CMSC 336
Type Systems for Programming Languages

David MacQueen

Winter, 2003

www.classes.cs.uchicago.edu/classes/archive/2003/winter/CS33600

CS Theory

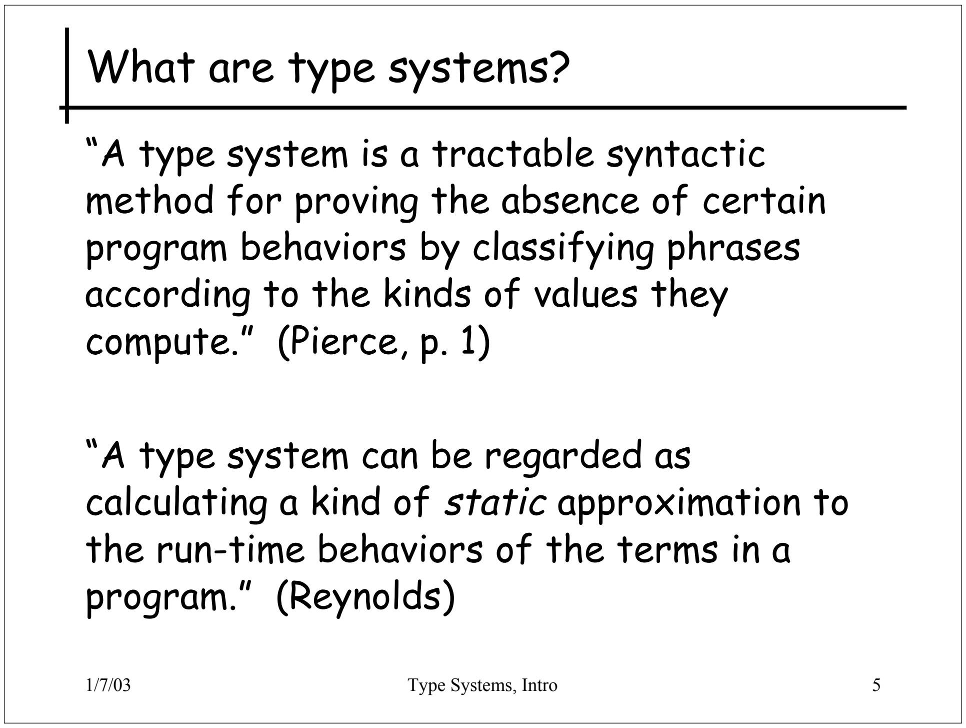
- Computer Science =
  applied mathematics + engineering
- CS theory is the applied mathematics part
- much of this concerns formalisms for computation (e.g. models of computation, programming languages) and their metatheory

Theory: Computability, Complexity

- computability theory
  - models of computation
  - what is computable, what is not
- complexity theory and analysis of algorithms
  - how hard or costly is it to compute something
  - what is *feasibly* computable
  - applications: design of efficient algorithms

Theory of programming

- semantics of computation
  - what do terms in a formalism mean?
- logics of computation (programming logics)
  - specifying computational tasks and verifying that programs satisfy their specifications
- computational logic
  - systems for automatic/interactive deduction
- type theory and type systems
  - which programs “make sense”

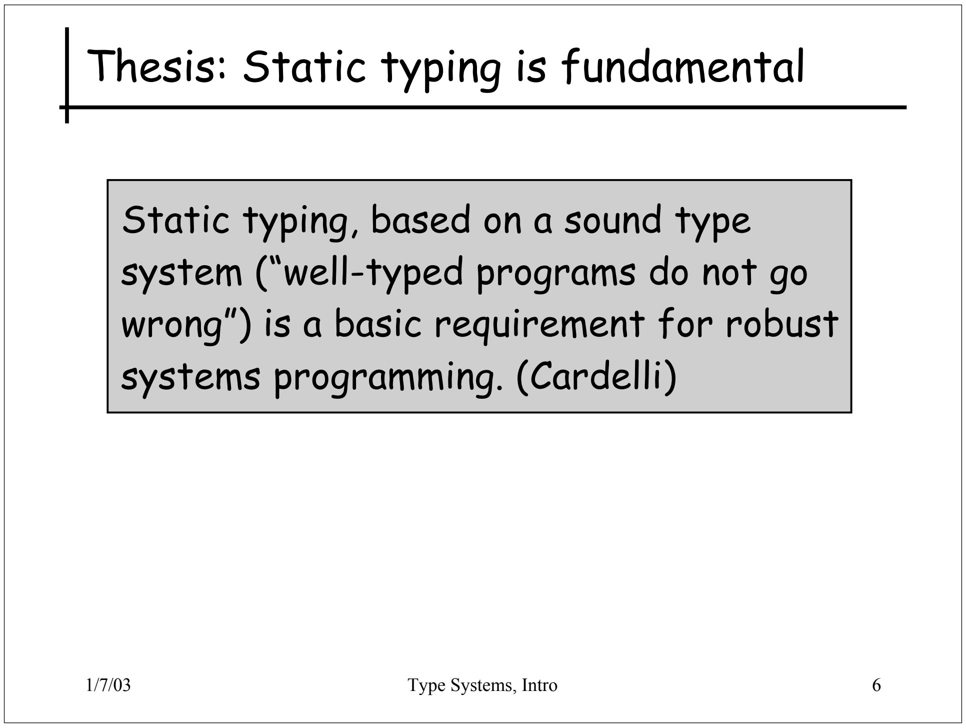


What are type systems?

"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." (Pierce, p. 1)

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

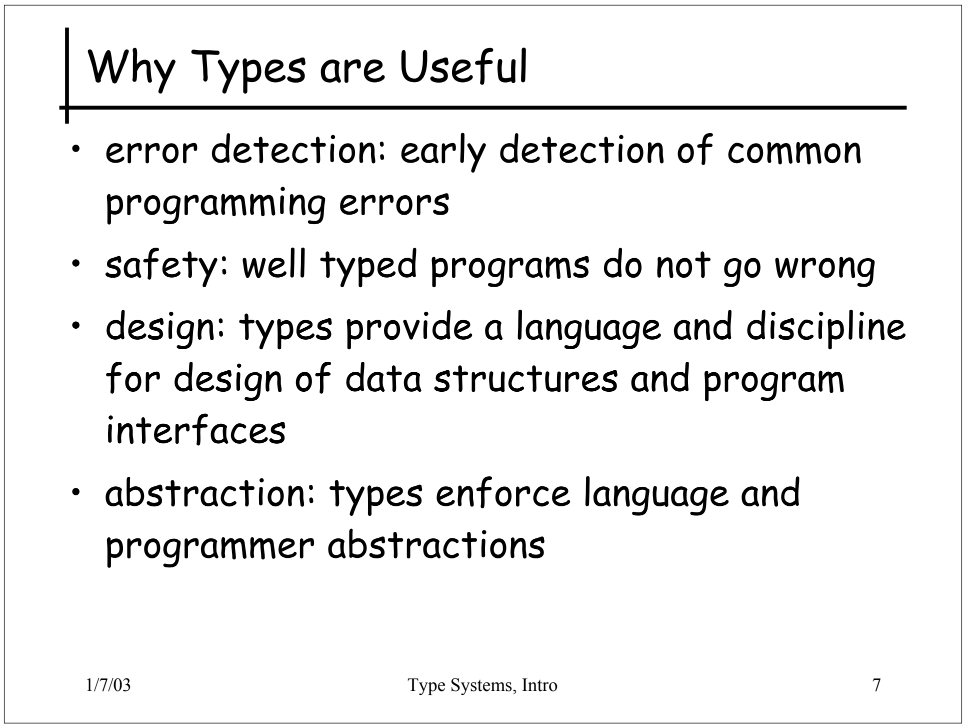
1/7/03 Type Systems, Intro 5



Thesis: Static typing is fundamental

Static typing, based on a sound type system ("well-typed programs do not go wrong") is a basic requirement for robust systems programming. (Cardelli)

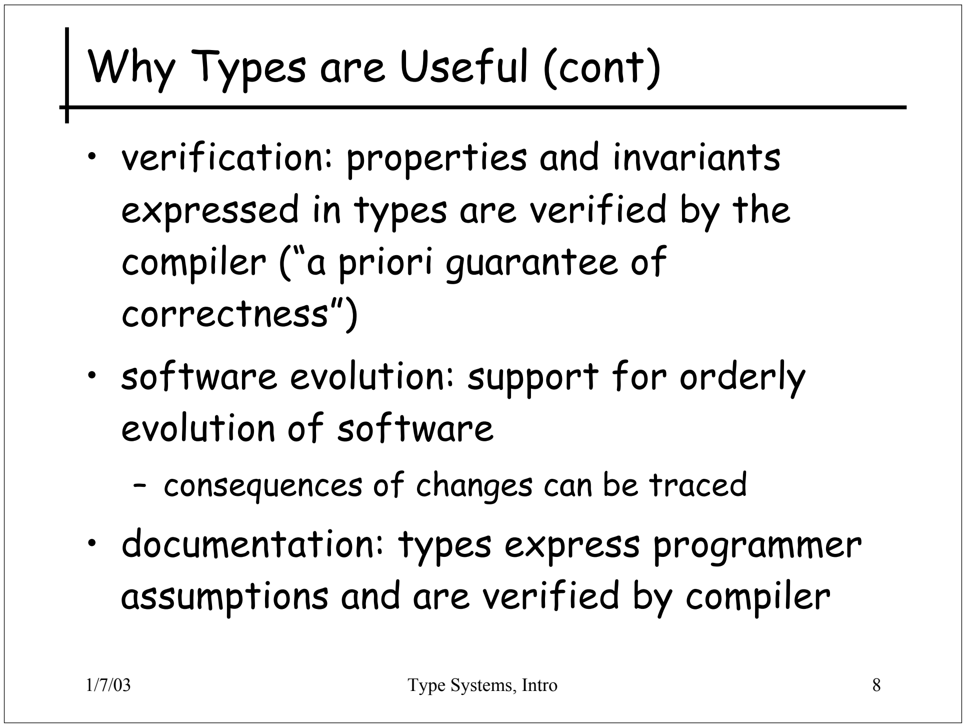
1/7/03 Type Systems, Intro 6



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

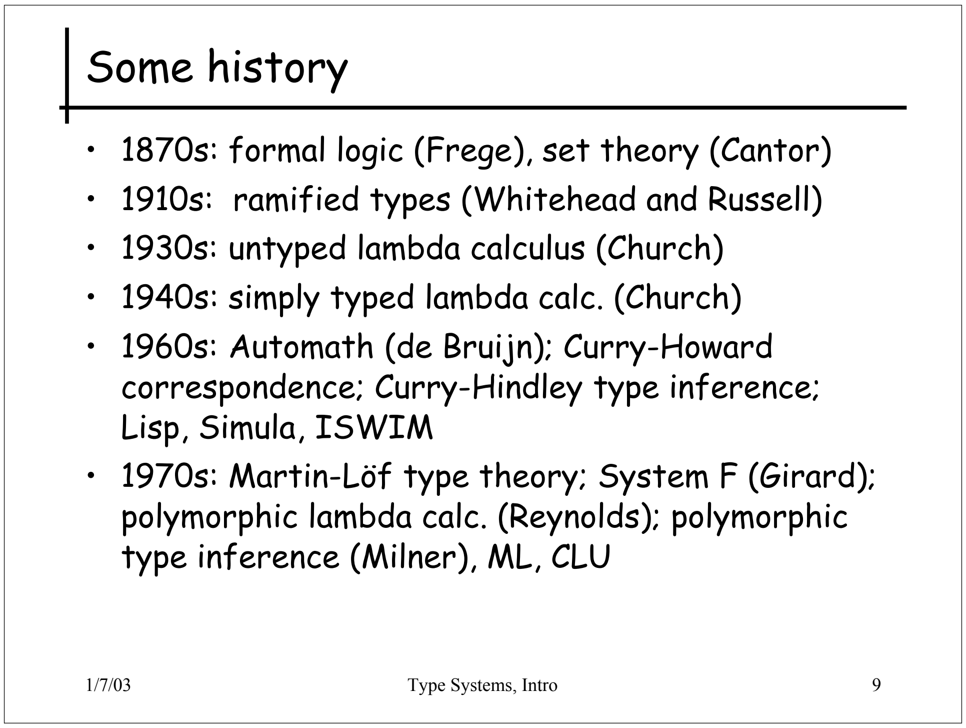
1/7/03 Type Systems, Intro 7



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

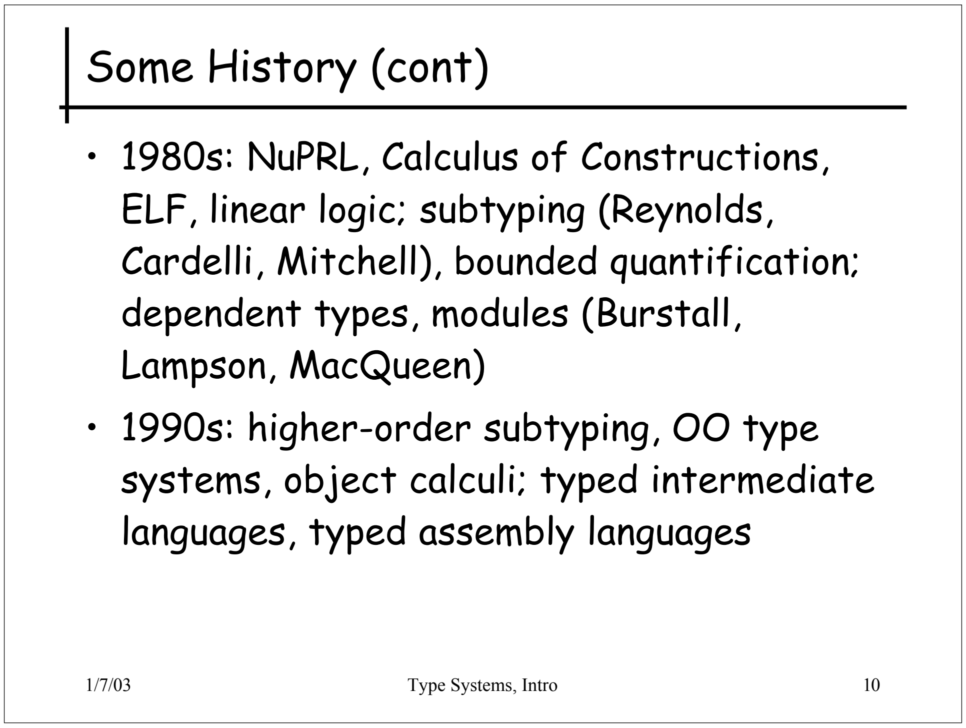
1/7/03 Type Systems, Intro 8



Some history

- 1870s: formal logic (Frege), set theory (Cantor)
- 1910s: ramified types (Whitehead and Russell)
- 1930s: untyped lambda calculus (Church)
- 1940s: simply typed lambda calc. (Church)
- 1960s: Automath (de Bruijn); Curry-Howard correspondence; Curry-Hindley type inference; Lisp, Simula, ISWIM
- 1970s: Martin-Löf type theory; System F (Girard); polymorphic lambda calc. (Reynolds); polymorphic type inference (Milner), ML, CLU

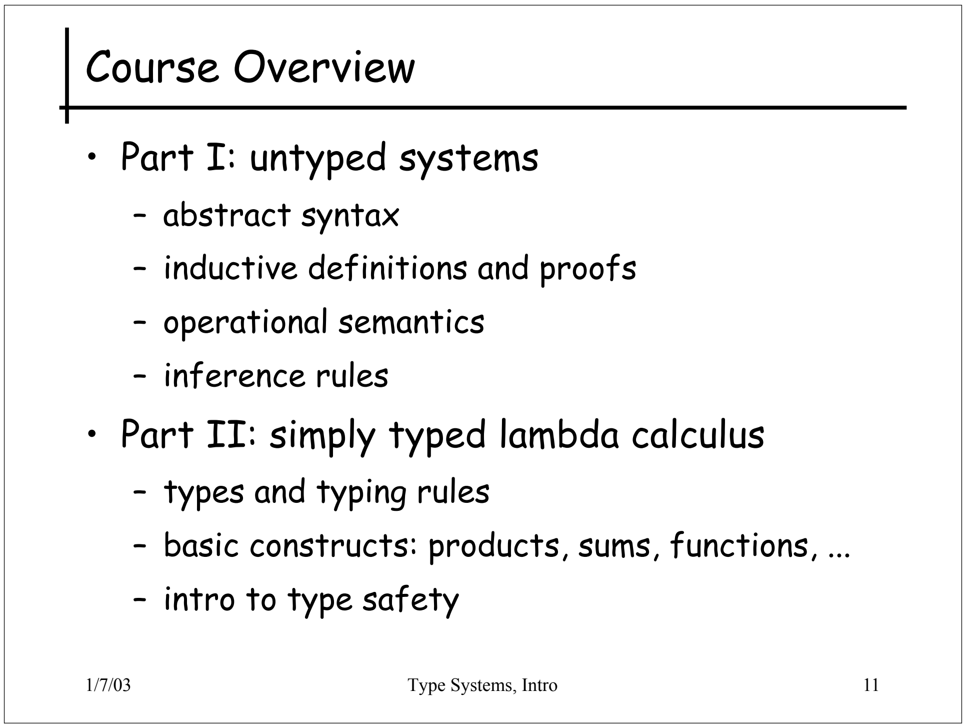
1/7/03 Type Systems, Intro 9



Some History (cont)

- 1980s: NuPRL, Calculus of Constructions, ELF, linear logic; subtyping (Reynolds, Cardelli, Mitchell), bounded quantification; dependent types, modules (Burstall, Lampson, MacQueen)
- 1990s: higher-order subtyping, OO type systems, object calculi; typed intermediate languages, typed assembly languages

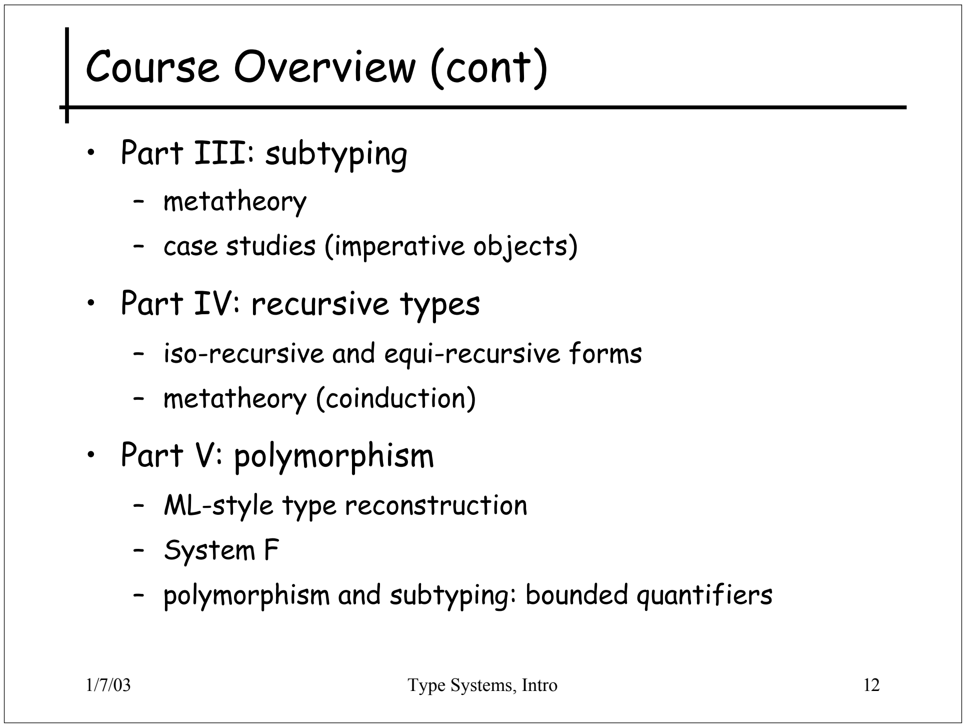
1/7/03 Type Systems, Intro 10



Course Overview

- Part I: untyped systems
  - abstract syntax
  - inductive definitions and proofs
  - operational semantics
  - inference rules
- Part II: simply typed lambda calculus
  - types and typing rules
  - basic constructs: products, sums, functions, ...
  - intro to type safety

1/7/03 Type Systems, Intro 11



Course Overview (cont)

- Part III: subtyping
  - metatheory
  - case studies (imperative objects)
- Part IV: recursive types
  - iso-recursive and equi-recursive forms
  - metatheory (coinduction)
- Part V: polymorphism
  - ML-style type reconstruction
  - System F
  - polymorphism and subtyping: bounded quantifiers

1/7/03 Type Systems, Intro 12

Course Overview (cont)

- Part VI: Type operators
  - higher-order type constructs
  - System F_{ω}
  - subtyping: System $F_{<:}^{\omega}$
  - case study: functional objects

Potential Advanced Topics

- type systems as logics
- *denotational* semantics of programs and types
- module systems
- full-featured object-oriented languages

Required background

The course is self-contained, but the following will be useful:

- "mathematical maturity"
- some familiarity with (naive) set theory, elementary logic, (structural) induction
- some familiarity with a higher-order functional language (e.g. scheme or ML or Haskell)

Implementation

- Several chapters present implementations of type checkers.
- The programming language used in the text is a simple subset of Ocaml. In the course, I will substitute code in a similar subset of Standard ML.
- For documentation/tutorials on Standard ML, see www.smlnj.org