Rules of Inference

- We have seen two examples of formal notation specifying parts of a compiler
  - Regular expressions
  - Context-free grammars
- 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 is easy to read with practice
- 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

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

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

$$\frac{\text{Hypothesis}_1 \cdots \text{Hypothesis}_n}{\text{Conclusion}}$$

- Cool type rules have hypotheses and conclusions
  ` e:T
- ` means "it is provable that . . ."

Two Rules

$$\frac{\text{i is an integer}}{\text{` i: Int}} \quad \text{[Int]}$$

$$\frac{\text{` } e_1 : \text{Int} \qquad \text{` } e_2 : \text{Int}}{\text{` } e_1 + e_2 : \text{Int}} \quad \text{[Add]}$$

Two Rules (Cont.)

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

Example: 1 + 2

$$\frac{\dfrac{\text{1 is an integer}}{\text{` 1: Int}} \qquad \dfrac{\text{2 is an integer}}{\text{` 2: Int}}}{\text{` 1+2: Int}}$$

Soundness

- A type system is *sound* if
  - Whenever ` e:T
  - Then e evaluates to a value of type T
- We only want sound rules
  - But some sound rules are better than others:

$$\frac{\text{i is an integer}}{\text{` i:Object}}$$

Type Checking Proofs

- Type checking proves facts e: T
  - Proof is on the structure of the AST
  - Proof has the shape of the AST
  - One type rule is used for each AST node
- In the type rule used for a node e:
  - Hypotheses are the proofs of types of e's subexpressions
  - Conclusion is the type of e
- Types are computed in a bottom-up pass over the AST

Rules for Constants

$$\frac{}{\text{` false: Bool}} \quad \text{[Bool]}$$

$$\frac{\text{s is a string constant}}{\text{` s: String}} \quad \text{[String]}$$

Rule for New

new T produces an object of type T

- Ignore SELF_TYPE for now . . .

$$\frac{}{\vdash \text{new } T : T} \quad [\text{New}]$$

Two More Rules

$$\frac{\vdash e : \text{Bool}}{\vdash \neg e : \text{Bool}} \quad [\text{Not}]$$

$$\frac{\begin{array}{c}\vdash e_1 : \text{Bool} \\ \vdash e_2 : T\end{array}}{\vdash \text{while } e_1 \text{ loop } e_2 \text{ pool} : \text{Object}} \quad [\text{Loop}]$$

qui, nella specifica del while, diversamente da Toy3,
sia il corpo e2 che il while stesso hanno un tipo.
Si noti che, qui, il tipo generico T di e2 non viene usato e che il tipo
del while è sempre Object

A Problem

- What is the type of a variable reference?

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

- The local, structural rule does not carry enough information to give x a type.

A Solution

- Put more information in the rules!
- A *type environment* gives types for *free* variables
  - A type environment is a function from ObjectIdentifiers to Types **ovvero: nome id -> tipo**
  - A variable is free in an expression if it is not defined within the expression

Type Environments

Let O be a function from ObjectIdentifiers to Types

The sentence

$$O \vdash e : T$$

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

Possiamo quindi pensare che O rappresenti la lista di tabelle dei simboli nel punto in cui consideriamo l'espressione e. Il tipo T per e viene calcolato consultando O per tutti i nomi id presenti in O e tenendo conto degli operatori coinvolti (per esempio e potrebbe essere x + y)

Modified Rules

The type environment is added to the earlier rules:

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

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

Per esempio, dato x : 1; e1 = x+2; e2 = x*3; s=e1+e2;
Dalla dichiarazione x : 1 in poi, O sarà data dalla sola associazione (x |-> Int)
La regola si legge così: se è possibile provare che e1 ed e2
sono Int nello stesso type environment O (ed in questo esempio lo sono) allora è
dimostrato che anche la somma e1+ e2 è Int *utilizzando lo stesso type environment O*.