Polymorphic References Revisited 9 March 1987 Mads
(OR: so what about this one, then?)

I shall try to document and make precise the idea that Dave MacQueen and I developed Friday.

We introduce a new kind of type variables $\underline{\alpha}, \underline{\beta}, \ldots$; let me call them <u>strong</u> type variables to avoid confusion with weak type variables and to indicate that there are strong ~~[illegible]~~ constraints on how they can be used. Intuitively a strong type variable stands for a fixed, but unknown mono type. The inference rules will satisfy that if $TE \vdash e : \sigma$ then any strong type variable free in σ is free in TE.* Generalization on strong type variables is permitted at once place only, namely the λ-abstraction.

Consider $e \equiv \lambda x.e'$. If the type of x contains a strong type variable $\underline{\alpha}$, say, then $\underline{\alpha}$ is a legal target for instantiations within e' i.e., we temporarily get a bit of freedom. For instance, ref is a function of type ~~[illegible]~~ $\forall \underline{\beta}. \underline{\beta} \to \underline{\beta}$ ref. In e' ref can get type $\underline{\alpha} \to \underline{\alpha}$ ref, but in itself it ~~must~~ can ~~to~~ only be instantiated to a mono morphic type (provided TE is closed).

* i.e. is (potentially) Λ bound.

1 Types

Types $\tau ::= \text{int} \mid \text{bool} \mid \tau \to \tau \mid \tau \text{ ref} \mid \alpha \mid \underline{\alpha}$
Type Schemes $\sigma ::= \tau \mid \forall \alpha . \sigma \mid \forall \underline{\alpha} . \sigma$

When we need to distinguish between the two kinds of type variables we shall talk about *strong* versus *liberal* type variables. A *mono type* is a type without any type variables. A *strong type* is a type which contains no liberal type variables (but perhaps strong type variables).
In type schemes the order of bound type variables is insignificant even when there are variables of both kinds.

2 Substitution and Generic Instance

A *substitution*, S, is a pair $(S^{(\ell)}, S^{(\mu)})$ of total functions where $S^{(\ell)}$ maps liberal type variables to types and $S^{(\mu)}$ maps strong type variables to strong types.

The *domain* of S is the set of type variables (liberal or strong) on which S isn't the identity.

Let $\sigma = \forall \alpha_1 \cdots \alpha_m \underline{\alpha}_1 \cdots \underline{\alpha}_n . \tau$. Then τ' is an *instance* of σ, written $\sigma > \tau'$, if there exists types $\tau_1, \ldots, \tau_m$ and strong types $\tau'_1, \ldots, \tau'_n$ s.t. $([\tau_i / \alpha_i], [\tau'_j ; \underline{\alpha}_j]) \tau = \tau'$.

This relation extends to relation on type schemes, $\sigma \geq \sigma'$, as usual.

3 A Language

$$e ::= x \mid \lambda x . e \mid e e' \mid \underline{\text{let}}\ x = e\ \underline{\text{in}}\ e'$$

Here ref, :=, and ! are treated as variables. In the initial static environment they have types:

$$\left.\begin{array}{lll} \text{ref} & : & \forall \underline{\alpha}.\ \underline{\alpha} \to \underline{\alpha}\ \text{ref} \\ ! & : & \forall \alpha .\ \alpha\ \text{ref} \to \alpha \\ := & : & \forall \alpha .\ \alpha\ \text{ref} \to \alpha \to \text{unit} \end{array}\right\} \quad \text{liberal } \alpha\ !$$

$$\text{ref} : \forall \underline{\alpha}_1 .\ \underline{\alpha}_1 \to \underline{\alpha}_1 \text{ref}$$

4 The Rules

VAR $$\frac{}{TE \vdash x : TE(x)}$$

LAM $$\frac{TE[x:\tau'] \vdash e : \tau}{TE \vdash \lambda x.e : SCl_{TE}(\tau' \to \tau)}$$

$$\frac{TE \vdash e : \tau' \to \tau \qquad TE \vdash e' : \tau'}{TE \vdash e e' : \tau}$$

$$\frac{TE \vdash e' : \sigma \qquad TE[x:\sigma] \vdash e : \tau}{TE \vdash \underline{\text{let}}\ x = e'\ \underline{\text{in}}\ e : \tau}$$

INST $$\frac{TE \vdash e : \sigma}{TE \vdash e : \sigma'} \quad \sigma \geq \sigma' \wedge SV(\sigma') \subseteq SV(TE)$$

$$\frac{TE \vdash e : \sigma}{TE \vdash e : \forall\alpha.\sigma} \quad \alpha \text{ not free in } TE$$

where SV means "the free strong type variables of", TE maps program variables to type schemes and $SCl_{TE}\,\tau$ = the strong TE closure of τ is τ closed ~~of~~ up with all those of its strong type variables that are not ~~in~~ free in TE.

Lemma 1 Everything one can infer in the purely applicative system (Damas/Milner paper) one can infer in the above system.

Proof Every DM type (i.e. "type" in the Damas/Milner sense) is a type. Every DM substitution is a substitution which neither touches nor produces strong type variables. Every DM type env is a type env without strong type variables. Thus, in ~~the~~ Rule LAM, $Gen_{TE}(\tau' \to \tau)$ is just $\tau' \to \tau$. In rule INST $SV(\sigma') = \emptyset$, so $SV(\sigma') \subseteq SV(TE)$ is no real restriction. ~~[illegible]~~ If in DM $\sigma \geq \sigma'$ then $\sigma \geq \sigma'$ without strong type variables. The rest of the rules are identical. $\square$

Lemma 2 If $TE \vdash e : \sigma$ then $SV(\sigma) \subseteq SV(TE)$.

Proof By induction on the depth of inference. The only case worth spelling out is the let-case:

$$\frac{TE \vdash e' : \sigma \qquad TE[x:\sigma] \vdash e : \tau}{TE \vdash \text{let } x = e' \text{ in } e \; : \tau}$$

By induction $SV(\sigma) \subseteq SV(TE)$. Thus $SV(TE[x:\sigma]) \subseteq SV(TE)$. By induction $SV(\tau) \subseteq SV(TE[x:\sigma])$. Thus $SV(\tau) \subseteq SV(TE)$. ■

We emphasize that there is only one place where generalisation on strong type variables can happen namely at λ-abstr. If the abstraction is in an application

$$(\lambda x. e) e' \quad \text{or} \quad e' (\lambda x. e)$$

then – assuming for simplicity that TE is closed – we will have to make the type of $\lambda x.e$ monomorphic before proceeding. However the polymorphism can be used in a let:

$$\underline{\text{let}}\ f = \lambda x.e$$
$$\underline{\text{in}}$$
$$\ldots f e' \ldots\ f e''$$

with different instantiations.

The system is general enough to give us reverse : $\forall \underline{\alpha}.\ \underline{\alpha}\ list \to \underline{\alpha}\ list$ for the imperative

```
fun reverse (l) =
    let v = ref l ; h = ref []
    in  while !v ≠ [] do
              (h := hd (!v) :: (!h) ; v := tl(!v)) ;
        !h
    end
```

On the negative side, consider

$$(\lambda x.\ \lambda y.\ e)\ e'$$

Still assuming TE closed this expression will not have a type with no strict type variables. The next section describes a tempting, but unsound, way of getting around this.

5 A tempting, but unsound "generalization" of the rules

It would be tempting to replace LAM by the ordinary

$$LAM_0: \quad \frac{TE[x:\tau] \vdash e : \tau'}{TE \vdash \lambda x.e : \tau \to \tau'}$$

and then add

$$GEN': \quad \frac{TE \vdash e : \forall \underline{\alpha}_1 \cdots \underline{\alpha}_m \alpha_1 \cdots \alpha_n . \tau \to \tau'}{TE \vdash e : \forall \underline{\alpha}\, \underline{\alpha}_1 \cdots \underline{\alpha}_m \alpha_1 \cdots \alpha_n . \tau \to \tau'}$$

$(\underline{\alpha}$ not free in $TE)$

Interestingly this gives an unsound system.
We can safely assume that anything e evaluates to is a closure. But it may happen in a non-trivial computation in which references are created and embedded in the closure. This happens with the following faulty program (a variation on the store/fetch example) which is typable with GEN':

$\underline{\alpha} \to \underline{\alpha}$ $\qquad (\underline{\alpha} \to \underline{\alpha})$ ref

let $f = ((\lambda x.$ **let** $loc = ref\ x$

$(\underline{\alpha} \to \underline{\alpha}) \to (\underline{\alpha} \to \underline{\alpha})$ **in** $\lambda y.$ **let** $a = !\,loc$

in $(loc := y ; a)$

$\underline{\alpha} \to \underline{\alpha}$ $\qquad \underline{\alpha} \to \underline{\alpha}$

$\forall \underline{\alpha} . (\underline{\alpha} \to \underline{\alpha}) \to (\underline{\alpha} \to \underline{\alpha})$)

using GEN' $\qquad (\lambda x . x)$ $\underline{\alpha} \to \underline{\alpha}$

)

in $(f$ succ $1 ;\ f (\lambda x . x)$ true$)$

int $\qquad$ bool

(applies succ to true)

THE END.