Consider the (untyped) lambda calculus

- $\text{false} = \lambda x.\lambda y.x$
- $\text{0 (Scott)} = \lambda x.\lambda y.x$

Everything is encoded as a function

- So we can easily misuse combinators
  - $\text{false 0 if 0 then ... etc...}$
- 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
  - $\text{false 0}$ // ill-typed: can’t apply a boolean
  - $1 + (\text{if true then 0 else false})$ // ill-typed: can’t add boolean to integer

### 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

### 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
    - $\text{false 0 if 0 then ... etc...}$
  - This is no better than assembly language!

### 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*

### 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

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

Example

$$A = - : \text{int} \to \text{int}$$

- $\not\in \text{dom}(A)$
- $A \vdash - : \text{int} \to \text{int}$
- $A \vdash 3 : \text{int}$
- $A \vdash \lambda x : \text{int} \to \text{int} \vdash x : \text{int} \to \text{int}$
- $A \vdash + : \text{int} \to \text{int}$
- $A \vdash + x : \text{int} \to \text{int}$
- $A \vdash + x 3 : \text{int}$
- $A \vdash (\lambda x : \text{int} \to \text{int} \vdash x : \text{int} \to \text{int}) : \text{int} \to \text{int}$
- $A \vdash 4 : \text{int}$

Another Example

$$A = + : \text{int} \to \text{int} \to \text{int}$$

- $\not\in \text{dom}(B)$
- $B \vdash 3 : \text{int}$
- $A \vdash 4 : \text{int}$
- $B \vdash + : B \vdash x : i$
- $B \vdash + x : \text{int} \to \text{int}$
- $B \vdash + x 3 : \text{int}$
- $A \vdash (\lambda x : \text{int} \to \text{int} \vdash x : \text{int} \to \text{int}) : \text{int} \to \text{int}$
- $A \vdash 4 : \text{int}$

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
  - $\text{TypeCheck} : \text{type env} \times \text{expression} \rightarrow \text{type}$
    - $\text{TypeCheck}(A, n) = \text{int}$
    - $\text{TypeCheck}(A, x) = \text{if } x \in \text{dom}(A) \text{ then } A(x) \text{ else } \text{fail}$
    - $\text{TypeCheck}(A, \lambda x : t. e) = \text{TypeCheck}((A, x : t), e)$
    - $\text{TypeCheck}(A, e_1 e_2) = \text{let } t_1 = \text{TypeCheck}(A, e_1) \text{ in }$
      - $\text{let } t_2 = \text{TypeCheck}(A, e_2) \text{ in }$
      - $\text{if } \text{dom}(t_1) = t_2 \text{ then } \text{range}(t_1) \text{ else } \text{fail}$

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
    - $(\lambda x. e_1) \ e_2 \rightarrow e_1[e_2/x]$  
    - $e_1 \rightarrow e_1'$
    - $e_2 \rightarrow e_2'$
    - $v_1 \ e_2 \rightarrow v_1 \ e_2'$
    - $e ::= v \mid x \mid e \ e$
    - $v ::= n \mid \lambda x : t. e$  
      - 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 \( \vdash e_2 : t' \).
  - 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 \( A, x : t \vdash e : t' \), then \( A \vdash e[v/x] : t' \)
- Proof: Induction on the structure of \( e \)

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

Product Types (Tuples)

- \( e ::= \ldots | (e, e) \) fst e snd e
  \[
  \frac{A \vdash e_1 : t \quad A \vdash e_2 : t'}{A \vdash (e_1, e_2) : t \times t'} \quad \frac{A \vdash e : t \times t'}{A \vdash \text{fst } e : t} \quad \frac{A \vdash e : t \times t'}{A \vdash \text{snd } e : t}
  \]
  - 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)

\[ e ::= \ldots | \text{inL}_{t_2} e | \text{inR}_{t_1} e \]

\[ \text{inL}_{t_2} e \vdash e : t_1 + t_2 \]

\[ \text{inR}_{t_1} e \vdash e : t_1 + t_2 \]

\[ (\text{case } e \text{ of } x_1 : t_1 \rightarrow e_1 | x_2 : t_2 \rightarrow e_2) \vdash e : t_1 + t_2 \]

Self Application and Types

- Self application is not checkable in our system
  \[ A, x : t \vdash \lambda x : t. x \vdash x : t \]
  \[ A, x : t \vdash x : t \]

- It would require a type \( t = t \rightarrow t' \)
  - (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: \( \mu \alpha.\text{int} \rightarrow \alpha \)

\[
\begin{align*}
\text{int} & \quad \text{or} \quad \text{int} \\
\text{int} & \quad \text{int} \quad \text{int}
\end{align*}
\]

Folding and Unfolding

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

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 a 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
  - $e ::= n \mid x \mid \lambda x : t . e \mid e \ e$
  - (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?

Type Language

- Problem: Consider the rule for functions
  - $A, x : \alpha \vdash e : t'$
  - $A \vdash \lambda x : \alpha . e : \alpha \to t'$

- 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

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

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) : \alpha \to \alpha = \text{int} \to \beta
A \vdash (\lambda x . x) \ 3 : \beta
```

- We can solve the constraint $\alpha \to \alpha = \text{int} \to \beta$
  - $\alpha = \text{int} = \beta$
- 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
  - $\text{C U } \{\text{int=int} \} \Rightarrow \text{C}$
  - $\text{C U } \{\alpha = \alpha \} \Rightarrow \text{C[\alpha]}$
  - $\text{C U } \{\text{t=t} \} \Rightarrow \text{C[\alpha]}$
  - $\text{C U } \{t_1 \rightarrow t_2 = t' \rightarrow t' \} \Rightarrow \text{C U } \{t_1=t' \} \cup \{t_2=t' \}$
  - $\text{C U } \{\text{int=int} \} \Rightarrow \text{unsatisfiable}$
  - $\text{C U } \{t_1 \rightarrow t_2 = \text{int} \} \Rightarrow \text{unsatisfiable}$

Solving Equality Constraints
**Occurs Check**

- We don’t have recursive types, so we shouldn’t infer them.
- So in the operation $C[t][\alpha]$, require that $\alpha \in \text{FV}(t)$.
- 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 $C[t][\alpha]$ 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.
  - Note: Only need to maintain ECR of variables, not of all terms, though doing terms as well has some potential advantages.

**Example**

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

**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.

**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.”

**Parametric Polymorphism**

- Observation: $\lambda x.x$ returns its argument exactly and does not place any 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.
**Instantiation**

- When we use a parametric polymorphic type, we instantiate it with a particular type
  - For now, the programmer specifies this by hand
  - \((\lambda x.x)[S] : S \to S\)
  - \((\lambda x.x)[T] : T \to T\)

- This is where the term *parametric* comes from
  - The type \(\forall \alpha. \alpha \to \alpha\) is a “function” in the domain of types, and it is passed a parameter at instantiation time
  - Sometimes this type is written \(\forall \alpha. \alpha \to \alpha\)

**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

**Substitution, Again**

- Define \(t[u|\alpha]\) as
  - \(\alpha[u|\alpha] = u\)
  - \(\beta[u|\alpha] = \beta\) where \(\beta \neq \alpha\)
  - \((t\to t')[u|\alpha] = t'[u|\alpha] \to t[u|\alpha]\)
  - \((\forall \beta. t)[u|\alpha] = \forall \beta. (t[u|\alpha])\) where \(\beta \neq \alpha\) and \(\beta \in \text{FV}(u)\)

- 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**

- 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 \(x\)
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.

\[
\begin{align*}
A, x: \text{int} & \vdash x : \text{int} \\
A & \vdash \lambda x. x + 3 : \text{int} \rightarrow \text{int}
\end{align*}
\]

- 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 \( x \).

\[
\begin{align*}
A, y: \alpha, x: \alpha & \vdash \text{if } p \text{ then } x \text{ else } y : \alpha \\
A, y: \alpha & \vdash \lambda x. \text{if } p \text{ then } x \text{ else } y : \alpha \rightarrow \alpha \\
A, y: \alpha, x: \text{int} & \vdash \text{if } p \text{ then } x \text{ else } y : \text{int} \\
A, y: \alpha & \vdash \lambda x. \text{if } p \text{ then } x \text{ else } y : \text{int} \rightarrow \text{int}
\end{align*}
\]

- We can generalize any type variable that is unconstrained by the environment.
  - Warning: This won’t quite work with refs.

\[
\begin{align*}
A & \vdash e : t \\
\alpha \in \text{FV}(A) & \vdash A \vdash e : \forall \alpha. t
\end{align*}
\]

- We’ve seen two forms of polymorphism.
  - Subtype polymorphism (see OOP)
  - Parametric polymorphism
    - A more restrictive variant is also called Hindley-Milner style polymorphism
  - Some languages also have ad-hoc polymorphism
    - E.g., \( + \) operator that works on ints and floats
    - E.g., overloading in Java

- We’d like to extend our algorithm to polymorphic type inference.
- Major problem: Our system for polymorphism is too expressive.
  - In fact, type inference is undecidable.
Hindley-Milner Polymorphism

- Restrict polymorphism to only the “top level”
- Only introduce polymorphism at `let`
- Always fully instantiate when we use a variable with a polymorphic type
  - `e ::= n | x | λx.e | e e | let x = e in e`
  - `s ::= t | ∀α.s`
    - These are type schemes
  - `t ::= α | int | t → t`

Old Type Inference Rules

<table>
<thead>
<tr>
<th>Rule</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>(A \vdash n : \text{int})</td>
<td><code>A := n : \text{int}</code></td>
</tr>
</tbody>
</table>
| \(A, x : \alpha \vdash e : t' \quad \alpha\text{ fresh}\) | `A := e1 : t1 \quad A := e2 : t2`
| \(A \vdash \lambda x.e : \alpha \rightarrow t'\) | `A := e1 : \beta\quad \beta\text{ fresh}`

New Type Inference Rules

- At `let`, generalize over all possible variables
  \(A \vdash e1 : t1 \quad A, x : \forall\alpha.t1 \vdash e2 : t2 \quad \forall\alpha = \text{FV}(t1) - \text{FV}(A)\)
  \(A \vdash \text{let } x = e1 \text{ in } e2 : t2\)
- At variable uses, instantiate to all fresh types
  \(A(x) = \forall\alpha.t \quad \beta\text{ fresh}\)
  \(A \vdash x : t[\beta[\alpha]]\)

Example

- Parametric polymorphic type inference
  
  | `let x = λx.x in` | `// x : ∀\alpha.\alpha \rightarrow \alpha`
  | `x 3;` | `// x : \beta \rightarrow \beta, \beta = \text{int}`
  | `x (λy.y)` | `// x : γ \rightarrow γ, γ = \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 previously algorithm, plus generalize at `let` and instantiate at variable uses

An Imperative Language

- `e ::= x | λx.e | e e`
  - `| ref e` allocation
  - `| !e` dereference
  - `| e := e` assignment
  - `| e; e` sequencing

- 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

\( \texttt{let } x = \texttt{ref } 0 \texttt{ in } x := !x + 1 \)

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

Type Checking Rules

- \( t ::= \ldots | \texttt{ref } t \)
  
  Note: in ML this type is written \( t \texttt{ ref} \)

\[
\begin{align*}
A \vdash e : t & \quad A \vdash \texttt{ref } e : \texttt{ref } t \\
A \vdash e : \texttt{ref } t & \quad A \vdash !e : t \\
A \vdash e_1 : \texttt{ref } t & \quad A \vdash e_2 : t \\
A \vdash e_1 := e_2 : \texttt{unit} \\
A \vdash () : \texttt{unit}
\end{align*}
\]

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:
  
  - \( e ::= \ldots | () \)
  
  - \( t ::= \ldots | \texttt{unit} \)

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
    
    - \( v ::= x | \lambda x. e \)
    
    - \( e ::= v | e e | \texttt{ref } e | !e | e := e \)

Operational Semantics (cont’d)

\[
\begin{align*}
\langle S, (\lambda x. e) \rangle & \rightarrow \langle S', (\lambda x. e) \rangle \\
\langle S, e_1 \rangle & \rightarrow \langle S', v \rangle \quad \langle S', e_2 \rangle \rightarrow \langle S'', v \rangle \\
\langle S, e_1; e_2 \rangle & \rightarrow \langle S'', v \rangle \\
\langle S, e \rangle & \rightarrow \langle S', v \rangle \quad \text{loc fresh} \\
\langle S, \texttt{ref } e \rangle & \rightarrow \langle S[\text{vloc}], \text{loc} \rangle
\end{align*}
\]

Operational Semantics (cont’d)

\[
\begin{align*}
\langle S, e \rangle & \rightarrow \langle S', \text{loc} \rangle \\
\langle S, !e \rangle & \rightarrow \langle S', S'[\text{loc}] \rangle \\
\langle S, e_1 \rangle & \rightarrow \langle S', \text{loc} \rangle \quad \langle S', e_2 \rangle \rightarrow \langle S'', v \rangle \\
\langle S, e_1 := e_2 \rangle & \rightarrow \langle S'', \text{vloc}, v \rangle \\
\langle S, e_1 \rangle & \rightarrow \langle S', \lambda x. e \rangle \quad \langle S', e_2 \rangle \rightarrow \langle S'', v \rangle \quad \langle S'', e[v\langle x \rangle] \rightarrow \langle S''', v \rangle \\
\langle S, e_1 e_2 \rangle & \rightarrow \langle S''', v \rangle
\end{align*}
\]
Polymorphism and References

- Suppose we want polymorphism in our imperative language
  - \( e ::= x \mid n \mid \lambda x.e \mid e \; e \mid \text{ref} \; e \mid !e \mid e := e \)
  - \( s ::= t \mid \alpha.s \)
  - \( t ::= \alpha \mid \text{int} \mid t \rightarrow t \mid \text{ref} \; t \)

- What if we try our standard rule?
  \[
  A \vdash e_1 : t_1 \quad A, x: \alpha.t_1 \vdash e_2 : t_2
  \]

- \( \alpha = \text{FV}(t_1) - \text{FV}(A) \)

- \( A \vdash \text{let} \; x = e_1 \; \text{in} \; e_2 : t_2 \)

Naive Generalization is Unsound

- Example (due to Tofte)
  - \( \text{let} \; r = \text{ref} \; (\lambda x.x) \; \text{in} \quad // \; r : \forall \alpha.\text{ref} \; (\alpha \rightarrow \alpha) \)
  - \( r ::= \lambda x.x+1; \quad // \; \text{checks; use} \; r \; \text{at ref} \; \text{(int} \rightarrow \text{int}) \)
  - \( (!r) \; \text{true} \quad // \; \text{oops!} \; \text{checks; use} \; r \; \text{at ref(bool} \rightarrow \text{bool}) \)

- \( \alpha \) 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

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