语义分析与类型(Part3)

基础软件理论与实践公开课

ZhangYu

Introduction

Previous class:

- Type checking
- Monomorphic type inference
  - Type variable
  - Unification (functional and imperative)

Today's class:

- Polymorphic type inference
  - Type scheme
  - Generalization and instantiation

Let-polymorphism

- Why do we need let-polymorphism
  - A sweet spot in the trade-off between powerful polymorphism and simplicity
  - Polymorphism: code re-use
  - Simplicity: decidable and practical
- How do we achieve an efficient implementation of let-polymorphism

idea 基础软件中心

Tiny language with types

- Types

```
type rec typ = TInt | TBool | TVar(ref<tvar>) | TArr(typ, typ)
and tvar = Nolink(string) | Linkto(typ)
```

- Expressions

```
type rec expr = CstI(int) | CstB(bool) | Var(string)
                | If(expr, expr, expr)
                | Fun(string, expr) | App(expr, expr)
                | Let(string, expr, expr)
```

- How do we infer the type of `let`

Review

Monomorphic type inference

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T}\text{T-Var} \qquad \frac{}{\Gamma \vdash i : \mathsf{Int}}\text{T-Int} \qquad \frac{}{\Gamma \vdash b : \mathsf{Bool}}\text{T-Bool}$$

$$\frac{\text{fresh } T \quad \Gamma \vdash t_i : T_i \quad U(T_1, \mathsf{Bool}) \quad U(T, T_2) \quad U(T, T_3)}{\Gamma \vdash \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T}\text{T-If}$$

where $i \in \{1, 2, 3\}$ in $\Gamma \vdash t_i : T_i$

Note $U(T_1, T_2)$ means unification on T_1 and T_2

基础软件中心

Review

$$\frac{\text{fresh } T \qquad \Gamma, x : T \vdash t : T_1}{\Gamma \vdash \lambda x.\, t : T \to T_1}\text{T-Abs}$$

$$\frac{\text{fresh } T \qquad \Gamma \vdash t_1 : T_1 \qquad \Gamma \vdash t_2 : T_2 \qquad U(T_1, T_2 \to T)}{\Gamma \vdash t_1\, t_2 : T}\text{T-App}$$

$$\frac{\Gamma \vdash t_1 : T_1 \qquad \Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \text{ let } x = t_1 \text{ in } t_2 : T_2}\text{T-MonoLet}$$

Let-polymorphism

Consider this example:

```
let id = fun x -> x in    // infer id has type T -> T
let a = id 42 in          // unify T with Int
let b = id true in        // unify T with Bool
...
```

Problem:

To achieve code reuse, we need to assign a polymorphic type to `id`

Type scheme

- Type scheme, written as $\forall X_1 \dots X_n. T$
- Restrictions
  - Predicative: the quantified type variable X in $\forall X. T$ cannot be a type scheme
  - Rank-1 (prenex): type scheme cannot appear on the left-hand sides of arrows
    - $(\forall X. X \to X) \to$ Int is rank-2
- Note the difference between
  - quantified type variables: $\forall X. X \to X$
  - free type variables (unification variables): $X \to X$

idea
基础软件中心

Intuition

To run type inference on `let x = t_1 in t_2`

1. Infer the type of `t_1 : T_1`, where unification has been applied whenever possible
2. **Generalize** the free type variables remaining in `T_1`

- for example, if `T_1 = X -> X`, we get the **type scheme** `forall X. X -> X`

3. Extend the typing environment to record the type scheme for `x`
4. Each time we encounter an occurence of `x` in `t_2`, the type scheme is **instantiated**

- for example, `forall X. X -> X` is instantiated to `X_1 -> X_1`
- for another occurence, it is instantiated to `X_2 -> X_2`

Unsound generalization

Consider the example:

```
let h = fun f -> let g = f in g(42)
in h (true)
```

is supposed to give a type error

- Expected type for `h` : `forall X. (Int -> X) -> X`
- Acutal type for `h` : `forall X Y. X -> Y`

Unsound generalization

Consider the simplified example

```
fun x -> let y = x in y
```

- Expected type: `forall X. X -> X`
- Actual type: `forall X Y. X -> Y`

The solution is **not** to generalize type variables in `T_1` that are also mentioned in the typing environment

Typing rule

- Generalization

$$\frac{\Gamma \vdash t_1 : T_1 \qquad \Gamma, x : \mathrm{GEN}(\Gamma, T_1) \vdash t_2 : T_2}{\Gamma \vdash \mathrm{let}\ x = t_1\ \mathrm{in}\ t_2 : T_2}\text{T-Let}$$

- Instantiation

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

Implementation

idea
基础软件中心

- Type scheme

```
type rec typ = TInt | TBool | TArr(typ, typ) | TVar(ref<tvar>)
                    | QVar(string) // quantified type variable
and tvar = Nolink(string) | Linkto(typ)
```

For example:

- `forall X. X -> X` is represented as `TArr(QVar(X), QVar(X))`
- `X_1 -> X_1` is represented as `TArr(TVar(X_1), TVar(X_1))`

Note:

- This only works for rank-1 polymorphism
- `QVar` cannot be used for unification

Implementation

- syntax directed

```
let inst = (ty: typ): typ => { ... }
let gen = (ty: typ, ctx: ctx): typ => { ... }
let rec check_expr = (ctx: ctx, expr: expr): typ => {
  | ...
  | Var(x) => inst(lookup(x, ctx))
  | Let(x, t1, t2) => {
      let ty1 = check_expr(ctx, e1)
      let ctx' = list{ (x, gen(ty1, ctx)), ...ctx }
      let ty2 = check_expr(ctx', e2)
      t2
    }
}
```

Instantiate

- `instantiate` replaces `QVar(...)` with fresh `TVar(Nolink(...))`

```
let inst = (ty: typ): typ => { ... }
```

- For example, `QVar(X) -> QVar(Y) -> QVar(X)` is instantiated to `TVar(X_1) -> TVar(Y_1) -> TVar(X_1)`
- Straightforward implementation by maintaining a map for substitution

Generalize

- `generalize` replaces `TVar(Nolink(...))` with `QVar(...)` depend on the typing context

```
let free_vars_in_ctx(ctx): list<string> => { ... }
let gen = (ty: typ, ctx: ctx): typ => {
  let free_vars = free_vars_in_ctx(ctx)
  let rec go = (ty: typ, subst): (typ, subst) => { ... }
  fst(go(ty, list{}))
}
```

- Inefficient to calculate the free type variables repeatedly

Level-based approach

When checking `let x = t_1 in ...`, suppose we have `t_1 : T_1`

- Question: how to tell whether a tvar in `T_1` is a free variable in the typing context
- Observation: a tvar appears free in the type context if it is created before typing `let`
- Key idea:
    - classify type variables according to where they are created
    - use level to track where the tvar is created

Level-based approach

- Key idea:
    - classify type variables according to where they are created
    - use level to track where the tvar is created
- For example,

```
fun x ->                              level 0
  let y = x                level 1
in y
```

Level-based approach

- Key idea:
  - classify type variables according to where they are created
  - use level to track where the tvar is created
- For another example,

```
fun x ->                         level 0
  let y = fun z -> x   level 1
in y
```

- The inferred type is: `forall X Y. X -> Y -> X`

Example

```
// expected [h: forall X. (Int -> X) -> X]
let h = fun f -> let g = f in g(42) in h (true)
```

```
typing let h = ... in h (true)
// level = 1
1. typing fun f -> let g = f in g(42)
   1.1 create tvar
   1.2 typing let g = f in g(42)
     // level = 2
     1.2.1 typing f
     // level = 1
     1.2.2 generalize
     1.2.3 typing g(42)
// level = 0
2. generalize
3. typing h(true) -- type error
```

Implementation

- Note the `int` in type variable

```
type rec typ = TInt | TBool | TArr(typ, typ)
  | TVar(ref<tvar>) | QVar(string)
and tvar = Nolink(string * int) | Linkto(typ)
```

- To keep track of where the new type variable is created

```
let new_tvar = (level: int): typ => {
  let name = fresh_name()
  TVar(ref(Nolink(name, level)))
}
```

Implementation

- Level manipulation

```
let rec check_expr = (ctx, expr, level: int): typ => {
  switch expr {
  | Let(x, t1, t2) => {
      let ty1 = check_expr(ctx, e1, level + 1) // increase level
      let ctx' = list{(x, gen(ty1, level)), ...ctx}
      let ty2 = check_expr(ctx', e2, level) // restore level
      ty2
    }
  | ...
  }
}
```

Implementation

基础软件中心

- Generalization: compare levels

```
let gen = (ty: typ, level: int): typ => {
  ...
  | TVar(link) => {
    match !link with
    | Nolink(name, ty_level) when ty_level > level {
      ...
    }
  }
}
```

- Instantiation
  - need the current level to create new tvar

```
let inst = (ty: typ, level: int): typ => { ... }
```

Implementation

- Unification
  - equating two types
  - occur check
  - adjust the level of tvar

For example,

- unify `ty1 = TVar(X_1,1)` with `ty2 = TVar(Y_1,2)`
  - result: `ty1 = TVar(ty2)` and `ty2 = TVar(Y_1, 1)`
- unify `ty1 = TVar(X_1, 2)` with `ty2 = TArr(TVar(Y_1, 1), TVar(Z_1, 3))`
  - result: `ty1 = TVar(ty2)` and `ty2 = TArr(TVar(Y_1, 1), TVar(Z_1, 2))`

Implementation

idea
基础软件中心

- Unification
  - equating two types
  - occur check
  - adjust the level of tvar

```
let prune_level (level: int, ty: typ) => { ... }
let rec unify = (t1: typ, t2: typ): unit => {
  switch (t1', t2') { // path compression omitted
  | (TVar(tvar), t) | (t, TVar(tvar)) =>
      assert !(occurs(tvar, t)) // error report
      prune_level(level_of(tvar), t) // adjust level in type t
      tvar := Linkto(t)
  | ...
  }
}
```

Value restriction

```
let x = ref (fun x -> x) in
x := fun x -> x + 1;
!x true
```

- Unsound if we generalize the type of `x` to `forall X. Ref(X -> X)`
- Only generalize when the right hand side is syntactically a value

Let-polymorphism

- Strength
  - decidable: no annotation required
  - efficient: almost linear
- Weakness
  - complex interaction with subtype, etc.
  - unfriendly error messages
  - hard to generalize to more expressive type systems

Homework

- Complete the implementation for let-polymorphism
- Think about how to handle recursive functions

  Note that we don't allow polymorphic recursive, i.e. something like

```
let rec f = fun x ->
  if true then 22
  else f 7 + f false
in ...
```

idea
基础软件中心

Recommonded reading and references

[1] Efficient and insightful generalization

[2] Section 6 in Programming Language Concepts