Compositional Bug Detection for Internally Unsafe Libraries

A Logical Approach to Type Unsoundness

Pedro Carrott
Imperial College London, UK

Sacha-Élie Ayoun
Imperial College London, UK

Azalea Raad
Imperial College London, UK

Abstract

Rust is a modern programming language marrying the best of language design by offering fine-grained control over system resources (thus enabling efficient implementations), while simultaneously ensuring memory safety and data-race freedom. However, ensuring type safety in *internally unsafe* libraries remains an important and non-trivial challenge, where unsafe features enable low-level control but can introduce undefined behaviour (UB). Existing works on reasoning about unsafe code either use *verification* techniques to show *correctness* (i.e. safety, useful only when the unsafe code is indeed correct), or use *bug-detection* techniques to show *incorrectness* (i.e. unsafety) but fail to do so *automatically* (to avoid developer burden), *compositionally* (to support libraries without a main function), *soundly* (without false positives) and *generally* (rather than applying to specific patterns).

We close this gap by developing RUXt, a *fully automatic*, *compositional* bug detection tool for detecting UB in internally unsafe Rust libraries with a formal *inadequacy* theorem providing a *no-false-positives guarantee*. RUXt leverages the Rust type system to under-approximate the set of safe values for user-defined types and detect safety violations without requiring manual annotations by the user. By encoding well-typed values in Rust as separation logic assertions, RUXt enables compositional reasoning about types to refute unsound type signatures and detect UB. The inadequacy theorem of RUXt ensures that each UB detected by RUXt in an internally unsafe library is a true safety violation that can be triggered by a *safe* program, i.e. one that solely interacts with the safe API of the library. To this end, when RUXt identifies a UB, it additionally produces a *safe program* witnessing the UB. To demonstrate the generality and applicability of RUXt, we develop a prototype implementation in OCaml, and apply it to several case studies, showcasing its ability to detect true safety violations.

2012 ACM Subject Classification Author: Please fill in 1 or more \ccsdesc macro

Keywords and phrases Rust, bug detection, static analysis, program logics, under-approximation

1 Introduction

Programming languages such as C and C++ provide developers with fine-grained control over system resources, thereby enabling efficient implementations, albeit at the cost of *safety*. By contrast, modern languages prioritise safety through high-level abstractions, compromising on low-level control. A long-standing goal in programming language design has been to bridge this divide and create a language that delivers both safety features and control over system resources. The Rust language [22] is the most significant industrial effort toward achieving this goal to date. Rust enables zero-cost abstractions for common systems programming patterns and operates without relying on a garbage collector. Moreover, Rust offers the safety features of high-level languages, ensuring *memory safety* and *data-race freedom*, thus guaranteeing the absence of *undefined behaviours* (UB).

To do this, Rust uses an ownership-based type system, enforcing a strict discipline that prevents mutation of aliased data. This discipline, *enforced statically* at compile time, ensures the absence of UB in Rust programs. Unfortunately, the Rust type system can be overly conservative at times, making it too restrictive to implement basic low-level data structures such as `Vec`, or synchronisation mechanisms such as `Mutex`. To circumvent this, Rust provides *unsafe* blocks and functions that unlock "unsafe superpowers" to bypass the Rust compiler's safety checks. Using unsafe code, developers can implement, for instance, the aforementioned `Vec` data structure, providing a *safe encapsulation* of the unsafe code. That is, using the safe `Vec` API, developers should not be able to trigger any undefined behaviour.

Effectively, defining such an *internally unsafe* data structure *extends* the Rust type system with a new type whose safety is not monitored by the compiler. That is, while the compiler *enforces* the safety guarantees for safe code, in the case of internally-unsafe code it simply *assumes* them to hold, and the onus is on the library developer to ensure their safety.

Ensuring the above, however, is error-prone and far from straightforward, as evidenced by the subtle bugs discovered in the literature [3, 4, 13]. This underlines the need for formal methods and rigorous techniques for reasoning about unsafe code. Indeed, there has been a large body of work on reasoning about unsafe code, falling into one of two main categories (see §7 for a comprehensive discussion): (i) *verification* techniques aimed at showing the *correctness* (safety) of unsafe code [1, 6, 9, 18]; or (ii) *bug detection* techniques aimed at showing the *incorrectness* (unsafety) of unsafe code [2, 5, 29]. Specifically, the approaches in i are useful when the unsafe code under consideration is correct and they guarantee the absence of *false negatives* (i.e. missed bugs). However, they require extensive annotation efforts from the user, they simply fail when the code is indeed unsafe, and they cannot be used to identify the causes of safety violations. By contrast, the approaches in ii focus on detecting the sources of safety violations.

A bug detection technique (in category ii) should *ideally* (1) be *fully automatic* ('push-button') and require no user input or annotations – this ensures that the technique can be used out of the box, without imposing a learning curve on the user or requiring a background in formal methods; (2) only detect *true positives* (genuine safety violations) without any *false positives* (spurious reports of safety violations); (3) be applicable to a wide range of scenarios rather than specific patterns; and (4) be *compositional* (i.e. support analysing incomplete programs rather than accepting only full programs with a main function) to ensure its scalability and thus wide applicability to large codebases.

Unfortunately, however, no existing bug detection technique meets all above criteria. Specifically, Rudra [2], Kani [29] and TraitInv [5] all suffer from false positives. Moreover, while Rudra and TraitInv are automatic, Kani is *not automatic* and relies on *user-supplied symbolic tests* for checking the desired properties. It only supports first-order, non-compositional properties (i.e. not ownership properties) and cannot reason about type safety. On the other hand, Rudra and TraitInv have limited applicability: Rudra uses linter-like *heuristics* to detect three specific patterns that are *likely* to be safety violations, while TraitInv can only detect violations of simple, first-order properties in the implementations of six standard *traits*, e.g. that a partial equality function is transitive and symmetric.

Our approach: RUXt. Our main contribution bridges this gap: RUXt is a *fully automatic*, *compositional* tool for detecting UB in (internally unsafe) Rust libraries, accompanied by a formal *true-positives* theorem, guaranteeing that all reported UB are true safety violations.

To do this, we leverage the Rust type system: the type signature of a Rust function *is* indeed a safety specification/annotation provided by the developer (in contrast to type annotations in languages such as C). To this end, we draw inspiration from the semantic model of Rust types as formalised in RustBelt [15]. Specifically, we focus on RUXtBelt, a large fragment of the RustBelt model without references, enhanced with a clear account of errors necessary for bug detection. (As our focus here is bug detection rather than verification, we can safely forgo reference semantics, at the cost of missing some safety violations, without compromising on our true positives result.) Nevertheless, as we discuss in §3, even though our RUXtBelt formalism does not account for references, in our analysis we can still account for references using a simple workaround: given any function that takes a reference type (as either input or return type), we can rewrite it as a function without references in the signature. We can then use RUXt to analyse the transformed function.

As in RustBelt, in RUXtBelt we use *separation logic* [10] and represent well-typed values as resources, enabling *compositional* reasoning about types. This way, given a library that introduces a new type `T`, we can reason about it in isolation and independently of a concrete client by considering only the `T` *type space*, namely the set of all allowed (safe) values in `T`. To ensure type soundness, the `T` type space *disallows* values that lead to UB, and we can find such UB by solving a *reachability* problem: we can explore the `T` type space to show that a set of values modeled by a separation logic assertion are reachable only through *safe* code. Specifically, given a function `fn f(x: U) → T`, all values reachable by executing `f` *must* belong to the `T` type space, as long as the input belongs to the `U` type space. On the other hand, if we can show that UB is reachable using inputs that *must* belong to the `U` type space, then we have found a safety violation. By iteratively exploring and enlarging our knowledge of the type spaces through symbolically executing functions from the library, we can reliably detect type safety bugs fully automatically.

Our reachability analysis introduces the notion of *type subvariants*, namely a *subset* of the `T` type space, thus *under-approximating* the `T` type space. We obtain this under-approximation by leveraging *incorrectness separation logic* (ISL) [23, 25] to derive reachable values and infer type subvariants. We thus develop RISL (Rust incorrectness separation logic, pronounced *rizzle*), a compositional program logic for detecting UB and show that it is *sound* against our RUXtBelt model. RISL is inspired by ISL and similarly enjoys a *true-positives* theorem, meaning that all type safety bugs found by RISL are indeed true safety violations.

We then develop RUXt as a symbolic execution engine that operates on RISL assertions. RUXt automates the process of deriving RISL triples (and thus detecting UB). We prove that RUXt enjoys a *no false positives* guarantee through our *inadequacy theorem*: every UB found is a true type unsoundness in the library. That is, when RUXt detects UB, it guarantees that it is possible to construct a trace of safe calls to the library witnessing the UB. This is a direct consequence of *under-approximating* type spaces.

We develop a prototype implementation of RUXt in OCaml, and we conduct three small case studies, showing that it can detect subtle safety violations resulting from the interaction of several independently-safe functions of a library.

Outline. The rest of this article is organised as follows. We present an intuitive account of our contributions in §2. In §3 we describe the RUXt algorithm and our prototype implementation in OCaml, and provide an informal argument for why every UB detected by RUXt is indeed a safety violation that can always be triggered. In §4 we present several examples of UB found by RUXt. In §5 we describe our RISL logic for detecting UB. In §6 we present our RUXtBelt model, show that RISL is sound with respect to RUXtBelt and then present our inadequacy theorem *formally* showing that RUXt enjoys a guarantee of no false positives. Finally, we discuss related work and conclude in §7.

```rust
struct Even { val: i32 }

fn zero() → Even {
    Even { val: 0 }
}
unsafe fn succ(x: Even) → Even {
    Even { val: x.val + 1 }
}
fn next(x: Even) → Even {
    unsafe { succ(succ(x)) }
}
fn noop(x: Even) → () {
    if x.val % 2 != 0 {
        unsafe { *(0 as *mut i32) = 1 }
    }
}
```

```c
typedef struct { int val } Even;

Even zero() {
    return (Even){.val = 0};
}
Even succ(Even x) {
    return (Even){.val = x.val + 1};
}
Even next(Even x) {
    return succ(succ(x));
}
void noop(Even x) {
    if (x.val % 2 != 0) {
        *((int *)0) = 1;
    }
}
```

Figure 1 Library for the `Even` type implemented in Rust (left), alongside its C analogue (right).

Supplementary material. The proofs of all our theorems are fully mechanised in Rocq and available at: `https://anonymous.4open.science/r/ruxt-model`. A prototype implementation of RUXt is available at: `https://anonymous.4open.science/r/ruxt-code`.

2 Overview

We present an intuitive account of our contributions through an example inspired by a case study from RefinedRust [9], as shown in Fig. 1 (left). We define a new type in Rust, `Even`, as an abstraction of even integers, along with a *safe* API for interacting with values of this type. We discuss the specificities and challenges of creating safe abstractions in Rust by showing an analogous C implementation of `Even` and highlighting the differences. Using this illustrative example, we first explain *why* Rust enables the detection of a new class of bugs, thanks to its more expressive type system, and then demonstrate *how* we can leverage Rust types to detect such bugs automatically.

2.1 Type Safety Bugs

We implement the `Even` type as a **`struct`** encapsulating an `i32` value (a 32-bit integer) in the `val` field. We also implement a library for interacting with `Even` through an API comprising three *safe* functions: (1) `zero` encapsulates 0 as the base value for `Even` and returns it; given an `Even` value, (2) `next` returns a new `Even` value encapsulating the next even integer, and (3) `noop` writes to an arbitrary pointer if the encapsulated integer is odd. The `zero` and `next` functions encode constructors for well-formed `Even` values: every `Even` value they return encapsulates an integer that is guaranteed to be even. As a result, `noop` effectively performs no operations on safe inputs: the encapsulated integer is never odd by construction. Were this not the case, the function would attempt to write to an arbitrary memory address, resulting in undefined behaviour (UB). Note that `noop` is typed as a *safe* function, even though it is implemented with *unsafe* code; the **`unsafe`** block is needed as manipulating raw pointers is, in general, unsafe. In this example, however, our usage of unsafe features is safely encapsulated by ensuring that we never perform the unsafe write operation for safe inputs and thus we never observe UB.

The library implementation also includes the `succ` function which we have not yet mentioned. Given an `Even` value as input, `succ` increments the encapsulated integer and returns the result as a value of type `Even`. This behaviour clearly breaks the 'evenness' abstraction that we desire for `Even`, and thus `succ` is marked as unsafe. This unsafe function is only used as an auxiliary function in the implementation of `next`, where it is called twice within an unsafe block, ensuring that its usage is safely encapsulated. Although `succ` itself does not contain any unsafe code, it allows odd integers to be encapsulated as `Even` values, breaking the requirements for the safety of `noop`. Therefore, defining `succ` as a safe function invalidates the safety of `noop`, meaning that both functions *cannot co-exist* as safe functions for the library to be *type-safe*.

By maintaining this boundary between safe and unsafe code, one can extend the language with new types that carry rich information besides structural layouts. For instance, `noop` restricts the `val` field of `Even` values to be even, since this is the only way of ensuring `noop` is safe. In C, this kind of information is unattainable as usage of unsafe features is unrestricted and thus safety cannot be expressed programmatically. The C analogue of our Rust library in Fig. 1 (right) exemplifies this: in `noop` the unsafe write is no longer wrapped in an unsafe block. Since we cannot distinguish safe functions from unsafe ones in C, one could assume that all functions are safe by default and apply the same reasoning as in Rust, but we have already shown that `succ` and `noop` cannot be simultaneously safe. Therefore, one must assume that all C functions are unsafe by default, meaning that we cannot leverage the type of `noop` to obtain information about the `Even` type.

Effectively, type signatures in Rust, including the **`unsafe`** keyword, are *specifications* written by the developer. In particular, safe functions guarantee the *absence* of UB for safe inputs, and we can invalidate such a specification by showing the *presence* of UB for such inputs, hence showing that the library violates safety. Functions such as `noop` and `succ` (if both are declared safe) constitute such a safety violation. Naturally, we refer to these as *type safety bugs*. These bugs are not detectable in C: unlike in Rust, analysis tools can only rely on the structural layout of data and not on the implicit *meaning* that programmers attribute to their programs.

2.2 Finding Real Bugs in Rust Libraries

The state-of-the-art Pulse analyser [19, 25] classifies memory safety bugs into two categories: (1) **manifest:** a bug that occurs in all calling contexts; and (2) **latent:** a bug that occurs only in specific calling contexts. Pulse always reports manifest bugs as they are *context-independent*: if a function *always* exhibits UB, then it is not safe to include it in a library. By contrast, a latent bug is only reported if the calling context that triggers it is reachable from a program entry point, e.g. the `main` function. In other words, Pulse cannot report *context-dependent* bugs as real bugs in libraries, relying on specific codebases to report real bugs that are latent.

We show that, in Rust, context-dependent bugs can be reported as real bugs as long as we restrict the conditions on calling contexts. In particular, type safety bugs are a class of context-dependent bugs where the calling context is obtained by executing only safe code, *including calls to internally unsafe functions.* For example, by making `succ` safe, we can construct an apparently safe program that encapsulates an odd integer as an `Even` value, resulting in a context where `noop` exhibits UB. Although this type safety bug could be reported as a real bug in the Rust implementation, Pulse would merely identify it as a latent bug in the C analogue. That is, as type safety is not expressible in C, it is impossible to restrict the calling context as in Rust.

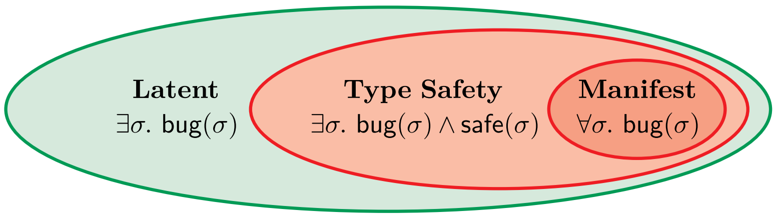
Latent
$\exists\sigma.\ \mathsf{bug}(\sigma)$
Type Safety
$\exists\sigma.\ \mathsf{bug}(\sigma) \wedge \mathsf{safe}(\sigma)$
Manifest
$\forall\sigma.\ \mathsf{bug}(\sigma)$

Figure 2 Taxonomy of bug categories. The variable σ represents a calling context, where $\mathsf{bug}(\sigma)$ expresses that UB is observable in σ and $\mathsf{safe}(\sigma)$ expresses that σ is obtained solely from safe code. Every type safety bug is a latent bug. Every manifest bug is a type safety bug: if every context triggers a bug, then the bug occurs in both safe and unsafe contexts. Manifest and type safety bugs are highlighted in red as they are always true bugs, in contrast to latent bugs.

As illustrated in Fig. 2, reasoning about type safety allows us to report more bugs: certain bugs that would be dismissed as latent in C can be reported as type safety bugs in Rust. Indeed, it is possible to detect *strictly more* real bugs than what Pulse currently reports, since every manifest bug is also a type safety bug. This opens the door to a new realm of analysis techniques for sound bug detection, which is the main contribution of our work.

2.3 Reasoning about Type Safety

Type safety, also known as *type soundness*, is a fundamental property of safe programming languages: well-typed programs (i.e. programs that type-check) will always exhibit well-defined behaviour (no UB). Conversely, one can think of a type safety bug as *type unsoundness*.

The RustBelt project [15] provided a semantic model of Rust types, mechanising and proving the type soundness of the *safe* fragments of Rust (which can be syntactically type-checked). RustBelt further provides a framework for *verifying the type soundness* of internally unsafe libraries (these are not covered by the type soundness proof), provided of course that they are indeed sound. While we focus on *detecting type unsoundness* in internally unsafe libraries, we introduce one of the main ideas behind RustBelt, from which we draw inspiration: *separation logic invariants* [10].

Safe values in RustBelt are interpreted as resources in separation logic, enabling *compositional* reasoning about types. This means that given a library that introduces a new type, we can reason about it in isolation and independently of a concrete client. To ensure type soundness is preserved when introducing a new type, RustBelt relies on user-defined *type invariants* that *over-approximate* the set of allowed values for a given type, which we refer to as *type spaces*. That is, while type spaces define the *exact* set of values inhabiting a given type, type invariants over-approximate types and thus a type invariant is a superset of its associated type space, as illustrated in Fig. 3. Hereafter, we refer to a value in a type space as a *safe value*. Type soundness guarantees that a safe program moves between safe values in type spaces. For example, since the `zero` function is safe, its output must necessarily belong to the `Even` type space. Similarly, given a value in the `Even` type space as input, the `next` function produces another value in the `Even` type space. That is, as both `zero` and `next` are safe functions, every even integer must inhabit the `Even` type space.

2.4 A Logical Approach to Type Unsoundness

We present RUXt, a *compositional*, *fully automated* analysis algorithm and tool for detecting *type safety bugs* in (internally unsafe) Rust libraries. RUXt does not require explicit definitions

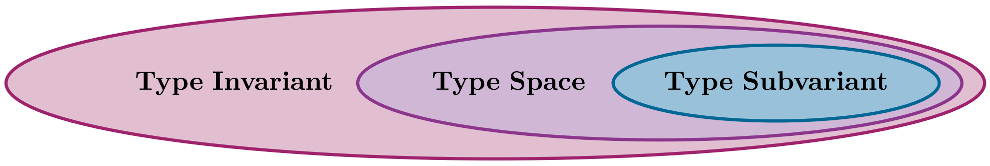

Figure 3 Type spaces represent the *exact* set of safe values of a given type; type invariants *over-approximate* type spaces; type subvariants *under-approximate* type spaces.

of type invariants from the user. Instead, it reduces the problem of detecting type unsoundness to a reachability problem. By leveraging type signatures and incorrectness separation logic [25], it automatically infers a set of safely reachable values for each type. The set of reachable values detected by our analysis constitutes an *under-approximation* of the type space, which we call a *type subvariant*. Specifically, we grow type subvariants by iteratively performing under-approximate compositional symbolic execution (à la Infer Pulse [19] or Gillian [20]) on the library functions. In turn, if the function `f(x: T)` can reach UB from an input `x` that belongs to a subvariant of `T`, then the library must be type-unsound. The relation between type spaces, type invariants and type subvariants is represented pictorially in Fig. 3.

Importantly, RUXt enjoys a guarantee of *no false positives*: every bug found corresponds to a true type unsoundness in the library. That is, when RUXt detects UB, it guarantees that it is possible to construct a trace of syntactically-safe calls to the library witnessing the UB. Furthermore, exploring type spaces is drastically more efficient than exploring the space of all possible *traces*, as we only need to find the safe *values*; that is, while many traces may construct the same safe value, we can mitigate repeated work using each safe value only once as an input to a function execution.

RUXt's symbolic execution engine is based on the set of proof rules that we define for RISL, our incorrectness separation logic for Rust, underpinned by our model of Rust (RUXtBelt), presented in §5. RISL allows us to derive an under-approximate postcondition for each safe function, given initial type subvariants for each of its inputs. In turn, we can use these postconditions to grow the type subvariants for the output types of the functions.

For instance, take the `Even` library defined in Fig. 1 (left). We demonstrate how RUXt can detect type unsoundness by describing the analysis step by step. Remember that the unsoundness arises when the `succ` function is marked as safe.

1. Initially, the subvariant for the `Even` type is empty (no safe values are known).
2. Because `next`, `noop`, and `succ` receive `Even` values as input, we cannot symbolically execute them without first inferring a non-empty subvariant for `Even`. However, `zero` does not require any input, and we can therefore symbolically execute starting from the `emp` assertion, capturing an empty heap (executing `zero` should not need any resources). We can then immediately infer that the output `Even { val: 0 }` must be a safe value, and we thus add it to our subvariant for `Even`.
3. We now have a non-empty subvariant for `Even` and can symbolically execute `next` from this subvariant, learning that the value `Even { val: 2 }` is also safe. Our newly-grown subvariant contains both `Even { val: 0 }` and `Even { val: 2 }`.
4. If we now symbolically execute `succ` from this subvariant, we learn that the values `Even { val: 1 }` and `Even { val: 3 }` are also safe.
5. Finally, if we symbolically execute `noop` from this subvariant, we can detect UB when dereferencing a null-like pointer, as the `Even` type space now includes non-even values.

As soon as we perform the last step and detect the UB, we can report the type unsoundness in the library, with a guarantee that the bug is real and can be triggered by a syntactically-safe program. Specifically:

```
fn exhibit_ub() {
    let x = zero();
    let y = succ(x);
    noop(y);
}
```

If, however, the `succ` function is correctly marked as unsafe, our analysis does not use it to grow the subvariant for `Even` and never learns that the `Even` type space contains non-even values. In this case, the library is sound, and we do not report any type unsoundness.

Note that, in this case, we would provide a notion of *fuel* to RUXt to prevent it from looping indefinitely, capping its number of iterations. In general, doing so is always sound: in the worst case we fail to detect bugs, but we would never report false positives.

3 RUXt: A Compositional Analysis for Type Safety Refutation

We present our algorithm for compositionally and automatically detecting type unsoundness in internally unsafe Rust libraries. To this end, we introduce the RUXt programming language (§3.1), formally describe type spaces and subvariants (§3.2) and present the RUXt algorithm and discuss our prototype implementation in OCaml (§3.3).

3.1 RUXt Programming Language

Our programming language, λ_{RUXt}, is inspired by those of both RustBelt (λ_{Rust} [15]) and ISL [25]. We consider a heap-manipulating *expression* language in the style of λ_{Rust}, similar to the *command* language in ISL.

$$\begin{array}{ll}
\textsc{Term} \ni t ::= v \mid x \in \textsc{Var} & \textsc{Val} \ni v ::= z \in \mathbb{Z} \mid b \in \mathbb{B} \mid l \in \textsc{Loc} \mid () \\
\textsc{Pure} \ni p ::= t \mid \ominus p \mid p_1 \oplus p_2 & \ominus ::= -_{\mathbb{Z}} \mid \neg_{\mathbb{B}} \qquad \oplus ::= +_{\mathbb{Z}} \mid \leq_{\mathbb{Z}} \\
\textsc{Expr} \ni e ::= p \mid \textsf{alloc} \mid \textsf{free}(t) \mid t_1 \leftarrow t_2 \mid {*t} \mid f(\bar{t}) \mid & \\
\qquad \textsf{assume}(t) \mid \textsf{error} \mid e_1 + e_2 \mid \textsf{let}\ x = e_1\ \textsf{in}\ e_2 &
\end{array}$$

The λ_{RUXt} (program) expressions are defined by the e grammar above and include *pure expressions*, standard heap-manipulating constructs for allocation (**alloc**), deallocation (**free**(t)), storing (heap mutation, $t_1 \leftarrow t_2$) and loading (heap lookup, $*t$), as well as function calls ($f(\bar{t})$, see below), assume expressions (**assume**(t)), explicit error expressions (**error**), non-deterministic choice ($e_1 + e_2$) and let bindings (**let** $x = e_1$ **in** e_2). Pure expressions (p) have no side effects and comprise *terms* as well as unary/binary operations on terms. A term is either a *value* or a *program variable* (x) meant to be substituted with concrete values. A value v may be an integer z, a boolean b, a heap location l or the unit value $()$. Note that as heap-manipulating and assume expressions contain terms instead of pure expressions, unary and binary operations must first be evaluated through a let binding, creating a new variable.

Other standard language constructs such as sequential composition, reference allocation, assert expressions and deterministic conditionals can be encoded using the above:

$$\begin{aligned}
e_1;\ e_2 &\triangleq \textbf{let}\ \_ = e_1\ \textbf{in}\ e_2 \\
\textbf{ref}(t) &\triangleq \textbf{let}\ x = \textbf{alloc}\ \textbf{in}\ x \leftarrow t;\ x \\
\textbf{assert}(t) &\triangleq \textbf{let}\ x = \neg_{\mathbb{B}} t\ \textbf{in}\ \textbf{assume}(t) + (\textbf{assume}(x);\ \textbf{error}) \\
\textbf{if}\ t\ \textbf{then}\ e_1\ \textbf{else}\ e_2 &\triangleq \textbf{let}\ x = \neg_{\mathbb{B}} t\ \textbf{in}\ (\textbf{assume}(t);\ e_1) + (\textbf{assume}(x);\ e_2)
\end{aligned}$$

Functions and Loops. As λ_{RUXt} supports function calls ($f(\overline{t})$), we omit loop expressions ($e^\star$), since the same behaviour can be obtained with recursive functions. Since λ_{RUXt} is not higher-order, functions must be externally provided as an *implementation context*, γ, mapping function identifiers to their concrete implementations. We write $f(\overline{x})\{e\} \in \gamma$ to denote that γ contains an implementation for f, where e is the f body and $\overline{x}$ are its arguments (i.e. free variables in e). For example, the `Even` library would correspond to an implementation context γ such that $\mathsf{dom}(\gamma) = \{\texttt{zero}, \texttt{succ}, \texttt{next}, \texttt{noop}\}$ and $\texttt{next}(x)\{\ \textbf{let}\ x = \texttt{succ}(x)\ \textbf{in}\ \texttt{succ}(x)\ \} \in \gamma$.

Type signatures. A function implementation is annotated with a type signature, recorded in a *signature context*, Δ. Signature contexts contain type signatures for *safe* functions only, as they are the only ones required to preserve type safety. Therefore, unsafe functions are excluded from signature contexts, even though they still appear in implementation contexts. We write $f(\overline{\tau}){\to}\tau \in \Delta$ to denote that Δ contains a signature for f where $\overline{\tau}$ are its input types and τ is its output type. In the case of `Even` with γ described above, we have $\mathsf{dom}(\Delta) = \mathsf{dom}(\gamma) \setminus \{\texttt{succ}\}$ and $\texttt{next}(\texttt{Even}){\to}\texttt{Even} \in \Delta$. In essence, excluding `succ` from Δ means that we cannot leverage type information from `succ` to infer anything about `Even`.

3.2 Type Spaces and Subvariants

Recall that we leverage signature contexts to construct type subvariants by inferring safe values of the types in a library. To this end, we introduce the notion of a *summary* and define a type subvariant as a set of summaries. We track the subvariants associated with each library type through a *summary context*.

Summaries. A summary, ς, is of the form $\mathsf{Summary}(v, P, e)$, where v is a value, P is a separation logic (SL) *assertion* P that constrains v, and e is a *safe program* (described below) that *witnesses* value v, i.e. executing e yields v. We define assertions as follows:

$$\begin{aligned}
\textsc{Asrt} \ni P, Q ::= &\ \llcorner \pi \lrcorner \mid \mathsf{True} \mid \mathsf{False} \mid P \wedge Q \mid P \vee Q \mid P \Rightarrow Q \mid \exists\, x.\ P \mid \\
&\ \mathsf{emp} \mid l \mapsto v \mid l \not\mapsto \mid l \mapsto_{?} \mid P * Q
\end{aligned}$$

Our assertion language is standard and includes truth, falsity, conjunction, disjunction, implication and existential quantification. A purely logical proposition[1] π may be lifted to an assertion as $\llcorner \pi \lrcorner$. Our assertions also include standard SL assertions for describing an empty heap (emp), a single-cell heap ($l \mapsto v$, comprising a single location l that holds value v) and composite heaps via the separating conjunction $*$ ($P * Q$, describing heap that can be decomposed into two parts, one described by P and another by Q). We further include the negative heap assertions from ISL, $l \not\mapsto$, and the unknown heap assertions $l \mapsto_{?}$, respectively describing a heap comprising a single location l that has been freed or is uninitialised.

[1] That is, any term definable in Rocq with type Prop, such as equality assertions or other similar relations.

For every summary $\mathsf{Summary}(v, P, e)$, e must be a *safe program*: a program e is safe if it is *syntactically* well-typed (i.e. e type-checks) and comprises solely a sequence of calls to the safe functions in the API of the current library. Furthermore, a safe program should require no initial resources to execute safely (i.e. should execute safely starting from emp).

A summary context Σ is a map from types to their subvariants (set of summaries). For example, the `zero` function always returns the value 0 in an empty heap, and thus when the summary context is given by Σ, then $\mathsf{Summary}(0, \mathsf{emp}, \texttt{zero}()) \in \Sigma(\texttt{Even})$. Note that the `Even` type is particularly simple and does not make use of the heap. In §4, we describe a more complex linked-list example where the constraining assertion is not emp.

Summary inference. We infer summaries using compositional symbolic execution [19, 20]: starting with an empty summary context Σ, we iteratively execute *safe* functions from the library to extend Σ. Given a safe function f with return type τ, every output value of f must be a safe value of τ (i.e. in the τ subvariant), and thus we can extend $\Sigma(\tau)$ to include the output value. More concretely, let f be the safe function we consider at the current iteration, where f takes n arguments of types $\tau_1 \cdots \tau_n$ and has output type τ. We then proceed as follows:

1. For each input type τ_i, we pick one summary[2] $\varsigma_i = (v_i, P_i, e_i)$ from $\Sigma(\tau_i)$. At this step we pick a function f only if we have at least one summary for each of its input types, which may not always be the case. The strategy for selecting f is immaterial for soundness of our approach, and thus we omit such details from our formalism. Nevertheless, each library typically has *constructor* functions which do not require an input, or require inputs of other types, for which our analysis would compute subvariants.
2. We combine $v_1 \cdots v_n$ into a list $\overline{v}$ to pass as arguments to f, and define a *precondition* $P \triangleq P_1 * \cdots * P_n$ for f. This yields a *safe context* for executing f: as the input values are safe by construction and f is safe, it *should* execute safely, returning a safe value of type τ. However, recall that this is not always the case when the library is internally unsafe.
3. We sequentially compose witness programs $e_1 \cdots e_n$, followed by a call to f on the associated values, resulting in the program $e \triangleq \textbf{let } x_i = e_1 \textbf{ in } \ldots \textbf{let } x_n = e_n \textbf{ in } f(\overline{x})$. Note that as the function call is syntactically well-typed and every program e_i is a safe program, the composite program e is also safe by construction. Program e will serve as a witness program for the result of executing f on these safe input values, whether it be a safe execution or not.
4. As f may be *internally* unsafe, its execution may reach an error. To distinguish safe executions from unsafe ones, we record *termination (exit) conditions* as follows:

$$\textsc{Exit} \ni \epsilon ::= \mathsf{ok}(v) \mid \mathsf{err} \mid \mathsf{miss}(l)$$

5. We analyse the resulting exit condition ϵ and proceed accordingly.
6. If ϵ is *successful*, denoted by $\mathsf{ok}(v)$, the return value v must be a safe value of τ; therefore,
   7. the symbolic execution engine constrains v by returning a postcondition Q, whereby we construct a new summary $\varsigma = \mathsf{Summary}(v, Q, e)$, where e (constructed as described above) is the safe program that witnesses ς as reachable; and
   8. we extend the subvariant of τ in Σ with ς.

[2] In our implementation, we have *symbolic* summaries, which capture more than one value at a time.

```
TryRefute($\gamma, \Delta, \Sigma$) $\to$ SummCtx + Expr
1: $(f, \overline{\varsigma}, \tau) \leftarrow$ pickCandidate$(\Delta, \Sigma)$
2: $(\overline{v}, P, \overline{e}) \leftarrow$ constructContext$(\overline{\varsigma})$
3: $e \leftarrow$ constructWitness$(\overline{e}, f)$
4: for $[\epsilon : Q] \in$ execute$(\gamma, f(\overline{v}), P)$ do
5:     match $\epsilon$ with
6:         case ok$(v)$
7:             $\varsigma \leftarrow$ Summary$(v, Q, e)$
8:             $\Sigma \leftarrow \Sigma[\tau \mapsto \varsigma, \Sigma(\tau)]$
9:         case err | miss$(-)$
10:            return e
11: return $\Sigma$
```

Summary contexts $\boxed{\gamma, \Delta \vdash \Sigma}$

R-Empty
$$\gamma, \Delta \vdash \emptyset$$

R-Infer
$$\frac{\gamma, \Delta \vdash \Sigma \quad \text{TryRefute}(\gamma, \Delta, \Sigma) = \Sigma'}{\gamma, \Delta \vdash \Sigma'}$$

Figure 4 The RUXt refutation procedure (left) and rules for inferring well-formed summary contexts (right).

In the future iterations, we can then use this summary as a safe value for functions that require an input of type τ. Note that we may re-execute previously executed functions with the new input summary Summary(v, Q, e): UB may now be reachable from this new summary, even if it was not reachable for prior safe inputs.

9. Otherwise ϵ is *unsuccessful*, given by either **(i)** err: denoting a memory safety error (e.g. performing a double free); or **(ii)** miss(l): denoting an access to a location l for which we do not own sufficient resources (e.g. an out-of-bounds access). We elaborate further on the distinction between these two cases later in §5.

10. Finally, we report the unsuccessful termination as a type unsoundness (safety violation), returning e as a witness program that triggers the unsoundness.

Note that we can soundly identify an unsuccessful exit as a true bug. Specifically, as we only consider safe functions with safe values (summaries) as input, we solely interact with the safe API of the library. As such, if a safe function does not terminate successfully on safe inputs, then the library API does not encapsulate its usage of unsafe code, resulting in UB and rendering the library type-unsound. In other words, by considering only safe contexts, we can identify latent bugs as type safety bugs thanks to the expressivity of the type system.

3.3 The RUXt Algorithm

We present RUXt, our compositional, fully automatic analysis for detecting safety violations in internally unsafe libraries. The RUXt algorithm comprises a meta-loop that calls the TryRefute procedure (Fig. 4, left) at each iteration. Each execution of TryRefute infers summaries for a safe function and updates Σ to some Σ', as discussed in §3.2, following the rules in Fig. 4 (right). This meta-loop is performed until a type unsoundness is found or a predefined limit ('fuel') is reached. Although the search is not exhaustive and some bugs may go undetected, we prioritise the guarantee of true positives by only exploring an under-approximation of the possible interactions with the library.

Refutation procedure. The TryRefute procedure is parametrised by the current summary context, Σ, along with an implementation context γ (for function implementations) and a signature context Δ (for function signature types), returning either the updated summary context or a witness program if a type unsoundness is found. Specifically, lines 1–10 in TryRefute correspond to steps 1–10 on page 10.

The pickCandidate procedure (line 1) selects a safe function f and its associated input summaries and output type (see step 1 on p. 10); that is, $f(\overline{\tau}){\to}\tau \in \Delta$ and $\forall i.\ \overline{\varsigma}_i \in \Sigma(\overline{\tau}_i)$. We then construct a safe context using constructContext (line 2) and a witness program using constructWitness (line 3), as we described previously (see steps 2 and 3 on p. 10, respectively). Given P, we then call execute (line 4) to symbolically execute the function call $f(\overline{v})$, which returns a *set* of postconditions (due to non-determinism). For each postcondition $[\epsilon : Q]$, we perform a case analysis (lines 5–10) on its exit condition ϵ, as discussed in steps 5–10 on p. 10. Finally, if no type unsoundness is found by the end of the meta-loop, the final summary context is returned (line 11).

Well-formed summary contexts. The judgement $\gamma, \Delta \vdash \Sigma$ (Fig. 4, right) denotes a *well-formed* summary context: every summary in Σ is reachable by interacting solely with the safe functions given in implementation context γ and signature context Δ. The R-Empty rule states that an empty summary context is always well-formed. The R-Infer rule allows a well-formed summary context to be extended: given a well-formed Σ, if TryRefute returns an updated summary context Σ', then Σ' is also well-formed. Since TryRefute maintains the previous summaries and only extends the summary context with new ones that are reachable from a safe function, well-formedness is preserved across iterations.

Soundness. The soundness of RUXt depends on the soundness of its underlying symbolic execution engine. Indeed, the execute procedure must derive a valid under-approximation of the values that are reachable by the program. To ensure this, we develop the RISL program logic (§5) and prove that it is sound with respect to our under-approximate semantics (§6).

The RISL soundness allows us to prove our main result, the *inadequacy theorem* (see Theorem 7): if TryRefute returns a witness program for a library, then the witness is a safe program for which UB is observable. The returned witness thus demonstrates that UB can emerge by interacting solely with the safe API of the library. In other words, we show that any violation reported by RUXt proves the existence of a true safety bug in the library.

Handling references. Our current formalisation does not handle references. Thankfully, we can apply a simple transformation to functions that manipulate mutable references in order to perform our analysis, as we describe below.

A Rust function `fn foo(x: &mut T)` which receives a mutable reference as input can be wrapped into a function `fn foo2(mut x: T) → T { inline_foo(&mut x); x }` that receives and returns a value of type `T` instead, where `inline_foo(&mut x)` denotes *inlining* the body of the `foo(&mut x)` call, carried out by our analysis. If `foo` is safe, then `foo2` *must* also be safe, and our analysis can be applied to `foo2` instead. The intuition for this wrapper function is that any safe mutating operation should always re-establish the safety of its input value, and we can use this information to grow our subvariants. Note that if a function receives several references, we can apply this transformation to each reference separately and analyse each wrapper independently to grow our subvariants. We demonstrate this transformation through a linked list example in §4.

Similarly, a function `fn bar(&mut T) → &mut U` that returns a mutable reference can be wrapped into a function that assigns an arbitrary value to the returned reference: `fn bar2(mut x: T, y: U) → T { *(inline_bar(&mut x)) = y; x }`. Again, the intuition is that assigning an arbitrary value to the reference should not break the safety of the input structure.

Shared references can be handled similarly by simply checking that reading from the reference does not trigger UB on safe inputs.

RUXt prototype implementation. We develop a prototype implementation of RUXt in OCaml. The symbolic execution engine of our prototype follows the RISL proof rules, ensuring that its inferred summaries do indeed correspond to reachable safe values. Furthermore, compared to our formalisation our implementation is enhanced in two regards.

First, we initialise our analysis with subvariants for primitive types such as integers and booleans. Second, while our formalisation describes subvariants as sets of summaries (or safe values), our prototype uses *symbolic* summaries. Instead of a (concrete) value v, a symbolic summary captures a symbolic value $\hat{v}$, which can represent an infinite number of concrete values at once. For instance, rather than individual summaries in our initial subvariant for the `int` type (i.e. $\mathsf{Summary}(0, \mathsf{emp}, 0)$, $\mathsf{Summary}(1, \mathsf{emp}, 1)$, $\ldots$), we initialise our analysis with a single symbolic summary $\mathsf{Summary}(\hat{v}, \hat{v} \in \mathbb{Z}, \hat{v})$, where $\hat{v}$ represents all integers, as constrained by the assertion. In §4, we exemplify the use of such symbolic summaries with a constructor function for the `Even` type that receives an integer as parameter.

The disadvantage of symbolic summaries is that constructing witness programs becomes slightly more complex, and our prototype does not currently generate witness programs upon finding a UB. However, SMT solvers do provide models for satisfiable constraints, and we plan to extend our prototype to generate witness programs in the future.

Finally, since the algorithm presented in Fig. 4 may loop indefinitely, our implementation receives an initial fuel parameter to limit the number of iterations.

4 Case Studies

To demonstrate the feasibility and applicability of RUXt, we run our prototype implementation on three λ_{RUXt} case studies and show that we can automatically detect type unsoundness in ill-defined libraries. First, we target the unsoundness in our running example `Even` (§2), and next we consider an extension of `Even` with a more general constructor. Finally, we define a `List` library in λ_{RUXt} with a function that causes an unsoundness. Note that as our prototype does not currently construct witness programs, we omit these from the summaries shown in this section.

4.1 Even Integer

We re-implement the functions presented in Fig. 1 within our prototype. Our implementation successfully finds the unsoundness when `succ` is marked as safe, and accurately does not when it is marked as **`unsafe`**.

Even integer extension. The `Even` example in §2 is simplified and does not fully demonstrate the benefits of using symbolic execution, as all inferred values are concrete. To illustrate the power of symbolic execution, we replace the `zero` constructor with a new function `new` that takes an integer `x` and returns it if `x` is even; otherwise, it increments `x` and returns it.

```
fn new(x: int) → Even {
  if (x % 2 = 0) { x }
  else { x + 1 }
}
```

Here, `int` is a base case: we know that the *exact* type space (i.e. neither over- nor under-approximate) of a value n of type `int` is "$n \in \mathbb{Z}$". We can use this type space as a subvariant to construct the input context for `new`. We then obtain the following subvariant for `Even` by symbolically executing `new` from that context:

$$\Sigma(\texttt{Even}) = \{\mathsf{Summary}(n, P)\} \qquad \text{with} \qquad P \triangleq n \in \mathbb{Z} \wedge (n\%2 = 0 \vee (n-1)\%2 = 1)$$

This newly obtained subvariant covers exactly the entire type space of `Even`. Using a simple SMT check, we can then prove that the concrete value returned by `zero` is included in this subvariant. This allows us to remove (or never add) the concrete value as a subvariant, thereby reducing path explosion. The same approach could be used for other base cases that are "native" to the language (e.g. the slice or `Box` types), where the type spaces are known exactly and can be formulated as simple separation logic assertions.

4.2 Linked List

We define a simple linked list library, `List`, in λ_{RUXt}. Unlike `Even`, linked lists are real-world data structures that are infamously difficult to implement soundly in Rust. Within `List` we introduce a subtle unsoundness and show that our prototype can detect it automatically. The `List` library contains the following functions:

- `fn new() → List`: creates a new empty list, a null pointer represented as the value 0;
- `fn push(l:List)→ List`: allocates a new node and adds it to the front of (input list) `l`;
- `fn drop(l: List)`: iterates through `l` and frees each node until a null pointer is found;
- `fn cycle(l: List) → List`: overrides the null pointer terminating `l` with a pointer to its first element, creating a *cycle* in the list.

In a real Rust implementation, `push` and `cycle` would receive a mutable reference to the list. As our formalism does not yet account for references, we instead receive a list and return a new list. However, note that as discussed in §3.3 (p. 12), this can be performed systematically: we transform the Rust implementation (receiving a mutable references to the list), `Rust_push`, by wrapping it in our `push` implementation as follows, where `inline_Rust_push(&mut l)` denotes inlining the body of `Rust_push(&mut l)`.

```
fn push(l: List) → List {
  inline_Rust_push(&mut l);
  l
}
```

The first three functions (`new`, `push` and `drop`) are standard and exist in the `LinkedList` implementation of Rust standard library. However, the `cycle` function is not standard and indeed introduces unsoundness: dropping a cyclic linked list will lead to a use-after-free error the second time the iteration reaches a node in the cycle[3].

RUXt can detect this unsoundness as follows.

1. At first, we can only pick the `new` function: it requires no arguments, while all other functions require an argument of type `List` for which we are yet to infer a subvariant.

[3] An alternative implementation of `drop` which would check for cycles would not suffer from this issue. However, the current implementation of the `LinkedList` structure in the Rust standard library does not check for cycles when iterating.

By executing `new`, we infer that the empty linked list – i.e. a `null` pointer – must be a safe value of type `List`. In other words, thus far we have inferred the type subvariant: $\Sigma(\texttt{List}) = \{\mathsf{Summary}(0, \mathsf{True})\}$

2. Later on, we pick `push` and execute it from our current `List` subvariant, learning a new subvariant for lists of size one: $\Sigma(\texttt{List}) = \{\mathsf{Summary}(0, \mathsf{True}), \mathsf{Summary}(l, l \mapsto 0)\}$.
3. Subsequently, we pick `cycle` and learn a new subvariant:

$$\Sigma(\texttt{List}) = \{\mathsf{Summary}(0, \mathsf{True}), \mathsf{Summary}(l, l \mapsto 0), \mathsf{Summary}(l, l \mapsto l)\}$$

4. Finally, we pick `drop`, and using our last subvariant we detect a use-after-free UB.

4.3 Key Takeaways

These examples highlights several key aspects of the kinds of unsoundness detected by RUXt:

- In Rust, functions such as `cycle` would *require* unsafe code, as safe Rust prevents the kind of aliasing required for a cyclic linked list. Importantly, reasoning about this aliasing is made substantially simpler thanks to separation logic.
- The `cycle` and `succ` functions are not inherently wrong. For instance, in the `List` case, the unsoundness arises from the interaction of `cycle` and `drop`; the library would be sound without either of these functions. Safety is, inherently, an inter-procedural property [12]. In our prototype, we also check that no unsoundness is found if either function is removed from the library, or if either is marked as unsafe.
- We do not need to explore all traces as we build subvariants. As demonstrated with the `zero` and `new` functions of the `Even` library, some subvariants might be removed from our Σ when a strictly larger subvariant is found. We believe that this compositional approach dramatically reduces path explosion issues in comparison to whole-program type-guided approaches such as Crabtree [28], described in more details in §7.

5 The RISL Program Logic

We now present RISL (Rust incorrectness separation logic), our under-approximate logic for deriving reachable states in RUXt. Our proof rules are standard and similar to those of ISL [25], with the caveat that λ_{RUXt} programs are not commands, but rather expressions that reduce to values on successful termination. Unlike ISL, RISL supports function calls and is the first separation logic to reason about missing resources (the miss exit condition).

5.1 Proof Rules

We present the RISL proof rules in Fig. 5. Intuitively, a RISL triple $\Gamma \vdash [P]\ e\ [\epsilon\colon Q]$ is interpreted similarly to ISL: every post-in the postcondition Q is reachable by executing program e starting from some pre-state in the precondition P, where ϵ is the exit condition, as described in §3.2 (p. 10). To reason about function *calls*, RISL allows reusing of derived specifications through a *specification context*, Γ, mapping each function to a list of *under-approximate* specifications. More concretely, $\left[(\overline{v})\ P \,\middle|\, \epsilon\colon Q\right] \in \Gamma(f)$ denotes that $[\epsilon\colon Q]$ is reachable by executing $f(\overline{v})$ starting from P (i.e. $\Gamma \vdash [P]\ f(\overline{v})\ [\epsilon\colon Q]$ holds).

Plain values always reduce to the value itself successfully (S-Val). Pure expressions reduce to the result of the underlying operation on the reduction values of its operands

Under-approximate triples $\boxed{\Gamma \vdash [P]\, e\, [\epsilon\colon Q]}$

S-VAL
$$\Gamma \vdash [\mathsf{emp}]\, v\, [\mathsf{ok}(v)\colon \mathsf{emp}]$$

S-ASSUME
$$\Gamma \vdash [\mathsf{emp}]\, \textbf{assume}(\mathsf{true})\, [\mathsf{ok}\colon \mathsf{emp}]$$

S-ERROR
$$\Gamma \vdash [\mathsf{emp}]\, \textbf{error}\, [\mathsf{err}\colon \mathsf{emp}]$$

S-MINUS
$$\frac{\Gamma \vdash [\mathsf{emp}]\, p\, [\mathsf{ok}(z)\colon \mathsf{emp}]}{\Gamma \vdash [\mathsf{emp}]\ -_{\mathbb{Z}}\, p\, [\mathsf{ok}(-z)\colon \mathsf{emp}]}$$

S-ADD
$$\frac{\Gamma \vdash [\mathsf{emp}]\, p_1\, [\mathsf{ok}(z_1)\colon \mathsf{emp}] \quad \Gamma \vdash [\mathsf{emp}]\, p_2\, [\mathsf{ok}(z_2)\colon \mathsf{emp}]}{\Gamma \vdash [\mathsf{emp}]\, p_1 +_{\mathbb{Z}} p_2\, [\mathsf{ok}(z_1 + z_2)\colon \mathsf{emp}]}$$

S-NOT
$$\frac{\Gamma \vdash [\mathsf{emp}]\, p\, [\mathsf{ok}(b)\colon \mathsf{emp}]}{\Gamma \vdash [\mathsf{emp}]\, \neg_{\mathbb{B}} p\, [\mathsf{ok}(\neg b)\colon \mathsf{emp}]}$$

S-LE
$$\frac{\Gamma \vdash [\mathsf{emp}]\, p_1\, [\mathsf{ok}(z_1)\colon \mathsf{emp}] \quad \Gamma \vdash [\mathsf{emp}]\, p_2\, [\mathsf{ok}(z_2)\colon \mathsf{emp}]}{\Gamma \vdash [\mathsf{emp}]\, p_1 \leq_{\mathbb{Z}} p_2\, [\mathsf{ok}(z_1 \leq z_2)\colon \mathsf{emp}]}$$

S-FREE
$$\Gamma \vdash [l \mapsto v]\, \textbf{free}(l)\, [\mathsf{ok}\colon l \not\mapsto]$$

S-STORE
$$\Gamma \vdash [l \mapsto v']\, l \leftarrow v\, [\mathsf{ok}\colon l \mapsto v]$$

S-LOAD
$$\Gamma \vdash [l \mapsto v]\, {}^*l\, [\mathsf{ok}(v)\colon l \mapsto v]$$

S-FREEUNINIT
$$\Gamma \vdash [l \mapsto ?]\, \textbf{free}(l)\, [\mathsf{ok}\colon l \not\mapsto]$$

S-STOREUNINIT
$$\Gamma \vdash [l \mapsto ?]\, l \leftarrow v\, [\mathsf{ok}\colon l \mapsto v]$$

S-LOADUNINIT
$$\Gamma \vdash [l \mapsto ?]\, {}^*l\, [\mathsf{err}\colon l \mapsto ?]$$

S-FREEFREED
$$\Gamma \vdash [l \not\mapsto]\, \textbf{free}(l)\, [\mathsf{err}\colon l \not\mapsto]$$

S-STOREFREED
$$\Gamma \vdash [l \not\mapsto]\, l \leftarrow v\, [\mathsf{err}\colon l \not\mapsto]$$

S-LOADFREED
$$\Gamma \vdash [l \not\mapsto]\, {}^*l\, [\mathsf{err}\colon l \not\mapsto]$$

S-FREEMISS
$$\Gamma \vdash [\mathsf{emp}]\, \textbf{free}(l)\, [\mathsf{miss}(l)\colon \mathsf{emp}]$$

S-STOREMISS
$$\Gamma \vdash [\mathsf{emp}]\, l \leftarrow v\, [\mathsf{miss}(l)\colon \mathsf{emp}]$$

S-LOADMISS
$$\Gamma \vdash [\mathsf{emp}]\, {}^*l\, [\mathsf{miss}(l)\colon \mathsf{emp}]$$

S-ALLOC
$$\Gamma \vdash [\mathsf{emp}]\, \textbf{alloc}\, [\mathsf{ok}(l)\colon l \mapsto ?]$$

S-CALL
$$\frac{\big[(\overline{v})\ P \ \big|\ \epsilon\colon Q\big] \in \Gamma(f)}{\Gamma \vdash [P]\, f(\overline{v})\, [\epsilon\colon Q]}$$

S-LETCUT
$$\frac{\Gamma \vdash [P]\, e_1\, [\epsilon\colon Q] \quad \epsilon \neq \mathsf{ok}(-)}{\Gamma \vdash [P]\, \textbf{let}\ x = e_1\ \textbf{in}\ e_2\, [\epsilon\colon Q]}$$

S-CHOICE
$$\frac{\Gamma \vdash [P]\, e_i\, [\epsilon\colon Q] \quad \exists\, i \in \{1, 2\}}{\Gamma \vdash [P]\, e_1 + e_2\, [\epsilon\colon Q]}$$

S-LET
$$\frac{\Gamma \vdash [P]\, e_1\, [\mathsf{ok}(v)\colon R] \quad \Gamma \vdash [R]\, e_2[v/x]\, [\epsilon\colon Q]}{\Gamma \vdash [P]\, \textbf{let}\ x = e_1\ \textbf{in}\ e_2\, [\epsilon\colon Q]}$$

S-DISJ
$$\frac{\Gamma \vdash [P_1]\, e\, [\epsilon\colon Q_1] \quad \Gamma \vdash [P_2]\, e\, [\epsilon\colon Q_2]}{\Gamma \vdash [P_1 \vee P_2]\, e\, [\epsilon\colon Q_1 \vee Q_2]}$$

S-FRAME
$$\frac{\Gamma \vdash [P]\, e\, [\epsilon\colon Q] \quad \mathsf{frameable}(\epsilon, R)}{\Gamma \vdash [P * R]\, e\, [\epsilon\colon Q * R]}$$

S-EXISTS
$$\frac{\Gamma \vdash [P]\, e\, [\epsilon\colon Q]}{\Gamma \vdash [\exists\, x.\ P]\, e\, [\epsilon\colon \exists\, x.\ Q]}$$

S-CONS
$$\frac{\Gamma' \subseteq \Gamma \quad \models (P' \Rightarrow P) \quad \Gamma' \vdash [P']\, e\, [\epsilon\colon Q'] \quad \models (Q \Rightarrow Q')}{\Gamma \vdash [P]\, e\, [\epsilon\colon Q]}$$

Figure 5 The RISL proof rules.

(S-MINUS, S-NOT, S-ADD and S-LE). Assume expressions only terminate if the term evaluates to true[4] (S-ASSUME). Error expressions always terminate erroneously (S-ERROR).

For heap-manipulating expressions, the successful cases (with ok) are analogous to their ISL counterparts (S-FREE, S-STORE, S-LOAD and S-ALLOC). As in ISL, accessing previously freed locations results in a memory safety issue and thus erroneous termination (S-FREEFREED,

[4] For readability, when programs reduce to () (unit value), we simply write ok rather than ok(()).

S-StoreFreed and S-LoadFreed). As we model uninitialised memory, compared to ISL we further have rules to account for accessing uninitialised memory. Specifically, while freeing and overwriting uninitialised locations terminates successfully (S-FreeUninit and S-StoreUninit), loading from such locations leads to UB (S-LoadUninit). We further introduce new rules to reason about missing resources (via the miss exit condition). For instance, when attempting to free a location l for which we do not own the necessary resources in the precondition, i.e. we have emp rather than $l \mapsto - \vee l \mapsto_?$, then we terminate unsuccessfully due to missing location l, denoted by the $\mathsf{miss}(l)$ exit condition in S-FreeMiss. The S-StoreMiss and S-LoadMiss describe analogous scenarios when storing to or loading from l. This miss exit condition could be surprising to readers familiar with ISL; we discuss it in more details shortly.

Let bindings correspond to sequential composition in ISL. As in ISL, S-LetCut captures the short-circuiting semantics of sequential executions: when the execution of the first expression e_1 terminates unsuccessfully, then the let expression **let** $x = e_1$ **in** e_2 also executes unsuccessfully. The S-Let cpatures the case where executing e_1 terminates successfully with value v, in which case (unlike in ISL) v is propagated to e_2 through variable substitution.

The S-Choice rule states that the states in Q are reachable when executing $e_1 + e_2$ if they are reachable from p when executing *either* branch. As in incorrectness (separation) logic, this is due to the *under-approximate* nature of RISL.

Missing resources. The miss exit condition is a novel aspect of RISL, allowing us to derive triples when the proof context does not carry sufficient resources to execute a program successfully. By contrast, in ISL, missing resources do not lead to a specific exit condition, but rather to a failure to derive a triple.

This is because unlike existing analyses based on ISL, RUXt is designed to reason about the soundness of Rust libraries, where types *capture the resources required for safe execution.* That is, if RUXt computes a subvariant assertion P for a given type `T`, then P *should* contain enough resources to execute the body of any safe function `f(x: T)`. Therefore, if we can derive a triple with a miss exit for the body of a function `f(x: T)` starting from a computed subvariant for `T`, we have shown that the resources associated with `T` are insufficient for the function to execute safely, and thus we can report it as a type safety violation.

Note that additional care is needed to ensure that triples with miss exits do not break the S-Frame rule. Specifically, when applying S-Frame to a triple with $\epsilon = \mathsf{miss}(l)$, we must ensure the framed-on resources R do not invalidate $\mathsf{miss}(l)$ by framing the missing location l. This is captured by the $\mathsf{frameable}(\epsilon, R)$ premise of the S-Frame rule, defined as follows:

$$\mathsf{frameable}(\epsilon, R) \triangleq \forall\, l.\ \epsilon = \mathsf{miss}(l) \Rightarrow \neg\, \mathsf{sat}(R \wedge (\mathsf{True} * (l \mapsto - \vee l \not\mapsto \vee\, l \mapsto_?)))$$

Note that when $\epsilon \in \{\mathsf{ok}, \mathsf{err}\}$, the $\mathsf{frameable}(\epsilon, R)$ premise is immediately satisfied, ensuring that we can *always* frame on additional resources R in such cases, as in ISL. That is, in these cases our frame rule is identical to that of ISL (which cannot account for missing resources). By contrast, in the case of $\epsilon = \mathsf{miss}(l)$, the right-hand side of the implication above ensures that R does not express ownership of l, i.e. does not contain $l \mapsto -$ or $l \not\mapsto$ or $l \mapsto_?$ (among other resources, as denoted by True).

Function calls. The S-Call rule allows specifications from the Γ context to be directly applied when they match a given function call. Rather than executing $f(\overline{v})$ directly, one can consult the specifications of f in Γ and check if there exists a specification $\left[(\overline{v})\ P \,\middle|\, \epsilon\!: Q\right] \in \Gamma(f)$ for the same input values $\overline{v}$.

The S-Cons rule further allows the specification context Γ' to be extended to Γ so long as Γ contains all specifications that Γ' does (i.e. $\Gamma' \subseteq \Gamma \triangleq \forall\, f.\ \Gamma'(f) \subseteq \Gamma(f)$). This ensures

Specification contexts $\boxed{\gamma <_{\mathsf{S}} \Gamma}$

$$\textsc{S-Empty}\quad \frac{}{\gamma <_{\mathsf{S}} \mathsf{empty}(\gamma)} \qquad \textsc{S-Extend}\quad \frac{\gamma <_{\mathsf{S}} \Gamma \quad f(\overline{x})\{e\} \in \gamma \quad \Gamma \vdash [P]\, e[\overline{v}/\overline{x}]\, [\epsilon\colon Q]}{\gamma <_{\mathsf{S}} \Gamma\big[f \mapsto \big[(\overline{v})\ P \,\big|\, \epsilon\colon Q\big],\ \Gamma(f)\big]}$$

Figure 6 Proof rules for well-formed specification contexts.

that Γ preserves any specification that may have been used to prove the original triple.

Note that RISL rules are only sound provided that the given specification context is consistent with the concrete implementation context under analysis. Therefore, we next show how to construct specification contexts soundly by deriving specifications in RISL.

5.2 Well-Formed Specification Contexts

The rules in Fig. 6 allow one to construct *well-formed* specification contexts. A specification context Γ is well-formed with respect to an implementation context γ, written $\gamma <_{\mathsf{S}} \Gamma$, when every specification in $\Gamma(f)$ holds of (can be derived for) the concrete implementation in $\gamma(f)$, for all f in γ. We write $\mathsf{empty}(\gamma)$ to denote an *empty specification context* for γ, i.e. one where $\Gamma(f)$ is an empty list, for all functions f in γ. An empty specification context for γ is vacuously well-formed with respect to γ (S-Empty). When $\gamma <_{\mathsf{S}} \Gamma$ holds, we can extend Γ (in a well-formed fashion) with a specification $\big[(\overline{v})\ P \,\big|\, \epsilon\colon Q\big]$ for f in γ, provided that we can prove it in RISL by directly executing the f body with concrete input values $\overline{v}$ (S-Extend).

In other words, by iterating over functions and deriving RISL (under-approximate) specifications for concrete inputs, we can dynamically construct Γ, tailor-made for a specific implementation context γ. This way, we can also reason about (mutually) recursive functions for a bounded number of iterations (i.e. under-approximate them) by first deriving specifications for non-recursive calls, which can then be used to derive specifications for the recursive calls. Moreover, note that the S-Cons rule ensures that derived RISL triples remain valid when extending the specification context, which in turn ensures that RISL is sound under specification context extension.

In the RUXt algorithm, we obtain reachable states by following the RISL proof (Fig. 5) and refinement (Fig. 6) rules. Indeed, every derived postcondition $[\epsilon\colon Q] \in \mathsf{execute}(\gamma, f(\overline{v}), P)$ on line 3 in Fig. 4 (left) implies that there exists Γ such that $\gamma <_{\mathsf{S}} \Gamma$ and $\Gamma \vdash [P]\, f(\overline{v})\, [\epsilon\colon Q]$ hold. As such, the soundness of RUXt follows directly from the soundness of RISL with respect to our under-approximate RUXtBelt semantics, which we describe next in §6.

6 RUXtBelt: The Semantic Model of RUXt

We present RUXtBelt, our under-approximate semantic model underpinning RISL and RUXt. As discussed in §1, RUXtBelt is closely related to RustBelt and covers a large fragment of RustBelt without references. Moreover, in RUXtBelt we explicitly account for memory safety errors (e.g. use after free) as this is required for bug detection. Specifically, while the RustBelt semantics simply gets 'stuck' when encountering a memory safety issue, in RUXtBelt we directly identify them as errors using our unsuccessful exit conditions.

In §6.1 we present the RUXtBelt operational semantics. In §6.2 we define the semantic interpretation of RISL triples and prove the RISL proof system sound against the RUXtBelt semantics. Finally in §6.3 we prove that the RUXt algorithm is sound through our inadequacy theorem, stating that every UB detected by RUXt is a true safety violation in the library.

6.1 RUXtBelt Operational Semantics

We present two characterisation of our RUXtBelt semantics. First, we describe our *big-step* semantics that explicitly accounts for unsuccessful termination, but does not distinguish between the two kinds of unsuccessful termination (err or miss). Specifically, in our big-step semantics we assume that we have the *full* heap at all times and do not miss any resources, i.e. that there are no additional resources in the frame. As such, when accessing a location l that is not within the underlying heap, we simply terminate with err (and not miss(l)). We thus refer to this big-step semantics as the RUXtBelt *full semantics*.

We next present the RUXtBelt *instrumented* (big-step) semantics – inspired by JaVerT2.0 [8] and Gillian [21] – that can operate on *partial* heaps and thus distinguishes between unsuccessful termination due to memory safety errors (err) and missing resources (miss). We then establish a semantics preservation lemma (Lemma 1) stating that our instrumented semantics can simply be erased to our big-step semantics, in that if a program e terminates with exit condition ϵ under the instrumented semantics, then (1) either $\epsilon =$ ok and e also terminates with ok in our big-step semantics; (2) or $\epsilon \neq$ ok (i.e. it is either err or miss) and e terminates with err in our big-step semantics.

As we show below, our instrumented semantics simplifies the task of proving RISL sound: miss RISL triples directly correspond to miss judgements in our instrumented semantics. As such, we first prove RISL sound against our instrumented semantics, and subsequently show that RISL is sound against our full semantics through our semantics preservation lemma.

RUXtBelt big-step semantics. We define our RUXtBelt big-step operational semantics in Fig. 7, through judgements of the form $\gamma \vdash \langle h \mid e \rangle \Downarrow \langle h' \mid \epsilon \rangle$, where γ is an implementation context (for function implementations), e is a λ_{RUXt} program (expression), ϵ is an exit condition and h, h' are *program states.* A program state is a heap, $h \in \textsc{Heap} \triangleq \textsc{Loc} \rightharpoonup \textsc{Val} \uplus \{☣, \varnothing\}$, mapping locations to either values in $\textsc{Val}$ or designated values for uninitialised (☣) and freed ($\varnothing$) locations. The $\gamma \vdash \langle h \mid e \rangle \Downarrow \langle h' \mid \epsilon \rangle$ judgement states that h' is reachable with exit condition ϵ by evaluating e starting from h, under the implementation context γ.

We further define partial evaluation functions for terms ($\lfloor\cdot\rfloor_{\mathsf{t}} : \textsc{Term} \rightharpoonup \textsc{Val}$) and pure expressions ($\lfloor\cdot\rfloor_{\mathsf{p}} : \textsc{Pure} \rightharpoonup \textsc{Val}$) as follows:

$$\begin{array}{lll}
\lfloor v \rfloor_{\mathsf{t}} \triangleq v & \lfloor -_{\mathbb{Z}} p \rfloor_{\mathsf{p}} \triangleq -\lfloor p \rfloor_{\mathsf{p}} \ \textbf{if}\ \lfloor p \rfloor_{\mathsf{p}} \in \mathbb{Z} & \lfloor p_1 +_{\mathbb{Z}} p_2 \rfloor_{\mathsf{p}} \triangleq \lfloor p_1 \rfloor_{\mathsf{p}} + \lfloor p_2 \rfloor_{\mathsf{p}} \ \textbf{if}\ \lfloor p_1 \rfloor_{\mathsf{p}}, \lfloor p_2 \rfloor_{\mathsf{p}} \in \mathbb{Z} \\
\lfloor t \rfloor_{\mathsf{p}} \triangleq \lfloor t \rfloor_{\mathsf{t}} & \lfloor \neg_{\mathbb{B}} p \rfloor_{\mathsf{p}} \triangleq \neg\lfloor p \rfloor_{\mathsf{p}} \ \textbf{if}\ \lfloor p \rfloor_{\mathsf{p}} \in \mathbb{B} & \lfloor p_1 \leq_{\mathbb{Z}} p_2 \rfloor_{\mathsf{p}} \triangleq \lfloor p_1 \rfloor_{\mathsf{p}} \leq \lfloor p_2 \rfloor_{\mathsf{p}} \ \textbf{if}\ \lfloor p_1 \rfloor_{\mathsf{p}}, \lfloor p_2 \rfloor_{\mathsf{p}} \in \mathbb{Z}
\end{array}$$

Most of the rules in Fig. 7 are standard. Pure expressions, assume expressions and explicit errors do not interact with the heap and leave it unchanged. Specifically, pure expressions simply reduce to their corresponding value (O-Pure), **assume**(t) is only reducible if t evaluates to true (O-Assume) and **error** terminates unsuccessfully with err (O-Error).

Heap allocation always returns a fresh, uninitialised location (O-Alloc). As in ISL [25], deallocation updates the value of the freed location to the designated $\varnothing$ value (O-Free). Storing to a heap location l is successful when l is not deallocated but may be potentially uninitialised (O-Store), while loading from l is only successful if l is initialised, in which case it returns the value at l (O-Load). Accessing a previously freed location or loading from an uninitialised location terminates erroneously (O-FreeErr, O-StoreErr and O-LoadErr). Moreover, accessing locations that are not part of the (full) heap also terminate erroneously

Big-step evaluation $\boxed{\gamma \vdash \langle h \mid e \rangle \Downarrow \langle h' \mid \epsilon \rangle}$

O-Pure
$$\frac{\lfloor p \rfloor_{\mathsf{p}} = v}{\gamma \vdash \langle h \mid p \rangle \Downarrow \langle h \mid \mathsf{ok}(v) \rangle}$$

O-Assume
$$\frac{\lfloor t \rfloor_{\mathsf{t}} = \mathsf{true}}{\gamma \vdash \langle h \mid \mathbf{assume}(t) \rangle \Downarrow \langle h \mid \mathsf{ok} \rangle}$$

O-Error
$$\frac{}{\gamma \vdash \langle h \mid \mathbf{error} \rangle \Downarrow \langle h \mid \mathsf{err} \rangle}$$

O-Alloc
$$\frac{l \notin \mathsf{dom}(h)}{\gamma \vdash \langle h \mid \mathbf{alloc} \rangle \Downarrow \langle h[l \mapsto ☣] \mid \mathsf{ok}(l) \rangle}$$

O-Free
$$\frac{\lfloor t \rfloor_{\mathsf{t}} = l \quad h(l) \in \textsc{Val} \uplus \{☣\}}{\gamma \vdash \langle h \mid \mathbf{free}(t) \rangle \Downarrow \langle h[l \mapsto \varnothing] \mid \mathsf{ok} \rangle}$$

O-Store
$$\frac{\lfloor t_1 \rfloor_{\mathsf{t}} = l \quad h(l) \in \textsc{Val} \uplus \{☣\} \quad \lfloor t_2 \rfloor_{\mathsf{t}} = v}{\gamma \vdash \langle h \mid t_1 \leftarrow t_2 \rangle \Downarrow \langle h[l \mapsto v] \mid \mathsf{ok} \rangle}$$

O-Load
$$\frac{\lfloor t \rfloor_{\mathsf{t}} = l \quad h(l) = v}{\gamma \vdash \langle h \mid {}^*t \rangle \Downarrow \langle h \mid \mathsf{ok}(v) \rangle}$$

O-FreeErr
$$\frac{\lfloor t \rfloor_{\mathsf{t}} = l \quad h(l) = \varnothing}{\gamma \vdash \langle h \mid \mathbf{free}(t) \rangle \Downarrow \langle h \mid \mathsf{err} \rangle}$$

O-StoreErr
$$\frac{\lfloor t_1 \rfloor_{\mathsf{t}} = l \quad h(l) = \varnothing}{\gamma \vdash \langle h \mid t_1 \leftarrow t_2 \rangle \Downarrow \langle h \mid \mathsf{err} \rangle}$$

O-LoadErr
$$\frac{\lfloor t \rfloor_{\mathsf{t}} = l \quad h(l) \in \{\varnothing, ☣\}}{\gamma \vdash \langle h \mid {}^*t \rangle \Downarrow \langle h \mid \mathsf{err} \rangle}$$

O-FreeMiss
$$\frac{\lfloor t \rfloor_{\mathsf{t}} = l \quad l \notin \mathsf{dom}(h)}{\gamma \vdash \langle h \mid \mathbf{free}(t) \rangle \Downarrow \langle h \mid \mathsf{err} \rangle}$$

O-StoreMiss
$$\frac{\lfloor t_1 \rfloor_{\mathsf{t}} = l \quad l \notin \mathsf{dom}(h)}{\gamma \vdash \langle h \mid t_1 \leftarrow t_2 \rangle \Downarrow \langle h \mid \mathsf{err} \rangle}$$

O-LoadMiss
$$\frac{\lfloor t \rfloor_{\mathsf{t}} = l \quad l \notin \mathsf{dom}(h)}{\gamma \vdash \langle h \mid {}^*t \rangle \Downarrow \langle h \mid \mathsf{err} \rangle}$$

O-Choice
$$\frac{\gamma \vdash \langle h \mid e_i \rangle \Downarrow \langle h' \mid \epsilon \rangle \quad \exists\, i \in \{1,2\}}{\gamma \vdash \langle h \mid e_1 + e_2 \rangle \Downarrow \langle h' \mid \epsilon \rangle}$$

O-LetCut
$$\frac{\gamma \vdash \langle h \mid e_1 \rangle \Downarrow \langle h' \mid \epsilon \rangle \quad \epsilon \neq \mathsf{ok}(-)}{\gamma \vdash \langle h \mid \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \rangle \Downarrow \langle h' \mid \epsilon \rangle}$$

O-Let
$$\frac{\gamma \vdash \langle h \mid e_1 \rangle \Downarrow \langle h'' \mid \mathsf{ok}(v) \rangle \quad \gamma \vdash \langle h'' \mid e_2[v/x] \rangle \Downarrow \langle h' \mid \epsilon \rangle}{\gamma \vdash \langle h \mid \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \rangle \Downarrow \langle h' \mid \epsilon \rangle}$$

O-Call
$$\frac{f(\overline{x})\{e\} \in \gamma \quad \gamma \vdash \langle h \mid e[\overline{t}/\overline{x}] \rangle \Downarrow \langle h' \mid \epsilon \rangle}{\gamma \vdash \langle h \mid f(\overline{t}) \rangle \Downarrow \langle h' \mid \epsilon \rangle}$$

Figure 7 The full RUXtBelt semantics; the instrumented RUXtBelt semantics is obtained by replacing err in O-FreeMiss, O-StoreMiss and O-LoadMiss with miss(l).

(O-FreeMiss, O-StoreMiss and O-LoadMiss). That is, the full heap must contain all resources required for successful termination or it terminates erroneously.

The remaining judgements closely mimic RISL rules in Fig. 5. Starting from h, non-deterministic choice $e_1 + e_2$ reduces to h' with ϵ if *either* e_1 *or* e_2 does so. A let binding short-circuits if e_1 terminates unsuccessfully (O-LetCut); otherwise, e_1 propagates its value v to e_2 by substituting the binder x (O-Let). A call to f is reduced by substituting its arguments with input terms in its body when f is in the implementation context (O-Call).

Instrumented semantics. We next define our *instrumented* semantics that operates on incomplete heaps and distinguishes err and miss conditions. Our instrumented semantics comprises judgements of the $\gamma \vdash \langle h \mid e \rangle \Downarrow_i \langle h' \mid \epsilon \rangle$ (note the i subscript in $\Downarrow$) and only differs from the full semantics by yielding a miss(l) exit in O-FreeMiss, O-StoreMiss and

O-LoadMiss. In effect, the full semantics maps every miss termination in the instrumented semantics to an err one, preserving all other outcomes, as we show in the following lemma.

▶ **Lemma 1** (Semantics preservation). *For all γ, h, h', e and ϵ, if $\gamma \vdash \langle h \mid e \rangle \Downarrow_i \langle h' \mid \epsilon \rangle$, then one of the following holds:*

1. $\epsilon = \mathsf{ok}(v)$ *and* $\gamma \vdash \langle h \mid e \rangle \Downarrow \langle h' \mid \mathsf{ok}(v) \rangle$.
2. $\epsilon \neq \mathsf{ok}(-)$ *holds and* $\gamma \vdash \langle h \mid e \rangle \Downarrow \langle h' \mid \mathsf{err} \rangle$.

As we show below, our instrumented semantics enjoys strong frame preservation properties, namely *frame subtraction* (Lemma 2) and *frame addition* (Lemma 3). The former describes the effects of removing fragments from the underlying heap, while the latter accounts for heap extensions. We write $h_s \# h_F$ to denote that h_s and h_F are *disjoint* (i.e. have disjoint domains) and can thus be composed (combined) through the disjoint union operator $\uplus$.

▶ **Lemma 2** (Frame subtraction). *For all γ, h_s, h_F, h'_s, e and ϵ, if $\gamma \vdash \langle h_s \uplus h_F \mid e \rangle \Downarrow_i \langle h'_s \uplus h_F \mid \epsilon \rangle$ and $h_s \# h_F$ holds, then one of the following holds:*

1. $\gamma \vdash \langle h_s \mid e \rangle \Downarrow_i \langle h'_s \mid \epsilon \rangle$ *and* $h'_s \# h_F$ *holds.*
2. *There exist h', l such that* $\gamma \vdash \langle h_s \mid e \rangle \Downarrow_i \langle h' \mid \mathsf{miss}(l) \rangle$, $h' \# h_F$ *and* $l \in \mathsf{dom}(h_F)$.

▶ **Lemma 3** (Frame addition). *For all γ, h, h', e and ϵ, if $\gamma \vdash \langle h \mid e \rangle \Downarrow_i \langle h' \mid \epsilon \rangle$, then for all h_F such that $h' \# h_F$, one of the following holds:*

1. $\gamma \vdash \langle h \uplus h_F \mid e \rangle \Downarrow_i \langle h' \uplus h_F \mid \epsilon \rangle$ *and* $h \# h_F$ *holds.*
2. *There exists l such that* $\epsilon = \mathsf{miss}(l)$ *and* $l \in \mathsf{dom}(h_F)$ *hold.*

Lemma 2 states that removing a fragment from the heap under execution either (1) preserves the behaviour of the program; or (2) terminates unsuccessfully due to a missing location in the frame. Conversely, Lemma 3 states that extending the underlying heap (1) preserves the behaviour of the program unless (2) the program terminates with a miss before adding the frame, and the frame contains the missing location. This is precisely why we need the $\mathsf{frameable}(\epsilon, R)$ requirement in the premise of the S-Frame rule, i.e. to ensure that the extension R does not include missing resources that may change the termination condition.

These lemmas demonstrate how (when dealing with incomplete heaps in our instrumented semantics) extending and shrinking the underlying heap may affect the termination condition with regards to missing resources. By contrast, in our full semantics we assume that the underlying heap is always complete, and thus we need not account for extending or shrinking it. As our instrumented semantics considers partial (incomplete) heaps, it closely captures the *compositional* nature of our RISL proof rules, where in each rule the pre- and post-conditions only describe the (partial) heap fragments needed for executing a given λ_{RUXt} program, and the underlying heap (resources) may always be extended using the S-Frame rule. As such, it is simpler first to show RISL sound against our instrumented semantics., and then to relate it to our full semantics using Lemma 1, as we show next in §6.2.

6.2 Semantic RISL Triples and Soundness

We next formalise the *semantic interpretation* of RISL triples. Recall that RISL triples (Fig. 5) are given using assertions. As such, we first present the semantics of RISL assertions.

Semantic interpretation $\boxed{\llparenthesis\cdot\rrparenthesis : \textsc{Asrt} \to \textsc{Heap} \to \textsc{Prop}}$

$$\llparenthesis \llcorner \pi \lrcorner \rrparenthesis \triangleq \lambda\, h.\ h = \emptyset \wedge \pi \qquad \llparenthesis \mathsf{True} \rrparenthesis \triangleq \lambda\, \_.\ \top \qquad \llparenthesis \mathsf{False} \rrparenthesis \triangleq \lambda\, \_.\ \bot$$

$$\llparenthesis P \wedge Q \rrparenthesis \triangleq \lambda\, h.\ \llparenthesis P \rrparenthesis(h) \wedge \llparenthesis Q \rrparenthesis(h) \qquad \llparenthesis P \vee Q \rrparenthesis \triangleq \lambda\, h.\ \llparenthesis P \rrparenthesis(h) \vee \llparenthesis Q \rrparenthesis(h)$$

$$\llparenthesis P \Rightarrow Q \rrparenthesis \triangleq \lambda\, h.\ \llparenthesis P \rrparenthesis(h) \Rightarrow \llparenthesis Q \rrparenthesis(h) \qquad \llparenthesis \exists\, x.\ P \rrparenthesis \triangleq \lambda\, h.\ \exists\, x.\ \llparenthesis P \rrparenthesis(h)$$

$$\llparenthesis l \mapsto v \rrparenthesis \triangleq \lambda\, h.\ h = \{l \mapsto v\} \qquad \llparenthesis l \mapsto_{?} \rrparenthesis \triangleq \lambda\, h.\ h = \{l \mapsto \text{☣}\} \qquad \llparenthesis l \not\mapsto \rrparenthesis \triangleq \lambda\, h.\ h = \{l \mapsto \varnothing\}$$

$$\llparenthesis \mathsf{emp} \rrparenthesis \triangleq \lambda\, h.\ h = \emptyset \qquad \llparenthesis P * Q \rrparenthesis \triangleq \lambda\, h.\ \exists\, h_P, h_Q.\ h = h_P \uplus h_Q \wedge \llparenthesis P \rrparenthesis(h_P) \wedge \llparenthesis Q \rrparenthesis(h_Q)$$

Figure 8 Assertion semantics.

Assertion semantics. We define the semantics of assertions in Fig. 8 through an interpretation function, $\llparenthesis\cdot\rrparenthesis$, that maps assertions to predicates ($\textsc{Prop}$ in Rocq) on heaps (i.e. our program states). Pure assertions require no heap resources and thus assert ownership of an empty heap. Semantics of truth, falsity do not depend on the heap, semantics of conjunction, disjunction, implication and existential quantification are defined inductively and as expected. For separation logic (SL) assertions, emp simply expresses ownership of an empty heap, while $l \mapsto v$ expresses ownership of a singleton heap mapping l to v. Similarly, $l \not\mapsto$ and $l \mapsto_{?}$ express ownership of a singleton heap mapping l to $\varnothing$ and ☣, respectively. Finally, the separating conjunction requires the heap to be split into two disjoint sub-heaps, each satisfying one sub-assertion. An assertion P is *valid*, written $\models P$, if it holds of *every* heap; P is *satisfiable*, written $\mathsf{sat}(P)$, if it holds of *some* heap:

$$\models P \triangleq \forall\, h.\ \llparenthesis P \rrparenthesis(h) \qquad \mathsf{sat}(P) \triangleq \exists\, h.\ \llparenthesis P \rrparenthesis(h)$$

Semantic RISL triples. A *semantic* triple (*cf.* *syntactic* RISL triples in Fig. 5), $\gamma \vDash [P]\, e\, [\epsilon\colon Q]$, states that every heap h_q satisfying Q is reachable under ϵ from some heap h_p satisfying P by executing e under implementation context γ:

$$\gamma \vDash [P]\, e\, [\epsilon\colon Q] \triangleq \forall\, h_q.\ \llparenthesis Q \rrparenthesis(h_q) \Rightarrow \exists\, h_p.\ \llparenthesis P \rrparenthesis(h_p) \wedge \gamma \vdash \langle h_p \mid e \rangle \Downarrow_i \langle h_q \mid \epsilon \rangle$$

Note that while the syntactic RISL triples (Fig. 5) are defined using a *specification* context Γ, semantic triples (above) are given using an *implementation* context γ. To give a semantic meaning to our RISL triples, we thus lift the notion of a semantic triple with γ, to one using Γ, provided that Γ is *well-formed* with respect to γ. Recall that in §5.2 we provided a *logical* notion of this well-formedness, $\gamma \lessdot_{\mathsf{S}} \Gamma$, that can be derived using the rules in Fig. 6. We analogously define a *semantic* notion of well-formedness as follows. We define the semantics of the refinement relation from §5.2 as follows:

$$\gamma \lll_{\mathsf{S}} \Gamma \triangleq \forall f, \overline{v}, P, \epsilon, Q.\ \left[(\overline{v})\ P \,\middle|\, \epsilon\colon Q\right] \in \Gamma(f) \Rightarrow \exists\, \overline{x}, e.\ f(\overline{x})\{e\} \in \gamma \wedge \gamma \vDash [P]\, e[\overline{v}/\overline{x}]\, [\epsilon\colon Q]$$

This ensures that every specification $\left[(\overline{v})\ P \,\middle|\, \epsilon\colon Q\right]$ of function f in Γ is the result of directly executing the function body with the given input values.

We next lift the notion of semantic triples (with γ) to *semantic* RISL *triples* (with Γ):

$$\Gamma \vDash [P]\, e\, [\epsilon\colon Q] \triangleq \forall\, \gamma.\ \gamma \lll_{\mathsf{S}} \Gamma \Rightarrow \gamma \vDash [P]\, e\, [\epsilon\colon Q]$$

That is, as RISL only relies on derived specifications and not on concrete implementations, a semantic RISL triple with Γ holds when the corresponding semantic triple holds for all implementation contexts γ with respect to which Γ is semantically well-formed.

Soundness of RISL. In Lemma 4 below we prove that (i) every RISL triple that can be derived using the rules in Fig. 5 is a valid semantic RISL triple.; and (ii) every well-formed specification context Γ constructed using the rules in Fig. 6 is semantically well-formed.

▶ **Lemma 4.** *For all* Γ, γ, P, Q, e *and* ϵ*:*

1. *if* $\Gamma \vdash [P]\, e\, [\epsilon\colon Q]$ *is derived using the rules in Fig. 5, then* $\Gamma \vDash [P]\, e\, [\epsilon\colon Q]$ *holds; and*
2. *if* $\gamma \prec_{\mathsf{S}} \Gamma$ *is derived using the rules in Fig. 6, then* $\gamma \lll_{\mathsf{S}} \Gamma$ *holds.*

Finally, we prove that the RISL proof system is sound by showing that every triple derived for a given implementation using the RISL rules in Fig. 5 is a (sound) semantic triple.

▶ **Theorem 5** (RISL soundness)**.** *For all* Γ, γ, P, Q, e *and* ϵ*, if* $\gamma \prec_{\mathsf{S}} \Gamma$ *and* $\Gamma \vdash [P]\, e\, [\epsilon\colon Q]$ *is derived using the rules in Fig. 5, then* $\gamma \vDash [P]\, e\, [\epsilon\colon Q]$ *holds.*

Crucially, RUXt relies on this result to justify the soundness of its summary inference. Specifically, every postcondition $[\epsilon\colon Q] \in \mathsf{execute}(\gamma, f(\overline{v}), P)$ derived by its underlying symbolic execution engine (underpinned by RISL) on line 3 (Fig. 4, left) must ensure that $\gamma \vDash [P]\, f(\overline{v})\, [\epsilon\colon Q]$ holds, which follows immediately from the RISL soundness above (Theorem 5). That is, by implementing the RISL proof system in its underlying symbolic execution engine, the RUXt algorithm thus guarantees that every summary it infers is a sound under-approximation of the reachable states.

6.3 Soundness of RUXt

We next show that RUXt is sound by proving its no-false-positives guarantee: when RUXt detect a UB with a witness program e, then e is indeed a safe program that results in UB, witnessing that the usage of unsafe code is not encapsulated by the safe API of the library. To this end, using the RISL soundness result (Theorem 5), in Theorem 6 below we first show that every summary derived by RUXt yields a sound RISL triple starting from an initial empty heap and terminating successfully. Furthermore, the witness program associated with inferred summary is guaranteed to be a safe program, meaning that the corresponding safe value can be constructed by solely interacting with the safe API of the library. More concretely, given an implementation context γ, a signature context Δ and a summary context Σ, if RUXt derives $\gamma, \Delta \vdash \Sigma$ using the rules in Fig. 4 (right) with (v, Q, e) in Σ, then (1) e is a safe program; and (2) $\gamma \vDash [\mathsf{emp}]\, e\, [\mathsf{ok}(v)\colon Q]$ holds.

▶ **Theorem 6** (Summary soundness)**.** *For all* γ, Δ, Σ, v, Q, e*, if* $\gamma, \Delta \vdash \Sigma$ *is derived using the rules in Fig. 4 and* $(v, Q, e) \in \Sigma$*, then* e *is a safe program and* $\gamma \vDash [\mathsf{emp}]\, e\, [\mathsf{ok}(v)\colon Q]$ *holds.*

Finally, using the soundness of summary inference in RUXt, we prove our key *inadequacy* theorem: given a well-formed summary context, if the refutation procedure reports a type unsoundness, then the returned witness is a safe program that leads to UB.

▶ **Theorem 7** (Inadequacy)**.** *For all* γ, Δ, Σ, e*, if* $\gamma, \Delta \vdash \Sigma$ *is derived using the rules in Fig. 4 and* $\mathsf{TryRefute}(\gamma, \Delta, \Sigma)$ *returns a witness program* e*, then* e *is a safe program and* $\gamma \vdash \langle \emptyset \mid e \rangle \Downarrow \langle h \mid \mathsf{err} \rangle$*, for some* h*.*

Note that our inadequacy theorem states that e results in an error in our *full* semantics starting from an empty heap. Although summaries are only shown to be reachable using the *instrumented* semantics (in the conclusion of Theorem 6), our semantics preservation lemma (Lemma 1) allows us to exchange any judgement of the instrumented semantics into one in our full semantics (by leaving successful judgements unchanged, while transforming unsuccessful ones with miss or err into erroneous ones with err), provided that we operate on the *full* heap. Indeed, the empty heap should suffice for safe programs to execute safely, and thus RUXt can soundly report miss exits as true UB. Concretely, our inadequacy theorem states that *any UB reported by* RUXt *is a true safety violation in the library*, that can be triggered by its returned witness program.

7 Related and Future Work

RUXt exists in a rich ecosystem of tools and techniques for analysing unsafe Rust code, as well as techniques that combine type-guided reasoning with under-approximate reasoning. We discuss these in turn, categorising Rust analysis tools into three categories: compositional verification, whole-program analysis and bug-finding tools.

Compositional type-safety verification for Rust. There are a number of tools that aim to *verify* the type safety of Rust programs. RustBelt [15] is an Iris mechanisation [16, 17] that provides a formal model of Rust's type system and provides a framework for compositionally verifying the type safety of internally unsafe libraries. Later, RefinedRust [9] (based on Lithium [27]), VeriFast for Rust [6] (based on VeriFast [11]) and Gillian-Rust [1] (based on Gillian [7, 21]) are all underpinned by RustBelt and aim to verify the type safety of Rust code, each with their own set of caveats and trade-offs. Verus [18] is a language (embedded in Rust) for writing formally verified code that can be extracted to idiomatic Rust code. However, Verus cannot verify existing unsafe Rust code that has not been written within Verus. While these tools are powerful, they all require a substantial amount of manual effort on the part of users and, as verification tools, are prone to false positives.

Whole-program analysis for Rust. Miri is a Rust interpreter distributed with the Rust compiler. It allows for the concrete execution of Rust programs and can detect many UB such as out-of-bounds memory accesses or more subtle Rust-specific issues, e.g. the violation of aliasing rules [14]. However, Miri is a testing tool, it requires the user to write tests, and can only explore one execution path.

Kani [29] is a whole-program bounded model checker for Rust. It performs symbolic execution, and as such can explore infinitely many execution paths. However, Kani still requires users to write tests and first-order logic assertions, and it is prone to *both* false positives (due to its over-approximate nature) *and* false negatives (as it is bounded). Moreover, Kani is unsound with respect to several Rust-specific features, e.g. the detection of uninitialised memory accesses, or the violation of aliasing rules. Neither Miri nor Kani can detect type unsoundness in Rust programs, as they cannot reason compositionally about library invariants.

Bug-finding tools for Rust. We know of three automatic bug-finders for Rust: RUDRA [2], TraitInv [5] and Crabtree [28]. RUDRA uses linter-like heuristics to detect three specific unsafe patterns in Rust code that are known to often lead to type unsoundness. It is automatic and was ran on the entirety of the Rust openly-available packages (`crates.io`), finding more than 250 new safety issues. However, its scope is limited to the detection of the three patterns, and it is prone to false positives.

TraitInv automatically synthesises symbolic tests that check certain properties expected to hold for eight core Rust traits (e.g. that `PartialEq` is symmetric and transitive). TraitInv then run these tests using Kani and reports violations. Unfortunately, TraitInv's approach is limited to these specific traits, and supporting new traits requires manual effort as it cannot infer new trait invariants. Furthermore, since it relies on Kani, TraitInv is also prone to false positives; it can only be extended to support first-order properties and cannot be used to enforce separation logic (ownership) properties such as that required by the `Copy` trait.

Crabtree is an automatic test synthesiser that uses a type-guided approach to increase coverage of the library under test. It can generate tests that make use of advanced Rust features such as higher-order trait functions; it uses Miri to execute the generated tests and can detect type unsoundness in the library if a test fails. Its type-guided approach is somewhat similar to ours. However, instead of inferring type subvariants, Crabtree notes when it finds a *path* that generates a value of a type not previously seen in the test suite, and uses that information to call functions that have not yet been covered because of the lack of input values for this type. This approach is, unlike ours, not compositional, and still quickly leads to combinatorial (path) explosion.

Type refutation and under-approximation. The recent development of Incorrectness Logic [25] has led to the exploration of reasoning techniques that combine type systems with under-approximation, starting with a theoretical experiment from Ramsay and Walpole [26] who introduced a two-sided type system that allows for refutation of type assignments.

Next, HAYSTACK [31] is an extension of LIQUIDHASKELL [30], a refinement type system for Haskell. HAYSTACK is similar to RUXt, in that it infers subsets of the refinement type spaces that are sufficient to refute the refinement types provided by the user. However, it differs from RUXt in several key aspects. First, HAYSTACK is used to provide explanations for type errors when the user provides a refinement type annotation that cannot be proven by LIQUIDHASKELL. However, as refinement type checking is inherently over-approximate, inability to type-check does not guarantee that the program is incorrect. Furthermore, HAYSTACK can only help finding counter-examples in Haskell programs that have been annotated with refinement types and is therefore not fully automatic. Finally, LIQUIDHASKELL refinement types are first-order and cannot express separation logic (ownership) properties.

Finally, Qian et al. [24] propose a semantic type refuter that can find type unsoundness in a simple functional language. However, their approach requires knowledge of the *semantic typing of all existing types*, which must be defined using first-order logic. As such, they cannot find type unsoundness in internally-unsafe libraries, where type invariants must be inferred from the code itself and must be expressed using separation logic.

Future Work. In the future, we will extend and build on our work here in several ways. First, we will extend our λ_{RUXt} language and RUXtBelt formalisation to account for references. We will further extend our formalism to include support for other Rust features such as polymorphism and higher-order functions. Second, we will show in our extended formalism that our current program transformation for handling references is sound, in that any UB in a transformed function without references is also a UB in the original Rust function with references. Third, we will extend RUXt accordingly to include support for these features, and further ensure that UB reports are accompanied by witness programs. Finally, we will perform a large-scale evaluation of RUXt by applying it to real Rust codebases.

References

1 Sacha-Élie Ayoun, Xavier Denis, Petar Maksimović, and Philippa Gardner. A hybrid approach to semi-automated rust verification, 2025. URL: https://arxiv.org/abs/2403.15122, arXiv:2403.15122.

2 Yechan Bae, Youngsuk Kim, Ammar Askar, Jungwon Lim, and Taesoo Kim. Rudra: Finding memory safety bugs in rust at the ecosystem scale. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 84–99, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3477132.3483570.

3 Ariel Ben-Yehuda. std::thread::joinguard (and scoped) are unsound because of reference cycles. rust issue #24292., 2015. URL: https://github.com/rust-lang/rust/issues/24292.

4 Christophe Biocca. std::vec::intoiter::as_mut_slice borrows &self, returns &mut of contents. rust issue #39465, 2017. URL: https://github.com/rust-lang/rust/issues/39465.

5 Twain Byrnes, Yoshiki Takashima, and Limin Jia. Automatically enforcing rust trait properties. In Rayna Dimitrova, Ori Lahav, and Sebastian Wolff, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 210–223, Cham, 2024. Springer Nature Switzerland.

6 Nima Rahimi Foroushaani and Bart Jacobs. Modular Formal Verification of Rust Programs with Unsafe Blocks, December 2022. arXiv:2212.12976, doi:10.48550/arXiv.2212.12976.

7 José Fragoso Santos, Petar Maksimović, Sacha-Élie Ayoun, and Philippa Gardner. Gillian, part i: A multi-language platform for symbolic execution. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, pages 927–942, New York, NY, USA, June 2020. Association for Computing Machinery. doi:10.1145/3385412.3386014.

8 José Fragoso Santos, Petar Maksimović, Gabriela Sampaio, and Philippa Gardner. JaVerT 2.0: Compositional symbolic execution for JavaScript. *Proceedings of the ACM on Programming Languages*, 3(POPL):66:1–66:31, January 2019. doi:10.1145/3290379.

9 Lennard Gäher, Michael Sammler, Ralf Jung, Robbert Krebbers, and Derek Dreyer. Refinedrust: A type system for high-assurance verification of rust programs. *Proc. ACM Program. Lang.*, 8(PLDI), June 2024. doi:10.1145/3656422.

10 Samin S. Ishtiaq and Peter W. O'Hearn. BI as an assertion language for mutable data structures. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL, page 14–26, New York, NY, USA, 2001. Association for Computing Machinery. URL: https://doi.org/10.1145/360204.375719.

11 Bart Jacobs, Jan Smans, and Frank Piessens. A Quick Tour of the VeriFast Program Verifier. In Kazunori Ueda, editor, *Programming Languages and Systems*, Lecture Notes in Computer Science, pages 304–311, Berlin, Heidelberg, 2010. Springer. doi:10.1007/978-3-642-17164-2_21.

12 Ralf Jung. The Scope of Unsafe. https://www.ralfj.de/blog/2016/01/09/the-scope-of-unsafe.html, January 2016.

13 Ralf Jung. Mutexguard<cell<i32» must not be sync. rust issue #41622, 2017. URL: https://github.com/rust-lang/rust/issues/41622.

14 Ralf Jung, Hoang-Hai Dang, Jeehoon Kang, and Derek Dreyer. Stacked borrows: An aliasing model for Rust. *Proceedings of the ACM on Programming Languages*, 4(POPL):41:1–41:32, December 2019. doi:10.1145/3371109.

15 Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. Rustbelt: securing the foundations of the rust programming language. *Proc. ACM Program. Lang.*, 2(POPL), December 2017. `doi:10.1145/3158154`.

16 Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming*, 28:e20, November 2018. `doi:10.1017/S0956796818000151`.

17 Robbert Krebbers, Jacques-Henri Jourdan, Ralf Jung, Joseph Tassarotti, Jan-Oliver Kaiser, Amin Timany, Arthur Charguéraud, and Derek Dreyer. MoSeL: a general, extensible modal framework for interactive proofs in separation logic. *Proceedings of the ACM on Programming Languages*, 2(ICFP), July 2018. `doi:10.1145/3236772`.

18 Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. Verus: Verifying Rust Programs using Linear Ghost Types. *Proceedings of the ACM on Programming Languages*, 7(OOPSLA1):85:286–85:315, April 2023. `doi:10.1145/3586037`.

19 Quang Loc Le, Azalea Raad, Jules Villard, Josh Berdine, Derek Dreyer, and Peter W. O'Hearn. Finding real bugs in big programs with incorrectness logic. *Proc. ACM Program. Lang.*, 6(OOPSLA1), apr 2022. `doi:10.1145/3527325`.

20 Andreas Lööw, Daniele Nantes-Sobrinho, Sacha-Élie Ayoun, Caroline Cronjäger, Petar Maksimović, and Philippa Gardner. Compositional Symbolic Execution for Correctness and Incorrectness Reasoning. In Jonathan Aldrich and Guido Salvaneschi, editors, *38th European Conference on Object-Oriented Programming (ECOOP 2024)*, volume 313 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 25:1–25:28, Dagstuhl, Germany, 2024. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. `doi:10.4230/LIPIcs.ECOOP.2024.25`.

21 Petar Maksimović, Sacha-Élie Ayoun, José Fragoso Santos, and Philippa Gardner. Gillian, Part II: Real-World Verification for JavaScript and C. In Alexandra Silva and K. Rustan M. Leino, editors, *Computer Aided Verification*, Lecture Notes in Computer Science, pages 827–850, Cham, 2021. Springer International Publishing. `doi:10.1007/978-3-030-81688-9_38`.

22 Nicholas D. Matsakis and Felix S. Klock. The rust language. In *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*, HILT '14, pages 103–104, New York, NY, USA, October 2014. Association for Computing Machinery. `doi:10.1145/2663171.2663188`.

23 Peter W. O'Hearn. Incorrectness logic. *Proc. ACM Program. Lang.*, 4(POPL):10:1–10:32, December 2019. URL: `http://doi.acm.org/10.1145/3371078`.

24 Kelvin Qian, Scott Smith, Brandon Stride, Shiwei Weng, and Ke Wu. Semantic-Type-Guided Bug Finding. *Software Artifact for Semantic-Type-Guided Bug Finding*, 8(OOPSLA2):348:2183–348:2210, October 2024. `doi:10.1145/3689788`.

25 Azalea Raad, Josh Berdine, Hoang-Hai Dang, Derek Dreyer, Peter O'Hearn, and Jules Villard. Local reasoning about the presence of bugs: Incorrectness separation logic. In Shuvendu K. Lahiri and Chao Wang, editors, *Computer Aided Verification*, pages 225–252, Cham, 2020. Springer International Publishing.

26 Steven Ramsay and Charlie Walpole. Ill-Typed Programs Don't Evaluate. *Proceedings of the ACM on Programming Languages*, 8(POPL):2010–2040, January 2024. `doi:10.1145/3632909`.

27 Michael Sammler, Rodolphe Lepigre, Robbert Krebbers, Kayvan Memarian, Derek Dreyer, and Deepak Garg. RefinedC: Automating the foundational verification of C code with

refined ownership types. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, pages 158–174, New York, NY, USA, June 2021. Association for Computing Machinery. doi:10.1145/3453483.3454036.

28 Yoshiki Takashima, Chanhee Cho, Ruben Martins, Limin Jia, and Corina S. Păsăreanu. Crabtree: Rust API Test Synthesis Guided by Coverage and Type. *Artifact Package for Paper "Crabtree: Rust API Test Synthesis Guided by Coverage and Type"*, 8(OOPSLA2):293:618–293:647, October 2024. doi:10.1145/3689733.

29 The Kani Team. How Open Source Projects are Using Kani to Write Better Software in Rust | AWS Open Source Blog. https://aws.amazon.com/blogs/opensource/how-open-source-projects-are-using-kani-to-write-better-software-in-rust/, November 2023.

30 Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. Refinement types for Haskell. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, pages 269–282, New York, NY, USA, August 2014. Association for Computing Machinery. doi:10.1145/2628136.2628161.

31 Robin Webbers, Klaus Von Gleissenthall, and Ranjit Jhala. Refinement Type Refutations. *Proceedings of the ACM on Programming Languages*, 8(OOPSLA2):962–987, October 2024. doi:10.1145/3689745.