Region Inference + Exists

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 $\mathrm{Region}\langle \dots \rangle$, we always generate $\exists \rho_0. \mathrm{Region}[\rho_0]\langle \pi_0^a \rangle \langle \dots \rangle$, 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><$\exists \rho$. $\mathrm{Region}[\rho\,]\langle \pi_1^a \rangle \langle \dots \rangle$>. 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<...>(*$\exists \rho\,.\,Region[\rho\,]\langle \dots \rangle \langle \dots \rangle$ *r)* rather than elaborating it to *void foo<...,ρ>(*$Region[\rho\,]\langle \pi_1^a \rangle \langle \dots \rangle$ *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*$\langle \dots \rangle$ *{*$\exists \rho\,.\,Region[\rho\,]\langle \dots \rangle \langle \dots \rangle$ *r; ...}* rather than *Class B*$\langle \dots, \rho \rangle$ *{*$Region[\rho\,]\langle \dots \rangle \langle \dots \rangle$ *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.

until a method-local variable is initialized to refer the 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

$cn \ \in Class\ Names\ (A, B, C \dots)$
$mn \in Method\ Names\ (m, n, \dots)$
$x, f \in Variables, fields$
$n \in Integers$
$Program = (CT, e)$
$c \ ::= \ n \mid () \mid true \mid false \mid Null$ //Constants
$N \ ::= \ cn\langle \bar{T} \rangle$ //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
$v \ ::= c \mid x \mid new\ N(\bar{v})$
$s \ ::= \cdot \mid let\ T\ x = e \mid x = e \mid e.f = e \mid \ letregion\ \{\ s\ \} \mid open\ x\ \{\ s\ \}$
$\quad \mid open^a\ x\ \{\ s\ \} \mid s; s \mid x.set(e) \mid x.transfer() \mid x.giveUp()$
$e \ ::= \ c \mid x \mid e.f \mid e.mn(\bar{e}) \mid \text{new}\ N(\bar{e}) \mid (N)\ e \mid x.get()$ //Expressions

The Target Language

$\rho, \pi, p \in \quad region\ names$
$cn \ \in Class\ Names\ (A, B, C \dots)$
$mn \in Method\ Names\ (m, n, \dots)$

$x, f \in Variables, fields$
$n \in Integers$
$Program = (CT, e)$
c ::= n | () | $true$ | $false$ | $Null$ //Constants
N ::= $cn\langle p^a \bar{p} \rangle\langle \bar{\tau} \rangle$ //Instantiated class type
C ::= class $cn\langle \rho^a \bar{\rho} \mid \phi \rangle \langle \overline{\alpha \lhd N} \rangle \lhd N \{\overline{\tau f}; k; \bar{d}\}$ //Class Definitions
k ::= $cn\ (\overline{\tau x})\{$ super $(\bar{v})$; $\overline{\text{this.} f = v;}\}$ //Constructors
d ::= $\tau\ mn\langle \rho^a \bar{\rho} \mid \phi \rangle\ (\overline{\tau x})\ \{s;$ return $e;\}$ //Methods
ϕ ::= $true \mid \rho \succcurlyeq \rho \mid \rho = \rho \mid \phi \wedge \phi$ // constraints on region params
$\tau_\lhd$ ::= $\alpha \mid N$ //Types that admit subtyping (subclassing)
τ ::= $\tau_\lhd \mid Object\langle p^a \rangle \mid Region[\rho]\langle p^a \rangle\langle \tau \rangle \mid int \mid bool \mid unit \mid \exists \rho. \tau$
v ::= $c \mid x \mid new\ N(\bar{v})$
s ::= $\cdot \mid let\ \tau\ x = e \mid x = e \mid e.f = e \mid letregion\langle \rho \rangle \{ s \} \mid open\ e \{ s \}$
$\mid open^a\ e \{ s \} \mid s; s \mid e.set(e) \mid e.transfer() \mid e.giveUp() \mid e.suck(e)$
$\mid let\ (\rho, \tau\ x) = unpack\ e$
e ::= $c \mid x \mid e.f \mid e.mn\langle p^a \bar{p} \rangle(\bar{e}) \mid$ new $N(\bar{e}) \mid (N)\ e \mid e.get() \mid newRgn\langle \rho \rangle\langle \tau \rangle()$
$\mid pack[\rho, e]\ as\ \exists \rho. \tau$ //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)
C1 = type-ok($\overline{N}$)
C2 = type-ok(N)
C3 = type-ok($\overline{\tau}$)
C = C1 ∧ C2 ∧ C3 ∧ $\overline{\rho} \succcurlyeq \rho^a$
(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)
$\overline{^XN}$ = templateTy($\overline{N_s}$) (* templateTy is an auxiliary fn defined at the end *)
^XN = templateTy(N_s)
$\overline{^X\tau}$ = templateTy($\overline{T}$)
ρ^a = allocRgn(^XN)
$\overline{\rho}$= (frv($\overline{^XN}, {^XN}, \overline{^X\tau}$)) $-\{\rho^a\}$
$\psi_i = [B\langle\rho^a\overline{\rho}\rangle\langle\overline{\alpha}\rangle / B\langle\overline{\alpha}\rangle]$ (* 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. *)
$\overline{N} = \psi_i(\overline{^XN})$
$N = \psi_i(^XN)$
$\overline{\tau} = \psi_i(\overline{^X\tau})$
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 *)
$B(\overline{T_x\ x})\{\mathrm{super}(\overline{\upsilon_g});\ \overline{\mathrm{this}.f = \upsilon_f;}\}$ = k_s
$\overline{\tau_a}$ = templateTy($\overline{T_x}$)
Ca = type-ok($\overline{\tau_a}$)
_ = CT'[B ↦ 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*)
Γ = $\cdot$, this: $B\langle\rho^a_B\overline{\rho_B}\rangle\langle\overline{\alpha}\rangle$, x: $\overline{\tau_x}$
Σ = $\rho^a_B \cup \overline{\rho_B}$
$\Delta = \overline{\alpha \lhd N_B}$
$(\overline{\upsilon'_g : \tau_g}$, Cg) = elab-expr(Σ; Δ; Γ; $\rho^a_B \vdash \overline{\upsilon_g}$)
Csub = subtype-ok($\Delta \vdash \overline{\tau_g} <: \overline{\tau_a}$) (* Actual types of args to super should be subtype of
expected types. *)
$(\overline{\mathrm{this}.f = \upsilon'_f}$, _, Cf) = elab-stmt(Σ; Δ; Γ; $\rho^a_B \vdash \overline{\mathrm{this}.f = \upsilon_f}$)

C = Ca $\wedge$ Cg $\wedge$ Csub $\wedge$ Cf $\wedge$ $(\overline{\rho_B} \succcurlyeq \rho^a_B) \wedge \phi_B$
(D, ψ_i) = normalize(C)
$(\overline{N}, N, \overline{\tau}, \overline{\tau_x}, \rho^a) = (\psi_i(\overline{N_B}), \psi_i(N_B), \psi_i(\overline{\tau_B}), \psi_i(\overline{\tau_a}), \psi_i(\rho^a_B))$
$\overline{\rho}$ = (frv($\overline{N}$, N, $\overline{\tau}$)) $- \{\rho^a\}$
ϕ = project-constraints(D, {ρ^a, $\overline{\rho}$}) - ($\overline{\rho} \succcurlyeq \rho^a$) (* Collect residual constraints over region params of class B that need to be recorded explicitly as refinement *)
($\overline{\upsilon''_g}$, $\overline{\upsilon''_f}$) = ($\psi_i(\overline{\upsilon'_g})$, $\overline{\psi_i(\upsilon'_f)}$)
k = $B(\overline{\tau_x\, x})\{\text{super}(\overline{\upsilon''_g});\ \overline{\text{this.f} = \upsilon''_f;}\}$
in
class $B\langle \rho^a \overline{\rho} \mid \phi\rangle\langle \overline{\alpha \lhd N}\rangle \lhd N\ \{\overline{\tau\ f}; k\}$
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 $B\langle\rho^a_B\overline{\rho_B} \mid \phi\rangle\langle\overline{\alpha \lhd N_B}\rangle \lhd N_B\ \{\overline{\tau_B\, f};\ k;\ \overline{d_B}\}$ = 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($\overline{T_x}$)
(π^a, ρ^a_m, $\overline{\pi}$) = (new(), new(), frv(τ_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*)
Γ = $\cdot$, this: $B\langle\rho^a_B\overline{\rho_B}\rangle\langle\overline{\alpha}\rangle$, x: $\overline{\tau_a}$
$\Sigma = \rho^a_B \cup \overline{\rho_B} \cup \pi^a \cup \overline{\pi}$
$\Delta = \overline{\alpha \lhd N_B}$
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', Γ', Cs) = elab-stmt(Σ; Δ; Γ; $\pi^a \vdash$ s0;s)
(e' : τ_q, Ce) = elab-expr(Σ; Δ; Γ'; $\pi^a \vdash$ e)
Csub = subtype-ok($\Delta \vdash \tau_q <: \tau_p$) (* 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)
$(\overline{N}, N, \overline{\tau}, \overline{\tau_x}, \tau_r, \rho^a) = (\psi_i(\overline{N_B}), \psi_i(N_B), \psi_i(\overline{\tau_B}), \psi_i(\overline{\tau_a}), \psi_i(\tau_p), \psi_i(\rho^a_B))$
$\overline{\rho}$ = (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^a_B and $\overline{\rho_B}$ as region params of class B *)
(ρ^a_m, ϕ^a_m) = if $\psi_i(\pi^a) \in \Sigma_\rho$
then (π^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 (ψ_i(π^a), T)  (* Else, simply do the substitution *)
    ρ_m‾ = (frv(τ_x‾, τ_r)) − {ρ^a_m}
    Σ_π = ρ^a_m ∪ ρ_m‾
    ϕ = project-constraints(D,Σ_ρ) - (ρ‾ ≽ ρ^a) (* Explicit constraints over region params of B *)
    ϕ_m = project-constraints(D,Σ_ρ ∪ Σ_π) (* Constraints over region params of method m *)
    (s'', e'') = (ψ_i(s'), ψ_i(e'))
    d = τ_x m⟨ρ^a_m ρ_m‾ | ϕ^a_m ∧ ϕ_m⟩(τ_x x‾){s''; return e''}
  in
    class B⟨ρ^a ρ‾ | ϕ⟩⟨α ◁ N‾⟩ ◁ N {τ f‾; k; d‾d}
  end

(* AUXILIARY FUNCTIONS *)

fun redec-rgn-handler(T x‾) =
  foldr (T x‾, 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(T‾) => if A ∈ dom(CT') ∧ CT'(A) = class A⟨ρ^a ρ‾ | ϕ⟩⟨α ◁ N‾⟩ ◁ N then
             then A<π^a π‾><τ‾> where new(π^a π‾) ∧ |π‾| = |ρ‾| ∧ τ‾ = templateTy(T‾)
             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 π‾><τ‾>) = case B of
  Object => {}
| Region[ρ] => Object<π^a>
| _ =>
    let class B⟨ρ^a ρ‾ | ϕ⟩⟨α ◁ N‾⟩ ◁ N = CT'(B)in
    let N' = [π‾/ρ‾][π^a/ρ^a] N in
        {N'} ∪ superClasses(N')
  | superClasses _ => error()

fun allocRgn(B<π^a π‾><τ‾>) = π^a
  | _ => error()

fun project-constraints (D,S) = case D of
  true => D
| φ ∧ D'=>
    let ϕ = project-constraints (D',S) in
      if frv(φ) ⊆ S then φ ∧ ϕ else ϕ

fun elab-stmt(Σ; Δ; Γ; ρ^a ⊢s) = case s of
```

```
  (let Region⟨T⟩ x = e) =>
  let
    ∃ρ0. Region[ρ0]⟨π0^a⟩⟨τ0⟩ = templateTy(Region⟨T⟩)
    Ct = type-ok(Region[ρ0]⟨π0^a⟩⟨τ0⟩)
    (e':τe, Ce) = elab-expr(Σ; Δ; Γ ; ρ^a ⊢ e)
    (τ2, s') = case τe of
        ∃ρ1.τ1 => ([ρ0/ρ1] τ1, let (ρ0, Region[ρ0]⟨π0^a⟩⟨τ0⟩ x) = unpack e')
      | _ => (τe, let Region[ρ0]⟨π0^a⟩⟨τ0⟩ x = e')
    Csub = subtype-ok(Δ ⊢ τ2 <: Region[ρ0]⟨π0^a⟩⟨τ0⟩ )
    C = Ct ∧ Ce ∧ Csub
    Γ' = Γ, x: Region[ρ0]⟨π0^a⟩⟨τ0⟩
  in
    (s',Γ',C)
  end
| (let T x = e) =>
  let
    τ = templateTy(T)
    Ct = type-ok(τ)
    (e':τe, Ce) = elab-expr(Σ; Δ; Γ ; ρ^a ⊢ e)
    Csub = subtype-ok(Δ ⊢ τe <: τ)
    C = Ct ∧ Ce ∧ Csub
    Γ' = Γ, x: τ
  in
    (let τ x = e', Γ', C)
  end
| (e1 = e2) where e1 ∈ {x, e.f} =>
  let
    (e1':τ1, C1) = elab-expr(Σ; Δ; Γ ; ρ^a ⊢ e1)
    (e2':τ2, C2) = elab-expr(Σ; Δ; Γ ; ρ^a ⊢ e2)
    (Csub, s') = case (τ1,τ2) of
        (∃ρ1. Region[ρ1]⟨π1^a⟩⟨τ1⟩, Region[ρ2]⟨π2^a⟩⟨τ2⟩) =>
          (subtype-ok(Δ ⊢ τ2 <: [ρ1/ρ0]τ1),
           e1 = pack[ρ2, e2] as ∃ρ1. Region[ρ1]⟨π1^a⟩⟨τ1⟩)
      | _ => (subtype-ok(Δ ⊢ τ2 <: τ1), e1' = e2')
    C = C1 ∧ C2 ∧ Csub
  in
    (s',Γ,C)
  end
| (open x { s }) =>
| (open^a x { s }) =>

fun elab-subtype-ok (Δ ⊢ e : τ1 <: τ2) = case (τ1,τ2) of
  (Region[ρ1]⟨π1^a⟩⟨τ1⟩, ∃ρ2. τ2') =>
  let
    Csub = subtype-ok(Δ ⊢ τ1 <: [ρ1/ρ2]τ2)
    e' = pack[ρ1, e] as ∃ρ2. τ2'
  in
    (e',Csub)
  end
| _ => (e, subtype-ok(Δ ⊢ τ1 <: τ2))

fun elab-expr (Σ; Δ; Γ ; ρ^a ⊢e) = case e of
  (x.get()) =>
```

```
  let
    (e′:Region[ρ_0]⟨π^a_0⟩⟨τ_0⟩,Cr) = elab-expr (Σ; Δ; Γ ; ρ^a ⊢x)
    C = Cr ∧ (ρ_0 ∈ Σ)
  in
    (e′.get():τ_0 ,C)
  end
| (new N(ē)) =>
  let
    N = templateTy (N_s)
    Cn = type-ok (N)
    π^a = allocRgn(N)
    \overline{τ_B} = ctype(N)
    (\overline{e_1} : \overline{τ_e},Ce) = elab-expr (Σ; Δ; Γ ; ρ^a ⊢ ē)
    (\overline{e_2},Csub) = elab-subtype-ok(Δ ⊢ \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(ē)) =>
  let
    (e′_0: τ_0 , C_0) = elab-expr(Σ; Δ; Γ; ρ^a; C_0 ⊢ e_0)
    ⟨ρ^a_m \overline{ρ_m} |ϕ_m⟩\overline{τ_x} → τ = mtype(m, bound_Δ(τ_0))
    π̄ = new(length(\overline{ρ_m}))
    ψ = [π̄/\overline{ρ_m}][ρ^a/ρ^a_m]
    Cx = type-ok(ψ(\overline{τ_x}))
    Cr = type-ok (ψ(τ))
    (\overline{e_1} : \overline{τ_e},Ce) = elab-expr (Σ; Δ; Γ ; ρ^a ⊢ ē)
    (\overline{e′},Csub) = elab-subtype-ok(Δ ⊢ \overline{e_1}: \overline{τ_e} <: ψ(\overline{τ_x}))
    C = Cx ∧ Cr ∧ Ce ∧ Csub ∧ ψ(ϕ_m)
  in
    (e′_0. m(\overline{e′}) : ψ(τ), C)
  end
|
```

```
(* 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.*)
```

Auxiliary judgments

For `type-ok`, `subtype-ok`, `elab-expr` and `elab-stmt`, we retain the judgment notation that we had in previous wiki, for they are already simple and easy to understand. As usual, the judgments are of form $Ctxt; C \vdash Q$ denoting that Q is derivable under context $Ctxt$, given that constraint C is satisfied.

- $C \vdash \tau\ OK$ is equivalent to saying that C = `type-ok(`τ`)`.
- $\Delta; C \vdash \tau_1 <: \tau_2$ is equivalent to saying that C = sub`type-ok(`$\Delta \vdash \tau_1 <: \tau_2$`)`
- $\Sigma; \Delta;\ \Gamma; \rho^a; C \vdash e \hookrightarrow e' : \tau$ is equivalent to saying that $(e' : \tau, C) =$ `elab-expr(`$\Sigma;\ \Delta;\ \Gamma\ ; \rho^a \vdash e$`)`

- $\Sigma; \Delta; \Gamma; \rho^a; C \vdash s \hookrightarrow s' \dashv \Gamma'$ is equivalent to saying that $(s', \Gamma', C) =$ `elab-expr(Σ; Δ; Γ ; ρ^a ⊢s)`.

$\boxed{C \vdash \tau\ OK}$

$$\top \vdash \alpha\ OK, int\ OK, bool\ OK \qquad \top \vdash Object\langle\rho\rangle\ OK \qquad \frac{frv(\tau)=\{\rho\}}{\top\vdash Region[\rho]\langle\rho^a\rangle\langle\tau\rangle\ OK}$$

$$\frac{\begin{array}{c} CT'(B)=class\ B\langle\rho^a\overline{\rho}\mid\phi\rangle\langle\overline{\alpha\lhd N}\rangle\lhd N\ \{\dots\} \\ C_\tau\vdash\overline{\tau}\ OK \quad \Delta=\overline{\alpha\lhd N} \quad new(\pi^a,\overline{\pi}) \quad |\overline{\pi}|=|\overline{\rho}| \quad \psi=[\overline{\pi}/\overline{\rho}][\pi^a/\rho^a] \\ \psi'=\psi\ o\ [\overline{\tau}/\overline{\alpha}] \quad C_\phi=\psi(\phi) \quad \Delta;C_\lhd\vdash\overline{\tau}<:\psi'(\overline{N}) \quad C=C_\tau\wedge C_\lhd\wedge C_\phi\wedge(\overline{\pi}\succcurlyeq\pi^a) \end{array}}{C\vdash B\langle\pi^a\overline{\pi}\rangle\langle\overline{\tau}\rangle\ OK}$$

$\boxed{\Delta;\ C \vdash \tau_1 <: \tau_2}$

$$\Delta; \top \vdash \tau <: \tau \qquad \Delta;\ \pi^a = \rho^a \vdash\ B\langle\pi^a\overline{\pi}\rangle\langle\overline{\tau}\rangle\ <:\ Object\langle\pi^a\rangle$$

$$\frac{\Delta;\ C\vdash\Delta(\alpha)<:\tau_2}{\Delta;\ C\vdash\alpha<:\tau_2} \qquad \frac{A\langle\pi_2^a\overline{\pi_2}\rangle\langle\overline{\tau_2}\rangle \in SuperClasses(B\langle\pi^a\overline{\pi}\rangle\langle\overline{\tau}\rangle)}{\Delta;\big(A\langle\pi_1^a\overline{\pi}_1\rangle\langle\overline{\tau}_1\rangle=A\langle\pi_2^a\overline{\pi_2}\rangle\langle\overline{\tau_2}\rangle\big)\vdash B\langle\pi^a\overline{\pi}\rangle\langle\overline{\tau}\rangle <: A\langle\pi_1^a\overline{\pi}_1\rangle\langle\overline{\tau}_1\rangle}$$

$\boxed{\Sigma; \Delta; \Gamma; \rho^a; C \vdash e \hookrightarrow e' : \tau}$

$$\frac{x{:}\tau \in \Gamma}{\Sigma;\Delta;\Gamma;\ \rho^a;\top\vdash x\hookrightarrow x{:}\tau} \qquad \frac{\begin{array}{c} \Sigma;\ \Delta;\ \Gamma;\ \rho^a;\ C\vdash e\hookrightarrow e'{:}\tau' \\ f{:}\tau\in fields(bound_\Delta(\tau')) \end{array}}{\Sigma;\ \Delta;\ \Gamma;\ \rho^a;\ C\vdash e.f\ \hookrightarrow e'.f{:}\tau}$$

$$\frac{\begin{array}{c} \rho\in\Sigma \\ \Sigma;\ \Delta;\ \Gamma;\ \rho^a;C_R\vdash e\hookrightarrow e'{:}Region[\rho]\langle\pi^a\rangle\langle\tau\rangle \end{array}}{\Sigma;\ \Delta;\ \Gamma;\ \rho^a;C_R\vdash e.get()\hookrightarrow e'.get(){:}\tau}$$

$$\frac{\begin{array}{c} N=templateTy(N_s) \quad C_N\vdash N\ OK \\ \pi^a=allocRgn(N) \quad \overline{\tau_B}=ctype(N) \quad \Sigma;\ \Delta;\ \Gamma;\rho^a;C_e\vdash\bar{e}\hookrightarrow\overline{e_1}{:}\overline{\tau_e} \\ \Delta;\ C_\lhd\vdash\overline{\tau_e}<:\overline{\tau_B} \quad C=C_N\wedge C_e\wedge C_\lhd\wedge(\rho^a\succcurlyeq\pi^a) \end{array}}{\Sigma;\ \Delta;\ \Gamma;\ \rho^a;\ C\vdash new\ N(\bar{e})\hookrightarrow new\ N(\overline{e_1}){:}N} \qquad \frac{eraseRgn\ (\tau)\lhd Object}{\Sigma;\Delta;\Gamma;\ \rho^a;\top\vdash Null{:}\tau}$$

$$\frac{\begin{array}{c} \Sigma;\ \Delta;\ \Gamma;\ \rho^a;\ C_0\vdash e_0\hookrightarrow e_0'{:}\tau_0 \quad mtype\big(m,bound_\Delta(\tau_0)\big) = \langle\rho_m^a\overline{\rho_m}\mid\phi_m\rangle\overline{\tau_x}\to\tau \\ new(\overline{\pi}) \quad |\overline{\pi}|=|\overline{\rho_m}| \quad \psi=[\overline{\pi}/\overline{\rho_m}][\rho^a/\rho_m^a] \quad C_x\vdash\overline{\psi(\tau_x)}\ OK \quad C_r\vdash\psi(\tau)\ OK \\ \Sigma;\ \Delta;\ \Gamma;\rho^a;C_e\vdash\bar{e}\hookrightarrow\overline{e'}{:}\overline{\tau_e} \quad \Delta;\ C_\lhd\vdash\overline{\tau_e}<:\overline{\psi(\tau_x)} \quad C=C_x\wedge C_r\wedge C_e\wedge C_\lhd\wedge\psi(\phi_m) \end{array}}{\Sigma;\ \Delta;\ \Gamma;\ \rho^a;\ C\vdash e_0.m(\bar{e})\hookrightarrow e_0'.m\langle\rho^a\overline{\pi}\rangle\big(\overline{e'}\big){:}\psi(\tau)}$$

$\boxed{\Sigma; \Delta; \Gamma; \rho^a; C \vdash s \hookrightarrow s' \dashv \Gamma'}$

$$\frac{\begin{array}{c} \tau=templateTy(T) \quad C_T\vdash\tau\ OK \quad \Gamma'=\Gamma,x{:}\tau \\ \Sigma;\Delta;\ \Gamma;\rho^a;C\vdash e\hookrightarrow e'{:}\tau_e \quad \Delta;\ C_\lhd\vdash\tau_e<:\tau \quad C=C_T\wedge C_e\wedge C_\lhd \end{array}}{\Sigma;\Delta;\Gamma;\ \rho^a;C\vdash let\ T\ x=e\hookrightarrow let\ \tau\ x=e'\dashv\Gamma'}$$

$$\frac{\begin{array}{c} e_1\in\{x,e.f\} \quad \Sigma;\ \Delta;\ \Gamma;\ \rho^a;\ C_1\vdash e_1\hookrightarrow e_1'{:}\tau_1 \\ \Sigma;\ \Delta;\ \Gamma;\ \rho^a;\ C_2\vdash e_2\hookrightarrow e_2'{:}\tau_2 \quad \Delta;\ C_\lhd\vdash\tau_2<:\tau_1 \\ C=C_1\wedge C_2\wedge C_\lhd \end{array}}{\Sigma;\Delta;\Gamma;\ \rho^a;C\vdash e_1=e_2\hookrightarrow e_1'=e_2'\dashv\Gamma}$$

$$\frac{new(\rho) \quad \Sigma\cup\{\rho\};\Delta;\Gamma;\rho;C_S\vdash s\hookrightarrow s'\dashv\Gamma' \quad C=\rho\notin\Sigma\wedge\Sigma\succcurlyeq\rho\wedge C_S}{\Sigma;\Delta;\Gamma;\rho^a;C\vdash letregion\,\{\,s\,\}\hookrightarrow letregion\langle\rho\rangle\,\{\,s'\,\}\dashv\Gamma}$$

$$\frac{\Sigma;\Delta;\Gamma;\rho^a;C_1\vdash s_1\hookrightarrow s_1'\dashv\Gamma_1 \quad \Sigma;\Delta;\Gamma_1;\rho^a;C_2\vdash s_2\hookrightarrow s_2'\dashv\Gamma' \quad C=C_1\wedge C_2}{\Sigma;\Delta;\Gamma;\rho^a;C\vdash s_1;s_2\hookrightarrow s_1';s_2'\dashv\Gamma'}$$

$$\frac{\Sigma;\Delta;\Gamma;\rho^a;C_R\vdash e\hookrightarrow e':Region[\rho]\langle\pi^a\rangle\langle\tau\rangle \quad \Sigma\cup\{\rho\};\Delta;\Gamma;\rho^a;C_S\vdash s\hookrightarrow s'\dashv\Gamma' \quad C=\rho\notin\Sigma\wedge C_R\wedge C_S}{\Sigma;\Delta;\Gamma;\rho^a;C\vdash open\ e\,\{\,s\,\}\hookrightarrow open\ e'\,\{\,s'\,\}\dashv\Gamma'}$$

$$\frac{\Sigma;\Delta;\Gamma;\rho^a;C_R\vdash e\hookrightarrow e':Region[\rho]\langle\pi^a\rangle\langle\tau\rangle \quad \Sigma\cup\{\rho\};\Delta;\Gamma;\rho;C_S\vdash s\hookrightarrow s'\dashv\Gamma' \quad C=\rho\notin\Sigma\wedge C_R\wedge C_S}{\Sigma;\Delta;\Gamma;\rho^a;C\vdash open^a\ e\,\{\,s\,\}\hookrightarrow open^a\ e'\,\{\,s'\,\}\dashv\Gamma'}$$

$$\frac{\Sigma;\Delta;\Gamma;\rho^a;C_R\vdash e\hookrightarrow e':Region[\rho]\langle\pi^a\rangle\langle\tau\rangle \quad \rho\in\Sigma \quad \Sigma;\Delta;\Gamma;\rho^a;C_2\vdash e_2\hookrightarrow e_2':\tau' \quad C=C_R\wedge C_2\wedge(\tau'=\tau)}{\Sigma;\Delta;\Gamma;\rho^a;C\vdash e.set(e_2)\hookrightarrow e'.set(e_2')\dashv\Gamma}$$

$$\frac{\Sigma;\Delta;\Gamma;\rho^a;C_R\vdash e\hookrightarrow e':Region[\rho]\langle\pi^a\rangle\langle\tau\rangle \quad \rho\notin\Sigma \quad a\in\{transfer,giveUp\}}{\Sigma;\Delta;\Gamma;\rho^a;C\vdash e.a()\hookrightarrow e'.a()\dashv\Gamma}$$