COMP 150-AVS
Fall 2018

Types and Type Inference
The Need for a Type System

- Consider the (untyped) lambda calculus
  - false = $\lambda x.\lambda y.x$
  - 0 (Scott) = $\lambda x.\lambda y.x$

- Everything is encoded as a function
  - So we can easily misuse combinators
    - false 0 if 0 then ... etc...
  - This is no better than assembly language!
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
  - $1 + (\text{if true then } 0 \text{ else false})$  // ill-typed: can’t add boolean to integer
“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*
Simply-Typed Lambda Calculus

- $e ::= n \mid x \mid \lambda x:t.e \mid e\ e$
  - Functions include the type of their argument
  - 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 a 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)
  - $\emptyset$ is the empty type environment
    - A closed term $e$ is well-typed if $\emptyset \vdash e : t$ for some $t$
    - We’ll abbreviate this as $\vdash e : t$
  - $A, x : t$ is just like $A$, except $x$ now has type $t$
    - The type of $x$ in $A, x : t$ is $t$
    - The type of $z \neq x$ in $A, x : t$ 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

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

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

\[ \vdash- : \text{int}\rightarrow\text{int} \quad \vdash3 : \text{int} \]

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

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

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

\[ B \vdash + x : \text{int} \rightarrow \text{int} \]
\[ B \vdash 3 : \text{int} \]

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

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

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
  ▪ \texttt{TypeCheck : type env} × \texttt{expression} → \texttt{type}

\begin{align*}
\text{TypeCheck}(A, n) &= \text{int} \\
\text{TypeCheck}(A, x) &= \text{if } x \text{ in } \text{dom}(A) \text{ then } A(x) \text{ else fail} \\
\text{TypeCheck}(A, \lambda x:t.e) &= t \rightarrow (\text{TypeCheck}((A, x:t), e)) \\
\text{TypeCheck}(A, e_1 \ e_2) &= \\
& \text{let } t_1 = \text{TypeCheck}(A, e_1) \text{ in} \\
& \text{let } t_2 = \text{TypeCheck}(A, e_2) \text{ in} \\
& \text{if } \text{dom}(t_1) = t_2 \text{ then } \text{range}(t_1) \text{ else fail}
\end{align*}
Semantics

• 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

\[
\begin{align*}
(\lambda x : t . e_1) \, v_2 & \to e_1 [x \mapsto v_2] \\
\text{if } e_1 \to e_1' & \Rightarrow e_1 \, e_2 \to e_1' \, e_2 \\
\text{if } e_2 \to e_2' & \Rightarrow \nu \, e_2 \to \nu \, e_2'
\end{align*}
\]

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

\[v ::= n \mid \lambda x : t . e\]

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 \( e_1 \, e_2 \) – If \( e_1 \) is not a value, then by induction we can evaluate it, so we’re done, and similarly for \( e_2 \). Otherwise both \( e_1 \) and \( e_2 \) are values. Inspection of the type rules shows that \( e_1 \) 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 \((\lambda x : t'. e) \, v \rightarrow e[x \mapsto v]\). Then we have

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

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

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

- And so we have preservation
Substitution Lemma

- If $A \vdash v : t$ and $A, x:t \vdash e : t'$, then $A \vdash e[x\rightarrow v] : t'$
- Proof: Induction on the structure of $e$
- For lazy semantics, we’d prove
  - If $A \vdash e_1 : t$ and $A, x:t \vdash e : t'$, then $A \vdash e[x\rightarrow e_1] : t'$
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”
Parametric Polymorphism

• Observation: \( \lambda x.x \) returns its argument exactly and places no constraints on the type of \( x \)
  - The identity function works for any argument type

• We can express this with universal quantification:
  - \( \lambda x.x : \forall \alpha. \alpha \rightarrow \alpha \)
  - For any type \( \alpha \), the identity function has type \( \alpha \rightarrow \alpha \)
  - This is also known as parametric polymorphism
System F: annotated polymorphism

Let’s extend our system as follows:

- \( t ::= \alpha \mid \text{int} \mid t \rightarrow t \mid \forall \alpha.t \)

- \( e ::= n \mid x \mid \lambda x.e \mid e \ e \mid \Lambda \alpha.e \mid e [t] \)

That is, we add polymorphic types, and we add explicit type abstraction (generalization) …

- Annotated code locations at which a value of polymorphic type is created

… and type application (instantiation)

- Explicitly annotated code locations at which a value of polymorphic type is used

This system due to Girard, concurrently Reynolds
Defining Polymorphic Functions

- Polymorphic functions map types to terms
  - Normal functions map terms to terms

Examples

- $\text{\(\Lambda\alpha.\lambda x:\alpha. x : \forall \alpha. \alpha \rightarrow \alpha\)}$
- $\text{\(\Lambda\alpha.\Lambda\beta.\lambda x:\alpha.\lambda y:\beta. x : \forall \alpha. \forall \beta. \alpha \rightarrow \beta \rightarrow \alpha\)}$
- $\text{\(\Lambda\alpha.\Lambda\beta.\lambda x:\alpha.\lambda y:\beta. y : \forall \alpha. \forall \beta. \alpha \rightarrow \beta \rightarrow \beta\)}$
Instantiation

• When we use a parametric polymorphic type, we apply (or instantiate) it with a particular type
  ▪ In System F this is done by hand:
    ▪ $(\Lambda \alpha. \lambda x: \alpha.x)[t1] : t1 \rightarrow t1$
    ▪ $(\Lambda \alpha. \lambda x: \alpha.x)[t2] : t2 \rightarrow t2$

• This is where the term parametric comes from
  ▪ The type $\forall \alpha. \alpha \rightarrow \alpha$ is a “function” in the domain of types, and it is passed a parameter at instantiation time
Type Rules

- Notice that there are no constructs for manipulating values of polymorphic type
  - This justifies instantiation with any type—that’s what the forall means!

- Note also that we are adding \( \alpha \) to \( A \); we could (should?) use this to ensure types are well-formed
Small-step Semantics Rules

\[ (\Lambda \alpha . e)[t] \rightarrow e[\alpha \mapsto t] \quad (\text{type-app}) \]

\[ e \rightarrow e' \quad (\text{tapp-cong}) \]

\[ e[t] \rightarrow e'[t] \]

- We have to extend substitution to include types; that’s up next … !
Free Variables, Again

• We’re going to need to perform substitutions on quantified types
  ▪ So just like with lambda calculus, we need to worry about free variables and capture-free substitution

• Define the free variables of a type
  ▪ $\text{FV}(\alpha) = \{\alpha\}$
  ▪ $\text{FV}(c) = \emptyset$
  ▪ $\text{FV}(t \rightarrow t') = \text{FV}(t) \cup \text{FV}(t')$
  ▪ $\text{FV}(\forall \alpha.t) = \text{FV}(t) - \{\alpha\}$
Substitution, Again

• Define $t[\alpha \mapsto u]$ as
  - $\alpha[\alpha \mapsto u] = u$
  - $\beta[\alpha \mapsto u] = \beta$ where $\beta \neq \alpha$
  - $(t \rightarrow t')[\alpha \mapsto u] = t[\alpha \mapsto u] \rightarrow t'[\alpha \mapsto u]$
  - $(\forall \beta. t)[\alpha \mapsto u] = \forall \beta.(t[\alpha \mapsto u])$ where $\beta \neq \alpha$ and $\beta \notin \text{FV}(u)$

• Define $e[\alpha \mapsto u]$ as
  - $(\lambda x: t. e)[\alpha \mapsto u] = \lambda x: t[\alpha \mapsto u]. e[\alpha \mapsto u]$
  - $(\Lambda \beta. e)[\alpha \mapsto u] = \Lambda \beta. e[\alpha \mapsto u]$ where $\beta \neq \alpha$ and $\beta \notin \text{FV}(u)$
  - $(e_1 e_2)[\alpha \mapsto u] = e_1[\alpha \mapsto u] e_2[\alpha \mapsto u]$
  - $x[\alpha \mapsto u] = x$ and $n[\alpha \mapsto u] = n$
Type Inference

• Let’s reconsider the simply typed lambda calculus with integers

  ▪ $e ::= n \mid x \mid \lambda x : t.e \mid e\ e$

  ▪ (No parametric polymorphism)

• 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?
• Problem: Consider the rule for functions

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

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

• 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

\[
\begin{align*}
A \vdash n : \text{int} \\
x \in \text{dom}(A) \quad A \vdash x : A(x) \\
A, x : \alpha \vdash e : t' \quad \alpha \text{ fresh} \\
A \vdash \lambda x. e : \alpha \rightarrow t' \\
\end{align*}
\]

\[
\begin{align*}
x \in \text{dom}(A) \quad A \vdash x : A(x) \\
A \vdash e_1 : t_1 \quad A \vdash e_2 : t_2 \\
t_1 = t_2 \rightarrow \beta \quad \beta \text{ fresh} \\
A \vdash e_1 \ e_2 : \beta \\
\end{align*}
\]

“Generated” constraint
We collect all constraints appearing in the derivation into some set $C$ to be solved.

Here, $C$ contains just $\alpha \rightarrow \alpha = \text{int} \rightarrow \beta$

- 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.

Example

\[
\begin{align*}
A, x: \alpha & \vdash x: \alpha \\
\hline
A & \vdash (\lambda x.x) : \alpha \rightarrow \alpha \quad A & \vdash 3 : \text{int} \quad \alpha \rightarrow \alpha = \text{int} \rightarrow \beta \\
A & \vdash (\lambda x.x) 3 : \beta
\end{align*}
\]
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
  - \( C \cup \{\text{int}=\text{int}\} \Rightarrow C \)
  - \( C \cup \{\alpha=t\} \Rightarrow C[\alpha\mapsto t] \)
  - \( C \cup \{t=\alpha\} \Rightarrow C[\alpha\mapsto t] \)
  - \( C \cup \{t_1\rightarrow t_2=t_1'\rightarrow t_2'\} \Rightarrow C \cup \{t_1=t_1'\} \cup \{t_2=t_2'\} \)
  - \( C \cup \{\text{int}=t_1 \rightarrow t_2\} \Rightarrow \text{unsatisfiable} \)
  - \( C \cup \{t_1 \rightarrow t_2=\text{int}\} \Rightarrow \text{unsatisfiable} \)
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 \( C[\alpha \mapsto t] \), require that \( \alpha \notin \text{FV}(t) \)
  
    ▪ (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 \( \alpha \in \text{FV}(t) \) and do the occurs check at the end
  
    ▪ But that can be awkward to implement
Unifying a Variable and a Type

• Computing $C[\alpha \mapsto t]$ 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

$\alpha = \text{int} \rightarrow \beta$

$\gamma = \text{int} \rightarrow \text{int}$

$\alpha = \gamma$
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
Discussion

• 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”
Inference for Polymorphism

• We would like to have the power of System F, and the ease of use of type inference
  ▪ In short: given an untyped lambda calculus term, can we discover the annotations necessary for typing the term in System F, if such a typing is possible?
  ▪ Unfortunately, no. This problem has been shown to be undecidable.

• Can we at least perform some type inference for parametric polymorphism?
  ▪ Yes. A sweet spot was found by Hindley and Milner
  ▪ But first, let’s consider the general case …
Attempting Type Inference

• Let’s extend simply-typed calculus as follows:
  ▪ \( t ::= \alpha \mid \text{int} \mid t \rightarrow t \mid \forall \alpha.t \)
  ▪ \( e ::= n \mid x \mid \lambda x.e \mid e e \)

• Type inference will automatically infer where to generalize a term, to introduce polymorphic types, and where to instantiate them.
Instantiation

\[ A \vdash e : \forall \alpha . t \]

\[ \frac{}{A \vdash e : t[\alpha \mapsto t']} \]

- This rule is exactly the same as System F, but we just “magically” pick which \( t' \) to instantiate with
  - You’re surely wondering about algorithmics. We’ll get to that …
Question: When is it safe to generalize (quantify) a type variable $\alpha$ in the type of expression $e$?

Answer: Whenever we can redo the typing proof for $e$, choosing $\alpha$ to be anything we want, and still have a valid typing proof.
Examples

- The choice of the type of $x$ is purely local to type checking $\lambda x.x$
  - There is no interaction with the outside environment
  - Thus we can generalize the type of $x$
Examples (cont’d)

\[ A, \, x: \text{int} \vdash x : \text{int} \]
\[ \quad \Rightarrow \quad \]
\[ A \vdash \lambda x. x + 3 : \text{int} \to \text{int} \]

- The function restricts the type of \( x \), so we cannot introduce a type variable
  - Thus we cannot generalize the type of \( x \)
  - We can only generalize when the function doesn’t “look at” its parameter
The choice of the type of $x$ depends on the type environment

- In the first derivation, $x$ and $y$ have the same type; if we generalize the type of $x$, they could have different types
- Thus we cannot generalize the type of $x$
Generalization Rule

\[ A \vdash e : t \quad \alpha \notin \text{FV}(A) \]

\[ \frac{}{A \vdash e : \forall \alpha . t} \]

• We can generalize any type variable that is unconstrained by the environment
  ▪ Warning: This won’t quite work with refs
Another Justification

• Suppose we have
  - \( A \vdash e : t \) and \( \alpha \not\in FV(A) \)

• Then let \( u \) be any type. By induction, can show
  - \( A[\alpha \mapsto u] \vdash e : t[\alpha \mapsto u] \)
  - But then since \( \alpha \not\in FV(A) \), that’s equivalent to
  - \( A \vdash e : t[\alpha \mapsto u] \)
Polymorphic Type Inference

• We’d like to extend our algorithm to polymorphic type inference
  ▪ Performance generalization and instantiation automatically (and deterministically)

• Major problem: Our system for polymorphism is too expressive
Hindley-Milner Polymorphism

• Restrict polymorphism to only the “top level”
  ■ Introduce polymorphism at `let`
  ■ Fully instantiate at use of a polymorphic type

• Here is our new language
  ■ `e ::= n | x | λx.e | e e | let x = e in e`
  ■ `t ::= α | int | t → t`
  ■ `s ::= t | ∀α.s`
  - These are type schemes
  ■ `A ::= ∅ | A, x:s`

■ Notice that, according to the prior instantiation rule, we won’t instantiate `α` with a scheme `s`, only a type `t`
Old Type Inference Rules

\[ A \vdash n : \text{int} \]

\[ A, x: \alpha \vdash e : t' \quad \alpha \text{ fresh} \]
\[ \frac{}{A \vdash \lambda x.e : \alpha \rightarrow t'} \]

\[ A \vdash e_1 : t_1 \quad A \vdash e_2 : t_2 \]
\[ t_1 = t_2 \rightarrow \beta \quad \beta \text{ fresh} \]
\[ \frac{}{A \vdash e_1 e_2 : \beta} \]
New Type Inference Rules

- At `let`, generalize over all possible variables

\[
A \vdash e_1 : t_1 \quad A, x : \forall \tilde{\alpha}. t_1 \vdash e_2 : t_2 \quad \tilde{\alpha} = \text{FV}(t_1) - \text{FV}(A)
\]

\[
A \vdash \text{let } x = e_1 \text{ in } e_2 : t_2
\]

- At variable uses, instantiate to all fresh types

\[
A(x) = \forall \tilde{\alpha}. t \quad \tilde{\beta} \text{ fresh}
\]

\[
A \vdash x : t[\tilde{\alpha} \mapsto \tilde{\beta}]
\]

- Here the $\tilde{\alpha}$ denotes a list of type variables
Algorithm W

- A type inference algorithm that explicitly solves the equality constraints on-line

- Instead of implicit global substitution (like we used before), threads the substitution through the inference

- In practice, use previous algorithm, plus generalize at let and instantiate at variable uses.
  - Solve for the type of e1, generalize it, then instantiate its solution when doing inference on e2
Example

• Parametric polymorphic type inference

  let x = \x.x in  // x : \forall \alpha. \alpha \to \alpha

  x 3;            // x : \beta \to \beta,  \beta = \text{int}

  x (\lambda y.y)  // x : \gamma \to \gamma,  \gamma = \delta \to \delta

• This would be untypable in a monomorphic type system
Kinds of Polymorphism

- We’ve just seen parametric polymorphism
  - System F and Hindley-Milner style polymorphism
- Another popular form is subtype polymorphism
  - As in OO programming
  - These two can be combined (e.g., Java Generics)
- Some languages also have *ad-hoc polymorphism*
  - E.g., + operator that works on ints and floats
  - E.g., overloading in Java
Polymorphism and References

• Suppose we want polymorphism in our imperative language
  
  - \( e ::= x \mid n \mid \lambda x.e \mid e\ e \mid \text{ref } e \mid !e \mid e := e \)
  
  - \( s ::= t \mid \forall \alpha.s \)
  
  - \( t ::= \alpha \mid \text{int } \mid t \rightarrow t \mid \text{ref } t \)
  
• What if we try our standard rule?

\[
\begin{align*}
A \vdash e_1 : t_1 \\
A, x: \forall \tilde{\alpha}.t_1 \vdash e_2 : t_2 \\
\tilde{\alpha} = \text{FV}(t_1) - \text{FV}(A)
\end{align*}
\]

\[
A \vdash \text{let } x = e_1 \text{ in } e_2 : t_2
\]
Naive Generalization is Unsound

- Example (due to Tofte)

```latex
\begin{verbatim}
let r = ref (λx.x) in  // r : ∀α.ref (α→α)
    r := λx.x+1;       // checks; use r at ref (int → int)
    (!r) true          // oops! checks; use r at ref(bool → bool)
\end{verbatim}
```

- $\alpha$ should not be generalized, because later uses of $r$ may place constraints on it

- Nobody realized there was a problem for a long time
Solution: The Value Restriction

• Only allow *values* to be generalized
  - \( v ::= x \mid n \mid \lambda x.e \)
  - \( e ::= v \mid e \; e \mid \text{ref } e \mid !e \mid e := e \)

\[
A \vdash v : t_1 \quad A,x: \forall \alpha . t \vdash e_2 : t_2 \quad \alpha = \text{FV}(t) - \text{FV}(A)
\]

\[
A \vdash \text{let } x = v \text{ in } e_2 : t_2
\]

• Intuition: Values cannot later be updated
• This solution due to Wright and Felleisen
  - Tofte found a much more complicated solution
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
- Polymorphism provides context-sensitivity
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

• Polymorphic type inference may not scale
  - Exponential in worst case
  - Seems fine in practice (witness ML)