Stacked Borrows: Appendix

Suppressed for anonymous submission

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, we replaced the memory *Mem* by a larger record *State* that tracks all the state we need, including the next unused pointer ID and call ID (so that we can pick a fresh ID when needed) and the stack of active calls. We also separated the values stored in memory from the stacks for each location; this better matches what we have formalized in Coq.

Finally, scalars include not just pointers and integers, but also ☠ ("poison"), which is used as initial value for freshly allocated memory, as function pointers $\mathsf{FnPointer}(f)$. 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.

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.

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:

$$
\begin{aligned}
\mathit{PtrId} &\triangleq \mathbb{N} \\
\mathit{CallId} &\triangleq \mathbb{N} \\
f \in \mathit{FnName} &\triangleq \mathit{String} \\
\ell \in \mathit{Loc} &\triangleq \mathbb{N} \\
X^? &\triangleq X \uplus \{\bot\} \\
\mathit{Item} &\triangleq \mathit{Permission} \times \mathit{Tag} \times \mathit{CallId}^? \\
t \in \mathit{Tag} &\triangleq \mathit{PtrId}^? \\
p \in \mathit{Permission} &\triangleq \mathsf{Unique} \mid \mathsf{SharedRW} \mid \mathsf{SharedRO} \mid \mathsf{Disabled} \\
\mathit{Stack} &\triangleq \mathit{List}(\mathit{Item}) \\
s \in \mathit{Scalar} &\triangleq \mathsf{Pointer}(\ell, t) \mid z \mid \cancel{\Theta} \mid \mathsf{FnPointer}(f) \qquad \text{where } z \in \mathbb{Z} \\
\sigma \in \mathit{State} &\triangleq \left\{ \begin{array}{r} \textsc{mem} : \mathit{Loc} \xrightarrow{\text{fin}} \mathit{Scalar}, \\ \textsc{stacks} : \mathit{Loc} \xrightarrow{\text{fin}} \mathit{Stack}, \\ \textsc{nextptr} : \mathit{PtrId}, \\ \textsc{calls} : \mathit{List}(\mathit{CallId}), \\ \textsc{nextcall} : \mathit{CallId}, \end{array} \right\}
\end{aligned}
$$

Figure 1: Stacked Borrows domains

1. Find the granting item for a write access with tag t in $\sigma.\textsc{stacks}(\ell)$ (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 $c \in \sigma.\textsc{calls}$, 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 $\sigma.\textsc{stacks}(\ell)$ (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 $c \in \sigma.\text{CALLS}$, 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).

In the previous section, 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. (`retag` is a typed operation, so it can differentiate a raw pointer cast from a reference being created.)

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 $t' \triangleq \sigma.\text{NEXTPTR}$ and increase NEXTPTR by 1. Otherwise, set $t' \triangleq \bot$.
2. Do the following for each location t_x in the range $[\ell_x, \ell_x + \mathsf{sizeof}(\texttt{T}))$:
   (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` $p \triangleq \mathit{PermSRW}$, else (outside `UnsafeCell`) $p \triangleq \mathsf{SharedRO}$.

   (b) The new item we want to add is (p, t', c) where $c \triangleq \mathsf{head}(\sigma.\text{CALLS})$ 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 $\sigma.\text{STACKS}(\ell)$.

   (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.

2 Simulation Setup

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

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.

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

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

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

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.