Type Systems for Programming Languages

CS4430/7430

Some history…

- Gottlob Frege was a German mathematician and philosopher working on the foundations of arithmetic.
- 1879: *Begriffsschrift.*
- 1884: *The Foundations of Arithmetic.*
- 1893: *Basic Laws of Arithmetic, vol. 1.*
- 1903: *Basic Laws of Arithmetic, vol. 2.*

Gottlob Frege
1848-1925

Some history…

- Gottlob Frege was a german mathematician and philosopher working on the foundations of arithmetic.
- 1879: *Begriffsschrift.*
- 1884: *The Foundations of Arithmetic.*
- 1893: *Basic Laws of Arithmetic, vol. 1.*
- 1903: *Basic Laws of Arithmetic, vol. 2.*

But just as Volume 2 was going to print, he received a letter…

Gottlob Frege
1848-1925

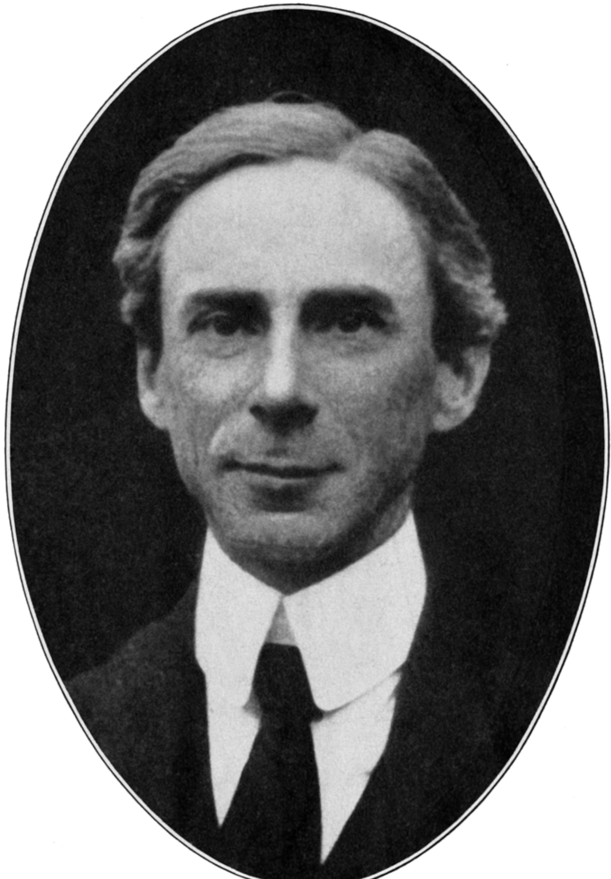
Some history…

Russell's paradox:

- Let X be the set of everything not in X, i.e.:

$$X = \{x \mid x \notin X\}$$

- Is $X \in X$?
  - Yes: But if $X \in X$, then $X \notin X$.
  - No: But if $X \notin X$, then $X \in X$.

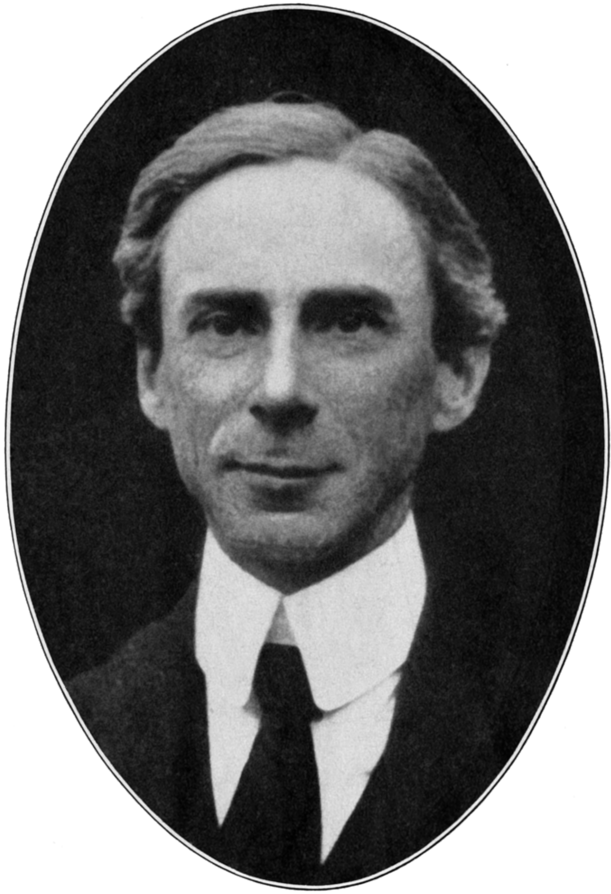


Bertrand Russell
1872-1970

Some history…

- 1908: Russell's fix: Types!
- 1910-1927 (with Whitehead): *Principia Mathematica.*
  - Goal: axioms and rules from which all mathematical truths could be derived.

> "From this proposition it will follow, when arithmetical addition has been defined, that 1+1=2."
>
> –Volume I, 1st edition, page 379



Bertrand Russell
1872-1970

What are type systems?

"A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute."

"A type system can be regarded as calculating a kind of *static* approximation to the run-time behaviors of the terms in a program."

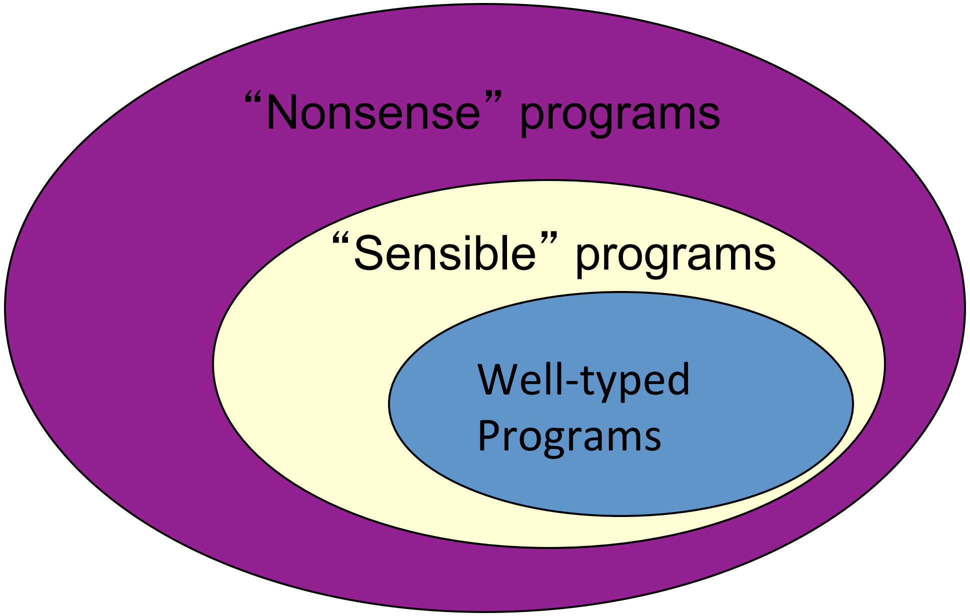
—Benjamin Pierce, *Types and Programming Languages*

What is a Type System?

- A type system is a syntactic discipline for enforcing levels of abstraction.
  - Ensures that bad things do not happen.

- A type system rules out nonsense programs.
  - Adding a function to a string
  - Interpreting an integer as a pointer
  - Violating interfaces

Type checking



What is a Type System?

- How can this be a good thing?
  - Expressiveness arises from strictures: restrictions entail stronger invariants
  - Flexibility arises from controlled relaxation of strictures, not from their absence.
- A type system is fundamentally a verification tool that suffices to ensure invariants on execution behavior.

Why Types are Useful

- error detection: early detection of common programming errors
- safety: well typed programs do not go wrong
- design: types provide a language and discipline for design of data structures and program interfaces
- abstraction: types enforce language and programmer abstractions

Why Types are Useful (cont)

- verification: properties and invariants expressed in types are verified by the compiler ("a priori guarantee of correctness")
- software evolution: support for orderly evolution of software
  - consequences of changes can be traced
- documentation: types express programmer assumptions and are verified by compiler

Types Induce Invariants

- Types induce invariants on programs.
  - If e : int, then its value must be an integer.
  - If e : int → int, then it must be a function taking and yielding integers.
  - If e : filedesc, then it must have been obtained by a call to open.
  - If e : int{H}, then no "low clearance" expression can read its value.

Types Induce Invariants

- These invariants provide
  - Safety properties: well-typed programs do not "go wrong".
  - Equational properties: when are two expressions interchangeable in all contexts.
  - Representation independence (parametricity).

Typing judgments

e : `int`

- Asserts that evaluation of *e* will result in a value of type `int`.
- But *e* must be *well-typed* for this assertion to actually hold.

Typing judgments

$$2 : \texttt{int}$$

- Asserts that evaluation of *e* will result in a value of type `int`.
- But *e* must be *well-typed* for this assertion to actually hold.

Typing judgments

1 + 2 : `int`

- Asserts that evaluation of *e* will result in a value of type `int`.
- But *e* must be *well-typed* for this assertion to actually hold.

Typing judgments

true : `int`

- Asserts that evaluation of *e* will result in a value of type `int`.
- But *e* must be *well-typed* for this assertion to actually hold.

Typing judgments

1 + true : `int`

- Asserts that evaluation of *e* will result in a value of type `int`.
- But *e* must be *well-typed* for this assertion to actually hold.

Typing judgments

Solution: a set of rules (a logic) which will derive *only* valid typing judgments. Now, if we can derive a judgment of the form:

$$e : t$$

It should be the case that the expression *e* is well-typed and when it is evaluated, the result will have type *t.*

Type judgments

$$(1)\quad \frac{e_1 : \texttt{int} \qquad e_2 : \texttt{int}}{e_1 + e_2 : \texttt{int}}$$

Type judgments

(2) $$\frac{}{n : \texttt{int}}$$ (Where *n* is an integer literal.)

(1) $$\frac{e_1 : \texttt{int} \qquad e_2 : \texttt{int}}{e_1 + e_2 : \texttt{int}}$$

Type judgments

$$\text{(2)}\ \frac{}{1 : \texttt{int}} \qquad \text{(2)}\ \frac{}{2 : \texttt{int}}$$

Type judgments

$$\dfrac{(2)\ \dfrac{}{1 : \texttt{int}} \qquad (2)\ \dfrac{}{2 : \texttt{int}}}{1 + 2 : \texttt{int}}\ (1)$$

Type judgments

$$(1)\;\dfrac{(2)\;\dfrac{}{1 : \texttt{int}} \qquad (2)\;\dfrac{}{2 : \texttt{int}}}{1 + 2 : \texttt{int}} \qquad (2)\;\dfrac{}{3 : \texttt{int}}$$

Type judgments

$$(1)\ \dfrac{(1)\ \dfrac{(2)\ \dfrac{}{1 : \texttt{int}} \qquad (2)\ \dfrac{}{2 : \texttt{int}}}{1 + 2 : \texttt{int}} \qquad (2)\ \dfrac{}{3 : \texttt{int}}}{1 + 2 + 3 : \texttt{int}}$$

Typing judgments

But what about variables?

$$x : t$$

What is *t*, where *x* is a variable?

Typing judgments

But what about variables?

$$\Gamma, x : t \vdash x : t$$

What is *t*, where *x* is a variable?

Solution: look it up in the environment (i.e., the symbol table).

Typing judgments

$$\Gamma, x : t \vdash x : t$$

Called the *context*—a mapping from name to types.

Type judgments

(2) $$\frac{}{n : \texttt{int}}$$ (Where *n* is an integer literal.)

(1) $$\frac{e_1 : \texttt{int} \qquad e_2 : \texttt{int}}{e_1 + e_2 : \texttt{int}}$$

Type judgments

$$\text{(2)}\ \frac{}{\Gamma \vdash n : \texttt{int}}$$ (Where n is an integer literal.)

$$\text{(1)}\ \frac{\Gamma \vdash e_1 : \texttt{int} \qquad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 + e_2 : \texttt{int}}$$

Where Γ is an arbitrary context.

Typing judgments

$$(3)\quad \frac{}{\Gamma, x : t \vdash x : t}$$

Typing judgments

$$(4)\quad \frac{\Gamma, x : t_1 \vdash stmts : t_2}{\Gamma \vdash \text{let } x : t_1 \,;\, stmts : t_2}$$

$$(3)\quad \frac{}{\Gamma, x : t \vdash x : t}$$

Typing judgments

$\Gamma \vdash$ let v : `int` ; v + 1 : `int`

Typing judgments

$$(4)\quad \frac{\Gamma, \mathrm{v} : \texttt{int} \vdash \mathrm{v} + 1 : \texttt{int}}{\Gamma \vdash \mathrm{let\ v} : \texttt{int}\ ;\ \mathrm{v} + 1 : \texttt{int}}$$

Typing judgments

$$\dfrac{\dfrac{\Gamma, \mathsf{v} : \texttt{int} \vdash \mathsf{v} : \texttt{int} \qquad \Gamma, \mathsf{v} : \texttt{int} \vdash 1 : \texttt{int}}{\Gamma, \mathsf{v} : \texttt{int} \vdash \mathsf{v} + 1 : \texttt{int}}\ (1)}{\Gamma \vdash \mathsf{let}\ \mathsf{v} : \texttt{int}\ ;\ \mathsf{v} + 1 : \texttt{int}}\ (4)$$

Typing judgments

(3) $$\frac{}{\Gamma, \mathsf{v} : \texttt{int} \vdash \mathsf{v} : \texttt{int}}$$

(2) $$\frac{}{\Gamma, \mathsf{v} : \texttt{int} \vdash 1 : \texttt{int}}$$

(1) $$\frac{}{\Gamma, \mathsf{v} : \texttt{int} \vdash \mathsf{v} + 1 : \texttt{int}}$$

(4) $$\frac{}{\Gamma \vdash \mathsf{let}\ \mathsf{v} : \texttt{int}\ ;\ \mathsf{v} + 1 : \texttt{int}}$$

Properties of type systems

- Uniqueness of types: For any expression e, is there at most one type *t* for which the judgment *e* : *t* holds?
- Uniqueness of derivations: Assuming such a *t* is unique, is the derivation also unique?
- Assuming the above, is finding that *t* (*type checking*) decidable?

Type Checking

- What type has every (sub-)expression?
- Is it consistent?
- How do you specify a language's typing semantics?
  - Sometimes called “static semantics”.
- What else might you wish to check?
  - In C: break only valid inside while & for loops.

Type systems and Languages

- Many modern programming languages are strongly-typed
  - Java, ML, Haskell,…
  - “strongly” meaning that each “subprogram” must be typed
- Some aren't (or barely are):
  - C, LISP, C++,PERL
- Why types?
  - allow static checking for common programming errors
  - data objects of a particular type can be reasoned about without thinking of their representations
  - E.g., consider a situation where that **is not** true
    - type-casting a pointer in C
- How do we specify type checking for languages like Java, ML, and Haskell?
  - type derivation systems = type judgments + inference rules

Typing judgments

Typical typing judgement:

$A \vdash e : T$

Can be read as:

in symbol table A, expression e has type T

“:” = “has type”

“$\vdash$” = “implies”

Type Inference Systems

$$\frac{A \vdash e1 : Bool \qquad A \vdash e2 : T \qquad A \vdash e3 : T}{A \vdash IF\ e1\ THEN\ e2\ ELSE\ e3 : T}$$

IF e1 can be shown to have type Bool
e2 can be shown to have type T
e3 can be shown to have (the same) type T

THEN
"if e1 then e2 else e3" can be shown to have the type T.

Inference rules for small language

Small Grammar

Exp → Exp + Exp
Exp → Exp = = Exp
Exp → IF Exp THEN Exp ELSE Exp
Exp → ID
Exp → INT
Exp → LET ID := Exp IN Exp

Grammar for Types
T → Bool
T → Int

$$\frac{A \vdash e1 : Int \qquad A \vdash e2 : Int}{A \vdash e1 + e2 : Int}$$

$$\frac{A \vdash e1 : T \qquad A \vdash e2 : T}{A \vdash e1 == e2 : Bool}$$

$$\frac{A \vdash e1 : Bool \qquad A \vdash e2 : T \qquad A \vdash e3 : T}{A \vdash \text{IF } e1 \text{ THEN } e2 \text{ ELSE } e3 : T}$$

Inference rules for small language

Observe the type variables here: stand for any type in the language

$$\frac{A \vdash e1 : Int \qquad A \vdash e2 : Int}{A \vdash e1 + e2 : Int}$$

$$\frac{A \vdash e1 : T \qquad A \vdash e2 : T}{A \vdash e1 == e2 : Bool}$$

$$\frac{A \vdash e1 : Bool \qquad A \vdash e2 : T \qquad A \vdash e3 : T}{A \vdash \text{IF } e1 \text{ THEN } e2 \text{ ELSE } e3 : T}$$

Inference rules for small language

Small Grammar

Exp → Exp + Exp
Exp → Exp = = Exp
Exp → IF Exp THEN Exp ELSE Exp
Exp → ID
Exp → INT
Exp → LET ID := Exp IN Exp

$$\frac{}{A + \{ ID \rightarrow T \} \vdash ID : T}$$

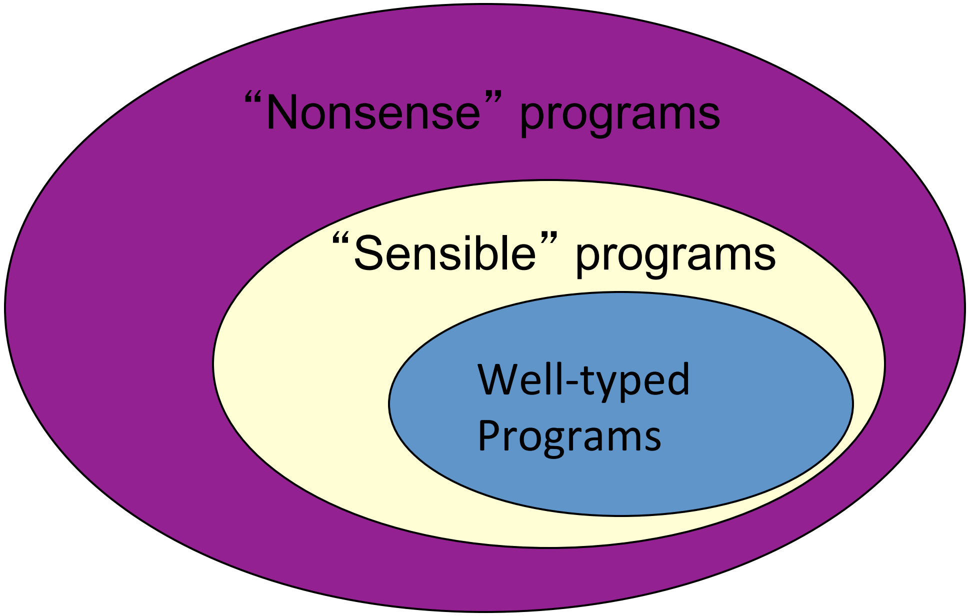
$$\frac{}{A \vdash INT : Int}$$

$$\frac{A \vdash e1 : T1 \qquad A + \{ v \rightarrow T1 \} \vdash e2 : T2}{A \vdash LET\ v := e1\ IN\ e2 : T2}$$

Uses of a type system

- Type checking problem:
  - Given a claim that program “e” and a type “T”
  - determine if “e” has type “T”
    - i.e., if “{} $\vdash$ e : T” is derivable using the rules
    - Tends to be straightforward
- Type inference problem:
  - Given a program “e”
  - determine which type(s) “e” has
  - This is the problem a compiler confronts
    - I.e., compile(e) isn't given the type of “e” and must calculate it itself

Type Systems are typically conservative



For practical reasons (e.g., decidability), type systems typically sacrifice some sensible programs when eliminating nonsense.