Polymorphic type inference

K. V. Raghavan

Indian Institute of Science, Bangalore

The term language

- Term language

$$\begin{aligned} \mathsf{e} \quad := \quad & \mathsf{Var} \mid \mathsf{e\ e} \mid \lambda\ \mathsf{Var}\ .\ \mathsf{e} \mid \\ & \mathsf{let\ Var} = \mathsf{e\ in\ e} \mid \mathsf{letrec\ Var} = \mathsf{e\ in\ e} \\ & \mathsf{true} \mid \mathsf{false} \mid \mathit{Int} \mid \mathsf{if\ e\ then\ e\ else\ e} \mid \end{aligned}$$

- Operational semantics

$$\mathsf{let\ v} = \mathsf{e\ in\ e1} \rightarrow (\lambda \mathsf{v.e1})\ \mathsf{e}$$

$$\frac{\mathsf{e1} \rightarrow \mathsf{e1'}}{(\mathsf{let\ v} = \mathsf{e\ in\ e1}) \rightarrow (\mathsf{let\ v} = \mathsf{e\ in\ e1'})}$$

$$\frac{\mathsf{f\ occurs\ free\ in\ e1}}{(\mathsf{letrec\ f} = \mathsf{e\ in\ e1}) \rightarrow (\mathsf{letrec\ f} = \mathsf{e\ in\ [f} \mapsto \mathsf{e]e1})}$$

$$\frac{\mathsf{f\ does\ not\ occur\ free\ in\ e1}}{(\mathsf{letrec\ f} = \mathsf{e\ in\ e1}) \rightarrow \mathsf{e1}}$$

$$\frac{\mathsf{e1} \rightarrow \mathsf{e1'}}{(\mathsf{letrec\ v} = \mathsf{e\ in\ e1}) \rightarrow (\mathsf{letrec\ v} = \mathsf{e\ in\ e1'})}$$

- Values

$$\mathsf{v} \quad := \quad \mathsf{true} \mid \mathsf{false} \mid \mathit{Int} \mid \lambda \mathsf{x.e}$$

Illustration of letrec

Let's and Letrec's – example

letrec fact = λn. if (n = 1) then 1 else
n * (fact (n − 1)) in (fact 2) →

letrec fact = . . . in
((λn . if (n = 1) then 1 else (n * (fact (n − 1)))) 2)
→

letrec fact = . . . in
if (2 = 1) then 1 else (2 * (fact (2 − 1))) →

letrec fact = . . . in (2 * (fact 1)) →

Illustration of letrec - II

letrec fact = ... in

2 *

((λn . if (n = 1) then 1 else (n * (fact (n − 1)))) 1)

→ letrec fact = ... in

2 * (if (1 = 1) then 1 else (1 * (fact 0)))

→ letrec fact = ... in 2 * 1 → 2 * 1 → 2

ML-style polymorphic type checking

How is this different from STLC?

- Programmer does not annotate types of variables. System checks well-typedness without annotations.
- Supports polymorphic function definitions, and even polymorphic recursive function definitions.
  - A polymorphic term is one that can be assigned many different types.
- System checks whether the (unannotated) term is well-typed, and if yes, *infers* a *principal type* (most general type) for it.

Examples

- Consider λ-calculus extended with "let"s. In STLC, we would need to write separate functions
  - idBool $= \lambda x$:Bool. x
  - idNat $= \lambda x$:Nat. x

  etc.
- These functions all have the same operational semantics. Hence, a redundancy!
- To fix this issue, we extend the language of types:

Type	:=	$\forall$ TVar . Type | UType
UType	:=	Nat | Bool | UType $\rightarrow$ UType | TVar
TVar	:=	A, B, C, $\ldots$

  Note: 'Type' is the domain of polymorphic types. UType is the domain of monomorphic types. We use T_1, T_2, etc. to denote poly types, and U_1, U_2, etc., to denote mono types.
- id $= \lambda x.x$ is well-typed, because it is possible to annotate it (in many ways, in fact) to yield a well-typed STLC term.

Polymorphic types

- An instance of a type $T_1 = \forall v. T_2$ is a type $T_3 = [v \mapsto U_1] T_2$, where U_1 is some mono type. We say T_1 is *more general* than T_3.
- Intuitively
  - Any polytype represents a family of monotypes, which are all (direct or transitive) instances of the polytype
  - If $t : T$, and T is a polytype, it is as if t is of every type in the family of T
- A principal type or most general type for an expression e is a type T such that every possible mono type U for e is an *instance* of T.
- Therefore, principal type for id is $\forall A.A \rightarrow A$.

Typing rules

(Note: T's are Types, U's are UTypes, and A's are TVars)

$$\frac{\text{v:T} \in \Gamma}{\Gamma \vdash \text{v:T}} \quad \text{[T-VAR]}$$

$$\frac{\Gamma \vdash \text{e1:U1} \rightarrow \text{U2}, \quad \Gamma \vdash \text{e2:U1}}{\Gamma \vdash \text{(e1 e2):U2}} \quad \text{[T-APP]}$$

$$\frac{\Gamma, \text{v:U1} \vdash \text{e:U}}{\Gamma \vdash (\lambda \text{v.e}):\text{U1} \rightarrow \text{U}} \quad \text{[T-ABS]}$$

$$\frac{\Gamma \vdash \text{e1:U1}, \ \Gamma \vdash \text{e2:U2}}{\Gamma \vdash \text{(e1, e2): (U1, U2)}} \quad \text{[T-PAIR]}$$

Typing rules – continued

$$\frac{\Gamma \vdash \text{e1:T}, \quad \Gamma, \text{v:T} \vdash \text{e:U}}{\Gamma \vdash (\text{let v=e1 in e}):\text{U}} \; [\text{T-Let}]$$

Note: Unlike in T-Abs, in T-Let we type-check the body e in an environment where v may have a polymorphic type T (derived from the inferred type of e1 using T-Gen).

- Therefore, different occurrences of v in e can have different types.

Typing rules – continued

$$\frac{\Gamma \vdash \text{e:U},\ A_1, \ldots, A_n \notin \mathrm{FV}(\Gamma)}{\Gamma \vdash e : \forall A_1 . \ldots . \forall A_n . \text{U}} \quad [\text{T-Gen}]$$

$$\frac{\begin{array}{c}\Gamma \vdash \text{e:}\forall A_1 \forall A_2 \ldots \forall A_n.\text{U},\\ \mathrm{FV}(\mathrm{U}_1) \cap (\mathrm{FV}(\Gamma) \cup \{A_2 \ldots A_n\}) = \phi\end{array}}{\Gamma \vdash \text{e:}\forall A_2 \ldots \forall A_n.[A_1 \mapsto \mathrm{U}_1]\text{U}} \quad [\text{T-Inst}]$$

Illustration 1

$$
\dfrac{
  A \notin FV(\emptyset)
  \qquad
  \dfrac{
    \dfrac{f : Nat \to A \vdash f : Nat \to A, \; 3 : Nat}{f : Nat \to A \vdash (f\ 3) : A}\ \text{T-App}
  }{\vdash (\lambda f.(f\ 3)) : (Nat \to A) \to A}\ \text{T-Abs}
}{\vdash (\lambda f.(f\ 3)) : \forall A.\,(Nat \to A) \to A}\ \text{T-Gen}
$$

Illustration 2

$$\frac{}{g:\ldots, n:\mathrm{Nat} \vdash (n+1):\mathrm{Nat}}\ \text{T-plus}$$

$$\frac{}{g:\forall A.(\mathrm{Nat}\Rightarrow A)\Rightarrow A \vdash (\lambda n.n+1):\mathrm{Nat}\Rightarrow\mathrm{Nat}}\ \text{T-Abs}$$

$$\frac{FV(\mathrm{Nat}) \cap FV(g:\forall A.(\mathrm{Nat}\Rightarrow A)\Rightarrow A)=\phi, \quad g:\forall A.(\mathrm{Nat}\Rightarrow A)\Rightarrow A \vdash g:\forall A.(\mathrm{Nat}\Rightarrow A)\Rightarrow A}{g:\forall A.(\mathrm{Nat}\Rightarrow A)\Rightarrow A \vdash g:(\mathrm{Nat}\Rightarrow\mathrm{Nat})\Rightarrow\mathrm{Nat}}\ \text{T-inst}$$

$$\frac{g:\forall A.(\mathrm{Nat}\Rightarrow A)\Rightarrow A \vdash g:(\mathrm{Nat}\Rightarrow\mathrm{Nat})\Rightarrow\mathrm{Nat}, \quad g:\forall A.(\mathrm{Nat}\to A)\Rightarrow A \vdash (\lambda n.n+1):\mathrm{Nat}\Rightarrow\mathrm{Nat}}{g:\forall A.(\mathrm{Nat}\to A)\Rightarrow A \vdash (g\ (\lambda n.n+1)):\mathrm{Nat}}\ \text{T-app}$$

$$\frac{g:\forall A.(\mathrm{Nat}\Rightarrow A)\Rightarrow A \vdash g:(\mathrm{Nat}\Rightarrow\mathrm{Bool})\Rightarrow\mathrm{Bool}, \quad g:\ldots \ \lambda n.n>4:\mathrm{Nat}\to\mathrm{Bool}}{g:\forall A.(\mathrm{Nat}\to A)\Rightarrow A \vdash (g\ (\lambda n.n>4)):\mathrm{Bool}}\ \text{T-app}$$

$$\frac{\ldots}{g:\forall A.(\mathrm{Nat}\Rightarrow A)\Rightarrow A \vdash ((g\ (\lambda n.n+1)), (g\ (\lambda n.n>4))):(\mathrm{Nat},\mathrm{Bool})}\ \text{T-Pair}$$

$$\frac{\text{See previous slide}}{\vdash (\lambda f.(f\ 3)):\forall A.(\mathrm{Nat}\Rightarrow A)\Rightarrow A}\ \text{T-gen}$$

$$\frac{\vdash (\lambda f.(f\ 3)):\forall A.(\mathrm{Nat}\Rightarrow A)\Rightarrow A \qquad g:\forall A.(\mathrm{Nat}\Rightarrow A)\Rightarrow A \vdash ((g\ (\lambda n.n+1)), (g\ (\lambda n.n>4))):(\mathrm{Nat},\mathrm{Bool})}{\vdash \text{let } g = \lambda f.(f\ 3) \text{ in } ((g\ (\lambda n.n+1)), (g\ (\lambda n.n>4))):(\mathrm{Nat},\mathrm{Bool})}\ \text{T-let}$$

An ill-typed lambda abstraction

Consider df $=\lambda$f.((f 3), (f true)). What is its type?

- It is *not* (Nat$\rightarrow A)\rightarrow (A, A)$,
  - '(f true)' has invalid argument.
- nor (Bool$\rightarrow A)\rightarrow (A, A)$,
- nor even $(B \rightarrow A)\rightarrow (A, A)$
  - Unquantified variables are implicitly existentially quantified. Think of the above type as being equivalent to $\exists A \exists B.(B \rightarrow A) \rightarrow (A, A)$. This is not the right type for df.
- What if it is $\forall A \forall B.(B \rightarrow A) \rightarrow (A, A)$?
  - (λf.((f 3), (f true))) (λn.n+1) type checks!
  - Reason: (Nat$\rightarrow$Nat)$\rightarrow$(Nat,Nat) is an instance of $\forall A \forall B.(B \rightarrow A) \rightarrow (A, A)$, and is applicable to λn.n+1.
- What if it is $\forall A.(\forall B.B \rightarrow A) \rightarrow (A, A)$?
  - Would have worked, except that it is a "deep" type, which the current type system does not support (deep type = all $\forall$'s are not at the outermost level).
    - An example of valid argument to df under this typing:

An ill-typed lambda abstraction

Consider df $= \lambda$f.((f 3), (f true)). What is its type?

- It is *not* (Nat$\rightarrow A$)$\rightarrow (A, A)$,
  - '(f true)' has invalid argument.
- nor (Bool$\rightarrow A$)$\rightarrow (A, A)$,
- nor even $(B \rightarrow A)\rightarrow (A, A)$
  - Unquantified variables are implicitly existentially quantified. Think of the above type as being equivalent to $\exists A \exists B.(B \rightarrow A) \rightarrow (A, A)$. This is not the right type for df.
- What if it is $\forall A \forall B.(B \rightarrow A) \rightarrow (A, A)$?
  - (λf.((f 3), (f true))) (λn.n+1) type checks!
  - Reason: (Nat$\rightarrow$Nat)$\rightarrow$(Nat,Nat) is an instance of $\forall A \forall B.(B \rightarrow A) \rightarrow (A, A)$, and is applicable to λn.n+1.
- What if it is $\forall A.(\forall B.B \rightarrow A) \rightarrow (A, A)$?
  - Would have worked, except that it is a "deep" type, which the current type system does not support (deep type = all $\forall$'s are not at the outermost level).
    - An example of valid argument to df under this typing: $\lambda x.5$
  - There exists no shallow type for df!
- Therefore, we declare df to be ill-typed.

A closer look at T-ABS

T-ABS is able to declare df ill-typed because ...

- It type checks the body of λv.e in an environment where v is monomorphic (v:U1), as opposed to (v:$\forall A_1 \ldots A_n$.U1).
- Therefore, all occurrences of v in e are required to have the same type (the types of the different occurrences cannot be instantiated to different types).
- Therefore, df fails to type check.

How to work around this problem?

- If we knew the type of the argument df is being applied to, we could use this to type-check df.
- However, we will not always know the type of the argument while type-checking df. Example: ((λf. (f (λx.x))) df).
- The way to make the type of the argument known is to use a "let":
  - "let f = λx.x in ((f 3), (f true))" will type-check.
  - "let f = λn.n+1 in ((f 3), (f true))" will not type-check.
  - In other words, df can be type-checked whenever its argument is hard-coded (via a "let"). In this scenario df essentially does not need the first argument (i.e., f), and hence does not need a deep type.

A closer look at T-GEN

Need for the pre-condition $A \notin \mathsf{FV}(\Gamma)$ in T-GEN:

- Consider t = "λf. let g = f in ((g 3), (g true))". Say, in the T-ABS rule we guess a type f:$B \rightarrow A$, this typing to the environment, and then proceed to type-check the sub-term "let g = f in ((g 3), (g true))".
- Term t1 = "((g 3), (g true))" would type-check if we generalized the type of g to $\forall B.B \rightarrow A$ while type-checking t1.
- However, this would implicitly force the type of t to become $\forall A.(\forall B.B \rightarrow A) \rightarrow (A, A)$, which is a deep type.
- Therefore, we include the pre-condition, which forces t1 to be type-checked under an environment wherein the type of f is monomorphic (i.e., $B \rightarrow A$), which in causes t1 to be called ill-typed.

Typing rule for letrec

$$\frac{\Gamma,\mathsf{v{:}U1} \vdash \mathsf{e1{:}U1}, \quad \Gamma,\ \mathsf{v}{:}\forall A_1 \ldots A_n.\mathsf{U1} \vdash \mathsf{e{:}U},\quad \{A_1 \ldots A_n\} \cap \mathrm{FV}(\Gamma) = \phi}{\Gamma \vdash (\mathsf{letrec\ v{=}e1\ in\ e}){:}\mathsf{U}} \ [\text{T-Letrec}]$$

Note:

- v's type is taken to be monomorphic while type-checking e1.
- v's type is taken to be polymorphic while type-checking e.
- This makes the type-system decidable.

Illustration 3

. . .

———————— T-abs

g : Nat → (A → A) ⊢ λn. λm. (if n > 0 then (g (n−1) m) else m) : Nat → (A → A)

. . .

———————— T-Pair

g : ∀A. Nat → (A → A) ⊢ (((g 5) 6), ((g 6) true)) : (Nat, Bool)

———————————————————— T-letrec

⊢ letrec g = λn. λm. (if n > 0 then (g (n−1) m) else m) in (((g 5) 6), ((g 6) true)) : (Nat, Bool)

Summary of polymorphic type system

- It is sound. That is, no term that can be given a type according to the type rules can ever reduce to a non-value normal form.
- It is incomplete. That is, there exist terms that can never reduce to a non-value form that are not typable.
- Every well-typed term has a unique principal type.
- The type system is decidable. That is, there exists an algorithm that can identify the principal type of every well-typed term.