Type Systems

CMSC 631 – Program Analysis and Understanding
Fall 2006

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

The Need for a Type System

• Recall for homework 2 that not all programs accepted by the grammar of the language are sensible:
  • case 0 of nil => c1 or x,y => c2
  • The case statement expects a list, but we have given it a numeral. There is no rule to evaluate such a program, so we’re “stuck.”
  • It would be great to rule out such non-sensical programs in advance, so we can a program can never reach a “stuck state.”

A Definition of 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.”
  – Benjamin Pierce, Types and Programming Languages

Simply-Typed Lambda Calculus

• e ::= n | x | λx.t.e | e e
  • Functions include the type t of their argument
  • We don’t really need this, but it will come in handy

• t ::= int | t → t
  • t1 → t2 is the type of a function that, given an argument of type t1, returns a result of type t2
  – t1 is the domain, and t2 is the range

Type Judgments

• Our type system will prove judgments of the form
  • A ⊢ e : t
  • “In type environment A, expression e has type t”
Type Environments

- A type environment (a.k.a. context) is a map from variables to types (a kind of symbol table)
  - $\emptyset$ is the empty type environment
  - A closed term is well-typed if $\emptyset \vdash e : t$ for some $t$
  - We'll abbreviate this as $\emptyset : e : t$
- $A, x : t$ 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

- $\emptyset \vdash n : \text{int}$
- $x \in \text{dom}(A)$
  - $A \vdash x : A(x)$
- $A, x : t$ $\vdash e : t'$
  - $A \vdash \lambda x : e : t' \rightarrow t''$
  - $A \vdash e_1 : t \rightarrow t''$
  - $A \vdash e_2 : t' \rightarrow t''$

Example

$A = \lambda x : \text{int}.3$

$A \vdash \lambda x : \text{int}.3 : \text{int}$

$A \vdash 3 : \text{int}$

Another Example

$A = x : \text{int} \rightarrow \text{int}$

$B = A, x : \text{int}$

$B \vdash + : \text{int} \rightarrow \text{int} \rightarrow \text{int}$

$B \vdash x : \text{int}$

$B \vdash + x : \text{int} \rightarrow \text{int}$

$B \vdash + x 3 : \text{int}$

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

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

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
  - $\text{TypeCheck} : \text{type env} \times \text{expression} \rightarrow \text{type}$
    - $\text{TypeCheck}(\emptyset, n) = \text{int}$
    - $\text{TypeCheck}(\emptyset, x) = \text{if } x \text{ in } \text{dom}(\emptyset) \text{ then } A(x) \text{ else fail}$
    - $\text{TypeCheck}(A, \lambda x : t.e) = \text{TypeCheck}(A, x : t) \vdash e$
    - $\text{TypeCheck}(A, e_1 e_2) =$
      - let $t_1 \rightarrow t_2 = \text{TypeCheck}(A, e_1)$ in
      - let $t_1' = \text{TypeCheck}(A, e_2)$ in
      - if $t_1 = t_1'$ then $t_2$ else fail

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
    - $(\lambda x.e1) v2 \rightarrow e1[v2\times]$  \hspace{1cm} $e1 \rightarrow e1'$
    - $e1 e2 \rightarrow e1' e2$
    - $e2 \rightarrow e2'$
    - $v1 e2 \rightarrow v1 e2'$
    - $e ::= v \mid x \mid e e$
    - $v ::= n \mid \lambda x.t.e$  \textit{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 $e$
  - Base cases $n, x, \lambda x.e$ – Impossible, since these terms don’t reduce
  - Induction. Assume $\vdash e_1 e_2 : t$ and $e_1 e_2 \rightarrow e'$. Then we have $\vdash e_1 : t \rightarrow t$ and $\vdash e_2 : t$. (Why?)
  - 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
  - $\vdash \lambda x : t' \vdash e : t$
  - $\vdash \lambda x : t' \vdash \lambda x : t' \rightarrow t$
  - Thus we have
    - $\vdash x : t' \vdash e : t$
    - $\vdash x : t' \vdash e'$
  - Then by the substitution lemma (not shown) we have
    - $\vdash e[v/x] : t$
  - And so we have preservation

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?
- Semantics define bad things (evaluation getting stuck)
- “Well-typed programs don’t go wrong”

Substitution Lemma

- If $A \vdash v : t$ and $A, x : t \vdash e : t'$, then $A \vdash e[v/x] : t'$
- Proof: Induction on the structure of $e$

Product Types (Tuples)

- $a ::= \ldots | (e, e) | \text{fst} e | \text{snd} e$

- Or, maybe, just add functions
  - $\text{pair} : t \rightarrow t' \rightarrow t \times t'$
  - $\text{fst} : t \times t' \rightarrow t$
  - $\text{snd} : t \times t' \rightarrow t'$
Sum Types (Tagged Unions)

- \( e ::= \ldots \mid \text{inL}_t e \mid \text{inR}_t e \)
- \( (\text{case } e \text{ of } x_1 : t_1 \rightarrow e_1 \mid x_2 : t_2 \rightarrow e_2) \)

- \( \frac{e : t_1}{A \vdash \text{inL}_{t_2} e : t_1 + t_2} \)
- \( \frac{e : t_2}{A \vdash \text{inR}_{t_1} e : t_1 + t_2} \)
- \( \frac{A \vdash e : t_1 + t_2}{A, x_1 : t_1 \vdash e_1 : t \quad A, x_2 : t_2 \vdash e_2 \vdash t} \)

Self Application and Types

- Self application is not checkable in our system
- \( A, x : \alpha \vdash x : t \rightarrow t' \quad A, x : \alpha \vdash \lambda x : \alpha. x : t \)
- \( A \vdash \lambda x : \alpha. x : t \)
- It would require a type \( t \) such that \( t = t \rightarrow t' \)

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: \( \mu \alpha. \text{int} \rightarrow \alpha \)

Folding and Unfolding

- We can check type equivalence with the previous definition
- Standard unification, omit occurs checks
- Alternative solution:
  - The programmer puts in explicit fold and unfold operations to expand/contract one “level” of the type trees
  - \( \text{unfold } \mu \alpha. t = t[\mu \alpha. t] \)
  - \( \text{fold } t[\mu \alpha. t] = \mu \alpha. t \)

Discussion

- 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 different name are considered different
  - E.g., struct foo { int x; struct foo *next; } is different from struct bar { int x; struct bar *next; }

Recap

- We’ve discussed simple types so far
  - Integers, functions, pairs, unions
  - Extensions for recursive types and updatable refs
- Type systems have nice properties
  - Type checking is straightforward (needs annotations)
  - Well typed programs don’t go “wrong”
    - They don’t get stuck in the operational semantics
- But... We can’t type check all good programs
**Up Next: Improving Types**

- How can we build more flexible type systems?
  - More programs type check
  - Type checking is still tractable

- How can reduce the annotation burden?
  - Type inference

---

**Type Inference**

- Let's consider the simply typed lambda calculus with integers
  - $e ::= n \mid x \mid \lambda x.e \mid e e$
  - (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?
  - Notice that lambda terms above have no type annotation

---

**Type Language**

- Problem: Consider the rule for functions

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

- Without type annotations, where do we get $t$?
  - We'll use type variables 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

---

**Example**

\[
A, x : \alpha \vdash x : \alpha \\
A \vdash (\lambda x.x) : \alpha \to \alpha \\
A \vdash 3 : \text{int} \\
A \vdash (\lambda x.x) 3 : \beta
\]

- We collect all constraints appearing in the derivation into some set $C$ to be solved
- Here, $C$ consists of one constraint $\alpha \to \alpha = \text{int} \to \beta$
  - 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

---

**Type Inference Rules**

- $A \vdash n : \text{int}$
- $x \in \text{dom}(A)$
  - $A \vdash x : A(x)$
- $A, x : \alpha \vdash e : t'$
  - $\alpha$ fresh
  - $A \vdash \lambda x.e : \alpha \to t'$
- $A \vdash e_1 : t_1$
  - $A \vdash e_2 : t_2$
  - $t_1 = t_2 \to \beta$
  - $\beta$ fresh
  - $A \vdash e_1 e_2 : \beta$

- “Generated” constraint

---

**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
  - $C \cup \{\text{int=\text{int}}\} \Rightarrow C$
  - $C \cup \{\alpha=\alpha\} \Rightarrow C[\alpha\alpha]$
  - $C \cup \{\alpha=\alpha\} \Rightarrow C[\alpha\alpha]$
  - $C \cup \{t_1 \to t_2 = t_1' \to t_2'\} \Rightarrow C \cup \{t_1=\alpha\} \cup \{t_2=\beta\}$
  - $C \cup \{\text{int=\text{int}}\} \Rightarrow \text{unsatisfiable}$
  - $C \cup \{t_1 \to \text{\text{int=\text{int}}}\} \Rightarrow \text{unsatisfiable}$
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
- In the operation $C[\alpha\alpha]$, require that $\alpha\epsilon FV(t)$ (where $FV(t)$ is as expected, defined formally later)
  - Called the “occurs check”
- In practice, it may better to allow $\alpha\epsilon FV(t)$ and do the occurs check at the end
  - But that can be awkward to implement

Unifying a Variable and a Type

- Computing $C[\alpha\alpha]$ by substitution is inefficient
- Instead, use a union-find data structure to represent equal types
  - The types are in a union-find forest
  - When a variable and a type are equated, we union them so they have the same ECR
    - Want the ECR to be the concrete type with which variables have been unified, if one exists. Thus we can read off the solution by reading the ECR for each set.

Example

\[
\begin{align*}
\alpha &\rightarrow int \\
\beta &\rightarrow int \\
Y &\rightarrow int \\
\alpha &= \int \rightarrow \beta \\
Y &= \int \rightarrow \int \\
\alpha &= Y
\end{align*}
\]

Discussion

- The algorithm we’ve given finds the most general type of a term
  - Any other valid type is “more specific,” e.g.,
    - $\lambda x : int \rightarrow int$
  - 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”
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:
  • For any type \( \alpha \), the identity function has type \( \alpha \rightarrow \alpha \)
  • This is also known as parametric polymorphism

“Manual” use of polymorphism

• Let’s extend our system in two simple ways:
  • \( t ::= \alpha \mid \text{int} \mid t \rightarrow t \mid \forall \alpha.t \)
  • \( e ::= n \mid x \mid \lambda x.e \mid e \ e \mid e[r] \)

• That is, we add polymorphic types, and we add explicit type instantiations
  • Explicitly annotated code locations at which a value of polymorphic type is used
  • We also need to know when to introduce polymorphic types
    • Called generalization
    • For now, we’ll just define suitable conditions, but won’t be syntax-driven

Instantiation

• When we use a parametric polymorphic type, we instantiate it with a particular type
  • Suppose the programmer specifies this by hand as
  1. \( (\lambda x.x)[t_1] : \alpha \rightarrow \alpha \)
  2. \( (\lambda x.x)[t_2] : \alpha \rightarrow \alpha \)

• 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

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
  • \( \text{FV}(\alpha) = \{ \alpha \} \)
  • \( \text{FV}(\beta) = \emptyset \)
  • \( \text{FV}(\epsilon) = \text{FV}(\epsilon) \cup \text{FV}(\epsilon') \)
  • \( \text{FV}(\forall \alpha. \epsilon) = \text{FV}(\epsilon) - \{ \alpha \} \)
  - Look familiar?

Substitution, Again

• Define \( u[\alpha] \) as
  • \( \alpha[u][\alpha] = u \)
  • \( \beta[u][\alpha] = \beta \) where \( \beta \neq \alpha \)
  • \( (\epsilon \rightarrow \epsilon')[u][\alpha] = \epsilon[u][\alpha] \rightarrow \epsilon'[u][\alpha] \)
  • \( (\forall \beta. \epsilon)[u][\alpha] = \forall \beta. (\epsilon[u][\alpha]) \) where \( \beta \neq \alpha \) and \( \beta \not\in \text{FV}(u) \)

• Look familiar?
Generalization

• 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

- $A, x: \alpha \vdash e: \alpha$
- $A \vdash \lambda x.x : \alpha \rightarrow \alpha$
- $A, x(\rightarrow i) \vdash x : (i \rightarrow i)$
- $A \vdash \lambda x : (i \rightarrow i) \rightarrow (i \rightarrow i)$

• 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)

- $A, x: \alpha \vdash e : \alpha$
- $A \vdash \lambda x.x : \alpha \rightarrow \alpha$
- $A, x(\rightarrow i) \vdash x : (i \rightarrow i)$
- $A \vdash \lambda x : (i \rightarrow i) \rightarrow (i \rightarrow i)$

• The choice of the type of $x$ depends on the type environment ($x$ must be $\alpha$ because $y$ is)
  - 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

- $A \vdash e : t$
- $\alpha \notin \text{FV}(A)$
- $A \vdash e : \forall \alpha t$

• 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 $\alpha \notin \text{FV}(A)$

• Then let $u$ be any type. By induction, can show
  - $A[\alpha t] \vdash e : t[u\alpha]$
  - But then since $\alpha \notin \text{FV}(A)$, that’s equivalent to
  - $A \vdash e : t[u\alpha]$
Kinds of Polymorphism

- We've just seen parametric polymorphism
  - A more restrictive variant is also called Hindley-Milner style polymorphism
- Another popular flavor 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

Polymorphic Type Inference

- We'd like to extend our algorithm to polymorphic type inference
  - Perform generalization and instantiation automatically (and deterministically); remove explicit type instantiations
- Major problem: Our system for polymorphism is too expressive
  - In fact, type inference is undecidable when generalization and instantiation can be at arbitrary syntactic positions

Hindley-Milner Polymorphism

- Restrict polymorphism to only the "top level"
  - Introduce polymorphism at let
  - Fully instantiate when we use a variable with a polymorphic type
- Here is our new language
  - $e ::= n \mid x \mid \lambda x.e \mid e_1 e_2 \mid \text{let } x = e \text{ in } e$
  - $t ::= \alpha \mid \text{int} \mid t \rightarrow t$
    - These are type schemes.
  - $A ::= \emptyset \mid A, x: s$
    - Notice that, according to the prior instantiation rule, we won't instantiate $\alpha$ with a scheme $s$, only a type $t$

Old Type Inference Rules

\[ A \vdash n : \text{int} \]
\[ A, x: \alpha \vdash e : t \quad (\alpha \\text{fresh}) \]
\[ A \vdash \lambda x.e : \alpha \rightarrow t \quad (\alpha \\text{fresh}) \]
\[ A \vdash e_1 e_2 : t \quad (t_1 = t_2) \quad (\beta \\text{fresh}) \]
\[ A \vdash \text{let } x = e_1 \text{ in } e_2 : t_2 \]

New Type Inference Rules

- At let, generalize over all possible variables
  \[
  A \vdash e_1 : t_1 \quad A, x: \forall \alpha.t_1 \vdash e_2 : t_2 \quad (\alpha = \text{FV}(t_1) - \text{FV}(A)) \]
  \[ A \vdash \text{let } x = e_1 \text{ in } e_2 : t_2 \]

- At variable uses, instantiate to all fresh types
  \[
  A(x) = \forall \alpha.t_1 \quad (\bar{\alpha} \\text{fresh}) \]
  \[ A \vdash x : t[\bar{\alpha} \bar{x}] \]

  - Here the $\bar{\alpha}$ denotes a list of type variables

Example

- Parametric polymorphic type inference
  - let $x = \lambda x.x$ in
  - $x \ 3$; $x : \beta \rightarrow \beta, \beta = \text{int}$
  - $x (\lambda y.y)$; $x : y \rightarrow y, y = \delta \rightarrow \delta$

  - This would be untypable in a monomorphic type system
**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

**An Imperative Language**

- \( e ::= x \mid \lambda x. e \mid e e \mid \text{ref } e \mid !e \mid e := e \mid e ; e \)
- Notice that this is not C
  - Variables cannot be updated; only references can
  - I.e., there are no l-values or r-values
- This is a language with updatable references

**Examples**

- \( !(\text{ref } 0) \)
- let x = ref 0 in
  - x := !x + 1
- let x = ref 0 in
  - \( \lambda y. x := !x + 1 ; !x \)

**Type Checking Rules**

- \( t ::= \ldots \mid \text{ref } t \)
- Note: in ML this type is written \( t \text{ ref} \)
- \[
  \frac{A \vdash e : t}{A \vdash \text{ref } e : \text{ref } t} \quad \frac{A \vdash e : \text{ref } t}{A \vdash !e : t} \\
  \frac{A \vdash e_1 : \text{ref } t \quad A \vdash e_2 : t}{A \vdash e_1 := e_2 : \text{unit}} \quad \frac{A \vdash () : \text{unit}}{}
  \]

**Unit and the Unit Type**

- Sometimes in imperative programs we write expressions that have some side effect but no interesting result
- To represent this directly, use unit:
  - \( e ::= \ldots \mid () \)
  - \( t ::= \ldots \mid \text{unit} \)
- \[
  \frac{A \vdash e_1 : \text{ref } t \quad A \vdash e_2 : t}{A \vdash e_1 := e_2 : \text{unit}} \quad \frac{A \vdash () : \text{unit}}{A \vdash e_1 := e_2 : \text{unit}}
  \]

**Operational Semantics**

- Now we need to keep track of memory
  - State is a map from locations to values
  - Our redexes will be tuples \( \langle \text{State}, \text{expression} \rangle \)
  - As a consequence, order of evaluation matters
- As before, evaluation will yield a fully-evaluated term, also called a value
  - \( v ::= x \mid \lambda x. e \)
  - \( e ::= v \mid e e \mid \text{ref } e \mid !e \mid e := e \)
Operational Semantics (cont’d)

\[
\begin{align*}
(S, (\lambda x.e)) & \rightarrow (S', (\lambda x.e)) \\
(S, e) & \rightarrow (S', v1) \quad (S', e2) \rightarrow (S'', v2) \\
(S, e1; e2) & \rightarrow (S, ((\lambda x.e)) \\
(S, e) & \rightarrow n \quad S, v1 \\
(S, e1; e2) & \rightarrow n \quad S, v2 \\
(S, e1) & \rightarrow n \quad S, v \\
(S, e1) & \rightarrow n \quad S, ref e \\
(S, e1 := e2) & \rightarrow n \quad S, e[v\alpha] \\
(S, !e) & \rightarrow n \quad S'[loc], v \\
(S, e1 := e2) & \rightarrow n \quad S', v \\
(S, e) & \rightarrow n \quad S'[loc], loc \\
(S, ref e) & \rightarrow n \quad S'[loc], loc
\end{align*}
\]

Polymorphism and References

- Suppose we want polymorphism in our imperative language
  - \[ e ::= x | n | \lambda x.e | e e | ref e | !e | e := e \]
  - \[ s ::= t | \alpha . s \]
  - \[ t ::= \alpha | int | t \rightarrow t | ref t \]

- What if we try our standard rule?
  \[
  A \vdash e1 : t1 \quad A, x:\alpha . t \vdash e2 : t2 \quad \alpha = FV(t) - FV(A)
  \]
  \[
  A \vdash let \ x = e1 \ in \ e2 : t2
  \]

Solution: The Value Restriction

- Only allow values to be generalized
  - \[ v ::= x | n \]
  - \[ e ::= v | e e \]

\[
\begin{align*}
A \vdash v : t1 \quad A, x:\forall \alpha . t = e2 : t2 \quad \alpha = FV(t) - FV(A) \\
A \vdash let \ x = v \ in \ e2 : t2
\end{align*}
\]

- Intuition: Values cannot later be updated
- This solution due to Wright and Felleisen – Tofta found a much more complicated solution

Naive Generalization is Unsound

- Example (due to Tofta)
  - \[ let r = ref (\lambda x.e) \in \quad \alpha \vdash r : \forall \alpha . ref (\alpha \rightarrow \alpha) \]
  - \[ r = \lambda x.e + 1; \quad ll \ checks; use \ r \ at \ ref (int \rightarrow int) \]
  - \[ \text{(l) true} \quad \alpha \vdash \text{let r = ref (bool \rightarrow bool)} \]

- \[ \alpha \] should not be generalized, because later uses of \( r \) may place constraints on it

- Nobody realized this problem for a long time

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

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