - One type rule is used for each kind of expression

- In the type rule used for an AST node e:
  - The hypotheses are the proofs of types of e's subexpressions
  - The conclusion is the proof of type of e

Rules for Constants

$$\frac{}{\vdash \text{false} : \text{Bool}}\ [\text{Bool}]$$

$$\frac{}{\vdash s : \text{String}}\ [\text{String}]$$

(s is a string constant)

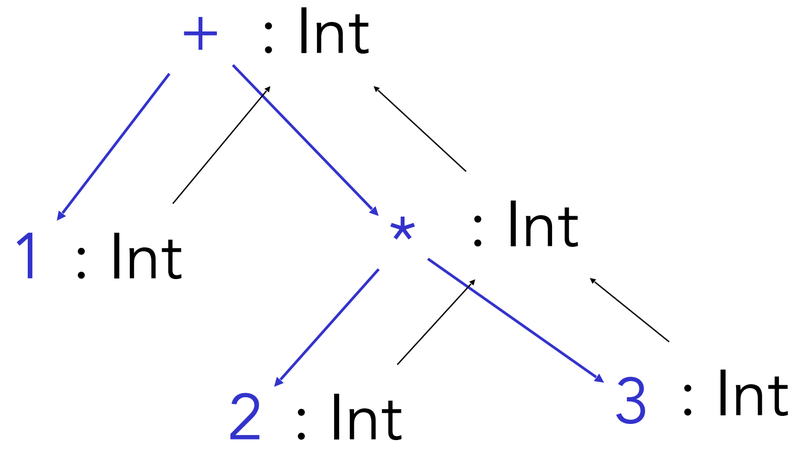
Two More Rules

$$\frac{\vdash e : \text{Bool}}{\vdash (\text{not } e) : \text{Bool}} \quad \text{[Not]}$$

$$\frac{\vdash e_1 : T_1 \to T_2 \qquad \vdash e_2 : T_1}{\vdash (e_1\ e_2) : T_2} \quad \text{[Function Application]}$$

Typing: Example

- Typing for (+ 1 (* 2 3))



Typing Derivations

- The typing reasoning can be expressed as a tree:

$$\dfrac{\dfrac{}{\vdash 1 : \mathrm{Int}} \qquad \dfrac{\dfrac{}{\vdash 2 : \mathrm{Int}} \qquad \dfrac{}{\vdash 3 : \mathrm{Int}}}{\vdash 2 * 3 : \mathrm{Int}}}{\vdash 1 + 2 * 3 : \mathrm{Int}}$$

- The root of the tree (at the bottom here) is the whole expression
- Each node is an instance of a typing rule
- Leaves are the rules with no hypotheses

A Problem

- What is the type of a variable reference?

$$\frac{}{\vdash x : ?}\ [\text{Var}] \qquad (x \text{ is an identifier})$$

- This rules does not have enough information to give a type.
  - We need a hypothesis of the form "we are in the scope of a declaration of x with type T")

Solution: Put more info in the rules!

- A type environment gives types for free variables
  - A type environment is a mapping from Identifiers to Types

Type Environments

Let Γ be a function from Identifiers to Types

The sentence $\Gamma \vdash e : T$

is read: Under the assumption that variables in the current scope have the types given by Γ, it is provable that the expression e has the type T

Modified Rules

The type environment is added to the earlier rules:

$$\frac{}{\Gamma \vdash i : \text{Int}}\ [\text{Int}] \quad (i \text{ is an integer})$$

$$\frac{\Gamma \vdash e_1 : \text{Int} \qquad \Gamma \vdash e_2 : \text{Int}}{\Gamma \vdash (+\ e_1\ e_2) : \text{Int}}\ [\text{Add}]$$

New Rules

And we can write new rules:

$$\frac{}{\Gamma \vdash x : T}\ \text{[Var]} \qquad (\text{if } \Gamma\ (x) = T)$$

Let

- Remember `let` creates a variable x with given type T_0 that is then defined throughout e_1

$$\frac{\Gamma \vdash e_0 : T_0 \qquad \Gamma\,[T_0/x] \vdash e_1 : T_1}{\Gamma \vdash (\text{let } ((x : T_0\ e_0))\ e_1 : T_1)}\ [\text{Let}]$$

Let Example.

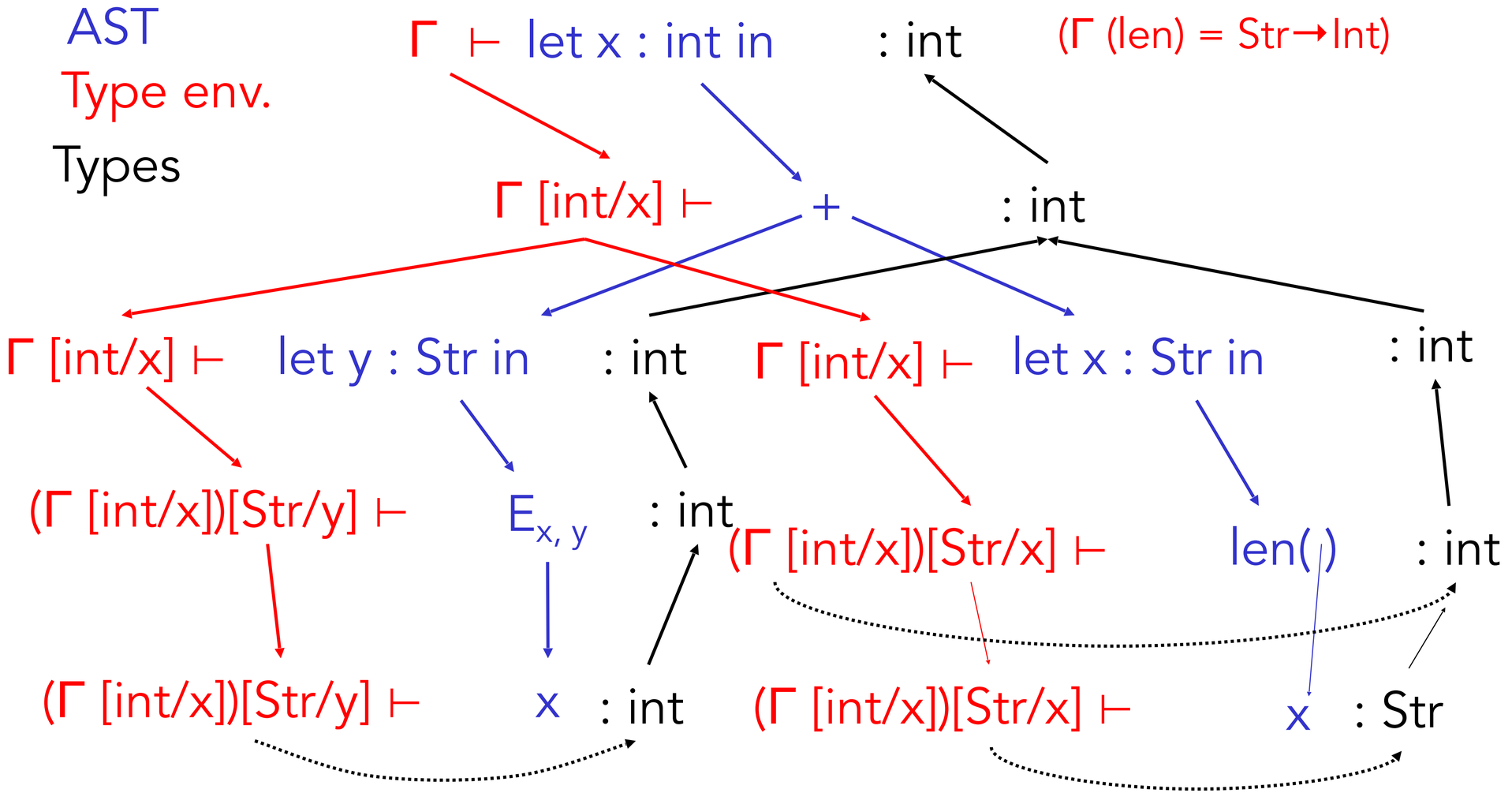
- Consider the expression

  (let ((x : T_0 G : T_0)) (+ (let ((y : T_1)) $E_{x, y}$) (let ((x : T_2)) $F_{x, y}$)))

  (where $E_{x, y}$ and $F_{x, y}$ are some expression that contain occurrences of "x" and "y")

- Scope
  - of "y" is $E_{x, y}$
  - of outer "x" is $E_{x, y}$
  - of inner "x" is $F_{x, y}$
- This is captured precisely in the typing rule.

Let Example (not with our language!)



Notes

- The type environment gives types to the free identifiers in the current scope

- The type environment is passed down the AST from the root towards the leaves

- Types are computed up the AST from the leaves towards the root

Where are types going these days?

- We might think this is a good program to allow:
  - `let adds_ints_or_floats` (our example from before)
- A lot of the modern work in type systems is for making more "good" programs allowable within type systems

- Turns out even in dynamically typed languages, people are wanting static types, so there's also work on putting them on top of dynamically typed languages:
  - TypeScript for JavaScript
  - Mypy for Python

Rules we've seen so far

$$\frac{}{\Gamma \vdash i : \text{Int}}\ [\text{Int}] \quad (i \text{ is an integer})$$

$$\frac{}{\Gamma \vdash \text{false} : \text{Bool}}\ [\text{Bool}]$$

$$\frac{\Gamma \vdash e : \text{Bool}}{\Gamma \vdash (\text{not } e) : \text{Bool}}\ [\text{Not}]$$

$$\frac{\Gamma \vdash e_1 : \text{Int} \qquad \Gamma \vdash e_2 : \text{Int}}{\Gamma \vdash (+\ e_1\ e_2) : \text{Int}}\ [\text{Add}]$$

$$\frac{\Gamma \vdash e_0 : T_0 \qquad \Gamma\,[T_0/x] \vdash e_1 : T_1}{\Gamma \vdash (\text{let } ((x : T_0\ e_0))\ e_1 : T_1)}\ [\text{Let}]$$

$$\frac{\Gamma \vdash e_1 : T_1 \to T_2 \qquad \Gamma \vdash e_2 : T_1}{\Gamma \vdash (e_1\ e_2) : T_2}\ [\text{Function Application}]$$

$$\frac{}{\Gamma \vdash x : T}\ [\text{Var}] \quad (\text{if } \Gamma(x) = T)$$

(1) What rule do we need to finish Bools?

$$\frac{}{\quad} \; [\text{Bool}_2]$$

(2) What rule do we need for less than?

$$\frac{}{\quad} \; [\text{Less Than}]$$

(3) What rule do we need for function *definition*?

$$\frac{}{\Gamma \vdash (\text{define } (f\ x{:}\ T_1)\ e_1) : T_1 \to T_2} \; [\text{Function Defn}]$$

(4) Draw the AST for: `(not (< 5 7))`

(5) Now associate a type with each node above.

(6) Now draw the typing derivation tree that produces those types.