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?

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 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, x: $\alpha$ $\vdash x: \alpha$$**
- **A $\vdash n : \text{int}$$**
- **A $\vdash x : A(x)$$**
- **A $\vdash e_1 : t_1 \quad A $\vdash e_2 : t_2$$**
- **$t_1 = t_2 \rightarrow \beta$$** (fresh)
- **A $\vdash e_1 \; e_2 : \beta$$**

“Generated” constraint

Example

- **A, x: $\alpha$ $\vdash x: \alpha$$**
- **A $\vdash (\lambda x. x) : \alpha \rightarrow \alpha$$**
- **A $\vdash 3 : \text{int}$$**
- **$\alpha \rightarrow \alpha = \text{int} \rightarrow \beta$$**
- **A $\vdash (\lambda x. x) \; 3 : \beta$$**

- 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:
  - \( C \cup \{ \text{int=\text{int}} \} \Rightarrow C \)
  - \( C \cup \{ \alpha=t \} \Rightarrow C[t\alpha] \)
  - \( C \cup \{ t=\alpha \} \Rightarrow C[t\alpha] \)
  - \( C \cup \{ t_1 \rightarrow t_2=t_1' \rightarrow t_2' \} \Rightarrow C \cup \{ t_1=t_1' \} \cup \{ t_2=t_2' \} \)
  - \( C \cup \{ \text{int=t} \rightarrow t_2 \} \Rightarrow \text{unsatisfiable} \)
  - \( C \cup \{ t_1 \rightarrow t_2=\text{int} \} \Rightarrow \text{unsatisfiable} \)

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 \( C[t\alpha] \), require that \( \alpha \notin \text{FV}(t) \)
  - (Except if \( t = a \), in which case there’s no recursion in the types, so unification should succeed)
- In practice, it may better to allow \( \alpha \notin \text{FV}(t) \) 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 (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**

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

\[
A \vdash e : t[t'[\alpha]]
\]

- 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**

\[
\begin{align*}
A, x: \alpha & \vdash e : \alpha \\
A & \vdash \lambda x.x : \alpha \rightarrow \alpha \\
A, x: (i \rightarrow i) & \vdash x : (i \rightarrow i)
\end{align*}
\]

- 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)

- 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

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

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 \( x \)

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

Generalization Rule

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

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

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
  - \( A[u \backslash \alpha] \vdash e : t[u \backslash \alpha] \)
  - But then since \( \alpha \notin \text{FV}(A) \), 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
  - 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
  - \( e ::= n \mid x \mid \lambda x.e \mid e_1 e_2 \mid \text{let } x = e \text{ in } e \)  
  - \( t ::= \alpha \mid \text{int} \mid t \to t \)  
  - \( s ::= t \mid \forall \alpha.s \)  

  - 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

\[
\begin{align*}
\frac{}{A \vdash n : \text{int}} \\
\frac{A, x: \alpha \vdash e : t’ \quad \alpha \text{ fresh}}{A \vdash \lambda x.e : \alpha \to t’}
\end{align*}
\]

\[
\begin{align*}
\frac{A \vdash e_1 : t_1 \quad A \vdash e_2 : t_2}{A \vdash \text{let } x = e_1 \text{ in } e_2 : t_2}
\end{align*}
\]

\[
\begin{align*}
\frac{A = \text{FV}(t_1) - \text{FV}(A)}{A \vdash \text{let } x = e_1 \text{ in } e_2 : t_2}
\end{align*}
\]

New Type Inference Rules

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

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

  - Here the \( \alpha \) 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
  
  \[
  \text{let } x = \lambda x. x \text{ in} \\
  x \ 3; \\
  x \ (\lambda y. y)
  \]

  \[
  x : \forall \alpha. \alpha \rightarrow \alpha \\
  x : \beta \rightarrow \beta, \ \beta = \text{int} \\
  x : \gamma \rightarrow \gamma, \ \gamma = \delta \rightarrow \delta
  \]

- This would be untypable in a monomorphic type system

Kinds of Polymorphism

- We’ve just seen parametric polymorphism
  - 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 | n | \lambda x. e | e e | ref e | !e | e := e
  - s ::= t | \forall \alpha. s
  - t ::= \alpha | \text{int} | t \rightarrow t | \text{ref } t

- What if we try our standard rule?
  \[
  A \vdash e : t_1 \\
  A, x : \forall \alpha. t_1 \vdash e_2 : t_2 \\
  \alpha = \text{FV}(t_1) - \text{FV}(A)
  \]

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

• Example (due to Tofte)
  let r = ref (\x.x) in   // r : \forall \alpha. ref (\alpha \to \alpha)
  r := \lambda x.x+1;   // checks; use r at ref (int \to int)
  (!r) true       // oops! checks; use r at ref(bool \to 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 | n | \lambda x.e
  - e ::= v | e e | ref e | !e | e := e

\[
\frac{v : t_1 \quad e_2 : t_2}{\alpha = \text{FV}(t) - \text{FV}(A)}\]

  A ⊢ let x = v in e_2 : t_2

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

Benefits of Type Inference

• Handles higher-order functions
• Handles data structures smoothly
• Works in infinite domains
  - Set of types is unlimited
• No forward/backward distinction
• Polymorphism provides context-sensitivity

Drawbacks to Type Inference

• Flow-insensitive
  - Types are the same at all program points
  - May produce coarse results
  - Type inference failure can be hard to understand

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