Parametric Polymorphism Motivation

Fundamental tension between static type safety and reusability.

- Static typechecking provides strong reliability guarantees.

  sum = λ l:Int List. if null(l) then 0 else head(l) + sum(tail(l))

  - ▷ $\vdash$ sum : Int List $\rightarrow$ Int
  - ▷ sum can "safely" accept integer lists
  - ▷ sum will only be passed integer lists

- However, the static typechecker is necessarily conservative.

  length = λ l:Int List. if null(l) then 0 else 1 + length(tail(l))

  - ▷ $\vdash$ length : Int List $\rightarrow$ Int
  - ▷ length can "safely" accept integer lists
  - ▷ length will only be passed integer lists
  - ▷ but I know it's safe to pass any kind of list to length!

Parametric Polymorphism

Introduce type variables, which can be instantiated with any type.

length = λ l:α list. if null(l) then 0 else 1 + sum(tail(l))

- α is a type variable
- $\vdash$ length: α List $\rightarrow$ Int

The type α List $\rightarrow$ Int is implicitly a type scheme $\forall\alpha.$(α List $\rightarrow$ Int)

- (length(Int) [1,2,3])
- (length(Bool) [true,false])

A given type variable must be instantiated identically everywhere.

id = λx:α.x

- $\vdash$ id: $\alpha \rightarrow \alpha$
- $\vdash$ id(Int) 3 : Int
- $\vdash$ id(Bool) true : Bool

Type Inference

Programs omit all type information.

Types are inferred (or reconstructed) by the compiler.

Provides the guarantees of an explicitly-typed language but the "feel" of an untyped language.

Polymorphic type inference

- Infer the types of bound variables.
  - ▷ $\vdash$ λ l. if null(l) then 0 else 1 + sum(tail(l)) : α list $\rightarrow$ Int
- Infer the instantiations of polymorphic functions.
  - ▷ (id true) : Bool

Principal Types

Polymorphism and type inference are orthogonal concepts.

- could have explicitly-typed polymorphism
- could infer monomorphic types

What makes the combination particularly natural is the principal type property: If expression e has a type, then it has a "best" type.

- every type for e is an instance of the "best" type.
- length has type (Int List $\rightarrow$ Int), (Bool List $\rightarrow$ Int), ...,(α List $\rightarrow$ Int)

Type inference will compute the best type for each expression.

Does the simply-typed lambda calculus enjoy the principal type property (for unannotated lambda-calculus expressions)?

The Language

e	$::=$	x	variable
		$\lambda x.e$	function
		$e_1\ e_2$	function call
		if e_1 then e_2 else e_3	conditional
		let $x = e_1$ in e_2	local declaration
		Bool/Int primitives	

T	$::=$	α	type variable
		$T_1 \rightarrow T_2$	function type
		Bool | Int	base types

Polymorphic Type Inference via Constraint Solving

$\lambda x.\lambda y.$if (not x) then y else $\lambda z.z$

Annotate each sub-expression with a fresh type variable.

$$(\lambda x : \alpha_1.(\lambda y : \alpha_2.(\text{if (not}: \alpha_3\ x : \alpha_4): \alpha_5 \text{ then } y : \alpha_6 \text{ else } (\lambda z : \alpha_7.z : \alpha_8) : \alpha_9) : \alpha_{10}) : \alpha_{11}) : \alpha$$

Generate constraints on the type variable for each sub-expression.

$\alpha_3 = \text{Bool} \rightarrow \text{Bool}$	[prim]
$\alpha_4 = \alpha_1$	[var]
$\alpha_3 = \alpha_{12} \rightarrow \alpha_{13}$, $\alpha_4 = \alpha_{12}$, $\alpha_5 = \alpha_{13}$	[app]
$\alpha_6 = \alpha_2$	[var]
$\alpha_8 = \alpha_7$	[var]
$\alpha_9 = \alpha_7 \rightarrow \alpha_8$	[lam]
$\alpha_5 = \text{Bool}$, $\alpha_6 = \alpha_9 = \alpha_{10}$	[if]
$\alpha_{11} = \alpha_2 \rightarrow \alpha_{10}$	[lam]
$\alpha = \alpha_1 \rightarrow \alpha_{11}$	[lam]

Polymorphic Type Inference via Constraint Solving (cont.)

$\lambda x.\lambda y.$if (not x) then y else $\lambda z.z$

Solve the constraints. If unsatisfiable, the program is ill-formed. Otherwise, return the (most general) solved form of α.

$\alpha_3 = \text{Bool} \rightarrow \text{Bool}$
$\alpha_4 = \alpha_1$
$\alpha_3 = \alpha_{12} \rightarrow \alpha_{13}$
$\alpha_4 = \alpha_{12}$
$\alpha_5 = \alpha_{13}$
$\alpha_6 = \alpha_2$
$\alpha_8 = \alpha_7$
$\alpha_9 = \alpha_7 \rightarrow \alpha_8$
$\alpha_5 = \text{Bool}$
$\alpha_6 = \alpha_9 = \alpha_{10}$
$\alpha_{11} = \alpha_2 \rightarrow \alpha_{10}$
$\alpha = \alpha_1 \rightarrow \alpha_{11}$

Hindley-Milner Type Inference

Solve constraints as they are generated, via unification.

Infer types compositionally, from the types of sub-expressions.

- Very similar to the way typing judgements are "inferred" in the typing rules.

An environment Γ maintains the type of each bound variable.

A bound variable is initially assumed to have a fresh type variable as its type. Unification updates the environment as new constraints are imposed.

Hindley-Milner Type Inference (cont.)

The Algorithm

- The unify function side-effects Γ to reflect new constraints.
- The freshTypeVar function “invents” a new type variable.

typeOf($\lambda x.e$, Γ) =
  Γ := $\Gamma \cup \{x$:freshTypeVar()$\}$;
  retType := typeOf(e, Γ);
  return $\Gamma(x) \rightarrow$ retType;

typeOf(x, Γ) = return $\Gamma(x)$;

typeOf(e_1 e_2, Γ) =
  funType := typeOf(e_1, Γ);
  argType := typeOf(e_2, Γ);
  resType := freshTypeVar();
  aType $\rightarrow$ rType := unify(funType, argType $\rightarrow$ resType, Γ);
  return rType;

Hindley-Milner Type Inference (cont.)

typeOf(if e_1 then e_2 else e_3, Γ) =
  testType := unify(typeOf(e_1, Γ), Bool);
  thenType := typeOf(e_2, Γ);
  elseType := typeOf(e_3, Γ);
  return unify(thenType, elseType, Γ);

typeOf(let $x = e_1$ in e_2, Γ) =
  varType := typeOf(e_1, Γ);
  return typeOf(e_2, $\Gamma \cup \{x$:varType$\}$);

Example

$\lambda x.\lambda y.$if (not x) then y else $\lambda z.z$

typeOf(not, $\{x : \alpha_1, y : \alpha_2\}$) = Bool $\rightarrow$ Bool
typeOf(x, $\{x : \alpha_1, y : \alpha_2\}$) = α_1
unify(Bool $\rightarrow$ Bool, $\alpha_1 \rightarrow \alpha_3$, $\{x : \alpha_1, y : \alpha_2\}$) = Bool $\rightarrow$ Bool

▷ α_1 = Bool

typeOf(not x, $\{x : \alpha_1, y : \alpha_2\}$) = Bool
unify(Bool, Bool, $\{x :$ Bool$, y : \alpha_2\}$) = Bool
typeOf(y, $\{x :$ Bool$, y : \alpha_2\}$) = α_2
typeOf(z, $\{x :$ Bool$, y : \alpha_2, z : \alpha_3\}$) = α_3
typeOf($\lambda z.z$, $\{x :$ Bool$, y : \alpha_2\}$) = $\alpha_3 \rightarrow \alpha_3$
unify(α_2, $\alpha_3 \rightarrow \alpha_3$,$\{x :$ Bool$, y : \alpha_2\}$) = $\alpha_3 \rightarrow \alpha_3$

▷ $\alpha_2 = \alpha_3 \rightarrow \alpha_3$

typeOf(if (not x) then y else $\lambda z.z$, $\{x : \alpha_1, y : \alpha_2\}$) = $\alpha_3 \rightarrow \alpha_3$
typeOf(λy.if ..., $\{x : \alpha_1\}$) = $(\alpha_3 \rightarrow \alpha_3) \rightarrow (\alpha_3 \rightarrow \alpha_3)$
typeOf($\lambda x.\lambda y$.if ...,$\{\}$) = Bool $\rightarrow (\alpha_3 \rightarrow \alpha_3) \rightarrow (\alpha_3 \rightarrow \alpha_3)$

Let-Polymorphism

We’ve seen how to infer a polymorphic type for a function.

For polymorphism to be useful, we must be able to invoke a function using different type instantiations.

let id = $\lambda x.x$ in (if id(true) then id(3) else 0)

The current algorithm doesn’t support polymorphism!

...
typeOf(id, $\{$id $: \alpha_1 \rightarrow \alpha_1\}$) = $\alpha_1 \rightarrow \alpha_1$
typeOf(true, $\{$id $: \alpha_1 \rightarrow \alpha_1\}$) = Bool
unify($\alpha_1 \rightarrow \alpha_1$, Bool $\rightarrow \alpha_2$, $\{$id $: \alpha_1 \rightarrow \alpha_1\}$) = Bool $\rightarrow$ Bool
...
typeOf(id, $\{$id :Bool $\rightarrow$ Bool$\}$) = Bool $\rightarrow$ Bool
typeOf(3, $\{$id :Bool $\rightarrow$ Bool$\}$) = Int
unify(Bool $\rightarrow$ Bool, Int $\rightarrow \alpha_3$, $\{$id :Bool $\rightarrow$ Bool$\}$) = FAIL

Generic Type Variables

Distinguish between generic (polymorphic) and non-generic (monomorphic) type variables.

- A generic variable has had all of its associated constraints solved.
- A non-generic variable may be further constrained, so it is unsafe to view it as polymorphic.

A type variable α in the type of expression e is generic if α is not in the formal-parameter type of a λ enclosing e.[1]

- let-bound variables may be polymorphic; formal parameters may not be.

The updated algorithm:

- The function copyGenericVars returns a type identical to the given one, but with each generic type variable replaced by a fresh type variable.

typeOf(x, Γ) = return copyGenericVars($\Gamma(x)$);

[1] See the caveat about recursive identifiers below.

Let-Polymorphism (cont.)

let id = $\lambda x.x$ in (if id(true) then id(3) else 0)

...
typeOf(id, $\{$id : $\alpha_1 \to \alpha_1\}$) = $\beta \to \beta$
typeOf(true, $\{$id : $\alpha_1 \to \alpha_1\}$) = Bool
unify($\beta \to \beta$, Bool $\to \alpha_2$, $\{$id : $\alpha_1 \to \alpha_1\}$) = Bool $\to$ Bool
...
typeOf(id, $\{$id : $\alpha_1 \to \alpha_1\}$) = $\gamma \to \gamma$
typeOf(3, $\{$id : $\alpha_1 \to \alpha_1\}$) = Int
unify($\gamma \to \gamma$, Int $\to \alpha_3$, $\{$id : $\alpha_1 \to \alpha_1\}$) = Int $\to$ Int

Non-generic variables cannot be copied.

λ id.(if id(true) then id(3) else 0) cannot be typed

- $(\alpha \to \alpha) \to$ Int is unsound
  - ▷ consider (λ id.(if id(true) then id(3) else 0))(factorial)
- $(\forall \alpha.(\alpha \to \alpha)) \to$ Int is sound but cannot be inferred

Recursion

Recursion is easy. Just introduce the recursive identifier into the environment before inferring its type.

typeOf(let $x = e_1$ in e_2, Γ) =
  $\Gamma := \Gamma \cup \{x$:freshTypeVar()$\}$;
  valType := typeOf(e_1, Γ);
  varType := unify($\Gamma(x)$, valType);
  return typeOf(e_2, Γ);

The type variable for x is treated as non-generic within e_1 (thereby qualifying the definition of generic variables given earlier).

This restriction implies that all recursive references to x in e_1 must use the same type instantiation.

Note that x may still be treated polymorphically in e_2.

Polymorphic Type Systems

Start with the simply-typed lambda calculus, where T now ranges over the augmented grammar of types (including type variables).

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \text{(T-Var)}$$

$$\frac{}{\Gamma \vdash \text{true} : \text{Bool}} \text{(T-True)}$$

$$\frac{}{\Gamma \vdash \text{false} : \text{Bool}} \text{(T-False)}$$

$$\frac{\Gamma \cup \{x : T_1\} \vdash e : T_2}{\Gamma \vdash (\lambda x : T_1.e) : T_1 \to T_2} \text{(T-Abs)}$$

$$\frac{\Gamma \vdash e_1 : \text{Bool} \quad \Gamma \vdash e_2 : T \quad \Gamma \vdash e_3 : T}{\Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : T} \text{(T-If)}$$

$$\frac{\Gamma \vdash e_1 : T_2 \to T \quad \Gamma \vdash e_2 : T_2}{\Gamma \vdash e_1\ e_2 : T} \text{(T-App)}$$

Polymorphic Type Systems (cont.)

Type inference is trivial.

- explicit typing

$$\frac{\Gamma \cup \{x : T_1\} \vdash e : T_2}{\Gamma \vdash (\lambda x : T_1.e) : T_1 \to T_2}\,\text{(T-Abs)}$$

- type inference

$$\frac{\Gamma \cup \{x : T_1\} \vdash e : T_2}{\Gamma \vdash (\lambda x.e) : T_1 \to T_2}\,\text{(T-Abs)}$$

  ▷ The Hindley-Milner algorithm tells us how to "guess" T_1.

Local variables

$$\frac{\Gamma \vdash e_1 : T_1 \quad \Gamma \cup \{x : T_1\} \vdash e : T_2}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : T_2}\,\text{(T-Let)}$$

Example Derivation

$$\cfrac{\cdots \quad \cdots \quad \cfrac{\cfrac{\cdots}{\{x : \text{Bool}, y : \alpha \to \alpha, z : \alpha\} \vdash z : \alpha}}{\{x : \text{Bool}, y : \alpha \to \alpha\} \vdash \lambda z.z : \alpha \to \alpha}}{\cfrac{\{x : \text{Bool}, y : \alpha \to \alpha\} \vdash \text{if (not } x\text{) then } y \text{ else } \lambda z.z : (\alpha \to \alpha)}{\cfrac{\{x : \text{Bool}\} \vdash \lambda y.\text{if (not } x\text{) then } y \text{ else } \lambda z.z : (\alpha \to \alpha) \to (\alpha \to \alpha)}{\vdash \lambda x.\lambda y.\text{if (not } x\text{) then } y \text{ else } \lambda z.z : \text{Bool} \to (\alpha \to \alpha) \to (\alpha \to \alpha)}}}$$

Generic and Non-generic Variables

Add explicit quantifiers to types.

$$T \quad ::= \quad \cdots \mid \forall \alpha.T$$

Only quantified type variables are generic.

$$\frac{\Gamma \vdash e : T \quad \alpha \text{ not free in } \Gamma}{\Gamma \vdash e : \forall \alpha.T}\,\text{(T-Gen)}$$

- second premise ensures α is generic

Generic type variables can be instantiated.

$$\frac{\Gamma \vdash e : \forall \alpha.T}{\Gamma \vdash e : [\alpha \mapsto T']T}\,\text{(T-Spec)}$$

- the analog of generic-variable copying in the Hindley-Milner algorithm.

A Derivation

let $\Gamma \equiv \{\text{id} : \forall \alpha.(\alpha \to \alpha)\}$

$$\cfrac{\cfrac{\cfrac{\cdots}{\vdash \lambda x.x : \alpha \to \alpha}}{\vdash \lambda x.x : \forall \alpha.(\alpha \to \alpha)} \quad \cfrac{\text{continued below}}{\Gamma \vdash \text{if id(true) then id(3) else } 0 : \text{Int}}}{\vdash \text{let id} = \lambda x.x \text{ in (if id(true) then id(3) else 0)} : \text{Int}}$$

$$\cfrac{\cfrac{\cfrac{\cfrac{\cdots}{\Gamma \vdash \text{id} : \forall \alpha.(\alpha \to \alpha)}}{\Gamma \vdash \text{id}:\text{Bool} \to \text{Bool}} \quad \cfrac{}{\Gamma \vdash \text{true}:\text{Bool}}}{\Gamma \vdash \text{id(true)}:\text{Bool}} \quad \cfrac{\cfrac{\cfrac{\cdots}{\Gamma \vdash \text{id} : \forall \alpha.(\alpha \to \alpha)}}{\Gamma \vdash \text{id}:\text{Int} \to \text{Int}} \quad \cfrac{}{\Gamma \vdash 3:\text{Int}}}{\Gamma \vdash \text{id(3)}:\text{Int}}}{\Gamma \vdash \text{if id(true) then id(3) else } 0 : \text{Int}}$$

A "Non-Shallow" Derivation

$$\frac{\dfrac{\text{same derivation as on previous slide}}{\{\text{id} : \forall\alpha.(\alpha \to \alpha)\} \vdash \text{if id(true) then id(3) else 0:Int}}}{\vdash \lambda\text{id.(if id(true) then id(3) else 0)} : (\forall\alpha.(\alpha \to \alpha)) \to \text{Int}}$$

This expression cannot be typed in the Hindley-Milner algorithm.

Note that it is not possible to derive the type $(\alpha \to \alpha) \to \text{Int}$ for the function.

Supporting Recursion

The traditional "let" rule

$$\frac{\Gamma \vdash e_1 : T_1 \quad \Gamma \cup \{x : T_1\} \vdash e : T_2}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : T_2} \text{(T-Let)}$$

Compute the type T_1 for e_1 under the assumption that x has type T_1!

$$\frac{\Gamma \cup \{x : T_1\} \vdash e_1 : T_1 \quad \Gamma \cup \{x : T_1\} \vdash e : T_2}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : T_2} \text{(T-Let)}$$

- Again, the Hindley-Milner algorithm tells us how to "guess" T_1.