CMSC 22600 **Compilers for Computer Languages** **Handout 4**
Autumn 2016 **November 1, 2016**

Basic polymorphic typechecking

1 Introduction

This handout presents the core of the ML polymorphic type system, and two algorithms for the system.

2 The basic ML type system

The core ideas of the ML type system can be presented in terms of a simple extended λ-calculus:

$$\begin{array}{rcll} e & ::= & x & \text{variable} \\ & | & \lambda\, x.e & \lambda \text{ abstraction} \\ & | & (e\ e') & \text{application} \\ & | & \texttt{let}\, x = e\, \texttt{in}\, e' & \text{let binding} \end{array}$$

The set of types ($\tau \in \text{TY}$) is defined by:

$$\begin{array}{rcll} \tau & ::= & \iota & \text{type constant} \\ & | & \alpha & \text{type variable} \\ & | & (\tau_1 \rightarrow \tau_2) & \text{function type} \end{array}$$

and the set of *type schemes* ($\sigma \in \text{TySCHEME}$) is defined by:

$$\begin{array}{rcl} \sigma & ::= & \tau \\ & | & \forall \alpha.\sigma \end{array}$$

The type schema $\sigma = \forall\alpha_1.\forall\alpha_2 \cdots \forall\alpha_n.\tau$ is abbreviated as $\forall\alpha_1\alpha_2 \cdots \alpha_n.\tau$. The type variables $\alpha_1, \ldots, \alpha_n$ are said to be *bound* in σ. A type variable that occurs in τ and is not bound is said to be *free* in σ. We write $\text{FTV}(\sigma)$ for the free type variables of σ. If $\text{FTV}(\tau) = \emptyset$, then τ is said to be a *monotype*. A *type environment* is a finite map from variables to type schemes

$$\text{TE} \in \text{TyEnv} = \text{Var} \xrightarrow{\text{fin}} \text{TySCHEME}$$

It is also useful to view a type environment as a finite set of *assumptions* about the types of variables. The set of *free type variables* of a type environment TE is defined to be

$$\text{FTV}(\text{TE}) = \bigcup_{\sigma \in \text{rng}(\text{TE})} \text{FTV}(\sigma)$$

$$\frac{x \in \mathrm{dom}(\mathrm{TE}) \quad \mathrm{TE}(x) \succ \tau}{\mathrm{TE} \vdash x : \tau}$$

$$\frac{\mathrm{TE} \pm \{x \mapsto \tau'\} \vdash e : \tau}{\mathrm{TE} \vdash \lambda\, x.e : (\tau' \to \tau)}$$

$$\frac{\mathrm{TE} \vdash e_1 : (\tau' \to \tau) \quad \mathrm{TE} \vdash e_2 : \tau'}{\mathrm{TE} \vdash (e_1\ e_2) : \tau}$$

$$\frac{\mathrm{TE} \vdash e_1 : \tau' \quad \mathrm{TE} \pm \{x \mapsto \textsc{Clos}_{\mathrm{TE}}(\tau')\} \vdash e_2 : \tau}{\mathrm{TE} \vdash \mathbf{let}\, x = e_1\, \mathbf{in}\, e_2 : \tau}$$

Figure 1: Type inference rules

The *closure*, with respect to a type environment TE, of a type τ is defined as

$$\textsc{Clos}_{\mathrm{TE}}(\tau) = \forall \alpha_1 \cdots \alpha_n.\tau$$

where $\{\alpha_1, \ldots, \alpha_n\} = \mathrm{FTV}(\tau) \setminus \mathrm{FTV}(\mathrm{TE})$.

A *substitution* is a map from type variables to types. A substitution S can be naturally extended to map types to types as follows:

$$\begin{array}{rcl} S\iota & = & \iota \\ S\alpha & = & S(\alpha) \\ S(\tau_1 \to \tau_2) & = & (S\tau_1 \to S\tau_2) \end{array}$$

Application of a substitution to a type schema respects bound variables and avoids capture. It is defined as:

$$S(\forall \alpha_1 \cdots \alpha_n.\tau) = \forall \beta_1, \ldots, \beta_n.S(\tau[\alpha_i \mapsto \beta_i])$$

where $\beta_i \notin \mathrm{dom}(S) \cup FTV(\mathrm{rng}(S))$. Application of a substitution S to a type environment TE is defined as $S(\mathrm{TE}) = S \circ \mathrm{TE}$. A type τ' is an *instance* of a type scheme $\sigma = \forall \alpha_1 \cdots \alpha_n.\tau$, written $\sigma \succ \tau'$, if there exists a finite substitution, S, with $\mathrm{dom}(S) = \{\alpha_1, \ldots \alpha_n\}$ and $S\tau = \tau'$. If $\sigma \succ \tau'$, then we say that σ is a *generalization* of τ'. Some examples are:

$$\begin{array}{rcl} \forall \alpha.\alpha & \succ & \tau, \text{ for any } \tau \in \textsc{Ty} \\ \forall \alpha, \beta.(\alpha \to \beta) & \succ & (\alpha \to \alpha) \\ \forall \alpha, \beta.(\alpha \to \beta) & \succ & (\alpha \to \mathbf{int}) \end{array}$$

The typing system is given as a set of rules from which sentences of the form "$\mathrm{TE} \vdash e : \tau$" can be inferred. This sentence is read as "e has the type τ under the set of typing assumptions TE." The rules are given in Figure 1

It is worth noting that there is exactly one typing rule for each syntactic form; thus, if we have a proof of $\mathrm{TE} \vdash e : \tau$, for some e, the form of e uniquely specifies which typing rule was the last applied in the deduction. This is the formulation of [Tof88] and differs from the system of [DM82], which has judgements that infer type schemas for expressions and rules for instantiating and generalizing type schemas. A proof of the equivalence of these two systems can be found in [CDDK86].

This type inference system is decidable; there exists an algorithm, called algorithm **W** [DM82] that infers the *principal type* (i.e., most general under the relation $\succ$) of an expression. Algorithm **W** is both sound and complete with respect to the inference system. See [DM82] or [Tof88] for the proof details.

3 Algorithm W

The SML code for Algorithm **W** is given in Figure 2. This relies on modules to implement types,

```
fun algW (env, e) = (case e
       of (L.Var x) => let
            val sigma = E.lookup(env, x)
            in
              (S.id, T.freshTy sigma)
            end
        | (L.App(e1, e2)) => let
            val (s1, t1) = algW(env, e1)
            val (s2, t2) = algW(S.applySubstToEnv(s1, env), e2)
            val beta = T.freshTyVar()
            val s3 = U.unify(S.applySubstToTy(s2, t1), T.FnTy(t2, beta))
            in
              (S.compose(s3, S.compose(s2, s1)), S.applySubstToTy(s3, beta))
            end
        | (L.Abs(x, e')) => let
            val beta = T.freshTyVar()
            val (s1, t1) = algW(E.insert(env, x, T.TyScheme([], beta)), e')
            in
              (s1, T.FnTy(S.applySubstToTy(s1, beta), t1))
            end
        | (L.Let(x, e1, e2)) => let
            val (s1, t1) = algW(env, e1)
            val xTy = T.closeTy(
                        E.freeVarsOfEnv(S.applySubstToEnv(s1, env)),
                        t1)
            val (s2, t2) = algW(E.insert(env, x, xTy), e2)
            in
              (S.compose(s2, s1), t2)
            end
      (* end case *))
```

Figure 2: Algorithm W

environments, substitutions, and unification. The unification algorithm, which is owed to Alan Robinson, is given in Figure 3. The `unify` function returns the *most general unifier*. By this, we mean that if `unify(`τ`,` τ'`)` returns S, and if R unifies τ and τ' (i.e., $R(\tau) = R(\tau')$), then there is a substitution R', such that $S = R' \circ R$. The function `occurs` is used for what is called the "*occurs check*." Since recursive types are not allowed, one cannot unify a type variable with a type that contains it. The occurs check detects this situation and avoids a possible infinite loop.

```
fun occurs (v1, T.VarTy v2) = (v1 = v2)
  | occurs (v, T.BaseTy _) = false
  | occurs (v, T.FnTy(ty1, ty2)) = occurs(v, ty1) orelse occurs(v, ty2)

fun unify (T.VarTy v1, ty2 as (T.VarTy v2)) =
      if (v1 = v2) then S.id else S.singleton(v1, ty2)
  | unify (T.VarTy v1, ty2) =
      if (occurs(v1, ty2)) then raise Unify else S.singleton(v1, ty2)
  | unify (ty1, T.VarTy v2) =
      if (occurs(v2, ty1)) then raise Unify else S.singleton(v2, ty1)
  | unify (T.BaseTy a, T.BaseTy b) =
      if (a = b) then S.id else raise Unify
  | unify (T.FnTy(ty1, ty1'), T.FnTy(ty2, ty2')) = let
      val s1 = unify (ty1, ty2)
      val s2 = unify (S.applySubstToTy(s1, ty1'), S.applySubstToTy(s1, ty2'))
      in
        S.compose(s2, s1)
      end
  | unify _ = raise Unify
```

Figure 3: Robinson's unification algorithm

4 A better algorithm

Algorithm **W** is not a practical algorithm. It suffers from two sources of inefficiency: the use of explicit substitutions and the need to determine the free variables in the environment when computing the type closure. Most real ML compilers use an algorithm that avoids these costs. In this algorithm, type variables are destructively updated during unification, which avoids the need for substitutions, and the λ-binding depth of variables is remembered as a way to detect which are λ-bound.

In this algorithm, type variables are represented as a record consisting of an unique ID, and a updatable kind (see Figure 4). The kind of a type variable starts out as `UNIV`, and is changed to `INSTANCE`, when the variable is unified to a type. The integer argument to `UNIV` is the λ-binding depth of the variable. The destructive unification algorithm is given in Figure 5. Note that when an `INSTANCE` type variable is encountered, the instance is followed. Also note that when a type variable is unified with a type, the depth of the type is adjusted to the minimum depth; this corresponds to applying the substitution to the variables in the type environment in algorithm **W**.

The typechecking algorithm is given in Figure 6. Like Algorithm **W**, it recursively walks the term being checked, but it takes an extra *depth* argument and does not return a substitution. The `pruneTy` function is used to prune instantiated type variables.

5 The value restriction

The type system in Figure 1 is sound for a pure calculus, but once we add effects, such as assignment, exceptions, or first-class continuations, the system becomes unsound. The easiest way to fix that problem is to introduce the *value restriction*, which replaces the rule for **`let`** with two rules based on whether the right-hand-side of the binding is a *value* or a potentially effectful expression [Wri95]. For this simple language, values (denoted v) are variables (x) and lambda abstractions

```
datatype tyvar = TYVAR of {
    id : int,
    kind : tvar_kind ref
  }

and tvar_kind
  = INSTANCE of ty
  | UNIV of int

and ty
  = VarTy of tyvar
  | BaseTy of string
  | FnTy of (ty * ty)

and ty_scheme
  = TyScheme of (tyvar list * ty)
```

Figure 4: Representation of types

```
fun unify (T.VarTy v1, T.VarTy v2) =
      if (v1 = v2) then () else unifyVars(v1, v2)
  | unify (T.VarTy v1, ty2) = unifyVarWithTy(v1, ty2)
  | unify (ty1, T.VarTy v2) = unifyVarWithTy(v2, ty1)
  | unify (T.BaseTy a, T.BaseTy b) =
      if (a = b) then () else raise Unify
  | unify (T.FnTy(ty1, ty1'), T.FnTy(ty2, ty2')) = (
      unify(ty1, ty2);
      unify(ty1', ty2'))
  | unify _ = raise Unify

and unifyVars (v1 as T.TYVAR{kind=k1, ...}, v2 as T.TYVAR{kind=k2, ...}) = (
      case (!k1, !k2)
       of (T.INSTANCE ty1, _) => unifyVarWithTy(v2, ty1)
        | (_, T.INSTANCE ty2) => unifyVarWithTy(v1, ty2)
        | (T.UNIV d1, T.UNIV d2) => if (d1 < d2)
            then k2 := T.INSTANCE(T.VarTy v1)
            else k1 := T.INSTANCE(T.VarTy v2)
      (* end case *))

and unifyVarWithTy (v as T.TYVAR{kind, ...}, ty) = (case !kind
       of (T.INSTANCE ty') => unify (ty', ty)
        | (T.UNIV d1) => if (occursIn(v, ty))
            then raise Unify
            else let
              val d2 = T.minDepth ty
              in
                if (d1 < d2) then T.adjustDepth(ty, d1) else ();
                kind := T.INSTANCE ty
              end
      (* end case *))
```

Figure 5: Destructive unification

```
fun check (env, depth, e) = (case e
       of (L.Var x) => T.freshTy(E.lookup(env, x))
        | (L.App(e1, e2)) => let
            val ty1 = check (env, depth, e1)
            val ty2 = check (env, depth, e2)
            val beta = T.freshTyVar depth
            in
              U.unify (ty1, T.FnTy(ty2, beta));
              T.pruneTy beta
            end
        | (L.Abs(x, e')) => let
            val beta = T.freshTyVar (depth+1)
            val ty = check(
                    E.insert(env, x, T.TyScheme([], beta)),
                    depth+1, e')
            in
              T.FnTy(T.pruneTy beta, ty)
            end
        | (L.Let(x, e1, e2)) => let
            val ty1 = check (env, depth, e1)
            val xTy = T.closeTy(depth, ty1)
            in
              check (E.insert(env, x, xTy), depth, e2)
            end
      (* end case *))

fun typecheck (env, e) = T.pruneTy(check(env, 0, e))
```

Figure 6: Typechecking with destructive unification

$(\lambda\, x.e)$. When the right-hand-side is a value, we allow the introduction of polymorphism:

$$\frac{\mathrm{TE} \vdash v : \tau' \quad \mathrm{TE} \pm \{x \mapsto \textsc{Clos}_{\mathrm{TE}}(\tau')\} \vdash e : \tau}{\mathrm{TE} \vdash \mathtt{let}\, x = v\, \mathtt{in}\, e : \tau}$$

whereas when it is an expression, we do not introduce polymorphism

$$\frac{\mathrm{TE} \vdash e_1 : \tau' \quad \mathrm{TE} \pm \{x \mapsto \tau'\} \vdash e_2 : \tau}{\mathrm{TE} \vdash \mathtt{let}\, x = e_1\, \mathtt{in}\, e_2 : \tau}$$

References

[CDDK86] Clément, D., J. Despeyroux, T. Despeyroux, and G. Kahn. A simple applicative language: Mini-ML. In *Conference record of the 1986 ACM Conference on Lisp and Functional Programming*, August 1986, pp. 13–27.

[DM82] Damas, L. and R. Milner. Principal types for functional programs. In *Conference Record of the 9th Annual ACM Symposium on Principles of Programming Languages*, January 1982, pp. 207–212.

[Tof88] Tofte, M. *Operational semantics and polymorphic type inference*. Ph.D. dissertation, Department of Computer Science, University of Edinburgh, May 1988.

[Wri95] Wright, A. K. Simple imperative polymorphism. *Lisp and Symbolic Computation*, **8**(4), December 1995, pp. 343–356.