The Need for a Type System

• Not all programs accepted by a language’s grammar are actually defined (e.g., they do not have a normal form):
  - (1 (λx. λy. x y))
  - The (app) rule expects a function as the first argument, but statement expects a list, but here it has been given 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.”
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 \( t \) 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 (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 $e$ 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
  - All of this is similar to the store $s$ we've been using for the language of commands, but here is applied to typing

Type Rules

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

$A \vdash x : A(x)$

$A, x : t \vdash e : t'$

$A \vdash \lambda x : t. e : t \rightarrow t'$

$A \vdash e_1 : t \rightarrow t'$

$A \vdash e_2 : t$

$A \vdash e_1 \; e_2 : t'$
Example

A = - : int→int

\[ - \epsilon \text{dom}(A) \]

\[ A \vdash - : \text{int}\rightarrow\text{int} \quad A \vdash 3 : \text{int} \]

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

Another Example

A = + : int→int→int
B = A, x : int

\[ + \epsilon \text{dom}(B) \quad x \epsilon \text{dom}(B) \]

\[ B \vdash + : i\rightarrow i\rightarrow i \quad B \vdash x : i \]

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

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

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

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

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

- Our type rules are deterministic
  - For each syntactic form, only one possible rule
- They define a natural type checking algorithm
  - TypeCheck : type env × expression → type
    - TypeCheck(A, n) = int
    - TypeCheck(A, x) = if x in dom(A) then A(x) else fail
    - TypeCheck(A, λx:t.e) = TypeCheck((A, x:t), e)
    - TypeCheck(A, e₁ e₂) =
      - let t₁ → t₂ = TypeCheck(A, e₁) in
      - let t₁’ = TypeCheck(A, e₂) in
      - if t₁ = t₁’ then t₂ 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

```
(λx:t.e₁) v₂ → e₁[v₂/x]  e₁ → e₁'
```

```
e₂ → e₂'
vl e₂ → vl e₂'
```

```
e ::= v | x | e e
v ::= n | λx: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 \( n, \lambda x : t. e \) – 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 \( A \vdash e : t \) and \( e \rightarrow e' \) then \( A \vdash e' : t \)
- Proof by induction on \( e \)
  - Base cases \( n, x, \lambda x : t. e \) – Impossible, since these terms don’t reduce
    - But: what if we were using the nondeterministic semantics?
  - Induction. Assume \( A \vdash e_1 e_2 : t \) and \( e_1 e_2 \rightarrow e' \). Then we have \( A \vdash e_1 : t' \rightarrow t \) and \( A \vdash e_2 : t' \). (Why?)
  - Then there are three cases.
    - If \( e_1 \rightarrow e_1' \) then by induction \( A \vdash e_1' : t' \rightarrow t \), so \( e_1' e_2 \) has type \( t \) by the typing rule for applications
    - If reduction inside \( e_2 \), similar
Preservation, cont’d

- Otherwise \((\lambda x. e) v \rightarrow e[v\backslash x]\). Then we have

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

- Thus we have

- \(-A, x: t' \vdash e: t\)
- \(-A \vdash v: t'\)

- Then by the substitution lemma (next slide) we have

- \(-A \vdash e[v\backslash x]: t\)

- And so we have preservation

Substitution Lemma

- If \(-A \vdash v: t\) and \(-A, x: t \vdash e: t'\), then \(-A \vdash e[v\backslash x]: t'\)

- Proof: Induction on the structure of \(e\)

- For the lazy semantics, we’d have to prove something stronger

- If \(-A \vdash e1: t\) and \(-A, x: t \vdash e: t'\), then \(-A \vdash e[e1\backslash x]: t'\)
Soundness

• So we have

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

• 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)

• $e ::= ... | (e, e) | \text{fst } e \mid \text{snd } e$

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

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

• 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)**

- $e ::= ... | \text{in}_{t_2} e | \text{in}_{t_1} e$
- $(\text{case } e \text{ of } x_1 : t_1 \rightarrow e_1 | x_2 : t_2 \rightarrow e_2)$

\[
\begin{align*}
A \vdash e : t_1 & \quad A \vdash e : t_2 \\
A \vdash \text{in}_{t_2} e : t_1 + t_2 & \quad A \vdash \text{in}_{t_1} e : t_1 + t_2 \\
A \vdash e : t_1 + t_2 & \\
A, x_1 : t_1 \vdash e_1 : t & A, x_2 : t_2 \vdash e_2 : t \\
A \vdash (\text{case } e \text{ of } x_1 : t_1 \rightarrow e_1 | x_2 : t_2 \rightarrow e_2) : t
\end{align*}
\]

**Self Application and Types**

- Self application is not checkable in our system

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

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

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

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

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
    - unfold \( \mu \alpha.t = t[\mu \alpha.t, \alpha] \)
    - fold \( t[\mu \alpha.t, \alpha] = \mu \alpha.t \)
Fold-based Recursive Types

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

\[
\frac{A \vdash e : t[\mu \alpha.t\alpha]}{A \vdash \text{fold } e : \mu \alpha.t}
\]

\[
\frac{A \vdash e : \mu \alpha.t}{A \vdash \text{unfold } e : t[\mu \alpha.t\alpha]}
\]

ML Datatypes

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

- **type list = Int of int | Cons of int * int list**
  - Equivalent to $\mu\alpha.\text{int}+(\text{int} \times \alpha)$
- **(Int 3)** equivalent to
  - $\text{fold (inL}_{\text{int}}\times\mu\beta.\text{int}+(\text{int} \times \beta) \ 3)$
- **(Cons (2,(Int 3)))** equivalent to
  - $\text{fold (inR}_{\text{int}} (2, \text{fold (inL}_{\text{int}}\times\mu\beta.\text{int}+(\text{int} \times \beta) \ 3)))$
- **match e with Int x -> e1 | Cons x -> e2** same as
  - **case (unfold e)**
    - $x\text{int} \rightarrow e1$
    - $| x: \text{int} \times (\mu\beta.\text{int}+(\text{int} \times \beta)) \rightarrow e2$

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

• Type systems have nice properties
  • Type checking is straightforward (may need 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 (untyped) lambda calculus programs
  • Can you come up with an example?

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 *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
  - $\Lambda\alpha.\lambda x:\alpha.x : \forall\alpha.\alpha \rightarrow \alpha$
  - $\Lambda\alpha.\Lambda\beta.\lambda x:\alpha.\lambda y:\beta.x : \forall\alpha.\forall\beta.\alpha \rightarrow \beta \rightarrow \alpha$
  - $\Lambda\alpha.\Lambda\beta.\lambda x:\alpha.\lambda y:\beta.y : \forall\alpha.\forall\beta.\alpha \rightarrow \beta \rightarrow \beta$

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**

\[
\begin{align*}
A, \alpha \vdash e : t & \quad A \vdash e : \forall \alpha. t \\
A \vdash \lambda \alpha. e : \forall \alpha. t & \quad A \vdash e[t'] : t[t'\alpha]
\end{align*}
\]

- 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**

\[
\begin{align*}
(\land \alpha.e)[t] & \rightarrow e[t\alpha] & e \rightarrow e' & \text{(tapp-cong)} \\
(\forall \alpha. e)[t] & \rightarrow e[t\alpha] & e[t] & \rightarrow e'[t]
\end{align*}
\]

- 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
  - \( \text{FV}(\alpha) = \{\alpha\} \)
  - \( \text{FV}(c) = \emptyset \)
  - \( \text{FV}(t \rightarrow t') = \text{FV}(t) \cup \text{FV}(t') \)
  - \( \text{FV}(\forall \alpha . t) = \text{FV}(t) - \{\alpha\} \)

  - Look familiar?

**Substitution, Again**

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

- Define \( e[u\alpha] \) as
  - \( (\lambda x : t . e)[u\alpha] = \lambda x : t[u\alpha] . e[u\alpha] \)
  - \( (\Lambda \beta . e)[u\alpha] = \Lambda \beta . e[u\alpha] \), where \( \beta \neq \alpha \) and \( \beta \notin \text{FV}(u) \)
  - \( (e_1 e_2)[u\alpha] = e_1[u\alpha] \cdot e_2[u\alpha] \)
  - \( x[u\alpha] = x \) and \( n[u\alpha] = n \)
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
  \[ A, x:t \vdash e : t' \]
  \[ A \vdash \lambda x.t.e : t \rightarrow t' \]

• Without type annotations, where do we get \( t \)?
  • We’ll use type variables 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

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

"Generated" constraint

Example

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

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

- We can see if these equality constraints have a solution by using the following rewrite rules, which reduce a larger set of constraints to a smaller set:
  - $C \cup \{\text{int=int}\} \Rightarrow C$
  - $C \cup \{\alpha=\tau\} \Rightarrow C[\tau|\alpha]$
  - $C \cup \{\tau=\alpha\} \Rightarrow C[\tau|\alpha]$
  - $C \cup \{\tau_1 \rightarrow \tau_2=\tau_1' \rightarrow \tau_2'\} \Rightarrow C \cup \{\tau_1=\tau_1'\} \cup \{\tau_2=\tau_2'\}$
  - $C \cup \{\text{int=\tau} \rightarrow \tau_2\} \Rightarrow \text{unsatisfiable}$
  - $C \cup \{\tau_1 \rightarrow \tau_2=\text{int}\} \Rightarrow \text{unsatisfiable}$

- (To determine the final solution of type inference, we should also perform the substitutions in the second two rules to the original derivation)

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 $C[t\alpha]$, require that $\alpha \in \text{FV}(t)$
  - Called the “occurs check”

- 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 types are in a union-find forest
  - When a variable and a type 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. Thus we can read off the solution by reading the ECR for each set.
Example

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

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.,
    - \( \lambda x. x : \text{int} \to \text{int} \)
  - 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

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

• This rule is exacty 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 $x$
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 ($x$ 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

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

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

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

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

• We’d like to extend our algorithm to polymorphic type inference
  ▪ Perform 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 when at variable use with a polymorphic type

• Here is our new language
  ▪ $e ::= n | x | \lambda x.e | e e | \text{let } x = e \text{ in } e$
  ▪ $t ::= \alpha | \text{int} | t \rightarrow t$
  ▪ $s ::= t | \forall \alpha.s$
    ▪ These are type schemes.
  ▪ $A ::= \emptyset | A, x : s$

  ▪ 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

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

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

\[ A \vdash e_1 : t_1 \quad A \vdash e_2 : t_2 \]
\[ t_1 = t_2 \rightarrow \beta \quad \beta \text{ fresh} \]
\[ A \vdash e_1 \\ e_2 : \beta \]

Added Type Inference Rules

- At \textit{let}, generalize over all possible variables

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

- At variable uses, instantiate to all fresh types

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

- 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, \beta=int
  • x (\lambda y.y) // x : \gamma \rightarrow \gamma, \gamma=\delta \rightarrow \delta

• This would be untypable in a monomorphic type system
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

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

- \(! (\text{ref } 0)\)

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

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

Type Checking Rules

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

\[
\begin{align*}
A \vdash e : t \\
\hline
A \vdash \text{ref } e : \text{ref } t
\end{align*}
\]

\[
\begin{align*}
A \vdash e : t \\
\hline
A \vdash !e : t
\end{align*}
\]

\[
\begin{align*}
A \vdash e_1 : \text{ref } t \\
A \vdash e_2 : t \\
\hline
A \vdash e_1 := e_2 : t
\end{align*}
\]
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 | unit

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

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 | v e | \text{ref } e | !e | e := e
Operational Semantics (cont’d)

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

\[ \langle S, e_1 \rangle \rightarrow \langle S', \nu_1 \rangle \quad \langle S', e_2 \rangle \rightarrow \langle S'', \nu_2 \rangle \]
\[ \langle S, e_1; e_2 \rangle \rightarrow \langle S'', \nu_2 \rangle \]

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

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

Naive Generalization is Unsound

• Example (due to Tofte)

  - let $r = \text{ref } (\lambda x . x)$ in \hspace{1cm} // $r : \forall \alpha . \text{ref } (\alpha \rightarrow \alpha)$
  - $r ::= \lambda x . x + 1$; \hspace{1cm} // checks; use $r$ at $\text{ref } (\text{int } \rightarrow \text{int})$
  - $(!r) \text{ true}$ \hspace{1cm} // oops! checks; use $r$ at $\text{ref } (\text{bool } \rightarrow \text{bool})$

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

• Nobody realized this problem for a long time
Solution: The Value Restriction

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

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

- 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)