Full First-Order Type Inference

Wednesday, August 13, 2014 12:42 PM

In this wiki, we extend type inference to existential types of transferable regions. The basic rule that guides inference is that the type of the transferable region is by default an existential with bound region name (eg: $\exists\rho_0.\,\mathrm{Region}[\rho_0]\langle\pi_0^{a}\rangle\langle\tau_0\rangle$). That is, when we generate a region type template for C# type $\mathrm{Region}\langle\ldots\rangle$, we always generate $\exists\rho_0.\,\mathrm{Region}[\rho_0]\langle\pi_0^{a}\rangle\langle\ldots\rangle$, where ρ_0 and π_0^{a} are both new. The default elaboration of transferable region type to an existential type simplfies inference, while not having a significant adverse impact on the expressivity. For example:

1. LinkedList<Region<...>> is by default elaborated to LinkedList<π_0^{a},π_1^{a}><$\exists\rho\,.\,\mathrm{Region}[\rho\,]\langle\pi_1^{a}\rangle\langle\ldots\rangle$>. Here, the elaboration has rightly decided to assign existential type to region handlers stored linked list.
2. *void foo(Region<...>r)* is appropriately elaborated to *void foo<...>(*$\exists\rho\,.\,Region[\rho\,]\langle\ldots\rangle\langle\ldots\rangle$ *r)* rather than elaborating it to *void foo<...,ρ>(*$Region[\rho\,]\langle\pi_1^{a}\rangle\langle\ldots\rangle$ *r)*. The latter requires ρ to be live (open) when *foo* is called, an assumption which we don't want functions like *foo* to make. We insist that functions explictly open any transferable region handlers they receive, rather than assuming that they are already open and that they satisfy certain outlives relationships.
3. *Class B {Region<...>r; ...}* is elaborated to *Class B*$\langle\ldots\rangle$ *{*$\exists\rho.\,Region[\rho\,]\langle\ldots\rangle\langle\ldots\rangle$ *r; ...}* rather than *Class B*$\langle\ldots,\rho\rangle$ $\{Region[\rho\,]\langle\ldots\rangle\langle\ldots\rangle$ *r; ...*$\}$. The latter requires ρ to be live (open) before any object of type *B* is created. Consequently, it disallows the common coding idiom where the constructor of a class creates and assigns a new transferable region to its instance variable.

To use a region handler that is stored in the instance variable of the class, a method-local variable needs to be initialized to the instance variable holding region handler, at which point the existential is *unpacked* and the name of transferable region is materialized. We constrain our source language such that only local variables of unpacked transferable region type (eg: $\mathrm{Region}[\rho_1]\langle\pi_1^{a}\rangle\langle\tau_1\rangle$, where ρ_1 is in scope) are allowed in *open* and *openAlloc* statements. Consequently, a transferable region *has* to be referred by a local variable before opening, and the corresponding variable declaration is elaborated by our algorithm to *unpack* statement for transferable region handler.

The Source Language

$cn \;\in Class\ Names\ (A,B,C\ \ldots)$
$mn \in Method\ Names\ (m,n,\ldots)$
$x,f \in Variables, fields$
$n \in Integers$
$Program = (CT,e)$
$c\ \ ::=\ \ n \mid () \mid true \mid false \mid Null$ //Constants
$N\ \ ::=\ \ cn\langle\overline{T}\rangle$ //Instantiated class type
$C\ \ ::=\ \ \text{class}\ cn\langle\overline{\alpha \lhd N}\rangle \lhd N\,\{\overline{T\,f}; k\,; \overline{d}\}$ //Class Definitions
$k\ \ ::=\ \ cn\ (\overline{T\,x})\{\ \text{super}\ (\bar{v});\ \overline{\text{this}.\,f = v;}\}$ //Constructors
$d\ \ ::=\ \ T\ mn\,(\overline{T\,x})\,\{s;\ \text{return}\ e;\}$ //Methods
$T ::= \alpha \mid N \mid Object \mid Region\langle T\rangle \mid int \mid bool \mid unit$ // Types
$v\ \ ::= c \mid x \mid new\ N(\bar{v})$
$s\ \ ::= \cdot \mid let\,T\,x = e \mid x = e \mid e.f\ = e \mid\ letregion\,\{\,s\,\} \mid open\,x\,\{\,s\,\}$
$\quad\quad \mid open^{a}\ x\,\{\,s\,\} \mid s;s \mid x.set(e) \mid x.transfer() \mid x.\,giveUp()$
$e\ \ ::=\ \ c \mid x \mid e.\,f \mid e.mn(\bar{e}) \mid \text{new}\ N(\bar{e}) \mid (N)\ e \mid x.get()$ //Expressions

The Target Language

$\rho,\pi,p \in \quad region\ names$
$cn \;\in Class\ Names\ (A,B,C\ \ldots)$

$mn \in Method\ Names\ (m, n, \dots)$
$x, f \in Variables, fields$
$n \in Integers$
$Program = (CT, e)$
$c\ \ ::=\ \ n \mid () \mid true \mid false \mid Null$ //Constants
$N\ \ ::=\ \ cn\langle p^a \bar{p}\ \rangle\langle\bar{\tau}\rangle$ //Instantiated class type
$C\ \ ::=\ \ \text{class}\ cn\langle \rho^a \bar{\rho} \mid \phi\rangle\langle\overline{\alpha \lhd N}\rangle \lhd N \left\{\overline{\tau\ f};\ k\ ;\bar{d}\right\}$//Class Definitions
$k\ \ ::=\ \ cn\ (\overline{\tau\ x})\{\ \text{super}\ (\bar{v});\ \overline{\text{this}.f = v;}\}$ //Constructors
$d\ \ ::=\ \ \tau\ mn\langle\rho^a\bar{\rho} \mid \phi\rangle\ (\overline{\tau\ x})\ \{s;\ \text{return}\ e;\}$//Methods
$\phi\ \ ::= true \mid \rho \succcurlyeq \rho \mid \rho = \rho \mid \phi \wedge \phi$ // constraints on region params
$\tau_{\lhd} ::=\ \ \alpha \mid N$ //Types that admit subtyping (subclassing)
$\tau\ \ ::=\ \ \tau_{\lhd} \mid Object\langle p^a\rangle \mid Region[\rho]\langle p^a\rangle\langle\tau\rangle \mid int \mid bool \mid unit\ \mid \exists\rho.\tau$
$v\ \ ::= c \mid x \mid new\ N(\bar{v})$
$s\ \ ::= \cdot \mid let\ \tau\ x = e \mid x = e \mid e.f\ = e \mid\ letregion\langle\rho\rangle\{\ s\ \} \mid open\ e\ \{\ s\ \}$
$\qquad \mid open^a\ e\ \{\ s\ \} \mid s; s \mid e.set(e) \mid e.transfer() \mid e.giveUp() \mid e.suck(e)$
$\qquad \mid\ let\ (\rho, \tau\ x) = unpack\ e$
$e\ \ ::=\ \ c \mid x \mid e.f \mid e.mn\langle p^a\bar{p}\rangle(\bar{e}) \mid \text{new}\ N(\bar{e}) \mid (N)\ e \mid e.get() \mid newRgn\langle\rho\rangle\langle\tau\rangle()$
$\qquad \mid pack[\rho, e]\ as\ \exists\rho.\tau$ //Expressions

Elaboration (Algorithm HM(ρ))

- The function `elaborate` describes an algorithm (HM(ρ)) to elaborate basic class definition to a class definition with region-annotated types (hereafter called as the elaborated definition). The algorithm generates constraints over region variables such that the elaborated definition is well-formed if and only if constraints are satisfiable. HM(ρ) uses a separate constraint solving algorithm (accessible through $normalize$ function) to solve constraints. The nature of constraints and constraint solving is described later in this wiki.
- the top-level `elaborate` function populates the class table (CT') with the elaborated definition of B. It makes use of `elaborate-header`, `elaborate-cons`, and `elaborate-methods` functions which elaborate header (signature and instance variables) of B, the constructor of B, and methods of B respectively. The three functions represent three kinds of occassions on which constraints are solved and solution is applied - after elaborating the header, after elaborating the constructor, and each time a method is elaborated.
- Rules make use of an environment Γ to map variables to their region-annotated types, an environment Δ to map type variables to their bounds, and a set Σ of region variables in scope.
- We define $bound_{\Delta}$ function over types (τ). For a given type, the $bound_{\Delta}$ function identifies the class where we need to look for fields or methods.
  $bound_{\Delta}(\alpha) = \Delta(\alpha)$
  $bound_{\Delta}(N) = N$
  $bound_{\Delta}(T) = T$

```
fun elaborate(B) =
  let
    hdB = elaborate-header(B)
    consB = elaborate-cons(B,hdB)
    fullB = elaborate-methods(B,consB)
  in
    CT'[B ↦ fullB]
  end

fun elaborate-header(B) =
```

let
class $B\langle\overline{\alpha \lhd N_s}\rangle \lhd N_s \{\overline{T\, f}; k_s; \overline{d_s}\}$ = CT(B)
class $B\langle\rho^a\overline{\rho} \mid \top\rangle\langle\overline{\alpha \lhd N}\rangle \lhd N \{\overline{\tau\, f}\}$ = header-template(B)
$\Sigma = \rho^a \cup \overline{\rho}$
$\Delta_\rho = \Sigma$
C1 = type-ok($\Sigma; \Delta_\rho \vdash \overline{N}$)
C2 = type-ok($\Sigma; \Delta_\rho \vdash N$)
C3 = type-ok($\Sigma; \Delta_\rho \vdash \overline{\tau}$)
C = C1 $\wedge$ C2 $\wedge$ C3 $\wedge$ $\overline{\rho} \succcurlyeq \rho^a$
(D, ψ_i) = normalize(C)
$\overline{N_T} = \psi_i(\overline{N})$
$N_T = \psi_i(N)$
$\overline{\tau_T} = \psi_i(\overline{\tau})$
$\rho^a_T = \psi_i(\rho^a)$
$\overline{\rho_T} =$ (frv($\overline{N_T}, N_T, \overline{\tau_T}$)) $- \{\rho^a_T\}$
$\phi = D - \{\overline{\rho_T} \succcurlyeq \rho^a_T\}$ (* We need not record implicit constraints*)
in
class $B\langle\rho^a_T\overline{\rho_T} \mid \phi\rangle\langle\overline{\alpha \lhd N_T}\rangle \lhd N_T \{\overline{\tau_T\, f}\}$
end

fun header-template (B) =
let
class $B\langle\overline{\alpha \lhd N_s}\rangle \lhd N_s \{\overline{T\, f}; k_s; \overline{d_s}\}$ = CT(B)
$\overline{{}^{X}N}$ = templateTy($\overline{N_s}$) (* templateTy is an auxiliary fn defined at the end *)
${}^{X}N$ = templateTy(N_s)
$\overline{{}^{X}\tau}$ = templateTy($\overline{T}$)
ρ^a = allocRgn(${}^{X}N$)
$\overline{\rho}$= (frv($\overline{{}^{X}N}, {}^{X}N, \overline{{}^{X}\tau}$)) $-\{\rho^a\}$
$\psi_i = [B\langle\rho^a\overline{\rho}\rangle\langle\overline{\alpha}\rangle / B\langle\overline{\alpha}\rangle]$ (* templateTy does not templatize recursive occurances of B,
because it doesn't know how many region params are there for B. But, now we know.
We substitute the region annotated type of B for its simple type in the class defn. *)
$\overline{N} = \psi_i(\overline{{}^{X}N})$
$N = \psi_i({}^{X}N)$
$\overline{\tau} = \psi_i(\overline{{}^{X}\tau})$
in
class $B\langle\rho^a\overline{\rho} \mid \top\rangle\langle\overline{\alpha \lhd N}\rangle \lhd N \{\overline{\tau\, f}\}$
end

fun elaborate-cons(B, hdB) =
let
class $B\langle\overline{\alpha \lhd N_s}\rangle \lhd N_s \{\overline{T\, f}; k_s; \overline{d_s}\}$ = CT(B)
class $B\langle\rho^a_B\overline{\rho_B} \mid \phi_B\rangle\langle\overline{\alpha \lhd N_B}\rangle \lhd N_B \{\overline{\tau_B\, f}\}$ = hdB
$\overline{\tau_A}$= ctype(N_B) (* Types of super class constructor args *)
$B(\overline{T_x\, x})\{\text{super}(\overline{\upsilon_g}); \overline{\text{this}.f = \upsilon_f;}\}$ = k_s
$\Sigma = \rho^a_B \cup \overline{\rho_B}$
$\Delta_\rho = \Sigma$ (* All region parameters to the class are assumed to be live. *)
$\overline{\tau_a}$ = templateTy($\overline{T_x}$)
Ca = type-ok($\Sigma; \Delta_\rho \vdash \overline{\tau_a}$)
_ = CT'[B $\mapsto$ class $B\langle\rho^a_B\overline{\rho_B} \mid \phi\rangle\langle\overline{\alpha \lhd N_B}\rangle \lhd N_B \{\overline{\tau_B\, f;}\}$] (* temporarily update CT'
so that "this.f" gives correct type for any field f of B*)
$\Gamma = \underline{\cdot, \text{this}}: B\langle\rho^a_B\overline{\rho_B}\rangle\langle\overline{\alpha}\rangle, x: \overline{\tau_x}$

$\Delta_\alpha = \overline{\alpha \lhd N_B}$
$(\overline{\upsilon'_g : \tau_g}, Cg)$ = elab-expr$(\Sigma;\ \Delta_\alpha;\ \Delta_\rho;\ \Gamma;\ \rho^a_B \vdash \overline{\upsilon_g})$
Csub = subtype-ok$(\Delta_\alpha \vdash \overline{\tau_g} <: \overline{\tau_a})$ (* Actual types of args to super should be subtype of expected types. *)
$(\overline{\text{this.f} = \upsilon'_f}, \_, \_, \_, Cf)$ = elab-stmt$(\Sigma;\ \Delta_\alpha;\ \Delta_\rho;\ \Gamma;\ \rho^a_B \vdash \overline{\text{this.f} = \upsilon_f})$
C = Ca $\wedge$ Cg $\wedge$ Csub $\wedge$ Cf $\wedge$ $(\overline{\rho_B} \succcurlyeq \rho^a_B) \wedge \phi_B$
(D, ψ_i) = normalize(C)
$(\overline{N}, N, \bar{\tau}, \overline{\tau_x}, \rho^a) = (\psi_i(\overline{N_B}), \psi_i(N_B), \psi_i(\overline{\tau_B}), \psi_i(\overline{\tau_a}), \psi_i(\rho^a_B))$
$\bar{\rho}$ = (frv$(\overline{N}, N, \bar{\tau})) - \{\rho^a\}$
ϕ = project-constraints$(D, \{\rho^a, \bar{\rho}\})$ - $(\bar{\rho} \succcurlyeq \rho^a)$ (* Collect residual constraints over region params of class B that need to be recorded explicitly as refinement *)
$(\overline{\upsilon''_g},\ \overline{\upsilon''_f}) = (\psi_i(\overline{\upsilon'_g}),\ \psi_i(\overline{\upsilon'_f}))$
k = $B(\overline{\tau_x\ x})\{\text{super}(\overline{\upsilon''_g});\ \overline{\text{this.f} = \upsilon''_f;}\}$
in
class $B\langle \rho^a \bar{\rho} \mid \phi \rangle \langle \overline{\alpha \lhd N} \rangle \lhd N\ \{\overline{\tau\ f}; k\}$
end

```
fun elaborate-methods (B, consB) =
    elaborate-methods-rec (CT(B), consB)

fun elaborate-methods-rec(Bdef, consB) = case Bdef of
```

class $B\langle \overline{\alpha \lhd N_s} \rangle \lhd N_s \{\overline{T\ f}; k_s;\}$ => consB (* If there are no methods, we are done *)
| class $B\langle \overline{\alpha \lhd N_s} \rangle \lhd N_s \{\overline{T\ f}; k_s; \overline{d_s} d_s\}$ =>
let
fullB' = elaborate-methods-rec (class $B\langle \overline{\alpha \lhd N_s} \rangle \lhd N_s \{\overline{T\ f}; k_s; \overline{d_s}\}$, consB)
class $B\langle \rho^a_B \overline{\rho_B} \mid \phi_B \rangle \langle \overline{\alpha \lhd N_B} \rangle \lhd N_B \{\overline{\tau_B\ f};\ k;\ \overline{d_B}\}$ = fullB'
(* Our task is to elaborate method d_s *)
$T_r\ m(\overline{T_x\ x})\{s; \text{return } e;\}\ =\ d_s$
τ_p = templateTy(T_r)
$\overline{\tau_a}$ = templateTy$(\overline{T_x})$
$(\pi^a, \rho^a_m, \bar{\pi})$ = (new(), new(), frv$(\tau_p, \overline{\tau_a})$) (* π^a denotes allocation context param of "m". ρ^a_m is to be used as a dummy variable to facilitate the unification of allocation contexts for recursive calls of "m" with π^a. In other words, no region polymorphic recursion *)
$d_t = \tau_p\ m\langle \rho^a_m \mid \rho^a_m = \pi^a \rangle (\overline{\tau_a\ x})\{\cdot\}$ (* We use this type of "m" to typecheck recursive applications.
Body of "m" is insignificant; We denote it with a hole. *)
_ = CT'[B $\mapsto$ class $B\langle \rho^a_B \overline{\rho_B} \mid \phi \rangle \langle \overline{\alpha \lhd N_B} \rangle \lhd N_B \{\overline{\tau_B\ f};\ k;\ \overline{d_B} d_t\}$] (* temporarily update CT' so that "*this.m*" gives correct type*)
$\Gamma\ =\ \cdot, \text{this}: B\langle \rho^a_B \overline{\rho_B} \rangle \langle \bar{\alpha} \rangle, x: \overline{\tau_a}$
$\Sigma = \rho^a_B \cup \overline{\rho_B} \cup \pi^a \cup \bar{\pi}$
$\Delta_\alpha = \overline{\alpha \lhd N_B}$
$\Delta_\rho = \Sigma$ (* All region parameters to the class and method are assumed to be live. *)
s0 = redec-rgn-handler$(\overline{T_x\ x})$ (* Re-declare any arguments that are transferable region handlers. Elaboration ensures that region handlers are unpacked *)
$(s', \Gamma', \Delta'_\rho, \Sigma', Cs)$ = elab-stmt$(\Sigma;\ \Delta_\alpha; \Delta_\rho;\ \Gamma; \pi^a \vdash s0; s)$
$(e' : \tau_q, Ce)$ = elab-expr$(\Sigma';\ \Delta_\alpha; \Delta'_\rho;\ \Gamma'; \pi^a \vdash e)$
Csub = subtype-ok$(\Delta_\alpha \vdash \tau_q <: \tau_p)$ (* Actual return type must be subtype of expected return type *)
C = Cs $\wedge$ Ce $\wedge$ Csub $\wedge (\overline{\rho_B} \succcurlyeq \rho^a_B) \wedge \phi_B$ (* Set of all constraints *)
(D, ψ_i) = normalize(C)
$(\overline{N}, N, \bar{\tau}, \overline{\tau_x}, \tau_r, \rho^a) = (\psi_i(\overline{N_B}), \psi_i(N_B), \psi_i(\overline{\tau_B}), \psi_i(\overline{\tau_a}), \psi_i(\tau_p), \psi_i(\rho^a_B))$

$\overline{\rho}$ = (frv($\overline{N}$, N, $\overline{\tau}$)) $- \{\rho^a\}$

Σ_ρ = $\rho^a \cup \overline{\rho}$ (* ρ^a and $\overline{\rho}$ are new region vars that replace ρ_B^a and $\overline{\rho_B}$ as region params of class B *)

(ρ_m^a, ϕ_m^a) = if $\psi_i(\pi^a) \in \Sigma_\rho$

then $(\pi^a, \pi^a = \psi_i(\pi^a))$ (* If allocation ctxt for method is required to be one of the

preexisting regions, then record it explicitly as an equality

constraint over allocation context parameter. *)

else $(\psi_i(\pi^a), \top)$ (* Else, simply do the substitution *)

$\overline{\rho_m}$ = (frv($\overline{\tau_x}$, τ_r)) $- \{\rho_m^a\}$

Σ_π = $\rho_m^a \cup \overline{\rho_m}$

ϕ = project-constraints(D, Σ_ρ) - ($\overline{\rho} \succcurlyeq \rho^a$) (* Explicit constraints over region params of B *)

ϕ_m = project-constraints(D, $\Sigma_\rho \cup \Sigma_\pi$) (* Constraints over region params of method m *)

(s'', e'') = (ψ_i(s'), ψ_i(e'))

d = τ_x m$\langle \rho_m^a \overline{\rho_m} \mid \phi_m^a \wedge \phi_m \rangle (\overline{\tau_x\ x})$ {s'';*return* e''}

in

class B$\langle \rho^a \overline{\rho} \mid \phi \rangle \langle \overline{\alpha \lhd N} \rangle \lhd$ N $\{\overline{\tau\, f};\ k;\ \overline{d}d\}$

end

```
(* AUXILIARY FUNCTIONS *)

fun redec-rgn-handler(T̅x̅) =
  foldr (T̅x̅, nop, fn (T x, s) => case T of
    Region⟨T'⟩ => (let T x = x); s
  | _ => s)

fun templateTy(T) = case T of
  α|int|bool|unit => T
| Object => Object<ρ> where new(ρ)
| A(T̅) => if A ∈ dom(CT') ∧ CT'(A) = class A⟨ρ^a ρ̅ | ϕ⟩⟨α̅ ◁ N̅⟩ ◁ N then
             then A<π^a π̅><τ̅> where new(π^a π̅) ∧ |π̅| = |ρ̅| ∧ τ̅ = templateTy(T̅)
             else T
| Region<T_root> => let τ' = templateTy(T_root) in
                     let τ_root = [ρ/frv(τ')] τ' where new(ρ) in
                         ∃ρ. Region[ρ]<π><τ_root> where new(π)

fun superClasses(B<π^a π̅><τ̅>) = case B of
  Object => {}
| Region[ρ] => Object<π^a>
| _ =>
    let class B⟨ρ^a ρ̅ | ϕ⟩⟨α̅ ◁ N̅⟩ ◁ N = CT'(B) in
    let N' = [π̅/ρ̅][π^a/ρ^a] N in
        {N'} ∪ superClasses(N')
  | superClasses _ => error()

fun allocRgn(B<π^a π̅><τ̅>) = π^a
  | _ => error()

fun project-constraints (D,S) = case D of
  true => D
```

| $\varphi \wedge$ D'=>
  let ϕ = project-constraints (D',S) in
    if frv(φ)$\subseteq$ S then $\varphi \wedge \phi$ else ϕ

fun type-ok ($\Sigma; \Delta_\rho \vdash \tau$) = case τ of
  int | bool | unit => $\top$
| $\mathrm{Region}[\rho_0]\langle\pi_0^a\rangle\langle\tau_0\rangle$ => $(\rho_0 \in \Sigma) \wedge (\pi_0^a \in \Delta_\rho) \wedge (\mathrm{frv}(\tau_0) \subseteq \{\rho_0\})$
| $\mathrm{Object}\langle\pi^a\rangle$ => $(\pi^a \in \Delta_\rho)$
| $B\langle\pi^a\overline{\pi}\rangle\langle\overline{\tau}\rangle$ =>
  let
    Ct = type-ok($\Sigma; \Delta_\rho \vdash \overline{\tau}$)
    Clv = $(\pi^a \in \Delta_\rho) \wedge (\overline{\pi} \subseteq \Delta_\rho)$
    class $B\langle\rho^a\overline{\rho} \mid \phi\rangle\langle\overline{\alpha \lhd N}\rangle \lhd N\{\ldots\}$ = CT'(B)
    C = Ct $\wedge$ Clv $\wedge$ $([\overline{\pi}/\overline{\rho}][\pi^a/\rho^a]\phi)$ $\wedge$ $(\overline{\pi} \succcurlyeq \pi^a)$
  in
    C
  end

fun subtype-ok($\Delta_\alpha \vdash \tau_1 <: \tau_2$) : C = case (τ_1, τ_2) of
  $(\mathrm{Region}[\rho_0]\langle\pi_0^a\rangle\langle\tau_0\rangle, \mathrm{Region}[\rho_1]\langle\pi_1^a\rangle\langle\tau_1\rangle)$ => $(\rho_0 = \rho_1) \wedge (\pi_0^a = \pi_1^a) \wedge (\tau_0 = \tau_1)$
| $(A\langle\pi_0^a\overline{\pi_0}\rangle\langle\overline{\tau_0}\rangle, A\langle\pi_1^a\overline{\pi_1}\rangle\langle\overline{\tau_1}\rangle)$ => $(\pi_0^a = \pi_1^a) \wedge$ $(\overline{\pi_0} = \overline{\pi_1})$ $\wedge (\overline{\tau_0} = \overline{\tau_1})$
| $(A\langle\pi_0^a\overline{\pi_0}\rangle\langle\overline{\tau_0}\rangle, \mathrm{Object}\langle\pi^a\rangle)$ => $(\pi_0^a = \pi^a)$
| $(\alpha, \_)$ => subtype-ok($\Delta_\alpha \vdash \Delta(\alpha) <: \tau_2$)
| $(A\langle\pi_0^a\overline{\pi_0}\rangle\langle\overline{\tau_0}\rangle, \_)$ => if $(A\langle\pi_1^a\overline{\pi_1}\rangle\langle\overline{\tau_1}\rangle \in$ superClasses$(\tau_2))$
    then subtype-ok($\Delta_\alpha \vdash A\langle\pi_0^a\overline{\pi_0}\rangle\langle\overline{\tau_0}\rangle <: A\langle\pi_1^a\overline{\pi_1}\rangle\langle\overline{\tau_1}\rangle$)
    else (assert false)

(* Note: e and e' are exprs in target language. *)
fun elab-pack-expr ($\Delta_\alpha \vdash e : \tau_1 <: \tau_2$):(e',C)=case (τ_1, τ_2) of
  $(\mathrm{Region}[\rho_1]\langle\pi_1^a\rangle\langle\tau_1\rangle, \exists\rho_2.\tau_2')$ =>
  let
    Csub = subtype-ok($\Delta_\alpha \vdash \tau_1 <: [\rho_1/\rho_2]\tau_2'$)
    e' = $\mathrm{pack}[\rho_1, e]$ as $\exists\rho_2.\tau_2'$
  in
    (e',Csub)
  end
| _ => (e, subtype-ok($\Delta_\alpha \vdash \tau_1 <: \tau_2$))

fun elab-expr ($\Sigma; \Delta_\alpha; \Delta_\rho; \Gamma; \rho^a \vdash e$):(e:$\tau$,C) = case e of
  (x.get()) =>
  let
    $(e' : \mathrm{Region}[\rho_0]\langle\pi_0^a\rangle\langle\tau_0\rangle$,Cr) = elab-expr ($\Sigma; \Delta_\alpha; \Delta_\rho; \Gamma; \rho^a \vdash x$)
    Clv = $(\rho_0 \in \Delta_\rho)$
    C = Cr $\wedge$ Clv
  in
    (e'.get(): τ_0, C)
  end
| (new $\mathrm{Region}\langle T\rangle$()) =>
  let
    $\exists\rho.\mathrm{Region}[\rho]\langle\pi^a\rangle\langle\tau\rangle$ as exty = templateTy ($\mathrm{Region}\langle T\rangle$)
    Cwf = type-ok($\Sigma; \Delta_\rho \vdash$exty)
    C = Cwf $\wedge$ $(\rho^a \succcurlyeq \pi^a)$
  in

```
      (newRgn⟨τ⟩() : exty, C)
    end
| (new N_s(\overline{e})) =>
    let
      N = templateTy (N_s)  (* Guaranteeed to return B<..><..> when Ns = B<T>
      Cn = type-ok (Σ; Δ_ρ ⊢ N)
      π^a = allocRgn(N)
      \overline{τ_B} = ctype(N)
      (\overline{e_1} : \overline{τ_e},Ce) = elab-expr (Σ; Δ_α; Δ_ρ; Γ ; ρ^a ⊢ \overline{e})
      (\overline{e_2}: \overline{τ_2}, Csub) = elab-pack-expr (Δ_α ⊢ \overline{e_1}: \overline{τ_e} <: \overline{τ_B})
      C = Cn ∧ Ce ∧ Csub ∧ (ρ^a ≽ π^a)
    in
      (new N(\overline{e_2}) : N, C)
    end
| (e_0. m(\overline{e})) =>
    let
      (e'_0:τ_0, C0) = elab-expr(Σ; Δ_α; Δ_ρ; Γ; ρ^a ⊢ e_0 )
      ⟨ρ^a_m \overline{ρ_m} |ϕ_m⟩\overline{τ_x} → τ = mtype(m, bound_Δ(τ_0))
      \overline{π} = new(length(\overline{ρ_m}))
      Cwf = (\overline{π} ⊆ Δ_ρ) ∧ ψ(ϕ_m)
      ψ = [\overline{π}/\overline{ρ_m}][ρ^a/ρ^a_m]
      (\overline{e_1} : \overline{τ_e},Ce) = elab-expr (Σ; Δ_α; Δ_ρ; Γ ; ρ^a ⊢ \overline{e})
      (\overline{e_2}: \overline{τ_2},Csub) = elab-pack-expr (Δ_α ⊢ \overline{e_1}: \overline{τ_e} <: ψ(\overline{τ_x}))
      C = C0 ∧ Cwf ∧ Ce ∧ Csub
    in
      (e'_0.m⟨ρ^a \overline{π}⟩(\overline{e_2}) : ψ(τ), C)
    end
| (x) => (x:Γ(x),⊤)
| (e.f) =>
    let
      (e':τ_e, Ce) = elab-expr(Σ; Δ_α; Δ_ρ; Γ; ρ^a ⊢e)
      τ_f = ftype(f, bound_Δ(τ_e))
    in
      (e'.f, Ce)
    end

fun elab-stmt(Σ; Δ_α; Δ_ρ; Γ ; ρ^a ⊢s) : (s',Γ',Δ'_ρ,Σ',C) = case s of
   (let Region⟨T⟩ x = e) =>
    let
      ∃ρ_0. Region[ρ_0]⟨π^a_0⟩⟨τ_0⟩ = templateTy(Region⟨T⟩)
      Ct = type-ok(Σ; Δ_ρ ⊢ Region[ρ_0]⟨π^a_0⟩⟨τ_0⟩)
      (e':τ_e,Ce) = elab-expr(Σ; Δ_α; Δ_ρ; Γ ; ρ^a ⊢ e)
      (τ_2, s', Σ', Crhs) = case τ_e of
           ∃ρ_1.τ_1 =>
            let
              s' = let (ρ_0, Region[ρ_0]⟨π^a_0⟩⟨τ_0⟩ x) = unpack e')
                        (* If RHS has existential type, we always unpack *)
              Cunq = (ρ_0 ∉ Σ)
            in
              ([ρ_0/ρ_1] τ_1, s', Σ ∪ ρ_0, Ce ∧ Cunq)
            end
          | _ => (τ_e, let Region[ρ_0]⟨π^a_0⟩⟨τ_0⟩ x = e', Σ, Ce)
      Csub = subtype-ok(Δ_α ⊢ τ_2 <: Region[ρ_0]⟨π^a_0⟩⟨τ_0⟩)
```

```
    C = Ct ∧ Crhs ∧ Csub
    Γ′ = Γ, x: Region[ρ0]⟨π0^a⟩⟨τ0⟩
  in
    (s',Γ′,Δρ, Σ′,C)  (* Note that Δρ is unchanged. Unpacking an existentially
                         bound region handler does not make it live.*)
  end
| (let T x = e) =>
  let
    τ = templateTy(T)
    Ct = type-ok(Σ; Δρ ⊢ τ)
    (e':τe,Ce) = elab-expr(Σ; Δα; Δρ; Γ ;ρ^a ⊢ e)
    Csub = subtype-ok(Δ ⊢ τe <:τ)
    C = Ct ∧ Ce ∧ Csub
    Γ′ = Γ, x: τ
  in
    (let τ x = e′, Γ′,Δρ,Σ,C)
  end
| (e1 = e2) where e1 ∈ {x, e. f} =>
  let
    (e1′ :τ1,C1) = elab-expr(Σ; Δα; Δρ; Γ ;ρ^a ⊢ e1 )
    (e2′ :τ2,C2) = elab-expr(Σ; Δα; Δρ; Γ ;ρ^a ⊢ e2)
    (e2″,Csub) = elab-pack-expr (Δα ⊢ e2′:τ2 <: τ1)
    s' = (e1′ = e2″)
    C = C1 ∧ C2 ∧ Csub
  in
    (s',Γ,Δρ,Σ,C)
  end
| (open x { s }) =>
  let
    (e′: Region[ρ0]⟨π0^a⟩⟨τ0⟩, Cx) = elab-expr(Σ; Δα; Δρ; Γ ;ρ^a ⊢ x)
    Clv = ρ0 ∈ Δρ
    (s', _, _, _,Cs) = elab-stmt(Σ; Δα; Δρ; Γ ;ρ^a ⊢s)
    C = Cx ∧ Cs ∧ Clv
  in
    (open e′ { s′ }, Γ, Δρ, Σ, C)
  end
| (open^a x { s }) =>
  let
    (e′: Region[ρ0]⟨π0^a⟩⟨τ0⟩, Cx) = elab-expr(Σ; Δα; Δρ; Γ ;ρ^a ⊢ x)
    Clv = ρ0 ∈ Δρ
    (s', _, _, _,Cs) = elab-stmt(Σ; Δα; Δρ; Γ ;ρ0 ⊢s)
    C = Cx ∧ Cs ∧ Clv
  in
    (open e′ { s′ }, Γ, Δρ,Σ, C)
  end
| (s1;s2) =>
  let
    (s1',Γ′,Δρ′,Σ′,C1) = elab-stmt(Σ; Δα; Δρ; Γ ;ρ^a ⊢s1)
    (s2',Γ″,Δρ″,Σ″,C2) = elab-stmt(Σ′; Δα; Δρ′; Γ′;ρ^a ⊢s2)
    C = C1 ∧ C2
    s' = s1';s2'
  in
    (s',Γ″,Δρ″,Σ″,C)
```

```
  end
| (x.set(e2)) =>
  let
    (e1':Region[ρ0]⟨π0^a⟩⟨τ0⟩, C1) = elab-expr(Σ; Δα; Δρ; Γ ;ρ^a ⊢x)
    (e2':τ2,C2) = elab-expr(Σ; Δα; Δρ; Γ ;ρ^a ⊢e2)
    Clv = (ρ0 ∈ Δρ)
    Csub = subtype-ok(Δα ⊢ τ2 <: τ0)
    C = C1 ∧ C2 ∧ Clv ∧ Csub
    s' = e1'.set(e2')
  in
    (s',Γ,Δρ,Σ,C)
  end
| (x.transfer()) | (x.giveUp()) =>
  let
    (e':Region[ρ0]⟨π0^a⟩⟨τ0⟩, C1) = elab-expr(Σ; Δα; Δρ; Γ ;ρ^a ⊢x)
    Clv = (ρ0 ∈ Δρ)
    C = C1 ∧ Clv
    s' = e'.transfer() (* or e'.giveUp() *)
    Δ'ρ = Δρ − ρ0 (* Meta-level minus *)
  in
    (s',Γ,Δ'ρ,Σ,C)
  end
```

```
Notes:

(* 1. Uniqueness constraints on transferable regions - only in first
branch of case expression. That is, only when new region name is
introduced. The second branch allows aliasing of region handlers,
while unifying the region names for aliases.
   2. Handling uninitialized variables. Needed for loops.
   3. Subtyping only has to consider two packed types or two unpacked
types. It is equality for former and alpha-equivalence for later.
   4. All set operations are meta-level *)
```

Nature of constraints

Algorithm HM(ρ) generates five kinds of constraints:

1. Region variable equality constraints ($\rho_1 = \rho_2$)
2. Region variable outlives constraints ($\rho_1 \succcurlyeq \rho_2$)
3. Object type equality constraints ($A\langle\pi_1^a\bar{\pi}_1\rangle\langle\bar{\tau}_1\rangle = A\langle\pi_2^a\overline{\pi_2}\rangle\langle\overline{\tau_2}\rangle$).
4. Region variable uniqueness constraints ($\rho \notin \Sigma$)
5. Region variable liveness constraints ($\rho \in \Delta_\rho$)

Constraints of type 3 can be elaborated trivially to equality constraints by point-wise comparision. Similarly, constraints of type 4 can be elaborated to disequality constraints. (Eg: $\rho \notin \{\rho1,\rho2\}$ can be elaborated to $\rho \neq \rho1 \wedge$
$\rho \neq \rho2$). On the other hand, constraints of type 4 can be elaborated to both equality and disequality constraints. For eg, ($\rho \in \{\rho_1, \rho_2\} - \{\rho_3\}$) can be elaborated to ($\underline{\rho = \rho_1 \vee \rho = \rho_2 \wedge \rho \neq \rho_3}$). After elaboration, we are left with conjunctions and disjunctions of 3 kinds of constraints:

1. Region variable equality constraints ($\rho_1 = \rho_2$)
2. Region variable disequality constraints ($\rho_1 \neq \rho_2$)
3. Region variable outlives constraints ($\rho_1 \succcurlyeq \rho_2$)

The normalize function, when successful, outputs a substitution and a set of residual constraints.

Residual constraints are outlives constraints over free region variables. Recall that free region varaibles are generalized as parameters at the beginning of class or method definitions. Along with region parameters, any residual constraints are also recorded as refinements (ϕ) so that they can be checked when region parameters are instantiated with concrete regions. This happens when the class is instantiated, or the method is called.

If normalize function encounters a contradiction (eg: $\rho_1 = \rho_2 \wedge \rho_1 \neq \rho_2$), it fails. This means that the program is not region safe. To demonstrate this point, consider two nested static regions:

```
letregion<R0>{
 letregion<R1>{
   ...
 }
}
```

The type inference algorithm generates constraints that R1 ≠ R0 and R0 ≽ R1 (as R0 outlives R1). Now, assume that code pushes an object allocated in R1 into a list allocated in R0. For this operation to be safe, R1 must outlive R0; so, HM(ρ) generates constraint that R1 ≽ R0. Now, $normalize$ has to solve following constraint:

R1 ≠ R0 ∧ R0 ≽ R1 ∧ R1 ≽ R0

Since R0 and R1 can outlive each other only when R0=R1, the constraint is simplified to:

R1 ≠ R0 ∧ R0 = R1

which is a contradiction! Hence, HM(ρ) fails to infer the type for this program.

The exact definition of constraint solving function ($normalize$) is yet to be formulated.