COMP 150-AVS
Fall 2018

Types and Type Inference

The Need for a Type System

- 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!

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 | x | λx:t.e | e e
  - Functions include the type of their argument
  - We don't really need this, but it will come in handy

- t ::= int | t → t
  - t1 → t2 is a the type of a function that, given an argument of type t1, returns a result of type t2
    - t1 is the *domain*, and t2 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

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

$$\frac{x \in \mathrm{dom}(A)}{A \vdash x : A(x)}$$

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

$$\frac{A \vdash e1 : t \to t' \qquad A \vdash e2 : t}{A \vdash e1\ e2 : t'}$$

Example

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

$$\dfrac{\dfrac{- \in \mathrm{dom}(A)}{A \vdash - : \mathrm{int} \to \mathrm{int}} \qquad A \vdash 3 : \mathrm{int}}{A \vdash -\ 3 : \mathrm{int}}$$

Another Example

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

$B = A, x : \text{int}$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{+ \in \mathrm{dom}(B)}{B \vdash + :} \quad \dfrac{x \in \mathrm{dom}(B)}{B \vdash x : \text{i}}}{B \vdash + \ x : \text{int} \to \text{int}} \quad B \vdash 3 : \text{int}}{B \vdash + \ x \ 3 : \text{int}}}{A \vdash (\lambda x{:}\text{int}.+ \ x \ 3) : \text{int} \to \text{int}} \quad A \vdash 4 : \text{int}}{A \vdash (\lambda x{:}\text{int}.+ \ x \ 3) \ 4 : \text{int}}$$

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) = t → (TypeCheck((A, x:t), e))
TypeCheck(A, e1 e2) =
    let t1 = TypeCheck(A, e1) in
    let t2 = TypeCheck(A, e2) in
        if dom(t1) = t2 then range(t1) 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*

$$\frac{}{(\lambda x{:}t.e1)\ v2 \rightarrow e1[x \mapsto v2]}$$

$$\frac{e1 \rightarrow e1'}{e1\ e2 \rightarrow e1'\ e2}$$

$$\frac{e2 \rightarrow e2'}{v1\ e2 \rightarrow v1\ e2'}$$

$e ::= v \mid x \mid e\ e$

$v ::= n \mid \lambda x{:}t.e$ *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, λx.e – these are values, so we're done
  - Base case x – can't happen (empty type environment)
  - Inductive case e1 e2 – If e1 is not a value, then by induction we can evaluate it, so we're done, and similarly for e2. Otherwise both e1 and e2 are values. Inspection of the type rules shows that e1 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 $e \to e'$ then $\vdash e' : t$
- Proof by induction on $e \to e'$
  - Induction (easier than the base case!). Expression e must have the form $e1\ e2$.
  - Assume $\vdash e1\ e2 : t$ and $e1\ e2 \to e'$. Then we have $\vdash e1 : t' \to t$ and $\vdash e2 : t'$.
  - Then there are three cases.
    - If $e1 \to e1'$, then by induction $\vdash e1' : t' \to t$, so $e1'\ e2$ has type t
    - If reduction inside $e2$, similar

Preservation, cont'd

- Otherwise $(\lambda x{:}t'.e)\ v \rightarrow e[x \mapsto v]$. Then we have

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

  - Thus we have
    - $x : t' \vdash e : t$
    - $\vdash v : t'$
  - Then by the substitution lemma (not shown) we have
    - $\vdash e[x \mapsto v] : t$
  - 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 $A \vdash e1 : t$ and $A, x{:}t \vdash e : t'$, then $A \vdash e[x \mapsto e1] : t'$

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"

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 α, 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 ::= α | int | t → t | ∀α.t
  - e ::= n | x | λx.e | e e | Λα.e | 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
  - $\Lambda\alpha.\lambda x{:}\alpha.x : \forall\alpha.\alpha\to\alpha$
  - $\Lambda\alpha.\Lambda\beta.\lambda x{:}\alpha.\lambda y{:}\beta.x : \forall\alpha.\forall\beta.\alpha\to\beta\to\alpha$
  - $\Lambda\alpha.\Lambda\beta.\lambda x{:}\alpha.\lambda y{:}\beta.y : \forall\alpha.\forall\beta.\alpha\to\beta\to\beta$

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:
  - $(\Lambda\alpha.\lambda x{:}\alpha.x)[t1] : t1 \to t1$
  - $(\Lambda\alpha.\lambda x{:}\alpha.x)[t2] : t2 \to t2$

- 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

Type Rules

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

- 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 α to A; we could (should?) use this to ensure types are well-formed

Small-step Semantics Rules

$$\frac{}{(\Lambda\alpha.e)[t] \rightarrow e[\alpha\mapsto t]}\ \text{(type-app)} \qquad \frac{e \rightarrow e'}{e[t] \rightarrow e'[t]}\ \text{(tapp-cong)}$$

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

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 $\beta\notin FV(u)$

- 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 $\beta\notin FV(u)$
  - $(e1\ e2)[\alpha\mapsto u] = e1[\alpha\mapsto u]\ e2[\alpha\mapsto u]$
  - $x[\alpha\mapsto u] = x$ and $n[\alpha\mapsto u] = n$

Type Inference

- Let’s reconsider the simply typed lambda calculus with integers
  - e ::= n | x | λx:t.e | 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

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

- 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

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

$$\frac{x \in \mathrm{dom}(A)}{A \vdash x : A(x)}$$

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

$$\frac{\begin{array}{c} A \vdash e1 : t1 \qquad A \vdash e2 : t2 \\ \boxed{t1 = t2 \to \beta} \qquad \beta \text{ fresh} \end{array}}{A \vdash e1\ e2 : \beta}$$

“Generated” constraint

Example

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

- We collect all constraints appearing in the derivation into some set C to be solved
- Here, C contains just $\alpha \to \alpha = \text{int} \to \beta$
    - Solution: $\alpha = \text{int} = \beta$
- 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
  - $C \cup \{int=int\} \Rightarrow C$
  - $C \cup \{\alpha=t\} \Rightarrow C[\alpha \mapsto t]$
  - $C \cup \{t=\alpha\} \Rightarrow C[\alpha \mapsto t]$
  - $C \cup \{t1 \to t2 = t1' \to t2'\} \Rightarrow C \cup \{t1=t1'\} \cup \{t2=t2'\}$
  - $C \cup \{int=t1 \to t2\} \Rightarrow$ 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 $C[\alpha \mapsto t]$, 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

$$\alpha = int \rightarrow \beta$$

$$\gamma = int \rightarrow int$$

$$\alpha = \gamma$$

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.,
    - λx.x : int → 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:
  - $t ::= \alpha \mid \mathrm{int} \mid t \to t \mid \forall\alpha.t$
  - $e ::= n \mid x \mid \lambda x.e \mid e\ e$
- 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 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

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

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

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

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

Examples (cont’d)

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

- 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)

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

$$\Downarrow$$

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

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

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

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

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

New Type Inference Rules

- At let, generalize over all possible variables

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

- At variable uses, instantiate to all fresh types

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

  - 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 : ∀α.α→α

    x 3;               // x : β→β,   β=int
    x (λy.y)           // x : γ→γ,   γ=δ→δ
```

- 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 | ∀α.s
  - t ::= α | int | t → t | ref t
- What if we try our standard rule?

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

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)