CMSC 430
Introduction to Compilers
Spring 2015

Type Systems
What is a Type System?

• A type system is some mechanism for distinguishing good programs from bad
  - Good programs = well typed
  - Bad programs = ill-typed or not typable

• Examples:
  - 0 + 1 // well typed
  - false 0 // ill-typed: can’t apply a boolean
  - 1 + (if true then 0 else false) // ill-typed: can’t add boolean to integer
    - Notice that the type system may be conservative — it may report programs as erroneous if they could run without type errors
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.”

– Benjamin Pierce, *Types and Programming Languages*
The Plan

• Start with lambda calculus (yay!)
• Add types to it
  ▪ Simply-typed lambda calculus
• Prove type soundness
  ▪ So we know what our types mean
  ▪ We’ll learn about structural induction here
• Discuss issues of types in real languages
  ▪ E.g., null, array bounds checks, etc
• Explain type inference
• Add subtyping (for OO) to all of the above
We’ll use lambda calculus as a “core language” to explain type systems

- Has essential features (functions)
- No overlapping constructs
- And none of the cruft
  - Extra features of full language can be defined in terms of the core language (“syntactic sugar”)

We will add features to lambda calculus as we go on
Simply-Typed Lambda Calculus

- $e ::= n \mid x \mid \lambda x:t.e \mid e\;e$
  - Functions include the type of their argument
  - We’ve added integers, so we can have (obvious) type errs
  - We don’t really need this, but it will come in handy

- $t ::= \text{int} \mid t \rightarrow t$
  - $t_1 \rightarrow t_2$ is the type of a function that, given an argument of type $t_1$, returns a result of type $t_2$
    - $t_1$ is the *domain*, and $t_2$ is the *range*
Type Judgments

• Our type system will prove judgments of the form
  - \( A \vdash e : t \)
  - “In type environment \( A \), expression \( e \) has type \( t \)”
Type Environments

• A *type environment* is a map from variables to types (a kind of symbol table)
  - · is the empty type environment
  - A closed term e is *well-typed* if · ⊢ e : t for some t
  - We’ll abbreviate this as ⊢ e : t

• x:t, A is just like A, except x now has type t
  - The type of x in x:t, A is t
  - The type of z≠x in x:t, A in the type of z in A

• When we see a variable in a program, we look in the type environment to find its type
Type Rules

\[
\begin{align*}
A \vdash n : \text{int} \\
A \vdash x : A(x) \\
x \in \text{dom}(A) \\
A \vdash \lambda x : t. e : t \to t' \\
A \vdash e_1 : t \to t' \\
A \vdash e_2 : t \\
A \vdash e_1 \ e_2 : t'
\end{align*}
\]
Example

\[ A = - : \text{int} \rightarrow \text{int} \]

\[
\begin{array}{c}
- \in \text{dom}(A) \\
\hline
A \vdash - : \text{int} \rightarrow \text{int} \quad A \vdash 3 : \text{int}
\end{array}
\]

\[ A \vdash - 3 : \text{int} \]
Another Example

\[ A = + : \text{int} \rightarrow \text{int} \rightarrow \text{int} \]
\[ B = x : \text{int}, A \]

\[ + \in \text{dom}(B) \quad x \in \text{dom}(B) \]
\[ B \vdash + : \text{int} \rightarrow \text{int} \rightarrow \text{int} \]
\[ B \vdash x : \text{int} \]
\[ B \vdash + x : \text{int} \rightarrow \text{int} \]
\[ B \vdash 3 : \text{int} \]
\[ B \vdash + x 3 : \text{int} \]

\[ A \vdash (\lambda x:\text{int}.+ x 3) : \text{int} \rightarrow \text{int} \]
\[ A \vdash 4 : \text{int} \]

\[ A \vdash (\lambda x:\text{int}.+ x 3) 4 : \text{int} \]

We’d usually use infix \( x + 3 \)
An Algorithm for Type Checking

• Our type rules are deterministic
  ▪ For each syntactic form, only one possible rule

• They define a natural type checking algorithm
  ▪ TypeCheck : type env × expression → type

    TypeCheck(A, n) = int
    TypeCheck(A, x) = if x in dom(A) then A(x) else fail
    TypeCheck(A, λx:t.e) = TypeCheck((A, x:t), e)
    TypeCheck(A, e1 e2) =
      let t1 = TypeCheck(A, e1) in
      let t2 = TypeCheck(A, e2) in
      if dom(t1) = t2 then range(t1) else fail
• Here is a small-step, call-by-value semantics

  - If an expression can’t be evaluated any more and is not a value, then it is stuck

\[
(\lambda x. e_1) v_2 \rightarrow e_1[v_2/x]
\]

\[
e_1 \rightarrow e_1'
\]

\[
e_1 e_2 \rightarrow e_1' e_2
\]

\[
e_2 \rightarrow e_2'
\]

\[
v_1 e_2 \rightarrow v_1 e_2'
\]

\[
e ::= v \mid x \mid e \ e
\]

\[
v ::= n \mid \lambda x : t . e \quad \text{values – not evaluated}
\]
Progress

• Suppose $\vdash e : t$. Then either $e$ is a value, or there exists $e'$ such that $e \rightarrow e'$

• Proof by induction on $e$
  ▪ Base cases $n$, $\lambda x.e$ – these are values, so we’re done
  ▪ Base case $x$ – can’t happen (empty type environment)
  ▪ Inductive case $e1 \ e2$ – If $e1$ is not a value, then by induction we can evaluate it, so we’re done, and similarly for $e2$. Otherwise both $e1$ and $e2$ are values. Inspection of the type rules shows that $e1$ must have a function type, and therefore must be a lambda since it’s a value. Therefore we can make progress.
Preservation

- If $\vdash e : t$ and $e \rightarrow e'$ then $\vdash e' : t$
- Proof by induction on $e \rightarrow e'$
  - Induction (easier than the base case!). Expression $e$ must have the form $e_1 e_2$.
  - Assume $\vdash e_1 e_2 : t$ and $e_1 e_2 \rightarrow e'$. Then we have $\vdash e_1 : t'$ $\rightarrow$ $t$ and $\vdash e_2 : t'$.
  - Then there are three cases.
    - If $e_1 \rightarrow e_1'$, then by induction $\vdash e_1 : t'$ $\rightarrow$ $t$, so $e_1' e_2$ has type $t$
    - If reduction inside $e_2$, similar
Preservation, cont’d

• Otherwise \((\lambda x. e) \; v \rightarrow e[v/x]\). Then we have

\[
\frac{x : t' \vdash e : t}{\vdash \lambda x. e : t' \rightarrow t}
\]

-Thus we have
  - \(x : t' \vdash e : t\)
  - \(\vdash v : t'\)

-Then by the substitution lemma (not shown) we have
  - \(\vdash e[v/x] : t\)

-And so we have preservation
Substitution Lemma

• If $A \vdash v : t$ and $x : t$, $A \vdash e : t'$, then $A \vdash e[v/x] : t'$

• Proof: Induction on the structure of $e$

• For lazy semantics, we’d prove
  - If $A \vdash e_1 : t$ and $x : t$, $A \vdash e : t'$, then $A \vdash e[e_1/x] : t'$
Soundness

- So we have
  - Progress: Suppose $\vdash e : t$. Then either $e$ is a value, or there exists $e'$ such that $e \rightarrow e'$
  - Preservation: If $\vdash e : t$ and $e \rightarrow e'$ then $\vdash e' : t$

- Putting these together, we get soundness
  - If $\vdash e : t$ then either there exists a value $v$ such that $e \rightarrow^* v$, or $e$ diverges (doesn’t terminate).

- What does this mean?
  - Evaluation getting stuck is bad, so
  - “Well-typed programs don’t go wrong”
Consequences of Soundness

- Progress—anything that can go wrong “locally” at run time should be forbidden in the type system
  - E.g., can’t “call” an int as if it were a function
  - To check this, identify all places where the semantics get stuck, and cross-reference with type rules

- Preservation—running a program can’t change types
  - E.g., after beta reduction, types still the same
  - To check this, ensure that for each possible way the semantics can take a step, types are preserved

- These problems greatly influence the way type systems are designed
Conditionals

e ::= ... | true | false | if e then e else e

\( A \vdash \text{true} : \text{bool} \)
\( A \vdash \text{false} : \text{bool} \)
\( A \vdash \text{el} : \text{bool} \)
\( A \vdash \text{e2} : t \)
\( A \vdash \text{e3} : t \)
\( A \vdash \text{if el then e2 else e3} : t \)
Conditionals (op sem)

\[ e ::= ... \mid \text{true} \mid \text{false} \mid \text{if } e \text{ then } e \text{ else } e \]

- \[ \text{if true then } e_2 \text{ else } e_3 \rightarrow e_2 \]
- \[ \text{if false then } e_2 \text{ else } e_3 \rightarrow e_3 \]
- \[ e_1 \rightarrow e_1' \]
  - \[ \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \rightarrow \]
  - \[ \text{if } e_1' \text{ then } e_2 \text{ else } e_3 \]

- Notice how need to satisfy progress and preservation influences type system, and interplay between operational semantics and types
Product Types (Tuples)

e ::= ... | (e, e) | fst e | snd e

\[
\begin{align*}
A \vdash e_1 : t & \quad A \vdash e_2 : t' \\
\hline
A \vdash (e_1, e_2) : t \times t' \\
\end{align*}
\]

\[
\begin{align*}
A \vdash e : t \times t' & \\
\hline
A \vdash \text{fst } e : t \\
\end{align*}
\]

\[
\begin{align*}
A \vdash e : t \times t' & \\
\hline
A \vdash \text{snd } e : t' \\
\end{align*}
\]

• Or, maybe, just add functions
  - pair : t → t' → t × t'
  - fst : t × t' → t
  - snd : t × t' → t'
Sum Types (Tagged Unions)

\[ e ::= \ldots \mid \text{inL}_{t_2} \; e \mid \text{inR}_{t_1} \; e \]
\[ \mid (\text{case } e \text{ of } x_1 : t_1 \rightarrow e_1 \mid x_2 : t_2 \rightarrow e_2) \]

\[
\frac{A \vdash e : t_1}{A \vdash \text{inL}_{t_2} \; e : t_1 + t_2}
\]
\[
\frac{A \vdash e : t_2}{A \vdash \text{inR}_{t_1} \; e : t_1 + t_2}
\]

\[
\frac{A \vdash e : t_1 + t_2}{x_1 : t_1, A \vdash e_1 : t \quad x_2 : t_2, A \vdash e_2 : t}
\]

\[
A \vdash (\text{case } e \text{ of } x_1 : t_1 \rightarrow e_1 \mid x_2 : t_2 \rightarrow e_2) : t
\]
Self Application and Types

• Self application is not checkable in our system

\[
\frac{x : ?, A \vdash x : t \rightarrow t'}{x : ?, A \vdash x : t} \quad \frac{x : ?, A \vdash x : t}{A \vdash \lambda x : ? . x \; x : \ldots}
\]

- It would require a type \( t \) such that \( t = t \rightarrow t' \)
  - (We’ll see this next, but so far...)

• The simply-typed lambda calculus is strongly normalizing
  - Every program has a normal form
  - I.e., every program halts!
Recursive Types

- We can type self application if we have a type to represent the solution to equations like $t = t \rightarrow t'$
  - We define the type $\mu \alpha.t$ to be the solution to the (recursive) equation $\alpha = t$
  - Example: $\mu \alpha.\text{int} \rightarrow \alpha$

![Diagram showing recursive type structure]

or

![Another diagram showing recursive type structure]
In the pure lambda calculus, every term is typable with recursive types
  - (Pure = variables, functions, applications only)

Most languages have some kind of “recursive” type
  - E.g., for data structures like lists, tree, etc.

However, usually two recursive types that define the same structure but use a different name are considered different
  - E.g., in C, `struct foo { int x; struct foo *next; }` is different from `struct bar { int x; struct bar *next; }`
Subtyping

• The Liskov Substitution Principle (paraphrased):

Let \( q(x) \) be a property provable about objects \( x \) of type \( T \). If \( S \) is a subtype of \( T \), then \( q(y) \) should be provable for objects \( y \) of type \( S \).

• In other words

If \( S \) is a subtype of \( T \), then an \( S \) can be used anywhere a \( T \) is expected.

• Common used in object-oriented programming
  ▪ Subclasses can be used where superclasses expected
  ▪ This is a kind of \textit{polymorphism}
Kinds of Polymorphism

• Parametric polymorphism
  ▪ Generics in Java, `a types in OCaml

• Another popular form is subtype polymorphism
  ▪ As in OO programming
  ▪ These two can be combined (c.f. Java)

• Some languages also have ad-hoc polymorphism
  ▪ E.g., + operator that works on ints and floats
  ▪ E.g., overloading in Java
Lambda Calc with Subtyping

- $e ::= n \mid f \mid x \mid \lambda x : t . e \mid e \ e$
  - We now have both floating point numbers and integers
  - We want to be able to implicitly use an integer wherever a floating point number is expected
  - Warning: This is a bad design! Don’t do this in real life

- $t ::= \text{int} \mid \text{float} \mid t \rightarrow t$
  - We want int to be a subtype of float
Subtyping

• We’ll write \( t_1 \leq t_2 \) if \( t_1 \) is a subtype of \( t_2 \)
• Define subtyping by more inference rules
• Base case

\[
\text{int} \leq \text{float}
\]

• (notice reverse is not allowed)

• What about function types?

\[
???
\]

\[
\text{t}_1 \rightarrow \text{t}_1' \leq \text{t}_2 \rightarrow \text{t}_2'
\]
Replacing “f x” by “g x”

• Suppose \( f : t_1 \to t_1' \) and \( g : t_2 \to t_2' \)
• When is \( t_1 \to t_1' \leq t_2 \to t_2' \)?

• Return type:
  ▪ We are expecting \( t_1' \) (f’s return type)
  ▪ So we can return \textit{at most} \( t_1' \)
  ▪ So need \( t_1' \leq t_2' \)

• Examples
  ▪ If we’re expecting \texttt{float}, can return \texttt{int} or \texttt{float}
  ▪ If we’re expecting \texttt{int}, can only return \texttt{int}
Replacing “f x” by “g x”

• Suppose $f : t_1 \rightarrow t_1'$ and $g : t_2 \rightarrow t_2'$
• When is $t_1 \rightarrow t_1' \leq t_2 \rightarrow t_2'$?

• Argument type:
  - We are supposed to accept expecting $t_1$ (f’s arg type)
  - So we must accept at least $t_1$
  - So need $t_2 \leq t_1$

• Examples
  - A function that accepts an int can be replaced by one that accepts int, or one that accepts float
  - A function that accepts a float can only be replaced by one that accepts float
Subtyping on Function Types

\[
\begin{align*}
\text{If } & t_2 \leq t_1, t_1' \leq t_2' \\
\text{then } & t_1 \rightarrow t_1' \leq t_2 \rightarrow t_2'
\end{align*}
\]

• We say that arrow is
  - Covariant in the range (subtyping dir the same)
  - Contravariant in the domain (subtyping dir flips)

• Some languages have gotten this wrong
  - Eiffel allows covariant parameter types
Similar Pattern for Pre/Post-conds

- class A { int f(int x) { ... } }
- class B extends A { int f(int x) { ... } }

- A.f — precondition Pre_A, postcondition Post_A
- B.f — precondition Pre_B, postcondition Post_B
- Relationship among \{Pre,Post\}_\{A,B\}?
  - Post_A ⇒ Post_B
  - Pre_B ⇒ Pre_A

- Example:
  - Pre_A = (x > 42), Post_A = (ret > 42)
  - Pre_B = (x > 0), Post_B = (ret > 100)
Type Rules, with Subtyping

\[
\begin{align*}
A \vdash n : \text{int} & \quad & A \vdash f : \text{float} \\
x \in \text{dom}(A) & \quad & x : t, A \vdash e : t' \\
A \vdash x : A(x) & \quad & A \vdash \lambda x : t.e : t \rightarrow t' \\
A \vdash e_1 : t_1 \rightarrow t_1' & \quad & A \vdash e_2 : t_2 \quad t_2 \leq t_1 \\
& \quad & A \vdash e_1 \ e_2 : t_1'
\end{align*}
\]
Soundness

• Progress and preservation still hold
  ▪ Slight tweak: as evaluation proceeds, expression’s type may “decrease” in the subtyping sense
  ▪ Example:
    - (if true then n else f) : float
    - But after taking one step, will have type \text{int} \leq \text{float}

• Proof: exercise for the reader
Subtyping, again

\[
\begin{align*}
A \vdash n : \text{int} & & A \vdash f : \text{float} \\
& & \\
A \vdash x : A(x) & & A \vdash \lambda x : t. e : t \rightarrow t' \\
& & A \vdash e_1 : t_1 \rightarrow t_1' \\
& & A \vdash e_2 : t_2 \\
& & A \vdash e_1 \; e_2 : t_1' \\
& & A \vdash e : t \quad t \leq t' \\
& & A \vdash e : t'
\end{align*}
\]
Subtyping, again (cont’d)

- Rule with subtyping is called *subsumption*
  - Very clearly captures subtyping property

- But system is no longer *syntax driven*
  - Given an expression $e$, there are two rules that apply to $e$ (*“regular” type rule, and subsumption rule*)

- Can prove that the two systems are equivalent
  - Exercise left to the reader
Lambda Calc with Updatable Refs

- \( e ::= \ldots | \text{ref } e | !e | e := e \)
  - ML-style updatable references
    - \text{ref } e — allocate memory and set its contents to \( e \); return pointer
    - \!e — dereference pointer and return contents
    - \( e_1 := e_2 \) — update contents pointed to by \( e_1 \) with \( e_2 \)

- \( t ::= \ldots | t \text{ ref} \)
  - A \( t \text{ ref} \) is a pointer to contents of type \( t \)
Type Rules for Refs

\[
\begin{align*}
\frac{A \vdash e : t}{A \vdash \text{ref} e : t \text{ ref}} & \quad \frac{A \vdash e : t \text{ ref}}{A \vdash \text{!} e : t} \\
\frac{A \vdash e_1 : t_1 \text{ ref} \quad A \vdash e_2 : t_2 \quad t_2 \leq t_1}{A \vdash e_1 := e_2 : t_1} & \quad \frac{A \vdash e_1 : t_1 \text{ ref} \quad A \vdash e_2 : t_2}{A \vdash e_1 := e_2 : t_1}
\end{align*}
\]
Subtyping Refs

• The wrong rule for subtyping refs is

\[ t_1 \leq t_2 \]

\[ \frac{t_1 \leq t_2}{t_1 \text{ ref} \leq t_2 \text{ ref}} \]

• Counterexample

\[
\begin{align*}
  & \text{let } x = \text{ref 3 in} \quad (* x : \text{int ref} *) \\
  & \text{let } y = x \text{ in} \quad (* y : \text{float ref} *) \\
  & y := 3.14 \quad (* \text{oops! } !x \text{ is now a float} *)
\end{align*}
\]
Aliasing

• We have multiple names for the same memory location
  ▪ But they have different types
  ▪ This we can write into the same memory at different types
Solution #1: Java’s Approach

• Java uses this subtyping rule
  - If $S$ is a subclass of $T$, then $S[]$ is a subclass of $T[]$

• Counterexample:
  - Foo[] $a = \text{new Foo}[5]$;
  - Object[] $b = a$;
  - $b[0] = \text{new Object}();$ // forbidden at runtime
  - $a[0].\text{foo}();$ // ...so this can’t happen
Solution #2: Purely Static

• Reason from rules for functions
  - A reference is like an object with two methods:
    • get : unit \( \rightarrow \) t
    • set : t \( \rightarrow \) unit
  - Notice that t occurs both co- and contravariantly
  - Thus it is non-variant

• The right rule:

\[
\begin{align*}
\text{if } t_1 \leq t_2 & \quad \text{and} \quad t_2 \leq t_1 \\
\text{then } t_1 \text{ ref} & \leq t_2 \text{ ref}
\end{align*}
\]

or

\[
\begin{align*}
\text{if } t_1 = t_2 \\
\text{then } t_1 \text{ ref} & \leq t_2 \text{ ref}
\end{align*}
\]
Type Inference

• Let’s consider the simply typed lambda calculus with integers
  - \( e ::= n \mid x \mid \lambda x : t. e \mid e\ e \)

• *Type inference*: Given a bare term (with no type annotations), can we reconstruct a valid typing for it, or show that it has no valid typing?
Type Language

- Problem: Consider the rule for functions

\[ x : t, A \vdash e : t' \]

\[ \frac{}{A \vdash \lambda x : t. e : t \to t'} \]

- Without type annotations, where do we get \( t \)?
  - We’ll use type variables to stand for as-yet-unknown types
    - \( t ::= \alpha \mid \text{int} \mid t \to t \)
  - We’ll generate equality constraints \( t = t \) among the types and type variables
    - And then we’ll solve the constraints to compute a typing
Type Inference Rules

\[
\begin{align*}
A \vdash n : \text{int} & \quad x \in \text{dom}(A) \\
x : \alpha, A \vdash e : t' & \quad \alpha \text{ fresh} \\
A \vdash \lambda x. e : \alpha \rightarrow t' & \quad \text{"Generated" constraint} \\
A \vdash e_1 : t_1 & \quad A \vdash e_2 : t_2 \\
t_1 = t_2 \rightarrow \beta & \quad \beta \text{ fresh} \\
A \vdash e_1 e_2 : \beta
\end{align*}
\]
Example

\[
\begin{align*}
\Gamma : \alpha, \quad A \vdash x : \alpha \\
\hline
A \vdash (\lambda x.x) : \alpha \to \alpha \quad A \vdash 3 : \text{int} \quad \alpha \to \alpha = \text{int} \to \beta \\
\hline
A \vdash (\lambda x.x) \ 3 : \beta
\end{align*}
\]

- We collect all constraints appearing in the derivation into some set \( C \) to be solved.
- Here, \( C \) contains just \( \alpha \to \alpha = \text{int} \to \beta \).
  - Solution: \( \alpha = \text{int} = \beta \).
- Thus this program is typable, and we can derive a typing by replacing \( \alpha \) and \( \beta \) by \( \text{int} \) in the proof tree.
Solving Equality Constraints

- We can solve the equality constraints using the following rewrite rules, which reduce a larger set of constraints to a smaller set:
  - $C \cup \{\text{int}=\text{int}\} \Rightarrow C$
  - $C \cup \{\alpha=t\} \Rightarrow C[t\backslash\alpha]$
  - $C \cup \{t=\alpha\} \Rightarrow C[t\backslash\alpha]$
  - $C \cup \{t_1 \rightarrow t_2 \rightarrow t_2' \} \Rightarrow C \cup \{t_1=t_1'\} \cup \{t_2=t_2'\}$
  - $C \cup \{\text{int}=t_1 \rightarrow t_2\} \Rightarrow \text{unsatisfiable}$
  - $C \cup \{t_1 \rightarrow t_2=\text{int}\} \Rightarrow \text{unsatisfiable}$
Termination

• We can prove that the constraint solving algorithm terminates.

• For each rewriting rule, either
  ▪ We reduce the size of the constraint set
  ▪ We reduce the number of “arrow” constructors in the constraint set

• As a result, the constraint always gets “smaller” and eventually becomes empty
  ▪ A similar argument is made for strong normalization in the simply-typed lambda calculus
Occurs Check

- We don’t have recursive types, so we shouldn’t infer them

- So in the operation \( C[t\alpha] \), require that \( \alpha \notin FV(t) \)
  - (Except if \( t = a \), in which case there’s no recursion in the types, so unification should succeed)

- In practice, it may better to allow \( \alpha \in FV(t) \) and do the occurs check at the end
  - But that can be awkward to implement
Unifying a Variable and a Type

• Computing $C[t\alpha]$ by substitution is inefficient

• Instead, use a union-find data structure to represent equal types
  ▪ The terms are in a union-find forest
  ▪ When a variable and a term are equated, we union them so they have the same ECR (equivalence class representative)
    - Want the ECR to be the concrete type with which variables have been unified, if one exists. Can read off solution by reading the ECR of each set.
Example

\[ \alpha = \text{int} \rightarrow \beta \]
\[ \gamma = \text{int} \rightarrow \text{int} \]
\[ \alpha = \gamma \]
Unification

• The process of finding a solution to a set of equality constraints is called *unification*
  - Original algorithm due to Robinson
    - But his algorithm was inefficient
  - Often written out in different form
    - See Algorithm W
  - Constraints usually solved on-line
    - As type inference rules applied
The algorithm we’ve given finds the **most general type** of a term

- Any other valid type is “more specific,” e.g.,
  - \( \lambda x.x : \text{int} \rightarrow \text{int} \)

- Formally, any other valid type can be gotten from the most general type by applying a substitution to the type variables

This is still a **monomorphic** type system

- \( \alpha \) stands for “some particular type, but it doesn’t matter exactly which type it is”
Benefits of Type Inference

- Handles higher-order functions
- Handles data structures smoothly
- Works in infinite domains
  - Set of types is unlimited
- No forward/backward distinction
  - (Compare to data flow analysis, next)
Drawbacks to Type Inference

- Flow-insensitive
  - Types are the same at all program points
  - May produce coarse results
  - Type inference failure can be hard to understand

- Polymorphism may not scale
  - Exponential in worst case
  - Seems fine in practice (witness ML)