

Use-Based Reference Types for Borrow Checking

Jasper Gräfllich

Agentur für Innovationen in der Cybersicherheit
Secure Systems
Halle (Saale), Germany
graeflich@cyberagentur.de
Potsdam University
Theoretical Computer Science
Potsdam, Germany
graeflich@uni-potsdam.de

Abstract

Borrowing systems like the one of Rust use various types of references to prevent mutable aliasing. By forbidding the creation of references that may violate the *Aliasing XOR Mutation (AXM)* principle, memory corruption and data races are prevented.

We present a more liberal approach that does not limit *creation* of references and instead utilizes compile-time proofs to ensure that references are never *used* in a way that could violate AXM.

Keywords: static analysis, borrowing, references, alias analysis

ACM Reference Format:

Jasper Gräfllich. 2023. Use-Based Reference Types for Borrow Checking. In *Proceedings of ACM Conference (Conference'17)*. ACM, New York, NY, USA, 3 pages. <https://doi.org/10.1145/nnnnnnn.nnnnnnn>

1 Introduction

The *Aliasing XOR Mutation (AXM)* principle stems from the observation that data races can never happen, as long as

- if several references point to the same data (that is, they *alias*), they all are read-only, and
- if a reference may mutate some data, no other reference may be around to observe this mutation.

In most programming languages, it is up to the programmer to validate the principle, but there are several approaches enforcing it through the compiler:

- In Rust, reading and writing permissions are tied to a reference at its creation and the compiler enforces that no two live references with conflicting permissions exist.
- *Uniqueness types* ensure that there is only one reference to some value and *linear types* ensure that no new references may be created.
- As a variant on linear types, ATS has proof objects. Proof objects are created together with a resource and

everytime the resource is used, a proof of permissions must be supplied. This is called proof threading.

There are some limitations to these approaches. Because Rusts borrow checker can't always verify AXM, there is the fallback mechanism of unsafe blocks in which the programmer can use raw, unchecked pointers but has to reason without help from the compiler. Uniqueness and linear types reject many correct programs. Proof threading à la ATS is a large overhead and syntactically loud, as half or more of a program's source code may be proof threading annotations.

We propose an alternative approach to borrow checking by introducing

- A region-based borrowing system using lightweight proof objects (section 2),
- use-based permission tracking (section 3), and
- a formalization of these concepts in an extension to a simple λ -calculus (section 4).

2 Regions and Aliasing

We propose that the construction of provenance sets shall only rely on the local context of a function f . There are three ways of bringing a reference into context, each of which comes with a different way to determine its region.

1. A borrow expression, $\&x$, creates a new reference.
2. The reference is passed as one of the arguments of f .
3. Within f , another function f' is called that returns a reference.

2.1 Provenance of Borrows

A borrow expression, $\&x$, creates a reference r with a fresh abstract location p and a region and type inherited from x . If x has type τ and is at location l , we write $x : [l]\tau$, and we have that $\&x : \&[r \mid l]\tau$.

2.2 Provenance of Function Arguments

If a reference is supplied as an argument to a function, its provenance set is disjoint from all references created by borrows or allocations within the function body. Since neither the concrete location for the reference nor its parent is known, a fresh abstract location is created for the reference.

If several references x_1, \dots, x_n are supplied as function arguments, every reference is assigned a unique abstract location p_1, \dots, p_n . However, when calling a function, several of the supplied arguments can originate from the same concrete location, as in

$$\begin{aligned} x &= \text{new}(l, e); \\ f(\&x, \&x) \end{aligned} \quad (1)$$

Therefore, the provenance set of each reference x_i consists of the locations of *all* arguments: $x_i : [p_i \mid p_1, \dots, p_n]\tau$. This limits the use of the arguments within the function body of f . To circumvent this, the provenance sets of arguments can be reduced by introducing *no alias proofs*. A function signature like

$$f[x_1 \leftrightarrow x_1](x_1 : \tau, x_2 : \tau) \rightarrow \tau'$$

ensures that x_1 and x_2 will never alias by preventing a call like in 1. Within f , we then have $x_i : [p_i \mid p_i]$ for $i = 1, 2$.

2.3 Provenance of Return Values

If a function returns a reference, it can be either an owning reference to a new allocation, or it can be derived from one or several of the inputs. By default, it must be assumed that the provenance set of a returned reference is the union of all provenance sets of input references. Proofs can reduce the provenance set, similar to 2.2. For example, with the function type

$$f[x_1 \leftrightarrow x_2](x_1 : [r_1]\tau, x_2 : [r_2]\tau) \rightarrow \&[r_1]\tau$$

we can be sure that the value returned by a call to f will share provenance with x_1 but not with x_2 .

3 Permissions

To track the usage of references, each reference has an attached *permission* for each point of execution that tracks the operations necessary for evaluation. The possible permissions are:

- An *inactive* reference, denoted $\&_i$, is not dereferenced. An inactive reference may be used in other ways, as assigned or referenced in turn.
- A reference with *read* permission, denoted $\&_r$, is dereferenced ($*x$), but not assigned to. It may also be assigned or referenced.
- A reference with *write* permission, denoted $\&_w$, is assigned to ($x := e$). It may also be used in any way references with inactive and read permissions may be used.

3.1 Permissions in sequential code

In sequential code, reads only happen in $*x$ expressions while writes only happen in $x := e$ expressions.

TODO: Function calls

EXPRESSIONS	$e ::= x = \text{new}(l); e \mid x = e; e \mid \&x \mid *x \mid e(\bar{e}) \mid x := e$
VALUES	$x ::= \text{fn } f[\bar{\eta}](\bar{x}) = e$
TYPES	$\tau ::= \&[\bar{\eta}]\tau \mid \text{fn}(\bar{\tau}) \rightarrow \tau \mid \perp$
PROOFS	$\pi ::= x_1 \leftrightarrow x_2$
LOCATIONS	$\eta ::= l \mid p$
TYPING CONTEXT	$\Gamma ::= \emptyset \mid \Gamma, x : [\eta]\tau$
LOCATION CONTEXT	$\Lambda ::= \emptyset \mid \Lambda, \eta \mapsto \tau$

Figure 1. Syntax of λ_{UR}

T-NEW	
$\Gamma; \Lambda, l \mapsto \perp \vdash e : \tau$	$\Gamma; \Lambda \vdash x = \text{new}(l); e : \tau$
T-ASSIGN	
$\Gamma; \Lambda \vdash x : \&[\bar{\eta}]\tau_1 \quad \Gamma; \Lambda \vdash e_1 : \tau_1 \quad \Gamma; \Lambda \vdash e_2 : \tau_2$	$\Gamma; \Lambda \vdash x = e_1; e_2 : \tau_2$
T-BORROW	
$\Gamma; \Lambda \vdash x : [\bar{\eta}]\tau$	$\Gamma; \Lambda \vdash \&x : \&[\bar{\eta}]\tau$
T-DEREF	
$\Gamma; \Lambda \vdash x : \&[\bar{\eta}]\tau$	$*x : \tau$
T-FUN	
$\Gamma, f : \text{fn}(\bar{\tau}) \rightarrow \tau \vdash e : \tau$	$\Gamma; \Lambda \vdash \text{fn } f(\bar{x}) = e : \text{fn}(\bar{\tau}) \rightarrow \tau$
T-VAR	
$x : \tau \in \Gamma$	$\Gamma, \Lambda \vdash x : \tau$

Figure 2. Typing of λ_{UR}

3.2 Permissions in parallel code

TODO: Parallel code

Now, if an expression e contains two references x_1 and x_2 , one the following conditions must be met:

4 Formalization

4.1 Syntax of λ_{UR}

Figure 1 summarizes the syntax of λ_{UR} .

Local variables can be introduced with bindings and are pure values. New allocations can be introduced by the $x = \text{new}(l); e$ construction, binding the local variable x to a concrete heap-allocated location represented by l .

A function is declared as $\text{fn } f(\bar{x}) = e$, where f is a binder for the (potentially recursive) function and \bar{x} is a list of parameter binders. Functions can be called using parentheses.

λ_{UR} introduces references, marked by $\&$. Reference types contain the type of the pointee and a list of possible locations of the pointee.

4.2 Type Checking of λ_{UR}

1. Use CFG: Each node of the CFG keeps track of all references, their state, and their potential provenances
2. Operational semantics to walk through the CFG

5 Related Work

1. ATS – manual proof threading
2. Rust – construction-based references
3. Vale – read/write regions
4. Linear programming – returning values back
5. Weaknesses???

6 Future Work

1. Adding product types/arrays/multi-location-allocations
2. Non-termination?
3. Generic/dependent proofs, monomorphization: One function may have several signatures depending on supplied argument provenances. Stronger input guarantees lead to stronger output guarantees?
4. Provenance set == region?

7 Editing Notes

- Enough examples?
- Intuition, not details?
- Sentence flow?
- One point per paragraph?
- Bigger picture clear?
- Consistent names?
- Context–Gap–Innovation
- Uniform format for algorithmic, syntax, semantics, and text.
- Does "new" need an expression?
- location + provenance set notation
- Do we need types? Pro: Deref should only work on references, not on pure values. Con: If we have types, we must pay attention to if/how types influence the ability of references to alias (can $x_1 : \tau_1$ and $x_2 : \tau_2$ alias for $\tau_1 \neq \tau_2$). Maybe add base types, even? They have no provenance but are pure values.
- Parallel execution in syntax/semantics? Async?
- $**x := e$ marks x as read but $*x$ as write – if we allow it (currently we don't)
- Double reads are a problem: x_1 read, x_2 write, x_1 read again. No race condition, but also no guarantee that two reads of a read-only reference will yield the same value.
- Sequence expressions
- 1. Vale regions: imm/rw
- 2. Builds on Ownership, needs One True Name/Provenance

Usage examples:

1. Known *concrete* provenance

2. Several references with the same provenance
3. Several references with different concrete provenance
4. Several references with *abstract* provenance: Need proofs
5. Passing proofs to functions/proofs in the signature
6. unsafe proofs

Semantics of Proof Finding:

1. Fresh allocations are unrestricted
2. Keep track of all (de-)activation points
3. Every reference has a list of possible provenances, that is, names it may alias with.
4. Having a point in the program at which two references which may alias and that are active in a way that violates AXM, is an error
5. Proofs can be passed as arguments/are part of function signatures
6. Generate proofs in an unsafe way