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

We introduce a new kind of type variables $\alpha, \beta, \ldots$; let me call them strong type variables to avoid confusion with weak type variables and to indicate that there are strong type checking 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 $\Gamma \vdash e : \sigma$ then any strong type variable put in $\sigma$ is free in $\Gamma e$. Generalization on strong type variables is permitted at one place only, namely the $\lambda$ abstraction.

Consider $e = \lambda x.e'$. If the type of $x$ contains a strong type variable $\alpha$, say, then $\alpha$ is a legal target for instantiations within $e'$, i.e., we temporarily get a bit of freedom. For instance, $\text{ref}$ is a function of type $\forall \beta. \beta \rightarrow \beta \text{ ref}$. So in $e'' \text{ ref}$ can get type $\alpha' \rightarrow \text{ ref}$, but in itself it can only be instantiated to a mono morphic type (provided $\Gamma e''$ is closed).

* i.e. is (potentially) a bound.
1. Types

Types \( \tau ::= \text{int} \mid \text{bool} \mid \tau \rightarrow \tau \mid \tau \times \tau \mid \alpha \mid \sigma \)

Type Schemes \( \sigma ::= \tau \mid \forall \alpha. \sigma \mid \forall \alpha. \tau \)

When we need to distinguish between the two kinds of type variables we shall talk about strong versus liberal type variables. A strong 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 \((S^{(t)}, S^{(u)})\) of total functions where \( S^{(t)} \) maps liberal type variables to types and \( S^{(u)} \) maps strong type variables to strong types.

The domain of \( S \) is the set of type variables (liberal or strong) on which \( S \) is defined. The identity \( S \)

\[ S^{(t)} \sigma = \forall \alpha_1 \ldots \forall \alpha_n \exists \beta \exists \gamma \ldots \exists \gamma \cdot \tau . \]

Then \( S \) is an instance of \( \sigma \) within \( \sigma \rightarrow S \), if there exist types \( \tau_1, \ldots, \tau_n \) and strong types \( \tau_1, \ldots, \tau_n \) such that \((\lbrack \tau_1 ; \alpha_1 \rbrack, \ldots, \lbrack \tau_n ; \alpha_n \rbrack) \tau = \tau'. \)
This relation extends to one relation on type schemes, \( \sigma \geq \sigma' \), as usual.

3. A Language

\[ e ::= x | \lambda x.e | ee' | let x=e in e' \]

The \( \text{ref} \), \( := \), and \( ! \) are treated as variables. In the initial static environment they have types:

\[ \text{ref} : \forall \alpha. \alpha \rightarrow \alpha \text{ref} \]
\[ ! : \forall \alpha. \alpha \text{ref} \rightarrow \alpha \]
\[ := : \forall \alpha, \beta. \alpha \text{ref} \rightarrow \beta \rightarrow \text{unit} \]

\[ \text{ref} : \forall \alpha_1. \alpha_1 \rightarrow \alpha_1 \text{ref} \]
4. The Rules

VAR
\[ \text{TE} \vdash x : \text{TE}(x) \]

LAM
\[ \text{TE}[x:z'] \vdash e : z \]
\[ \text{TE} \vdash \lambda x . e : \text{SGL}_{\text{TE}}(z' \rightarrow z') \]
\[ \text{TE} \vdash e : z' \rightarrow z \]
\[ \text{TE} \vdash e' : z' \]
\[ \text{TE} \vdash ee' : z \]

INST
\[ \text{TE} \vdash e' : \sigma \]
\[ \text{TE}[x: \sigma] \vdash e : z \]
\[ \text{TE} \vdash \delta \lambda x : e' \in e : z \]

\[ \text{INST} \]
\[ \text{TE} \vdash e : \sigma \]
\[ \sigma \rightarrow \sigma' \wedge \text{SV}(\sigma') \subseteq \text{SV}(\text{TE}) \]
\[ \text{TE} \vdash e : \forall x . \sigma \]

where \( \text{SV} \) means "the strong type variables of",
\( \text{TE} \) maps program variables to type schemas,
and \( \text{SGL}_{\text{TE}}(z' \rightarrow z) \) is the strong \( \text{TE} \) closure of \( z' \rightarrow z \)

is \( \delta \) closed at \( \delta \) with all those of its strong

type variables that are not in \( \delta \) in \( \text{TE} \).
Lemma 1. Everything one can infer in the purely applicative system (Damas/Milner style) 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 with holes no produces strong type variable. Every DM type var is a type var without strong type variable. Thus, in the Rule LM,
\[ S : (T' \rightarrow T) \text{ is just } T' \rightarrow T. \]

By rule INST
\[ SV(\sigma') = \emptyset, \text{ or } SV(\sigma') \subseteq SV(TE); \text{ no real restriction.} \]

If \( \sigma \rightarrow \sigma' \) in DM \( \sigma \rightarrow \sigma' \) then \( \sigma \rightarrow \sigma' \) without strong type variable. 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:
\[ TE \vdash e' : \sigma, TE \left[ x : \sigma \right] \vdash e : \tau \]
\[ TE \vdash \text{let } x = e' \text{ in } e : \tau \]

By induction \( SV(\sigma) \subseteq SV(TE) \). Thus \( SV(TE \left[ x : \sigma \right]) \subseteq \)
\[ SV(TE). \]

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

We emphasize that there is only one place where generalisation on strong type variable can happen, namely at \( \lambda - \) abstr. If the abstraction is in an application
\[ (\lambda x.e)e' \]
\[ \sigma \]
\[ e'(\lambda x.e) \]
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:

\[
\text{let } f' = \lambda x.e' \\
\text{in } \\
... f'e' ... f'e''
\]

with different instantiations.

The system is general enough to give us
reverse: \( V_k. k \text{ let } x \rightarrow x \text{ let for the imperative} \)

\[
\text{fun reverse (l) =} \\
\text{let } n = \text{ref } l ; h = \text{ref } [] \\
\text{in while } !n \neq [] \text{ do} \\
\quad (h := \text{hd } (!n) :: (!h) ; n := M (!n)) ; \\
\quad !h \\
\text{end}
\]

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 variable. The rigid
system describes a tempting, but unsound,
way of getting around this.
5. A tempting, but unsound "generalization" of the rule

It would be tempting to replace \( \text{LAM} \) by the rule

\[
\text{LAM}_0: \quad \frac{\text{TE}[x; z] \vdash e : t}{\text{TE} \vdash \lambda x. e : z \rightarrow t'}
\]

and then add

\[
\text{GEN}': \quad \frac{\text{TE} \vdash e : \forall x_1 \ldots \forall x_n . t \rightarrow t'}{\text{TE} \vdash e : \forall x_1 \ldots \forall x_n . t \rightarrow t'}
\]

(\( x \) not free in \( e \))

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 show/fetch example) which \( e \) is typable with \( \text{GEN} \):

\[
\begin{array}{c}
\forall x_1 \ldots \forall x_n \exists z \forall y . \exists a \vdash e \rightarrow z \\
\text{let } f = (\lambda x. \text{let } \text{loc} = \text{ref } x \\
(\text{ref } y) \uparrow \uparrow \text{ in } \lambda y. \text{let } a = ! \text{loc} \\
\text{let } (\text{loc } := y; a) \uparrow \uparrow \text{ in } (\text{loc } := y; a) \\
\forall y. (\text{ref } y) \rightarrow (\lambda x. x) \rightarrow z \\
\text{using } \text{GEN} \quad (\lambda x. x) \rightarrow z \\
\uparrow \uparrow \text{ in } (\lambda x. x) \rightarrow \uparrow \uparrow \text{ and } \uparrow \uparrow (\text{apply } \uparrow \uparrow \text{ suc } z) \rightarrow \uparrow \uparrow \text{ true}
\end{array}
\]

THE END.