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

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

- Without type annotations, where do we get t?
  - We'll use *type variables* to stand for as-yet-unknown types
    - $t ::= \alpha \mid int \mid t \to t$
  - 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

$$\frac{}{A \vdash n : int} \qquad \frac{x \in dom(A)}{A \vdash x : A(x)}$$

$$\frac{A, x{:}\alpha \vdash e : t' \quad \alpha \text{ fresh}}{A \vdash \lambda x.e : \alpha \to t'} \qquad \frac{A \vdash e1 : t1 \quad A \vdash e2 : t2 \quad \boxed{t1 = t2 \to \beta} \quad \beta \text{ fresh}}{A \vdash e1\ e2 : \beta}$$

"Generated" constraint

Example

$$\frac{\dfrac{A, x{:}\alpha \vdash x : \alpha}{A \vdash (\lambda x.x) : \alpha \to \alpha} \quad A \vdash 3 : int \quad \alpha \to \alpha = int \to \beta}{A \vdash (\lambda x.x)\ 3 : \beta}$$

- We collect all constraints appearing in the derivation into some set C to be solved
- Here, C contains just $\alpha \to \alpha = int \to \beta$
  - Solution: $\alpha = int = \beta$
- Thus this program is typable, and we can derive a typing by replacing α and β by 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
  - $C \cup \{int=int\} \Rightarrow C$
  - $C \cup \{\alpha=t\} \Rightarrow C[t\backslash\alpha]$
  - $C \cup \{t=\alpha\} \Rightarrow C[t\backslash\alpha]$
  - $C \cup \{t1\to t2=t1'\to t2'\} \Rightarrow C \cup \{t1=t1'\} \cup \{t2=t2'\}$
  - $C \cup \{int=t1\to t2\} \Rightarrow$ unsatisfiable
  - $C \cup \{t1\to t2=int\} \Rightarrow$ 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 $C[t\backslash\alpha]$, 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 $C[t\backslash\alpha]$ 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

$\alpha = int \to \beta$

$\gamma = int \to int$

$\alpha = \gamma$

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.,
    - $\lambda x.x : int \to int$
  - 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
  - α 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:
  - $t ::= \alpha \mid int \mid t \rightarrow t \mid \forall\alpha.t$
  - $e ::= n \mid x \mid \lambda x.e \mid e\ e$
- Type inference will automatically infer where to generalize a term, to introduce polymorphic types, and where to instantiate them

Instantiation

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

- 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 α in the type of expression e?
- Answer: Whenever we can redo the typing proof for e, choosing α to be anything we want, and still have a valid typing proof.

Examples

$$\frac{A, x{:}\alpha \vdash e : \alpha}{A \vdash \lambda x.x : \alpha\rightarrow\alpha}$$

$$\frac{A, x{:}int \vdash x : int}{A \vdash \lambda x.x : int\rightarrow int}$$

$$\frac{A, x{:}(i\rightarrow i) \vdash x : (i\rightarrow i)}{A \vdash \lambda x.x : (i\rightarrow i)\rightarrow(i\rightarrow i)}$$

- The choice of the type of x is purely local to type checking $\lambda x.x$
  - There is no interaction with the outside environment
  - Thus we can generalize the type of x

Examples (cont'd)

$$\frac{A, x{:}int \vdash x : int}{A \vdash \lambda x.x{+}3 : int \rightarrow int}$$

- 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)

$$\frac{A, y{:}\alpha,\ x{:}\alpha \vdash \text{if } p \text{ then } x \text{ else } y : \alpha}{A, y{:}\alpha \vdash \lambda x.\text{if } p \text{ then } x \text{ else } y :}$$

$$\Downarrow$$

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

- 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

$$\frac{A \vdash e : t \qquad \alpha \notin FV(A)}{A \vdash e : \forall\alpha.t}$$

- 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
  - $A \vdash e : t$ and $\alpha \notin FV(A)$
- 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
  - 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*
  - A ::= ∅ | A,x:s
  - 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

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

$$\frac{A, x{:}\alpha \vdash e : t' \qquad \alpha \text{ fresh}}{A \vdash \lambda x.e : \alpha \to t'}$$

$$\frac{A \vdash e1 : t1 \qquad A \vdash e2 : t2 \qquad t1 = t2 \to \beta \qquad \beta \text{ fresh}}{A \vdash e1\ e2 : \beta}$$

New Type Inference Rules

- At let, generalize over all possible variables

$$\frac{A \vdash e1 : t1 \qquad A, x{:}\forall\vec{\alpha}.t1 \vdash e2 : t2 \qquad \vec{\alpha} = FV(t1) - FV(A)}{A \vdash \text{let } x = e1 \text{ in } e2 : t2}$$

- At variable uses, instantiate to all fresh types

$$\frac{A(x) = \forall\vec{\alpha}.t \qquad \vec{\beta} \text{ fresh}}{A \vdash x : t[\vec{\beta}\backslash\vec{\alpha}]}$$

- Here the $\vec{\alpha}$ 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 = λx.x in // x : $\forall\alpha.\alpha\to\alpha$

  x 3; // x : $\beta\to\beta$, β=int

  x (λy.y) // x : $\gamma\to\gamma$, $\gamma=\delta\to\delta$

- 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 | λx.e | e e | ref e | !e | e := e
  - s ::= t | $\forall\alpha$.s
  - t ::= α | int | t $\to$ t | ref t
- What if we try our standard rule?

$$\frac{A \vdash e1 : t1 \quad A, x{:}\forall\vec{\alpha}.t1 \vdash e2 : t2 \quad \vec{\alpha}=FV(t1)-FV(A)}{A \vdash \text{let } x = e1 \text{ in } e2 : t2}$$

Naive Generalization is Unsound

- Example (due to Tofte)

  let r = ref (λx.x) in // r : ∀α.ref (α→α)

  r := λx.x+1; // checks; use r at ref (int → int)

  (!r) true // oops! checks; use r at ref(bool →bool)

- α 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 | λx.e
  - e ::= v | e e | ref e | !e | e := e

$$\frac{A \vdash v : t1 \qquad A, x{:}\forall\alpha.t \vdash e2 : t2 \qquad \alpha = FV(t) - FV(A)}{A \vdash \text{let } x = v \text{ in } e2 : t2}$$

  - 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)