Stacked Borrows: Appendix

Ralf Jung, Hoang-Hai Dang, Jeehoon Kang, Derek Dreyer

1 Recap of the full Stacked Borrows model

The final domains that are relevant for Stacked Borrows are defined in Figure 1. We have changed *Item* into a triple of a *permission* (Unique, SharedRO, SharedRW or Disabled), a tag (which is an optional pointer ID) and a protector (which is an optional call ID) to be able to treat items more uniformly. This also better matches the Rust implementation. We can still use $\mathsf{Unique}(t, c)$ as notation for (Unique, t, c) and similar for the other permissions.

Compared to the presentation in the paper, instead of a memory *Mem* storing both the value and the stacks for each location, we separate the Stacked Borrows state. This simplifies the formal definition by not carrying around state we do not care about. The Stacked Borrows State ς is a record that consists of the stacks for each location (STACKS), the list of active call IDs (CALLS), and two counters used to hand out fresh tags (NEXTPTR) and call IDs (NEXTCALL) when needed. The full state σ additionally contains a map MEM storing the value at each location.

Finally, scalars include not just pointers and integers, but also function pointers $\mathsf{FnPointer}(f)$ as well as ☣ ("poison"), which is used as initial value for freshly allocated memory. We assume a global table that maps function names to the body of the respective functions. Our language operates post closure-conversion, so Rust closures are represented as tuples consisting of the function pointer and the environment.

1.1 Memory accesses

Here we discuss the rules Stacked Borrows enforces on each memory access.

Granting item. One key notion that was implicit in the discussion in the paper is the idea of a *granting item*. The granting item for a particular access (read or write) and a particular tag t is the item that allows this access to happen.

Definition 1. *The* granting item *in a stack S for a read or write access with a tag t is the topmost item in the stack satisfying all of the following conditions:*

- *The item's tag is t.*
- *The item's permission is not* Disabled.
- *If this is a write access, the item's permission is not* SharedRO.

The granting item may not exist. In that case, Stacked Borrows raises an error.

$$
\begin{aligned}
\mathit{PtrId} &\triangleq \mathbb{N} \\
c \in \mathit{CallId} &\triangleq \mathbb{N} \\
f \in \mathit{FnName} &\triangleq \mathit{String} \\
\ell \in \mathit{Loc} &\triangleq \mathbb{N} \\
X^? &\triangleq X \uplus \{\bot\} \\
\iota \in \mathit{Item} &\triangleq \mathit{Permission} \times \mathit{Tag} \times \mathit{CallId}^? \\
t \in \mathit{Tag} &\triangleq \mathit{PtrId}^? \\
p \in \mathit{Permission} &\triangleq \mathsf{Unique} \mid \mathsf{SharedRW} \mid \mathsf{SharedRO} \mid \mathsf{Disabled} \\
s \in \mathit{Scalar} &\triangleq \mathsf{Pointer}(\ell, t) \mid z \mid \not{\mathsf{x}} \mid \mathsf{FnPointer}(f) \qquad \text{where } z \in \mathbb{Z} \\
S \in \mathit{Stack} &\triangleq \mathit{List}(\mathit{Item}) \\
\xi \in \mathit{Stacks} &\triangleq \mathit{Loc} \xrightarrow{\text{fin}} \mathit{Stack} \\
m \in \mathit{Mutability} &\triangleq \mathsf{Mutable} \mid \mathsf{Immutable} \\
\mathit{PtrKind} &\triangleq \mathsf{Ref}(m) \mid \mathsf{Raw}(m) \mid \mathsf{Box} \\
\tau \in \mathit{Type} &\triangleq \mid \mathsf{FixedSize}(n) \mid \mathsf{Ptr}(k, \tau) \qquad \text{where } n \in \mathbb{N},\ k \in \mathit{PtrKind} \\
&\quad \mid \mathsf{UnsafeCell}(\tau) \\
&\quad \mid \mathsf{Union}(\tau^*) \mid \mathsf{Prod}(\tau^*) \mid \mathsf{Sum}(\tau^*) \\
\mathit{RetagKind} &\triangleq \mathsf{Default} \mid \mathsf{Raw} \mid \mathsf{FnEntry} \\
\mathit{AccessType} &\triangleq \mathsf{AccessRead} \mid \mathsf{AccessWrite} \\
\varsigma \in \mathit{SState} &\triangleq \left\{ \begin{array}{r} \textsc{stacks} : \mathit{Stacks}, \\ \textsc{nextptr} : \mathit{PtrId}, \\ \textsc{calls} : \mathit{List}(\mathit{CallId}), \\ \textsc{nextcall} : \mathit{CallId} \end{array} \right\}
\end{aligned}
$$

Figure 1: Stacked Borrows domains

Stacked Borrows rules. With this, we can define the extra effects that a memory access has in Stacked Borrows. (Remember that read and write accesses in general affect multiple locations.)

Rule (WRITE)**.** When a location ℓ is written to as part of a write access with pointer value $\mathsf{Pointer}(\_, t)$ with the current state being ς, the following steps are taken to compute the next state:

1. Find the granting item for a write access with tag t in $\varsigma.\textsc{stacks}(\ell)$ (that is the borrow stack of ℓ).
2. Check if the granting item has permission Unique.
   - If yes, pop the stack until the granting item is at the top.
   - Otherwise, it has permission SharedRW. In that case, pop the stack until all items above the granting item also have permission SharedRW (and stop immediately when that is the case).
3. If any of the popped-off items has a non-$\bot$ call ID c such that $c \in \varsigma.\textsc{calls}$, Stacked Borrows raises an error.

This matches WRITE-1: we need to find a Unique or SharedRW with the right tag, and then we pop some of the items above the granting item, respecting protectors.

Rule (READ)**.** When a location ℓ is read from as part of a read access with pointer value $\mathsf{Pointer}(\_, t)$ with the current state being ς, the following steps are taken to compute the next state:

1. Find the granting item for a read access with tag t in $\varsigma.\textsc{stacks}(\ell)$ (that is the borrow stack of ℓ).
2. For all the items above the granting item that have permission Unique, change their permission to Disabled.
3. If any of these items has a non-$\bot$ call ID c such that $c \in \varsigma.\textsc{calls}$, Stacked Borrows raises an error.

This matches READ-2 with the "final tweak": instead of popping until there are no Unique above the granting item, we merely disable those items (respecting protectors).

The rules for read and write access to individual locations are formally defined in Figure 2 and Figure 3. In Figure 4, we lift them to blocks of memory, and we also define what happens for deallocation.

A few words on the style of these definitions: Stacked Borrows is mostly about computing, given the current state and details about the memory access that happens, whether that access is okay and what the next state is. Hence the semantics is defined in functional (not relational) style. Most of these functions are fallible, where failure represents "this operation is invalid".

We use $\mathbf{bind}\, x = c\, \mathbf{in}\, e$ as notation for the monadic bind: if c is $\bot$, then the entire expression is $\bot$; otherwise e gets evaluated with x bound to the result of c. Monadic return is an implicit coercion; we think of X being implicitly injected into $X^?$. In case we want to handle failure of a subcomputation instead of propagating it, we write $\mathbf{if\, bind}\, x = c\, \mathbf{then}\, e_1\, \mathbf{else}\, e_2$, where e_1 is the success case (x is bound there) and e_2 the expression used in case c failed.

Moreover, we often need to update a single field of the *SState* record or a finite map; for that we use $x\, \mathbf{with}\, [f := e]$ where f says which field/element of x is being updated; the other fields/elements remain unchanged. If e is a partial function, **with** propagates failure: the entire expression is $\bot$ if e is $\bot$. When updating finite maps, we update many elements at once by writing $x\, \mathbf{with}_{\, i \in I} [i := e]$.

(* Defines whether p can be used to justify accesses of type a. *)

$\mathsf{Grants}(p : \mathit{Permission}, a : \mathit{AccessType}) : \mathbb{B}$

$$
\begin{array}{lcl}
\mathsf{Grants}(\mathsf{Disabled}, \_) & \triangleq & \textbf{false} \\
\mathsf{Grants}(\mathsf{SharedRO}, \mathsf{AccessWrite}) & \triangleq & \textbf{false} \\
\mathsf{Grants}(\_, \_) & \triangleq & \textbf{true}
\end{array}
$$

(* Finds the topmost item in S that grants access a to t. Returns the index in the stack (0 = bottom) and the permission of the granting item. *)

$\mathsf{FindGranting}(S : \mathit{Stack}, a : \mathit{AccessType}, t : \mathit{Tag}^?) : (\mathbb{N} \times \mathit{Permission})^?$

$$
\begin{array}{lcl}
\mathsf{FindGranting}([], a, t) & \triangleq & \bot \\
\mathsf{FindGranting}(S +\!\!+ [\iota], a, t) & \triangleq & \textbf{if}\,(\iota.\textsc{tag} = t \wedge \mathsf{Grants}(\iota.\textsc{perm}, a)) \\
 & & \textbf{then}\,(|S|, \iota.\textsc{perm})\,\textbf{else}\,\mathsf{FindGranting}(S, a, t)
\end{array}
$$

(* Finds the bottommost item above i that is incompatible with a write access justified by p. *)

$\mathsf{FindFirstWIncompat}(S : \mathit{Stack}, i : \mathbb{N}, p : \mathit{Permission}) : \mathbb{N}^?$

$$
\begin{array}{lcl}
\mathsf{FindFirstWIncompat}(\_, \_, \mathsf{Disabled}) & \triangleq & \bot \\
\mathsf{FindFirstWIncompat}(\_, \_, \mathsf{SharedRO}) & \triangleq & \bot \\
\quad (* \text{ Writes to } \mathsf{Unique} \text{ are incompatible with everything above. } *) & & \\
\mathsf{FindFirstWIncompat}(S, i, \mathsf{Unique}) & \triangleq & i + 1 \\
\quad (* \text{ Writes to } \mathsf{SharedRW} \text{ are } \textit{compatible} \text{ with adjacent } \mathsf{SharedRW}\text{, and nothing else. } *) & & \\
\mathsf{FindFirstWIncompat}(S, i, \mathsf{SharedRW}) & \triangleq & \textbf{if}\,(i \geq |S| \vee S(i).\textsc{perm} \neq \mathsf{SharedRW}) \\
 & & \textbf{then}\,i\,\textbf{else}\,\mathsf{FindFirstWIncompat}(S, i + 1, \mathsf{SharedRW})
\end{array}
$$

Figure 2: Stacked Borrows per-location access semantics: helper functions.

(* Computes the new stack after an access of type a with tag t. Also depends on the active calls tracked by C. *)

$\mathsf{Access}(a : \mathit{AccessType}, S : \mathit{Stack}, t : \mathit{Tag}^?, C : \mathit{List}(\mathit{CallId})) : \mathit{Stack}^?$

$$
\begin{aligned}
&\mathsf{Access}(\mathsf{AccessRead}, S, t, C) \triangleq\ && \textbf{bind}\ (i, \_) = \mathsf{FindGranting}(S, \mathsf{AccessRead}, t)\ \textbf{in}\\
&&& \texttt{(* Disable all Unique above } i\texttt{; error out if any of them is protected. *)}\\
&&& \textbf{if}\ (\{S(j).\textsc{protector} \mid j \in [i, |S|) \wedge S(j).\textsc{perm} = \mathsf{Unique}\} \cap C = \emptyset)\\
&&& \textbf{then}\ S\ \textbf{with}\ _{j \in [i,|S|) \wedge S(j).\textsc{perm} = \mathsf{Unique}}[j := (S(j)\ \textbf{with}\ [\textsc{perm} := \mathsf{Disabled}])]\\
&&& \textbf{else}\ \bot\\
&\mathsf{Access}(\mathsf{AccessWrite}, S, t, C) \triangleq\ && \textbf{bind}\ (i, p) = \mathsf{FindGranting}(S, \mathsf{AccessWrite}, t)\ \textbf{in}\\
&&& \textbf{bind}\ j = \mathsf{FindFirstWIncompat}(S, i, p)\ \textbf{in}\\
&&& \texttt{(* Remove items at } j \texttt{ and above; error out if any of them is protected. *)}\\
&&& \textbf{if}\ (\{S(i').\textsc{protector} \mid i' \in [j, |S|)\} \cap C = \emptyset)\\
&&& \textbf{then}\ S|_{[0,j)}\\
&&& \textbf{else}\ \bot
\end{aligned}
$$

Figure 3: Stacked Borrows per-location access semantics.

(* Read and write accesses are just lifted pointwise for each location. *)

$\mathsf{MemAccessed}(\xi : \mathit{Stacks}, C : \mathit{List}(\mathit{CallId}), a, \mathsf{Pointer}(\ell, t), n : \mathbb{N}) : \mathit{Stacks}^? \triangleq$

$\quad \xi\ \textbf{with}\ _{\ell' \in [\ell, \ell+n)}[\ell' := \mathsf{Access}(a, \xi(\ell'), t, C)]$

(* Deallocation is like a write, but also errors out if any item is still protected. *)

$\mathsf{Dealloc}(S : \mathit{Stack}, t : \mathit{Tag}^?, C : \mathit{List}(\mathit{CallId})) : 1^? \triangleq$

$\quad \textbf{bind}\ \_ = \mathsf{FindGranting}(S, \mathsf{AccessWrite}, t)\ \textbf{in}$

$\quad \textbf{if}\ (\{\iota.\textsc{protector} \mid \iota \in S\} \cap C = \emptyset)\ \textbf{then}\ ()\ \textbf{else}\ \bot$

$\mathsf{MemDeallocated}(\xi : \mathit{Stacks}, C : \mathit{List}(\mathit{CallId}), \mathsf{Pointer}(\ell, t), n : \mathbb{N}) : \mathit{Stacks}^? \triangleq$

$\quad \xi\ \textbf{with}\ _{\ell' \in [\ell, \ell+n)}[\ell' := (\textbf{bind}\ \_ = \mathsf{Dealloc}(\xi(\ell'), t, C)\ \textbf{in}\ \bot)]$

Figure 4: Stacked Borrows access semantics for location blocks (and deallocation).

1.2 Retagging

In the discussion in the paper, there were also extra steps taken any time a reference is created (NEW-MUTABLE-REF and NEW-SHARED-REF-2) or cast to a raw pointer (NEW-MUTABLE-RAW and NEW-CONST-RAW-2). It turns out that we can reduce those to instances of `retag`: an assignment like `let x = &mut expr` in the Rust source program becomes `let x = &mut expr; retag x` in our language with explicit retagging. Similar for creating shared references, and for casts to raw pointers: we always insert a `retag` on the result of those operations, and that will take care of all the side-effects Stacked Borrows requires.

So, with that, `retag` becomes the last piece of Stacked Borrows that we have to define:

Rule (RETAG)**.** When executing a `retag x` or `retag[fn] x` on some local variable `x` with value $\mathsf{Pointer}(\ell_x, t_x)$ and type `&mut T`, `&T`, `*mut T` or `*const T`, with the current state being ς, the following steps are taken to compute the next state:

1. Pick a new tag t': if `x` is a (mutable or shared) reference, set $t' \triangleq \varsigma.\text{NEXTPTR}$ and increase NEXTPTR by 1. Otherwise, set $t' \triangleq \bot$.
2. Do the following for each location ℓ in the range $[\ell_x, \ell_x + \mathsf{sizeof}(\texttt{T}))$:
   (We refer to these steps as *reborrowing*.)
   (a) Compute the permission p we are going to grant to t' for ℓ:
      - If `x` is a mutable reference, $p \triangleq \mathsf{Unique}$.
      - If `x` is a mutable raw pointer, $p \triangleq \mathsf{SharedRW}$.
      - If `x` is a shared reference or a constant raw pointer, then if ℓ is inside an `UnsafeCell` $p \triangleq \mathsf{SharedRW}$, else (outside `UnsafeCell`) $p \triangleq \mathsf{SharedRO}$.

   (b) The new item we want to add is (p, t', c) where $c \triangleq \mathsf{head}(\varsigma.\text{CALLS})$ if this is a `retag[fn]` and $c \triangleq \bot$ otherwise.

   (c) The "access" that this operation corresponds to is a read access if $p = \mathsf{SharedRO}$, and a write access otherwise.

   (d) Find the granting item for this access with tag t_x in $\varsigma.\text{STACKS}(\ell)$.

   (e) Check if $p = \mathsf{SharedRW}$.
      - If yes, add the new item just above the granting item.
      - Otherwise, perform the effects of a read/write access (as determined in (c)) with t_x (that is, with the *old* tag) to ℓ (see READ and WRITE). Then, push the new item to the top of the stack.

This matches exactly what we did for NEW-MUTABLE-REF, NEW-SHARED-REF-2, NEW-MUTABLE-RAW and NEW-CONST-RAW-2. You can find the formal definitions of reborrowing (the per-location action (2a) – (2e)) and retagging in Figure 5 and Figure 6.

(* Inserts ι into S at index i. *)

$\mathsf{InsertAt}(S : \mathit{Stack}, \iota : \mathit{Item}, i : \mathbb{N}) \triangleq S|_{[0,i)} +\!\!+ [\iota] +\!\!+ S|_{[i,|S|)}$

(* Computes the new stack after inserting new item ι_{new} derived from old tag t_{old}. Also depends on the list of active calls C (used by Access). *)

$\mathsf{GrantTag}(S : \mathit{Stack}, t_{old} : \mathit{Tag}^?, \iota_{new} : \mathit{Item}, C : \mathit{List}(\mathit{CallId})) : \mathit{Stack}^? \triangleq$

(* Determine the "access" this operation corresponds to. Step (2c). *)

let $a = (\textbf{if } \mathsf{Grants}(\iota_{new}.\text{PERM}, \mathsf{AccessWrite}) \textbf{ then } \mathsf{AccessWrite} \textbf{ else } \mathsf{AccessRead})$ **in**

(* Find the item matching the old tag. Step (2d). *)

bind $(i, p) = \mathsf{FindGranting}(S, a, t_{old})$ **in**

if $\iota_{new}.\text{PERM} = \mathsf{SharedRW}$ **then**

(* A SharedRW just gets inserted next to the granting item. Step (2e). *)

bind $j = \mathsf{FindFirstWIncompat}(S, i, p)$ **in** $\mathsf{InsertAt}(S, \iota_{new}, j)$

else

(* Otherwise, perform the effects of an access and add item at the top. Step (2e). *)

bind $S' = \mathsf{Access}(a, S, t_{old}, C)$ **in** $\mathsf{InsertAt}(S', \iota_{new}, |S'|)$

(* Lists all locations covered by a value of type τ stored at location ℓ, and indicate for each location whether it is frozen (outside an `UnsafeCell`) or not. *)

$\mathsf{FrozenIter}(\ell : \mathit{Loc}, \tau : \mathit{Type}) : \mathit{List}(\mathit{Loc} \times \mathbb{B}) \triangleq$

$\big[(\ell', b) \mid \ell' \in [\ell, \ell + |\tau|) \wedge b = (\ell' \text{ is } \textit{outside} \text{ of an } \texttt{UnsafeCell})\big]$

(* Computes the new permission granted to a reborrow with pointer kind k; fr indicates if this location is frozen or not. Step (2a). *)

$\mathsf{NewPerm}(k : \mathit{PtrKind}, fr : \mathbb{B}) : \mathit{Permission}$

(* Mutable references and boxes get Unique permission. *)

$\mathsf{NewPerm}(\mathsf{Ref}(\mathsf{Mutable}), \_) \triangleq \mathsf{Unique}$

$\mathsf{NewPerm}(\mathsf{Box}, \_) \triangleq \mathsf{Unique}$

(* Mutable raw pointers get SharedRW permission. *)

$\mathsf{NewPerm}(\mathsf{Raw}(\mathsf{Mutable}), \_) \triangleq \mathsf{SharedRW}$

(* Shared references and const raw pointers permission depends on whether the location is frozen or not. *)

$\mathsf{NewPerm}(\mathsf{Ref}(\mathsf{Immutable}), fr) \triangleq \textbf{if } fr \textbf{ then } \mathsf{SharedRO} \textbf{ else } \mathsf{SharedRW}$

$\mathsf{NewPerm}(\mathsf{Raw}(\mathsf{Immutable}), fr) \triangleq \textbf{if } fr \textbf{ then } \mathsf{SharedRO} \textbf{ else } \mathsf{SharedRW}$

Figure 5: Stacked Borrows retagging semantics: helper functions.

(* Reborrow the memory pointed to by $\mathsf{Pointer}(\ell, t_{old})$ for pointer kind k and pointee type τ.
prot indicates if the item should be protected. Step (2). *)

$\mathsf{Reborrow}(\xi : \mathit{Stacks}, C : \mathit{List}(\mathit{CallId}), \mathsf{Pointer}(\ell, t_{old}), \tau : \mathit{Type}, k : \mathit{PtrKind}, t_{new} : \mathit{Tag}^?, \mathit{prot} : \mathit{CallId}^?)$
$: \mathit{Stacks}^? \triangleq$

(* For each location, determine the new permission and add a corresponding item. *)

ξ **with** $_{(\ell', fr) \in \mathsf{FrozenIter}(\ell, \tau)}[\ell' :=$
  let $p_{new} = \mathsf{NewPerm}(k, fr)$ **in**
  let $\iota_{new} = (p_{new}, t_{new}, pro)$ **in** (* Step (2b). *)
  $\mathsf{GrantTag}(\xi(\ell'), t_{old}, \iota_{new}, C)$
]

(* For a given pointer kind and retag kind, determine the tag and protector used for reborrowing. Also returns the new "next tag". *)

$\mathsf{NewTagAndProtector}(n : \mathbb{N}, k : \mathit{PtrKind}, k' : \mathit{RetagKind}, C : \mathit{List}(\mathit{CallId})) : (\mathit{Tag}^? \times \mathit{CallId}^? \times \mathbb{N})^?$

(* References get a fresh tag and sometimes a protector. *)
$\mathsf{NewTagAndProtector}(n, \mathsf{Ref}(\_), \_, C) \triangleq (n, \textbf{if } k' = \mathsf{FnEntry} \textbf{ then } \mathsf{top}(C) \textbf{ else } \bot, n+1)$
(* Boxes get a fresh tag and never a protector. *)
$\mathsf{NewTagAndProtector}(n, \mathsf{Box}, \_, \_) \triangleq (n, \bot, n+1)$
(* Raw retags are used for reference-to-raw casts: the pointer gets untagged. *)
$\mathsf{NewTagAndProtector}(n, \mathsf{Raw}(\_), \mathsf{Raw}, \_) \triangleq (\bot, \bot, n)$
(* Nothing to do otherwise. *)
$\mathsf{NewTagAndProtector}(n, \_, \_, \_) \triangleq \bot$

(* Top-level retag operation. Computes the new tag and new state. *)

$\mathsf{Retag}(\varsigma : \mathit{SState}, \mathsf{Pointer}(\ell, t_{old}), \tau : \mathit{Type}, k : \mathit{PtrKind}, k' : \mathit{RetagKind}) : (\mathit{Tag} \times \mathit{SState})^? \triangleq$

(* If we can compute the next tag and protector, then try reborrowing. *)
if bind $(t_{new}, pro', n') = \mathsf{NewTagAndProtector}(\varsigma.\textsc{nextptr}, k, k', \varsigma.\textsc{calls})$ **then**
  bind $\xi' = \mathsf{Reborrow}(\varsigma.\textsc{stacks}, \varsigma.\textsc{calls}, \mathsf{Pointer}(\ell, t_{old}), \tau, k, t_{new}, pro')$ **in**
  $(t_{new}, \varsigma$ **with** $[\textsc{stacks} := \xi', \textsc{nextptr} := n'])$
else
  (* Otherwise, do nothing. *)
  (t_{old}, ς)

Figure 6: Stacked Borrows retagging semantics.

1.3 Relational semantics

Finally, we wrap the operations we have just defined in a relational semantics that defines how *memory events* (see Figure 1) affect the state ς maintained by Stacked Borrows: see Figure 7. This makes it easy to add Stacked Borrows to an existing memory model: just add the extra state it needs, and perform steps with the appropriate memory events on each allocation, deallocation, read/write access, begin/end of a function call and retag operation.

$$\varepsilon \in \mathit{Event} \triangleq \begin{array}{l} \mid \mathsf{EAccess}(a, \mathsf{Pointer}(\ell, t), \tau) \\ \mid \mathsf{ERetag}(\mathsf{Pointer}(\ell, t_{old}), t_{new}, \tau, k, k') \qquad \text{where } k \in \mathit{PtrKind},\ k' \in \mathit{RetagKind} \\ \mid \mathsf{EAlloc}(\mathsf{Pointer}(\ell, t), \tau) \mid \mathsf{EDealloc}(\mathsf{Pointer}(\ell, t), \tau) \\ \mid \mathsf{EInitCall}(c) \mid \mathsf{EEndCall}(c) \qquad \text{where } c \in \mathit{CallId} \end{array}$$

$$\boxed{\varsigma \xrightarrow{\varepsilon} \varsigma'}$$

$$\text{OS-ALLOC}\ \frac{\begin{array}{c}\forall \ell' \in [\ell, \ell + |\tau|).\ \ell' \notin \mathrm{dom}(\varsigma.\textsc{stacks}) \qquad t = \varsigma.\textsc{nextptr} \\ \xi' = \varsigma.\textsc{stacks}\ \mathbf{with}\ _{\ell' \in [\ell, \ell+|\tau|)}[\ell' := [(\mathsf{Unique}, t, \bot)]] \\ \varsigma' = \varsigma\ \mathbf{with}\ [\textsc{stacks} := \xi', \textsc{nextptr} := \varsigma.\textsc{nextptr} + 1]\end{array}}{\varsigma \xrightarrow{\mathsf{EAlloc}(\mathsf{Pointer}(\ell,t),\tau)} \varsigma'}$$

$$\text{OS-DEALLOC}\ \frac{\mathsf{MemDeallocated}(\varsigma.\textsc{stacks}, \varsigma.\textsc{calls}, \mathsf{Pointer}(\ell, t), |\tau|) = \xi' \qquad \varsigma' = \varsigma\ \mathbf{with}\ [\textsc{stacks} := \xi']}{\varsigma \xrightarrow{\mathsf{EDealloc}(\mathsf{Pointer}(\ell,t),\tau)} \varsigma'}$$

$$\text{OS-ACCESS}\ \frac{\mathsf{MemAccessed}(\varsigma.\textsc{stacks}, \varsigma.\textsc{calls}, a, \mathsf{Pointer}(\ell, t), |\tau|) = \xi' \qquad \varsigma' = \varsigma\ \mathbf{with}\ [\textsc{stacks} := \xi']}{\varsigma \xrightarrow{\mathsf{EAccess}(a,\mathsf{Pointer}(\ell,t),\tau)} \varsigma'}$$

$$\text{OS-RETAG}\ \frac{\begin{array}{c}\mathsf{Retag}(\varsigma.\textsc{stacks}, \varsigma.\textsc{nextptr}, \varsigma.\textsc{calls}, \mathsf{Pointer}(\ell, t_{old}), \tau, k, k') = (t_{new}, \xi', n') \\ \varsigma' = \varsigma\ \mathbf{with}\ [\textsc{stacks} := \xi', \textsc{nextptr} := n']\end{array}}{\varsigma \xrightarrow{\mathsf{ERetag}(\mathsf{Pointer}(\ell,t_{old}),t_{new},\tau,k,k')} \varsigma'}$$

$$\text{OS-INITCALL}\ \frac{c = \varsigma.\textsc{nextcall} \qquad \varsigma' = \varsigma\ \mathbf{with}\ [\textsc{calls} := \varsigma.\textsc{calls} +\!\!+ [c], \textsc{nextcall} := c + 1]}{\varsigma \xrightarrow{\mathsf{EInitCall}(c)} \varsigma'}$$

$$\text{OS-ENDCALL}\ \frac{\varsigma.\textsc{calls} = C +\!\!+ [c] \qquad \varsigma' = \varsigma\ \mathbf{with}\ [\textsc{calls} := C]}{\varsigma \xrightarrow{\mathsf{EEndCall}(c)} \varsigma'}$$

Figure 7: Relational Stacked Borrows semantics with events.

2 Simulation Setup

Here we give a rough idea of the simulation relation that we use in the Coq formalization.

Machine states. The machine state includes the heap and the instrumented state managed by Stacked Borrows.

$$\sigma \in \mathit{State} \triangleq \left\{ \begin{array}{c} \textsc{mem} : \mathit{Loc} \xrightarrow{\text{fin}} \mathit{Scalar}, \\ \_ : \mathit{SState} \end{array} \right\}$$

We directly use the projections of *SState* (STACKS and CALLS and so on) also on *State* to avoid having to test projections.

Well-formed states. A state σ is *well-formed* if (1) MEM and STACKS have the same domain, (2) NEXTPTR is bigger than all the tags in all the pointers stored in MEM, (3) NEXTPTR is bigger than all the tags in all the stacks and NEXTCALL is bigger than all the call IDs in all the stacks, (4) no tag appears twice in the same stack, (5) no call ID appears twice in CALLS, and (6) NEXTCALL is bigger than all call IDs in CALLS.

$$\begin{aligned}
\mathsf{StateWf}(\sigma) \triangleq\ & \mathrm{dom}(\sigma.\textsc{mem}) = \mathrm{dom}(\sigma.\textsc{stacks}) \wedge \\
& \big(\forall(\ell \mapsto \mathsf{Pointer}(\ell', t)) \in \sigma.\textsc{mem}.\ t \neq \bot \Rightarrow t < \sigma.\textsc{nextptr}\big) \wedge \\
& \big(\forall(\ell \mapsto S) \in \sigma.\textsc{stacks}.\ S \neq () \wedge \forall(_, t, c) \in S. \\
& \quad (t \neq \bot \Rightarrow t < \sigma.\textsc{nextptr}) \wedge (c \neq \bot \Rightarrow c < \sigma.\textsc{nextcall})\big) \wedge \\
& \big(\forall(\ell \mapsto S) \in \sigma.\textsc{stacks}, t.\ |\{i \mid S.i = (_, t, _)\}| \leq 1\big) \wedge \\
& \big(\forall c.\ |\{i \mid \sigma.\textsc{calls}.i = c\}| \leq 1\big) \wedge \\
& \forall c \in \sigma.\ \textsc{calls}.c < \sigma.\textsc{nextcall}
\end{aligned}$$

Resources and their interpretation. Resources $r \in \mathit{Res}$ are elements of the following record:

$$r \ni \mathit{Res} \triangleq \left\{ \begin{array}{l} \textsc{tmap} : \mathit{PtrId} \xrightarrow{\text{fin}} \mathit{TagKind} \times (\mathit{Loc} \xrightarrow{\text{fin}} \mathit{Scalar} \times \mathit{Scalar}), \\ \textsc{cmap} : \mathit{CallId} \xrightarrow{\text{fin}} (\mathit{PtrId} \xrightarrow{\text{fin}} \wp(\mathit{Loc})) \end{array} \right\}$$

$$\mathit{TagKind} \triangleq \{\mathsf{Local}, \mathsf{Unique}, \mathsf{Pub}\}$$

Res is an RA, by mapping into the following:

$$\left\{ \begin{array}{l} \textsc{tmap} : \mathit{PtrId} \xrightarrow{\text{fin}} (\mathit{Ex}() + \mathit{Ex}() + ()) \times (\mathit{Loc} \xrightarrow{\text{fin}} \mathit{Ag}(\mathit{Scalar} \times \mathit{Scalar})), \\ \textsc{cmap} : \mathit{CallId} \xrightarrow{\text{fin}} \mathit{Ex}(\mathit{PtrId} \xrightarrow{\text{fin}} \wp(\mathit{Loc})) \end{array} \right\}$$

We pointwise use the RA structure on finite partial function. *Ag* and *Ex* are the agreement RA from Iris, and + is the sum RA. *TagKind* is an RA where Local and Unique are exclusive and Pub duplicable (*i.e.*, it is isomorphic to $\mathit{Ex}() + \mathit{Ex}() + ()$).

The state relation $\mathcal{S}$ and the scalar relation $\mathcal{A}$ are defined as follows:

Pointers are related if they are equal and the tag is public:

$\mathcal{A}(r) \triangleq \{(\mathsf{Pointer}(\ell, t), \mathsf{Pointer}(\ell, t)) \mid t \neq \bot \Rightarrow r.\textsc{tmap}(t).0 = \mathsf{Pub}\} \cup \ldots$

Two states are related if they are sufficiently equal and all the resource-related invariants hold:

$$\mathcal{S}(r) \triangleq \left\{ (\sigma_s, \sigma_t) \;\middle|\; \begin{array}{l} \mathsf{StateWf}(\sigma_s) \land \mathsf{StateWf}(\sigma_t) \land \mathcal{V}(r) \land \mathsf{SrcTgtRel}(r, \sigma_s, \sigma_t) \land \\ \mathsf{TagMapInv}(r, \sigma_s, \sigma_t) \land \mathsf{CallMapInv}(r, \sigma_t) \end{array} \right\}$$

The *active* SharedRO of a stack are those at the top:

$$\mathsf{ActiveSharedRO} : \mathit{Stack} \to \wp(\mathit{PtrId})$$

$$\mathsf{ActiveSharedRO}(S) \triangleq \begin{cases} \emptyset & \text{where } \mathsf{head}(S) \neq (\mathsf{SharedRO}, \_, \_) \\ \{t\} \cup \mathsf{HeadSharedRO}(\mathsf{tail}(S)) & \text{where } \mathsf{head}(S) = (\mathsf{SharedRO}, t, \_) \end{cases}$$

The heaplets associated with the tags agree with the state as long as the tags are in the stack:

$$\mathsf{TagMapInvPre}(k, t, \ell, \sigma_t) \triangleq \begin{cases} \mathsf{True} & \text{where } k = \mathsf{Local} \\ \big(\exists p.\ (p, t, \_) \in \sigma_t.\textsc{stacks}(\ell) \wedge p \neq \mathsf{Disabled}\big) & \text{otherwise} \end{cases}$$

$$\mathsf{TagMapInvPost}(k, t, \ell, \sigma_t) \triangleq \begin{cases} [(\mathsf{Unique}, t, \bot)] = \sigma_t.\textsc{stacks}(\ell) & \text{where } k = \mathsf{Local} \\ (\mathsf{Unique}, t, \_) = \mathsf{head}(\sigma_t.\textsc{stacks}(\ell)) & \text{where } k = \mathsf{Unique} \\ t \in \mathsf{ActiveSharedRO}(\sigma_t.\textsc{stacks}(\ell)) & \text{where } k = \mathsf{Pub} \end{cases}$$

$$\mathsf{TagMapInv} : \mathit{Res} \times \mathit{State} \times \mathit{State} \to \mathit{Prop}$$

$$\begin{aligned} \mathsf{TagMapInv}(r, \sigma_s, \sigma_t) \triangleq\ & \forall (t \mapsto (k, h)) \in r.\textsc{tmap}, (\ell \mapsto (s_s, s_t)) \in h.\ t < \sigma_t.\textsc{nextptr} \wedge \\ & \mathsf{TagMapInvPre}(k, t, \ell, \sigma_t) \Rightarrow \\ & \sigma_s.\textsc{mem}(\ell) = s_s \wedge \sigma_t.\textsc{mem}(\ell) = s_t \wedge \mathsf{TagMapInvPost}(k, t, \ell, \sigma_t) \end{aligned}$$

All owned Call IDs are still in the call stack, and they protect the tags in T:

$$\mathsf{CallMapInv} : \mathit{Res} \times \mathit{State} \to \mathit{Prop}$$

$$\begin{aligned} \mathsf{CallMapInv}(r, \sigma) \triangleq\ & \forall (c \mapsto T) \in \mathrm{dom}(r.\textsc{cmap}).\ c \in \sigma.\textsc{calls} \wedge \\ & \forall (t \mapsto L) \in T, \ell \in L.\ t < \sigma.\textsc{nextptr} \wedge \\ & \exists p.\ (p, t, c) \in \sigma.\textsc{stacks}(\ell) \wedge p \neq \mathsf{Disabled} \end{aligned}$$

A location is private with some tag either it is protected by an owned call ID, or has a local tag

$$\mathsf{PrivLoc} : \mathit{Res} \times \mathit{Loc} \times \mathit{PtrId} \to \mathit{Prop}$$

$$\begin{aligned} \mathsf{PrivLoc}(r, \ell, t) \triangleq\ & \ell \in \mathrm{dom}(r.\textsc{tmap}(t).1) \wedge \\ & \big(r.\textsc{tmap}(t).0 = \mathsf{Local} \vee \big(r.\textsc{tmap}(t).0 = \mathsf{Unique} \wedge \ell \in r.\textsc{cmap}(c)(t)\big)\big) \end{aligned}$$

The two physical states must mostly be the same, except for the values of private locations:

$$\mathsf{SrcTgtRel} : \mathit{Res} \times \mathit{State} \times \mathit{State} \to \mathit{Prop}$$

$$\begin{aligned} \mathsf{SrcTgtRel}(r, \sigma_s, \sigma_t) \triangleq\ & \sigma_s.\textsc{stacks} = \sigma_t.\textsc{stacks} \wedge \sigma_s.\textsc{nextptr} = \sigma_t.\textsc{nextptr} \wedge \\ & \sigma_s.\textsc{calls} = \sigma_t.\textsc{calls} \wedge \sigma_s.\textsc{nextcall} = \sigma_t.\textsc{nextcall} \wedge \\ & \forall \ell \in \mathrm{dom}(\sigma_t.\textsc{mem}).\ ((\sigma_s.\textsc{mem}(\ell), \sigma_t.\textsc{mem}(\ell)) \in \mathcal{A}(r) \vee \exists t.\ \mathsf{PrivLoc}(r, \ell, t)) \end{aligned}$$

The idea of a *private location* reflects the concept that not all locations are "accessible" to unknown code, and hence not all locations must be related between source and target state. The two allowed exceptions are *protected locations* (they have a protected item at the top of their stack, meaning any access with another tag is UB) and *local locations* (they have a singleton stack, meaning any access with another tag is UB).

The key properties of a private location satisfying $\mathsf{PrivLoc}(r, \ell, t)$ are:

- Accesses with any tag other than t are immediate UB, including read accesses.
- t cannot be a public tag.

Together, these mean that any access with a public tag is either UB or does not involve a private location, which is what we need for the adequacy proof.