Modified Damas Algorithm for Typechecking with References

Type variable attributes:

weak : bool

binding_level : nat

If a type variable is weak, then it is involved in the type of the contents of a reference value. Weak variables can be generic when they occur

(a) outside the "scope" of the ref value they are associated with, [scope = "extent"?]

(b) associated with a potential (or deceased) rather than actual ref value.

[(a) and (b) are probably two ways of saying the same thing.]

The binding_level of an ordinary type variable represents the minimum nesting depth of the λ-bound variables with which the type variable is associated in the current type assignment. A generic (i.e. free or universally bound) type variable has binding_level ∞. An ordinary type variable is non-generic when [its binding_level (and the current λ-nesting depth) indicate that] it is involved in the type of a

variable which is currently λ-bound. (i.e. the current expression being typed is in the scope of a binder λx and the type variable in question occurs in the type assigned to x by the current type assignment). This status can be determined by comparing the binding-level of the type variable with the current λ-nesting level (i.e. that of the atom currently being typed).* *

A non-generic type variable represents an unknown but "bound" (and therefore "particular" or "definite" (or "constant") type. It can be instantiated to a type term, thus further determining the unknown but definite type. But it cannot be duplicated or copied for each occurence of a variable whose type contains it, as can a generic variable. I.e. within the scope of a λx binder, the type of x acts as an unknown (or partially unknown) but constant type.

Notation: A weak variable is indicated by using a $\sim$, and the current binding level is indicated by a subscript. Thus $\tilde{t}_2$ is a weak type variable

---

* Generic variables will always be given a binding-level of ∞ as soon as they are seen to be generic (they are universally quantified as soon as they are poped off the type assignment.)

The typing algorithm:

$A \in TA$ type assignments
$e \in Exp$ expressions
$level \in nat$ λ-binding nesting
$\mathcal{T}$: Exp x TA x nat → Texp.

$\mathcal{T}(x, A, \ell)$: generic instance of $A[\![x]\!]$
{ $A[\![x]\!]$ if binding style of x is λ }

$\mathcal{T}(e_1, e_2, A, \ell)$:
let $\tau_1 = \mathcal{T}(e_1, A, \ell)$
for each weak type variable $\tilde{\alpha}$ in τ_1
binding_level($\tilde{\alpha}$) = min(binding_level($\tilde{\alpha}$), ℓ)
let $\tau_2 = \mathcal{T}(e_2, A, \ell)$
unify($\tau_1, \tau_2 \to \beta_\infty$) [new β]
return(β) [i.e. its instantiation].

$\mathcal{T}(\lambda x.e, A, \ell)$:
let $\beta_{\ell+1}$ be a new type variable
let $\tau = \mathcal{T}(e, A[\beta_{\ell+1}/x], \ell+1)$
for each type variable α in β such that
binding_level(α) > ℓ [will only be $\ell+1$ and ∞]
set binding_level(α) = ∞ (marking as generic)
return($\beta \to \tau$)

$\mathcal{T}$(let $x = e$ in e_1, A, l):
$\mathcal{T}(e_1, A[\mathcal{T}(e, A, l)/x], l)$

Generic Variables and Generic Instances.

A type variable α is generic if binding_level(α) $> l$, (the current λ-nesting level). [actually if $= \infty$]

Defn: σ' is a generic instance of σ if σ' is a copy of σ with each generic variable of σ replaced by a new copy of itself.

Propagation of Type-variable attributes during Unification:

When instantiating α to τ do

(i) if α is weak, make all variables of τ weak.

(ii) for each β in τ, set
binding_level(β) = min(binding_level(β), binding_level(α)).

Justification of the Algorithm

The use of binding-level compared with λ-nesting to determine genericity has already been discussed.

The main point of interest is the case of application, where the binding-levels of the weak type variables in the operator type are set to the current λ-nesting level (unless they already have a smaller binding level). The reason for this is that the invocation of a function with weak (generic - since only weak generic variables are affected) variables in its type involves the (potential) internal creation of new ref values. The extent of these values (and thus the non-genericity of their associated weak type variables) is limited to the current λ-scope (unless their types unify with less generic variables with smaller binding-levels - in which case they wouldn't have been generic at this level). Therefore when we leave the lambda scope we can be sure that no actual ref values associated with these weak variables can exist, and so they can be made generic once again.

Conjecture: Unweakening.

In certain circumstances type variables remain weak even though there seems to be no justification. For example:

$$\lambda x.\ !(\mathrm{ref}\ x) \ :\ \forall \alpha_{\infty}.\ \alpha_{\infty} \to \alpha_{\infty}$$

I conjecture that if after typing a lambda expression we have a generic weak type variable which does not occur in a ref type subexpression, then that type variable can be made non-weak. [A stronger conjecture is that this can be done if it doesn't occur in a ref subexpression of the result type of the function.].

Types of ref operators

$$\mathrm{ref} : \forall \alpha.\ \alpha \to \alpha\ \mathrm{ref}$$
$$:= \ :\ \forall \alpha.\ \alpha\ \mathrm{ref} \times \alpha \to \mathrm{unit} \quad \text{(assignment)}$$
$$! \ :\ \forall \alpha.\ \alpha\ \mathrm{ref} \to \alpha \quad \text{(contents)}$$

Only ref introduces weak type variables, because only it creates reference values.