Sound Borrow-Checking for Rust via Symbolic Semantics

SON HO, Inria, France
AYMERIC FROMHERZ, Inria, France
JONATHAN PROTZENKO, Microsoft Azure Research, USA

The Rust programming language continues to rise in popularity, and as such, warrants the close attention of the programming languages community. In this work, we present a new foundational contribution towards the theoretical understanding of Rust's semantics. We prove that LLBC, a high-level, borrow-centric model previously proposed for Rust's semantics and execution, is sound with regards to a low-level pointer-based language *à la* CompCert. Specifically, we prove the following: that LLBC is a correct view over a traditional model of execution; that LLBC's symbolic semantics are a correct abstraction of LLBC programs; and that LLBC's symbolic semantics act as a borrow-checker for LLBC, i.e. that symbolically-checked LLBC programs do not get stuck when executed on a heap-and-addresses model of execution.

To prove these results, we introduce a new proof style that considerably simplifies our proofs of simulation, which relies on a notion of hybrid states. Equipped with this reasoning framework, we show that a new addition to LLBC's symbolic semantics, namely a join operation, preserves the abstraction and borrow-checking properties. This in turn allows us to add support for loops to the Aeneas framework; we show, using a series of examples and case studies, that this unlocks new expressive power for Aeneas.

CCS Concepts: • **Theory of computation** → **Operational semantics**.

Additional Key Words and Phrases: Rust, Verification, Semantics

ACM Reference Format:
Son Ho, Aymeric Fromherz, and Jonathan Protzenko. 2024. Sound Borrow-Checking for Rust via Symbolic Semantics. *Proc. ACM Program. Lang.* 8, ICFP, Article 251 (August 2024), 29 pages. https://doi.org/10.1145/3674640

1 Introduction

The Rust programming language continues to rise in popularity. Rust has by now become the darling of developers, voted the most admired language for the 8th year in a row [StackOverflow 2023]; a favorite of governments, who promote safe languages in the context of cybersecurity [National Security Agency 2022]; an industry bet, wherein large corporations are actively migrating to Rust [The Register 2022]; and an active topic in the programming languages research community.

Two research directions have emerged over the past few years. First, many tools now set out to verify Rust programs and prove properties about their behavior, e.g., show that a Rust program matches its specification. But to confidently reason about Rust programs, one needs a solid semantic foundation to build upon. This is the second research direction, namely, understanding the semantics of Rust itself, clarifying the language specification, and showing the soundness of Rust's type system.

For Rust verification, we find tools such as Creusot [Denis et al. 2022], Prusti [Astrauskas et al. 2019], Verus [Lattuada et al. 2023], or hax [Kiefer and Franceschino 2023]. All of those leverage one key insight: the strong ownership discipline enforced by the Rust type system makes verification

Authors' Contact Information: Son Ho, Inria, Paris, France, son.ho@inria.fr; Aymeric Fromherz, Inria, Paris, France, aymeric.fromherz@inria.fr; Jonathan Protzenko, Microsoft Azure Research, Redmond, USA, protz@microsoft.com.

ACM 2475-1421/2024/8-ART251
https://doi.org/10.1145/3674640

easier. For Creusot and Verus, this observation turns into a first-order logical encoding of Rust programs that can then be discharged to an SMT solver. For Prusti, the Rust discipline guides the application of separation logic rules in the underlying Viper framework [Müller et al. 2016]. And for hax, restricting programs to a pure value-passing subset of Rust allows writing an almost identity-like translation to backends such as F* [Swamy et al. 2016] or ProVerif [Blanchet 2001].

For Rust semantics, RustBelt [Jung et al. 2017] aims to prove the soundness of Rust's type system using a minimalistic model called λ_{Rust}, whose operational semantics is defined by compilation to a core language. The MIRI project [Miri Team 2021] aims to provide a reference operational semantics of Rust, even in the presence of unsafe code. Stacked [Jung et al. 2019] and Tree Borrows [Villani 2023] aim to clarify the aliasing contract between the programmer and the Rust compiler.

At the intersection of these two axes is RustHornBelt [Matsushita et al. 2022], which aims to prove that the RustHorn [Matsushita et al. 2020] logical encoding of Rust programs (as used in Creusot) is sound with regards to the semantics of λ_{Rust}, and consequently, that properties proven thanks to the logical encoding hold for the original program.

Also straddling both axes is Aeneas [Ho and Protzenko 2022], a Rust verification project that tackles the question of Rust source semantics too. For semantics, Aeneas introduces the low-level borrow calculus (LLBC), a core language inspired by Rust's Mid-Level IR (MIR) and that directly models borrows, loans, and places as found in source Rust programs, as well as delicate patterns such as reborrows or two-phase borrows. Importantly, the LLBC semantics relies on a map from variables to borrow-centric values, instead of a traditional heap and addresses. For verification, Aeneas relies on two steps: first, an abstraction of LLBC called the symbolic semantics, which operates in a modular fashion by summarizing the ownership behavior of functions based on their types; second, a translation that consumes the execution traces of the symbolic semantics to construct a pure, executable program that is functionally equivalent to the source Rust. Properties can then be proven upon the pure, translated code, and carry over to the original Rust code.

Aeneas, as currently presented, exhibits two issues. First, the correctness of the Aeneas endeavor is predicated on the LLBC model being a sound foundation. LLBC has several unusual features, such as attaching values to pointers rather than to the underlying memory location, or not relying on an explicit heap; right now, it requires a leap of faith to believe that this is an acceptable way to model Rust programs. The value of LLBC is that it explains, checks and provides a semantic foundation for reasoning about many Rust features; but the drawback is that one has to trust that this semantics makes sense. We remark already that this question is orthogonal to the RustBelt line of work. RustBelt attempts to establish the soundness of Rust's type system with regards to λ_{Rust}, whose classic, unsurprising semantics does not warrant scrutiny. Clarifying the link between LLBC and a standard heap-and-addresses model is not just a matter of theory: once the Rust compiler emits LLVM bitcode, the heap and addresses become real; if Aeneas is proving properties about an execution model that cannot be connected to such semantics, then the validity of the whole Aeneas endeavor is threatened.

The second shortcoming of Aeneas concerns its symbolic semantics, which trades concrete program executions for abstract ones that forget about precise values, and that rely on function signatures for modularity. This symbolic semantics is an essential step to materialize the functional translation. Yet, in spite of carefully-crafted rules that emphasize readability and simplicity, there is no formal argument that it correctly approximates LLBC. Specifically, the Aeneas paper claims that the interpreter for the symbolic semantics acts as a borrow-checker for LLBC programs, but has no formal justification for this claim. The symbolic semantics also suffer from limitations related to control-flow: continuations of if-then-elses are duplicated in each branch, and there is no support for loops. We posit that, lacking a formal basis in which to reason about the correctness of the

symbolic semantics, designing a general-purpose "join" operation that can reconcile two symbolic states was out of reach.

In this work, we set out to address both shortcomings of Aeneas, and introduce new proof techniques to do so. First, we establish that LLBC is, indeed, a reasonable model of execution, and that in spite of some mildly exotic features, it really does connect to a traditional heap-and-addresses model of execution. Second, we show that the symbolic semantics of LLBC correctly approximates its concrete semantics, and thus that successful executions in the symbolic semantics guarantee the soundness of concrete executions – in short, that the symbolic interpreter acts as a borrow checker.

Because LLBC can, in some situations, analyze code more precisely than the Rust borrow-checker, we are therefore able to prove the soundness of more programs than the Rust compiler itself. We thus hope that this work sheds some light on potential improvements to the current implementation of the borrow-checker, backed by an actual formal argument. Our contributions are as follows:

We introduce PL, for "pointer language", which uses a traditional, explicit heap inspired by CompCert's C memory model, and show that it refines LLBC (§3). This allows us to establish that a low-level model (where values live at a given address) refines the Rust model given by LLBC (where borrows hold the value they are borrowing). These two semantics have opposite perspectives on what it *means* to have a pointer; to the best of our knowledge, such a connection between two *executable* semantics is novel.

We prove that LLBC itself refines the symbolic version of LLBC, henceforth known as LLBC$^\#$ (§4). Combined with the previous result, this allows us to precisely state why LLBC$^\#$ is a borrow-checker for LLBC: if an LLBC$^\#$ execution succeeds, then any corresponding low-level PL execution is safe for all inputs. We obtain this result by a combination of forward simulations, along with the determinism of the target (PL) language; we also reason about what it means for a low-level (PL) initial state to satisfy a function signature with ownership constraints in LLBC$^\#$.

To conduct these proofs of refinement, we introduce a novel proof technique that relies on the fact that our languages operate over the same grammar of expressions, but give it different *meanings*, or *views* (§2). For instance: both our heap-and-address (PL) and borrows-and-loans (LLBC) views operate over the same program syntax, but have different types for their state and reduction rules. Rather than go full-throttle with a standard compilation-style proof of simulation, we reason modularly over local or pointwise transformations that rewrite one piece of state to another – proofs over those elementary transformations are much easier. We then show that two states that are related by the transitive closure of these elementary transformations continue to relate throughout their execution, in the *union* of the semantics, ultimately giving a proof of refinement.

Equipped with a framework in which to reason about the soundness of LLBC$^\#$ with regards to LLBC, we define and prove correct a new operation that was previously missing from LLBC$^\#$: the join operation, which can reconcile two branches of control-flow (§5.1). Inspired by joins in abstract interpretation (and specifically, joins in shape analysis), this new operation gives us a symbolic interpreter (i.e., a borrow-checker) that can handle loops (§5.2), and does not exhibit pathological complexity on conditionals. Furthermore, our support naturally extends to the functional translation, meaning that Aeneas now supports verification of loops. We leave a discussion of the correctness of the Aeneas functional translation to future work.

We evaluate the effectiveness of our join operation (§6), specifically when used to compute shape fixpoints for loops, over a series of small examples and case studies. We find that, in spite of being (naturally) incomplete, our join operation handles all of the examples we could muster. Our interpretation is that because Rust imposes so many invariants, a join procedure can leverage this structure and fare better than, say, a general-purpose join operation for analyzing C programs.

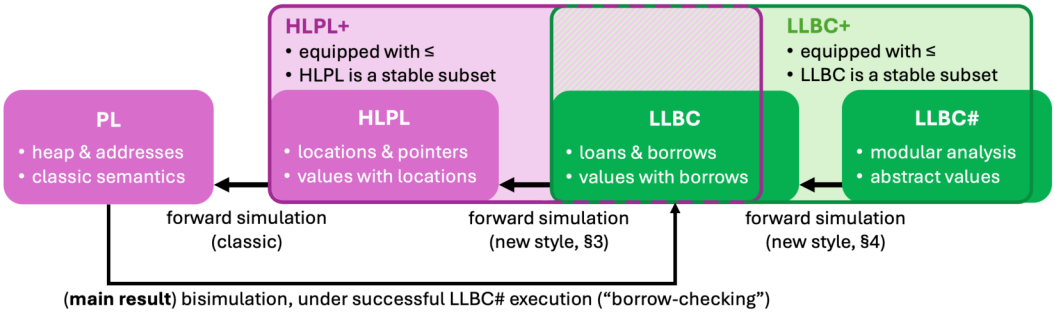

Fig. 1. The architecture of our proof

2 A Generic Approach to Proving Language Simulations

As is standard in this kind of work, we go through several intermediate languages in order to relate our high-level (LLBC$^\#$) and low-level (PL) languages (Figure 1). This allows for modular reasoning, where each step focuses on a particular semantic concept.

To conduct these various refinement steps, we design a new, reusable proof methodology that allows efficiently establishing simulations between languages via the use of local and pointwise reasoning. We use this generic methodology repeatedly throughout the rest of the paper, so as to make our various reasoning steps much more effective; we now present the idea in the abstract, and apply it to our use-cases in subsequent sections (§3, §4).

Our approach consists of a generic proof template for establishing simulations between two languages L_l and L_h that share a grammar of statements, but whose semantics and notions of states differ. To simplify the presentation, we focus on forward simulations, but the methodology can be easily adapted to work dually for backward simulations. We briefly touch on this at the end of this section, and later discuss why we focus on forward simulations in this work, in §3.2.

Formally, we write Ω_l, Ω_h for low-level and high-level states respectively; $\leq$, for a given relation that *refines* a high-level state into a low-level state, i.e., $\Omega_l \leq \Omega_h$; and $\Omega_l \vdash_l e \Downarrow_l (e', \Omega'_l)$ (respectively, h), for the reduction of expressions into another expression and resulting state. We consider variations of such a relation throughout the paper, to relate the reduction of statements to *outcomes* [Huisman and Jacobs 2000; Norrish 1998] used to model non-local control-flow (e.g., `return`, `break`), or the reduction of expressions to values, in which case we consider a relation $\leq$ which refines pairs of high-level states and values to low-level states and values. We assume $\Downarrow$ defines a big-step reduction over which one can perform inductive reasoning; we leave the problem of adapting this proof technique to small-step semantics for future work. We may omit some of the indices when clear from the context.

We aim to establish (variations of) the following property:

Definition 2.1 (Forward Simulation). For all Ω_l, Ω_h, Ω'_h states, e, e' expressions, we have:

$$\Omega_l \leq \Omega_h \Rightarrow \Omega_h \vdash_h e \Downarrow_h (e', \Omega'_h) \Rightarrow \exists\, \Omega'_l,\ \Omega_l \vdash_l e \Downarrow_l (e', \Omega'_l) \wedge \Omega'_l \leq \Omega'_h$$

Proving this property in a direct fashion can be tedious. When states Ω_l and Ω_h heavily differ, for instance because they operate at different levels of abstraction, the state relation commonly consists of complex global invariants, which are tricky to both correctly define and reason about, and oftentimes require maintaining auxiliary data structures, such as maps between high and low, for the purposes of the proof. To circumvent this issue, our approach relies on two key components. First, instead of a global relation between states Ω_l and Ω_h, we define a set of small, local rules, the transitive closure of which constitutes $\leq$ (§2.1). These can then be reasoned upon individually.

Second, we add the ability to reason about states that contain a mixture of high and low, which we dub "hybrid" (§2.2). These now occur because our rewritings operate incrementally, and thus may give rise to states that belong neither to L_l nor L_h. Since the proof of forward simulation involves an induction on $\Downarrow$, we now must reason about the reduction of terms in hybrid states. Note that, if the empty states in L_l and L_h are related, we retrieve standard simulation properties, namely that the execution of a closed program in L_l is related to its execution in L_h. This will be the case for the state relations in §3 and §4.

2.1 Local State Transformations

To illustrate the idea of state relations based on local transformations, we take the simplistic example of a state that contains two integer variables x and y; in Ω_l, x and y contain concrete values; in Ω_h, x and y contain symbolic (or abstract) values. This is a much-simplified version of the full proof we later develop in §4; the setting of abstract/concrete values makes it easy to illustrate the concept. In this example, our goal is to prove that L_h implements a sound symbolic execution for L_l. Instead of defining a global relation using universal quantification on all variables, we instead define a local relation that, for a given variable, swaps its concrete value n and its corresponding abstract version σ in L_h. We concisely write $\Omega[n] \leq \Omega[\sigma]$, leveraging a state-with-holes notation for the state: $\Omega[.]$ is a state with one hole, while $\Omega[v]$ is the same state where the hole has been filled with value v. Establishing $L_l \leq L_h$ involves repeatedly applying this relation; to either x followed by y, or y followed by x; the locality of the transformation enables us to reason modularly about both variables. Naturally, once the reasoning becomes more complex, the ability to consider a single transformation at a time is crucial.

We remark that these individual transformations are non-directed, and can be read either left-to-right or right-to-left. In our example, when read left-to-right, we have an abstraction; when read right-to-left we have a concretization. This supports our later claim that this methodology works for both forward and backward simulations.

2.2 Reasoning over a Superset Language

By defining the state relation $\leq$ as the reflexive, transitive closure of local relations, we can now attempt to prove forward simulation by a standard induction on the evaluation in L_h, followed by an induction on the $\leq$ relation. However, one problem arises: intermediate states do not belong to either language, and hence do not have semantics. To see why, consider in our proof the induction step corresponding to the transitivity of the relation, where we have an intermediate state Ω_m such that $\Omega_l \leq \Omega_m \leq \Omega_h$, and we want to use an induction hypothesis on $\Omega_m \leq \Omega_h$. We do so in order to establish that if $\Omega_h \vdash_{\mathsf{h}} e \Downarrow_h (e', \Omega'_h)$ (for some e', Ω'_h) and $\Omega_m \leq \Omega_h$, then there exists Ω'_m such that $\Omega_m \vdash e \Downarrow (e', \Omega'_m)$ and $\Omega'_m \leq \Omega'_h$ – this is the induction on the size of $\leq$ for the purposes of establishing the forward simulation. This is fine, except reduction is not defined for hybrid states like Ω_m which contain a mixture of high-level and low-level. To address this issue, we instead consider a superset language L^+ that contains semantics from both L_l and L_h.

Coming back to our previous example, let us assume that we want to update variable x. If x has already been rewritten, we use the corresponding semantic rule in L_h, otherwise we execute the program according to the semantics in L_l. For our simplistic example, no further rules are needed, and the strict union suffices, i.e. $L^+ = L_l \cup L_h$. We will see shortly that this strict union is not always adequate in the general case.

With this approach, we establish the following theorem for two concrete languages L_l and L_h (note that in the following $\leq$ will always refer to the reflexive, transitive closure relation):

Theorem 2.2 (Forward Simulation on Superset Language). *For all L^+ states Ω_1, Ω_2, Ω'_2, expressions e, e', we have:*

$$\Omega_1 \leq \Omega_2 \Rightarrow \Omega_2 \vdash_+ e \Downarrow_+ (e', \Omega'_2) \Rightarrow \exists\, \Omega'_1,\ \Omega_1 \vdash_+ e \Downarrow_+ (e', \Omega'_1) \wedge \Omega'_1 \leq \Omega'_2$$

Unfortunately, instantiating this theorem with states $\Omega_1 = \Omega_l \in L_l$ and $\Omega_2 = \Omega_h \in L_h$ does not allow us to derive Theorem 2.1, which was our initial goal. Indeed, Ω_1 initially belongs to L_l but reduces with $\Downarrow_+$, i.e., with the union of the semantics, and so far nothing allows us to conclude that the resulting Ω'_1 is still in L_l. To ensure that this execution is valid with respect to L_l, we need to restrict L^+ by excluding a set of rules R from L_h so that L_l and L^+ satisfy the following property.

Definition 2.3 (Stability). Given two languages L_l, L^+ such that L^+ is a superset of L_l, we say that L_l is a stable subset of L^+ if, for all $e, e', \Omega \in L_l, \Omega^+ \in L^+$, if $\Omega \vdash_+ e \Downarrow_+ (e', \Omega^+)$, then $\Omega^+ \in L_l$ and $\Omega \vdash_l e \Downarrow_l (e', \Omega^+)$

Combined with stability, Theorem 2.2 allows us to directly derive that L_l and L^+ satisfy a forward simulation relation. To conclude, the last remaining step is to establish the same property between L^+ and L_h, which only requires reasoning about the excluded rule set R.

The earlier simplistic example of concrete/abstract integers requires no particular care when defining L_+, and the union of the rules suffices (i.e., $R = \varnothing$). But for our real use-case, one can see from Figure 1 that the semantics of HLPL$^+$ exclude some of the rules from LLBC. We also point out that for our real use-cases, the superset L_+ requires additional administrative rules, to make sure high and low compose (as we will see shortly).

Forward vs. *backward.* While the presentation focused on forward simulations, the approach can be easily adapted to backward simulations. Leveraging the duality between both relations, it is sufficient to exclude a set of rules R from L_l instead of L_h so that L_h becomes a stable subset of L^+, and to similarly conclude by reasoning over the rules in R.

3 A Heap-and-Addresses Interpretation of Valued Borrows

Equipped with our generic proof methodology, we now turn to the Low-Level Borrow Calculus (LLBC), first introduced by Ho and Protzenko [2022], which aims to provide an operational semantics for a core subset of the Rust language. To remain conceptually close to Rust, LLBC operates on states containing loans and borrows. While this helps understanding and explaining the language from the programmer's perspective, this model departs from standard operational semantics for heap-manipulating programs, which commonly rely on a low-level model based on memory addresses and offsets [Jung et al. 2017; Leroy 2009; Leroy et al. 2012]. In this section, we thus bridge the abstraction gap, by relating LLBC to a standard, low-level operational semantics, therefore establishing LLBC as a sound semantic foundation for Rust programs. To do so, we introduce the Pointer Language (PL), a small language explicitly modeling the heap using a model inspired by CompCert's C memory model, and establish a relation between LLBC and PL that demonstrates that LLBC's borrow-centric view of memory is compatible with classic pointers.

We start this section by presenting selected rules from LLBC exhibiting the core concepts, before applying the methodology from §2 to prove a simulation relation between LLBC and PL. Due to space constraints, we only present representative rules of the different languages that exhibit the salient parts of the proofs; the complete presentation is available in the appendix.

3.1 Background: Low-Level Borrow Calculus (LLBC)

LLBC aims to model a core subset of the Rust language after desugaring (Figure 2), where for instance move and copy operations are explicit; conceptually, it is close to Rust's Mid-level IR (MIR). To provide a quick overview of LLBC, we will rely on the following running example, inherited

τ ::=		type	rv ::=		assignable rvalues
	bool | uint32 | ...	literal types		op	operand
	$\&^{\rho}$ mut τ	mutable borrow		&mut p	mutable borrow
	$\&^{\rho}\ \tau$	immutable (shared) borrow		$\& p$	immutable (shared) borrow
	...			...	
ρ		region (or lifetime)	op ::=		operand
s ::=		statement		move p	ownership transfer
	$\emptyset$	empty statement (nil)		copy p	scalar copy
	s; s	sequence (cons)		true | false | n_{u32} | ...	literal constants
	$p := rv$	assignment		...	
	panic	unrecoverable error	P ::=		path
	...			[.]	base case
x		variable		$*P$	deref
p ::= $P[x]$		place		$P.f$	field selection

Fig. 2. The Low-Level Borrow Calculus (LLBC): Syntax (Selected Constructs).

from Ho and Protzenko [2022], which exhibits several salient Rust features, namely, mutable borrows and reborrows. At each program point, we annotate this example with the corresponding LLBC state (we use the terms environments and states interchangeably), which we present below.

```
1  x  = 0;           // x ↦ 0
2  px = &mut x;      // x ↦ loan^m ℓ0,   px ↦ borrow^m ℓ0 0
3  px = &mut (*px); // x ↦ loan^m ℓ0,   _ ↦ borrow^m ℓ0 (loan^m ℓ1),   px ↦ borrow^m ℓ1 0
4  assert!(x = 0);  // x ↦ 0,   _ ↦ ⊥,   px ↦ ⊥
```

Environments (or States). LLBC relies on a borrow-centric view of values, and operates on environments that map variables to their corresponding values. We present below an excerpt of LLBC's grammar of values; the full version also includes support for sums, pairs, recursive types, and reserved borrows which are used to model two-phase borrows [The Rust Compiler Team 2024].

$$v := n \mid \bot \mid \text{loan}^m\ \ell \mid \text{borrow}^m\ \ell\ v \mid \text{loan}^s\ \ell\ v \mid \text{borrow}^s\ \ell$$

The most interesting values consist of borrows and loans, which can either be shared (annotated with the exponent s) or mutable (annotated with the exponent m). Both borrows and loans are associated to a loan identifier ℓ, which can be seen as an abstract notion of location. A loan identifier uniquely identifies a *loan* value, while different borrows may refer to the same identifier in the case of shared borrows. Additionally, mutable borrows carry the borrowed value v. Other values include constants n (e.g., integers), as well as the $\bot$ value which represents both uninitialized and invalidated values, and is needed to model terminating borrows as well as Rust's move semantics.

Semantics. LLBC relies on the operational semantics presented in Figure 3. The judgment $\Omega \vdash s \rightsquigarrow (r,\ \Omega')$ models a big-step semantics where executing statement s in the initial environment Ω yields a new environment Ω', and a control-flow outcomes $r \in \{(), \text{return}, \text{panic}, \text{continue}\ i, \text{break}\ i\}$. The judgment $\Omega \vdash rv \Downarrow (v,\ \Omega')$ is similar, but operates on assignable rvalues (that we also refer to as *expressions*) and returns values.

Initially, the variable x is declared to be 0, and the corresponding mapping is added to the environment (line 1). To execute the mutable borrow at line 2, we turn to the rule E-MutBorrow.

The first premise of the rule ($\vdash \Omega(p) \overset{\text{mut}}{\Rightarrow} v$) retrieves the value v associated to a path p in state Ω. Accesses (Read) and updates (Write) to environments are annotated with one of mut, imm or mov. These capabilities refine the behavior of $\Rightarrow$: for instance, the creation of shared borrows

E-MutBorrow **(LLBC only)**

$$\frac{\vdash \Omega(p) \overset{\text{mut}}{\Rightarrow} v \quad \bot, \text{loan}^{s,m} \notin v \quad \ell \text{ fresh} \quad \Omega \vdash p \leftarrow \text{loan}^m\, \ell \overset{\text{mut}}{\Rightarrow} \Omega'}{\Omega \vdash \&\text{mut}\, p \Downarrow (\text{borrow}^m\, \ell\, v, \Omega')}$$

E-SharedBorrow **(LLBC only)**

$$\frac{\vdash \Omega(p) \overset{\text{imm}}{\Rightarrow} v \quad \bot, \text{loan}^m \notin v \quad \Omega \vdash p \leftarrow v' \overset{\text{imm}}{\Rightarrow} \Omega' \quad v' = \begin{cases} \text{loan}^s\, \ell\, v'' & \text{if } v = \text{loan}^s\, \ell\, v'' \\ \text{loan}^s\, \ell\, v & \ell \text{ fresh otherwise} \end{cases}}{\Omega \vdash \&\, p \Downarrow (\text{borrow}^s\, \ell, \Omega')}$$

E-Move

$$\frac{\vdash \Omega(p) \overset{\text{mov}}{\Rightarrow} v \quad \bot, \text{loan}^{s,m} \notin v \quad \Omega \vdash p \leftarrow \bot \overset{\text{mov}}{\Rightarrow} \Omega'}{\Omega \vdash \text{move}\, p \Downarrow (v, \Omega')}$$

E-Reorg

$$\frac{\Omega_0 \hookrightarrow \Omega_1 \quad \Omega_1 \vdash s \rightsquigarrow (r, \Omega_2)}{\Omega_0 \vdash s \rightsquigarrow (r, \Omega_2)}$$

E-Panic

$$\frac{}{\Omega \vdash \text{panic} \rightsquigarrow (\text{panic}, \Omega)}$$

E-Seq-Unit

$$\frac{\Omega_0 \vdash s_0 \rightsquigarrow ((), \Omega_1) \quad \Omega_1 \vdash s_1 \rightsquigarrow (r, \Omega_2)}{\Omega_0 \vdash s_0; s_1 \rightsquigarrow (r, \Omega_2)}$$

E-Assign **(LLBC only)**

$$\frac{\Omega \vdash rv \Downarrow (v, \Omega') \quad \vdash \Omega'(p) \overset{\text{mut}}{\Rightarrow} v_p \quad v_p \text{ has no outer loan}^{s,m} \quad \Omega' \vdash p \leftarrow v \overset{\text{mut}}{\Rightarrow} \Omega'' \quad \Omega''' = \Omega'', \_ \rightarrow v_p}{\Omega \vdash p := rv \rightsquigarrow ((), \Omega''')}$$

Reorg-End-MutBorrow

$$\frac{\text{hole of } \Omega[\text{loan}^m\, \ell, .] \text{ not inside a borrow} \quad \text{loan}^{s,m} \notin v}{\Omega[\text{loan}^m\, \ell, \text{borrow}^m\, \ell\, v] \hookrightarrow \Omega[v, \bot]}$$

Read

$$\frac{p = P[x] \quad \Omega(x) = v \quad \Omega \vdash P(v) \overset{\text{k}}{\Rightarrow} v'}{\vdash \Omega(p) \overset{\text{k}}{\Rightarrow} v'}$$

R-Base

$$\frac{}{\Omega \vdash [.](v) \overset{\text{k}}{\Rightarrow} v}$$

R-Deref-SharedBorrow

$$\frac{\text{loan}^s\, \ell\, v \in \Omega \quad \Omega \vdash P(v) \overset{\text{imm}}{\Rightarrow} v'}{\Omega \vdash P(*(\text{borrow}^s\, \ell)) \overset{\text{imm}}{\Rightarrow} v'}$$

Write

$$\frac{p = P[x] \quad \Omega(x) = v \quad \Omega \vdash P(v) \leftarrow w \overset{\text{k}}{\Rightarrow} (v', \Omega) \quad \Omega'' = (\Omega'(x) := v')}{\vdash \Omega(p) \leftarrow w \overset{\text{k}}{\Rightarrow} \Omega''}$$

W-Base

$$\frac{}{\Omega \vdash [.](v) \leftarrow w \overset{\text{k}}{\Rightarrow} (w, \Omega)}$$

Fig. 3. Operational Semantics for LLBC (Selected Rules)

(E-SharedBorrow) allows following existing shared borrow indirections along path P and uses the imm ("immutable") capability (R-Deref-SharedBorrow); but moving (E-Move) won't follow dereferences and uses the mov ("move") capability: one can't move a value under a borrow. Once the value is retrieved, LLBC creates a fresh loan identifier ℓ, and sets the value associated to p to $\text{loan}^m\, \ell$, modeling that p has been borrowed. The mutable borrow is finally evaluated to $\text{borrow}^m\, \ell\, v$

The pattern at line 3 is known as a reborrow, and is frequently introduced by the Rust compiler when desugaring the user-written code. After inserting a fresh loan identifier (E-MutBorrow), the right-hand side of the assignment reduces to $\text{borrow}^m\, \ell_1\ 0$, and px *temporarily* maps to $\text{borrow}^m\, \ell_0\ (\text{loan}^m\, \ell_1)$ (E-Assign, Ω'). We now assign $\text{borrow}^m\, \ell_1\ 0$ to px (E-Assign, Ω''), and doing so, we do not directly override the value at px but save it in a ghost, anonymous value to remember the reborrow information that we will need to use later when ending, e.g., ℓ_0 (E-Assign, Ω''').

We now evaluate the `assert`, which is syntactic sugar for `if not ... then panic`. We need to read x, but its value has been mutably borrowed ($\text{loan}^m\, \ell_0$) and is thus not accessible (there is no corresponding $\Rightarrow$ rule to read from a loan): we need to end the borrow ℓ_0. We do this during a reorganization phase, by which we can arbitrarily end borrows before evaluating any statement (E-Reorg). One has to note that it is always possible to end a borrow early, but this may lead to

an unnecessarily stuck execution, if, e.g., we later need to dereference this borrow. We attempt to use Reorg-End-MutBorrow by which we can reinsert a mutably borrowed value back into its loan. However, this rule requires that we have full access to the borrowed value (premise $\text{loan}^{s,m} \notin v$, which states that v doesn't contain any loans, shared or mutable) which is not the case because the value was reborrowed through ℓ_1. We thus apply Reorg-End-MutBorrow on ℓ_1 first, which yields the environment $x \mapsto \text{loan}^m\ \ell_0,\ \_ \mapsto \text{borrow}^m\ \ell_0\ 0,\ p \mapsto \bot$. We finally apply Reorg-End-MutBorrow again on ℓ_0, which yields the final environment.

Shared Borrows. We briefly describe shared borrows, which are simpler than mutable borrows and do not pose a challenge in our proof of soundness. The value of a shared borrow remains attached to the *loan*, as no modifications are permitted anyhow through the shared borrows. E-SharedBorrow creates a new fresh loan if there isn't one at the path p already, and R-Deref-SharedBorrow merely looks up the value at path p in order to access the corresponding value v'. As mentioned earlier, reading *through* shared loans is permitted since we don't modify the value, meaning we don't have to end the corresponding borrows to evaluate the `assert`.

Earlier Formalism. A reader familiar with the original presentation of LLBC might have noticed several minor differences compared to the version presented in this paper. As part of our work on relating LLBC to a lower-level language, we identified several improvements to simplify both the formalism and the proofs. Of particular note, instead of a set of loan identifiers, shared loans now carry a single identifier. We found through our formal analysis that the former was not needed, and that the latter supports a more local style of reasoning that avoids non-local lookup-then-update operations on the originating loan. Another improvement concerns the capabilities for the read and write judgments; not only does this lead to better reuse of rules (no need for separate judgments), it also makes some rules more explicit. We also clarified notations to distinguish between reductions operating on expressions (i.e., rvalues), statements, and environments. Finally, we also extended the grammar of LLBC and the control-flow outcomes to include support for loops.

3.2 Simulation Proof

Our main objective in this section is to prove that LLBC accurately models the Rust semantics, that is, that every execution at the LLBC level admits a corresponding execution in a more standard, heap-manipulating language that we dub the Pointer Language (PL).

Forward vs *Backward.* In our proof, the source and target language are the same – we are not doing a compiler correctness proof. Rather, we reconcile two models of execution over the same syntax of programs, i.e., given a PL state that *concretizes* the LLBC state, the program computes in PL the same result as in LLBC. This is similar to the original circa-2008 style of CompCert proofs, and qualifies as a forward simulation [Leroy 2009].

Indeed, at this stage, it is not true that every PL execution can be simulated backwards by an LLBC execution; simply said, a PL program could be safe for reasons that cannot be explained by LLBC's borrow semantics. We will see in the next section (§4.4) how we can obtain the backwards direction, provided the program is borrow-checked – a result that is akin to a typing result. For now, we simply seek to establish that the execution model of LLBC is a correct restriction of a traditional heap-and-addresses model. To that end, we use the methodology from §2, and for now only use the forward direction it gives us.

Difficulties and Methodology. Instead of environments containing borrows and loans, PL operates on an explicit heap adapted from the CompCert memory model [Leroy et al. 2012]. In particular, loan identifiers are replaced by memory addresses, consisting of a block identifier and an offset, and the memory layout is made semi-explicit, by including a notion of *size* for each type.

When attempting to relate LLBC to a lower-level language like PL, two main difficulties arise. First, manipulating an explicit heap requires operating on sequences of words, which need to be

$$\textsc{E-Pointer}\quad \frac{\vdash \Omega(p) \overset{\text{imm,mut}}{\Rightarrow} v \qquad \vdash \Omega(p) \leftarrow v' \overset{\text{imm,mut}}{\Rightarrow} \Omega' \qquad v' = \begin{cases} v & \text{if } v = \text{loc}\,\ell\,v'' \\ \text{loc}\,\ell\,v & \ell \text{ fresh otherwise} \end{cases}}{\Omega \vdash \{\&p, \&\text{mut}\,p\} \Downarrow (\text{ptr}\,\ell, \Omega')}$$

$$\textsc{Reorg-End-Pointer}\quad \frac{}{\Omega[\text{ptr}\,\ell] \hookrightarrow \Omega[\bot]}$$

$$\textsc{Reorg-End-Loc}\quad \frac{\text{ptr}\,\ell \notin \Omega[\text{loc}\,\ell\,v]}{\Omega[\text{loc}\,\ell\,v] \hookrightarrow \Omega[v]}$$

$$\textsc{R-Loc}\quad \frac{P \neq [.] \qquad \Omega \vdash P(v) \overset{\text{imm,mut}}{\Rightarrow} v'}{\Omega(p) \vdash P(\text{loc}\,\ell\,v) \overset{\text{imm,mut}}{\Rightarrow} v'}$$

$$\textsc{R-Deref-Ptr-Loc}\quad \frac{\text{loc}\,\ell\,v \in \Omega \qquad \Omega \vdash P(\text{loc}\,\ell\,v) \overset{\text{imm,mut}}{\Rightarrow} v'}{\Omega \vdash P(*(\text{ptr}\,\ell)) \overset{\text{imm,mut}}{\Rightarrow} v'}$$

Fig. 4. Selected Rules for HLPL

reconciled with more abstract values. Second, one needs to relate borrows and loans to low-level pointers. The core of the difficulty lies in the fact that in LLBC, mutable borrows "carry" the value they borrow, making it hard to reason about the provenance of a value in the presence of reborrows. This is where our methodology comes in (§2). To make the proof simpler and more modular, we proceed in two steps, via the addition of an intermediary language dubbed HLPL.

An intermediary language: HLPL (High-Level Pointer Language). As before, the program syntax remains the same; what differs now is that HLPL states are half-way between PL states and LLBC states. HLPL states no longer feature borrows and loans; but instead of manipulating a heap they retain an abstract notion of *pointers* and *locations*, denoted respectively $\text{ptr}\,\ell$ and $\text{loc}\,\ell\,v$ (see below).

The simulation from HLPL to PL is standard and only consists in materializing locations as a global heap, and mapping pointers to corresponding addresses. We instead focus on where the challenge lies, namely, going from HLPL (value is with the location) to LLBC (value is with the borrow). Furthermore, we focus specifically on the case of mutable borrows: shared borrows have the value attached to the loan, meaning that the crucial discrepancy only appears in the case of mutable borrows. We cover shared borrows in the long version of the paper [Ho et al. 2024b].

We define the operational semantics of HLPL in Figure 4. Pointers (resp., locations) behave basically like shared borrows (resp., loans), in that the value lives with the location. Unlike shared borrows, we permit modifications through the pointer. In particular, in HLPL we evaluate borrowing expressions like `&mut p` and `&p` with E-Pointer, which is very similar to E-SharedBorrow. We remark that HLPL retains some structure still: one can only update a value $x \longrightarrow \text{loc}\,\ell\,0$ via a corresponding pointer $p \longrightarrow \text{ptr}\,\ell$, e.g., to obtain $x \longrightarrow \text{loc}\,\ell\,1$ (W-Deref-Ptr-Loc, in Appendix). Should one want to update x *itself*, there must be no outstanding pointers to it (Reorg-End-Pointer), and the location itself must have been forgotten (Reorg-End-Loc). Just like in LLBC, these reorganizations may happen at any time, and as in LLBC, a poor choice of reorganizations may lead to a stuck execution.

We annotated the reborrowing example with mutable borrows from the previous section to show the HLPL environments at each program point.

```
1  x  = 0;           // x ↦ 0
2  px = &mut x;      // x ↦ loc ℓ 0,   px ↦ ptr ℓ
3  px = &mut (*px); // x ↦ loc ℓ 0,   _ ↦ ptr ℓ,   px ↦ ptr ℓ
4  assert!(x == 0); // x ↦ loc ℓ 0,   _ ↦ ptr ℓ,   px ↦ ptr ℓ
```

At line 2, we use E-Pointer to introduce a fresh location in x and evaluate `&mut x` to a fresh pointer. Contrary to what happens in LLBC, in HLPL we leave pointed values in place. At line 3, we use E-Pointer again, but this time dereference px (which maps to $\text{ptr}\,\ell$), which yields $\text{loc}\,\ell\,0$, and create another pointer for this location. Because the value we create a pointer to is already a location value, we do not introduce a fresh location and simply evaluate `&mut (*px)` to $\text{ptr}\,\ell$. We now need to

$$\textsc{Le-Shared-To-Ptr}\ \frac{}{\Omega[\text{ptr}\,\ell] \le \Omega[\text{borrow}^s\,\ell]} \qquad \textsc{Le-MutBorrow-To-Ptr}\ \frac{\ell \notin \Omega[.,.] \qquad \ell \notin v}{\Omega[\text{loc}\,\ell\,v,\ \text{ptr}\,\ell] \le \Omega[\text{loan}^m\,\ell,\ \text{borrow}^m\,\ell\,v]}$$

$$\textsc{Le-Merge-Locs}\ \frac{\forall\, v', \text{loc}\,\ell'\,v' \notin \Omega[\text{loc}\,\ell\,v] \qquad \Omega' = \left[\ell \middle/ \ell'\right](\Omega[\text{loc}\,\ell\,v])}{\Omega' \le \Omega[\text{loc}\,\ell\,(\text{loc}\,\ell'\,v)]} \qquad \textsc{Le-SharedLoan-To-Loc}\ \frac{\text{borrow}^s\,\ell \notin \Omega[\text{loc}\,\ell\,v]}{\Omega[\text{loc}\,\ell\,v] \le \Omega[\text{loan}^s\,\ell\,v]}$$

Fig. 5. The $\le$ Relation on HLPL$^+$ states (Selected Rules)

evaluate the assignment to px (left hand side of `px = &mut (*px)`): we move the current value of px (ptr ℓ) to a fresh anonymous value, then override the value of px with the result of evaluating the right-hand side (ptr ℓ). Finally, at line 4, we need to read x. As we can read *through* locations we do not *need* to end any pointer at this point and leave the environment unchanged; note that we *could* preemptively end location ℓ to get the same environment as with the LLBC semantics.

We now use the proof methodology of §2 to show the forward simulation from LLBC to HLPL.

The $\le$ relation between HLPL and LLBC states. Following §2, the first step consists in introducing a series of local rewriting rules (Figure 5), whose transitive closure, written $\le$, relates an HLPL state Ω_{hlpl} to an LLBC state Ω_{llbc}. The relation $\Omega_{\text{hlpl}} \le \Omega_{\text{llbc}}$ can be read in both directions; but since we are concerned here with a forward simulation, we read these rules right to left, that is, we *gradually* transform borrows and loans (from LLBC) into pointers and locations (from HLPL).

In effect, this amounts to losing information about the nature of the borrows, in order to only retain an aliasing graph. Continuing with the right to left intuition, Le-MutBorrow-To-Ptr states that we can collapse a pair of a mutable loan and its corresponding borrow to a location and a pointer. Notice how the value v moves from being attached to the borrow to being attached to the location; this is the crucial rule that moves from a borrow-centric view to a location-centric view.

Going back to the reborrowing example with mutable borrows from above, we have at line 2 that the LLBC state is related to the HLPL state by Le-MutBorrow-To-Ptr. The reborrow at line 3 is more interesting. We start from the LLBC state and apply Le-MutBorrow-ToPtr twice, then use Le-Merge-Locs, yielding the HLPL state.

Working in HLPL$^+$. Naturally, reasoning about $\le$ (in order to establish the forward simulation, as we do in the next paragraph) is conducted via reasoning by induction. Specifically, we do the proof by induction on the evaluation derivation, then in each sub-case do an induction on $\le$. This is the essence of our proof technique, which emphasizes local and pointwise reasoning rather than global invariants. In particular, we took great care to define the rules of $\le$ either as *local* transformations (by defining them with states with holes), or as *pointwise* transformations (Le-Merge-Locs).

As we explained earlier (§2), this leads us to reason about hybrid states that contain both loans and borrows (like LLBC states) as well as locations and pointers (like HLPL states). We call such states HLPL$^+$ states, and per §2 set out to give an operational semantics to HLPL+. Since HLPL$^+$ shares the same syntax as HLPL and LLBC, it suffices to take the union of the rules from HLPL and LLBC, *adding* HLPL$^+$-specific rules (Figure 6), and *excluding* the rules marked **(LLBC only)**. The LLBC-only rules introduce new loans and borrows; by excluding those, we get that HLPL is a stable subset of HLPL$^+$ (Definition 2.3).

The HLPL$^+$ rules are in Figure 6. Their purpose is to replace rules that would produce new borrows (E-MutBorrow, E-SharedBorrow) with HLPL+-E-Pointer, which directly reduces to a pointer. In a similar vein, R-Deref-Ptr-SharedLoan deals with a hybrid state where the state still contains a loan, but the value being reduced is an HLPL pointer, not a borrow. Finally, HLPL+-E-Assign shows

HLPL+-E-Pointer

$$\frac{\vdash \Omega(p) \overset{\text{imm,mut}}{\Rightarrow} v \qquad \text{no outer loan}^m \in v \qquad \vdash \Omega(p) \leftarrow v' \overset{\text{imm,mut}}{\Rightarrow} \Omega' \qquad v' = \begin{cases} v & \text{if } v = \text{loc}\,\ell\,v'' \\ v & \text{if } v = \text{loan}^s\,\ell\,v'' \\ \text{loc}\,\ell\,v & \ell \text{ fresh other.} \end{cases}}{\Omega \vdash \{\&p, \&\text{mut}\,p\} \Downarrow (\text{ptr}\,\ell, \Omega')}$$

HLPL+-E-Assign

$$\frac{\Omega \vdash rv \Downarrow (v, \Omega') \qquad \vdash \Omega'(p) \overset{\text{mut}}{\Rightarrow} v_p \qquad v_p \text{ has no outer loan}^{s,m}, \text{ no outer loc} \qquad \vdash \Omega'(p) \leftarrow v \overset{\text{mut}}{\Rightarrow} \Omega'' \qquad \Omega''' = \Omega'', \_ \rightarrow v_p}{\Omega \vdash p := rv \Downarrow ((), \Omega''')}$$

R-Deref-Ptr-SharedLoan

$$\frac{\text{loan}^s\,\ell\,v \in \Omega \qquad \Omega \vdash P(\text{loan}^s\,\ell\,v) \overset{\text{imm}}{\Rightarrow} v'}{\Omega \vdash P(*(\text{ptr}\,\ell)) \overset{\text{imm}}{\Rightarrow} v'}$$

Fig. 6. Selected Additional Rules for HLPL$^+$

how to add extra preconditions to extend the "no outer loan" condition[1] (required in LLBC for soundness) to a hybrid world in which there might be locations too.

Equipped with our individual rewriting rules (which form $\leq$) and a semantics in which those rules operate (HLPL$^+$, the union of HLPL and LLBC, crafted to make HLPL a stable subset), we now prove that reduction preserves $\leq$.

Theorem 3.1 (Eval-Preserves-HLPL+-Rel). *For all Ω_l, Ω_r HLPL$^+$ states, we have:*

$$\forall\, s\, r\, \Omega'_r,\ \Omega_l \leq \Omega_r \Rightarrow \Omega_r \vdash_{\text{hlpl+}} s \rightsquigarrow (r, \Omega'_r) \Rightarrow \exists\, \Omega'_l,\ \Omega_l \vdash_{\text{hlpl+}} s \rightsquigarrow (r, \Omega'_l) \wedge \Omega'_l \leq \Omega'_r$$

The proof is in the long version [Ho et al. 2024b]; it consists in a nested case analysis; first, for each reduction step; then, for each $\leq$ step. We use the fact that if two states are related by one of the $\leq$ rules, then the structure enforced by this rule is generally preserved after the evaluation.

From HLPL$^+$ to HLPL. HLPL$^+$ is merely a proof device; our ultimate goal is to relate LLBC to HLPL, not HLPL$^+$. Because we excluded (above) from HLPL$^+$ those rules that might create new loans or borrows, we trivially have the fact that the semantics of HLPL and HLPL$^+$ coincide on HLPL states (i.e., HLPL$^+$ states that don't contain loans or borrows, and thus belong to the HLPL subset of HLPL$^+$); that is, that HLPL is a stable subset of HLPL$^+$ in the sense of Definition 2.3. From this we deduce that there is a forward relation from HLPL$^+$ to HLPL:

Theorem 3.2 (Forward Relation for HLPL$^+$ and HLPL). *For Ω_l HLPL state, Ω_r HLPL$^+$ state:*

$$\forall\, s\, r\, \Omega'_r,\ \Omega_l \leq \Omega_r \Rightarrow \Omega_r \vdash_{\text{hlpl+}} s \rightsquigarrow (r, \Omega'_r) \Rightarrow \exists\, \Omega'_l,\ \Omega_l \vdash_{\text{hlpl}} s \rightsquigarrow (r, \Omega'_l) \wedge \Omega'_l \leq \Omega'_r$$

From LLBC to HLPL$^+$. We have just shown that the lower bound, Ω_l, remains in HLPL (by virtue of stability, Definition 2.3), and therefore $\leq$ can preserve the relation between HLPL and HLPL$^+$. It now remains to show the link between HLPL$^+$ and LLBC, i.e., the upper bound, Ω_h, reduces in LLBC in a way that preserves $\leq$.

Because HLPL$^+$ excludes several rules from LLBC (remember that some rules are marked **(LLBC only)**), LLBC is not a subset of HLPL$^+$, which prevents us from deriving that result instantly. Instead, we remark that we can still reconstruct the missing LLBC semantics using well-chosen combinations of evaluation rules from HLPL$^+$ and refinement rules from $\leq$. For instance, E-MutBorrow states that evaluating &mut p in LLBC leads to a state with a mutable loan and a mutable borrow. We can build the same state by using the HLPL$^+$ rule E-Pointer to introduce a pointer and a location, then

[1] An outer loan is a loan which is not itself inside a borrow; $\text{loan}^m\,\ell_0$ contains one, while $\text{borrow}^m\,\ell_1\ (\text{loan}^m\,\ell_0)$ doesn't.

by using Le-MutBorrow-To-Ptr to transform this state into a *related* state (in the sense of $\leq$), by converting this pointer and this location to a borrow and a loan. This means that a reduction in LLBC can always be completed to correspond to a reduction in HLPL$^+$, which allows us to prove the following theorem stating that, given an LLBC state Ω, evaluating a statement following the semantics of LLBC leads to a state *in relation* with the state resulting from the HLPL$^+$ semantics.

Theorem 3.3 (Eval-LLBC-Preserve-Rel). *For all Ω LLBC state we have:*

$$\forall\ s\ r\ \Omega_r,\ \Omega \vdash_{\mathsf{llbc}}\ s \rightsquigarrow (r,\ \Omega_r) \Rightarrow \exists\ \Omega_l,\ \Omega \vdash_{\mathsf{hlpl+}}\ s \rightsquigarrow (r,\ \Omega_l) \wedge \Omega_l \leq \Omega_r$$

Putting Eval-Preserves-HLPL-HLPL+-Rel and Eval-LLbc-Preserves-Rel together and using the transitivity of $\leq$ we finally get the preservation theorem we were aiming at: LLBC is in forward simulation with HLPL (Eval-Preserves-HLPL-LLBC-Rel).

Form of our theorems. We take great care to start from *any* initial states Ω_l and Ω_r rather than requiring a program execution with a closed term and with an empty state (i.e., a "main" function). The reason is, as mentioned before, we see the LLBC$^\#$ execution as borrow-checking, which we aim to perform modularly at the function-level granularity. This design for our theorems will later on allow us, as we connect LLBC$^\#$ all the way down to PL, to reason about the execution of a PL function that starts in an initial state *compatible with the lifetime signature* of the function in LLBC (§4.4). Should we wish to do so, and using the fact that empty states are in relation with each other, we can specialize our main theorem to closed programs executing in the empty state.

3.3 The Pointer Language (PL)

So far, we have only proven the central arrow of Figure 1. We now describe the simulation between HLPL and PL; we do so briefly, since the techniques are standard, and do not leverage our new proof methodology. The full proof is in the long version [Ho et al. 2024b]. PL uses an explicit heap adapted from the CompCert memory model; this is similar to RustBelt, except we have no extra state to account for concurrency. As before, we retain the same syntax of programs; this is why we can, ultimately, conclude that LLBC is a sound restriction over the execution of PL programs.

For the proof of simulation between PL and HLPL, we have no choice but to introduce a global map from location identifiers to concrete addresses, along with an explicit notion of heap. In short, we adopt the traditional global-invariant style of proof; this is the only arrow from Figure 1 where we cannot apply our methodology. However, as we took care to design HLPL so that its pointers leave values in place, the structure of HLPL states and PL states is very close; as a consequence the proof is technical, but quite straightforward. It crucially leverages the fact that the rules of HLPL were carefully crafted so that it is not possible to move a location (see the premises of HLPL-E-Move or HLPL-E-Assign for instance), as it would break the relation between the HLPL locations and the PL addresses. Moving up the hierarchy of languages, this is the reason why we forbid moving *outer* loans in the LLBC rules, that is loans which are not inside a borrow (see E-Assign in particular). We elide the exact statements of the theorems, which can be found in the long version.

3.4 Divergence and Step-Indexed Semantics

The forward simulation theorem between PL and LLBC relates terminating executions. Anticipating on the next section where we will consider LLBC$^\#$ executions as borrow-checking certificates, this gives us in particular that the existence of safe executions for LLBC implies that the PL executions are safe. Unfortunately, as LLBC is defined in a big-step fashion, one problem arises. Big-step semantics do not allow to reason about programs that safely diverge, that is, programs that never get stuck or crash, but do not terminate: these programs cannot be distingushed from stuck programs; in both cases, no evaluation exists. This is a known problem when studying type

$$\frac{}{\Omega \vdash s \underset{0}{\rightsquigarrow} \infty}\ \textsc{E-Step-Zero} \qquad \frac{}{\Omega \vdash \mathsf{return} \underset{n+1}{\rightsquigarrow} (\mathsf{return}, \Omega)}\ \textsc{E-Step-Return}$$

$$\frac{\Omega_0 \vdash s_0 \underset{n+1}{\rightsquigarrow} ((), \Omega_1) \qquad \Omega_1 \vdash s_1 \underset{n+1}{\rightsquigarrow} res}{\Omega_0 \vdash s_0; s_1 \underset{n+1}{\rightsquigarrow} res}\ \textsc{E-Step-Seq-Unit}$$

$$\textsc{E-Step-Call}\quad \frac{\begin{array}{c} f\langle \vec{\_}, \vec{\tau} \rangle = \mathsf{fn}\ \langle \vec{\_} \rangle\ (\overrightarrow{x_i : \tau_i})\ (\overrightarrow{y_j : \tau_j})\ (x_{\mathsf{ret}} : \tau)\ \{\ s\ \} \qquad \forall\ j, \Omega^{(j)} \vdash op_j \rightarrow (v_j,\ \Omega^{(j+1)}) \\ \vdash \mathsf{push\_stack}\ \left([x_{\mathsf{ret}} \rightarrow \bot] +\!\!+ [\overrightarrow{x_j \rightarrow v_j}] +\!\!+ [\overrightarrow{y_k \rightarrow \bot}]\right) \Omega^{(m)} = \Omega_0 \qquad \Omega_0 \vdash body \underset{n}{\rightsquigarrow} res \end{array}}{\Omega^{(0)} \vdash p := f(\overrightarrow{op_j}) \underset{n+1}{\rightsquigarrow} res'}$$

$$res' = \begin{cases} (\mathsf{panic}, \Omega_1) & \text{if } res = (\mathsf{panic}, \Omega_1) \\ ((), \Omega_3) & \text{if } res = (\mathsf{return}, \Omega_1) \wedge \vdash \mathsf{pop\_stack}\ \Omega_1 = (v, \Omega_2) \wedge \Omega_2 \vdash p := v \rightsquigarrow ((), \Omega_3) \\ \infty & \text{if } res = \infty \end{cases}$$

Fig. 7. Semantics of LLBC with Step-Indexing (Selected Rules)

systems; previously proposed workarounds include defining semantics coinductively to model divergence [Leroy 2006; Nakata and Uustalu 2009].

To avoid tricky coinductive reasoning, we instead rely on step-indexed semantics. We present in Figure 7 several of the updated rules for the step-indexed LLBC. As for related work, we add a step index to the judgment which evaluates statements (e.g., E-Step-Seq). This step index can be seen as a standard notion of fuel [Owens et al. 2016]; when the index is equal to zero, i.e., the execution is out of fuel, we stop the evaluation and return the ∞ value (E-Step-Zero). Otherwise, when evaluating possibly non-terminating statements (e.g., function calls as described by E-Step-Call), we decrement the step index in the recursive evaluation (e.g., the evaluation of the function body).

Building on this semantics, we can now define the following evaluation judgment that hides the step indices, and returns either ∞ for diverging computations, or the previously seen pair of a control-flow outcome and an environment:

$$\Omega \vdash s \rightsquigarrow \infty := \forall\ n, \Omega \vdash s \underset{n}{\rightsquigarrow} \infty$$

$$\Omega \vdash s \rightsquigarrow res := \exists\ n, \Omega \vdash s \underset{n}{\rightsquigarrow} res \ \wedge\ res \neq \infty$$

We follow a similar approach to extend the PL and HLPL languages to model divergence. Adapting the proofs and theorems from §3 to the step-indexed semantics and proving Theorem 3.4 is straightforward, and we omit the complete presentation for brevity. The core idea is that evaluations for the same program have identical step indices in PL, HLPL, and LLBC: the step index only decrements when entering a function call (or a loop), which is shared across the three languages.

Theorem 3.4. *For all Ω_l PL state and Ω_h LLBC state, we have:*

$$\Omega_l \leq \Omega_h \Rightarrow \forall\ s,\ \Omega_h \vdash_{\mathsf{llbc}} s \rightsquigarrow \infty \Rightarrow \Omega_l \vdash_{\mathsf{pl}} s \rightsquigarrow \infty$$

4 LLBC# is a Sound Approximation, a.k.a., Borrow-Checker for LLBC

Building on top of LLBC, Ho and Protzenko [2022] also proposed a region-centric shape analysis, which they formalized as a symbolic (or abstract) interpreter for the LLBC semantics. In their approach, this interpreter was the backbone of a translation of Rust programs to functional models in different proof assistants, enabling the verification of safe Rust programs. However, as observed by the authors, this symbolic semantics also acted as a *borrow checker* for LLBC programs.

In this section, we aim to study the soundness of this borrow checker with respect to the LLBC semantics. Borrow checking is, conceptually, similar to type checking. A sound borrow checker for

LLBC therefore ensures that, for a given program, if the borrow checker succeeds, then the program executes safely. In our setting, the definition of the borrow checking rules is however non-standard: instead of a set of inference rules, the borrow checker is formalized using a symbolic semantics, which we dub LLBC$^\#$. Importantly, this semantics is not deterministic; its implementation relies on several heuristics to choose the right rules to apply. This is to be contrasted with the current implementation of borrow-checking in the Rust compiler: while deterministic, it relies on a mostly lexical lifetime mechanism; we claim that our approach emphasizes *semantic* rather than *syntactic* borrow-checking.

We therefore aim to establish a forward simulation from LLBC$^\#$ to LLBC, that is, that for all successful LLBC$^\#$ evaluations for a given program, there exists a related, valid execution in LLBC. By composing this property with the forward simulation proven in the previous section, we therefore obtain that if the borrow-checker (LLBC$^\#$) succeeds for a given program, then this program safely executes at the PL level. The forward simulation from LLBC$^\#$ to LLBC also allows us to strengthen the relation between PL and LLBC: by definition of the simulation, successfully borrow checking a program implies the existence of a safe LLBC evaluation, which then allows us to conclude that, because the semantics of PL are deterministic, we actually have a bisimulation between PL and LLBC for programs that pass borrow checking.

In the rest of this section, we focus on applying the proof methodology from §2 to prove that LLBC and LLBC$^\#$ admit a forward simulation. We then show how to combine this result with the determinism of PL to obtain a bisimulation for PL and LLBC, under successful borrow-checking.

4.1 Background: LLBC$^\#$

We start by providing some background on LLBC$^\#$ before delving into the proof. As before, (§3.1), we pick an example from Ho and Protzenko [2022], this time to illustrate LLBC$^\#$. The difficulty of borrow-checking this example lies in the fact that we don't know which borrow is returned by **choose**. One has to note that the assert may or may not succeed (one could show that it does with Aeneas); either way, the program remains in safe Rust.

```
1  fn choose<'a, T>(b : bool, x : &'a mut T, y : &'a mut T) -> &'a mut T { ... }
2
3  let mut x = 0; let mut y = 0; let px = &mut x; let py = &mut y;
4  // x ↦ loan^m ℓ_x,   y ↦ loan^m ℓ_y,   px ↦ borrow^m ℓ_x 0,   py ↦ borrow^m ℓ_y 0
5  let pz = choose(true, move px, move py);
6  // x ↦ loan^m ℓ_x,   y ↦ loan^m ℓ_y,   px ↦ ⊥,   py ↦ ⊥,   pz ↦ borrow^m ℓ_r σ,
7  // A(ρ) {   borrow^m ℓ_x _,   borrow^m ℓ_y _,   loan^m ℓ_r   }
8  *pz = 1;
9  // x ↦ loan^m ℓ_x,   y ↦ loan^m ℓ_y,   px ↦ ⊥,   py ↦ ⊥,   pz ↦ borrow^m ℓ_r σ',                    step 0
10 //   A(ρ) {   borrow^m ℓ_x _,   borrow^m ℓ_y _,   loan^m ℓ_r   }
11 // x ↦ loan^m ℓ_x,   y ↦ loan^m ℓ_y,   px ↦ ⊥,   py ↦ ⊥,   pz ↦ ⊥,                               step 1
12 //   A(ρ) {   borrow^m ℓ_x _,   borrow^m ℓ_y _,   σ'   }
13 // x ↦ loan^m ℓ_x,   y ↦ loan^m ℓ_y,   px ↦ ⊥,   py ↦ ⊥,   pz ↦ ⊥,                               step 2
14 //   _ ↦ borrow^m ℓ_x σ_x,   _ ↦ borrow^m ℓ_y σ_y
15 // x ↦ σ_x,   y ↦ loan^m ℓ_y,   px ↦ ⊥,   py ↦ ⊥,   pz ↦ ⊥,   _ ↦ ⊥,   _ ↦ borrow^m ℓ_y σ_y      step 3
16 assert!(x >= 0);
```

Up to line 4, LLBC$^\#$ coincides with LLBC; we have local knowledge regarding variables in scope, and symbolic execution coincides with concrete execution. Line 5 is where the action happens: the caller invokes **choose** without any knowledge of its definition; LLBC$^\#$ is a modular analysis. The only information is from the type of **choose**, which states that upon calling **choose**, the caller relinquishes ownership of the arguments (here, *px* and *py*); in exchange, the caller obtains a new borrow (here, *pz*). The region (also called lifetime) annotations, which state that the input and outputs borrow have the same lifetime **'a**, have to be understood as follows: for as long as the

output borrow, i.e., pz, is alive, we have to consider that the input borrows are alive, i.e., that x and y are borrowed and thus not accessible.

To account for this call, LLBC$^\#$ uses several mechanisms. First, pz points to a symbolic value, σ. Second, px and py now point to $\bot$, to account for the fact that they have been moved out (in practice, the Rust compiler inserts reborrows that we elide here for readability). Third, for each lifetime, a fresh region ρ is introduced together with $A(\rho)$ ("the region abstraction of ρ"), a device which encodes the relationship between the input and output values. Namely, $A(\rho)$ encodes both that the callee owns the arguments provided by the caller ($A(\rho)$ now borrows x and y through ℓ_x and ℓ_y; the borrowed values are replaced with _, because remembering them is not useful), and that the caller owns a borrow that belongs to the region of the callee ($A(\rho)$ loans out the value σ to the caller, through ℓ_r). The content of the fresh region abstraction ($A(\rho)$) and the value assigned to pz are derived from the signature of **choose** (E-Call-Symbolic). We leave inst_sig to the long version; suffices to say that, in order to instantiate a signature, one has to project the borrows inside each argument (here, ℓ_x and ℓ_y) onto the region abstraction they belong to; dually, borrows within the output (here, ℓ_r) need to be associated to loans in their corresponding region abstractions.

Symbolic values and region abstractions are the two key ingredients that turn LLBC into LLBC$^\#$, its symbolic counterpart. Symbolic values behave as expected: as we increment pz (line 8), we simply learn that pz points to a fresh symbolic value σ' (step 0). Region abstractions can be transformed according to specific rules; we show a few intuitive ones, namely: a loan inside a region abstraction can be ended like any regular loan (step 1, where we invalidate pz); a borrow inside a region abstraction cannot be directly terminated, however region abstraction themselves can be terminated provided they don't contain loans anymore (Reorg-End-Abs), handing all the borrows being held back to the caller, inside fresh (ghost) anonymous variables (step 2, where we get access back to the borrows of x and y); as the values of x and y might have been modified by **choose** or through the borrow pz, we introduce fresh symbolic values (σ_x and σ_y) when putting those borrows back in the environment. Previous LLBC rules also apply over symbolic values, meaning a borrow can be terminated and the loan originator regains the value, albeit a symbolic one (step 3, where we regain access to x by ending ℓ_x), thus allowing the assert to borrow-check. We could also regain access to y by ending ℓ_y; we do not do so as it is not necessary.

The rules applying to region abstractions encode the contracts enforced by function signatures; in the present case with **choose** we get that: for as long as the borrow pz is alive, x and y are borrowed and thus inaccessible; if we want to retrieve access to x or y, we have to end pz first (step 1), which gives us access back to *both* the borrows of x and y at the same time (step 2), in return allowing us to get access back to x and y (step 3), albeit with (potentially) updated values.

4.2 Simulation Relation

Considering LLBC$^\#$ as a borrow checker, we now aim to establish a property akin to type safety. As LLBC$^\#$ is defined as a semantics instead of a set of inference rules, this corresponds to a forward simulation. We assume that programs are executing in an environment $\mathcal{P}$, which consists of a set of function definitions alongside their signature. Formally, we aim to prove the following property:

Theorem 4.1. *For all states Ω and $\Omega^\#$, statement s, and $S^\#$ set of states with outcomes (i.e., pairs of a control-flow outcome and a state), we have:*

$$(\forall\ f \in \mathcal{P},\ \text{borrow\_checks}\ f) \Rightarrow \Omega \leq \Omega^\# \Rightarrow \Omega^\# \vdash_{\text{llbc\#}}\ s \rightsquigarrow S^\# \Rightarrow$$
$$(\Omega \vdash_{\text{llbc}}\ s \rightsquigarrow \infty)\ \vee\ (\exists\ \Omega_1,\ \Omega \vdash_{\text{llbc}}\ s \rightsquigarrow (\text{panic},\ \Omega_1))\ \vee$$
$$(\exists\ \Omega_1\ \Omega_1^\#\ r \in \{(), \text{return}, \text{break}\ i, \text{continue}\ i\},\ \Omega \vdash_{\text{llbc}}\ s \rightsquigarrow (r,\ \Omega_1)\ \wedge\ \Omega_1 \leq \Omega_1^\#\ \wedge\ (r,\ \Omega_1^\#) \in S^\#)$$

$$\textsc{Le-ToSymbolic}\ \frac{\text{borrow}^{s,m}, \text{loan}^{s,m}, \bot \notin v \quad \sigma \text{ fresh}}{\Omega[v] \leq \Omega[\sigma]} \qquad \textsc{Le-ToAbs}\ \frac{\vdash v \prec^{\text{to-abs}} \overrightarrow{A}}{\Omega,\ \_ \rightarrow v \leq \Omega,\ \overrightarrow{A}} \qquad \textsc{Le-MergeAbs}\ \frac{\vdash A_0 \bowtie A_1 = A}{\Omega,\ A_0,\ A_1 \leq \Omega,\ A} \qquad \textsc{MergeAbs-Union}\ \frac{}{\vdash A_0 \bowtie A_1 = A_0 \cup A_1}$$

$$\textsc{Le-MoveValue}\ \frac{\text{no outer loans in } v \quad \text{hole of } \Omega[.] \text{ not inside a shared loan or a region abstraction}}{\Omega[v] \leq \Omega[\bot],\ \_ \mapsto v} \qquad \textsc{ToAbs-Pair}\ \frac{\vdash v_l \prec^{\text{to-abs}} \overrightarrow{A}_l \qquad \vdash v_r \prec^{\text{to-abs}} \overrightarrow{A}_r}{\vdash (v_l, v_r) \prec^{\text{to-abs}} \overrightarrow{A}_l,\ \overrightarrow{A}_r}$$

$$\textsc{Le-Reborrow-MutBorrow-Abs}\ \frac{\ell_1,\ A \text{ fresh}}{\Omega[\text{borrow}^m\ \ell_0\ v] \leq \Omega[\text{borrow}^m\ \ell_1\ v],\ A\,\{\,\text{borrow}^m\ \ell_0\ \_,\ \text{loan}^m\ \ell_1\,\}} \qquad \textsc{ToAbs-MutLoan}\ \frac{A \text{ fresh}}{\vdash \text{loan}^m\ \ell \prec^{\text{to-abs}} A\{\text{loan}^m\ \ell\}}$$

$$\textsc{MergeAbs-Mut}\ \frac{\vdash A_0 \bowtie A_1 = A}{\vdash (A_0 \cup \{\text{loan}^m\ \ell\}) \bowtie (A_1 \cup \{\text{borrow}^m\ \ell\ \_\}) = A} \qquad \textsc{ToAbs-MutBorrow}\ \frac{\text{no borrows}, \bot \in v \qquad v \prec^{\text{to-abs}} \overrightarrow{A}}{\vdash \text{borrow}^m\ \ell\ v \prec^{\text{to-abs}} (\cup\overrightarrow{A}) \cup \{\text{borrow}^m\ \ell\ \_\}}$$

Fig. 8. The Relation $\leq$ about LLBC$^+$ States (Selected Rules)

This property should be understood as follows. We assume that all functions appearing in the environment have been borrow-checked to match their signature, as represented by the predicate borrow_checks f; in practice, this can be done independently for each function, relying on the modularity of borrow-checking (see §4.3 below). Then for all states Ω in LLBC, $\Omega^\#$ in LLBC$^\#$ initially in relation, for all evaluations of a program s starting from $\Omega^\#$ and returning a set of abstract states $S^\#$, there exists a related execution of s in LLBC that either diverges, panics (panicking is safe), or yields a result related to one of the states in $S^\#$. We note that, as in our case empty environments are trivially in relation, we get the standard typing result, that is: given some entry point to the program, say a `fn main()` function, an evaluation of the program starting in the empty state is safe.

Local Transformations. Similarly to the previous section, this proof will rely on the methodology outlined in §2. We describe below its different components. The first step in our methodology is to define a set of local transformations, whose reflexive transitive closure will allow turning a concrete LLBC state into its abstract LLBC$^\#$ counterpart; we present selected rules in Figure 8. Similarly to how transformations between HLPL and LLBC led to the HLPL$^+$ language, transformations between LLBC and LLBC$^\#$ commonly span hybrid states that do not belong to either language. We dub this hybrid, union language LLBC$^+$, and define transformations as operating on LLBC$^+$ states. The semantics of LLBC$^+$ is almost exactly the union of LLBC and LLBC$^\#$. To ensure that LLBC remains a stable subset of LLBC$^+$, a key ingredient of our approach, we exclude E-Call-Symbolic, the only rule introducing symbolic values and region abstractions; similarly to our handling of loans and borrows in HLPL$^+$, they will instead be introduced through the state transformations. We now detail several of the rules that induce the $\leq$ relation on LLBC$^+$ states. Contrary to HLPL$^+$ where we explained the transformations from right to left, in the case of LLBC$^+$ it is more natural to go from left to right, from concrete to abstract; we adopt this convention in the rest of the section.

We designed the rules of $\leq$ so that they allow us to lose information about the state. Le-ToSymbolic is one of the simplest transformation rules. If we have a plain, concrete value (i.e., that does not contain any loans, borrows, or $\bot$), then we can forget its precise value by transforming it into a fresh symbolic value. Le-MoveValue implements a move: so long as no one else relies on v (as captured by the premises), v can be moved out into an anonymous variable; in effect, this allows us

to lose the information that the owner of v has access to it. Le-ToAbs captures the core rewriting to go from LLBC to LLBC$^{\#}$: it allows abstracting away parts of the borrow graph by moving borrows and loans associated to anonymous variables into fresh region abstractions, thus forgetting their precise relationships. To do so, it relies on $\bowtie$, which we explain below, and on the $\prec^{\text{to-abs}}$ judgement, which transforms a (possibly complex) value v into a set of fresh region abstractions $\overrightarrow{A}$.

The $\prec^{\text{to-abs}}$ judgment relies on $\cup$, which is plain set union. ToAbs-MutLoan simply transfers the loan to a fresh region abstraction. Some values, such as pairs, may contain several loan identifiers, and as such, give rise to several, independent region abstractions (ToAbs-Pair). ToAbs-MutBorrow converts the inner borrowed value into a set of region abstractions, computes their union, then adds the outer mutable borrow to the result. It is particularly useful to abstract away reborrow patterns: going back to the example of §3.1, after the reborrow (line 3) we have the state: $x \mapsto \text{loan}^m\ \ell_0,\ \_ \mapsto \text{borrow}^m\ \ell_0\ (\text{loan}^m\ \ell_1),\ px \mapsto \text{borrow}^m\ \ell_1\ 0$. We can abstract away the link between px and x by using Le-ToAbs, resulting in the state: $x \mapsto \text{loan}^m\ \ell_0,\ A\ \{\text{borrow}^m\ \ell_0\ \_, \text{loan}^m\ \ell_1\},\ px \mapsto \text{borrow}^m\ \ell_1\ 0$. The presence of A enforces the constraint that: if we want to end ℓ_0 to retrieve access to x, then we first need to invalidate px by ending ℓ_1 (by Reorg-End-Abs we can end a region abstraction only if it doesn't contain any loans); this constraint is very similar to what we had in the original state, albeit the fact that we don't know anymore that ℓ_1 is *exactly* a reborrow of ℓ_0.

The non-deterministic $\bowtie$ operator *merges* two different region abstractions using semantic criteria. When interpreting Rust programs, a region abstraction can be understood as a set of values associated to a given lifetime. Merging two region abstractions therefore corresponds to a notion of lifetime weakening: if we have two distinct lifetimes, Rust allows adding lifetime constraints to guarantee that borrows associated to both lifetimes will be ended at the same time. This pattern frequently occurs when typechecking a function body against a more restrictive lifetime signature.

A naive, but sound interpretation of merging regions A_0 and A_1 consists of taking the union of all values in both regions using rule MergeAbs-Union. Consider for instance the state: $x \mapsto \text{loan}^m\ \ell_x,\ y \mapsto \text{loan}^m\ \ell_y,\ A_0\ \{\text{borrow}^m\ \ell_x\ \_, \text{loan}^m\ \ell_p\},\ A_1\ \{\text{borrow}^m\ \ell_y\ \_\},\ p \mapsto \text{borrow}^m\ \ell_p\ \sigma$. In this state, we have to invalidate p to get access back to x, but can independently get access back to y by ending A_1 then ℓ_y. If we merge A_0 and A_1 we produce $A_2\ \{\text{borrow}^m\ \ell_x\ \_, \text{borrow}^m\ \ell_y\ \_, \text{loan}^m\ \ell_p\}$; in the new state we can no longer retrieve access to y without invalidating p.

However, we can also perform more precise transformations, for instance by removing a mutable loan and its associated borrow (MergeAbs-Mut). Intuitively, this rule allows hiding them in the internals of the region abstraction: ending the merged abstraction amounts, in the original state, to ending the first abstraction, all the borrows and loans that were hidden by means of MergeAbs-Mut, then the second abstraction; in the state resulting from the merge, we simply abstract all those steps away. For instance, we can merge two region abstractions $A_0\ \{\text{borrow}^m\ \ell_0\ \_, \text{loan}^m\ \ell_1\}$ and $A_1\ \{\text{borrow}^m\ \ell_1\ \_, \text{loan}^m\ \ell_2\}$ into $A_2\ \{\text{borrow}^m\ \ell_0\ \_, \text{loan}^m\ \ell_2\}$; this makes ℓ_1 disappear.

In practice, we devise an algorithm to judiciously apply the $\bowtie$ rules. Indeed, greedily applying MergeAbs-Union instead of leveraging MergeAbs-Mut creates an abstraction which contains both a loan and its associated borrow; we can never end such abstractions because of a cyclic dependency between ending the abstraction and ending the borrow, eventually leading the symbolic evaluation to get stuck. We emphasize that $\bowtie$ is not symmetric: MergeAbs-Mut is directed, and there is no version of it with a borrow on the left, and a loan on the right. This is necessary for soundness. Consider environments $A_0 = \{\text{borrow}^m\ \ell_2\ v,\ \text{loan}^m\ \ell_0,\ \text{borrow}^m\ \ell_1\ v_1\}$ and $A_1 = \{\text{loan}^m\ \ell_1,\ \text{borrow}^m\ \ell_0\ v_0\}$. This symbolic state features a cyclic dependency (perhaps, because of poorly chosen uses of Le-MergeAbs) – it is thus crucial, for our proof of forward simulation, to make sure that such a symbolic state remains stuck. The directionality of MergeAbs-Mut guarantees

E-Call-Symbolic **(LLBC$^\#$ only)**

$$\frac{f\langle\vec{\rho},\ \vec{\tau}\rangle = \mathsf{fn}\ \langle\vec{\rho}\rangle\ (\overrightarrow{x : \tau})\ (\overrightarrow{y : \tau'})\ (x_{\mathrm{ret}} : \tau_{\mathrm{ret}})\ \{\ s\ \} \qquad \Omega_j \vdash op_j \Downarrow (v_j,\ \Omega_{j+1}) \qquad \overrightarrow{\rho}\ \text{fresh} \qquad \overrightarrow{A_{\mathrm{sig}}(\rho)},\ v_{\mathrm{out}} = \mathrm{inst\_sig}(\Omega_n, \vec{\rho}, \vec{v}, \tau_{\mathrm{ret}}) \qquad \Omega_n,\ \overrightarrow{A_{\mathrm{sig}}(\rho)} \vdash p := v_{\mathrm{out}} \rightsquigarrow ((),\ \Omega')}{\Omega_0 \vdash p := f\langle\_,\ \vec{\tau}\rangle(\overrightarrow{op}) \rightsquigarrow ((),\ \Omega')}$$

Fig. 9. Operational Semantics for LLBC$^\#$ (Selected Rules)

$$\begin{array}{l}
\mathrm{borrow_checks}\ (\mathsf{fn}\ \langle\overrightarrow{\rho}\rangle\ (\overrightarrow{x : \tau})\ (\overrightarrow{y : \tau'})\ (x_{\mathrm{ret}} : \tau_{\mathrm{ret}})\ \{\ s\ \})\ :=\\
\quad \mathsf{let}\ \vec{v},\ \overrightarrow{A_{\mathrm{in}}(\rho)} = \mathrm{init}(\vec{\rho}, \vec{\tau})\\
\quad \mathsf{let}\ \Omega_0^\# = \overrightarrow{A_{\mathrm{in}}(\rho)},\ \overrightarrow{x \to v},\ \overrightarrow{y \to \bot},\ x_{\mathrm{ret}} \to \bot\\
\quad \mathsf{let}\ v_{\mathrm{out}},\ \overrightarrow{A_{\mathrm{sig}}(\rho)} = \mathrm{final}(\vec{\rho}, \vec{\tau}, \tau_{\mathrm{ret}})\\
\quad \mathsf{let}\ \Omega_1^\# = \overrightarrow{A_{\mathrm{sig}}(\rho)},\ \overrightarrow{x \to \bot},\ \overrightarrow{y \to \bot},\ x_{\mathrm{ret}} \to v_{\mathrm{out}}\\
\quad \exists\ S^\#,\ \Omega_0^\# \vdash_{\mathrm{llbc\#}} s \rightsquigarrow S^\# \wedge\\
\qquad \forall\ res \in S^\#,\ \exists\ \Omega^\#,\ res = (\mathrm{panic},\ \Omega^\#)\ \vee\ (res = (\mathrm{return},\ \Omega^\#)\ \wedge\ \Omega^\# \leq \Omega_1^\#)
\end{array}$$

Fig. 10. The Borrow Checking Predicate For Functions

just that: if A_0 is on the left of the $\bowtie$ we can eliminate ℓ_0 but not ℓ_1; if A_0 is on the right we can eliminate ℓ_1 but not ℓ_0; we thus rightfully prevent borrow-checking from succeeding.

Building on the transformations previously presented, we now focus on the simulation proof itself. As outlined in §2, the proof can be broken down in three steps: we need to establish a forward simulation on LLBC$^+$, prove that LLBC is a stable subset of LLBC$^+$, and conclude by reasoning on the remaining rules in LLBC$^\#$ that were excluded from LLBC$^+$.

The second point can be easily obtained by induction on an LLBC$^+$ evaluation. The first point is tedious, but straightforward and similar to the proof between HLPL and LLBC. The core idea is that the local transformations turn a state into a more abstract version, thus allowing fewer execution steps. Compared to HLPL$^+$, the novel part of the proof is the third point, which requires relating an abstract, modular execution of a function call (E-Call-Symbolic, Figure 9) to its concrete counterpart which enters the function body (E-Step-Call, Figure 7). Before doing so, we focus briefly on how exactly symbolic execution works, and show how to modularly borrow-check a function in LLBC$^\#$.

4.3 Borrow-Checking a Program

The actual setup of the symbolic execution is performed by borrow_checks (Figure 10). The init function (in the long version), given the types of the arguments $\vec{\tau}$, initializes a set of input values $\vec{v}$ by allocating fresh symbolic values and borrows, along with an initial set of region abstractions $\overrightarrow{A_{\mathrm{in}}(\rho)}$ which contain their associated loans, and materialize the signature and its lifetimes; this then serves to create the initial symbolic environment $\Omega_0^\#$, where the input parameters are initialized with $\vec{v}$, and where the remaining local variables and the special return variable are uninitialized. Symmetrically, $\Omega_1^\#$ captures the expected region abstractions upon exiting the function, and all local variables must be uninitialized except the return variable which must contain some value v_{out}; note that this value might contain borrows which refer to loans appearing in the region abstractions $\overrightarrow{A_{\mathrm{sig}}(\rho)}$, as is actually the case for **choose** below. The function then borrow-checks if executing the body s in $\Omega_0^\#$ is safe, and leads to states which either panic or are in relation with $\Omega_1^\#$.

We illustrate borrow-checking on the **choose** function, below. The init function computes an initial state following the function signature, lines 2-3. The function has no local variables, so the $\vec{y}$ from borrow_checks are absent. The return value is uninitialized (line 3). In this state, the variable

b contains a symbolic value σ_b, while x and y borrow some other symbolic values, their associated loans being placed in a region abstraction A_{in} which holds all the loans of lifetime `'a`; as **choose** has only one lifetime, the initial state holds a unique region abstraction. Importantly, and this is a minor difference with the original formalism, A_{in} also contains some borrows $\ell_x^{(0)}$ and $\ell_y^{(0)}$, whose corresponding loans are in an *unspecified* place (i.e., not in the current state). As such, this initial state is a partial state that can be composed with other partial states to form a complete state, where in particular all borrows have an associated loan; we will use this in the proof of the forward simulation, by applying framing lemmas in the same spirit as the frame rule in separation logic [O'Hearn et al. 2001]. In this context, A_{in} really acts as an abstraction barrier between the local state (the callee) and some external state (the caller); intuitively, $\ell_x^{(0)}$ should be exactly ℓ_x while $\ell_y^{(0)}$ should be exactly ℓ_y, but we abstracted this information away.

```
1  fn choose<'a, T>(b : bool, x : &'a mut T, y : &'a mut T) -> &'a mut T {
2    // A_in { borrow^m ℓ_x^(0) _, borrow^m ℓ_y^(0) _, loan^m ℓ_x, loan^m ℓ_y },
3    // b ↦ σ_b,   x ↦ borrow^m ℓ_x σ_x,   y ↦ borrow^m ℓ_y σ_y,   x_ret ↦ ⊥
4    if b { ret = move x;
5      // A_in { borrow^m ℓ_x^(0) _, borrow^m ℓ_y^(0) _loan^m ℓ_x, loan^m ℓ_y },
6      // b ↦ true,   x ↦ ⊥,   y ↦ borrow^m ℓ_y σ_y,   x_ret ↦ borrow^m ℓ_x σ_x
7      return; // ≤ A_out { borrow^m ℓ_x^(0) _, borrow^m ℓ_y^(0) _, loan^m ℓ }, b ↦ ⊥, x ↦ ⊥, y ↦ ⊥, x_ret ↦ borrow^m ℓ σ
8    } else { ret = move y; return; } }
```

Upon reaching the **return**, symbolic execution yields the state at lines 5-6, where the special return value variable *ret* now contains the borrow coming from x. We show at line 7 the target output state, computed via final: the goal is now to establish that the final environment (as computed by the symbolic execution) refines the output environment (as determined by the function signature).

We first reorganize the context by ending all the outer loans (loans which are themselves not inside borrows) which we see in local variables. We then repeatedly compare the two states while applying $\leq$ rules until they match: we move the values contained by the local variables (except *ret*) into fresh region abstractions (Le-MoveValue, Le-ToAbs), transform the values contained by the return variable *ret* into symbolic values (Le-ToSymbolic), and merge region abstractions together (Le-MergeAbs). We end up in the target state of line 7; we then do the same for the **else** branch (omitted), which allows us to conclude that **choose** satisfies its signature. In other words, **choose** borrow-checks.

Forward Simulation Between LLBC$^+$ and LLBC$^\#$. We now resume the presentation of the proof of the simulation between LLBC and LLBC$^\#$: there remains to show that we can replace E-Call with E-Call-Symbolic. There are several crucial points. First, evaluation rules are local, which means the evaluation relation is also defined for partial states in which borrows don't necessarily have an associated loan; equipped with partial states, we define and prove a framing lemma in the spirit of separation logic, which states that if state $\Omega_0^\#$ evaluates to $\Omega_1^\#$, we can compose $\Omega_0^\#$ with a disjoint frame $\Omega_f^\#$ which doesn't get modified during the evaluation. As the $\leq$ rules are also local, we define a similar framing property for the $\leq$ relation. Finally, we carefully designed E-Call, E-Call-Symbolic and the borrow_checks predicate so that: 1. we can always turn a part of the concrete input state into a more abstract partial state that is in relation with $\Omega_0^\#$; 2. applying the frame rules to the state appearing in the conclusion of borrow_checks produces the state resulting from E-Call-Symbolic. There are some other technicalities; we refer the interested reader to the long version of the paper.

4.4 Backward Simulation From LLBC to PL

The forward simulation from LLBC$^\#$ to LLBC allows us to conclude about the soundness of LLBC$^\#$ as a borrow-checker. However, it also guarantees the existence of a backward simulation between LLBC and PL for programs that successfully borrow-check; we formalize this in Theorem 4.2. This

theorem states that, assuming that all functions in program P borrow-check, and that the state Ω^{llbc} is in relation with the initial borrow-checking state $\Omega_0^\#$ for function g, then any PL execution of g starting from a related state Ω^{pl} has a related LLBC execution. Combined with the forward simulation from LLBC to PL proven in §3, this provides the main result of our paper: we have a bisimulation relation between LLBC and PL for programs that borrow-check.

Theorem 4.2 (Backward Simulation Between PL and LLBC). *For all Ω^{pl} PL state and Ω^{llbc} LLBC state, for all function $g\langle\vec{\rho}, \vec{\tau}\rangle$, we have:*

$$\text{let } \vec{v},\ \overrightarrow{A_{\text{in}}(\rho)} = \text{init}(\vec{\rho}, \vec{\tau});\ \ \text{let } \Omega_0^\# = \overrightarrow{A_{\text{in}}(\rho)},\ \overrightarrow{x \longrightarrow v},\ \overrightarrow{y \longrightarrow \bot},\ x_{\text{ret}} \longrightarrow \bot;$$

$$(\forall\ f \in \mathcal{P},\ \text{borrow\_checks}\ f) \Rightarrow \Omega^{pl} \leq \Omega^{llbc} \Rightarrow \Omega^{llbc} \leq \Omega_0^\# \Rightarrow \forall\ res,\ \Omega^{pl} \vdash g.\text{body} \rightsquigarrow res \Rightarrow$$

$$\exists\ res',\ \Omega^{llbc} \vdash g.\text{body} \rightsquigarrow res' \wedge$$

$$(res = res' = \infty)\ \vee\ (\exists\ r\ \Omega_1^{pl}\ \Omega_1^{llbc},\ res = (r,\ \Omega_1^{pl})\ \wedge\ res' = (r,\ \Omega_1^{llbc})\ \wedge\ \Omega_1^{pl} \leq \Omega_1^{llbc})$$

Building on the previous sections, the proof is straightforward: if all functions in program $\mathcal{P}$ borrow-check, so does g, which by definition means that there exists an LLBC$^\#$ evaluation of g.body starting from $\Omega_0^\#$. By forward simulation, this implies the existence of an LLBC evaluation starting from Ω^{llbc} and returning a result res_{llbc}, which in turn implies the existence of a related PL evaluation starting from Ω^{pl} and returning res_{pl}. As PL is deterministic, we derive $res = res_{pl}$ for any PL execution starting from Ω^{pl}, and conclude by exhibiting the witness $res' = res_{llbc}$.

5 Merging Abstract Environments

As we saw in the previous section, LLBC$^\#$ offers a sound, modular borrow checker for the LLBC semantics. However, its approach based on a symbolic collecting semantics struggles with scalability when considering disjunctive control flow. Consider the rule E-IfThenElse-Symbolic, which defines the LLBC$^\#$ semantics for evaluating common if-then-else constructs. This rule evaluates both branches independently, yielding sets of states with outcomes $S_0^\#$ and $S_1^\#$, and finally returns their union; each state in the resulting set is then considered as a starting point to evaluate the rest of the program. When a program contains many branching constructs, this leads to a combinatorial explosion known to symbolic execution practitioners as the path explosion problem. This issue becomes even worse when considering loops, as the number of paths to analyze can be infinite.

To circumvent this issue, one possible solution is to merge control flow paths at different program points, typically after the end of a branching statement. Doing so requires soundly merging environments or, borrowing terminology from abstract interpretation, "computing a join" [Cousot and Cousot 1977]. In this section, we show how to define such an operator for abstract, borrow-centric environments. Leveraging the forward simulation from LLBC$^\#$ to LLBC, we then prove its soundness with respect to the LLBC semantics.

5.1 Joining Environments

We present in Figure 11 representative rules for our join operator. To make the presentation easier to follow, we will rely on the following small example, which is a standard Rust pattern after desugaring. We annotate this program with the LLBC$^\#$ environments at relevant program points. Our goal is to compute the join of the environments after evaluating both branches; we show the resulting environment below, and explain in the rest of this section how it is computed.

```
// b ↦ σ0
x = 0; y = 1; px = &mut x; py = &mut y;
// b ↦ σ0, x ↦ loan^m ℓ0, y ↦ loan^m ℓ1, px ↦ borrow^m ℓ0 0, py ↦ borrow^m ℓ1 1
if b { p = move px; } // b ↦ true, x ↦ loan^m ℓ0, y ↦ loan^m ℓ1, px ↦ ⊥, py ↦ borrow^m ℓ1 1, p ↦ borrow^m ℓ0 0
```

```
else { p = move py; } // b ↦ false, x ↦ loan^m ℓ0, y ↦ loan^m ℓ1, px ↦ borrow^m ℓ0 0, py ↦ ⊥, p ↦ borrow^m ℓ1 1
// Result of the join:
// b ↦ σ1, x ↦ loan^m ℓ0, y ↦ loan^m ℓ1, px ↦ ⊥, py ↦ ⊥,
// p ↦ borrow^m ℓ2 σ, A { borrow^m ℓ0 _, borrow^m ℓ1 _, loan^m ℓ2 }
```

We start with some high-level explanations before giving a formal description of the join operation. Intuitively, our goal is to compute an environment Ω_2 which is more general than the environments Ω_0 and Ω_1 resulting from evaluating the **then** and **else** branches, respectively. Our target property is that the resulting environment is in relation with the environments we join, that is: $\Omega_{0,1} \leq \Omega_2$; we can then resume evaluation with Ω_2 instead of Ω_0 and Ω_1, and still get a forward simulation between LLBC and LLBC$^\#$.

A join is naturally computed as a pointwise operation over the variables in both environments: for each variable, we compute the join of its associated values. If we have the same value on both sides we keep it unchanged (e.g., x and y above). If the values differ but don't contain loans, borrows, or $\bot$ (e.g., b which maps to either true or false), we simply introduce a fresh symbolic value to account for the fact that both environments have access to a valid value, but we don't have more information about it. An interesting case happens when we have $\bot$ on one side but not on the other (e.g., px and py): if one environment doesn't have ownership of the value at a given place, we have to consider that the joined environment doesn't have either, and thus contains $\bot$. Generally speaking, when the permissions to access a place differ in both environments, we have to be conservative by taking their greatest lower bound; for instance, if a value is loaned in one environment, we have to consider it as loaned after the join. The last interesting case happens when borrows are involved, for instance with p which borrows either x (through ℓ_0) or y (through ℓ_1). The device which allows encoding a loss of information in the borrow graph is the region abstraction, which was initially introduced to handle function calls (§4.1), but works for joins as well. In the present case, we can encode the constraint that p borrows either x or y by introducing a fresh borrow ℓ_2 linked to ℓ_0 and ℓ_1 through the fresh region abstraction A; in the resulting environment, we get that p is a valid borrow, x and y are loaned, and recovering access to x *or* y requires invalidating p.

In practice, we perform the join in two phases. We first compute a pointwise join of the variables in both environments. As this pointwise join may produce an ill-formed environment where some borrows and loans are duplicated (see below), we need to chain it with a *collapse* phase, which removes those duplications to produce a well-formed environment. The join then collapse operation satisfies the target property; that is, if successful, it produces an environment in relation with the environments before the join (Theorem 5.1).

Formally, we write $\Omega,\ \Omega' \vdash \mathsf{join}_\Omega\ \Omega_0\ \Omega_1 \rightsquigarrow \Omega_2$ to denote that the join of environments Ω_0 and Ω_1 yields a new environment Ω_2. We define the join operator inductively on the environments. The states Ω and Ω' on the left of the turnstile correspond to the top-level environments being joined, which might differ from Ω_0 and Ω_1 inside recursive derivations. A few specific rules in our complete presentation require access to them, but they can be safely ignored in this section.

Joining Values. Joins are computed pointwise: if a variable x is present in both environments, we perform a join on its associated value (Join-Var). Value joins are formally defined using the judgement $\Omega_0,\ \Omega_1 \vdash \mathsf{join}_v\ v_0\ v_1 \Downarrow v \mid \overrightarrow{A}$. This judgment is similar to the environment join above: it states that merging values v_0 and v_1 yields a new value v, and creates a set of region abstractions $\overrightarrow{A}$ to be added to the current environment. When values are the same, for instance x and y in our example above, the join is the identity (Join-Same). When values differ but do not contain borrows, loans, or $\bot$, e.g., variable b, a fresh symbolic value is returned (Join-Symbolic).

The more interesting cases occur when borrows or loans are involved. In the example above, let us look at the variable p, which corresponds to a mutable borrow in both branches, although

associated to loan identifiers ℓ_0, ℓ_1 and values 0, 1 respectively. A naive way to join these borrows would be to create a new borrow associated to a fresh loan identifier ℓ_2 and a fresh symbolic value σ, and to constrain ℓ_2 to end whenever trying to end either ℓ_0 or ℓ_1; this can be done by creating a fresh region abstraction containing $\mathsf{borrow}^m\ \ell_0\ 0$, $\mathsf{borrow}^m\ \ell_1\ 1$, and $\mathsf{loan}^m\ \ell_2$. This attempt unfortunately does not work in practice, as it might lead to invalid states containing duplicated mutable borrows: coming back to our example, the left environment still contains $\mathsf{borrow}^m\ \ell_1\ 1$ associated to y. To fix this issue, we additionally keep track of the origin of values (Join-MutBorrows). For presentation purposes, we will denote a value v coming from the left (resp. right) environment as $\boxed{v}$ (resp. $\underline{\overline{v}}$). We follow a similar approach to join a value with $\bot$, e.g., for variables px and py (Join-Bottom-Other, Join-Other-Bottom), ultimately leading to the joined environment below.

$$b \mapsto \sigma_1, \quad x \mapsto \mathsf{loan}^m\ \ell_0, \quad y \mapsto \mathsf{loan}^m\ \ell_1,$$
$$px \mapsto \bot, \quad A_0\,\{\underline{\overline{\mathsf{borrow}^m\ \ell_0\ \_}}\}, \quad py \mapsto \bot, \quad A_1\,\{\boxed{\mathsf{borrow}^m\ \ell_1\ \_}\},$$
$$p \mapsto \mathsf{borrow}^m\ \ell_2\ \sigma, \quad A_2\,\{\boxed{\mathsf{borrow}^m\ \ell_0\ \_}, \underline{\overline{\mathsf{borrow}^m\ \ell_1\ \_}}, \mathsf{loan}^m\ \ell_2\}$$

Collapsing Environments. While keeping track of the origin of values avoids inconsistencies due to duplicated borrows, it gives rise to new environments that do not belong to LLBC$^\#$. Instead of extending the semantics, we propose a set of local transformations that gradually turns an environment with marked values (e.g., $\boxed{v}$) back into an LLBC$^\#$ state. We dub their transitive closure the *collapse* operator, which we denote $\searrow$, and show several rules in Figure 11.

Coming back to our running example, we aim to collapse the joined environment to remove all markers. To do so, we first apply Collapse-Merge-Abs twice to merge abstractions A_0, A_1, and A_2, using the merge rules seen in §4.2. We finally use Collapse-Dup-MutBorrow twice to simplify $\underline{\overline{\mathsf{borrow}^m\ \ell_0\ \_}}$ and $\boxed{\mathsf{borrow}^m\ \ell_0\ \_}$ into $\mathsf{borrow}^m\ \ell_0\ \_$, then $\boxed{\mathsf{borrow}^m\ \ell_1\ \_}$ and $\underline{\overline{\mathsf{borrow}^m\ \ell_1\ \_}}$ into $\mathsf{borrow}^m\ \ell_1\ \_$ to obtain the following LLBC$^\#$ environment.

$$b \mapsto \sigma_1, \quad x \mapsto \mathsf{loan}^m\ \ell_0, \quad y \mapsto \mathsf{loan}^m\ \ell_1, \quad px \mapsto \bot, \quad py \mapsto \bot,$$
$$p \mapsto \mathsf{borrow}^m\ \ell_2\ \sigma, \quad A_3\,\{\mathsf{borrow}^m\ \ell_0\ \_, \mathsf{borrow}^m\ \ell_1\ \_, \mathsf{loan}^m\ \ell_2\}$$

Soundness. We now set our sights on proving the soundness of the join and collapse operators. Formally, we aim to prove the following theorem, which states that for any LLBC$^\#$ states Ω_l, Ω_r, if the composition of join and collapse yields an LLBC$^\#$ state Ω_c, that is, a state with no marked values, then this state is related to both Ω_l and Ω_r. We can then resume the evaluation with the joined state, instead of the set of states resulting from the two branches.

Theorem 5.1 (Join-Collapse-Le). *For all Ω_l, Ω_r, Ω_j, Ω_c we have:*

$$\Omega_l,\ \Omega_r \vdash \mathsf{join}_\Omega\ \Omega_l\ \Omega_r \rightsquigarrow \Omega_j \Rightarrow \vdash \Omega_j \searrow \Omega_c \Rightarrow \textit{no marked value in } \Omega_c \Rightarrow \forall\ m \in \{l, r\},\ \Omega_m \leq \Omega_c$$

The proof relies on an induction on the reductions for join and collapse. To explain the intuition behind the proof, we will consider state projections keeping marked values from only one side. For presentation purposes, we will focus on the environment on the left. The state projection is then defined as discarding all values from the right side (e.g., $\underline{\overline{v}}$) and removing the left markers (e.g., replacing $\boxed{v}$ by v). Then, the intuition is that the left environment will always be in relation with the left projection of the join. Using this notion of projection, there is almost a one-to-one mapping between the rules defining join and collapse on one side, and the rules defining the $\leq$ relation on the other. For instance, applying the left projection to the conclusion of Join-MutBorrows yields exactly Le-Reborrow-MutBorrow-Abs.

One important point of this soundness theorem is that it requires that the result of collapse does not contain any marked value; in our implementation of these rules, we rely on several heuristics to find a derivation satisfying this condition, and raise an error when unsuccessful. As we will see in §6, while incomplete, these heuristics are sufficient to cover a large subset of Rust.

5.2 Extending Support to Loops

The join and collapse operator we presented allows us to handle disjunctive control flow without demultiplying the number of states to consider. While the presentation focused on simple branching, i.e., merging two environments after an if-then-else statement, this approach also applies to more complex constructs, such as loops. As an example, consider the toy program below which iteratively increments variable x through its mutable borrow p. While this program is purposedly simple, its reborrow inside a loop is a characteristic pattern when iterating over recursive data structures in Rust; we provide a more realistic example in the long version of the paper [Ho et al. 2024b].

```
x  = 0; p = &mut x; // x ↦ loan^m ℓ0, p ↦ borrow^m ℓ0 0
loop {
  p = &mut (*p); // x ↦ loan^m ℓ0, _ ↦ borrow^m ℓ0 (loan^m ℓ1), p ↦ borrow^m ℓ1 0
  *p += 1;       // x ↦ loan^m ℓ0, _ ↦ borrow^m ℓ0 (loan^m ℓ1), p ↦ borrow^m ℓ1 1
  continue; }
```

To borrow-check this loop, our goal is to derive a state general enough to encompass all possible states upon entering the loop; this is known as computing a fixpoint in program analysis. Unfortunately, our example shows that using state inclusion to determine if a given state is a fixpoint is not sufficient. Starting from a hypothetical fixpoint, executing the loop body will create a new loan identifier and an anonymous mapping during the reborrow, which after joining with the initial environment will yield new region abstractions. Our main observation is that it is actually sufficient to consider a fixpoint up to loan and region identifier substitution. The intuition is that, seeing LLBC$^\#$ states as shape graphs [Chang and Rival 2008; Laviron et al. 2010], the substitutions correspond to graph isomorphisms, which preserve the semantics of a program.

Additionally, compared to arbitrary joins, loops follow a generic pattern. The loop body possibly introduces fresh borrows and anonymous mappings, before being merged with an earlier snapshot. To compute a fixpoint according to the LLBC$^\#$ semantics, we therefore rely on several heuristics. First, we convert all fresh anonymous mappings to region abstractions using Le-ToAbs. Second, we flatten the environment by merging freshly introduced region abstractions that contain related borrows and loans, that is, values associated to the same loan identifier. We finally apply the join operator presented in the previous section to the resulting state and the initial environment, and repeat this approach until we find a fixpoint. In our example, we get the environment below, which remains the same after executing the loop body, up to substitution of ℓ_2 and A_2 by fresh identifiers.

$$x \mapsto \mathsf{loan}^m\ \ell_0,\ A_2\ \{\ \mathsf{borrow}^m\ \ell_0\ \_,\ \mathsf{loan}^m\ \ell_2\ \},\ p \mapsto \mathsf{borrow}^m\ \ell_2\ \sigma$$

Note that, while conceptually similar to widening operators in abstract interpretation, we do not claim that our approach terminates. Our implementation of LLBC$^\#$ fails if the fixpoint computation does not converge after a fixed number of steps. In practice, we however observe that these heuristics are sufficient to handle a wide range of examples, as we demonstrate in the next section; in those examples the computation actually converges in one step.

6 Implementation and Evaluation

In the previous sections, we focused on establishing simulations to demonstrate the soundness of LLBC$^\#$ with respect to PL, a low-level, heap-manipulating semantics. Our theorems establish that, for any execution in LLBC$^\#$, there exists a related execution in LLBC, and hence in PL by composing simulations. As PL is deterministic, it gives us that all PL states in relation with the initial LLBC$^\#$ state safely execute. One key question remains however: seeing LLBC$^\#$ as a borrow-checker for LLBC, are we able to construct LLBC$^\#$ derivations in order to apply our theoretical results?

To this end, we implement an LLBC$^\#$ interpreter, and evalute it on a set of Rust programs. We start from Aeneas' symbolic execution [Ho and Protzenko 2022], which implements the LLBC$^\#$ semantics

$$\textsc{Join-Same}\ \frac{}{\Omega_0,\ \Omega_1 \vdash \mathsf{join}_v\ v\ v \Downarrow v \mid \emptyset}$$

$$\textsc{Join-Symbolic}\ \frac{\text{no borrows, loans, } \bot \in v_0,\ v_1 \qquad \sigma \text{ fresh}}{\Omega_0,\ \Omega_1 \vdash \mathsf{join}_v\ v_0\ v_1 \Downarrow \sigma \mid \emptyset}$$

$$\textsc{Join-Bottom-Other}\ \frac{\text{no outer loan} \in v \qquad \vdash v \prec^{\text{to-abs}} \overrightarrow{A}}{\Omega_0,\ \Omega_1 \vdash \mathsf{join}_v\ \bot\ v \Downarrow \bot \mid \boxed{\overrightarrow{A}}}$$

$$\textsc{Join-MutBorrows}\ \frac{\ell_2,\ A' \text{ fresh} \qquad \Omega_0,\ \Omega_1 \vdash \mathsf{join}_v\ v_0\ v_1 \Downarrow v_2 \mid \overrightarrow{A}}{\Omega_0,\ \Omega_1 \vdash \mathsf{join}_v\ (\mathsf{borrow}^m\ \ell_0\ v_0)\ (\mathsf{borrow}^m\ \ell_1\ v_1) \Downarrow \mathsf{borrow}^m\ \ell_2\ v_2 \mid A' \{ \boxed{\mathsf{borrow}^m\ \ell_0\ \_}, \boxed{\mathsf{borrow}^m\ \ell_1\ \_}, \mathsf{loan}^m\ \ell_2 \}, \overrightarrow{A}}$$

$$\textsc{Join-Var}\ \frac{\Omega_0,\ \Omega_1 \vdash \mathsf{join}_v\ v_0\ v_1 \Downarrow v_2 \mid \overrightarrow{A} \qquad \Omega_0,\ \Omega_1 \vdash \mathsf{join}_\Omega\ \Omega'_0\ \Omega'_1 \rightsquigarrow \Omega_2}{\Omega_0,\ \Omega_1 \vdash \mathsf{join}_\Omega\ (x \to v_0,\ \Omega'_0)\ (x \to v_1,\ \Omega'_1) \rightsquigarrow x \to v_2,\ \overrightarrow{A},\ \Omega_2}$$

$$\textsc{Collapse-Merge-Abs}\ \frac{\vdash A_0 \bowtie A_1 = A}{\Omega,\ A_0,\ A_1 \searrow \Omega,\ A}$$

$$\textsc{Collapse-Dup-MutBorrow}\ \frac{}{\Omega,\ A \cup \{ \boxed{\mathsf{borrow}^m\ \ell\ \_}, \boxed{\mathsf{borrow}^m\ \ell\ \_} \} \searrow \Omega,\ A \cup \{ \mathsf{borrow}^m\ \ell\ \_ \}}$$

$$\textsc{MergeAbs-Mut-MarkedLeft}\ \frac{\vdash A_0 \bowtie A_1 = A}{\vdash (A_0 \cup \{\boxed{\mathsf{loan}^m\ \ell}\}) \bowtie (A_1 \cup \{\boxed{\mathsf{borrow}^m\ \ell\ \_}\}) = A}$$

Fig. 11. Join and Collapse (Selected Rules)

presented in §4.1. We extend this interpreter with our novel support for merging environments and handling loops, which represents about 3000 lines of OCaml code.

Our test suite consists of two categories of programs. First, we implement a collection of 22 micro-tests (totalling 300 LoCs, without blanks and comments) spanning various Rust patterns based on loops, such as incrementing counters, updating values in a vector (for instance, to reinitialize it), summing the elements in an array or a slice, reversing a list in place, or retrieving a shared or mutable borrow to the n-th element of a list. Second, we evaluate our approach on the hashmap and the b-ϵ tree implemented and presented in the original Aeneas paper, consisting of 867 LoCs in total. Due to Aeneas' previous limitations, those implementations were based on recursive functions to, e.g., insert an element or resize the hashmap. Here, we update them to use more idiomatic loops for recursive data structure traversals. Leveraging our extension for joins and loops, the LLBC$^\#$ interpreter successfully borrow-checks all the examples, requiring less than 1s for the whole test suite. Interestingly, the precedent paper noted that some functions of the b-ϵ tree are not accepted by Rust's current borrow checker. Similarly, we observe that only Polonius and Aeneas are able to borrow check the updated version; we show one such (minimized) function below, with the errors given by rustc. We provide the code and the tests in the supplementary material [Ho et al. 2024a].

```
enum List<T> { Cons(T, Box<List<T>>), Nil }
// error[E0499]: cannot borrow `*ls` as mutable more than once at a time
fn get_suffix_at_x<'a>(mut ls: &'a mut List<u32>, x: u32) -> &'a mut List<u32> {
  while  let List::Cons(hd, tl) = ls {
   if *hd == x { break } // ^^ first mutable borrow occurs here
   else { ls = tl } }
  ls } // < second mutable borrow occurs here
```

7 Related Work

RustHorn. RustHorn [Matsushita et al. 2020] operates on the Calculus of Ownership and Reference (COR), a Rust-like core calculus inspired by λ_{Rust}. Given a COR program, RustHorn computes a

first-order logical encoding that can then serve as a basis for reasoning upon the COR program. The RustHorn paper provides a proof of soundness and completeness of the encoding. Specifically, the authors establish a proof of bisimulation between COR, and the execution of a custom resolution procedure (dubbed SDLC) that mimics program execution when executed over the logical encoding.

The main difference with our work is that COR already takes for granted the ownership discipline of Rust, and materializes lifetimes within its program syntax: COR features instructions for creating or ending a lifetime, and asserting that a lifetime outlives another. One consequence is that COR cannot exhibit more behaviors than SDLC, hence why the bisimulation can be established.

In contrast, our low-level language, PL, does not feature lifetimes, and as such exhibits more behaviors than LLBC. This means we can target the underlying execution model that Rust programs run on, rather than taking lifetimes as an immutable, granted analysis that we have to trust. A drawback is that we can only establish a forward simulation in the general case. Second, we do not commit to lifetimes, nor to any other particular implementation strategy of borrow-checking (e.g., Polonius [The Rust Compiler Team 2021]). Instead, we declaratively state what ownership-related operations may be performed in LLBC. It is then up to a particular borrow-checker implementation (e.g., Aeneas) to be proven sound with regards to this semantics. Notably, LLBC is not deterministic, and several executions may be valid simultaneously, e.g., by terminating borrows at different points (eagerly or lazily).

To summarize, RustHorn focuses on establishing the soundness of a logical encoding with regards to a model of the Rust semantics that assumes borrow-checking has been performed and can be trusted; here, we establish that LLBC is a correct model of execution for Rust programs, that LLBC$^{\#}$ is a valid borrow-checker for the LLBC semantics, and that LLBC$^{\#}$'s borrow-checking does indeed guarantee soundness of execution for LLBC programs.

RustBelt. In a similar fashion, RustBelt [Jung et al. 2017] is built atop λ_{Rust}, a core calculus that is already annotated with operations to create and end lifetimes. The operational semantics of λ_{Rust} itself is given by translation; the lifetime operations are assumed to be given by the Rust compiler. RustBelt focuses on a proof of semantic typing, with two chief goals: first, prove the lifetime-based type system sound with regards to the (lifetime-annotated) core language; second, use the semantic typing relation to establish that pieces of unsafe code do satisfy the type they export.

Our work differs from RustBelt in several ways. From the technical standpoint, RustBelt assumes lifetimes are given. Whether the lifetimes annotations are correct, and whether they lead to a successful execution is irrelevant – if the input program features improper lifetime annotations, this is outside of RustBelt's purview. In contrast, we attempt to determine what needs to be established from a semantic perspective in order to borrow-check a Rust program, and prove that successful borrow-checking entails safety of execution. From a goals standpoint, RustBelt attempts to understand the expected behavior of a Rust program that features unsafe blocks, using semantic typing. We do not consider unsafe code, but we intend to tackle this in future work.

RustHornBelt. The combination of RustHorn and RustBelt, RustHornBelt [Matsushita et al. 2022], aims to establish that the logical encoding of RustHorn is sound with regards to λ_{Rust}. RustHornBelt extends the methodology of RustBelt; at a very high-level, RustHornBelt proves the encoding of RustHorn, but with λ_{Rust} instead of COR, and with a machine-checked proof instead of pen-and-paper. For the same reasons as above, we see this endeavor as addressing a different problem than ours: RustHornBelt is concerned with a logical encoding that leverages lifetimes as a central piece of information, rather than giving a functional, semantic account of the borrow-checking and execution of Rust programs.

Tree Borrows. Tree Borrows [Villani 2023], the successor of Stacked Borrows [Jung et al. 2019], attempts to provide a semantics for correct borrow handling in the presence of unsafe code. This allows detecting, at run-time, violations of the contract (undefined behavior). Tree Borrows, unlike

this work, operates at runtime by tracking permissions at the level of memory cells. We operate statically, and focus on safe code, proposing a new notion of borrow-checking that we prove to be semantically sound.

Shape Analysis. Perhaps more closely connected to this work is the field of shape analysis [Berdine et al. 2007; Calcagno et al. 2009; Chang and Rival 2008; Distefano et al. 2006]. Very active in the 2000s, the goal was to design abstract domains that would be able to infer shape predicates for pointer languages. Using familiar notions of concrete and symbolic executions, the analysis would then be able to identify bugs in programs via abstract interpretation. This has led to industrial tools such as Meta (née Facebook)'s Infer [Calcagno et al. 2015].

We differ from these works in several ways. First, we operate in a much more structured language than, say, C; these works traditionally operate over pointer languages, with NULL pointers, and untagged unions (anonymous sums). In our setting, we can enforce much more discipline onto the original language, and benefit from a lot more structure than languages like C may exhibit. However, this requires reasoning about and proving the correctness of a non-standard, borrow-centric semantics, and developing novel borrow-centric shape analyses, i.e., LLBC$^{\#}$. Second, our analysis does not exactly fit within the static analysis framework, and is merely inspired by it. We exhibit similarities in the design of our join operation, which just like in shape analysis involves reconciling competing shapes, folding inductive predicates, and abstracting over differing concrete values [Chang and Rival 2008; Illous et al. 2017].

Mezzo. The Mezzo programming language [Pottier and Protzenko 2013] blends type system, ownership and shape analysis. Mezzo is equipped with a syntactic proof of type soundness [Balabonski et al. 2016], but for an operational semantics à la ML. The "merge operation" [Protzenko 2014] is akin to our join operation, and served to some degree as inspiration. It is, however, much more involved, supporting singleton types, substructural typing, and multiple types for a given variable x (top, "dynamic", "singleton x", and other degrees of folding/unfolding of a substructural type). Furthermore, the Mezzo merge algorithm is much more sophisticated: it interleaves backtracking, quantifier instantiation strategies, and folding of existential predicates. We perform none of these.

Other Rust verification tools. Creusot [Denis et al. 2021] loosely followed the RustHornBelt approach, and as such, benefits from its formalization and mechanization. As mentioned by Matsushita et al. [2022], there remain some discrepancies, namely that RustHornBelt operates on a core language (instead of surface Rust), and that Creusot does not use the predicate transformers RustHornBelt relies on. Verus [Lattuada et al. 2023] contains a pen-and-paper formalization, not about Rust itself, lifetimes, or borrow-checking, but rather about the soundness and termination of their approach to specifications relying on ghost permissions. As such, the proof for Verus answers a different question, namely, whether their design on top of existing Rust is sound. The proof remains of limited scope, only taking into account two possible lifetimes. Prusti [Astrauskas et al. 2019] translates Rust programs into Viper's core logic; the soundness of verifying Rust code then depends on the soundness of Viper, and of the translation itself. To the best of our knowledge, no formal argument exists as to the soundness of the translation.

Acknowledgements

Sidney Congard contributed to a preliminary exploration of the problem of joining environments in LLBC in 2022; the join operation introduced in the present work is a complete rewrite of this first attempt. We thank François Pottier for various advice and comments about working with simulations and step-indexing. We thank Raphaël Monat for helping us clarify several key points of abstract interpretation and shape analysis. Finally, we thank Ralf Jung for many insightful discussions about the semantics of Rust, and for suggesting to not use sets of loan identifiers to handle shared loans in the semantics of LLBC.

References

Vytautas Astrauskas, Peter Müller, Federico Poli, and Alexander J. Summers. 2019. Leveraging Rust Types for Modular Specification and Verification, In Object-Oriented Programming Systems, Languages, and Applications (OOPSLA). *Proc. ACM Program. Lang.* 3, OOPSLA, 147:1–147:30. https://doi.org/10.1145/3360573

Thibaut Balabonski, François Pottier, and Jonathan Protzenko. 2016. The design and formalization of Mezzo, a permission-based programming language. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 38, 4 (2016), 1–94.

Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, Peter W O'hearn, Thomas Wies, and Hongseok Yang. 2007. Shape analysis for composite data structures. In *Computer Aided Verification: 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007. Proceedings 19*. Springer, 178–192.

Bruno Blanchet. 2001. An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. In *14th IEEE Computer Security Foundations Workshop (CSFW-14)*. IEEE Computer Society, Cape Breton, Nova Scotia, Canada, 82–96. This paper received a **test of time award** at the CSF'23 conference.

Cristiano Calcagno, Dino Distefano, Jérémy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. 2015. Moving fast with software verification. In *Proceedings of the NASA Formal Methods Symposium (NFM)*. Springer, 3–11.

Cristiano Calcagno, Dino Distefano, Peter O'Hearn, and Hongseok Yang. 2009. Compositional shape analysis by means of bi-abduction. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 289–300.

Bor-Yuh Evan Chang and Xavier Rival. 2008. Relational inductive shape analysis. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*.

Patrick Cousot and Radhia Cousot. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM Press, New York, NY, Los Angeles, California, 238–252.

Xavier Denis, Jacques-Henri Jourdan, and Claude Marché. 2021. *The Creusot Environment for the Deductive Verification of Rust Programs*. Research Report RR-9448. Inria Saclay - Île de France. https://hal.inria.fr/hal-03526634

Xavier Denis, Jacques-Henri Jourdan, and Claude Marché. 2022. Creusot: a foundry for the deductive verification of rust programs. In *International Conference on Formal Engineering Methods (ICFEM)*. Springer, 90–105.

Dino Distefano, Peter W O'hearn, and Hongseok Yang. 2006. A local shape analysis based on separation logic. In *Tools and Algorithms for the Construction and Analysis of Systems: 12th International Conference, TACAS 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 25-April 2, 2006. Proceedings 12*. Springer, 287–302.

Son Ho, Aymeric Fromherz, and Jonathan Protzenko. 2024a. *Artifact for: Sound Borrow-Checking for Rust via Symbolic Semantics*. https://doi.org/10.5281/zenodo.11500453

Son Ho, Aymeric Fromherz, and Jonathan Protzenko. 2024b. Sound Borrow-Checking for Rust via Symbolic Semantics (Long Version). https://doi.org/10.48550/arXiv.2404.02680 arXiv:2404.02680 [cs.PL]

Son Ho and Jonathan Protzenko. 2022. Aeneas: Rust verification by functional translation. *Proceedings of the ACM on Programming Languages* 6, ICFP (Aug. 2022), 711–741. https://doi.org/10.1145/3547647

Marieke Huisman and Bart Jacobs. 2000. Java Program Verification via a Hoare Logic with Abrupt Termination, Vol. 1783. 284–303. https://doi.org/10.1007/3-540-46428-X_20

Hugo Illous, Matthieu Lemerre, and Xavier Rival. 2017. A relational shape abstract domain. In *Proceedings of the NASA Formal Methods Symposium (NFM)*. Springer, 212–229.

Ralf Jung, Hoang-Hai Dang, Jeehoon Kang, and Derek Dreyer. 2019. Stacked borrows: an aliasing model for Rust. *Proceedings of the ACM on Programming Languages* 4, POPL (2019), 1–32.

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2017. RustBelt: Securing the foundations of the Rust programming language. *Proceedings of the ACM on Programming Languages* 2, POPL (2017), 1–34.

Franziskus Kiefer and Lucas Franceschino. 2023. Introducing hax. https://hacspec.org/blog/posts/hax-v0-1/.

Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. 2023. Verus: Verifying Rust Programs using Linear Ghost Types. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*.

Vincent Laviron, Bor-Yuh Evan Chang, and Xavier Rival. 2010. Separating shape graphs. In *Proceedings of the European Symposium on Programming (ESOP)*. Springer, 387–406.

Xavier Leroy. 2006. Coinductive big-step operational semantics. In *Proceedings of the European Symposium on Programming (ESOP)*. Springer, 54–68.

Xavier Leroy. 2009. Formal verification of a realistic compiler. *Communications of the ACM (CACM)* 52, 7 (2009), 107–115.

Xavier Leroy, Andrew W. Appel, Sandrine Blazy, and Gordon Stewart. 2012. *The CompCert Memory Model, Version 2*. Research report RR-7987. INRIA. http://hal.inria.fr/hal-00703441

Yusuke Matsushita, Xavier Denis, Jacques-Henri Jourdan, and Derek Dreyer. 2022. RustHorn-Belt: A semantic foundation for functional verification of Rust programs with unsafe code. In *Proceedings of the 43rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

Yusuke Matsushita, Takeshi Tsukada, and Naoki Kobayashi. 2020. RustHorn: CHC-Based Verification for Rust Programs.. In *ESOP*. 484–514.

The Miri Team. 2021. Miri. https://github.com/rust-lang/miri/.

Peter Müller, Malter Schwerhoff, and Alexander J. Summers. 2016. Viper: A Verification Infrastructure for Permission-Based Reasoning. In *Verification, Model Checking, and Abstract Interpretation (VMCAI) (LNCS, Vol. 9583)*, B. Jobstmann and K. R. M. Leino (Eds.). Springer-Verlag, 41–62. https://doi.org/10.1007/978-3-662-49122-5_2

Keiko Nakata and Tarmo Uustalu. 2009. Trace-based coinductive operational semantics for While: big-step and small-step, relational and functional styles. In *International Conference on Theorem Proving in Higher Order Logics*. Springer, 375–390.

National Security Agency. 2022. Software Memory Safety. https://media.defense.gov/2022/Nov/10/2003112742/-1/-1/0/CSI_SOFTWARE_MEMORY_SAFETY.PDF.

Michael Norrish. 1998. C formalised in HOL. (1998).

Scott Owens, Magnus O Myreen, Ramana Kumar, and Yong Kiam Tan. 2016. Functional big-step semantics. In *Proceedings of the European Symposium on Programming*. Springer, 589–615.

Peter O'Hearn, John Reynolds, and Hongseok Yang. 2001. Local reasoning about programs that alter data structures. In *International Workshop on Computer Science Logic (CSL)*. Springer, 1–19.

François Pottier and Jonathan Protzenko. 2013. Programming with permissions in Mezzo. *ACM SIGPLAN Notices* 48, 9 (2013), 173–184.

Jonathan Protzenko. 2014. *Mezzo: a typed language for safe effectful concurrent programs*. Ph. D. Dissertation. Université Paris Diderot-Paris 7.

StackOverflow. 2023. 2023 Developer Survey. https://survey.stackoverflow.co/2023/#section-admired-and-desired-programming-scripting-and-markup-languages.

Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, et al. 2016. Dependent types and multi-monadic effects in F. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*. 256–270.

The Register. 2022. In Rust We Trust: Microsoft Azure CTO shuns C and C++. https://www.theregister.com/2022/09/20/rust_microsoft_c/.

The Rust Compiler Team. 2021. The Polonius Book. https://rust-lang.github.io/polonius/.

The Rust Compiler Team. 2024. Guide to rustc development. https://rustc-dev-guide.rust-lang.org/borrow_check/two_phase_borrows.html.

Neven Villani. 2023. Tree Borrows, a new aliasing model for Rust. https://perso.crans.org/vanille/treebor/.