Type Inference and the Hindley-Milner Type System

Armando Solar-Lezama
Computer Science and Artificial Intelligence Laboratory
M.I.T.

With slides from Arvind. Used with permission.

September 29, 2011

Type Inference

- Consider the following expression
  - (λf:int→ int. f 5) (λx:int. x + 1)
    - Is it well typed in F_1?

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}$$

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash (\lambda x : \tau_1\ e) : \tau_1 \to \tau_2}$$

$$\frac{\Gamma \vdash e_1 : \tau' \to \tau \quad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash e_1 e_2 : \tau}$$

$$\frac{}{\Gamma \vdash N : int}$$

$$\frac{\Gamma \vdash e1 : int \qquad \Gamma \vdash e2 : int}{\Gamma \vdash e1 + e2 : int}$$

Type Inference

- There wasn't a single point in the derivation where we had to look at the type labels in order to know what rule to apply!
  - we could have written the derivation without the labels

- The labels helped us determine the actual types for all the τs in the typing rules.
  - we could have figured these out even without the labels
  - this is the key idea behind type inference!

Type Inference Strategy 1

- 1. Use the typing rules to define constraints on the possible types of expressions
- 2. Solve the resulting constraint system

Deducing Types

```
twice f x = f (f x)
```

What is the most "general type" for twice?

1. Assign types to every subexpression

   ```
   x :: t0              f :: t1
   f x :: t2      f (f x) :: t3
   ```

   ⇒ `twice :: t1 -> t0 -> t3` **?**

2. Set up the constraints

   `t1 = t0 -> t2` because of `(f x)`

   `t1 = t2 -> t3` because of `f (f x)`

3. Resolve the constraints

   `t0 -> t2 = t2 -> t3`

   ⇒ `t0 = t2 and t2 = t3` ⇒ `t0 = t2 = t3`

   ⇒ `twice :: (t0 -> t0) -> t0 -> t0`

The language of Equality Constraints

- Consider the following Language of Constraints

$$C ::= \tau_1 = \tau_2 \mid C \wedge C \mid \exists \tau . C$$

- Constraints in this language have a lot of good properties
  - Nice and compositional
  - Linear time solution algorithm

Building Constraints from Typing Rules

- Notation

$$[\![Judgment]\!] = Constraints$$

  - The constraints on the right ensure that the judgment on the left holds
  - This mapping is defined recursively.

- Base cases

$$[\![\Gamma \vdash x : \tau]\!] = \Gamma(x) = \tau \qquad [\![\Gamma \vdash N : \tau]\!] = \mathrm{int} = \tau$$

- Inductive Cases

$$[\![\Gamma \vdash e_1 e_2 : \tau]\!] = \exists \mathrm{a} (\, [\![\Gamma \vdash \mathrm{e}_1 : \mathrm{a} \to \tau]\!] \wedge [\![\Gamma \vdash \mathrm{e}_2 : \mathrm{a}]\!] \,)$$

$$[\![\Gamma \vdash \lambda x. e : \tau]\!] = \exists \mathrm{a}_1 \mathrm{a}_2 . (\, [\![\Gamma ; \mathrm{x} : \mathrm{a}_1 \vdash \mathrm{e} : \mathrm{a}_2]\!] \wedge \tau = \mathrm{a}_1 \to \mathrm{a}_2 \,)$$

$$[\![\Gamma \vdash e_1 + e_2 : \tau]\!] = [\![\Gamma \vdash e_1 : int]\!] \wedge [\![\Gamma \vdash e_2 : int]\!] \wedge \tau = int$$

Back to our example

$(\lambda f.\ f\ 5)\ (\lambda x.\ x + 1)$

$$[\![\Gamma \vdash x : \tau]\!] = \Gamma(x) = \tau$$

$$[\![\Gamma \vdash e_1 e_2 : \tau]\!] = \exists a(\ [\![\Gamma \vdash e_1 : a \to \tau]\!] \wedge [\![\Gamma \vdash e_2 : a]\!]\ )$$

$$[\![\Gamma \vdash N : \tau]\!] = \mathrm{int} = \tau$$

$$[\![\Gamma \vdash \lambda x.e : \tau]\!] = \exists a_1 a_2.(\ [\![\Gamma; x : a_1 \vdash e : a_2]\!] \wedge \tau = a_1 \to a_2\ )$$

$$[\![\Gamma \vdash e_1 + e_2 : \tau]\!] = [\![\Gamma \vdash e_1 : int]\!] \wedge [\![\Gamma \vdash e_2 : int]\!] \wedge \tau = int$$

Equality and Unification

- What does it mean for two types τ_a and τ_b to be equal?
  - *Structural Equality*

    Suppose $\tau_a = \tau_1 \text{ -> } \tau_2$

    $\tau_b = \tau_3 \text{ -> } \tau_4$

    Is $\tau_a = \tau_b$?

    iff $\tau_1 = \tau_3$ and $\tau_2 = \tau_4$

- Can two types be made equal by choosing appropriate substitutions for their type variables?
  - *Robinson's unification algorithm*

    Suppose $\tau_a = t_1 \text{ -> Bool}$

    $\tau_b = \text{Int -> } t_2$

    Are τ_a and τ_b unifiable ?

    if t_1= Int and t_2= Bool

    Suppose $\tau_a = t_1 \text{-> Bool}$

    $\tau_b = \text{Int -> Int}$

    Are τ_a and τ_b unifiable ?

    No

Simple Type Substitutions
needed to define type unification

Types

$\tau ::= \iota$ base types (Int, Bool ..)

$\quad |\ t$ type variables

$\quad |\ \tau_1 \text{ -> } \tau_2$ Function types

A substitution is a map

S : Type Variables $\rightarrow$ Types

$S = [\tau_1 / t_1, \ldots, \tau_n / t_n]$

$\tau' = S\,\tau$ $\quad\tau'$ is a *Substitution Instance of* τ

Example:

$S = [(t \text{ -> } Bool) / t_1]$

$S\,(t_1 \text{ -> } t_1) = \boxed{(t \text{ -> } Bool) \text{ -> } (t \text{ -> } Bool)}$?

Substitutions can be *composed,* i.e., $S_2\,S_1$

Example:

$S_1 = [(t \text{ -> } Bool) / t_1]\ ;\ S_2 = [Int / t]$

$S_2\,S_1\,(t_1 \text{ -> } t_1) = S_2\,((t \text{ -> } Bool) \text{ -> } (t \text{ -> } Bool))$?

$= (Int \text{ -> } Bool) \text{ -> } (Int \text{ -> } Bool)$

Unification

An essential subroutine for type inference

Unify(τ_1, τ_2) tries to unify τ_1 and τ_2 and returns a substitution if successful

def Unify(τ_1, τ_2) =
  case (τ_1, τ_2) *of*
    (τ_1, t_2) = [τ_1 / t_2] provided $t_2 \notin FV(\tau_1)$
    (t_1, τ_2) = [τ_2 / t_1] provided $t_1 \notin FV(\tau_2)$
    (ι_1, ι_2) = *if* (eq? ι_1 ι_2) *then* [ ]
                else *fail*
    (τ_{11}->τ_{12}, τ_{21} ->τ_{22})
      = *let* S_1=Unify(τ_{11}, τ_{21})
          S_2=Unify($S_1(\tau_{12}), S_1(\tau_{22})$)
        in $S_2 S_1$

otherwise = *fail*

Does the order matter? No

Type inference strategy 2

- Like strategy 1, but we solve the constraints as we see them
  - Build the substitution map incrementally

Simple Inference Algorithm

W(TE, e) returns (S,τ) such that $S\ (TE) \vdash e : \tau$

This is just Γ (it's hard to write Γ in code)

The type environment TE records the most general type of each identifier while the substitution S records the changes in the type variables

Def W(TE, e) =
    Case e *of*
        x = ...
        n = ...
        λx.e = ...
        $(e_1\ e_2)$ = ...
    ...

Simple Inference Algorithm *(cont-1)*

u's represent new type variables

Def W(TE, e) =
Case e *of*

$[\![\Gamma \vdash N : \tau]\!] = \text{int} = \tau$

N = ({}, Typeof(N))

$[\![\Gamma \vdash x : \tau]\!] = \Gamma(x) = \tau$

x = *if* (x $\notin$ Dom(TE)) *then* Fail
else *let* τ = TE(x);
in ({}, τ)

$[\![\Gamma \vdash \lambda x. e : \tau]\!] = \exists a_1 a_2 . ([\![\Gamma ; x : a_1 \vdash e : a_2]\!] \wedge \tau = a_1 \rightarrow a_2)$

λx.e = *let* (S_1, τ_2)= W(TE + { x : u }, e)
in (,)

$[\![\Gamma \vdash e_1 e_2 : \tau]\!] = \exists a ([\![\Gamma \vdash e_1 : a \rightarrow \tau]\!] \wedge [\![\Gamma \vdash e_2 : a]\!])$

(e_1 e_2) = *let* (S_1, τ_1) = W(TE, e_1);
(S_2, τ_2) = W(S_1(TE), e_2);
S_3 = Unify($S_2(\tau_1)$, τ_2 -> u);
in ($S_3 S_2 S_1$, S_3(u))

Simple Inference Algorithm *(cont-1)*

u's represent new type variables

Def W(TE, e) =
  Case e *of*

c = ({}, Typeof(c))

x = *if* ($x \notin$ Dom(TE)) *then* Fail
  else *let* τ = TE(x);
    in ({}, τ)

λx.e = *let* (S_1, τ_1) = W(TE + { x : u }, e)
  in $(S_1, S_1(u) \text{ -> } \tau_1)$

$(e_1\ e_2)$ = *let* (S_1, τ_1) = W(TE, e_1);
  (S_2, τ_2) = W(S_1(TE), e_2);
  S_3 = Unify($S_2(\tau_1)$, τ_2 -> u);
  in $(S_3 S_2 S_1, S_3(u))$

let x = e_1 *in* e_2
  = *let* (S_1, τ_1) = W(TE + {x : u}, e_1);
  S_2 = Unify($S_1(u)$, τ_1);
  (S_3, τ_2) = W($S_2 S_1$(TE) + {x : τ_1}, e_2);
  in $(S_3 S_2 S_1, \tau_2)$

Example

Def W(TE, e) =

Case e *of*

...

λx.e = *let* $(S_1, \tau_1) = W(TE + \{ x : u \}, e)$
in $(S_1, S_1(u) \to \tau_1)$

$(e_1\ e_2)$ = *let* $(S_1, \tau_1) = W(TE, e_1)$;
$(S_2, \tau_2) = W(S_1(TE), e_2)$;
$S_3 = Unify(S_2(\tau_1), \tau_2 \to u)$;
in $(S_3 S_2 S_1, S_3(u))$

$$W(\{f : u_0\}, f) = (\emptyset, u_0) \qquad W(\{f : u_0\}, 5) = (\emptyset, Int)$$

$$Unify(u_0, Int \to u_1) =$$

$$W(\{f : u_0\}, f\ 5) =$$

$$W\big(\emptyset, (\lambda f. f\ 5)\big) =$$

$$W(\emptyset, (\lambda f. f\ 5)(\lambda x. x))$$

Example

def Unify(τ_1, τ_2) =
case (τ_1, τ_2) *of*

(τ_1, t_2) = [τ_1 / t_2] provided $t_2 \notin FV(\tau_1)$

(t_1, τ_2) = [τ_2 / t_1] provided $t_1 \notin FV(\tau_2)$

(ι_1, ι_2) = *if* (eq? ι_1 ι_2) *then* [ ]
else fail

(τ_{11}->τ_{12}, τ_{21} ->τ_{22})
= *let* S_1=Unify(τ_{11}, τ_{21})
S_2=Unify($S_1(\tau_{12})$, $S_1(\tau_{22})$)

in $S_2 S_1$

$$W(\{f\colon u_0\}, f) = (\emptyset, u_0) \qquad W(\{f\colon u_0\}, 5) = (\emptyset, Int)$$

$$Unify(u_0, Int \rightarrow u_1) = [(Int \rightarrow u_1)/u_0]$$

$$W(\{f\colon u_0\}, f\ 5) =$$

$$W\big(\emptyset, (\lambda f. f\ 5)\big) =$$

$$W(\emptyset, (\lambda f. f\ 5)(\lambda x. x))$$

Example

Def W(TE, e) =

Case e *of*

...

λx.e = *let* (S_1, τ_1) = W(TE + { x : u }, e)
in $(S_1, S_1(u) \text{ -> } \tau_1)$

$(e_1\ e_2)$ = *let* $(S_1, \tau_1) = W(TE, e_1)$;
$(S_2, \tau_2) = W(S_1(TE), e_2)$;
$S_3 = Unify(S_2(\tau_1), \tau_2 \text{ -> } u)$;
in $(S_3\, S_2\, S_1, S_3(u))$

$$W(\{f : u_0\}, f) = (\emptyset, u_0) \qquad W(\{f : u_0\}, 5) = (\emptyset, Int)$$

$$Unify(u_0, Int \rightarrow u_1) = [(Int \rightarrow u_1)/u_0]$$

$$W(\{f : u_0\}, f\ 5) = ([Int \rightarrow u_1/u_0], u_1)$$

$$W\big(\emptyset, (\lambda f.\, f\ 5)\big) = ([(Int \rightarrow u_1)/u_0], (Int\ \rightarrow u_1) \rightarrow u_1)$$

$$W(\emptyset, (\lambda f.\, f\ 5)(\lambda x.\, x))$$

Example

Def W(TE, e) =

Case e *of*

...

λx.e = *let* $(S_1, \tau_1) = W(TE + \{ x : u \}, e)$
in $(S_1, S_1(u) \text{ -> } \tau_1)$

$(e_1\ e_2)$ = *let* $(S_1, \tau_1) = W(TE, e_1)$;
$(S_2, \tau_2) = W(S_1(TE), e_2)$;
$S_3 = Unify(S_2(\tau_1), \tau_2 \text{ -> } u)$;
in $(S_3\, S_2\, S_1, S_3(u))$

$$W\big(\emptyset, (\lambda f. f\ 5)\big) = ([(Int \to u_1)/u_0], (Int\ \to u_1) \to u_1)$$

$$W\big(\emptyset, (\lambda x. x)\big) = \ (\emptyset, u_3 \to u_3)$$

$$Unify\big((Int\ \to u_1) \to u_1, (u_3 \to u_3) \to u_4\big) =$$

$$W(\emptyset, (\lambda f. f\ 5)(\lambda x. x))$$

Example

def Unify(τ_1, τ_2) =
case (τ_1, τ_2) *of*

(τ_1, t_2) = [τ_1 / t_2] provided $t_2 \notin FV(\tau_1)$

(t_1, τ_2) = [τ_2 / t_1] provided $t_1 \notin FV(\tau_2)$

(ι_1, ι_2) = *if* (eq? ι_1 ι_2) *then* [ ]
else *fail*

(τ_{11}->τ_{12}, τ_{21} ->τ_{22})
= *let* S_1=Unify(τ_{11}, τ_{21})
S_2=Unify($S_1(\tau_{12})$, $S_1(\tau_{22})$)

in $S_2 S_1$

$$Unify\big((Int \to u_1),(u_3 \to u_3)\big) = [Int/u_3\,;Int/u_1]$$

$$Unify(Int, u_4) = [Int/u_4]$$

$$Unify\big((Int \to u_1) \to u_1,(u_3 \to u_3) \to u_4\big) = [Int/u_3\,;Int/u_1\,;Int/u_4]$$

$$W(\emptyset,(\lambda f.\,f\ 5)(\lambda x.\,x))$$

Example

Def W(TE, e) =

Case e *of*

...

λx.e = *let* (S_1, τ_1) = W(TE + { x : u }, e)
in $(S_1, S_1(u) \to \tau_1)$

$(e_1\ e_2)$ = *let* $(S_1, \tau_1) = W(TE, e_1)$;
$(S_2, \tau_2) = W(S_1(TE), e_2)$;
$S_3 = Unify(S_2(\tau_1), \tau_2 \to u)$;
in $(S_3 S_2 S_1, S_3(u))$

$$W\big(\emptyset, (\lambda f. f\ 5)\big) = ([(Int \to u_1)/u_0], (Int \ \to u_1) \to u_1)$$

$$W\big(\emptyset, (\lambda x. x)\big) = \ (\emptyset, u_3 \to u_3)$$

$$Unify\big((Int \ \to u_1) \to u_1, (u_3 \to u_3) \to u_4\big) = [Int/u_3\,; Int/u_1\,; Int/u_4]$$

$$W\big(\emptyset, (\lambda f. f\ 5)(\lambda x. x)\big) = \ ([(Int \to u_1)/u_0\,; Int/u_3\,; Int/u_1\,; Int/u_4], Int)$$

What about Let?

- $let\ x = e_1\ in\ e_2$
- Typing rule

This is Hindley Milner
without polymorphism

  – $$\frac{\Gamma;x{:}\tau' \vdash e_1{:}\tau' \qquad \Gamma;x{:}\tau' \vdash e_2{:}\tau}{\Gamma \vdash let\ x{=}e_1\ in\ e_2{:}\tau}$$

- Constraints
  – $[\![\Gamma \ \vdash let\ x = e_1 in\ e_2 : \tau]\!] =$
    $\exists\tau', \qquad [\![\Gamma; x{:}\tau' \vdash e_1{:}\tau']\!] \wedge [\![\Gamma; x{:}\tau' \vdash e_2{:}\tau]\!]$

- Algorithm

Case Exp = *let* x = e_1 *in* e_2
=> *let* $(S_1, \tau_1) = W(TE + \{x : u\}, e_1)$;
$S_2 = Unify(S_1(u), \tau_1)$;
$(S_3, \tau_2) = W(S_2\, S_1(TE) + \{x : \tau_1\}, e_2)$;
in $(S_3\, S_2\, S_1, \tau_2)$

Polymorphism

Some observations

- A type system restricts the class of programs that are considered "legal"
- It is possible a term in the untyped λ–calculus may be reducible to a value but may not be typeable in a particular type system

```
let
      id = λx. x
in
     ... (id True) ... (id 1) ...
```

This term is not typeable in the simple type system we have discussed so far. However, it is typeable in the Hindley-Milner system

Explicit polymorphism

- You've seen this before

```
public interface List<E>{
        void add(E x);
        E get();
}
```

```
List<String> ls = ...
ls.add("Hello");
String hello = ls.get(0);
```

- How do we formalize this?

$$\frac{\Gamma \vdash e : \tau}{\Gamma \ \vdash \ \Lambda t.e : \forall t.\tau}$$

$$\frac{\Gamma \vdash e : \forall t.\tau'}{\Gamma \ \vdash e[\tau] : \tau'[\tau \ / \ t]}$$

- Example

$$id = \Lambda T.\lambda x : T.x$$

$$id[int]\ 5$$

Different Styles of Polymorphism

- Impredicative Polymorphism

  $$\tau ::= b \mid \tau_1 \to \tau_2 \mid T \mid \forall T.\tau$$

  $$\mathrm{e} ::= x \mid \lambda x{:}\,\tau.e \mid e_1 e_2 \mid \Lambda \mathrm{T.e} \mid \mathrm{e}[\tau]$$

- Very powerful
  - Although you still can't express recursion
- Type inference is undecidable !

Different Styles of Polymorphism

- Predicative Polymorphism

$$\tau ::= b \mid \tau_1 \to \tau_2 \mid T$$
$$\sigma \ ::= \tau \mid \forall T.\sigma \mid \sigma_1 \to \sigma_2$$
$$\mathrm{e} ::= x \mid \lambda x{:}\,\sigma.e \mid e_1 e_2 \mid \Lambda \mathrm{T.e} \mid \mathrm{e}[\tau]$$

- Still very powerful
  - But you can no longer instantiate with a polymorphic type
- Type inference is still undecidable !

Different Styles of Polymorphism

- Prenex Predicative Polymorphism

  $\tau ::= b \mid \tau_1 \to \tau_2 \mid T$

  $\sigma \ ::= \tau \mid \forall T.\sigma$

  $\mathrm{e} ::= x \mid \lambda x{:}\tau.e \mid e_1 e_2 \mid \Lambda \mathrm{T}.\mathrm{e} \mid \mathrm{e}[\tau]$

- Now we have decidable type inference
- But polymorphism is now very limited
  - We can't pass polymorphic functions as arguments!!
  - $(\lambda s{:}\forall T\,.\tau\ \ldots s[int]x\ \ldots s[bool]x)(\Lambda T.code\ for\ sort)$

Let polymorphism

- Introduce let x = e1 in e2
  - Just like saying $(\lambda x. e2\ )e1$
  - Except x can be polymorphic

- Good engineering compromise
  - Enhance expressiveness
  - Preserve decidability

- This is the Hindley Milner type system

Type inference with polymorphism

Polymorphic Types

let
    id = λx. x
in
    ... (id True) ... (id 1) ...

Constraints:

id :: t_1 --> t_1
id :: Int --> t_2
id :: Bool --> t_3

Does not unify!!

Solution: Generalize the type variable

id :: $\forall t_1.\ t_1$ --> t_1

Different uses of a generalized type variable may be *instantiated* differently

id_2 : Bool --> Bool
id_1 : Int --> Int

When can we generalize?

A mini Language
to study Hindley-Milner Types

Expressions

$E ::= c$		constant
$\mid$	x	variable
$\mid$	$\lambda x.\ E$	abstraction
$\mid$	$(E_1\ E_2)$	application
$\mid$	*let* $x = E_1$ *in* E_2	let-block

- There are no types in the syntax of the language!
- The type of each subexpression is derived by *the Hindley-Milner type inference algorithm.*

A Formal Type System

Types

τ ::= ι — base types

| t — type variables

| τ_1 -> τ_2 — Function types

Type Schemes

σ ::= τ

| $\forall$t. σ

Type Environments

TE ::= Identifiers $\rightarrow$ Type Schemes

Note, all the $\forall$'s occur in the beginning of a type scheme, i.e., a type τ cannot contain a type scheme σ

Instantiations

$$\boxed{\sigma = \forall t_1 \ldots t_n.\ \tau}$$

- Type scheme σ can be *instantiated* into a type τ' by *substituting types for the bound variables of* σ, i.e.,

  $\tau' = S\ \tau$ for some S s.t. $Dom(S) \subseteq BV(\sigma)$

  - τ' is said to be an *instance of* σ ($\sigma > \tau'$)

  - τ' is said to be a *generic instance of* σ when S maps variables to new variables.

Example:

$$\sigma = \forall t_1.\ t_1 \text{ -> } t_2$$

$t_3 \text{ -> } t_2$ is a generic instance of σ

$Int \text{ -> } t_2$ is a non generic instance of σ

Generalization *aka Closing*

$$\mathrm{Gen}(TE,\tau) = \forall\, t_1 \ldots t_n.\ \tau \quad \text{where } \{ t_1 \ldots t_n \} = FV(\tau) - FV(TE)$$

- *Generalization* introduces polymorphism
- Quantify type variables that are free in τ but not *free* in the type environment (TE)
- Captures the notion of *new* type variables of τ

HM Type Inference Rules

Remember, τ stands for a monotype, σ for a polymorphic type

(App)
$$\frac{\Gamma \vdash e_1 : \tau \to \tau' \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash (e_1 e_2) : \tau'}$$

(Abs)
$$\frac{\Gamma ; \{x : \tau\} \vdash e : \tau'}{\Gamma \vdash \lambda x.e : \tau \to \tau'}$$

(Var)
$$\frac{(x : \sigma) \in \Gamma \quad \sigma \geq \tau}{\Gamma \vdash x : \tau}$$

x can be considered of type τ as long as its type as specified in the environment can be specialized to τ (i.e. τ is an instance of σ)

(Const)
$$\frac{typeof(c) \geq \tau}{\Gamma \vdash c : \tau}$$

Note: x has a different type in e_1 than in e_2. In e_1, x is not a polymorphic type, but in e_2 it gets generalized into one.

(Let)
$$\frac{\Gamma ; \{x : \tau\} \vdash e_1 : \tau \quad \Gamma ; \{x : Gen(\Gamma, \tau)\} \vdash e_2 : \tau'}{\Gamma \vdash (let\ x = e_1\ in\ e_2) : \tau'}$$

HM Inference Algorithm

u's represent new type variables

$$
\begin{array}{lll}
\textit{Def}\ W(TE, e) & = & \textit{Case}\ e\ \textit{of} \\
\quad c & = & (\{\}, \mathrm{Typeof}(c)) \\
\quad x & = & \textit{if}\ \ (x \notin \mathrm{Dom}(TE))\ \textit{then}\ \mathrm{Fail} \\
 & & \quad \textit{else}\ \ \textit{let}\ \ \forall t_1 \ldots t_n . \tau\ = TE(x); \\
 & & \qquad\qquad \textit{in}\ \ (\ \{\ \}, [u_i / t_i]\ \tau) \\
\quad \lambda x.e & = & \textit{let}\ \ (S_1, \tau_1) = W(TE + \{\ x : u\ \}, e); \\
 & & \quad \textit{in}\ \ (S_1, S_1(u) \text{ -> } \tau_1) \\
\quad (e_1\ e_2) & = & \textit{let}\ (S_1, \tau_1) = W(TE, e_1); \\
 & & \qquad (S_2, \tau_2) = W(S_1(TE), e_2); \\
 & & \qquad\quad S_3 \quad = \mathrm{Unify}(S_2(\tau_1), \tau_2 \text{ -> } u); \\
 & & \quad \textit{in}\ \ (S_3\, S_2\, S_1, S_3(u)) \\
\quad \textit{let}\ \ x = e_1\ \textit{in}\ \ e_2 & & \\
 & = & \textit{let}\ (S_1, \tau_1) = W(TE + \{x : u\}, e_1); \\
 & & \qquad S_2 \qquad = \mathrm{Unify}(S_1(u), \tau_1); \\
 & & \qquad \sigma \qquad\ = \mathrm{Gen}(S_2\, S_1(TE), S_2(\tau_1)\ ); \\
 & & \qquad (S_3, \tau_2) = W(S_2\, S_1(TE) + \{x : \sigma\}, e_2); \\
 & & \quad \textit{in}\ \ (S_3\, S_2\, S_1, \tau_2)
\end{array}
$$

Hindley-Milner: Example

$$\boxed{\lambda x.\ \boxed{\begin{array}{l}\texttt{let f = }\lambda y.x \\ \texttt{in (f 1, f True)}\end{array}}^{B}}^{A}$$

W(∅, A) = ([ ] , u_1 -> (u_1,u_1))

W({x : u_1}, B) = ([ ] , (u_1,u_1))

W({x : u_1, f : u_2}, **λy.x**) = ([ ] , u_3 -> u_1)

W({x : u_1, f : u_2, y : u_3}, **x**) = ([ ] , u_1)

Unify(u_2 , u_3 -> u_1) = [(u_3 -> u_1) / u_2]

Gen({x : u_1}, u_3 -> u_1) = $\forall u_3.u_3$ -> u_1

TE = {x : u_1, f : $\forall u_3.u_3$ -> u_1}

W(TE, **(f 1)**) = ([ ] , u_1)

W(TE, **f**) = ([ ] , u_4 -> u_1)

W(TE, 1) = ([ ] , **Int**)

Unify(u_4 -> u_1 , **Int** -> u_5) = [**Int** / u_4 , u_1 / u_5]

...

Important Observations

- Do not generalize over type variables used elsewhere
- Let is the only way of defining polymorphic constructs
- Generalize the types of let-bound identifiers only after processing their definitions

Properties of HM Type Inference

- It is sound with respect to the type system.
  An inferred type is verifiable.

- It generates most general types of expressions.
  Any verifiable type is inferred.

- Complexity
  PSPACE-Hard
  Nested *let* blocks

Extensions

- Type Declarations
  Sanity check; can relax restrictions

- Incremental Type checking
  The whole program is not given at the same time, sound inferencing when types of some functions are not known

- Typing references to mutable objects
  Hindley-Milner system is unsound for a language with refs (mutable locations)

- Overloading Resolution

HM Limitations:
λ-bound vs Let-bound Variables

Only let-bound identifiers can be instantiated differently.

```
let
    twice f x = f (f x)
in
    twice twice succ 4
```

versus

```
let
    twice f x = f (f x)
    foo g = (g g succ) 4
in
   foo twice
```

foo is not type correct !

Generic vs. Non-generic type variables

Puzzle: Another set of Inference rules

(Gen) $$\frac{TE \vdash e : \tau \quad t \notin FV(TE)}{TE \vdash e : \forall t.\tau}$$

(Spec) $$\frac{TE \vdash e : \forall t.\tau}{TE \vdash e : \tau\,[u/t]}$$

(Var) $$\frac{(x : \tau) \in TE}{TE \vdash x : \tau}$$

(Let) $$\frac{TE+\{x:\tau\} \vdash e_1 : \tau \quad TE+\{x:\tau\} \vdash e_2 : \tau'}{TE \vdash (\textit{let}\ x = e_1\ \textit{in}\ e_2) : \tau'}$$

(App) and (Abs) rules remain unchanged.

Sound but no inference algorithm !

MIT OpenCourseWare
http://ocw.mit.edu

6.820 Fundamentals of Program Analysis
Fall 2015

For information about citing these materials or our Terms of Use, visit: http://ocw.mit.edu/terms.