Type Systems: Big Idea

Static vs. Dynamic Typing

- Expressiveness (+ Dynamic)
- Don’t have to worry about types (+ Dynamic)
- Dependent on input (- Dynamic)
- Runtime overhead (- Dynamic)
- Serve as documentation (+ Static)
- Catch errors at compile time (+ Static)
- Used in optimization (+ Static)
Type Systems: Big Idea

• Undecideability forces tradeoff:
  – Dynamic or
  – Approximate or
  – Non-terminating

• Example: array bounds checking
  – Occasional negative consequences: e.g., Heartbleed
Type Systems: Mechanics

- Monomorphic and Polymorphic Types
- Types, Type Constructors, Quantified Types $(\forall \alpha. \tau)$
- Kinds ($\kappa$) classify types:
  - well-formed,
  - types ($*$),
  - type constructors: $\kappa \Rightarrow \kappa$
- Type Environments: type identifiers $\rightarrow$ kinds
- Typing Rules
  - Introduction and Elimination forms
- Type Checking
- Induction and Recursion
Hindley-Milner Type Inference: Big Idea

- Inferred vs Declared Types
  - Advantages of Inference: write fewer types, infer more general types.
  - Advantages of Declarations: better documentations, more general type system.

- Canonical example of static analysis:
  - Proving properties of programs based only on text of program.
  - Useful for compilers and security analysis.
Hindley-Milner Type Inference: Mechanics

- Use fresh type variables to represent unknown types
- Generate constraints that collect clues
- Solve constraints just before introducing quantifiers
- Compromises to preserve decideability:
  - Only generalize lets and top-level declarations
  - Polymorphic functions aren’t first-class
Module Systems a la SML: Big Ideas

- “Programming-in-the-large”
- Separate implementation from interface
- Enforced modularity
  - Swap implementations without breaking client code
Module Systems a la SML: Mechanics

- Signatures describe interfaces
  - types, values, exceptions, substructures
  - include to extend
- Structures provide implementations
- Signature ascription hides structure contents
  \((\text{Heap} :> \text{HEAP})\)
- Functors
  - Functions over structures
  - Executed at compile time
Object-Oriented Programming: Big Ideas

• “Programming-in-the-medium”
• Advantages and Disadvantages
  – Enables code reuse
  – Easy to add new kinds of objects
  – Hard to add new operations
  – Algorithms smeared across many classes
  – Hard to know what code is executing
• Good match for GUI programming
• Smalltalk mantra: Everything is an Object
  – Can redefine basic operations
Object-Oriented Programming: Mechanics

- Classes and objects
- Computation via sending messages
- Double-dispatch
- Inheritance for implementation reuse
- Subtyping ("duck typing") for client code reuse
- Subtyping is not Inheritance
- `self` and `super`
- Blocks to code anonymous functions & continuations
Lambda Calculus: Big Ideas

• Three forms:
  \[ e ::= x \mid \lambda x. e \mid e_1 e_2 \]

• Church-Turing Thesis:
  - All computable functions expressable in lambda calculus
  - booleans, pairs, lists, naturals, recursion, ...
Lambda Calculus: Mechanics

- Bound vs. Free variables
- \( \alpha \)-conversion: Names of bound variables don’t matter.
- \( \beta \)-reduction: Models computation.
- Capture-avoiding substitution (Why important?)
- Recursion via fixed points
- \( Y \) combinator calculates fixed points:
  - \( Y = \lambda f. (\lambda x. f(x x))(\lambda x. f(x x)) \)
Substitution Example

Consider

\[
\text{let } x = 10 \text{ in } \\
(\lambda y.\lambda x. x + y) \; x \; 3
\]

Naive substitution (wrong!):

\[
\text{let } x = 10 \text{ in } \\
(\lambda x. x + x) \; 3
\]

Capture-avoiding substitution (right!):

\[
\text{let } x = 10 \text{ in } \\
(\lambda z. z + x) \; 3
\]

Naive version evaluates to 6, correct version to 13.
Programming Experience

• Recursion and higher-order functions are now second-nature to you.
  – You’ll miss pattern matching and algebraic data types in any language you use that doesn’t have them!
• C for impcore (imperative language)
• Scheme (dynamically typed functional language)
• ML (statically typed functional language)
• uSmalltalk (dynamically typed OO language)
Built substantial pieces of code

- SAT solver using continuations
- Type checker (ML pattern matching!)
- Type inference system (using constraints, reading typing rules)
- Game solver (SML module system)
- BigNums (Power of OO abstractions; resulting challenges)
Where might you go from here?
Haskell

- At the research frontier: Still evolving.
- **Lazy:**
  - Expressions only evaluated when needed.
  - Conflict with side-effects.
  - Solution: Monads (computation abstraction)
- **Type Classes:**
  - Ad hoc polymorphism (aka, overloading)
  - ML: Hard-wire certain operations (+, *)
  - Haskell: User programmable.
Prolog

- Based on logic.
- Performs proof search over inference rules.
- Can leave "blanks" and ask the system to figure out what they must be.
Ruby

• If you liked smalltalk.
Additional Courses

• Compilers
• Special Topics:
  – Domain-specific Languages
  – Probabilistic Programming Languages
  – Advanced Functional Programming
Big-picture questions?
Studying for the Exam

- Exam will be like midterm
- Expect to write some code (SML, uSmalltalk)
- Review homework assignments
- Review recitation materials
- Make sure you understand Big Ideas/Tradeoffs
Other Questions?
Course feedback

In future courses

• What should we keep the same?
• How can we improve?
Congratulations!

• You have learned an amazing amount.
• You have really impressed me.
• Good luck on the exam!

**Thank you!**