RefinedRust: Technical Documentation

Lennard Gäher Michael Sammler Ralf Jung Robbert Krebbers

Derek Dreyer

1 Typing Rules

We present a few selected typing rules. RefinedRust's type checker proceeds in a procedural way, which we explain through monadic-style code here.

Figure 1 shows the procedure CHK-MUT-BOR for type checking a mutable borrow of expression e at lifetime `'a`.

Figure 2 shows the procedure CHK-PLACE-ACCESS for type checking a place access of to a location l_o with type ρ_o using place accesses $\mathcal{P}$, which is used by both CHK-MUT-BOR and CHK-READ.

Figure 3 shows the procedure CHK-READ for type checking a read of expression e.

2 Model

In this section, we explain a few of the challenges in the implementation of the RefinedRust type system. Our type system is developed using the technique of *semantic typing* within the Iris separation logic framework embedded in the Coq proof assistant. This means that for each type, the meaning of its ownership predicate (*e.g.*, $l \lhd x \mathbin{@} \rho$) is defined as a separation logic predicate, and the typing rules are proved as lemmas with respect to this model.

Reference types are a particular challenge to model, because it is complicated to state their borrow protocol formally: we need to describe what it means to borrow the reference's contents from the lender for a particular lifetime, and what it means for the lender to regain ownership after

procedure CHK-MUT-BOR(e, `'a`)
  $(\mathcal{P}, l_o) \leftarrow$ DECOMPOSE-EXPRESSION(e)
  $(x_o, \rho_o) \leftarrow$ FIND-ASSIGNMENT-FOR(l_o) — find type assignment $l_o \lhd x_o \mathbin{@} \rho_o$
  $(l_i, x_i, \rho_i, k_{\min}, \rho[\cdot]) \leftarrow$ CHK-PLACE-ACCESS($l_o, x_o, \rho_o, \mathcal{P}$)
  — apply place accesses $\mathcal{P}$ to l_o, resulting in $l_i \lhd x_i \mathbin{@} \rho_i$
  $(x_i, T_i) \leftarrow$ STRATIFY($l_i, x_i, \rho_i, k_{\min}$) — stratify to $l_i \lhd \#x_i \mathbin{@} \textbf{place}\ T_i$
  assert(k_{min} allows mutable borrow at `'a`)
  $\gamma \leftarrow$ CREATE-BORROW-NAME(x_i)
  $(x'_o, \rho'_o) \leftarrow$ FILL-PCTX($\rho[\cdot], *\gamma, \textbf{blocked}^{\texttt{'a}}\ T_i$) — new type for l_o: $l_o \lhd x'_o \mathbin{@} \rho'_o$
  ADD-TO-CONTEXT($l_o \lhd x'_o \mathbin{@} \rho'_o$) — release updated ownership
  return($(\#x_i, \gamma) \mathbin{@} \&^{\texttt{'a}}_{\text{mut}}\ T_i$) — return type of expression

Figure 1: Procedure for type checking mutable borrows.

procedure CHK-PLACE-ACCESS(l_o, x_o, ρ_o, $\mathcal{P}$)
  $(k_{\min}, l_i, x_i, \rho_i, \rho[\cdot]) \leftarrow (\mathsf{Owned}, l_o, x_o, \rho_o, \cdot)$
  while $(a :: \mathcal{P}) \leftarrow \mathcal{P}$ **do**
    $x_i \leftarrow$ USE-BORROW-RESOLUTION(ρ_i, x_i) — Use resolutions at the head
    $\rho_i \leftarrow$ PLACE-UNFOLD-HEAD(ρ_i) — Unfold **place** T at the head, if necessary
    match a, ρ_i **with**
      case $\mathsf{Field}(f)$, $\mathbf{struct}_{sd}\ \vec{\rho}$:
        assert(field f is a field of sd)
        $(\rho_f, x_f) \leftarrow (\vec{\rho}!!f, x_i!!f)$ — look up the type x_f @ ρ_f of the field
        $\rho[\cdot] \leftarrow \ldots$
        $(k_{\min}, l_i, x_i, \rho_i) \leftarrow (k_{\min}, l_i\ \mathrm{AtField}_{sd}\ f, x_f, \rho_f)$
      case Deref, $\&^{\kappa}_{\mathbf{mut}}\rho$: $\ldots$
  return$(l_i, x_i, \rho_i, k_{\min}, \rho[\cdot])$

Figure 2: Procedure for type checking place accesses.

procedure CHK-READ(e)
  $(\mathcal{P}, l_o) \leftarrow$ DECOMPOSE-EXPRESSION(e)
  $(x_o, \rho_o) \leftarrow$ FIND-ASSIGNMENT-FOR(l_o) — have type $l \lhd x_o$ @ ρ_o
  — access l_o with place accesses $\mathcal{P}$, resulting in $l_i \lhd x_i$ @ ρ_i
  $(l_i, x_i, \rho_i, k_{\min}, \rho[\cdot]) \leftarrow$ TYPECHECK-PLACE-ACCESS$(l_o, x_o, \rho_o, \mathcal{P})$
  $(x_i, T_i) \leftarrow$ STRATIFY$(l_i, x_i, \rho_i, k_{\min})$ — stratify to $l_i \lhd \#x_i$ @ **place** T_i
  if T_i is **Copy then**
    assert($k_{\min}$ allows reads)
    $(x'_o, \rho'_o) \leftarrow$ FILL-PCTX$(\rho[\cdot], \#x_i, \textbf{place}\ T_i)$ — place stratified type in $\rho[\cdot]$
    ADD-TO-CONTEXT$(l_o \lhd x'_o$ @ $\rho'_o)$
    return$(x_i$ @ $T_i)$ — return result of expression
  else
    assert$(k_{\min} = \mathsf{Owned})$ — move out ownership
    $(x'_o, \rho'_o) \leftarrow$ FILL-PCTX$(\rho[\cdot], _, \textbf{place}(\mathsf{uninit}))$ — leave uninitialized memory
    ADD-TO-CONTEXT$(l_o \lhd x'_o$ @ $\rho'_o)$
    return$(l_i \lhd_v x_i$ @ $\tau_i)$ — return result of expression

Figure 3: Procedure for reads.

FULL-BOR-CREATE

$$P \Rrightarrow \&^{\kappa}_{\mathbf{full}} P * (\kappa \text{ dead} \Rrightarrow P)$$

FULL-BOR-ACC-STRONG

$$\&^{\kappa}_{\mathbf{full}} P * [\kappa]_q \Rrightarrow P * \left(\forall P'.\, P' * (P' * \kappa \text{ dead} \Rrightarrow P) \Rrightarrow \&^{\kappa}_{\mathbf{full}} P' * [\kappa]_q\right)$$

Figure 4: Selection of rules of RustBelt's lifetime logic (simplified).

the lifetime has ended. To this end, RustBelt has developed the *lifetime logic* which makes the notions of borrows and lifetimes precise.

Background: Lifetime logic The lifetime logic defines borrows and lifetimes as generic separation logic mechanisms. For modelling mutable references, the lifetime logic's *full borrows* allow splitting the ownership of a separation logic proposition in time—to see how, consider the rule FULL-BOR-CREATE. This rule states that ownership of P can be split into two parts by performing an *update* $\Rrightarrow$ (essentially, updates are a version of implication that is suitable for Iris): First, $\&^{\kappa}_{\mathbf{full}} P$ gives access to P for as long as the symbolic lifetime κ is alive. Secondly, the *inheritance* κ dead $\Rrightarrow P$ states what happens after the lifetime κ has ended: then, a death certificate κ dead for κ can be used to get back P (with 0% inheritance tax!). The combination of these two parts is conjoined with separation logic's usual separating conjunction for providing spatial separation.

Now, for accessing a borrow, the lifetime logic provides the rule FULL-BOR-ACC-STRONG that can be used to access a borrow. Assuming ownership of the borrow $\&^{\kappa}_{\mathbf{full}} P$ and a certificate $[\kappa]_q$ that κ is still alive, the rule gives access to P as well as a way to close the borrow again. Let us explore how this rule allows closing a borrow in a bit more detail:

$$\forall P'.\, P' * (P' * \kappa \text{ dead} \Rrightarrow P) \Rrightarrow \&^{\kappa}_{\mathbf{full}} P' * [\kappa]_q$$

Essentially, it allows to pick new contents of the borrow P'. In addition, the user needs to show that P' can be shifted back to P once κ is dead. This is required so that the lender can get back their expected ownership. Under these conditions, the full borrow $\&^{\kappa}_{\mathbf{full}} P'$ can be established again and the proof of liveness for κ is returned.

Model of Mutable references Let us now consider a simplified version of the model of mutable references for RefinedRust's value type. Leaving aside refinements and other details, we can essentially define value ownership of a mutable reference (in the same way as RustBelt) as

$$l \lhd_v \_ \,@\, \&^{\kappa}_{\mathrm{mut}} T \triangleq \&^{\kappa}_{\mathbf{full}}(\exists v'.\, l \mapsto v' * v' \lhd_v \_ \,@\, T),$$

where we use a borrow $\&^{\kappa}_{\mathbf{full}}(\exists v'.\ \ldots)$ to control ownership of the reference's contents. In particular, the *points-to* connective $l \mapsto v'$ of separation logic states that the memory location l is fully owned and stores value v'. Then, the semantic interpretation essentially states that the reference owns the memory location and the recursive interpretation of T for the stored value v', for as long as κ is alive.

Now, let us consider how we can model unique ownership of RefinedRust's place types. One key difference to the value type for mutable references seen before is that place types also need to allow

PINNED-BOR-FOLD

$$\&^{\kappa}_{\mathbf{pin}}(P \mid P) \Rrightarrow \&^{\kappa}_{\mathbf{full}} P$$

PINNED-BOR-UNFOLD

$$\&^{\kappa}_{\mathbf{full}} P \Rrightarrow \&^{\kappa}_{\mathbf{pin}}(P \mid P)$$

PINNED-BOR-ACC-STRONG

$$\&^{\kappa}_{\mathbf{pin}}(Q \mid P) * [\kappa]_q \Rrightarrow P * \left(\forall P'.\, P' * (P' * \kappa \text{ dead} \Rrightarrow P \vee Q) \Rrightarrow \&^{\kappa}_{\mathbf{pin}}(Q \mid P') * [\kappa]_q\right)$$

Figure 5: Excerpt of the rules supported by pinned borrows (simplified).

lending out their ownership, in case nested borrows happen below it: for instance, after a reborrow below two nested references, the place type $\&^{\kappa_1}_{\mathbf{mut}}(\&^{\kappa_2}_{\mathbf{mut}}(\mathbf{blocked}^{\kappa}\ T))$ will occur.

Suppose we use the lifetime logic's full borrows to model place types in the Uniq ownership mode. The interpretation for $\&_{\mathbf{mut}}$ might look like this:

$$l \lhd^{\mathsf{Uniq}_{\kappa_1},\_} \_ @ \&^{\kappa_2}_{\mathbf{mut}} \rho \triangleq \&^{\kappa_1}_{\mathbf{full}}(\exists l'.\, l \mapsto l' * l' \lhd^{\mathsf{Uniq}_{\kappa_2},\_} \_ @ \rho)$$

Note how the ownership modes essentially push down the interpretation of the borrow and we interpret the borrow of the reference above this one at κ_1 in this definition. Suppose we now write to this reference, overwriting its current contents. In the proof of the corresponding proof rule, we would open the borrow using FULL-BOR-ACC-STRONG and then perform the write to l. Afterwards, we would close the borrow again, instantiating the new ownership P' with the ownership of the newly written reference. But now we need to show that the new contents of the borrow can be shifted to the old contents! In the case that the place type contained **blocked** components before (possibly nested below other types, or with only partially blocked components of a struct type), this can become arbitrarily complex — we now need to show that we can somehow "imagine" that the type was blocked!

Intuitively, this first attempt at a model for the place type has a mismatch: why should we always need to show that the contents of the borrow can be returned to the previous contents? After all, the lender of the borrow anyways does not expect to get any **blocked** back—this needs to be in turn shifted back to what the lender actually expects back. In principle, we would like to be able to take a shortcut here: just show that whatever we write to the mutable reference can be returned to what the lender expects back, and then we should be done!

Pinned Borrows To tackle this mismatch, we introduce a new kind of borrows, pinned borrows, that precisely allow us to take this shortcut. Pinned borrows $\&^{\kappa}_{\mathbf{pin}}(Q \mid P)$ precisely remember the ownership Q the lender expects back after κ ends, in addition to the currently available ownership P. This avoids the issue of "lost information" we encountered before.

Pinned borrows can be obtained from regular borrows at any time, and folded back when the current contents P match the pinned ownership Q, as expressed by the rules PINNED-BOR-FOLD and PINNED-BOR-UNFOLD.

The key new rule provided by pinned borrows is PINNED-BOR-ACC-STRONG: it states how the contents P of a pinned borrow $\&^{\kappa}_{\mathbf{pin}}(Q \mid P)$ may be accessed and modified by opening the borrow. It differs from FULL-BOR-ACC-STRONG in precisely one place: instead of showing that the new ownership P' can be returned to the previous contents P, they need to be returned to $P \vee Q$. This gives the user the choice of just showing that ownership can be returned to the previous state P, or showing that it can be returned to the pinned state Q that the lender expects back.

We have chosen to implement pinned borrows as an abstraction on top of the existing lifetime logic, instead of directly altering its model. In doing so, we had to deal with the usual issues with layering abstractions in Iris that arise from its use of the step-indexed later modality for ensuring soundness. The recently-proposed later credits mechanism Spies et al. (2022) enabled us to work around these issues. These complications do not show up in the simplified rules given in Figure 5, but we explain them in §3.

Having pinned borrows, we can implement the Uniq mode for place types as

$$l \lhd^{\mathsf{Uniq}_{\kappa_1},\_} \_ @ \mathbf{\&}^{\kappa_2}_{\mathbf{mut}} \rho \triangleq \&^{\kappa_1}_{\mathbf{pin}}((\exists l'.\, l \mapsto l' * l' \lhd^{\mathsf{Uniq}_{\kappa_2},\_} \_ @ \lfloor \rho \rfloor) \mid (\exists l'.\, l \mapsto l' * l' \lhd^{\mathsf{Uniq}_{\kappa_2},\_} \_ @ \rho)),$$

where $\lfloor \rho \rfloor$ denotes the so-called *core* of a place type—the place type obtained by "unblocking" everything in it. This core is precisely what the lender of the reference expects to get back.[1] Now, when accessing the reference in the proofs of the typing rules, we just need to show that the contents are compatible with the core of the current contents. Precisely, we can define compatibility $\mathsf{compatible}(\rho_1, \rho_2)$ of two place types as follows: $\mathsf{compatible}(\rho_1, \rho_2) \triangleq \lfloor \rho_1 \rfloor \equiv \lfloor \rho_2 \rfloor$.

3 Model of Pinned Borrows

For our model of types, we extend the lifetime logic with another kind of borrows, *pinned borrows* $\&^{\kappa}_{\mathbf{pin}}(Q \mid P)$. In contrast to full borrows, pinned borrows take another propositional parameter Q that "pins"/exposes the information which ownership Q the lender eventually expects, once κ has ended. This allows us to obtain stronger accessors that allow changing the current proposition P to P', but only requires to prove a viewshift going back to Q (instead of P), intuitively (the following rule omits some details, like laters):

LftL-pinned-bor-acc-strong-proto
$$\&^{\kappa}_{\mathbf{pin}}(Q \mid P) * [\kappa]_q \Rrightarrow_{\mathcal{N}_{\mathrm{lft}}} P * \left(\forall P'.\ (P' * \kappa \text{ dead} \Rrightarrow\!\!\!\!\!\!\ast_{\emptyset} Q) * P' \Rrightarrow\!\!\!\!\!\!\ast_{\mathcal{N}_{\mathrm{lft}}} \&^{\kappa}_{\mathbf{pin}}(Q \mid P') * [\kappa]_q\right)$$

Rules for pinned borrows The full set of rules we have is:

LftL-pinned-bor-fake
$$\kappa \text{ dead} \Rrightarrow_{\mathcal{N}_{\mathrm{lft}}} \&^{\kappa}_{\mathbf{pin}}(Q \mid P)$$

LftL-pinned-bor-shorten
$$\kappa \sqsubseteq \kappa' * \&^{\kappa'}_{\mathbf{pin}}(Q \mid P) \twoheadrightarrow \&^{\kappa}_{\mathbf{pin}}(Q \mid P)$$

LftL-pinned-bor-fold
$$\&^{\kappa}_{\mathbf{pin}}(P \mid P) * \pounds(1) \Rrightarrow_{\mathcal{N}_{\mathrm{lft}}} \&^{\kappa}_{\mathbf{full}} P$$

LftL-pinned-bor-unfold
$$\&^{\kappa}_{\mathbf{full}} P * \pounds(1) \Rrightarrow_{\mathcal{N}_{\mathrm{lft}}} \&^{\kappa}_{\mathbf{pin}}(P \mid P)$$

LftL-pinned-bor-acc-strong
$$\&^{\kappa}_{\mathbf{pin}}(Q \mid P) * [\kappa]_q \Rrightarrow_{\mathcal{N}_{\mathrm{lft}}} \exists \kappa'.\, \kappa \sqsubseteq \kappa' * \triangleright P * \triangleright(P * \kappa' \text{ dead} \Rrightarrow\!\!\!\!\!\!\ast_{\emptyset} \triangleright Q) *$$
$$\left(\forall P'.\ \triangleright(P' * \kappa' \text{ dead} \Rrightarrow\!\!\!\!\!\!\ast_{\emptyset} \triangleright Q) * \pounds(1) * \triangleright P' \Rrightarrow\!\!\!\!\!\!\ast_{\mathcal{N}_{\mathrm{lft}}} \&^{\kappa'}_{\mathbf{pin}}(Q \mid P') * [\kappa]_q\right)$$

LftL-pinned-bor-impl
$$\&^{\kappa}_{\mathbf{pin}}(Q \mid P) * \triangleright \Box(P \to P') * \triangleright \Box(P' \to Q) \twoheadrightarrow \&^{\kappa}_{\mathbf{pin}}(Q \mid P')$$

LftL-pinned-bor-iff
$$\&^{\kappa}_{\mathbf{pin}}(Q \mid P) * \triangleright \Box(P \leftrightarrow P') * \triangleright \Box(Q \leftrightarrow Q') \twoheadrightarrow \&^{\kappa}_{\mathbf{pin}}(Q' \mid P')$$

[1] In order to define the notion of a core, we need the set of place types to be closed.

The rules use a recently-proposed extension of Iris, Later Credits Spies et al. (2022), in order to encode the rules impredicatively in Iris's step-indexed logic.

The use of later credits in the folding/unfolding rules as well as the accessors stems from the fact that we currently model pinned borrows on top of the existing lifetime logic (see below). This nesting of higher-order abstractions is problematic, because we need additional higher-order ghost state on top of the existing abstraction provided by the lifetime logic, thus giving us *two* laters in many places. With two laters, even a lot of timelessness reasoning breaks, which is why we include credits in the rules to just get one later, but with the obligation to provide a credit in some places.

We moreover obtain the following derived rules:

LftL-pinned-bor-acc-back

$$\&^{\kappa}_{\mathbf{pin}}(Q \mid P) * [\kappa]_q \Rrightarrow_{\mathcal{N}_{\mathrm{lft}}} \triangleright P * \left(\triangleright Q * \pounds(1) \Rrightarrow\!\!\!\!\ast_{\mathcal{N}_{\mathrm{lft}}} \&^{\kappa}_{\mathbf{pin}}(Q \mid Q) * [\kappa]_q\right)$$

LftL-pinned-bor-acc

$$\&^{\kappa}_{\mathbf{pin}}(Q \mid P) * [\kappa]_q \Rrightarrow_{\mathcal{N}_{\mathrm{lft}}} \triangleright P * \left(\triangleright P * \pounds(1) \Rrightarrow\!\!\!\!\ast_{\mathcal{N}_{\mathrm{lft}}} \&^{\kappa}_{\mathbf{pin}}(Q \mid P) * [\kappa]_q\right)$$

Shallow model for pinned borrows We now define a model of pinned borrows on top of the existing lifetime logic. The idea for defining $\&^{\kappa}_{\mathbf{pin}}(Q \mid P)$ is to use a regular full borrow that stores both the current contents P as well as a viewshift from P to Q that can be executed as part of the inheritance viewshift for returning ownership to the lender. The crucial part is that we need to introduce one level of indirection for storing P: if we would directly store $P * (P \Rrightarrow\!\!\!\!\ast\, Q)$ (roughly) in the borrow, we would run into the same problem in proving LftL-pinned-bor-acc-strong that motivates pinned borrows in the first place. We would have to prove that $P' * (P' \Rrightarrow\!\!\!\!\ast\, Q)$ can be updated back to $P * (P \Rrightarrow\!\!\!\!\ast\, Q)$ (the old contents of the full borrow), but in general there will be no connection between P and P'! To solve this, we existentially quantify over P in the full borrow and link this up to the outside world with a mutable fractional saved proposition.

$$\begin{aligned}\&^{\kappa}_{\mathbf{pin}}(Q \mid P) \triangleq \exists \gamma, \kappa', P'.\, \kappa \sqsubseteq \kappa' * \mathsf{saved}^{\gamma}_{1/2}(P') *\\ &\triangleright \square(P' \to P) *\\ &\triangleright \square(P \to P' \vee Q) *\\ &\&^{\kappa'}_{\mathbf{full}}\left(\exists P_0.\, \mathsf{saved}^{\gamma}_{1/2}(P_0) * P_0 * \pounds(1) * (P_0 \mathrel{-\!\!\!*} \kappa' \text{ dead} \Rrightarrow\!\!\!\!\ast_{\emptyset} \triangleright Q)\right)\end{aligned}$$

Defining a "proper" extension of the existing lifetime logic that yields the same rules as above, but without the later credits, should be perfectly feasible, but would require changing large parts of the foundations of the lifetime logic (namely, borrow boxes) in non-trivial ways.

References

Simon Spies, Lennard Gäher, Joseph Tassarotti, Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. 2022. Later credits: resourceful reasoning for the later modality. *Proc. ACM Program. Lang.* 6, ICFP (2022), 283–311. https://doi.org/10.1145/3547631