CMSC 631 – Program Analysis and Understanding
Spring 2013

Type Systems
Consider the (untyped) lambda calculus:
- false = λx.λy.x
- 0 (Scott) = λx.λy.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!

The Need for a Type System
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
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

- $e ::= n \mid x \mid \lambda x:t.e \mid e\;e$
  - Functions include the type of their argument
  - We don’t really need this, but it will come in handy

- $t ::= \text{int} \mid t \to t$
  - $t_1 \to t_2$ 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)
  - $\emptyset$ 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

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

\[ A \vdash x : A(x) \]

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

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

\[ A \vdash e_1 : t \to t' \quad A \vdash e_2 : t \]

\[ A \vdash e_1 \; e_2 : t' \]
Example

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

\[
\frac{\text{-} \in \text{dom}(A)}{
A \vdash \text{- : int} \to \text{int} \quad A \vdash 3 : \text{int}
}
\]

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

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

\[
\begin{array}{c}
\text{+} \in \text{dom}(B) & \quad x \in \text{dom}(B) \\
B \vdash + : & \quad B \vdash x : \text{int} \\
B \vdash + x : \text{int} \rightarrow \text{int} & \quad B \vdash 3 : \text{int} \\
B \vdash + x 3 : \text{int} & \quad A \vdash 4 : \text{int} \\
A \vdash (\lambda x: \text{int}. + x 3) : \text{int} \rightarrow \text{int} \\
A \vdash (\lambda x: \text{int}. + x 3) 4 : \text{int}
\end{array}
\]

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
  - \( \text{TypeCheck} : \text{type env} \times \text{expression} \rightarrow \text{type} \)

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

\[
\frac{(\lambda x:t.e_1) v_2 \rightarrow e_1[x \mapsto v_2]}{el \rightarrow el'}
\]

\[
\frac{e_1 e_2 \rightarrow e_1' e_2}{el e_2 \rightarrow el' e_2'}
\]

\[
\frac{e_2 \rightarrow e_2'}{v_1 e_2 \rightarrow v_1 e_2'}
\]

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

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

\[
\begin{align*}
x : t' & \vdash e : t \\
\hline
\vdash \lambda x : t'. e : t' \rightarrow t
\end{align*}
\]

- 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 $A \vdash e[x \mapsto v] : t'$

• 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 $A \vdash e[x \mapsto e_1] : t'$
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”
Product Types (Tuples)

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

\[ A \vdash e_1 : t \quad A \vdash e_2 : t' \]
\[ \frac{}{A \vdash (e_1, e_2) : t \times t'} \]

\[ A \vdash e : t \times t' \]
\[ \frac{}{A \vdash fst e : t} \quad \frac{}{A \vdash snd e : t'} \]

• Or, maybe, just add functions

  - pair : t → t' → t × t'
  - fst : t × t' → t
  - snd : t × t' → t'

**Sum Types (Tagged Unions)**

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

\[
\begin{array}{c}
A \vdash e : t_1 \\
A \vdash \text{inL}_{t_2} e : t_1 + t_2 \\
A \vdash e : t_1 + t_2
\end{array}
\]

\[
\begin{array}{c}
A \vdash e : t_2 \\
A \vdash \text{inR}_{t_1} e : t_1 + t_2 \\
A \vdash e : t_1 + t_2
\end{array}
\]

\[
\begin{array}{c}
A, x_1 : t_1 \vdash e_1 : t \\
A, x_2 : t_2 \vdash e_2 : t
\end{array}
\]

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

• Self application is not checkable in our system

\[
\begin{align*}
A, x:? \vdash x : t \rightarrow t' \\
\hline
A, x:? \vdash x : t \\
\hline
A, x:? \vdash x \ x : ...
\end{align*}
\]

- 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 $t = t \to t'$
  - We define the type $\mu \alpha.t$ to be the solution to the (recursive) equation $\alpha = t$
  - Example: $\mu \alpha.\text{int} \to \alpha$

```
  int
 /   \
/     \
int    int
 / \
/   \
int int
```

or
```
  int
 /   \
/     \
int    int
 / \
/   \
int
```
Folding and Unfolding

• We can check type equivalence with the previous definition (equi-recursive types)
  - Standard unification, omit occurs checks

• Alternative solution (iso-recursive types):
  - The programmer puts in explicit fold and unfold operations to expand/contract one “level” of the type trees
    - unfold $\mu\alpha.t = t[\alpha \mapsto \mu\alpha.t]
    - fold $t[\alpha \mapsto \mu\alpha.t] = \mu\alpha.t$
Iso-recursive Types

\[ e ::= \ldots | \text{fold } e | \text{unfold } e \]

\[
\begin{align*}
A \vdash e : t[\alpha \mapsto \mu \alpha.t] & \quad & A \vdash e : \mu \alpha.t \\
A \vdash \text{fold } e : \mu \alpha.t & \quad & A \vdash \text{unfold } e : t[\alpha \mapsto \mu \alpha.t]
\end{align*}
\]
ML Datatypes

• Combines iso-recursive and sum types
  ▪ Each occurrence of a type constructor when producing a value corresponds to occurrences of \texttt{inL/inR} and, when recursion is involved, \texttt{fold}
  ▪ Each occurrence of a type constructor in a pattern match corresponds to a \texttt{case} and, when recursion is involved, (at least one) \texttt{unfold}
ML Datatypes Example

- `type intlist = Int of int | Cons of int * intlist`
  - Equivalent to \( \mu \alpha.\text{int+}(\text{int} \times \alpha) \)

- `(Int 3)` equivalent to
  - `fold (inL_{\text{int} \times \mu \beta.\text{int+}(\text{int} \times \beta)} 3)`

- `(Cons (2,(Int 3)))` equivalent to
  - `fold (inR_{\text{int}} 2, fold (inL_{\text{int} \times \mu \beta.\text{int+}(\text{int} \times \beta)} 3))`

- `match e with Int x -> e1 | Cons x -> e2` same as
  - `case (unfold e)`
    - `x:int -> e1`
    - `| x: \text{int} \times (\mu \beta.\text{int+}(\text{int} \times \beta)) -> e2`
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
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 \textit{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 \textit{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
  - $\lambda \alpha. \lambda x: \alpha. x : \forall \alpha. \alpha \rightarrow \alpha$
  - $\lambda \alpha. \lambda \beta. \lambda x: \alpha. \lambda y: \beta. x : \forall \alpha. \forall \beta. \alpha \rightarrow \beta \rightarrow \alpha$
  - $\lambda \alpha. \lambda \beta. \lambda x: \alpha. \lambda y: \beta. y : \forall \alpha. \forall \beta. \alpha \rightarrow \beta \rightarrow \beta$
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

\[ A, \alpha \vdash e : t \]

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

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

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

• 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
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
  ▪ $FV(\alpha) = \{\alpha\}$
  ▪ $FV(c) = \emptyset$
  ▪ $FV(t \rightarrow t') = FV(t) \cup FV(t')$
  ▪ $FV(\forall \alpha . t) = FV(t) - \{\alpha\}$
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 $n[\alpha \mapsto u] = n$
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

!(ref 0)

let x = ref 0 in
  x := !x + 1

let x = ref 0 in
  λy. x := !x + 1; !x
Type Checking Rules

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

\[
\begin{align*}
    & A \vdash e : t \\
    & A \vdash \text{ref } e : \text{ref } t \\
    & A \vdash \text{} \\
    & A \vdash e_1 : \text{ref } t \\
    & A \vdash e_2 : t \\
    & A \vdash e_1 := e_2 : t
\end{align*}
\]
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`:
  
  - $e ::= \ldots | ()$
  - $t ::= \ldots | \text{unit}$

  
  $A \vdash e_1 : \text{ref} \ t \quad A \vdash e_2 : t$

  $A \vdash () : \text{unit} \quad A \vdash e_1 := e_2 : \text{unit}$
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 \textit{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)

\[ \langle S, (\lambda x. e_1) \rangle \rightarrow \langle S, (\lambda x. e_1) \rangle \]

\[ \langle S, e_1 \rangle \rightarrow \langle S', v_1 \rangle \quad \langle S', e_2 \rangle \rightarrow \langle S'', v_2 \rangle \]

\[ \langle S, e_1; e_2 \rangle \rightarrow \langle S'', v_2 \rangle \]

\[ \langle S, e \rangle \rightarrow \langle S', v \rangle \quad \text{loc fresh} \]

\[ \langle S, \text{ref } e \rangle \rightarrow \langle S[\text{loc} \mapsto v], \text{loc} \rangle \]
Operational Semantics (cont’d)

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

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

\[
A \vdash \lambda x : t.e : t \rightarrow t'
\]

Without type annotations, where do we get \( t \)?

- We’ll use type variables to stand for as-yet-unknown types
  - \( t ::= \alpha \mid \text{int} \mid t \rightarrow t \)

- 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

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

\[ A \vdash x : A(x) \]

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

\[ A \vdash \lambda x. e : \alpha \rightarrow t' \]

\[ \alpha \in \text{dom}(A) \]

\[ A \vdash e_1 : t_1 \quad A \vdash e_2 : t_2 \]

\[ t_1 = t_2 \rightarrow \beta \quad \beta \text{ fresh} \]

\[ A \vdash e_1 \ e_2 : \beta \]

"Generated" constraint
Example

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.
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 $\alpha \notin FV(t)$
  - (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 $C[\alpha \mapsto t]$ 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

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

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

- 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 …
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 $x$
Examples (cont’d)

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

- 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
Examples (cont’d)

- 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

\[ A \vdash e : t \quad \alpha \notin FV(A) \hspace{1cm} \]
\[ A \vdash e : \forall \alpha . t \]

• 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
  - $A[\alpha \mapsto u] \vdash e : t[\alpha \mapsto u]$
  - But then since $\alpha \notin \text{FV}(A)$, 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

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

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

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

\[
A \vdash \lambda x.e : \alpha \rightarrow t'
\]

\[
A \vdash e_1 : t_1 \quad A \vdash e_2 : t_2
\]

\[
t_1 = t_2 \rightarrow \beta \quad \beta \text{ fresh}
\]

\[
A \vdash e_1 e_2 : \beta
\]
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 = \( \lambda x.x \) in // \( x : \forall \alpha.\alpha \rightarrow \alpha \)

x 3; // \( x : \beta \rightarrow \beta, \quad \beta = \text{int} \)

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

• 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

  - $e ::= x \mid n \mid \lambda x.e \mid e \; e \mid \text{ref } e \mid !e \mid e := e$

  - $s ::= t \mid \forall \alpha.s$

  - $t ::= \alpha \mid \text{int} \mid t \to t \mid \text{ref } t$

• What if we try our standard rule?

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

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

• Example (due to Tofte)

```plaintext
let r = ref (\x.x) in  // r : \forall \alpha. ref (\alpha \rightarrow \alpha)

r := \lambda x.x+1;  // checks; use r at ref (int \rightarrow int)
(!r) true  // oops! checks; use r at ref(bool \rightarrow 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
  ▪ $v ::= x \mid n \mid \lambda x.e$
  ▪ $e ::= v \mid e \; e \mid \text{ref} \; e \mid \text{!}e \mid e := e$

\[
\begin{align*}
A \vdash v : t_1 & \quad A, x : \forall \alpha. t \vdash e_2 : t_2 \\
& \quad \alpha = \text{FV}(t) - \text{FV}(A) \\
\hline
A \vdash \text{let } x = v \text{ in } e_2 : t_2
\end{align*}
\]

▪ 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)