JOT
JOURNAL OF
OBJECT TECHNOLOGY

Journal of Object Technology | **RESEARCH ARTICLE**

Revisiting Borrow Checking with Abstract Interpretation

Aurélien Coet and Didier Buchs
University of Geneva, Switzerland

ABSTRACT The prevalence of memory safety vulnerabilities in contemporary software systems has made the adoption of memory-safe programming languages increasingly critical. Among the available options, borrow checking, introduced by the Rust programming language, offers a compelling balance between safety and performance by enforcing disciplined ownership and reference semantics at compile time. However, further formalisation and refinement of the underlying framework is necessary, particularly to enhance its precision in the presence of flow-sensitive programming patterns.

In this paper, we propose a novel reinterpretation of Rust's borrow checking mechanism as a form of abstract interpretation over a dedicated intermediate representation (IR). Our methodology captures the aliasing and initialisation properties of variables through the construction of abstract program traces that enable fine-grained, path-sensitive verification of memory safety properties. We formalise the syntax and operational semantics of the IR and develop a sound type system that enforces origin-based constraints on references. Through a series of examples, we show how our framework can successfully validate programs that are rejected by Rust's current implementation due to conservative approximations. By decoupling borrow checking from Rust's surface syntax and embedding it within a language-agnostic IR, our work lays the foundation for integrating Rust-style memory safety into a broader class of programming languages and compiler infrastructures.

KEYWORDS Borrow Checking, Abstract Interpretation, Memory Safety, Intermediate Representation

1. Introduction

Memory safety violations have long been recognised as a major source of subtle, difficult-to-diagnose software bugs. Beyond their impact on software reliability, these errors also pose significant security risks, as they often serve as vectors for critical vulnerabilities and system exploits (Van Der Veen et al. 2012; Song et al. 2019). Numerous empirical studies estimate that approximately 70% of documented security vulnerabilities are attributable to memory safety issues, underscoring their central role in contemporary software security challenges (Kehrer 2019; Miller 2019; (MSRC) 2019; Rebert & Kern 2024).

In response to the prevalence and severity of these vulnerabilities, leading stakeholders across industry, academia, and government have increasingly advocated for the adoption of memory-safe technologies, in particular memory-safe programming languages (Watson et al. 2025). Evidence suggests that such initiatives yield tangible security benefits: for instance, Google reported a reduction in the proportion of memory safety vulnerabilities in Android from 76% to 24% over a six-year period, a change attributed to the gradual adoption of memory-safe languages in system components (Stoep & Rebert 2024).

Traditionally, memory safety in programming languages has been closely associated with dynamic memory management techniques such as garbage collection and reference counting. While effective at eliminating broad classes of memory errors, these techniques introduce runtime overhead and unpredictability, characteristics that are often unacceptable in resource-constrained or performance-critical environments. Consequently, memory-unsafe languages such as C and C++ have historically remained dominant in these settings.

Recent advances in programming language technology have however significantly changed the situation. Spearheaded by *Rust*[1] and building upon decades of research, a new generation of systems programming languages incorporating strong mem-

JOT reference format:
Aurélien Coet and Didier Buchs. *Revisiting Borrow Checking with Abstract Interpretation*. Journal of Object Technology. Vol. 25, No. 1, 2026.
http://dx.doi.org/10.5381/jot.2026.25.1.a14

[1] https://www.rust-lang.org/

ory safety guarantees without relying on garbage collection is emerging (Carruth 2022; McCall 2023; Abrahams 2024).

At the core of Rust's memory model lie the concepts of *ownership* and *borrowing*, which together define a set of statically enforced constraints on memory usage. These constraints are verified by the *borrow checker*, a compile-time analysis that integrates type system rules with dataflow reasoning to ensure safe memory access patterns. By enforcing strict invariants on aliasing and mutation, the borrow checker enables the Rust compiler to perform deterministic memory management and statically insert allocation and deallocation operations in programs, thus eliminating the need for other runtime memory management mechanisms.

Despite the practical success of borrow checking and its adoption in other languages (McCall 2017; Racordon & Abrahams 2023; Lorenzen et al. 2024), the approach has not yet been rigorously and comprehensively formalised. While the design draws upon a rich body of prior work (Reynolds 1978; Smith et al. 2000; Walker et al. 2000; Fahndrich & DeLine 2002; Jim et al. 2002; Boyland 2003; Haller & Odersky 2010), the authoritative reference for its behaviour remains its implementation within the compiler. Several efforts have sought to formalise aspects of borrow checking using type-theoretic frameworks (Jung et al. 2018; Pearce 2021; Weiss et al. 2021), but these often only partially reflect the behaviour of the actual analysis, especially with regards to its flow-sensitivity.

In practice, borrow checking in Rust is not implemented purely through type system rules, but rather as a form of dataflow analysis performed over a Mid-level Intermediate Representation (MIR). While MIR abstracts away much of the high-level syntax of the language, it remains tightly coupled with the internal architecture of its compiler. This close integration complicates reuse and poses barriers for adoption in other compiler infrastructures. As a result, language designers seeking to adopt borrow-checking-like mechanisms are often required to develop bespoke implementations, leading to duplication of effort and reduced interoperability.

In this paper, we propose a formalisation of the borrow checker within the framework of *abstract interpretation* (Cousot & Cousot 1977). By embedding borrow checking into this well-established framework, we provide a mathematically rigorous foundation that enables formal reasoning about soundness. Additionally, the flexibility of abstract interpretation in the selection and design of abstract domains allows us to refine the analysis to improve its precision in specific cases, enabling it to accept a broader class of correct programs while maintaining soundness guarantees.

Our formalisation is built upon a lightweight intermediate representation (IR) designed to capture the essential memory operations relevant to borrow checking. We have implemented this IR within a prototype compiler and demonstrate its utility by translating a small Rust-inspired language into this representation. While our current implementation serves primarily as a proof of concept, we argue that a more complete and scalable version could serve as a re-targetable, source-language-agnostic borrow-checking IR. Such an IR would provide compiler developers with a reusable and extensible foundation for integrating borrow checking into a variety of languages, much like existing compiler infrastructures such as LLVM (Lattner & Adve 2004), MLIR (Lattner et al. 2021) and Cranelift (Alliance 2023) do for optimization and code generation.

2. Memory Safety in Rust

2.1. Ownership

In Rust, every object is associated with a unique *owner* that holds exclusive authority over the value's representation in memory. By default, assigning a value to another variable transfers ownership in an operation known as a *move*, which invalidates the original owner and guarantees that no two variables can simultaneously mutate or deallocate the same memory location.

```
fn f(owner3: String) {
  // Some code...
}

fn main() {
  let owner1 = "Some string".to_string();
  let owner2 = owner1;
  f(owner2);
  println!("{}", owner2);
}
```

Listing 1 Move operations in Rust

Listing 1 illustrates this concept through a simple example. In this program, a string is first allocated and assigned to the variable `owner1`, which becomes its initial *owner*. When the value of `owner1` is assigned to `owner2`, ownership of the string is *moved* to `owner2`. As a result, the lifetime of the string becomes tied to `owner2`, and `owner1` is invalidated: accessing it later in the program is a compile-time error. Subsequently, when `owner2` is passed as an argument to the function `f`, ownership is once again moved, this time into the parameter `owner3`. Finally, an attempt to use `owner2` after this transfer results in a compilation error, as the value it once held has been moved and is no longer accessible.

The ownership discipline in Rust is enforced through an *affine substructural type system* (Walker 2004), which ensures that values with move semantics are used at most once. This invariant enables the compiler to automatically and safely deallocate memory when its owning variable goes out of scope, eliminating the need for runtime mechanisms like garbage collection.

2.2. Borrowing

To enhance expressiveness, Rust permits data types to be designated as either explicitly cloneable or implicitly copyable, thereby overriding the default move semantics that govern assignments. More fundamentally, the language introduces the notion of references, which provide temporary access to values without transferring ownership through an operation known as *borrowing*. References may be immutable (`&T`) or mutable (`&mut T`), and they are constrained by the single-writer, multiple-reader discipline: multiple immutable references may coexist, or a single mutable reference may exist, but never both simultaneously. This invariant guarantees the safety of concurrent

access patterns and ensures the correctness of statically inserted deallocations.

```
fn f(r: &mut String) {
  *r = "Some text".to_string();
}

fn main() {
  let mut text = "Some string".to_string();
  f(&mut text);
  println!("{}", text);
}
```

Listing 2 References in Rust

Listing 2 illustrates the use of a mutable reference to modify the contents of a string in place, without transferring its ownership. In this example, the string `"Some string"` is first allocated in memory and bound to the variable `text`. A mutable reference to `text` is then passed as an argument to the function `f`, which updates the value pointed to by its parameter `r` to `"Some text"`. After `f` returns, the program prints the value of `text`, which now reflects the modification and produces the output `"Some text"`.

```
fn f(r: &mut String) {
  *r = "Some text".to_string();
}

fn main() {
  let mut text = "Some string".to_string();
  let r = &text;
  f(&mut text);
  println!("{}", *r);
  println!("{}", text);
}
```

Listing 3 A borrow error in Rust

If we modify the example from listing 2 into the version shown in listing 3, the program no longer compiles due to a violation of the single-writer, multiple-readers rule. At the point where a mutable reference to `text` is passed to `f`, an immutable reference to the same value is still active in variable `r`. Allowing such a program would compromise *referential transparency*: the value accessible through `r` could change as a result of the call to `f`, despite `r` being an immutable reference. This would violate Rust's safety guarantees by introducing the possibility of observing mutations through ostensibly immutable references.

However, if the use of `r` immediately after the call to `f` is removed, the program compiles successfully. In this case, the immutable reference is no longer considered *live* when the mutable borrow occurs, and no aliasing conflict is detected. This behaviour illustrates Rust's notion of *non-lexical lifetimes*[2]: rather than relying solely on the syntactic scope of a reference, the compiler performs a live-variable analysis to determine the actual points at which references are active. This enables a more precise enforcement of aliasing rules and permits more flexible patterns of reference usage.

Rust's strict aliasing constraints are enforced by the borrow checker, a dataflow analysis that operates over MIR, a control-flow-graph-based intermediate representation within the language's compiler. This analysis tracks the lifetimes of references, the memory locations they may point to, and the initialisation state of variables to ensure that values are never accessed before initialisation or used after invalidation.

[2] https://rust-lang.github.io/rfcs/2094-nll.html

2.3. Lifetimes

To avoid the complexity of whole-program inter-procedural reasoning, Rust requires all references in function signatures to be annotated with explicit *lifetime parameters*[3], which encode relationships between input and output references. These annotations are governed by a set of subtyping rules that are statically enforced by the type checker, ensuring, for instance, that references to local variables cannot escape the function in which they are defined. Crucially, lifetime annotations also provide the borrow checker with sufficient information to reason locally about reference flow across function boundaries. Memory safety guarantees are thus the product of the integration of elaborate type checking with a flow-sensitive analysis.

```
fn f<'a, 'b>(
  a: &'a mut i32,
  b: &'b mut i32
) -> &'a mut i32 {
  *a = *b;
  a
}

fn main() {
  let mut x = 0;
  let mut y = 55;
  let r = f(&mut x, &mut y);
  *r = *r + y;
  println!("{}", *r);
}
```

Listing 4 Lifetime annotations in Rust

Listing 4 presents a Rust program featuring a function annotated with explicit lifetime parameters. The function `f` takes two mutable references to 32-bit integers annotated with distinct lifetimes `'a` and `'b`, and returns a mutable reference annotated with lifetime `'a`. From the function signature alone, the compiler can deduce that the returned reference must originate from the input bound to lifetime `'a`, namely the argument `a`. As a result, when the return value of `f` is assigned to the variable `r` in `main`, the compiler infers that `r` contains a mutable reference to `x`, not `y`. Consequently, it permits the subsequent use of `y`, since no mutable reference to it is live at that point. However, if the program were modified such that `x` were used in the addition instead of `y`, the compiler would reject the code. This is because accessing `x` while a mutable reference to it is still live in `r` violates Rust's aliasing rules, enforced by the borrow checker.

2.4. Rust's Compiler Pipeline

Figure 1 illustrates a simplified view of Rust's compiler pipeline. Compilation begins with a parsing phase, which constructs an initial abstract syntax tree (AST) from the source code. This is immediately followed by a de-sugaring phase that produces the High-level Intermediate Representation (HIR), a structured, de-sugared form of the AST. Subsequently, a combination of trait resolution, type inference, and type checking is performed

[3] In many cases, lifetime annotations may be omitted, as the compiler is able to infer them through *lifetime elision* rules.

on the HIR to ensure that the program is well-formed and free of type errors. During this phase, the compiler also verifies certain ownership and lifetime constraints. The result is the Typed HIR (THIR), which enriches the HIR with type annotations inferred during the analysis. The THIR is then lowered into the Mid-level Intermediate Representation (MIR), a control-flow graph representation of the program that abstracts away much of the surface syntax. At this stage, the borrow checker operates over the MIR to enforce Rust's aliasing and ownership rules, most notably, the single-writer, multiple-readers invariant. It also verifies definite initialisation of variables and performs liveness analysis to enforce Rust's model of non-lexical lifetimes. Finally, assuming the program passes all type and borrow checking phases, the MIR is translated into LLVM IR, which undergoes optimization and code generation to produce the final machine code.

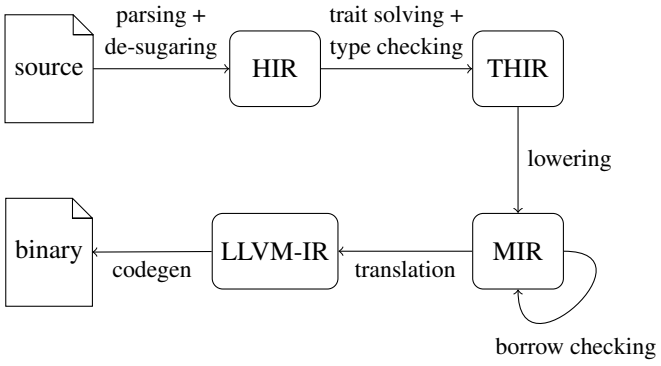


Figure 1 A simplified view of Rust's compiler pipeline

Although Rust's MIR abstracts away much of the surface syntax of the language, it remains deeply intertwined with Rust-specific language features and compiler internals. For example, the MIR representation reuses the types of Rust's THIR for the values it manipulates, and it relies on Rust-specific constructs such as traits to encode aspects of memory management, e.g., whether a value is cloneable or copyable. As a result, adopting Rust's MIR and its borrow checker in the context of a different programming language is highly impractical. The tight coupling between these components and the Rust language design means they are not easily transferable or adaptable to alternative syntaxes, type systems, or runtime semantics. Consequently, compiler developers seeking to integrate ownership and borrowing mechanisms into other languages are often forced to re-implement much of the functionality from scratch, duplicating significant effort already invested in Rust's compiler infrastructure.

3. Borrow Checking as Abstract Interpretation

In the remainder of this paper, we formalise the verification performed by the borrow checker as an instance of abstract interpretation (Cousot & Cousot 1977) applied to a custom source-agnostic intermediate representation inspired by MIR. Within this framework, borrow checking is formalised as a four-parts process. In the first part, a *live variables analysis* determines the set of variables that are live at each program point, enabling the implementation of Rust's non-lexical lifetimes. The second phase performs *alias analysis* to compute the set of memory locations that may be referenced by each variable, to help us detect potential conflicts between mutable and immutable references. Third, an *initialisation analysis* tracks the initialisation status of each memory location, ensuring that immutable variables are assigned at most once and that no variable is accessed before being initialised. Finally, the results of these analyses are combined in an abstract program trace that over-approximates all possible execution behaviours. This trace is checked to ensure that all constraints governing safe memory access and reference usage are respected throughout the function's execution.

3.1. A Simple Example

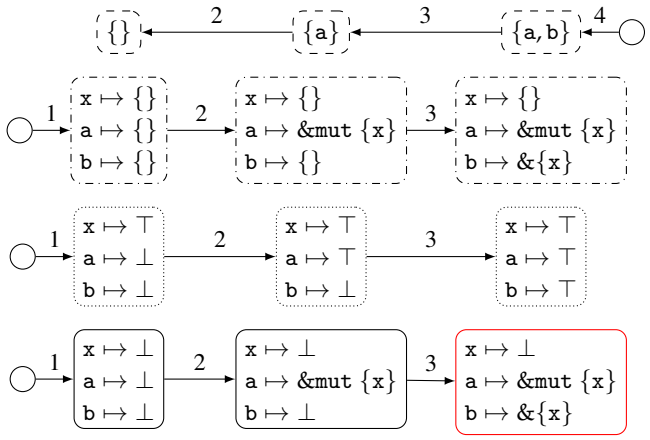
To illustrate this approach concretely, let us analyse the simplest example of Rust program that violates the single-writer, multiple-readers (SWMR) aliasing rule:

```
1 let mut x = 42;
2 let a = &mut x;
3 let b = &x;
4 println!("{}", *a + *b);
```

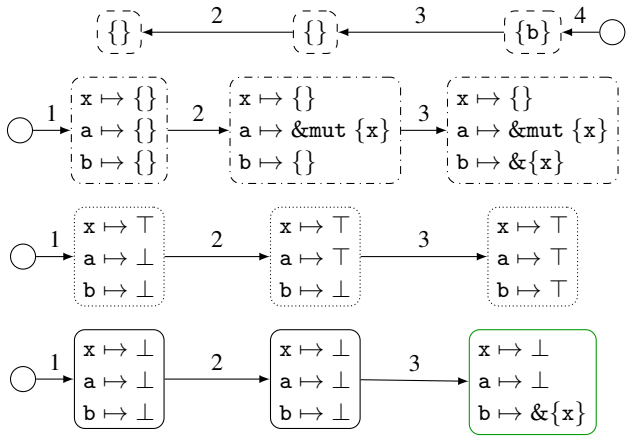
Listing 5 A violation of the SWMR rule

Figure 2a presents the program traces generated by our approach to diagnose the example as invalid. The first sequence of nodes illustrates the result of the live-variable analysis. The trace is represented as a directed graph in which edges – labelled by instruction line numbers – are oriented backward, from the program's end toward its beginning. Each node corresponds to the set of variables that are live immediately before the execution of the instruction identified by the label on its incoming edge. The second and third sequences depict the outcomes of alias analysis and initialization analysis, respectively, both of which proceed forward along the control flow. In the alias analysis trace, nodes map variables to their sets of aliases together with mutability information. Initialisation analysis maps variables to $\top$ when they are initialised, $\bot$ when they are not, and ? when they may be initialised or not. The final sequence integrates the information from all three analyses, yielding an abstract trace that combines liveness, aliasing, and initialization information. In this unified abstract domain, each variable is mapped to the set of places it may reference. A mapping to $\bot$ denotes that the variable is either uninitialized or not live at the corresponding program point, while a $\top$ denotes a variable that is live and initialised but isn't a reference, and ? indicates a variable whose state is unknown.

Initially, following the execution of the first statement, only the variable `x` is initialised. As `x` is an integer value rather than a reference, it is mapped to the empty alias set. Upon executing the second statement, the mutable reference `a` is created, and the domain maps it to the singleton set containing `x`. This aliasing relationship remains live in subsequent states since `a` is used again in the final statement. When the immutable reference `b` is introduced, it is also mapped to `x`, which results in the simultaneous existence of a live mutable and a live immutable reference to the same memory location. The analysis thus detects a violation of the aliasing constraints and reports an error.



(a) Invalid program trace



(b) Valid program trace

Figure 2 Borrow checking with abstract program traces

If the example program is modified such that only the value of `b` is accessed on line 4, the program becomes valid. As illustrated in Figure 2b, this is due to the fact that, at the point where the immutable reference `b` is created (line 3), the mutable reference `a` is no longer live and is thus mapped to $\bot$ in the abstract domain. As a result, no overlapping live references to `x` exist, and the program satisfies Rust's aliasing constraints.

3.2. Refinement of the Abstract Domain

To demonstrate the interest of our approach, we now turn to a more compelling example that incorporates conditional control flow:

```
1  let mut v0: i32;
2  let mut v1: i32;
3  let rf: &mut i32;
4  if random() {
5    rf = &mut v0; v1 = 42;
6  } else {
7    rf = &mut v1; v0 = 1337;
8  }
9  *rf = 23;
10 println!("{:?}", v0 + v1);
```

Listing 6 An example with conditional control flow

In this program, depending on the test at line 4, `rf` (a mutable reference to an integer) is assigned to `v0`'s address and `v1` to `42`, or `rf` is assigned to `v1`'s address and `v0` to `1337`. Regardless, the memory pointed by `rf` is assigned the value `23` at line 9 and the program eventually prints the sum of both integer variables at line 10.

Although the program does not violate any aliasing constraints, it is nonetheless rejected by Rust's current borrow checker implementation due to its restriction against creating references to uninitialised variables. Specifically, when `rf` is assigned the value of either `v0` or `v1` on lines 5 and 7, the compiler emits an error, as neither variable has been set at that point.

The cause for this restriction becomes particularly evident when examined through the lens of our framework. Figure 3 presents the final trace computed by our formulation of the borrow checker throughout the program. The key observation lies in the state at the exit of the conditional construct: because control may flow through either branch, the analysis must conservatively merge the abstract information from both paths. The compiler thus over-approximates by computing the union of alias sets for references and by treating variables that are uninitialised in any branch as maybe initialised in the resulting state. This behaviour is captured by the abstract domain used in our analysis: following the execution of line 8, the reference `rf` is mapped to the set `&mut{v0, v1}`, while both `v0` and `v1` are assigned ?, indicating that they may be uninitialised in some execution paths.

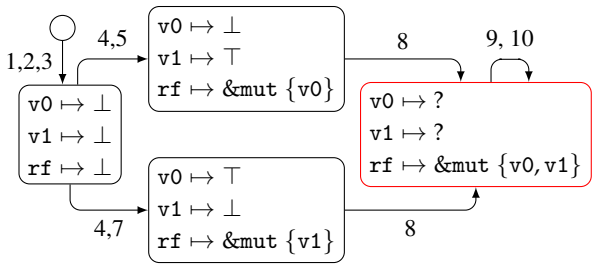


Figure 3 Borrow checking control flow constructs

Careful examination of the code reveals that the borrow checker's behaviour is overly conservative in this instance. In practice, both `v0` and `v1` are guaranteed to be valid by the time control reaches line 10, making it safe to access their values. Specifically, if the first branch of the conditional is executed, `v1` is initialised on line 5, and `v0` subsequently receives a value via `rf` on line 9. Conversely, if the second branch is taken, `v0` is initialised on line 7, and `v1` is assigned through `rf`. Thus, regardless of the control path, both variables are guaranteed to be fully initialised prior to their use.

To increase the precision of the borrow checker and allow it to accept programs exhibiting such flow-sensitive patterns, we propose a refinement of the underlying abstract domain used in the analysis. Rather than mapping variables to sets of potentially referenced locations, we represent abstract states as

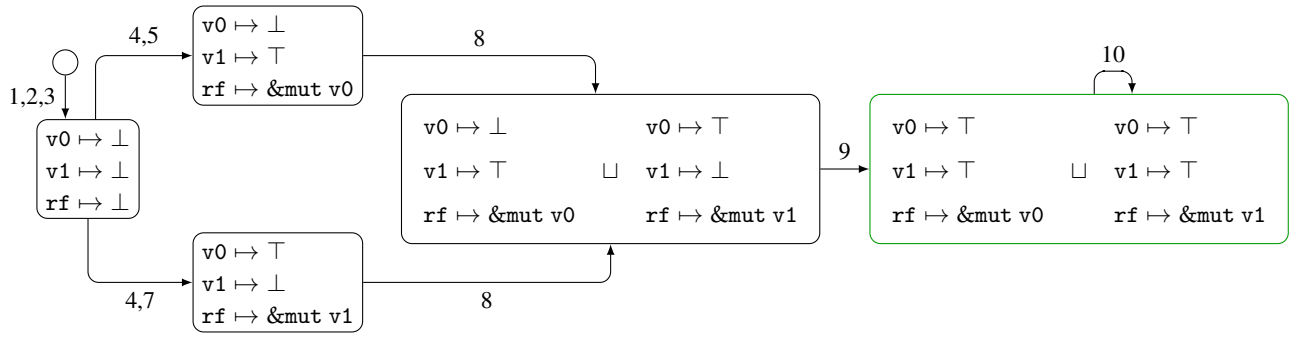


Figure 4 Borrow checking with a more precise abstract domain

sets of environments, where each environment maps variables to the unique memory location they reference. This representation allows the analysis to preserve path-sensitive information that would otherwise be lost during control-flow joins.

Figure 4 illustrates this refinement: after the conditional expression in our example program, instead of merging environments by taking the union of alias sets, we maintain the union of both path-specific environments. Consequently, when `rf` is dereferenced and used to update a memory location on line 9, the analysis reflects the fact that, along each path, a distinct variable is being updated. This increased precision allows the analysis to prove the program as memory-safe and accept it.

Listing 7 illustrates another well-known example of a function that the Rust compiler rejects under its current borrow checking implementation, despite the program being semantically correct (Matsakis 2020).

```
fn get_insert<'a>(
  map: &'a mut HashMap<u32, String>
) -> &'a String {
  match HashMap::get(&*map, &0) {
    Some(v) => v,
    None => {
      map.insert(0, String::from("default"));
      &map[&0]
    }
  }
}
```

Listing 7 A well-known example of correct code rejected by Rust's current borrow checker

The rejection arises from a flow-sensitive borrowing pattern that the compiler's conservative analysis fails to validate. At the entry of the `match` expression, an immutable reference to the `map` parameter is created. An immutable borrow of one of `map`'s elements is then derived from this reference and returned in the first branch. The borrow checker thus consequently treats the initial immutable reference as live for the entire scope of the `match` expression. As a result, when a mutable reference to `map` is introduced in the second branch to update the hash map, the compiler reports a violation of the single-writer, multiple-readers rule.

In contrast, our path-sensitive formulation of borrow checking assigns distinct environments to the two branches, without merging them at the control-flow join. This separation allows the analysis to establish that the immutable reference returned in the first branch does not interfere with the mutable reference in the second. Consequently, the program can be validated as correct.

3.3. Intra-procedural Reasoning

To keep the borrow checker for our IR strictly intra-procedural, we rely on a similar model as the Rust compiler. Function signatures need to be annotated with *origin annotations* that indicate which references passed as argument to a function can be returned. These origins are governed by subtyping rules enforced by a type checker that is run on the IR before borrow checking. If the IR type checks, the analysis can proceed and treat the annotations as indicators of which references a function call might return.

Figure 5 shows the effect of origin annotations on the analysis of the following program:

```
fn f<'m, 'n>(
  a: &'m i32,
  b: &'n i32,
  c: &'m i32
) -> &'m cst i32 {
  // Some code...
}

let x = 1;
let y = 2;
let z = 3;
let r = f(&x, &y, &z);
println!("{:?}", *r);
```

Listing 8 A function call with origin annotations

Following the function call on line 12, the reference `r` may alias either `x` or `z`, but not `y`, as the origin `'n` associated with `y` is incompatible with the origin annotation of the function's return type. This constraint allows the borrow checker to refine its analysis by narrowing the set of possible aliasing environments that may hold after the call.

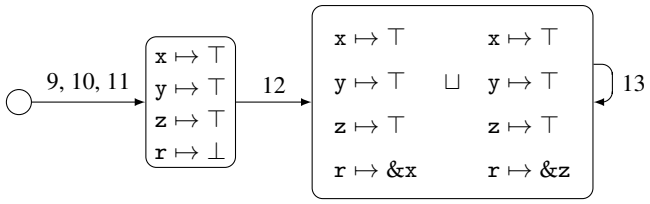


Figure 5 Borrow checking function calls

3.4. Proposed Compiler Architecture

A summary of the overall architecture of our proposed integration of borrow checking into compiler infrastructures is depicted in Figure 6. The figure illustrates the intended design of our IR as a unified intermediate representation serving as a common target for multiple compiler frontends. Rather than maintaining separate IRs and bespoke borrow-checking implementations, multiple languages could instead share a common infrastructure for ownership tracking and memory safety enforcement through a single intermediate language – which we call MIMIR – that would perform its own type and memory safety checks with the help of our abstract interpretation based formulation.

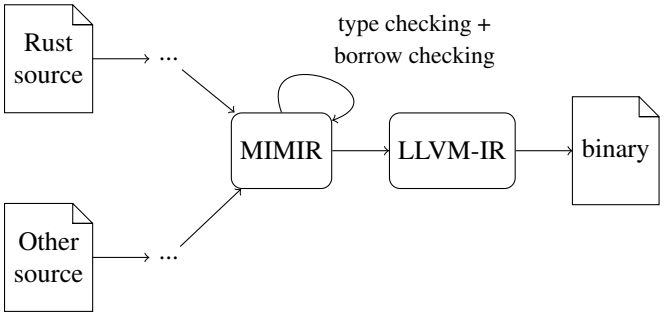


Figure 6 Proposed compiler pipeline of our approach

4. A Compiler IR for Borrow Checking

This section introduces the syntax and formal semantics of the intermediate representation employed in our borrow checking framework, the Memory Inspection Mid-level Intermediate Representation (MIMIR). The language is intentionally designed to capture the minimal set of constructs necessary to model borrow checking, balancing expressiveness with simplicity to facilitate the formalisation of our abstract interpretation-based analysis. In consequence, our design omits various features that would be essential in a production-grade compiler IR. In particular, MIMIR only models types with copy semantics and excludes constructs such as move assignments, thereby simplifying the formulation of its type system. While the extension of MIMIR to accommodate more advanced language features is a promising direction for future work, it falls outside the scope of this paper. Nonetheless, to demonstrate the practicality of our design, we have developed a prototype implementation of the IR within a simple compiler that can translate a small subset of Rust into MIMIR[4].

4.1. Syntax

Figure 7 describes the abstract syntax of MIMIR. Elements denoted by $\overline{t}$ in the syntax represent (possibly empty) sequences of terms $t_1, t_2 \ldots, t_n$ of length $n \in \mathbb{N}$. We write $\overline{a} \cdot \overline{b}$ to denote the concatenation of two sequences $\overline{a}$ and $\overline{b}$ of arbitrary length.

A program in the IR consists of a sequence of function definitions, one of which serves as its designated entry point. In line with Rust's convention, we assume this entry point is a function named `main` that takes no arguments.

[4] https://github.com/coetaur0/inox

Origin name	$r \in R$		*Variable name* $x \in X$
Function name	$f \in F$		*Integer number* $j \in \mathbb{Z}$
Module	$m \in M$	::=	$\overline{d}$
Declaration	$d \in D$	::=	`fn` f<$\overline{r}$>$(\overline{l}) \to t$ { $\overline{l'}$ i }
Local	$l \in L$	::=	$c\ x : t$
Type	$t \in T$	::=	
Function		|	$(\overline{t}) \to t$
Reference		|	&r c t
Integer		|	`Int`
Boolean		|	`Bool`
Capability	$c \in C$	::=	`mut` | `cst`
Instruction	$i \in I$	::=	
Loop		|	`while` o { i }
Conditional		|	`if` o { i } `else` { i' }
Sequence		|	$i; i'$
Call		|	$p = o(\overline{o})$
Borrow		|	p = &c p'
Assign		|	$p = o$
Return		|	`return`
Operand	$o \in O$	::=	
Place		|	p
Function		|	f<$\overline{r}$>
Integer		|	j
Boolean		|	`true` | `false`
Place	$p \in P$	::=	
Dereference		|	$*p$
Variable		|	x

Figure 7 MIMIR abstract syntax

Functions in the IR are declared using the `fn` keyword, followed by the function name, a sequence of origin declarations, a type signature, and a function body. The type signature defines the function's input parameters and return type. The body begins with declarations of local variables, followed by an instruction specifying the function's behaviour. Each parameter and local variable has a unique identifier and is annotated with a mutability qualifier and a type. Additionally, each function implicitly defines a special variable named `ret`, which holds its return value.

```
fn f<'a>(a: &'a mut Int, b: Bool) -> Int {
  let mut x: Int;
  let mut y: Int;
  // Some code...
}
```

Listing 9 A function declaration in MIMIR

Listing 9 presents an example of a function declaration in MIMIR. The function, named `f`, takes two parameters as input: `a`, a mutable reference to an integer with a generic origin `'a`, and `b`, a boolean. It returns an integer. Inside the function's body, two mutable local variables of type `Int` – `x` and `y` – are

declared. In addition, a special variable named `ret` of type `Int` is implicitly in scope. Assigning a value to `ret` and reaching the end of the function body is semantically equivalent to returning that value.

MIMIR supports four categories of types:

- `Int`, representing (32-bit) integers.
- `Bool`, for boolean values.
- Function types, written as a sequence of parameter types enclosed in parentheses, followed by an arrow and a return type (e.g., `(Int, Int) -> Int`).
- Reference types, denoted by a leading `&`, followed by an origin annotation and a mutability qualifier (or *capability*): either `cst` for immutable references, or `mut` for mutable ones.

Origin annotations are used to express relationships between references in function signatures. Listing 10 shows a function that takes two references with distinct origins, `'a` and `'b`, as input. Because the return type is annotated with origin `'a`, it is immediately evident from the signature that only the reference bound to `'a` may be returned.

```
fn first<'a, 'b>(
  a: &'a Int,
  b: &'b Int
) -> &'a Int {
  // Some code...
}
```

Listing 10 Origin annotations in MIMIR

A special origin, written as `'_`, denotes references confined to the function's local scope. This origin is automatically assigned to references to local variables created within a function body, and such references can never be returned, as they would necessarily become dangling upon function exit.

The instruction set of MIMIR includes both control flow constructs, such as conditionals and loops, and memory-related operations, such as assignments to local variables, function calls, and borrowing. Unlike other intermediate representations such as Rust's MIR, MIMIR models control flow using structured syntactic constructs rather than basic blocks and labelled jumps. This design choice is deliberate: it simplifies the formalisation of both the type system and the abstract interpreter, while preserving an equivalent level of expressiveness.

By default, the execution of a function concludes after the final instruction is executed, with the value held in the special local variable `ret` returned as the result. Alternatively, the `return` instruction can be used to exit the function prematurely.

Operands in MIMIR represent the valid arguments to instructions and include literal values (integers, booleans, and functions) as well as place expressions, which denote paths to memory locations. Function operands are written as a function name followed by a sequence of origin arguments, indicating instantiation for a particular set of origins. Place expressions encompass variable identifiers and dereference operations.

4.2. Dynamic Semantics

We now present the formal semantics of MIMIR. These are defined using inference rules in the style of natural semantics (Kahn 1987) and describe the behaviour of MIMIR instructions in terms of their effects on an *environment*. This environment comprises three components: a *module*, which represents the program being executed; a *context*, which is a partial function mapping variable names within the current function scope to their corresponding *memory addresses*; and a *memory*, which maps those addresses to concrete *values*. The notation $A \rightharpoonup B$ is used to denote a *partial* function with *domain* A and *co-domain* B. Values may include function identifiers, memory addresses, integers, booleans, or the special value $\bot$, representing an uninitialised memory location.

$$
\begin{array}{llcl}
\textit{Environment} & \epsilon \in \mathcal{E} & = & M \times \mathcal{K} \times \mathcal{M} \\
\textit{Context} & \kappa \in \mathcal{K} & = & X \rightharpoonup \mathcal{A} \\
\textit{Memory} & \mu \in \mathcal{M} & = & \mathcal{A} \rightharpoonup \mathcal{V} \\
\textit{Value} & \nu \in \mathcal{V} & = & F \cup \mathcal{A} \cup \mathbb{Z} \cup \{\texttt{true}, \texttt{false}, \bot\} \\
\textit{Address} & \alpha \in \mathcal{A} & &
\end{array}
$$

Instruction evaluation is defined as a relation that maps an environment and an instruction to a new memory state. Note that this relation is partial and undefined for programs that contain errors related to memory access.

$$\_ \vdash \_ \Downarrow \_ : \mathcal{E} \times I \rightharpoonup \mathcal{M}$$

The semantics for control-flow constructs are standard: the condition is evaluated and the body is executed or not based on its outcome.

$$\textsc{E-while-true}\quad \frac{\epsilon \vdash o \Downarrow_O \texttt{true} \qquad \epsilon \vdash i; \texttt{while}\ o\ \{\ i\ \} \Downarrow \mu}{\epsilon \vdash \texttt{while}\ o\ \{\ i\ \} \Downarrow \mu}$$

$$\textsc{E-while-false}\quad \frac{<m,\kappa,\mu> \vdash o \Downarrow_O \texttt{false}}{<m,\kappa,\mu> \vdash \texttt{while}\ o\ \{\ i\ \} \Downarrow \mu}$$

$$\textsc{E-if-true}\quad \frac{\epsilon \vdash o \Downarrow_O \texttt{true} \qquad \epsilon \vdash i \Downarrow \mu}{\epsilon \vdash \texttt{if}\ \{\ i\ \}\ \texttt{else}\ \{\ i'\ \} \Downarrow \mu}$$

$$\textsc{E-if-false}\quad \frac{\epsilon \vdash o \Downarrow_O \texttt{false} \qquad \epsilon \vdash i' \Downarrow \mu}{\epsilon \vdash \texttt{if}\ \{\ i\ \}\ \texttt{else}\ \{\ i'\ \} \Downarrow \mu}$$

These rules depend on the evaluation of instruction operands, which is formalised as a relation that maps an environment and an operand to the corresponding value: $\_ \vdash \_ \Downarrow_O \_ : \mathcal{E} \times O \rightharpoonup \mathcal{V}$.

$$\textsc{E-place}\quad \frac{<m,\kappa,\mu> \vdash p \Downarrow_P \alpha \qquad \mu(\alpha) = \nu}{<m,\kappa,\mu> \vdash p \Downarrow_O \nu} \qquad \textsc{E-function}\quad \frac{}{\epsilon \vdash f<\bar{r}> \Downarrow_O f}$$

$$\textsc{E-integer}\quad \frac{}{\epsilon \vdash j \Downarrow_O j} \qquad \textsc{E-true}\quad \frac{}{\epsilon \vdash \texttt{true} \Downarrow_O \texttt{true}} \qquad \textsc{E-false}\quad \frac{}{\epsilon \vdash \texttt{false} \Downarrow_O \texttt{false}}$$

The rules for operand evaluation rely in turn on the rules for the evaluation of places, which map an environment and a place expression to its corresponding memory address: $\_ \vdash \_ \Downarrow_P \_ : \mathcal{E} \times P \rightharpoonup \mathcal{A}$.

E-DEREFERENCE
$$\frac{<m, \kappa, \mu> \vdash p \Downarrow_P \alpha \qquad \mu(\alpha) = \alpha'}{<m, \kappa, \mu> \vdash *p \Downarrow_P \alpha'}$$

E-VARIABLE
$$\frac{\kappa(x) = \alpha}{<m, \kappa, \mu> \vdash x \Downarrow_P \alpha}$$

Returning to the evaluation of instructions, the semantics of instruction sequences are straightforward: two instructions are executed in order, with the memory resulting from the first used as the input to the second. If the first instruction is a `return`, the second is skipped entirely. Moreover, a `return` instruction on its own has no operational effect beyond signalling the end of execution.

E-SEQUENCE
$$\frac{i \neq \texttt{return} \qquad <m, \kappa, \mu> \vdash i \Downarrow \mu' \qquad <m, \kappa, \mu'> \vdash i' \Downarrow \mu''}{<m, \kappa, \mu> \vdash i; i' \Downarrow \mu''}$$

E-RETURN-SEQUENCE
$$\frac{}{<m, \kappa, \mu> \vdash \texttt{return}; i \Downarrow \mu}$$

E-RETURN
$$\frac{}{<m, \kappa, \mu> \vdash \texttt{return} \Downarrow \mu}$$

The evaluation rules for borrow and assignment operations specify how memory is updated at the address designated by a place expression. For an assignment, the address of the target place is first computed, and the value stored at that address is then replaced with the value of the right-hand side. In the case of a borrow, the stored value is itself an address, representing the reference being created.

The notation $\mu[\alpha \mapsto \nu]$ denotes a new partial function identical to μ except at α. If α is already in the domain of μ, its value is updated to ν; otherwise, the domain of μ is extended with a fresh mapping from α to ν.

E-BORROW
$$\frac{<m, \kappa, \mu> \vdash p \Downarrow_P \alpha \qquad <m, \kappa, \mu> \vdash p' \Downarrow_P \alpha' \qquad \mu' = \mu[\alpha \mapsto \alpha']}{<m, \kappa, \mu> \vdash p = \&c\ p' \Downarrow \mu'}$$

E-ASSIGN
$$\frac{<m, \kappa, \mu> \vdash p \Downarrow_P \alpha \qquad <m, \kappa, \mu> \vdash o \Downarrow_O \nu \qquad \mu' = \mu[\alpha \mapsto \nu]}{<m, \kappa, \mu> \vdash p = o \Downarrow \mu'}$$

Finally, the rule for function calls encapsulates the mechanisms for both the allocation of a new context for the function being called, and the extension of the memory with new cells for the function's parameters and local variables.

E-CALL
$$\frac{\begin{array}{c}<m, \kappa, \mu> \vdash o \Downarrow_O f \\ m(f) = \texttt{fn}\ f<\overline{r}>(\overline{c\ x : t}) \rightarrow t\ \{\ \overline{\texttt{let}\ c'\ x' : t'}\ i\ \} \\ \forall o_i \in \overline{o},\ <m, \kappa, \mu> \vdash o_i \Downarrow_O \nu_i \\ \text{alloc}(\mu, \overline{x = \nu} \cdot \overline{x' = \bot} \cdot \texttt{ret} = \bot) = <\kappa', \mu'> \\ <m, \kappa', \mu'> \vdash i \Downarrow \mu'' \\ <m, \kappa, \mu> \vdash p \Downarrow_P \alpha \qquad \mu''' = \mu''[\alpha \mapsto \mu''(\kappa'(ret))]\end{array}}{<m, \kappa, \mu> \vdash p = o(\overline{o}) \Downarrow \mu'''}$$

In this rule, the place denoting the callee is first evaluated and must resolve to a function identifier. The corresponding function definition is then retrieved from the module m currently being executed. Next, the call's argument expressions are evaluated to obtain their values. The *alloc* predicate is then applied to extend the current memory μ with fresh cells for each function parameter, each local variable, and the special `ret` variable. The parameters are initialised in the new memory with the values computed for them, whereas all local variables and `ret` are set to $\bot$, indicating that they are initially undefined. This allocation step yields the extended memory μ' together with a new context κ', which maps every parameter and local variable to the address allocated for it in μ'. Finally, the function body i is executed in the new environment $<m, \kappa', \mu'>$, and the value stored at the address of `ret` in the callee's context is written to the address of p, the target of the call instruction.

The rule for function calls relies on an *alloc* predicate which updates the current memory and returns a new context:

ALLOC-EMPTY
$$\frac{}{\text{alloc}(\mu, \cdot) = <[], \mu>}$$

ALLOC-SEQ
$$\frac{\text{alloc}(\mu, \overline{x = \nu}) = <\kappa, \mu'> \qquad \alpha \notin dom(\mu') \qquad \kappa' = \kappa[x \mapsto \alpha] \qquad \mu'' = \mu'[\alpha \mapsto \nu]}{\text{alloc}(\mu, x = \nu \cdot \overline{x = \nu}) = <\kappa', \mu''>}$$

In rule ALLOC-EMPTY, which represents the base case of the *alloc* predicate, the symbol $\cdot$ without elements to its left and right denotes an empty sequence of variable-to-value assignments. Rule ALLOC-SEQ specifies how, in the inductive case, the context and memory for a function call are extended: the context is augmented with a new mapping from a variable to a new, previously undefined address, and the memory is updated with a corresponding mapping from that address to its initial value.

5. Type System

MIMIR's type system is designed to ensure that programs expressed in the IR are well-formed and that reference origin annotations are used correctly. In particular, the typing rules governing origin annotations guarantee that functions cannot return references to local values, thereby preventing use-after-free errors. These guarantees form the foundation for the borrow-checking algorithm which we will introduce in the next chapter.

5.1. Typing Rules

The formal rules of MIMIR's type system operate over a *context* that maps variable identifiers to their type, in order to capture the local environment within each function definition:

$$\textit{Context} \quad \gamma \in \Gamma \quad = \quad X \rightharpoonup T$$

For a module to be considered well-formed, each of its function definitions must satisfy the type-checking rule presented below.

T-FUNCTION
$$\frac{\begin{array}{c}\forall t_i \in \overline{t},\ \overline{r} \vdash t_i,\ \neg local(t_i) \\ \overline{r} \vdash t \qquad \neg local(t) \qquad \forall t'_i \in \overline{t'},\ \overline{r} \vdash t'_i \\ \gamma = [\overline{x \mapsto t}][\overline{x' \mapsto t'}][\texttt{ret} \mapsto t] \qquad m; \overline{r}; \gamma \vdash i\end{array}}{m \vdash \texttt{fn}\ f \texttt{<}\overline{r}\texttt{>}(\overline{c\ x : t}) \to t\ \{\ \overline{c'\ x' : t'}\ i\ \}}$$

This rule incorporates the predicate $\neg local$, which recursively ensures that no reference type within the function's signature is annotated with a local origin.

$$\frac{\forall t_i \in \overline{t},\ \neg local(t_i) \qquad \neg local(t)}{\neg local((\overline{t}) \to t)}$$

$$\frac{r \neq \text{'\_} \qquad \neg local(t)}{\neg local(\&r\ c\ t)} \qquad \frac{}{\neg local(\texttt{Int})} \qquad \frac{}{\neg local(\texttt{Bool})}$$

Rules of the form $\overline{r} \vdash t$ specify that a type t is well-formed with respect to a given sequence of origin declarations $\overline{r}$. These rules ensure, in particular, that all origin annotations appearing in the type are properly declared.

T-FN-TY
$$\frac{\forall t_i \in \overline{t},\ \overline{r} \vdash t_i \qquad \overline{r} \vdash t}{\overline{r} \vdash (\overline{t}) \to t}$$

T-REF-TY
$$\frac{r = \text{'\_} \vee r \in \overline{r} \qquad \overline{r} \vdash t}{\overline{r} \vdash \&r\ c\ t}$$

T-INT-TY
$$\frac{}{\overline{r} \vdash \texttt{Int}}$$

T-BOOL-TY
$$\frac{}{\overline{r} \vdash \texttt{Bool}}$$

The typing rules for control-flow constructs require that the condition operand evaluates to a boolean and recursively verify the well-typedness of each instruction in the corresponding branch. For sequences, the rule simply checks that both consecutive instructions are well-typed.

T-IF
$$\frac{m; \overline{r}; \gamma \vdash o : \texttt{Bool} \qquad m; \overline{r}; \gamma \vdash i \qquad m; \overline{r}; \gamma \vdash i'}{m; \overline{r}; \gamma \vdash \texttt{if}\ o\ \{\ i\ \}\ \texttt{else}\ \{\ i'\ \}}$$

T-WHILE
$$\frac{m; \overline{r}; \gamma \vdash o : \texttt{Bool} \qquad m; \overline{r}; \gamma \vdash i}{m; \overline{r}; \gamma \vdash \texttt{while}\ o\ \{\ i\ \}}$$

T-SEQ
$$\frac{m; \overline{r}; \gamma \vdash i \qquad m; \overline{r}; \gamma \vdash i'}{m; \overline{r}; \gamma \vdash i; i'}$$

The most critical rules in the type system govern the assignment of values to places within a program. The typing rule for function calls ensures that the callee is a valid function, that the number and types of the arguments conform to the function's declared signature, and that the return type is a subtype of the type expected by the target operand. For borrow instructions, the type system enforces that a local reference to the type of place p' be a subtype of the type of place p. This constraint guarantees that references with local origins cannot be assigned to references with external origins.

T-CALL
$$\frac{\begin{array}{c}m; \overline{r}; \gamma \vdash p : t \qquad m; \overline{r}; \gamma \vdash o : (\overline{t'}) \to t' \\ |\overline{o}| = |\overline{t'}| \qquad \forall o_i \in \overline{o},\ m; \overline{r}; \gamma \vdash o_i : t_i,\ t_i <: t'_i \qquad t' <: t\end{array}}{m; \overline{r}; \gamma \vdash p = o(\overline{o})}$$

T-BORROW
$$\frac{m; \overline{r}; \gamma \vdash p : t \qquad m; \overline{r}; \gamma \vdash p' : t' \qquad \&'\_\ c\ t' <: t}{m; \overline{r}; \gamma \vdash p = \&c\ p'}$$

T-ASSIGN
$$\frac{m; \overline{r}; \gamma \vdash p : t \qquad m; \overline{r}; \gamma \vdash o : t' \qquad t' <: t}{m; \overline{r}; \gamma \vdash p = o}$$

Subtyping rules specify which types may be substituted for others in a type-safe manner. In particular, the subtyping rule for reference types enforces that references annotated with non-local origins may be assigned to both similarly annotated references and local references, whereas local references may not be assigned to references with broader lifetimes. The rule for function types adheres to the standard variance principles: parameter types must be contravariant, and return types covariant.

S-FN-TY
$$\frac{|\overline{t}| = |\overline{t'}| \qquad \forall t'_i \in \overline{t'},\ t'_i <: t_i \qquad t <: t'}{(\overline{t}) \to t <: (\overline{t'}) \to t'}$$

S-REF-TY
$$\frac{r <: r' \qquad c <: c' \qquad t <: t'}{\&r\ c\ t <: \&r'\ c'\ t'}$$

S-TRANSITIVE
$$\frac{t <: t' \qquad t' <: t''}{t <: t''}$$

S-REFLEXIVE
$$\frac{}{t <: t}$$

SO-LOCAL
$$\frac{}{r <: \text{'\_}}$$

SO-REFLEXIVE
$$\frac{}{r <: r}$$

SC-MUT
$$\frac{}{\texttt{mut} <: cst}$$

SC-REFLEXIVE
$$\frac{}{c <: c}$$

The rules for typing instruction operands follow standard conventions, assigning types to operands based on the current context. Of particular interest among them is the rule for function instantiation. When a function identifier is encountered with a sequence of origin arguments, the rule first verifies that each origin is well-formed and declared in the current context. The corresponding function definition is then retrieved from the module, and its type signature is extracted. Origin parameters within the signature are substituted with the provided arguments (denoted by $t[\overline{r''}/\overline{r'}]$ below) and the resulting type returned.

T-FN

$$\frac{\forall r'_i \in \overline{r'},\ r'_i = {}'\_ \vee r'_i \in \overline{r} \qquad m(f) = \texttt{fn}\ f{<}\overline{r''}{>}(\overline{c\ x : t}) \to t\ \{\ \overline{\texttt{let}\ c'\ x' : t'\ i}\ \} \qquad |\overline{r'}| = |\overline{r''}| \qquad \forall t_i \in \overline{t},\ t'_i = t_i[\overline{r''}/\overline{r'}] \qquad t' = t[\overline{r''}/\overline{r'}]}{m;\overline{r};\gamma \vdash f{<}\overline{r'}{>} : (\overline{t'}) \to t'}$$

T-INTEGER

$$\frac{}{m;\overline{r};\gamma \vdash j : \texttt{Int}}$$

T-TRUE

$$\frac{}{m;\overline{r};\gamma \vdash \texttt{true} : \texttt{Bool}}$$

T-FALSE

$$\frac{}{m;\overline{r};\gamma \vdash \texttt{false} : \texttt{Bool}}$$

T-DEREF

$$\frac{m;\overline{r};\gamma \vdash p : \&r\ c\ t}{m;\overline{r};\gamma \vdash *p : t}$$

T-LOCAL

$$\frac{\gamma(x) = t}{m;\overline{r};\gamma \vdash x : t}$$

5.2. Type Safety

The soundness of our type system ensures that well-typed programs preserve type safety during execution, specifically by preventing the assignment of values to memory locations in a manner that violates their declared types. In particular, it guarantees that references to local values cannot escape the scope of a function, and that only references annotated with an origin compatible with the function's return type may be returned. This invariant is critical to the design of our abstract interpreter, as it allows intra-procedural reasoning to be based solely on function signatures, eliminating the need to inspect the bodies of callees when analysing program behaviour across function boundaries.

Since the dynamic semantics of MIMIR is defined using big-step inference rules, the soundness proof of our type system departs from the standard *progress and preservation* methodology (Wright & Felleisen 1994). Instead, we follow the approach based on *definitional interpreters* introduced by Amin et al. (Amin & Rompf 2017). For conciseness, and given that the argument closely follows established patterns, we omit the full proof here and direct the reader to the cited work, which provides a detailed presentation of the technique and its underlying intuition.

6. Abstract Interpreter

The following section formalizes the application of abstract interpretation to borrow checking. As is customary in the framework, our abstract interpreter is systematically derived from the concrete semantics of MIMIR by introducing an abstraction of program states and redefining the evaluation rules to operate over these abstract states rather than concrete values. This enables static reasoning about program behaviour while preserving soundness with respect to the underlying dynamic semantics.

In our abstract interpretation based formulation of borrow checking, the analysis of a function declaration is modelled as a multi-stage process. First, *liveness analysis* is performed over the function body to determine the set of variables that are live at each program point, enabling the implementation of Rust's non-lexical lifetimes. Second, *alias analysis* computes the memory locations that every variable may reference after each instruction, to help us detect potential conflicts between mutable and immutable references. Third, an *initialisation analysis* tracks the initialisation status of each memory location, ensuring that immutable variables are assigned at most once and that no variable is accessed before being initialised. Finally, the results of these analyses are combined in an abstract program trace that over-approximates all possible execution behaviours. This trace is checked to ensure that all constraints governing safe memory access and reference usage are respected throughout the function's execution.

6.1. Live Variable Analysis

The first phase of our analysis computes the set of variables that are live at each program point. Following standard practice, the abstract states in our liveness analysis, which we call *live sets*, are represented as elements of the *powerset* lattice $(\mathcal{P}(X_f), \subseteq)$, where X_f denotes the set of all variables occurring syntactically in a function f. The lattice, ordered by subset inclusion $\subseteq$, is necessarily finite and of bounded height because the set X_f is always finite for a given f.

$$\textit{Live set} \quad \lambda \in \Lambda \quad = \quad \mathcal{P}(X_f)$$

Liveness analysis proceeds backwards over a function's representation to determine which memory locations will be used later by instructions. Formally, the abstract semantics are defined by the relation

$$\_ \vdash \_ \uparrow \_ : \Lambda \times I \rightharpoonup \Lambda$$

whose inference rules traverse functions in reverse order, starting from the end and moving toward the entry point.

To handle branching control flow, the rule for conditional instructions computes the liveness sets independently within each branch and merges the resulting states using set union. Any variable live in either branch is thus considered live at the instruction's entry.

L-IF

$$\frac{\lambda \vdash i \uparrow \lambda' \qquad \lambda \vdash i' \uparrow \lambda'' \qquad \lambda''' = \lambda \cup \lambda' \cup \mathit{var}(o)}{\lambda \vdash \texttt{if}\ o\ \{\ i\ \}\ \texttt{else}\ \{\ i'\ \} \uparrow \lambda'''}$$

The *var* predicate used in the rule's premise extracts the set of variables that occur within a given operand; these are added to the live set prior to the conditional instruction to account for their usage in the condition expression.

$$\frac{o \notin p}{\mathit{var}(o) = \emptyset} \qquad \frac{\mathit{var}(p) = \lambda}{\mathit{var}(*p) = \lambda} \qquad \frac{}{\mathit{var}(x) = \{x\}}$$

For while loops, the analysis computes a fixed point to determine the set of live variables at the loop's entry, in order to soundly account for all possible iterations.

L-WHILE
$$\frac{\lambda_0 = \lambda \qquad \lambda' = \bigcup_{j\geq 1}\Big(\lambda_j \cup var(o)\Big) \text{ where } (\lambda_{j-1} \vdash i \uparrow \lambda_j)}{\lambda \vdash \texttt{while}\ o\ \{\ i\ \} \uparrow \lambda'}$$

Let λ_j denote the set of variables live at the loop entry after the j-th iteration of the computation. The rule expresses this as

$$\bigcup_{j\geq 1}\Big(\lambda_j \cup var(o)\Big) \text{ where } (\lambda_{j-1} \vdash i \uparrow \lambda_j)$$

Starting from an initial approximation λ_0, liveness is propagated backwards through the loop body, and the result is iteratively updated until it stabilises. Because the set of variables in a function is finite and set union is a monotone operator, each iteration can only preserve or increase the set of live variables. Thus, the computation is guaranteed to terminate after at most $|X_f|$ iterations; in the worst case, the fixed point is reached when every variable is considered live at the loop's entry.

Instructions that assign a value to a place adhere to the standard *gen* and *kill* semantics of liveness analysis: the target variable of the assignment is removed from the set of live variables prior to the instruction, while any variables appearing on the right-hand side are added to it.

L-BORROW
$$\frac{\lambda' = \lambda \setminus var(p) \cup var(p')}{\lambda \vdash p = \&c\ p' \uparrow \lambda'}$$

L-ASSIGN
$$\frac{\lambda' = \lambda \setminus var(p) \cup var(o)}{\lambda \vdash p = o \uparrow \lambda'}$$

L-CALL
$$\frac{\lambda' = \lambda \setminus var(p) \cup var(o) \cup \Big(\bigcup_{o_j \in \overline{o}} var(o_j)\Big)}{\lambda \vdash p = o(\overline{o}) \uparrow \lambda'}$$

Finally, the rules for sequences and return instructions are straightforward. For a sequence, liveness is propagated backwards by first analysing the second instruction to obtain its live-variable set, which is then used as the input for the first instruction. The output of this first instruction becomes the set of variables that are live before the sequence. For a return instruction, the special variable `ret` is simply added to the live set, since its value is read when the function terminates.

L-SEQ
$$\frac{\lambda \vdash i' \uparrow \lambda' \qquad \lambda' \vdash i \uparrow \lambda''}{\lambda \vdash i; i' \uparrow \lambda''}$$

L-RETURN
$$\frac{\lambda' = \lambda \cup \{\texttt{ret}\}}{\lambda \vdash \texttt{return} \uparrow \lambda'}$$

6.2. Alias and Initialisation Analysis

Once liveness information has been computed, the analysis of aliasing and initialisation information can proceed. Although Section 3 presented these two phases as separate processes, in practice they are carried out simultaneously in a forward traversal of the program.

The abstract domain for the analysis is defined as the powerset $\mathcal{P}(\Psi)$ of functions $\psi \in \Psi$ that map the variables in a function to the reference they may contain in a specific environment. The set of possible values for a variable in Ψ is extended with the elements $\bot$ and $\top$ to denote uninitialised variables and initialised variables that do not contain references, respectively.

$$\begin{array}{lrcl} \textit{State} & \sigma \in \Sigma & = & \mathcal{P}(\Psi) \\ \textit{Map} & \psi \in \Psi & = & X_f \rightharpoonup \Omega \\ \textit{Alias} & \omega \in \Omega & = & \{\&c\ x \mid c \in C,\ x \in X_f\} \cup \{\top, \bot\} \end{array}$$

Again, set inclusion is used as the comparison operator for the abstract domain and again, its lattice is of finite size and height. Furthermore, the abstract domain we define represents a sound over-approximation of program behaviours, since it covers each possible path in a program and all possible values for variables are accounted for in Ω: references and undefined values are precisely tracked, while all other values are mapped to $\top$.

The abstract semantics of our analysis are defined by a relation

$$\_;\_;\_ \vdash \_ \downarrow_I \_ : M \times \Gamma \times \Sigma \times I \to \Sigma$$

whose inference rules traverse function bodies in their natural order of execution. Each rule takes as input: the MIMIR module under analysis, a typing environment $\gamma \in \Gamma$ mapping variable names to their types, the current aliasing state $\sigma \in \Sigma$, and an instruction $i \in I$. The rule then produces an updated aliasing state that reflects the effect of executing i on each $\psi_j \in \sigma$.

The rule for conditional instructions follows the same principle as in liveness analysis: aliasing information is computed independently for each branch, and the resulting abstract states are then merged by applying the union operator to the corresponding alias maps.

A-IF
$$\frac{m;\gamma;\sigma \vdash i \downarrow_I \sigma' \qquad m;\gamma;\sigma \vdash i' \downarrow_I \sigma'' \qquad \sigma''' = \sigma' \cup \sigma''}{m;\gamma;\sigma \vdash \texttt{if}\ o\ \{\ i\ \}\ \texttt{else}\ \{\ i'\ \} \downarrow_I \sigma'''}$$

Loop constructs are handled analogously to those in liveness analysis, with abstract states computed via fixed-point iteration. Sequences of instructions are also treated in a similar manner, with the difference that they are analysed in the natural order of execution of the function, rather than in reverse.

A-WHILE
$$\frac{\sigma_0 = \sigma \qquad \sigma' = \bigcup_{j\geq 1} \sigma_j \text{ where } (m;\gamma;\sigma_{j-1} \vdash i \downarrow_I \sigma_j)}{m;\gamma;\sigma \vdash \texttt{while}\ o\ \{\ i\ \} \downarrow_I \sigma'''}$$

A-SEQ
$$\frac{m;\gamma;\sigma \vdash i \downarrow_I \sigma' \qquad m;\gamma;\sigma' \vdash i' \downarrow_I \sigma''}{m;\gamma;\sigma \vdash i; i' \downarrow_I \sigma''}$$

The rules for borrow and assignment instructions specify how the aliasing state is updated for a given place expression p. First, for each environment ψ_j in σ, the variable x referenced by

p is identified. Next, the abstract reference corresponding to the value assigned to p is computed within the same environment. Finally, the aliasing state is updated so that x is mapped to the computed reference in each ψ_j.

A-BORROW
$$\frac{\forall \psi_j \in \sigma,\ \psi_j \vdash p \downarrow_P x,\ \ \psi_j \vdash p' \downarrow_P x',\ \ \psi'_j = \psi_j[x \mapsto \&c\ x'] \qquad \sigma' = \bigcup_{j \in 1..|\sigma|} \psi'_j}{m; \gamma; \sigma \vdash p = \&c\ p' \downarrow_I \sigma'}$$

A-ASSIGN
$$\frac{\forall \psi_j \in \sigma,\ \psi_j \vdash p \downarrow_P x,\ \ \psi_j \vdash o \downarrow_O \omega,\ \ \psi'_j = \psi_j[x \mapsto \omega] \qquad \sigma' = \bigcup_{j \in 1..|\sigma|} \psi'_j}{m; \gamma; \sigma \vdash p = o \downarrow_I \sigma'}$$

The rules above employ the relation

$$\_ \vdash \_ \downarrow_P \_ : \Psi \times P \rightharpoonup X_f$$

to determine the variable to which a given place expression refers in a specific environment.

A-DEREF
$$\frac{\psi \vdash p \downarrow_P x \qquad \psi(x) = \&c\ x'}{\psi \vdash *p \downarrow_P x'}$$

A-VARIABLE
$$\frac{}{\psi \vdash x \downarrow_P x}$$

The assignment rule additionally relies on the relation

$$\_ \vdash \_ \downarrow_O \_ : \Psi \times O \rightharpoonup \Omega$$

to determine the alias associated with an instruction operand. Operands that are not place expressions are mapped to $\top$.

A-OPERAND
$$\frac{o \notin p}{\psi \vdash o \downarrow_O \top}$$

A-PLACE
$$\frac{\psi \vdash p \downarrow_P x \qquad \psi(x) = \omega}{\psi \vdash p \downarrow_O \omega}$$

The rule for call instructions demonstrates how origin annotations constrain the set of references that may be assigned to the instruction's target place.

A-CALL
$$\frac{m; \gamma \vdash o : (\bar{t}) \to t \qquad \forall \psi_j \in \sigma,\ \psi_j \vdash p \downarrow_P x, \qquad \Omega_j = \bigcup_{t_i \in \bar{t}} alias(\psi_j, t, t_i, o_i), \qquad \sigma_j = \bigcup_{\omega \in \Omega_j} \psi_j[x \mapsto \omega] \qquad \sigma' = \bigcup_{j \in 1..|\sigma|} \sigma_j}{m; \gamma; \sigma \vdash p = o(\bar{o}) \downarrow_I \sigma'}$$

Specifically, it relies on the *alias* predicate, which selects the subset of arguments occupying the positions of parameters whose types are allowed as return values by the function's signature and returns their corresponding aliases.

$$\frac{t' <: t \qquad \psi \vdash o \downarrow_O \omega}{alias(\psi, t, t', o) = \{\omega\}}$$

$$\frac{\&r\ c'\ t' \not<: t \qquad alias(\psi, t, t', *o) = \Omega}{alias(\psi, t, \&r\ c'\ t', o) = \Omega}$$

$$\frac{t' \not<: t \qquad t' \neq \&r\ c\ t''}{alias(\psi, t, t', o) = \emptyset}$$

Finally, return instructions leave the aliasing state unchanged.

A-RETURN
$$\frac{}{m; \gamma; \sigma \vdash \texttt{return} \downarrow_I \sigma}$$

6.3. Borrow Checking

Once liveness, aliasing, and initialization information have been computed for every instruction in a function body, each abstract state is systematically examined to detect potential violations of the aliasing constraints imposed by the borrow-checking discipline. Liveness and aliasing information are jointly analysed to ensure that no live mutable and immutable references to the same variable coexist, and that at most one live mutable reference to a given location exists at any program point. In addition, all places accessed by each instruction are checked for proper initialization prior to use. If no violations are identified throughout the trace, the program is classified as valid.

6.4. Efficient Representation of Abstract States

The increased precision of the abstract domain employed in our formulation of the borrow checker comes with the trade-off of potentially larger abstract states in the program traces it produces. Specifically, by representing abstract states as sets of maps rather than as maps with sets in their codomains, we allow for finer-grained tracking of aliasing relationships, but at the cost of possible duplication of information across environments. While the size of these environment sets is theoretically bounded by the cardinality of the Cartesian product of the places present in a function, this upper bound can become significant in pathological cases. Moreover, updating each environment individually may incur non-trivial computational overhead, especially in a naïve implementation. To address this concern, we propose leveraging the structure of *Map Family Decision Diagrams* (MFDDs) (Racordon et al. 2020; Fossati & Coet 2022), a symbolic representation well-suited to compactly encoding large sets of partial maps with identical domains and co-domains.

Figure 8 illustrates how the following example set of partial maps can be compactly represented using an MFDD:

$$\begin{aligned}\psi = \{&[x \mapsto \texttt{\&cst}\ a, y \mapsto \texttt{\&mut}\ b], [y \mapsto \texttt{\&mut}\ a],\\ &[y \mapsto \texttt{\&mut}\ b, z \mapsto \texttt{\&cst}\ a], [x \mapsto \texttt{\&cst}\ a]\}\end{aligned}$$

As shown in the figure, MFDDs represent sets of maps as directed acyclic graphs, where each node corresponds to a key

(i.e., a domain element) and edges are labelled with values from the codomain. Each path from the root to the terminal node $\top$ encodes a distinct map in the set. Dashed edges indicate the absence of a key in the partial map corresponding to a path.

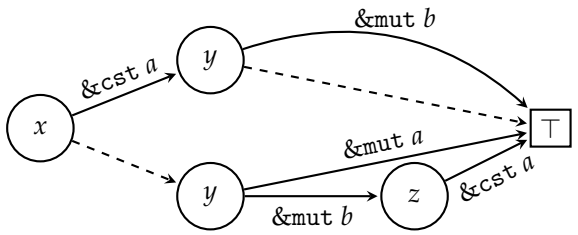


Figure 8 An MFDD for ψ

This representation enables structural sharing of common prefixes and suffixes among maps, allowing MFDDs to efficiently encode large or repetitive sets while minimizing memory usage. Moreover, standard operations on MFDDs such as union, intersection, or bulk updates, are expressed as *homomorphisms* that apply transformations simultaneously across all represented maps and therefore support scalable manipulation of abstract states.

Maintaining the uniqueness of equivalent nodes in an MFDD, however, incurs additional computational overhead. In practice, some form of *hash-consing* is required to guarantee node uniqueness when MFDDs are constructed or merged. As a result, the cost of building and maintaining MFDDs may, in certain cases, outweigh the benefits of their compact shared representation – particularly in pathological scenarios, such as when MFDDs remain very small or when only a limited number of maps exhibit structural overlap.

To establish whether MFDDs provide a practical solution to the potential state space explosion inherent to borrow checking, empirical evaluation through benchmarks on real-world programs is required. Such benchmarks would clarify whether MFDDs provide tangible improvements in memory consumption and performance, as well as determine whether state space explosion arises in realistic scenarios or is limited to highly contrived cases. In particular, we anticipate that the principal benefit of MFDDs lies in their ability to mitigate exponential growth when analysing programs with large branching structures. Although a comprehensive evaluation remains pending, we expect scenarios involving deeply nested conditional constructs over a relatively small set of variables to be especially well-suited to this technique, offering clear advantages over naïve brute-force enumeration.

7. Related Work

Our work builds upon and contributes to the growing body of research dedicated to the formalisation of Rust's memory safety guarantees (Jung et al. 2018, 2020; Pearce 2021; Weiss et al. 2021). As discussed in the introduction, the crucial difference of previous efforts compared to our approach lies in their use of type-theoretic foundations to capture ownership and aliasing constraints, rather than our proposed use of static analysis.

The use of abstract interpretation to detect memory safety violations has been extensively studied in the context of other languages (Blanchet et al. 2003; Kirchner et al. 2015; Saan et al. 2024; Racordon & Abrahams 2023). However, to the best of our knowledge, its application to borrow checking, especially as observed in Rust, has not previously been explored. Our formulation demonstrates that the key concepts underpinning Rust's borrow checker can be captured through a sound and scalable abstract interpretation framework.

Our methodology draws from classical techniques in pointer analysis (Smaragdakis & Balatsouras 2015) and shape analysis (Rinetzky & Sagiv 2001), with particular resemblance to the principles underlying *compositional shape analysis* (Calcagno et al. 2011), given the intra-procedural focus of our framework.

Related efforts have also explored the design of intermediate representations aimed at addressing memory safety concerns (Walker & Morrisett 2001; Smith et al. 2000; Morrisett et al. 2005; Racordon et al. 2022). Unlike these approaches, which primarily rely on type checking or dynamic enforcement mechanisms, our proposal is grounded in static analysis via abstract interpretation.

8. Conclusion

In this paper, we introduced a novel formulation of Rust's borrow checking mechanism within the framework of abstract interpretation over a custom intermediate representation. By carefully designing its abstract domain, our analysis supports path-sensitive reasoning about aliasing and initialization, thereby capturing program behaviours that exceed the capabilities of Rust's conservative static checks. The formulation incorporates a sound type system, based on origin-tracked references, which guarantees memory safety properties such as non-escaping local references and safe aliasing.

By decoupling borrow checking from Rust's surface syntax and embedding it into a general-purpose intermediate representation, we establish a foundation for transferring Rust-style memory safety guarantees to a wider range of programming languages and compiler infrastructures.

To address the potential complexity and size of abstract states, we proposed the use of Map Family Decision Diagrams (MFDDs) as a compact and efficient representation of sets of abstract environments. An important direction for future research is to evaluate whether this approach yields significant performance and memory improvements when applied to large-scale programs.

Another promising line of work is the extension of the IR with richer language constructs, including user-defined types, polymorphism, and higher-order functions, in order to broaden the applicability of the framework. Of particular interest is the integration of move semantics, which would require enriching the type system with substructural rules akin to those employed in Rust to faithfully capture ownership and resource transfer.

References

Abrahams, D. (2024). *Hylo - the safe systems and generic-programming language built on value semantics.* Retrieved

from https://www.youtube.com/watch?v=5lecIqUhEl4

Alliance, B. (2023). *The cranelift compiler.* Retrieved from https://github.com/bytecodealliance/wasmtime/tree/main/cranelift

Amin, N., & Rompf, T. (2017). Type soundness proofs with definitional interpreters. In *Proceedings of the 44th ACM SIGPLAN symposium on principles of programming languages* (pp. 666–679). ACM. Retrieved from https://dl.acm.org/doi/10.1145/3009837.3009866 doi: 10.1145/3009837.3009866

Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., … Rival, X. (2003). A static analyzer for large safety-critical software. , *38*(5), 196–207. Retrieved from https://dl.acm.org/doi/10.1145/780822.781153 doi: 10.1145/780822.781153

Boyland, J. (2003). Checking interference with fractional permissions. In R. Cousot (Ed.), *Static analysis* (Vol. 2694, pp. 55–72). Springer Berlin Heidelberg. Retrieved from http://link.springer.com/10.1007/3-540-44898-5_4

Calcagno, C., Distefano, D., O'Hearn, P. W., & Yang, H. (2011). Compositional shape analysis by means of bi-abduction. , *58*(6), 1–66. Retrieved from https://dl.acm.org/doi/10.1145/2049697.2049700 doi: 10.1145/2049697.2049700

Carruth, C. (2022). *Carbon language: An experimental successor to c++.* Retrieved from https://www.youtube.com/watch?v=omrY53kbVoA

Cousot, P., & Cousot, R. (1977). Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th acm sigact-sigplan symposium on principles of programming languages* (p. 238–252). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/512950.512973 doi: 10.1145/512950.512973

Fahndrich, M., & DeLine, R. (2002). Adoption and focus: practical linear types for imperative programming. In *Proceedings of the ACM SIGPLAN 2002 conference on programming language design and implementation* (pp. 13–24). Association for Computing Machinery. Retrieved from https://doi.org/10.1145/512529.512532 doi: 10.1145/512529.512532

Fossati, S., & Coet, A. (2022). Belief programming with map family decision diagrams.

Haller, P., & Odersky, M. (2010). Capabilities for uniqueness and borrowing. In T. D'Hondt (Ed.), *ECOOP 2010 – object-oriented programming* (Vol. 6183, pp. 354–378). Springer Berlin Heidelberg. Retrieved from http://link.springer.com/10.1007/978-3-642-14107-2_17

Jim, T., Morrisett, J. G., Grossman, D., Hicks, M. W., Cheney, J., & Wang, Y. (2002). Cyclone: A safe dialect of c. In *Proceedings of the general track of the annual conference on usenix annual technical conference* (p. 275–288). USA: USENIX Association.

Jung, R., Dang, H.-H., Kang, J., & Dreyer, D. (2020). Stacked borrows: an aliasing model for rust. , *4*, 1–32. Retrieved from https://dl.acm.org/doi/10.1145/3371109 doi: 10.1145/3371109

Jung, R., Jourdan, J.-H., Krebbers, R., & Dreyer, D. (2018). RustBelt: Securing the foundations of the rust programming language. , *2*, 1–34. Retrieved from https://dl.acm.org/doi/10.1145/3158154 doi: 10.1145/3158154

Kahn, G. (1987). Natural semantics. In *Annual symposium on theoretical aspects of computer science* (pp. 22–39).

Kehrer, P. (2019). *Memory unsafety in apple's operating systems.* Retrieved from https://langui.sh/2019/07/23/apple-memory-safety/#fn:2

Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., & Yakobowski, B. (2015). Frama-c: A software analysis perspective. , *27*(3), 573–609. Retrieved from https://dl.acm.org/doi/10.1007/s00165-014-0326-7 doi: 10.1007/s00165-014-0326-7

Lattner, C., & Adve, V. (2004). Llvm: a compilation framework for lifelong program analysis & transformation. In *International symposium on code generation and optimization, 2004. cgo 2004.* (p. 75-86). doi: 10.1109/CGO.2004.1281665

Lattner, C., Amini, M., Bondhugula, U., Cohen, A., Davis, A., Pienaar, J., … Zinenko, O. (2021). MLIR: Scaling compiler infrastructure for domain specific computation. In *2021 IEEE/ACM international symposium on code generation and optimization (CGO)* (pp. 2–14). Retrieved from https://ieeexplore.ieee.org/document/9370308/?arnumber=9370308 doi: 10.1109/CGO51591.2021.9370308

Lorenzen, A., White, L., Dolan, S., Eisenberg, R. A., & Lindley, S. (2024). Oxidizing ocaml with modal memory management. *Proc. ACM Program. Lang.*, *8*(ICFP). Retrieved from https://doi.org/10.1145/3674642 doi: 10.1145/3674642

Matsakis, N. (2020). *Polonius: Either borrower or lender be, but responsibly.* Retrieved from https://www.youtube.com/watch?v=_agDeiWek8w

McCall, J. (2017). *The swift ownership manifesto.* Retrieved from https://github.com/swiftlang/swift/blob/main/docs/OwnershipManifesto.md

McCall, J. (2023). *Introducing a memory-safe successor language in large c++ code bases.* Retrieved from https://www.youtube.com/watch?v=lgivCGdmFrw

Miller, M. (2019). *Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape.* Retrieved from https://www.youtube.com/watch?v=PjbGojjnBZQ

Morrisett, G., Ahmed, A., & Fluet, M. (2005). L3: A linear language with locations. In P. Urzyczyn (Ed.), *Typed lambda calculi and applications* (pp. 293–307). Springer. doi: 10.1007/11417170_22

(MSRC), M. S. R. C. (2019). *A proactive approach to more secure code.* Retrieved from https://msrc.microsoft.com/blog/2019/07/a-proactive-approach-to-more-secure-code/

Pearce, D. J. (2021). A lightweight formalism for reference lifetimes and borrowing in rust. *ACM Trans. Program. Lang. Syst.*, *43*(1). Retrieved from https://doi.org/10.1145/3443420 doi: 10.1145/3443420

Racordon, D., & Abrahams, D. (2023). *Borrow checking hylo.* Retrieved from https://2023.splashcon.org/details/iwaco-2023-papers/5/Borrow-checking-Hylo

Racordon, D., Coet, A., & Buchs, D. (2022). Toward a lingua franca for memory safety. *Journal of Object Technology*, *21*(2), 2:1-11. Retrieved from http://www.jot.fm/contents/issue_2022_02/article3.html (ECOOP 2021 Workshops) doi:

10.5381/jot.2022.21.2.a3
Racordon, D., Coet, A., Stachtiari, E., & Buchs, D. (2020). Solving schedulability as a search space problem with decision diagrams. In A. Aleti & A. Panichella (Eds.), *Search-based software engineering* (pp. 73–87). Springer International Publishing.
Rebert, A., & Kern, C. (2024). *Secure by design: Google's perspective on memory safety* (Tech. Rep.). Google Security Engineering.
Reynolds, J. C. (1978). Syntactic control of interference. In *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on principles of programming languages* (pp. 39–46). Association for Computing Machinery. Retrieved from https://doi.org/10.1145/512760.512766 doi: 10.1145/512760.512766
Rinetzky, N., & Sagiv, M. (2001). Interprocedural shape analysis for recursive programs. In R. Wilhelm (Ed.), *Compiler construction* (pp. 133–149). Berlin, Heidelberg: Springer Berlin Heidelberg.
Saan, S., Erhard, J., Schwarz, M., Bozhilov, S., Holter, K., Tilscher, S., … Seidl, H. (2024). Goblint: Abstract interpretation for memory safety and termination. In B. Finkbeiner & L. Kovács (Eds.), *Tools and algorithms for the construction and analysis of systems* (pp. 381–386). Springer Nature Switzerland. doi: 10.1007/978-3-031-57256-2_25
Smaragdakis, Y., & Balatsouras, G. (2015). Pointer analysis. *Found. Trends Program. Lang.*, *2*(1), 1–69. Retrieved from https://doi.org/10.1561/2500000014 doi: 10.1561/2500000014
Smith, F., Walker, D., & Morrisett, G. (2000). Alias types. In G. Smolka (Ed.), *Programming languages and systems* (pp. 366–381). Springer. doi: 10.1007/3-540-46425-5_24
Song, D., Lettner, J., Rajasekaran, P., Na, Y., Volckaert, S., Larsen, P., & Franz, M. (2019). SoK: Sanitizing for security. In *2019 IEEE symposium on security and privacy (SP)* (pp. 1275–1295). IEEE. Retrieved from https://ieeexplore.ieee.org/document/8835294/ doi: 10.1109/SP.2019.00010
Stoep, J. V., & Rebert, A. (2024). *Eliminating memory safety vulnerabilities at the source.* Retrieved from https://security.googleblog.com/2024/09/eliminating-memory-safety-vulnerabilities-Android.html
Van Der Veen, V., dutt Sharma, N., Cavallaro, L., & Bos, H. (2012). Memory errors: The past, the present, and the future. In D. Balzarotti, S. J. Stolfo, & M. Cova (Eds.), *Research in attacks, intrusions, and defenses* (Vol. 7462, pp. 86–106). Springer Berlin Heidelberg. Retrieved from https://link.springer.com/10.1007/978-3-642-33338-5_5
Walker, D. (2004). Substructural type systems. In *Advanced topics in types and programming languages.* The MIT Press. Retrieved from https://doi.org/10.7551/mitpress/1104.003.0003 doi: 10.7551/mitpress/1104.003.0003
Walker, D., Crary, K., & Morrisett, G. (2000). Typed memory management via static capabilities. , *22*(4), 701–771. Retrieved from https://dl.acm.org/doi/10.1145/363911.363923 doi: 10.1145/363911.363923
Walker, D., & Morrisett, G. (2001). Alias types for recursive data structures. In R. Harper (Ed.), *Types in compilation* (pp. 177–206). Springer Berlin Heidelberg.
Watson, R. N. M., Baldwin, J., Chisnall, D., Chen, T., Clarke, J., Davis, B., … Witaszczyk, K. (2025). *It is time to standardize principles and practices for software memory safety.* Retrieved from https://cacm.acm.org/opinion/it-is-time-to-standardize-principles-and-practices-for-software-memorysafety/#B12
Weiss, A., Gierczak, O., Patterson, D., & Ahmed, A. (2021). *Oxide: The essence of rust* (No. arXiv:1903.00982). Retrieved from http://arxiv.org/abs/1903.00982 doi: 10.48550/arXiv.1903.00982
Wright, A. K., & Felleisen, M. (1994). A syntactic approach to type soundness. , *115*(1), 38–94. Retrieved from https://www.sciencedirect.com/science/article/pii/S0890540184710935 doi: 10.1006/inco.1994.1093

About the authors

Aurélien Coet is a PhD candidate at the University of Geneva, Switzerland, in the *Semantics, Modelling and Verification* group, under the supervision of Professor Didier Buchs. His research focuses on the use of static analysis to enforce memory safety in programming languages. You can contact the author at aurelien.coet@unige.ch.

Didier Buchs is honorary professor at the University of Geneva, Switzerland, and the head of the *Semantics, Modelling and Verification* group. His research focuses primarily on formal specifications and validation techniques for complex, distributed systems. You can contact the author at didier.buchs@unige.ch.