RustHorn: CHC-based Verification for Rust Programs

YUSUKE MATSUSHITA, The University of Tokyo, Japan
TAKESHI TSUKADA, Chiba University, Japan
NAOKI KOBAYASHI, The University of Tokyo, Japan

Reduction to satisfiability of constrained Horn clauses (CHCs) is a widely studied approach to automated program verification. Current CHC-based methods, however, do not work very well for pointer-manipulating programs, especially those with dynamic memory allocation. This paper presents a novel reduction of pointer-manipulating Rust programs into CHCs, which clears away pointers and memory states by leveraging Rust's guarantees on permission. We formalize our reduction for a simplified core of Rust and prove its soundness and completeness. We have implemented a prototype verifier for a subset of Rust and confirmed the effectiveness of our method.

CCS Concepts: • **Theory of computation → Program verification**; **Type theory**.

Additional Key Words and Phrases: Rust, permission, ownership, pointer, CHC, automated verification

ACM Reference Format:
Yusuke Matsushita, Takeshi Tsukada, and Naoki Kobayashi. 2021. RustHorn: CHC-based Verification for Rust Programs. *ACM Trans. Program. Lang. Syst.* x, x, Article 1 (January 2021), 53 pages. https://doi.org/10.1145/3462205

1 INTRODUCTION

Reduction to *constrained Horn clauses (CHCs)* is a widely studied approach to automated program verification of functional correctness [6, 24].

Technically, a CHC is a Horn clause [32] equipped with constraints, i.e., a formula of the form $\varphi \Longleftarrow \bigwedge \vec{\psi}$, where each of the formulas $\varphi, \psi_0, \ldots, \psi_{m-1}$ is either a constraint (e.g., $a < b + 1$) or an atomic formula of the form $f(\vec{t})$, where f is a *predicate variable* and $\vec{t} = t_0, \ldots, t_{n-1}$ are terms. Each free variable in a CHC is semantically universally quantified over some fixed sort (e.g., int, bool), which we usually omit for brevity. To aid understanding, we extend the notion of CHCs to allow disjunctions and existential quantifiers in the body (i.e., the right-hand side of the implication). Any CHC in this extended form can easily be translated into a conjunction of standard CHCs. A *system of CHCs* or a *CHC system* is a finite set of CHCs, which semantically means conjunction of the component CHCs.

CHC solving is the process of deciding whether a given system of CHCs has a *solution*, i.e., a valuation of predicate variables which makes all the CHCs in the system valid. We say that a system of CHCs is *satisfiable* if it has a solution. A variety of program verification problems can be naturally reduced to CHC solving [6, 24].

For example, let us consider the following C code that defines McCarthy's 91 function.

Authors' addresses: Yusuke Matsushita, The University of Tokyo, Tokyo, Japan, yskm24t@is.s.u-tokyo.ac.jp; Takeshi Tsukada, Chiba University, Chiba, Japan, tsukada@math.s.chiba-u.ac.jp; Naoki Kobayashi, The University of Tokyo, Tokyo, Japan, koba@is.s.u-tokyo.ac.jp.

0164-0925/2021/1-ART1 $15.00
https://doi.org/10.1145/3462205

```
int mc91(int n) {
  if (n > 100) return n - 10; else return mc91(mc91(n + 11));
}
```

Suppose we wish to verify that, for any $n \leq 101$, `mc91`(n) returns 91 if the computation terminates, which is a kind of (partial) *functional correctness* of the function `mc91`.[1] The verified property is equivalent to the satisfiability of the following system of CHCs (here, if φ then ψ else ψ' is sugar for $(\varphi \land \psi) \lor (\neg\varphi \land \psi')$):[2] [3]

$$\begin{aligned} &\mathit{Mc91}(n,r) \impliedby \text{if } n > 100 \text{ then } r = n - 10 \text{ else } \exists r'.\, \mathit{Mc91}(n+11,r') \land \mathit{Mc91}(r',r) \\ &r = 91 \impliedby n \leq 101 \land \mathit{Mc91}(n,r) \end{aligned}$$

The predicate $\mathit{Mc91}(n,r)$ means that `mc91`(n) returns r (if it terminates). The first CHC in the system above describes the specification of `mc91` and the second one describes the required property of `mc91`. We can verify the expected property by finding a solution to the system like below.

$$\mathit{Mc91}(n,r) \;:\Longleftrightarrow\; r = 91 \lor (n > 100 \land r = n - 10).$$

As observed in the example above, finding a solution to CHCs generated from a program with loops and recursions is strongly related to finding the *invariant* on loops and recursions.

A *CHC solver* provides a common infrastructure for a variety of programming languages and properties to be verified. There are efficient CHC solvers [13, 20, 31, 42] that can solve instances obtained from actual programs. For example, the above CHC system on *Mc91* can be solved instantly by many CHC solvers, including Spacer [42] and HoIce [13]. As a consequence, many modern automated program verification tools [25, 27, 30, 39, 40, 66] reduce verification programs to CHCs and use CHC solvers.

Current CHC-based methods, however, do not work very well for *pointer*-manipulating programs, especially those with *dynamic memory allocation*, as we see in §1.1. In this paper, we focus on programs written in the *Rust* programming language, which provides strong guarantees on *permission* or *ownership* of pointers. We present a novel reduction of Rust programs into CHCs, which clears away explicit representation of pointers and memory states for smooth verification, as we overview in §1.2.

1.1 Challenges in Verifying Pointer-Manipulating Programs

A standard CHC-based approach [25] for pointer-manipulating programs represents the memory state as an *array* that maps each address to the data at the address, which is passed around as an argument of each predicate (cf. the *store-passing style*). In particular, SeaHorn [25], a standard CHC-based verification tool for C/C++, uses this array-based reduction.

For example, let us consider the following pointer-manipulating variation of the previous program.

```
void mc91p(int n, int* r) {
  if (n > 100) *r = n - 10;
```

[1] To be precise, `int` in C usually represents a 32-bit integer. However, in this paper, we just consider unbounded integers for simplicity.

[2] Note that this system can be straightforwardly transformed into the following system of standard CHCs.

$$\begin{aligned} &\mathit{Mc91}(n,r) \impliedby n > 100 \land r = n - 10 \\ &\mathit{Mc91}(n,r) \impliedby \neg(n > 100) \land \mathit{Mc91}(n+11,r') \land \mathit{Mc91}(r',r) \\ &r = 91 \impliedby n \leq 101 \land \mathit{Mc91}(n,r) \end{aligned}$$

[3] Although some CHC-based verifiers use forward reduction, where the implication of each CHC goes in the direction of program execution, in this paper we use backward reduction, where the implication goes in the opposite direction.

```
  else { int s; mc91p(n + 11, &s); mc91p(s, r); }
}
```

It is reduced into the following system of CHCs by the array-based approach.

$$\begin{aligned} \mathit{Mc91p}(n, r, h, h') &\impliedby \text{if } n > 100 \text{ then } h' = h\{r \leftarrow n - 10\} \\ &\qquad \text{else } \exists s', h''.\ \mathit{Mc91p}(n + 11, s', h, h'') \land \mathit{Mc91p}(h''[s'], r, h'', h') \\ h'[r] = 91 &\impliedby n \le 101 \land \mathit{Mc91p}(n, r, h, h') \end{aligned}$$

Here, $h\{r \leftarrow v\}$ denotes the array made from h by replacing the value at index r with v, and $h[r]$ denotes the value of the array h at index r. Unlike *Mc91*, the predicate *Mc91p* additionally takes two arrays h and h', which respectively represent the memory states before and after the call of `mc91p`. The second argument r of *Mc91p*, representing the pointer argument `r` of `mc91p`, is an index for the memory-state arrays. So the assignment `*r = n - 10` is modeled in the then part of the first CHC as $h' = h\{r \leftarrow n - 10\}$, obtained by updating the r-th element of the memory-state array. In the else part, s' represents `&s`. This CHC system has a simple solution

$$\mathit{Mc91p}(n, r, h, h') \ :\Longleftrightarrow\ h'[r] = 91 \lor (n > 100 \land h'[r] = n - 10),$$

which can be found by some array-supporting CHC solvers including Spacer [42] with the support of arrays by the underlying SMT solvers [10, 67].

However, the array-based approach has some shortcomings. Let us consider, for example, the following innocent-looking code (here, `rand()` is a non-deterministic function that can return any integer value).

```
bool just_rec(int* ma) {
  if (rand() > 0) return true;
  int a0 = *ma; int b = rand(); just_rec(&b); return (a0 == *ma);
}
```

Depending on the return value of `rand()`, `just_rec(ma)` either (i) immediately returns `true` or (ii) recursively calls itself and checks whether the target of `ma` remains unchanged through the recursive call. Since the target object of `ma` is not modified through the call of `just_rec`, the return value `a0 == *ma` is always `true`. A tricky point is that the function can modify the memory by newly allocating the data of `b`.

Suppose we wish to verify that `just_rec` never returns `false`. The array-based reduction generates a system of CHCs like the following.

$$\begin{aligned} \mathit{JustRec}(ma, h, h', r) &\impliedby (h' = h \land r = \mathrm{true}) \lor \\ &\qquad \big(\exists b, mb.\ mb \neq ma \land \mathit{JustRec}(mb, h\{mb \leftarrow b\}, h', \_) \land r = (h[ma] == h'[ma])\big) \\ r = \mathrm{true} &\impliedby \mathit{JustRec}(ma, h, h', r) \end{aligned}$$

Here, we have omitted the allocation for `a0` for simplicity. We use ==, !=, >=, && to denote binary operations that return a boolean value. An underscore '_' denotes any fresh variable, which semantically means that we don't care the value.

Unfortunately, the CHC system above is *not* satisfiable, which causes a *false alarm* of unsafety. This is because mb may not necessarily be completely fresh in this formulation. Although mb is made different from the argument ma of the current call, it may coincide with ma of some ancestor call. For example, we can derive contradiction from the CHCs above as follows; here, $[0 \mapsto m,\ 1 \mapsto n]$ denotes the array that maps the address 0 to m and 1 to n (and any other address to 0).

$$\mathit{JustRec}(0, [0 \mapsto 5,\ 1 \mapsto 4], [0 \mapsto 5,\ 1 \mapsto 4], \mathrm{true}) \quad (\because \text{immediate return})$$

$$\begin{aligned}
&\therefore\ \mathit{JustRec}(1, [0 \mapsto 3,\ 1 \mapsto 4], [0 \mapsto 5,\ 1 \mapsto 4], \mathsf{true}) \quad (\because \text{ set } b = 5,\ mb = 0)\\
&\therefore\ \mathit{JustRec}(0, [0 \mapsto 3,\ 1 \mapsto 0], [0 \mapsto 5,\ 1 \mapsto 4], \mathsf{false}) \quad (\because \text{ set } b = 4,\ mb = 1)\\
&\therefore\ \mathsf{false} = \mathsf{true} \quad (\because \text{ the required property})
\end{aligned}$$

The simplest remedy would be to model a memory allocation strategy more faithfully. For example, one can also manage the stack pointer *sp*, which represents the maximum address index that has been used for memory allocation so far.

$$\begin{aligned}
&\mathit{JustRec}_+(ma, h, sp, h', sp', r) \impliedby (h' = h \land sp' = sp \land r = \mathsf{true}) \lor\\
&\qquad (\exists b.\ \mathit{JustRec}_+(sp+1, h\{sp+1 \leftarrow b\}, sp+1, h', sp', \_) \land r = (h[ma] == h'[ma]))\\
&r = \mathsf{true} \impliedby \mathit{JustRec}_+(ma, h, sp, h', sp', r) \land ma \le sp
\end{aligned}$$

The resulting CHC system now has a solution, but it involves a *quantifier*.

$$\mathit{JustRec}_+(ma, h, sp, h', sp', r) \ :\Longleftrightarrow\ r = \mathsf{true} \land ma \le sp \le sp' \land \forall i \le sp.\ h[i] = h'[i]$$

Here, we need a quantified invariant $\forall i \le sp.\ h[i] = h'[i]$, which states that the memory region of $\{i \mid i \le sp\}$, which has an unbounded size, remains unchanged.

Finding quantified invariants is known to be very difficult in general, despite active studies on it [2, 21, 28, 38, 44]. The quantified formula needed above is fairly simple, but for more realistic programs, much more complex quantified formulas can be necessary to represent solutions. Therefore, current array-supporting CHC solvers usually fail in finding quantified invariants for CHC outputs of the existing method. Indeed, the Spacer CHC solver fails in solving even the above CHC system for $\mathit{JustRec}_+$.

In order to avoid this kind of difficulty, many verification tools for pointer-manipulating programs analyze pointer usage to refine the memory model [25, 26, 39]. For example, for the verification problem of `just_rec`, SeaHorn generates CHCs *without arrays*, by successfully analyzing that no effective destructive update happens, although SeaHorn usually uses the array-based reduction. Still, such analyses are usually more or less ad-hoc and can easily fail for advanced pointer uses.[4] Existing verifiers like SeaHorn target programming languages like C/C++ and Java, which do not restrict *aliasing* of pointers, which causes the difficulties in program verification.

1.2 Our Approach: Leverage Rust's Guarantees on Permission

Rust [47, 58] is a systems programming language that supports low-level efficient memory operations like C/C++ and at the same time provides *high-level safety guarantees* using a *permission*-based type system. Despite its unique type system, Rust attains high productivity and has been widely used in industry recently [18, 51, 53].

This paper proposes a novel approach to CHC-based automated verification of programs written in Rust. Our method clears away explicit representation of pointers and memory states by leveraging Rust's permission guarantees.

Rust's Permission Control. Before describing our approach, we briefly explain the permission control mechanism of Rust. Various styles of *permission/ownership/capability* have been introduced to control and reason about pointers in programming language design, program analysis and verification [7–9, 14, 33, 68, 69]. The permission control mechanism of Rust's type system, which we focus on, inherits a lot from existing approaches but also has some unique features.

In Rust, whenever an alias (or pointer) accesses an object, it needs *permission* for that. There are two types of permission in Rust: *update permission*, which allows both write and read access, and *read permission*, which allows only read access. At a high level, Rust's permission control

[4] We examined SeaHorn in our experiments, which are reported in §5.

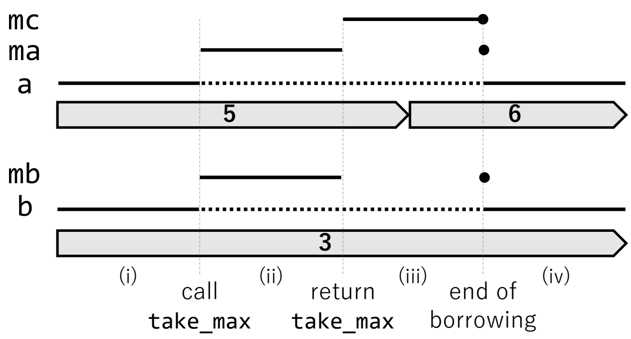

Fig. 1. Values and aliases of a and b in executing `inc_max(5, 3)`. Each row shows each alias/variable's timeline of the update permission. A solid line expresses possession of the update permission. A bullet shows the time point when the borrowed update permission is given back. For example, b has the permission to update its integer object during (i) and (iv), but temporarily loses it during (ii) and (iii) because the pointer mb, created upon the function call of `take_max(&a, &b)`, *borrows* b until the end of (iii).

guarantees that whenever an alias can read from an object (with update or read permission) any other alias cannot write to the object (i.e., does not have update permission). In this paper, we mainly focus only on update permission. For understanding our approach, it suffices to keep in mind that at most one alias can have update permission to each object.

For flexible permission control, Rust supports an operation called *borrowing*. In short, it is a temporary transfer of permission to a newly created pointer called a *reference*. A reference that borrows update or read permission is called a *mutable* or *immutable* reference, respectively.[5] When borrowing is performed, the *deadline* is determined. The reference can use its permission only until this deadline.

As a simple example of Rust's borrowing, let us consider the program below, which is also an interesting target of verification. It is written in C to aid understanding for a wide range of readers; the version in Rust is presented later in this subsection.

```
1 int* take_max(int* ma, int* mb) {
2   if (*ma >= *mb) return ma; else return mb;
3 }
4 bool inc_max(int a, int b) {
5   { int* mc = take_max(&a, &b); // borrow a and b
6     *mc += 1; }                 // deadline of both borrows
7   return (a != b);
8 }
```

Figure 1 illustrates which alias/variable has the permission to update the integer objects of a and b during the execution of `inc_max(5, 3)`. In `inc_max`, on line 5, the permission to update the integer object of a is *borrowed* by a newly taken pointer, or *mutable reference*, &a, and we similarly perform borrowing on b. For both borrows, the deadline is set to the end of the inner block (} in line 6). Until the deadline, the two references have the update permission on the integer objects whereas the lenders a and b temporarily lose all the permission. Now the function `take_max` is called with

[5] This terminology is standard but a bit confusing, since the word 'mutable/immutable' describes the property of the target object of the reference, rather than the property of the reference itself. Another terminology is 'unique/shared reference', which can be less confusing.

arguments &a and &b. The function take_max takes two integer mutable references ma and mb and returns the one with the larger target value. An interesting point is that the returned address is determined by a dynamic condition. After the call, we get a mutable reference mc, which points to the integer object of either a or b. The reference mc owns the update permission until the end of the inner block; this property is inherited from ma and mb. In line 6, mc increments the integer object by *mc += 1. Just after the deadline (in line 7), a and b retrieve the update permission to their integer objects and thus can read from them. Here, we check if a != b holds. The property we want to verify on this program is that inc_max returns true for any inputs a and b. It holds because, by incrementing the larger side of a and b, the difference between a and b increases by one.

Rust achieves the permission control described above using an elaborate type system. In particular, Rust uses the notion of *lifetime* to statically manage the deadline of each borrow. The type system of Rust will be discussed more in depth in §2.

The Key Idea of Our Reduction. Although Rust's permission-based type system cleverly ensures memory safety, we wish to verify a more fine-grained property, *functional correctness*. For smooth verification, we leverage Rust's permission guarantees to reduce Rust programs into CHCs without explicit representation of pointers and memory states. A naive approach would be to model each pointer as the value of its target object. However, if we do so, the lender of a *mutable borrow* do not know the value of the borrowed object just after the deadline. For example, if we took this naive approach for the take_max/inc_max program, we would get CHCs of the following form.

$$\begin{aligned} &\mathit{TakeMax}(a,b,r) \iff \text{if } a \ge b \text{ then } r = a \text{ else } r = b \\ &\mathit{IncMax}(a,b,r) \iff \exists c, c'.\ \mathit{TakeMax}(a,b,c) \land c' = c+1 \land r = (?\ !=\ ?) \\ &r = \text{true} \iff \mathit{IncMax}(a,b,r). \end{aligned}$$

The problem is, we do not know how to represent the values of a and b after the deadline of the borrows. There is no way to fill the parts ? in the second CHC. So we need a better way to model mutable references.

The key idea of our method is to represent a *mutable reference* ma as a pair $\langle a, a_\circ \rangle$ consisting of the values of the target object of *ma at two time points — the current value a and the *value at the deadline of the borrow* $a_\circ$. The trick is that we access some *future information* $a_\circ$, which is related to the notion of *prophecy variable* [1, 36, 73].

For example, our approach reduces the previous verification problem to the following system of CHCs.

$$\begin{aligned} &\mathit{TakeMax}(\langle a,a_\circ\rangle, \langle b,b_\circ\rangle, r) \iff \text{if } a \ge b \\ &\qquad\qquad \text{then } b_\circ = b \land r = \langle a,a_\circ\rangle \text{ else } a_\circ = a \land r = \langle b,b_\circ\rangle \\ &\mathit{IncMax}(a,b,r) \iff \exists a_\circ, b_\circ, c, c_\circ, c'.\ \mathit{TakeMax}(\langle a,a_\circ\rangle, \langle b,b_\circ\rangle, \langle c,c_\circ\rangle) \\ &\qquad\qquad\qquad\qquad \land c' = c+1 \land c_\circ = c' \land r = (a_\circ \mathrel{!=} b_\circ) \\ &r = \text{true} \iff \mathit{IncMax}(a,b,r). \end{aligned}$$

The mutable reference ma is now represented as $\langle a, a_\circ \rangle$, and similarly for mb and mc. In the then part of the first CHC, we have the constraint $b_\circ = b$, because now we throw away mb and thus the final target value $b_\circ$ of mb is now set to the current target value b. The constraint $r = \langle a, a_\circ \rangle$ corresponds to **return** ma in the program. The same reasoning applies to the else part of the first CHC. In the second CHC, the mutable reference mc is modeled as the pair $\langle c, c_\circ \rangle$. After incrementing the value of mc (expressed by $c' = c + 1$), the borrowed update permission of mc is released, which is expressed by $c_\circ = c'$. Now, the final check a != b is simply modeled as $a_\circ \mathrel{!=} b_\circ$, because the new values of a and b are available as $a_\circ$ and $b_\circ$. The important point is that both the values $a_\circ$ and $b_\circ$ have been determined at this point; one is determined in *TakeMax* (by either $b_\circ = b$ or $a_\circ = a$), and the other is determined in *IncMax* by $c_\circ = c'$. For example, in evaluating `inc_max(5, 3)` (as in Figure 1), the pointers `ma` and `mb` passed to `take_max` are modeled as $\langle 5, 6\rangle$ and $\langle 3, 3\rangle$ respectively. Although the verified program uses pointer manipulation, the system of CHCs obtained by our reduction is free of complex features like arrays, and thus can easily be solved by many CHC solvers.

Also, our reduction turns the verification problem on `just_rec` discussed in §1.1 into the following system of pretty simple CHCs.

$$\begin{aligned}
&\mathit{JustRec}(\langle a, a_\circ\rangle, r) \;\Longleftarrow\; (a_\circ = a \;\land\; r = \mathrm{true}) \;\lor \\
&\qquad\qquad\qquad\qquad (\,\exists b, b_\circ.\; \mathit{JustRec}(\langle b, b_\circ\rangle, \_) \;\land\; a_\circ = a \;\land\; r = (a == a_\circ)\,) \\
&r = \mathrm{true} \;\Longleftarrow\; \mathit{JustRec}(\langle a, a_\circ\rangle, r).
\end{aligned}$$

This CHC system has a very simple solution $\mathit{JustRec}(ma, r) \;:\Longleftrightarrow\; r = \mathrm{true}$, which can easily be found by standard CHC solvers. Remarkably, unlike the array-based reduction discussed in §1.1, the CHC system output by our reduction is free of arrays and its solution does not require quantifiers.

Our reduction can be flexibly applied to various features of Rust, such as reborrowing, nested references, and recursive data types. Our approach can reduce a substantial subset of Rust to CHCs in a fairly uniform manner. In §3.4, we present some advanced examples of our verification method. Example 5 presented there features a Rust program that handles a mutable reference to a singly linked list, where our reduction experimentally succeeded in automated verification of a fairly challenging property.

Formalizing Our Reduction. Later in §2 and §3, we formalize (a subset of) Rust and our reduction. Here we provide an informal overview of the formalization.

As a running example, we reuse the `take_max`/`inc_max` program discussed earlier. In Rust, the program is written as follows. To aid understanding, we added some ghost annotations in cyan.

```
fn take_max<'a>(ma: &'a mut i32, mb: &'a mut i32)-> &'a mut i32 {
  if *ma >= *mb { ma } else { mb }
}
fn inc_max(mut a: i32, mut b: i32) -> bool {
  { let mc = take_max<'l>(&'l mut a, &'l mut b);  *mc += 1; }('l)
  a != b
}
```

The type `i32` represents a (32-bit) integer. The type `&'a mut i32` represents a *mutable reference* to an integer that is governed under the *lifetime* `'a`, which represents the deadline of a borrow.[6] In Rust, the *permission* of each pointer is expressed in the type. The function `take_max` takes two integer mutable references of some lifetime `'a` and returns an integer mutable reference of the lifetime `'a` (the function is parametrized over `'a`). In the function `inc_max`, we perform borrowing. The time point at the end of the inner scope is named `'l` here. We mutably borrow the integer variables `a` and `b` under this lifetime `'l`, and pass them to the function `take_max`. The output `mc` has the type `&'l mut i32`.[7]

For formalization, we use a normalized program like below, where each function body is decomposed into a set of simple *instructions* labeled by *program points*. This is also similar to an intermediate representation used by the Rust compiler, which is called *MIR (mid-level intermediate*

[6] In the standard terminology of Rust, a lifetime often means a time range where a borrow is active. In this paper, however, we use the term lifetime to refer to the time point when a borrow ends.

[7] In Rust, we do not need to (and actually cannot) write annotations on *local lifetimes*, like `'l` used above. The Rust compiler performs a very clever inference on local lifetimes.

representation) [54]. This representation is convenient for the formalization of the type system and the reduction to CHCs.

```
fn take_max<'a>(ma: &'a mut i32, mb: &'a mut i32)-> &'a mut i32 {
  if T0: *ma >= *mb { T1: ma } else { T2: mb }
}
fn inc_max(mut a: i32, mut b: i32) -> bool {
  { I0: let ma = &'l mut a;  I1: let mb = &'l mut b;
    I2: let mc = take_max<'l>(ma, mb);  I3: *mc += 1;  I4: }('l)
  I5: a != b
}
```

In the function `take_max`, we jump from `T0` to either `T1` or `T2` depending on the condition `*ma >= *mb`. In the function `inc_max`, the function call `let mc = take_max(&mut a, &mut b);` is decomposed into three instructions, namely those at `I0`, `I1` and `I2`. For convenience of explanation, we set a program point `I4` at the end of the inner scope.

Now we describe how Rust's type system works. The type system of Rust gives some *permission* to a pointer, which changes in the process of execution. As a result, the type system is *flow-sensitive* and assigns a different type context to each program point. For example, the following is the function `inc_max` with the type context assigned to each program point.

```
fn inc_max(mut a: i32, mut b: i32) -> bool {
  { I0: {a, b: i32} let ma = &'l mut a;
    I1: {ma: &'l mut i32; a:['l] i32; b: i32} let mb = &'l mut b;
    I2: {ma, mb: &'l mut i32; a, b:['l] i32}
                                      let mc = take_max<'l>(ma, mb);
    I3: {mc: &'l mut i32; a, b:['l] i32} *mc += 1;
    I4: {a, b:['l] i32} }('l)
  I5: {a, b: i32} a != b
}
```

The variable `a` temporarily *loses the permission* on its integer object until the deadline of the borrow `'l`, which we say that `a` is *frozen* under the lifetime `'l`. A similar thing applies to `b`. The type context has the information about which variables are frozen under which lifetime. (In the notation used above, `a:['l] i32` means that `a` is typed `i32` but frozen under `'l`.) When we move from `I4` to `I5`, the lifetime `'l` comes and thus the variables `a` and `b` retrieve the permission.

Now we sketch the formalization of our reduction to CHCs. Our reduction of Rust programs to CHCs is *type-directed*, in that it leverages the type assigned to each variable by Rust's type system to decide the model of the variable. For example, a reference of the type `&mut i32` like `ma` is modeled as a pair of the current and final integer values, whereas an integer variable like `a`, which is essentially a pointer to an integer object, is modeled simply as its target value. In our formalized reduction, for each program point, we introduce a predicate variable and generate a CHC that models the instruction at the point. For example, the function `take_max` is reduced to the following CHCs, where three predicate variables T_0, T_1 and T_2 represent the program points `T0`, `T1` and `T2`.

$$T_0(\langle a, a_\circ\rangle, \langle b, b_\circ\rangle, r) \iff \text{if } a \geq b \text{ then } T_1(\langle a, a_\circ\rangle, \langle b, b_\circ\rangle, r) \text{ else } T_2(\langle a, a_\circ\rangle, \langle b, b_\circ\rangle, r)$$

$$T_1(ma, \langle b, b_\circ\rangle, r) \iff b_\circ = b \land r = ma$$

$$T_2(\langle a, a_\circ\rangle, mb, r) \iff a_\circ = a \land r = mb$$

The predicate variable for each program point models the relation between the values of the local variables in that point and the return value of the function the point belongs to. For example, the predicate variable T_1 models the relation between the values of `ma` and `mb` ($ma, \langle b, b_\circ \rangle$) and the return value of `take_max` (r). At `T1`, we release a mutable reference `mb`, which is modeled as $\langle b, b_\circ \rangle$. In order to ensure that what we took as the final value $b_\circ$ agrees with the actual final value, we add the constraint $b_\circ = b$ here. The function `inc_max` is reduced to the following CHCs.

$$\begin{aligned}
I_0(a, b, r) &\Longleftarrow I_1(\langle a, a_\circ \rangle, a_\circ, b, r) \\
I_1(ma, a, b, r) &\Longleftarrow I_2(ma, \langle b, b_\circ \rangle, a, b_\circ, r) \\
I_2(ma, mb, a, b, r) &\Longleftarrow T_0(ma, mb, mc) \land I_3(mc, a, b, r) \\
I_3(\langle c, c_\circ \rangle, a, b, r) &\Longleftarrow I_4(\langle c+1, c_\circ \rangle, a, b, r) \\
I_4(\langle c, c_\circ \rangle, a, b, r) &\Longleftarrow c_\circ = c \land I_5(a, b, r) \\
I_5(a, b, r) &\Longleftarrow r = (a \mathrel{!=} b)
\end{aligned}$$

At `I0`, we borrow `a` and obtain a mutable reference `ma`. Here, we take a *fresh variable* $a_\circ$ for the final target value, i.e., the value of the borrowed integer object at the deadline of the borrow. We model the *mutable reference* `ma` as $\langle a, a_\circ \rangle$. Now we can simply model `a` as $a_\circ$ here, because the type system ensures that `a` cannot be accessed, or is *frozen*, until the deadline of the borrow `'l`. At `I3`, we perform a destructive update, incrementing the target integer of `mc`. Here, letting $\langle c, c_\circ \rangle$ be the value of `mc` at `I3`, we set `mc`'s value at `I4` to $\langle c+1, c_\circ \rangle$. At `I5`, we can access `a` and `b` now because the lifetime `'l` is over. We can simply use the first argument a of `I_5` for the value of `a` here, because it was set to the final target value $a_\circ$ when we performed the borrow at `I0`.

Contributions. We have developed a novel method of reducing Rust programs to CHCs that leverages permission guarantees provided by Rust's type system, as introduced above. We have formalized our reduction of Rust programs to CHCs on a newly formalized core language of Rust and proved the soundness and completeness of this reduction. We have also implemented a prototype automated verifier for the core of Rust based on the idea and confirmed the effectiveness of our approach through preliminary experiments. The core language we support includes particularly reborrow and recursive types. Our approach has succeeded in automated verification of some non-trivial properties of programs with destructive update via pointers on recursive data types like lists and trees.

This article is a revised and extended version of the same-titled paper published in the proceedings of ESOP 2020 [49]. Compared with the conference version, in this article we have augmented explanations, polished the formalization, added the proof of the main theorem, and expanded the experiments.

Structure of the Rest of the Paper. In §2, we provide a formalized core of Rust. In §3, we formalize our reduction from programs to CHCs and outline the proof of its soundness and completeness; we also introduce advanced examples on our reduction and discuss extension of our method. In §4, we give the complete proof of the soundness and completeness of our reduction. In §5, we report on the implementation and the experimental results. In §6 we discuss related work and in §7 we conclude the paper.

2 FORMALIZATION OF RUST: CALCULUS OF OWNERSHIP AND REFERENCE

Now we present our formalization of the core of Rust, which we call *Calculus of Ownership and Reference (COR).* It is a typed procedural calculus with a lifetime-based permission control system in the style of Rust. Its design is inspired by λ_{Rust} [34].

The calculus is carefully designed to simplify our reduction of Rust programs into CHCs (presented in §3) and the proof of its soundness and completeness (given in §4). For simplicity, we impose some restrictions in this calculus. Lifetime information, type information and data releases should be explicitly annotated in the program. Each function body should be written as a set of primitive commands connected with goto jumps. Also, each variable should be a pointer. Still, the calculus covers various features of Rust, as we see later.

In §2.1 we introduce the syntax of this calculus. Then we present the type system in §2.2 and the operational semantics in §2.3.

Notation. An arrow above a variable denotes a sequence (e.g., $\vec{x} = x_0, \dots, x_{n-1}$). It can be used in a composite form; for example, $\overrightarrow{x\colon T}$ denotes $x_0\colon T_0,\ \dots,\ x_{n-1}\colon T_{n-1}$. The empty sequence can be denoted by ϵ. Also, the length of a sequence can be specified with a superscript (e.g., $\vec{x}^N = x_0, \dots, x_{N-1}$). We sometimes omit commas used as separators in a sequence.

The set operation $A + B$ (or more generally $\sum_\lambda A_\lambda$) denotes the disjoint union, i.e., the union $A \cup B$ (or $\bigcup_\lambda A_\lambda$) defined only if the arguments are disjoint. The set operation $A - B$ denotes the proper set difference, i.e., the set difference $A \setminus B$ that is defined only if $A \supseteq B$.

2.1 Syntax

The syntax of this calculus is as follows.

$$\text{(program)}\ \ \Pi\ ::=\ \vec{F}\ \ \text{(function definitions)}$$

$$\text{(function definition)}\ \ F\ ::=\ \mathsf{fn}\ f\ \Sigma\ \{\,\overrightarrow{L\colon S}\,\}\ \ \text{(name, signature, and labeled statements)}$$

$$\text{(function signature)}\ \ \Sigma\ ::=\ \langle \vec{\alpha} \mid \overrightarrow{\alpha_a \le \alpha_b} \rangle\ (\overrightarrow{x\colon T}) \to U\ \ \text{(lifetime params. and constraints, inputs, and return type)}$$

$$\begin{aligned}
\text{(statement)}\ \ S\ ::=\ & I;\ \mathsf{goto}\,L\ \ \text{(perform } I \text{ and jump to } L)\ \mid\ \mathsf{return}\,x\ \ \text{(return from a function with } x)\\
\mid\ & \mathsf{match}\ {*x}\ \{\ \mathsf{inj}_0{*y_0} \to \mathsf{goto}\,L_0,\ \mathsf{inj}_1{*y_1} \to \mathsf{goto}\,L_1\ \}\ \ \text{(conditionally branch by the tag)}
\end{aligned}$$

$$\begin{aligned}
\text{(instruction)}\ \ I\ ::=\ & \mathsf{let}\,y = \mathsf{mutbor}_\alpha\,x\ \ \text{(mutably (re)borrow)}\ \mid\ \mathsf{drop}\,x\ \ \text{(release a variable and its target object)}\\
\mid\ & \mathsf{immut}\,x\ \ \text{(weaken a mutable reference immutable)}\ \mid\ \mathsf{swap}({*x}, {*y})\ \ \text{(swap target objects)}\\
\mid\ & \mathsf{let}\,{*y} = x\ \ \text{(create a pointer)}\ \mid\ \mathsf{let}\,y = {*x}\ \ \text{(dereference a pointer)}\\
\mid\ & \mathsf{let}\,{*y} = \mathsf{copy}\,{*x}\ \ \text{(copy the target object)}\ \mid\ x\ \mathsf{as}\ T\ \ \text{(re-type a variable)}\\
\mid\ & \mathsf{let}\,y = f\langle\vec{\alpha}\rangle(\vec{x})\ \ \text{(call a function)}\ \mid\ \mathsf{intro}\,\alpha\ \ \text{(introduce a lifetime var.)}\\
\mid\ & \mathsf{now}\,\alpha\ \ \text{(eliminate a lifetime var.)}\ \mid\ \alpha \le \beta\ \ \text{(promise ordering on lifetime vars.)}\\
\mid\ & \mathsf{let}\,{*y} = \mathit{const}\ \ \text{(get a constant)}\ \mid\ \mathsf{let}\,{*y} = {*x}\ \mathit{op}\ {*x'}\ \ \text{(get the integer operation result)}\\
\mid\ & \mathsf{let}\,{*y} = \mathsf{rand}()\ \ \text{(get a random integer)}\ \mid\ \mathsf{let}\,{*y} = \mathsf{inj}_i^{T_0+T_1}\,{*x}\ \ \text{(create a variant)}\\
\mid\ & \mathsf{let}\,{*y} = ({*x}, {*x'})\ \ \text{(create a pair)}\ \mid\ \mathsf{let}\,({*y}, {*y'}) = {*x}\ \ \text{(destruct a pair)}
\end{aligned}$$

$$\begin{aligned}
\text{(type)}\ \ T, U\ ::=\ & P\,T\ \ \text{(pointer type)}\ \mid\ T + T'\ \ \text{(variant type)}\ \mid\ T \times T'\ \ \text{(pair type)}\ \mid\\
\mid\ & X\ \ \text{(type variable)}\ \mid\ \mu X.T\ \ \text{(equi-recursive type)}\ \mid\ \mathsf{int}\ \ \text{(integer type)}\ \mid\ \mathsf{unit}\ \ \text{(unit type)}
\end{aligned}$$

$$\text{(pointer kind)}\ \ P\ ::=\ \mathsf{own}\ \ \text{(owning pointer)}\ \mid\ R_\alpha\ \ \text{(reference)}$$

$$\text{(reference kind)}\ \ R\ ::=\ \mathsf{mut}\ \ \text{(mutable)}\ \mid\ \mathsf{immut}\ \ \text{(immutable)}$$

$$\alpha, \beta, \gamma\ \ \text{(lifetime var.)}\qquad X, Y\ \ \text{(type var.)}\qquad x, y\ \ \text{(data var.)}\qquad f, g\ \ \text{(function name)}\qquad L\ \ \text{(label)}$$

$$\text{(constant)}\ \ \mathit{const}\ ::=\ n\ \ \text{(integer)}\ \mid\ ()\ \ \text{(unit)}\qquad \mathsf{bool}\ :=\ \mathsf{unit} + \mathsf{unit}$$

(integer operator) $op ::= op_{\text{int}} \mid op_{\text{bool}} \qquad op_{\text{int}} ::= + \mid - \mid \cdots \qquad op_{\text{bool}} ::= \texttt{>=} \mid \texttt{==} \mid \texttt{!=} \mid \cdots$

We also use a meta-variable $\check{P}$ for a non-mutable-reference pointer kind, i.e., own or immut_α.

Program, Function and Statement. A *program* Π is a sequence of function definitions $\vec{F}$. For simplicity, here we do not specify the entry function (i.e., the `main` function in Rust and C/C++).

A *function definition* F consists of the name f, the signature Σ and the body $\overrightarrow{L\colon S}$. A function is parametrized over constrained lifetime parameters, but for simplicity our calculus does not support polymorphism over types, like λ_{Rust} [34]. For simplicity, the input/output types of a function are restricted to *pointer types*, i.e., types of the form $P\,T$. In a function signature, we simplify $\langle \vec{\alpha} \mid \rangle$ to $\langle \vec{\alpha} \rangle$ and omit $\langle \mid \rangle$.

A *label* L is a program point that contains a *statement* S, which performs some simple command and jumps to some label or return from the function. Later, this style with labels and unstructured control flow simplifies the formalization of our reduction in §3.2. We require that the function body contains the entry point label entry. Also, we require that every label in the function is syntactically reachable from the label entry (i.e., reachable in the directed graph whose vertices are the labels and whose edges are goto jumps); this restriction is for uniqueness of typing, as we see in §2.2. There are three types of statements. A statement $I; \mathsf{goto}\, L$ performs the instruction I and jump to the label L. A statement $\mathsf{return}\, x$ returns from the function with the variable x. A statement $\mathsf{match}\, {*x}\, \{ \overrightarrow{\mathsf{inj}\, {*y} \to \mathsf{goto}\, L} \}$ conditionally branches to a label L_i by the tag of the variant $*x$ and take a pointer y_i to the body of the variant.

Instruction. An *instruction* I performs a simple command. We have various types of instructions, whose meanings are briefly explained above.

For most kinds of instructions, the inputs are *consumed*. Only for the copy instruction $\mathsf{let}\, {*y} = \mathsf{copy}\, {*x}$ and the operation instruction $\mathsf{let}\, {*y} = {*x_0}\ op\ {*x_1}$, the inputs are not consumed.

The swap instruction $\mathsf{swap}({*x}, {*y})$ takes pointers x and y and swaps the target objects of x and y. An unusual design of this calculus is that it uses swap instead of assignment for the primitive for destructive update. Assignment is a bit trickier than swap in terms of resource management, because when some object is assigned to a variable, the old object of the variable is implicitly released. We can still express assignment combining a number of instructions. For example, if we have a pointer px to an integer and wish to assign its integer value to a mutable reference my, we can do that by the following sequential execution: $\mathsf{let}\, {*px'} = \mathsf{copy}\, {*px};\ \mathsf{swap}({*my}, {*px'});\ \mathsf{drop}\, px'$.

Pointer, Borrow and Lifetime. A pointer can be either an *owning pointer* or a *reference*. An *owning pointer* models Rust's box pointer `Box<T>`. It can freely update, read and release its target object. As informally explained in §1.2, a *mutable* or *immutable reference* is a pointer that targets an object owned by some owning pointer and has the update or read permission to the object under until some *lifetime*. We use lifetime variables α, β, γ to denote lifetimes. A lifetime variable can be either (i) a *lifetime parameter* taken by a function or (ii) a *local lifetime* introduced within a function.

By the instruction $\mathsf{let}\, y = \mathsf{mutbor}_\alpha\, x$, we mutably borrow x under the lifetime α and obtain a mutable reference y. Here, x can be either an (unfrozen) owning pointer or mutable reference (when x is a mutable reference, this operation is called a *reborrow*). Also, by the instruction $\mathsf{immut}\, x$, we can weaken a mutable reference x into an immutable reference.

We have three lifetime-related ghost instructions. The instruction $\mathsf{intro}\, \alpha$ introduces a local lifetime α. The instruction $\mathsf{now}\, \alpha$ sets a local lifetime α to the current moment and eliminates it. The instruction $\alpha \le \beta$ promises that $\mathsf{now}\, \alpha$ comes earlier than $\mathsf{now}\, \beta$ in the process of computation.

We can *subdivide* pointers in various ways. The instruction $\mathsf{let}\, ({*y}, {*y'}) = {*x}$ splits a pointer x to a pair into pointers y, y' to each element of the pair. The statement $\mathsf{match}\, {*x}\, \{ \overrightarrow{\mathsf{inj}\, {*y} \to \mathsf{goto}\, L} \}$

turns a pointer x to a variant into a pointer y_i to the body object of the variant, discarding the permission to the tag of the variant. The instruction $\mathsf{let}\, y = *x$ takes a pointer to a pointer x and returns a pointer y to the inner target object of x, which can also be regarded as a pointer subdivision.

Type. In the calculus, various forms of *types* T are supported, whose meanings are briefly explained above. A *pointer type* has the form $P\,T$, where P is called the *pointer kind*. The type of an owning pointer is $\mathsf{own}\, T$, which corresponds to Rust's `Box<T>` (or simply `T`). The types of a mutable reference and an immutable reference are $\mathsf{mut}_\alpha\, T$ and $\mathsf{immut}_\alpha\, T$, which correspond to `&'a mut T` and `&'a T` in Rust.

We say that a type is *complete* if it satisfies the following: every occurrence of a type variable in T should be bound by the recursive binder μ and guarded by a pointer constructor P inside the binder. A type T that appears in a program (not just as a substructure of some type) should be complete. For example, the singly linked integer list type $\mu X.\, \mathsf{unit} + \mathsf{int} \times \mathsf{own}\, X$ is complete, whereas $\mu X.\, \mathsf{unit} + \mathsf{int} \times X$ (without own) is not complete.

Remark 1 (Expressivity and Limitations). Although older versions of Rust determined lifetimes just by lexical scopes, the current versions of Rust have a mechanism that overcomes that restriction, which is called *non-lexical lifetime* [57]. The Rust borrow checker uses a flow sensitive analysis to determine the lifetimes of references and allows many flexible borrow patterns. Our calculus can the core behavior of non-lexical lifetimes. The point is that, even under non-lexical lifetimes, the set of program points where a borrow is active forms a continuous range. [8]

A major limitation of our calculus is that it does not support *unsafe code blocks* and also lacks *type traits and closures*. How to overcome them is discussed later in §3.5. Another limitation of COR is that, unlike Rust and λ_{Rust}, we do not have a primitive for directly modifying or borrowing a substructure of a variable (e.g., the first element of a pair). Still, combining some operations, we can modify or borrow a substructure by borrowing the whole variable first and then subdividing pointers (e.g., $\mathsf{let}\,(*y, *y') = *x$). Nevertheless, this borrow-and-subdivide strategy cannot fully support some advanced borrow patterns like `get_default` in 'Problem Case #3' of [57].

Example 1 (Program). The Rust program with `take_max` and `inc_max` presented in §1.2 is modeled as follows in this calculus.

$$
\begin{array}{l}
\mathsf{fn\ take\text{-}max}\ \langle\alpha\rangle\ (ma\colon \mathsf{mut}_\alpha\ \mathsf{int},\ mb\colon \mathsf{mut}_\alpha\ \mathsf{int}) \to \mathsf{mut}_\alpha\ \mathsf{int}\ \{\\
\quad \mathsf{entry\colon\ let}\ *ord = *ma >= *mb;^{\mathsf{L1}}\ \mathsf{match}\ *ord\ \{\ \mathsf{inj}_1 *ou \to \mathsf{goto\ L2},\ \mathsf{inj}_0 *ou \to \mathsf{goto\ L5}\ \}\\
\quad \mathsf{L2\colon\ drop}\ ou;^{\mathsf{L3}}\ \mathsf{drop}\ mb;^{\mathsf{L4}}\ \mathsf{return}\ ma \qquad \mathsf{L5\colon\ drop}\ ou;^{\mathsf{L6}}\ \mathsf{drop}\ ma;^{\mathsf{L7}}\ \mathsf{return}\ mb\\
\}
\end{array}
$$

$$
\begin{array}{l}
\mathsf{fn\ inc\text{-}max}(oa\colon \mathsf{own\ int},\ ob\colon \mathsf{own\ int}) \to \mathsf{own\ bool}\ \{\\
\quad \mathsf{entry\colon\ intro}\ \alpha;^{\mathsf{L1}}\ \mathsf{let}\ ma = \mathsf{mutbor}_\alpha\ oa;^{\mathsf{L2}}\ \mathsf{let}\ mb = \mathsf{mutbor}_\alpha\ ob;^{\mathsf{L3}}\\
\quad\ \ \mathsf{let}\ mc = \mathsf{take\text{-}max}\langle\alpha\rangle(ma, mb);^{\mathsf{L4}}\\
\quad\ \ \mathsf{let}\ *o1 = 1;^{\mathsf{L5}}\ \mathsf{let}\ *oc' = *mc + *o1;^{\mathsf{L6}}\ \mathsf{drop}\ o1;^{\mathsf{L7}}\ \mathsf{swap}(mc, oc');^{\mathsf{L8}}\ \mathsf{drop}\ oc';^{\mathsf{L9}}\ \mathsf{drop}\ mc;^{\mathsf{L10}}\\
\quad\ \ \mathsf{now}\ \alpha;^{\mathsf{L11}}\ \mathsf{let}\ *or = *oa\ != *ob;^{\mathsf{L12}}\ \mathsf{drop}\ oa;^{\mathsf{L13}}\ \mathsf{drop}\ ob;^{\mathsf{L14}}\ \mathsf{return}\ or\\
\}
\end{array}
$$

The first letter of a variable name indicates the pointer kind (o for an owning pointer and m for a mutable reference). We swapped the two branches of the match statement in take-max to make

[8] Strictly speaking, this property is broken by recently adopted (implicit) two-phase borrows [56, 65]. However, by shallow syntactical reordering, a program with two-phase borrows can be fit into usual borrow patterns.

the order the same as if-else branching. We use shorthand for sequential execution; for example, $L_0\colon I_0;^{L_1} I_1;\ \mathrm{goto}\, L_2$ denotes for two labeled statements $L_0\colon I_0;\ \mathrm{goto}\, L_1$ and $L_1\colon I_1;\ \mathrm{goto}\, L_2$.

Here we have more program points than in the informal description of §1.2. Each time we release a variable x we need an instruction drop x, which simplifies formalization. Also, we use ghost instructions intro α and now α to manage lifetimes.

2.2 Type System

The type system assigns a *whole context* $(\boldsymbol{\Gamma}, \mathbf{A})$ to each label (program point) of each function in a program. A whole context is a pair of a *data context* $\boldsymbol{\Gamma}$ and a *lifetime context* $\mathbf{A}$. A data context manages the information on data variables and a lifetime context manages the information on lifetime variables. These notions are explained in more detail soon.

Context. A *data context* $\boldsymbol{\Gamma}$ is a finite set of items of the form $x{:}^{\mathbf{a}}\, T$, where T should be a complete *pointer* type and $\mathbf{a}$ (which we call *activeness*) is of the form either 'actv' (active; i.e., the permission is not borrowed) or '$\dagger\alpha$' (*frozen* until lifetime α; i.e., the permission is borrowed until α). For simplicity, we do not consider the situation where only the write permission of a variable is frozen. When a variable x is tagged $x{:}^{\dagger\alpha}$, we cannot read or update the target object of x through x. We usually abbreviate $x{:}^{\mathrm{actv}}\, T$ to $x\colon T$. A data context should not contain two items on the same variable.

A *lifetime context* $\mathbf{A} = (A, R)$ is a finite preordered set of lifetime variables, where A is the underlying set and R is the preorder. We write $|\mathbf{A}|$ and $\leq_{\mathbf{A}}$ to refer to A and R.

Finally, a *whole context* $(\boldsymbol{\Gamma}, \mathbf{A})$ is a pair of a data context $\boldsymbol{\Gamma}$ and a lifetime context $\mathbf{A}$ such that every lifetime variable in $\boldsymbol{\Gamma}$ is contained in $\mathbf{A}$.

Auxiliary Judgments. The subtyping judgment is of the form $\mathbf{A}, \Xi \vdash T \leq U$, where Ξ is a finite set of assumptions of the form $X \leq Y$, which are used for *coinductive* reasoning on recursive types. The common subtyping judgment $\mathbf{A} \vdash T \leq U$ is defined as $\mathbf{A}, \varnothing \vdash T \leq U$, where we have no assumptions. The full subtyping judgment $\mathbf{A}, \Xi \vdash T \leq U$ is defined by the following rules.

$$\frac{X \leq Y \in \Xi}{\mathbf{A}, \Xi \vdash X \leq Y} \qquad \frac{\mathbf{A}, \Xi \vdash T \leq U}{\mathbf{A}, \Xi \vdash \check{P}\, T \leq \check{P}\, U} \qquad \frac{\mathbf{A}, \Xi \vdash T \leq U,\ U \leq T}{\mathbf{A}, \Xi \vdash \mathsf{mut}_\alpha T \leq \mathsf{mut}_\alpha U} \qquad \frac{\beta \leq_{\mathbf{A}} \alpha}{\mathbf{A}, \Xi \vdash R_\alpha T \leq R_\beta T}$$

$$\frac{\mathbf{A}, \Xi \vdash T_0 \leq U_0,\ T_1 \leq U_1}{\mathbf{A}, \Xi \vdash T_0 + T_1 \leq U_0 + U_1} \qquad \frac{\mathbf{A}, \Xi \vdash T_0 \leq U_0,\ T_1 \leq U_1}{\mathbf{A}, \Xi \vdash T_0 \times T_1 \leq U_0 \times U_1}$$

$$\mathbf{A}, \Xi \vdash \mu X.T \leq T[\mu X.T/X] \qquad \mathbf{A}, \Xi \vdash T[\mu X.T/X] \leq \mu X.T$$

$$\frac{X', Y' \text{ are fresh in } \Xi \qquad \Xi + \{X' \leq Y'\} \vdash T[X'/X] \leq U[Y'/Y]}{\mathbf{A}, \Xi \vdash \mu X.T \leq \mu Y.U} \ (\textsc{Subtype-Rec-Covar})$$

$$\frac{X', Y' \text{ are fresh in } \Xi \qquad \Xi + \{X' \leq Y', Y' \leq X'\} \vdash T[X'/X] \leq U[Y'/Y],\ U[Y'/Y] \leq T[X'/X]}{\mathbf{A}, \Xi \vdash \mu X.T \leq \mu Y.U}$$

(Subtype-Rec-Invar)

$$\mathbf{A}, \Xi \vdash T \leq T \qquad \frac{\mathbf{A}, \Xi \vdash T \leq T',\ T' \leq T''}{\mathbf{A}, \Xi \vdash T \leq T''}$$

We have two rules for judging $\mu X.T \leq \mu Y.U$, Subtype-Rec-Covar and Subtype-Rec-Invar, which admit *coinductive* reasoning of a simple form. The former Subtype-Rec-Covar is provided for the case where X appears *covariantly* in T. For example, the judgment $\mu X.\, \mathsf{unit} + \mathsf{immut}_\alpha X \leq \mu Y.\, \mathsf{unit} + \mathsf{immut}_\beta Y$ holds when $\alpha \leq_{\mathbf{A}} \beta$ holds. The latter Subtype-Rec-Invar is provided for the case where X appears *invariantly* in T (e.g., X is under a mutable reference). For example, the judgment $\mu X.\, \mathsf{unit} + \mathsf{mut}_\alpha X \leq \mu Y.\, \mathsf{unit} + \mathsf{mut}_\beta Y$ holds when both $\alpha \leq_{\mathbf{A}} \beta$ and $\beta \leq_{\mathbf{A}} \alpha$ hold.

We also introduce the following copyability judgment $T\colon \mathsf{copy}$.

$$\mathsf{int}\colon \mathsf{copy} \qquad \mathsf{unit}\colon \mathsf{copy} \qquad \mathsf{immut}_\alpha\, T\colon \mathsf{copy} \qquad \frac{T\colon \mathsf{copy}}{\mu X.T\colon \mathsf{copy}} \qquad \frac{T, T'\colon \mathsf{copy}}{T + T'\colon \mathsf{copy}} \qquad \frac{T, T'\colon \mathsf{copy}}{T \times T'\colon \mathsf{copy}}$$

In short, $T\colon \mathsf{copy}$ means that the owning pointer own and mutable reference mut_β constructors do not occur in T except under the immutable reference constructor immut_α.

Typing Judgment for Instructions. The instruction typing judgment is of the form $\Pi, f, (\mathbf{\Gamma}, \mathbf{A}) \vdash I : (\mathbf{\Gamma}', \mathbf{A}')$. It means that, by running the instruction I in (Π, f) under the whole context $(\mathbf{\Gamma}, \mathbf{A})$, a renewed whole context $(\mathbf{\Gamma}', \mathbf{A}')$ is obtained. Below are the complete rules for the judgment. For brevity, we omit Π and f here (except in $A_{\mathrm{ex}\,f}$) because they are always fixed. Also, we additionally require that every variable can be used at most once in each instruction.

$$A_{\mathrm{ex}\,f}\text{: the set of lifetime parameters of } f$$

$$\frac{P = \mathsf{own}, \mathsf{mut}_\beta \qquad \alpha \notin A_{\mathrm{ex}\,f} \qquad \text{for each lifetime variable } \gamma \text{ in } P\,T,\ \ \alpha \le_{\mathbf{A}} \gamma}{(\mathbf{\Gamma} + \{x\colon P\,T\},\ \mathbf{A}) \vdash \mathsf{let}\, y = \mathsf{mutbor}_\alpha\, x : (\mathbf{\Gamma} + \{y\colon \mathsf{mut}_\alpha\, T,\ x\colon^{\dagger\alpha} P\,T\},\ \mathbf{A})} \quad \text{(Type-Inst-Mutbor)}$$

$$\frac{\text{if } T \text{ is of the form } \mathsf{own}\, U,\ U\colon \mathsf{copy} \text{ holds}}{(\mathbf{\Gamma} + \{x\colon T\},\ \mathbf{A}) \vdash \mathsf{drop}\, x : (\mathbf{\Gamma}, \mathbf{A})} \quad \text{(Type-Inst-Drop)}$$

$$(\mathbf{\Gamma} + \{x\colon \mathsf{mut}_\alpha\, T\},\ \mathbf{A}) \vdash \mathsf{immut}\, x : (\mathbf{\Gamma} + \{x\colon \mathsf{immut}_\alpha\, T\},\ \mathbf{A}) \quad \text{(Type-Inst-Immut)}$$

$$\frac{x\colon \mathsf{mut}_\alpha\, T,\ y\colon P\,T \in \mathbf{\Gamma} \qquad P = \mathsf{own}, \mathsf{mut}_\beta}{(\mathbf{\Gamma}, \mathbf{A}) \vdash \mathsf{swap}(*x, *y) : (\mathbf{\Gamma}, \mathbf{A})} \quad \text{(Type-Inst-Swap)}$$

$$(\mathbf{\Gamma} + \{x\colon T\},\ \mathbf{A}) \vdash \mathsf{let}\, *y = x : (\mathbf{\Gamma} + \{y\colon \mathsf{own}\, T\},\ \mathbf{A}) \quad \text{(Type-Inst-Own)}$$

$$(\mathbf{\Gamma} + \{x\colon P\,P'\,T\},\ \mathbf{A}) \vdash \mathsf{let}\, y = *x : (\mathbf{\Gamma} + \{y\colon (P \cdot P')\,T\},\ \mathbf{A}) \quad \text{(Type-Inst-Deref)}$$

$$P \cdot \mathsf{own} := P \qquad \mathsf{own} \cdot P := P \qquad R_\alpha \cdot R'_\beta := R''_\alpha \text{ where } R'' = \begin{cases} \mathsf{mut} & (R = R' = \mathsf{mut}) \\ \mathsf{immut} & (\text{otherwise}) \end{cases}$$

$$\frac{x\colon P\,T \in \mathbf{\Gamma} \qquad T\colon \mathsf{copy}}{(\mathbf{\Gamma}, \mathbf{A}) \vdash \mathsf{let}\, *y = \mathsf{copy}\, *x : (\mathbf{\Gamma} + \{y\colon \mathsf{own}\, T\},\ \mathbf{A})} \quad \text{(Type-Inst-Copy)}$$

$$\frac{\mathbf{A} \vdash T \le U}{(\mathbf{\Gamma} + \{x\colon T\},\ \mathbf{A}) \vdash x\ \mathsf{as}\ U : (\mathbf{\Gamma} + \{x\colon U\},\ \mathbf{A})} \quad \text{(Type-Inst-As)}$$

$$\frac{\begin{array}{c}\Sigma_g = \langle \vec{\alpha}' \mid \overrightarrow{\alpha'_a \le \alpha'_b} \rangle (\overrightarrow{x'\colon T'}) \to U' \\ \text{for each } j,\ \alpha_{a_j} \le_{\mathbf{A}} \alpha_{b_j} \qquad \text{for each } i,\ T_i = T'_i[\overrightarrow{\alpha/\alpha'}] \qquad U = U'[\overrightarrow{\alpha/\alpha'}]\end{array}}{(\mathbf{\Gamma} + \{\overrightarrow{x\colon T}\},\ \mathbf{A}) \vdash \mathsf{let}\, y = g\langle \vec{\alpha} \rangle (\vec{x}) : (\mathbf{\Gamma} + \{y\colon U\},\ \mathbf{A})} \quad \text{(Type-Inst-Call)}$$

$$(\mathbf{\Gamma}, (A, R)) \vdash \mathsf{intro}\, \alpha : \big(\mathbf{\Gamma}, (\{\alpha\} + A,\ \{\alpha\} \times (\{\alpha\} + A_{\mathrm{ex}\,f}) + R)\big) \quad \text{(Type-Inst-Intro)}$$

$$\frac{\alpha \notin A_{\mathrm{ex}\,f} \qquad R' = \{(\beta, \gamma) \in R \mid \beta \ne \alpha\} \qquad R' \text{ has no element of the form } (\_, \alpha)}{\big(\mathbf{\Gamma}, (\{\alpha\} + A,\ R)\big) \vdash \mathsf{now}\, \alpha \colon \big(\{\mathrm{thaw}_\alpha(x\colon^{\mathrm{a}} T) \mid x\colon^{\mathrm{a}} T \in \mathbf{\Gamma}\},\ (A, R')\big)} \quad \text{(Type-Inst-Now)}$$

$$\mathrm{thaw}_\alpha(x\colon^{\mathrm{a}} T) := \begin{cases} x\colon T & (\mathrm{a} = \dagger\alpha) \\ x\colon^{\mathrm{a}} T & (\text{otherwise}) \end{cases}$$

$$\frac{\alpha, \beta \notin A_{\mathrm{ex}\, f}}{(\mathbf{\Gamma}, (A, R)) \vdash \alpha \leq \beta \colon \big(\mathbf{\Gamma},\ (A,\ (\{(\alpha, \beta)\} \cup R)^+)\big)} \quad (\textsc{Type-Inst-LftIn})$$

$$(\mathbf{\Gamma}, \mathbf{A}) \vdash \mathsf{let}\, {*y} = \mathit{const} \colon (\mathbf{\Gamma} + \{y \colon \mathsf{own}\, T_{const}\},\ \mathbf{A}) \quad (\textsc{Type-Inst-Const})$$

$$T_{const} \colon\ \mathsf{int}\ \text{if}\ \mathit{const} = n\ \text{and}\ \mathsf{unit}\ \text{if}\ \mathit{const} = ()$$

$$\frac{x \colon P\, \mathsf{int},\ x' \colon P'\, \mathsf{int} \in \mathbf{\Gamma}}{(\mathbf{\Gamma}, \mathbf{A}) \vdash \mathsf{let}\, {*y} = {*x}\ \mathit{op}_T\ {*x'} \colon (\mathbf{\Gamma} + \{y \colon \mathsf{own}\, T\},\ \mathbf{A})} \quad (\textsc{Type-Inst-Op})$$

$$(\mathbf{\Gamma}, \mathbf{A}) \vdash \mathsf{let}\, {*y} = \mathsf{rand}() \colon (\mathbf{\Gamma} + \{y \colon \mathsf{own}\, \mathsf{int}\},\ \mathbf{A}) \quad (\textsc{Type-Inst-Rand})$$

$$(\mathbf{\Gamma} + \{x \colon \mathsf{own}\, T_i\},\ \mathbf{A}) \vdash \mathsf{let}\, {*y} = \mathsf{inj}_i^{T_0+T_1}\, {*x} \colon (\mathbf{\Gamma} + \{y \colon \mathsf{own}\, (T_0 + T_1)\},\ \mathbf{A}) \quad (\textsc{Type-Inst-Inj})$$

$$(\mathbf{\Gamma} + \{x \colon \mathsf{own}\, T,\ x' \colon \mathsf{own}\, T'\},\ \mathbf{A}) \vdash \mathsf{let}\, {*y} = ({*x}, {*x'}) \colon (\mathbf{\Gamma} + \{y \colon \mathsf{own}\, (T \times T')\},\ \mathbf{A}) \quad (\textsc{Type-Inst-Pair})$$

$$(\mathbf{\Gamma} + \{x \colon P\, (T \times T')\},\ \mathbf{A}) \vdash \mathsf{let}\, ({*y}, {*y'}) = {*x} \colon (\mathbf{\Gamma} + \{y \colon P\, T,\ y' \colon P\, T'\},\ \mathbf{A}) \quad (\textsc{Type-Inst-PairDestr})$$

For most instructions, the input variables are *consumed* and thus do not appear in the output type context. The typing rules above are defined so that an instruction has a unique type; more precisely, for any $(\Gamma, \mathbf{A})$ and I, there exists at most one whole context $(\mathbf{\Gamma}', \mathbf{A}')$ satisfying $(\Gamma, \mathbf{A}) \vdash I \colon (\mathbf{\Gamma}', \mathbf{A}')$.

The instruction $\mathsf{let}\, y = \mathsf{mutbor}_\alpha\, x$ mutably borrows an (unfrozen) owning pointer or mutable reference $x \colon P\, T$ under the lifetime α (Type-Inst-Mutbor). After the borrow, x gets *frozen* until α, being registered to the type context as $x \colon^{\dagger\alpha} P\, T$. We have a precondition that α is a *local variable* that is outlived by any lifetime in x. Because α is local, the borrow *ends within a function* and the created reference does not leak outside the function.

The instruction $\mathsf{drop}\, x$ removes x from the type context (Type-Inst-Drop). The precondition on $\mathsf{drop}\, x$ says that when the dropped variable is an owning pointer its target type should be copyable. This precondition does not weaken the expressivity because we can always satisfy this precondition by repeating subdivision of pointers beforehand (by dereference $\mathsf{let}\, y = {*x}$, pair destruction $\mathsf{let}\, ({*y}, {*y'}) = {*x}$ and variant destruction $\mathsf{match}\, {*x}\, \{\cdots\}$). Thanks to the precondition, we do not need nested releases of owning pointers in the operational semantics and can avoid adding complicated constraints on mutable references in our reduction.

The instruction $\mathsf{immut}\, x$ weakens a mutable reference x into an immutable reference (Type-Inst-Immut). Technically, this is a variant of $\mathsf{drop}\, x$ on a mutable reference x where we retain an immutable reference.

The instruction $\mathsf{swap}({*x}, {*y})$ destructively updates the targets of x and y, swapping the target objects (Type-Inst-Swap). To reduce the number of patterns to consider, we restrict x to a mutable reference, whereas we let y be either an owning pointer or a mutable reference. We do not lose expressivity by this; swap between two owning pointer variables can be performed by swapping just the names of the two variables.

The instruction $\mathsf{let}\, {*y} = x$ *consumes* the variable x and allocates its object to get an owning pointer y (Type-Inst-Own).

The instruction $\mathsf{let}\, y = {*x}$ dereferences a pointer to a pointer (Type-Inst-Deref). The kind of the output pointer type is determined from the outer and inner pointer kinds P, P' of the input, by an auxiliary operation $P \cdot P'$. The kind own is an identity on this operation. When we compose reference pointer kinds $R_\alpha \cdot R'_\beta$, the output is R''_α, where R'' is the weakest of R and R'. Here, we can just take the lifetime of the outer reference α, which is safe because when we performed a borrow we ensured that the lifetime is outlived by any lifetimes in the target.

The instruction let $*y = \mathsf{copy}\,*x$ copies the target object of x typed T and wraps it into an owning variable typed $\mathsf{own}\,T$ (TYPE-INST-COPY). We have a precondition $T\colon \mathsf{copy}$, which prevents us from copying mutable references and owning pointers.

The instruction x as U modifies the type T of a variable x into another type U (TYPE-INST-AS). We have a precondition $\mathbf{A} \vdash T \le U$ on subtyping.

The instruction let $y = g\langle\vec{\alpha}\rangle(\vec{x})$ calls a function g with lifetime arguments $\vec{\alpha}$ and data arguments $\vec{x}$. The inputs $\vec{x}$ are *consumed* by the function g and thus do not appear in the output type context.

The instruction intro α introduces a new local lifetime α (TYPE-INST-INTRO). We promise that the new local lifetime α is outlived by any lifetime parameters, because it will be eliminated in the current function. The instruction now α eliminates the local lifetime α and reactivates every variable frozen under α in the data context (TYPE-INST-NOW). As a precondition, we check that α is strictly the least element in the input local lifetime context, i.e., $\beta \le \alpha$ does not hold for any lifetime variable β other than α. The instruction $\beta \le \alpha$ adds a promise on the elimination order of local lifetimes (TYPE-INST-LFTIN). This promise is registered to the lifetime context and can be used for subtyping.

The instructions let $*y = \mathit{const}$, let $*y = *x\ \mathit{op}\,*x'$, and let $*y = \mathsf{rand}()$ respectively newly allocate a constant, the result of an integer operation, or a non-deterministic integer to get an owning pointer to the result y (TYPE-INST-CONST, TYPE-INST-OP, TYPE-INST-RAND). Note that the inputs x, x' are *not* consumed for the instruction let $*y = *x\ \mathit{op}\,*x'$.

The instructions let $*y = \mathsf{inj}_i^{T+T'}\,*x$ and let $*y = (*x, *x')$ respectively allocate a variant object or a pair by consuming the input owning pointer(s) (TYPE-INST-INJ, TYPE-INST-PAIR). The instruction let $(*y, *y') = *x$ *splits* a pointer to a pair into pointers to each element of the pair, retaining the pointer kind (TYPE-INST-PAIRDESTR). For example, by splitting a mutable reference to a pair, we get mutable references to each element of the pair.

Typing Judgment for Statements. The statement typing judgment is of the form $\Pi, f, (\mathbf{\Gamma}, \mathbf{A}) \vdash S\colon \{\overrightarrow{(L, (\mathbf{\Gamma}', \mathbf{A}'))}\}$. It means that the statement S in the function f in the program Π under the whole context $(\mathbf{\Gamma}, \mathbf{A})$ jumps to a label L_i with a whole context $(\mathbf{\Gamma}'_i, \mathbf{A}'_i)$ or safely returns from the current function call. The following are the rules for the judgment (we omit here Π and f, except in Σ_f).

$$\Sigma_f\colon \text{ the function signature of } f$$

$$\frac{(\mathbf{\Gamma}, \mathbf{A}) \vdash I\colon (\mathbf{\Gamma}', \mathbf{A}')}{(\mathbf{\Gamma}, \mathbf{A}) \vdash I;\ \mathsf{goto}\,L\colon \{(L, (\mathbf{\Gamma}', \mathbf{A}'))\}} \qquad \frac{\Sigma_f = \langle\vec{\alpha} \mid \cdots\rangle\,(\cdots) \to T}{(\{x\colon T\}, (\{\vec{\alpha}\}, R)) \vdash \mathsf{return}\,x\colon \varnothing}$$

$$\frac{x\colon P\,(T_0 + T_1) \in \mathbf{\Gamma} \qquad \text{for each } i,\ (\mathbf{\Gamma}'_i, \mathbf{A}'_i) = (\mathbf{\Gamma} - \{x\colon P\,(T_0 + T_1)\} + \{y_i\colon P\,T_i\},\ \mathbf{A})}{(\mathbf{\Gamma}, \mathbf{A}) \vdash \mathsf{match}\,*x\,\{\,\overrightarrow{\mathsf{inj}\,*y \to \mathsf{goto}\,L}\,\}\colon \{\overrightarrow{(L, (\mathbf{\Gamma}', \mathbf{A}'))}\}}$$

The rule for the instruction statement I; goto L simply uses the typing judgment for I. In the rule for the return statement return x, we require that there remain no extra variables and no local lifetimes. In the rule for the match statement, we check both branches. The input variable x of the type $P\,(T_0 + T_1)$ is consumed and in the inj_i branch we get a new variable y_i of the type $P\,T_i$.

We use a meta-variable LCtx to denote a finite map from labels to whole contexts. The typing judgment for statements is defined so that for any $(\mathbf{\Gamma}, \mathbf{A})$ and S, there exists at most one whole context assignment LCtx such that $(\mathbf{\Gamma}, \mathbf{A}) \vdash S\colon \mathit{LCtx}$ holds. This uniqueness can easily be proved, using the typing uniqueness on instructions.

Typing Judgment for Functions. The typing judgment for functions is $\Pi \vdash F\colon (\mathbf{\Gamma}_L, \mathbf{A}_L)_{L \in \mathsf{Lbl}_F}$, where Lbl_F denotes the set of labels in F. In short, the judgment assigns a whole context to each label in the function. The judgment is defined by the following rule.

$$\frac{\begin{array}{c}F \;=\; \mathsf{fn}\ f\ \langle\vec{\alpha}\,|\,\overrightarrow{\alpha_a \le \alpha_b}\rangle\ (\overrightarrow{x{:}\,T}) \to U\ \{\cdots\} \qquad \mathbf{\Gamma}_{\mathsf{entry}} = \{\overrightarrow{x{:}\,T}\} \qquad \mathbf{A}_{\mathsf{entry}} = \big(\{\vec{\alpha}\}, \big(\mathrm{Id}_{\{\vec{\alpha}\}} \cup \{\overrightarrow{(\alpha_a, \alpha_b)}\}\big)^{+}\big) \\ \text{for each } L_0{:}\, S \in \mathsf{LStmt}_F,\ \text{for some } LCtx \subseteq (\mathbf{\Gamma}_L, \mathbf{A}_L)_{L\in\mathsf{Lbl}_F},\ \Pi, f, (\mathbf{\Gamma}_L, \mathbf{A}_L) \vdash S{:}\, LCtx\end{array}}{\Pi \vdash F{:}\ (\mathbf{\Gamma}_L, \mathbf{A}_L)_{L\in\mathsf{Lbl}_F}}$$

LStmt_F: the set of labeled statements in F

Id_A: the identity relation on A $\qquad$ R^{+}: the transitive closure of R

The initial whole context at entry is constructed from the function signature (the second and third preconditions) and then the contexts for other labels are examined (the fourth precondition). The whole context for each label can be determined in the order of the distance from the label entry in the directed graph by the goto jumps. Therefore, a typing derivation on a function is unique. That is, for any Π and F, there exists at most one whole context assignment $(\mathbf{\Gamma}_L, \mathbf{A}_L)_{L\in\mathsf{Lbl}_F}$ such that $\Pi \vdash F{:}\ (\mathbf{\Gamma}_L, \mathbf{A}_L)_{L\in\mathsf{Lbl}_F}$ holds.

Typing Judgment for Programs. The typing judgment for programs is $\vdash \Pi{:}\,(\mathbf{\Gamma}_{f,L}, \mathbf{A}_{f,L})_{(f,L)\in\mathsf{FnLbl}_\Pi}$, where FnLbl_Π is the set of program points (f, L) in Π (f can be any function in the program Π and L can be any label in the function f). In short, the judgment assigns a whole context to each program point in the program. The judgment is defined simply by the following rule.

$$\frac{\text{for each } F \text{ in } \Pi,\ \Pi \vdash F{:}\ (\mathbf{\Gamma}_{\mathsf{name}\,F,\,L}, \mathbf{A}_{\mathsf{name}\,F,\,L})_{L\in\mathsf{Lbl}_F}}{\vdash \Pi{:}\ (\mathbf{\Gamma}_{f,L}, \mathbf{A}_{f,L})_{(f,L)\,\in\,\mathsf{FnLbl}_\Pi}}$$

$\mathsf{name}\,F$: the function name of F

For any program Π, there exists at most one whole context assignment $(\mathbf{\Gamma}_{f,L}, \mathbf{A}_{f,L})_{(f,L)\,\in\,\mathsf{FnLbl}_\Pi}$ such that $\vdash \Pi{:}\ (\mathbf{\Gamma}_{f,L}, \mathbf{A}_{f,L})_{(f,L)\,\in\,\mathsf{FnLbl}_\Pi}$ holds. We say that a program Π is *well typed* when it has a whole context assignment in this judgment.

Remark 2 (Soundness of the Type System). This type system is sound but to fully state the theorem we must also formally describe the safety condition on concrete configurations. The safety condition is introduced later in §4.2. The progress and preservation properties of the safety condition over well-typed programs are then proved (Proposition 3 and Corollary 5).

2.3 Operational Semantics

The following are the basic concepts of the operational semantics.

$$\text{(concrete configuration)}\ \ \mathbf{C} \ ::=\ [f_0, L_0]\,\mathbf{F};\ [f_1, L_1]\,x_1, \mathbf{F}_1;\ \cdots\ ;\ [f_n, L_n]\,x_n, \mathbf{F}_n \mid \mathbf{H}$$

$$\text{(heap memory)}\ \ \mathbf{H} \ ::=\ \text{(a finite map from addresses (integers) to memory-cell values (integers))}$$

$$\text{(concrete stack frame)}\ \ \mathbf{F} \ ::=\ \text{(a finite map from variables to addresses)}$$

The configuration consists of stack frames and a heap memory. Each stack frame is accompanied by $[f, L]$, which indicates the program point (the function and the label). Each non-top stack frame also has 'x,', which specifies the variable that will receive the return value of the function call of the stack frame just above. We also use a meta-variable **S** for a sequence of non-top stack frames $[f_1, L_1]\,x_1, \mathbf{F}_1;\ \cdots\ ;\ [f_n, L_n]\,x_n, \mathbf{F}_n$.

We also define the type size $|T|$, which represents how many memory cells the type T takes at the outermost level, as follows.

$$|P\,T| \;:=\; 1 \qquad |\mu X.T| \;:=\; |T| \qquad |T_0 + T_1| \;:=\; 1 + \max\{|T_0|, |T_1|\}$$
$$|T_0 \times T_1| \;:=\; |T_0| + |T_1| \qquad |\mathsf{int}| \;:=\; 1 \qquad |\mathsf{unit}| = 0$$

Although we do not define the type size for a type variable X, the definition above determines the type size for every *complete* type (defined in §2.1), in which occurrences of type variables are restricted.

The operational semantics is characterized by the one-step transition judgment $\Pi \vdash \mathbf{C} \to \mathbf{C}'$ and the termination judgment $\Pi \vdash \mathbf{C}\colon \mathsf{end}$. We assume that the program Π is well-typed. The type information we use here is quite limited; we use the type size to know how many memory cells are required and also check whether the pointer kind of a variable is own or R. The following are the complete rules for the two judgments. We omit Π here.

$S_{f,L}$: the statement at (f,L) $\quad$ $\mathsf{Ty}_{f,L}(x)$: the type of the variable x at the program point (f,L)

$$x \mapsto \vec{d}^N := \{(x,d_0),(x+1,d_1),\ldots,(x+N-1,d_{N-1})\}$$

$$\frac{S_{f,L} = \mathsf{let}\, y = \mathsf{mutbor}_\alpha\, x;\ \mathsf{goto}\, L' \quad \mathbf{F}(x) = a}{\vdash\ [f,L]\, \mathbf{F};\, \mathbf{S} \mid \mathbf{H} \ \to\ [f,L']\, \mathbf{F} + \{(y,a)\};\, \mathbf{S} \mid \mathbf{H}}$$

$$\frac{S_{f,L} = \mathsf{drop}\, x;\ \mathsf{goto}\, L' \quad \mathsf{Ty}_{f,L}(x) = \mathsf{own}\, T}{\vdash\ [f,L]\, \mathbf{F} + \{(x,a)\};\, \mathbf{S} \mid \mathbf{H} + (a \mapsto \vec{d}^{|T|}) \ \to\ [f,L']\, \mathbf{F};\, \mathbf{S} \mid \mathbf{H}} \quad (\textsc{Step-Drop-Own})$$

$$\frac{S_{f,L} = \mathsf{drop}\, x;\ \mathsf{goto}\, L' \quad \mathsf{Ty}_{f,L}(x) = R_\alpha\, T}{\vdash\ [f,L]\, \mathbf{F} + \{(x,\_)\};\, \mathbf{S} \mid \mathbf{H} \ \to\ [f,L']\, \mathbf{F};\, \mathbf{S} \mid \mathbf{H}} \quad (\textsc{Step-Drop-Ref})$$

$$\frac{S_{f,L} = I;\ \mathsf{goto}\, L' \quad I = x\ \mathsf{as}\ T,\ \mathsf{intro}\,\alpha,\ \mathsf{now}\,\alpha,\ \alpha \le \beta}{\vdash\ [f,L]\, \mathbf{F};\, \mathbf{S} \mid \mathbf{H} \ \to\ [f,L']\, \mathbf{F};\, \mathbf{S} \mid \mathbf{H}}$$

$$\frac{S_{f,L} = \mathsf{swap}(*x,*y);\ \mathsf{goto}\, L' \quad \mathsf{Ty}_{f,L}(x) = P\,T \quad \mathbf{F}(x) = a \quad \mathbf{F}(y) = b}{\vdash\ [f,L]\, \mathbf{F};\, \mathbf{S} \mid \mathbf{H} + (a \mapsto \vec{d}^{|T|}) + (b \mapsto \vec{e}^{|T|}) \ \to\ [f,L']\, \mathbf{F};\, \mathbf{S} \mid \mathbf{H} + (a \mapsto \vec{e}^{|T|}) + (b \mapsto \vec{d}^{|T|})}$$

$$\frac{S_{f,L} = \mathsf{let}\, *y = x;\ \mathsf{goto}\, L'}{\vdash\ [f,L]\, \mathbf{F} + \{(x,a')\};\, \mathbf{S} \mid \mathbf{H} \ \to\ [f,L']\, \mathbf{F} + \{(y,a)\};\, \mathbf{S} \mid \mathbf{H} + (a \mapsto a')}$$

$$\frac{S_{f,L} = \mathsf{let}\, y = *x;\ \mathsf{goto}\, L' \quad \mathsf{Ty}_{f,L}(x) = \mathsf{own}\, T}{\vdash\ [f,L]\, \mathbf{F} + \{(x,a)\};\, \mathbf{S} \mid \mathbf{H} + (a \mapsto a') \ \to\ [f,L']\, \mathbf{F} + \{(y,a')\};\, \mathbf{S} \mid \mathbf{H}} \quad (\textsc{Step-Deref-Own})$$

$$\frac{S_{f,L} = \mathsf{let}\, y = *x;\ \mathsf{goto}\, L' \quad \mathsf{Ty}_{f,L}(x) = R_\alpha\, T \quad (a \mapsto a') \subseteq \mathbf{H}}{\vdash\ [f,L]\, \mathbf{F} + \{(x,a)\};\, \mathbf{S} \mid \mathbf{H} \ \to\ [f,L']\, \mathbf{F} + \{(y,a')\};\, \mathbf{S} \mid \mathbf{H}} \quad (\textsc{Step-Deref-Ref})$$

$$\frac{S_{f,L} = \mathsf{let}\, *y = \mathsf{copy}\, *x;\ \mathsf{goto}\, L' \quad \mathsf{Ty}_{f,L}(x) = P\,T \quad (\mathbf{F}(x) \mapsto \vec{d}^{|T|}) \subseteq \mathbf{H}}{\vdash\ [f,L]\, \mathbf{F};\, \mathbf{S} \mid \mathbf{H} \ \to\ [f,L']\, \mathbf{F} + \{(y,a)\};\, \mathbf{S} \mid \mathbf{H} + (a \mapsto \vec{d}^{|T|})}$$

$$\frac{S_{f,L} = \mathsf{let}\, y = g\langle\cdots\rangle(\vec{x});\ \mathsf{goto}\, L' \quad \Sigma_g = \langle\cdots\rangle(\overrightarrow{x'\colon T}) \to U}{\vdash\ [f,L]\, \mathbf{F} + \{\overrightarrow{(x,a)}\};\, \mathbf{S} \mid \mathbf{H} \ \to\ [g,\mathsf{entry}]\, \{\overrightarrow{(x',a)}\};\ [f,L']\, y, \mathbf{F};\, \mathbf{S} \mid \mathbf{H}} \quad (\textsc{Step-Call})$$

$$\frac{S_{f,L} = \mathsf{return}\, x}{\vdash\ [f,L]\, \{(x,a)\};\, [g,L']\, x', \mathbf{F}';\mathbf{S} \mid \mathbf{H} \ \to\ [g,L']\, \mathbf{F}' + \{(x',a)\};\mathbf{S} \mid \mathbf{H}} \quad (\textsc{Step-Return})$$

$$\frac{S_{f,L} = \mathsf{return}\, x}{\vdash\ [f,L]\, \{(x,a)\} \mid \mathbf{H}\colon \mathsf{end}} \quad (\textsc{End-Return})$$

$$\frac{S_{f,L} = \mathsf{let}\ {*y} = \mathit{const};\ \mathsf{goto}\, L' \quad \vec{d} = \begin{cases} n & (\mathit{const} = n) \\ \epsilon & (\mathit{const} = ()) \end{cases}}{\vdash\ [f, L]\,\mathbf{F};\, \mathbf{S} \mid \mathbf{H}\ \rightarrow\ [f, L']\,\mathbf{F} + \{(y, a)\};\, \mathbf{S} \mid \mathbf{H} + (a \mapsto \vec{d})} \quad (\textsc{Step-Const})$$

$$\frac{S_{f,L} = \mathsf{let}\ {*y} = {*x}\ \mathit{op}\ {*x'};\ \mathsf{goto}\, L' \quad (\mathbf{F}(x) \mapsto m), (\mathbf{F}(x') \mapsto n) \subseteq \mathbf{H}}{\vdash\ [f, L]\,\mathbf{F};\, \mathbf{S} \mid \mathbf{H}\ \rightarrow\ [f, L']\,\mathbf{F} + \{(y, b)\};\, \mathbf{S} \mid \mathbf{H} + (b \mapsto m\, \langle \mathit{op} \rangle\, n)}$$

$\langle op \rangle$: op as a binary operation on integers, where true and false are encoded as 1 and 0

$$\frac{S_{f,L} = \mathsf{let}\ {*y} = \mathsf{rand}();\ \mathsf{goto}\, L'}{\vdash\ [f, L]\,\mathbf{F};\, \mathbf{S} \mid \mathbf{H}\ \rightarrow\ [f, L']\,\mathbf{F} + \{(y, a)\};\, \mathbf{S} \mid \mathbf{H} + (a \mapsto n)}$$

$$\frac{S_{f,L} = \mathsf{let}\ {*y} = \mathsf{inj}_i^{T_0+T_1}\ {*x};\ \mathsf{goto}\, L' \quad N = \max\{|T_{1-i}| - |T_i|, 0\}}{\vdash\ [f, L]\,\mathbf{F} + \{(x, a)\};\, \mathbf{S} \mid \mathbf{H} + (a \mapsto \vec{d}^{|T_i|})\ \rightarrow\ [f, L']\,\mathbf{F} + \{(y, a')\};\, \mathbf{S} \mid \mathbf{H} + (a' \mapsto\ i, \vec{d}^{|T_i|}, \vec{e}^{N})} \quad (\textsc{Step-Inj})$$

$$\frac{\begin{gathered} S_{f,L} = \mathsf{match}\ {*x}\, \{\, \overrightarrow{\mathsf{inj}\ {*y} \rightarrow \mathsf{goto}\, L'}\, \} \\ \mathsf{Ty}_{f,L}(x) = \mathsf{own}\, (T_0 + T_1) \quad i \in \{0, 1\} \quad N = \max\{|T_{1-i}| - |T_i|, 0\} \end{gathered}}{\vdash\ [f, L]\,\mathbf{F} + \{(x, a)\};\, \mathbf{S} \mid \mathbf{H} + (a \mapsto i) + (a + 1 + |T_i|\ \mapsto \vec{e}^{N})\ \rightarrow\ [f, L'_i]\,\mathbf{F} + \{(y_i, a + 1)\};\, \mathbf{S} \mid \mathbf{H}} \quad (\textsc{Step-Match-Own})$$

$$\frac{S_{f,L} = \mathsf{match}\ {*x}\, \{\, \overrightarrow{\mathsf{inj}\ {*y} \rightarrow \mathsf{goto}\, L'}\, \} \quad \mathsf{Ty}_{f,L}(x) = R_\alpha\, (T_0 + T_1) \quad \mathbf{H}(a) = i \in \{0, 1\}}{\vdash\ [f, L]\,\mathbf{F} + \{(x, a)\};\, \mathbf{S} \mid \mathbf{H}\ \rightarrow\ [f, L'_i]\,\mathbf{F} + \{(y_i, a + 1)\};\, \mathbf{S} \mid \mathbf{H}} \quad (\textsc{Step-Match-Ref})$$

$$\frac{S_{f,L} = \mathsf{let}\ {*y} = ({*x_0}, {*x_1});\ \mathsf{goto}\, L' \quad \text{for each } i,\ \mathsf{Ty}_{f,L}(x_i) = \mathsf{own}\, T_i}{\begin{gathered} \vdash\ [f, L]\,\mathbf{F} + \{(x_0, a_0), (x_1, a_1)\};\, \mathbf{S} \mid \mathbf{H} + \textstyle\sum_{i \in \{0,1\}} (a_i \mapsto \overrightarrow{d_i}^{|T_i|}) \\ \rightarrow\ [f, L']\,\mathbf{F} + \{(y, a')\};\, \mathbf{S} \mid \mathbf{H} + (a' \mapsto \overrightarrow{d_0}^{|T_0|}, \overrightarrow{d_1}^{|T_1|}) \end{gathered}}$$

$$\frac{S_{f,L} = \mathsf{let}\ ({*y}, {*y'}) = {*x};\ \mathsf{goto}\, L' \quad \mathsf{Ty}_{f,L}(x) = P\, (T \times T')}{\vdash\ [f, L]\,\mathbf{F} + \{(x, a)\};\, \mathbf{S} \mid \mathbf{H}\ \rightarrow\ [f, L']\,\mathbf{F} + \{(y, a), (y', a + |T|)\};\, \mathbf{S} \mid \mathbf{H}}$$

At each step, we remove invalidated variables from the concrete stack frame **F**, just as we did in the type system.

On a function call, we add a new stack frame to the head of the stack (Step-Call). The initial label is set to the entry point entry. When we return from a function, we remove the head stack frame from the stack and continue computation if we have remaining stack frames (Step-Return). If the current stack frame is the only stack frame in the stack, the computation ends by the rule End-Return (actually this is the only rule for the judgment $\Pi\ \vdash\ \mathbf{C}\colon \mathsf{end}$).

In general, instructions of the form let $*y = \cdots$ allocate memory cells for the newly created owning pointer y. For example, an instruction let $*y = n$ allocates a memory cell for the integer data n (Step-Const).

Some operations behave differently for depending on whether the input is an owning pointer or a reference. The instruction drop x deallocates the target object from the heap if x is an owning pointer (Step-Drop-Own) but does not perform deallocation if x is a reference (Step-Drop-Ref). The instruction let $y = *x$ performs deallocation of the target memory cell of x if x is an owning pointer (Step-Deref-Own) but does not otherwise (Step-Deref-Ref). Similarly, the match statement

$\mathsf{match}\ {*x}\ \{\ \overrightarrow{\mathsf{inj}\ {*y} \to \mathsf{goto}\, L'}\ \}$ deallocates the memory cells for the index and the padding if x is an ownership pointer (Step-Match-Own) but does not otherwise (Step-Match-Ref).

When we create a variant object by the instruction $\mathsf{let}\ {*y} = \mathsf{inj}_i^{T_0+T_1}\ {*x}$ (Type-Inst-Inj), we allocate a padding by zeroes if T_i has a smaller size than T_{1-i} does, which makes the size of the variant object $1 + \max\{T_0, T_1\}$ in total.

Example 2 (Execution in the Operational Semantics). The following is an execution sequence in the operational semantics for the program presented in Example 1. The inputs to *oa* and *ob* are set to 5 and 3. The symbols $\spadesuit$, $\heartsuit$, $\diamondsuit$, $\clubsuit$ represent some mutually distinct addresses.

$$
\begin{aligned}
&[\mathsf{inc\text{-}max}, \mathsf{entry}]\ \{(oa, \spadesuit), (ob, \heartsuit)\} \mid \{(\spadesuit, 5), (\heartsuit, 3)\} \\
&\to\ [\mathsf{inc\text{-}max}, \mathsf{L1}]\ \{(oa, \spadesuit), (ob, \heartsuit)\} \mid \{(\spadesuit, 5), (\heartsuit, 3)\} \\
&\to^{+}\ [\mathsf{inc\text{-}max}, \mathsf{L3}]\ \{(ma, \spadesuit), (mb, \heartsuit), (oa, \spadesuit), (ob, \heartsuit)\} \mid \{(\spadesuit, 5), (\heartsuit, 3)\} \\
&\to\ [\mathsf{take\text{-}max}, \mathsf{entry}]\ \{(ma, \spadesuit), (mb, \heartsuit)\};\ [\mathsf{inc\text{-}max}, \mathsf{L4}]\ mc, \{(oa, \spadesuit), (ob, \heartsuit)\} \mid \{(\spadesuit, 5), (\heartsuit, 3)\} \\
&\to\ [\mathsf{take\text{-}max}, \mathsf{L1}]\ \{(ord, \diamondsuit), (ma, \spadesuit), (mb, \heartsuit)\};\ [\mathsf{inc\text{-}max}, \mathsf{L4}]\ mc, \{(oa, \spadesuit), (ob, \heartsuit)\} \\
&\qquad \mid \{(\spadesuit, 5), (\heartsuit, 3), (\diamondsuit, 1)\} \\
&\to^{+}\ [\mathsf{take\text{-}max}, \mathsf{L3}]\ \{(ma, \spadesuit), (mb, \heartsuit)\};\ [\mathsf{inc\text{-}max}, \mathsf{L4}]\ mc, \{(oa, \spadesuit), (ob, \heartsuit)\} \mid \{(\spadesuit, 5), (\heartsuit, 3)\} \\
&\to\ [\mathsf{take\text{-}max}, \mathsf{L4}]\ \{(ma, \spadesuit)\};\ [\mathsf{inc\text{-}max}, \mathsf{L4}]\ mc, \{(oa, \spadesuit), (ob, \heartsuit)\} \mid \{(\spadesuit, 5), (\heartsuit, 3)\} \\
&\to\ [\mathsf{inc\text{-}max}, \mathsf{L4}]\ \{(mc, \spadesuit), (oa, \spadesuit), (ob, \heartsuit)\} \mid \{(\spadesuit, 5), (\heartsuit, 3)\} \\
&\to\ [\mathsf{inc\text{-}max}, \mathsf{L5}]\ \{(o1, \diamondsuit), (mc, \spadesuit), (oa, \spadesuit), (ob, \heartsuit)\} \mid \{(\spadesuit, 5), (\heartsuit, 3), (\diamondsuit, 1)\} \\
&\to^{+}\ [\mathsf{inc\text{-}max}, \mathsf{L7}]\ \{(oc', \clubsuit), (mc, \spadesuit), (oa, \spadesuit), (ob, \heartsuit)\} \mid \{(\spadesuit, 5), (\heartsuit, 3), (\clubsuit, 6)\} \\
&\to^{+}\ [\mathsf{inc\text{-}max}, \mathsf{L9}]\ \{(mc, \spadesuit), (oa, \spadesuit), (ob, \heartsuit)\} \mid \{(\spadesuit, 6), (\heartsuit, 3)\} \\
&\to\ [\mathsf{inc\text{-}max}, \mathsf{L10}]\ \{(oa, \spadesuit), (ob, \heartsuit)\} \mid \{(\spadesuit, 6), (\heartsuit, 3)\} \\
&\to^{+}\ [\mathsf{inc\text{-}max}, \mathsf{L14}]\ \{(ores, \diamondsuit)\} \mid \{(\diamondsuit, 1)\}
\end{aligned}
$$

In the stack frames each variable just has the address data. Integer objects are all stored in the heap memory.

3 OUR REDUCTION FROM RUST PROGRAMS TO CHCS

Now we formalize our reduction from Rust programs to CHCs, discussed in §1 as a reduction from a program in our calculus COR to a CHC system, which is guaranteed to precisely characterize the input-output relation of each function in the program. We first define the first-order multi-sorted logic for CHCs in §3.1. We then formally describe our reduction in §3.2. We formalize its soundness and completeness and outline the proof of that in §3.3 (we present the complete proof in §4). Also, we examine effectiveness of our approach with advanced examples in §3.4 and discuss various topics about our idea in §3.5.

3.1 Multi-sorted Logic for CHCs

To begin with, we introduce a first-order multi-sorted logic for CHCs.

Syntax. The following is the syntax of the logic.

$$
\text{(CHC)}\ \ \Phi\ ::=\ \forall\, \overrightarrow{x\colon \sigma}.\ \check{\varphi}\ \Longleftarrow\ \bigwedge \vec{\psi} \qquad \top\ :=\ \bigwedge \epsilon
$$

$$
\text{(pattern formula)}\ \ \check{\varphi}\ ::=\ f(\vec{p}) \qquad \text{(formula)}\ \ \varphi, \psi\ ::=\ f(\vec{t})
$$

$$
\text{(term)}\ \ t\ ::=\ x \mid \langle t \rangle\ \text{(box)} \mid \langle t_*, t_\circ \rangle\ \text{(mut)} \mid \mathsf{inj}_i\, t \mid (t_0, t_1) \mid \mathit{const} \mid t\ \mathit{op}\ t'
$$

$$\text{(pattern)}\ \ p,q\ ::=\ x\ \mid\ \langle p\rangle\ \mid\ \langle p_*,p_\circ\rangle\ \mid\ \mathsf{inj}_i\,p\ \mid\ (p_0,p_1)\ \mid\ \mathit{const}$$

$$\text{(value)}\ \ v,w\ ::=\ \langle v\rangle\ \mid\ \langle v_*,v_\circ\rangle\ \mid\ \mathsf{inj}_i\,v\ \mid\ (v_0,v_1)\ \mid\ \mathit{const}$$

$$\text{(sort)}\ \ \sigma,\tau\ ::=\ C\,\sigma\ \mid\ \sigma_0+\sigma_1\ \mid\ \sigma_0\times\sigma_1\ \mid\ X\ \mid\ \mu X.\sigma\ \mid\ \mathsf{int}\ \mid\ \mathsf{unit}$$

$$\text{(container kind)}\ \ C\ ::=\ \mathsf{box}\ \ (\text{box; }\langle v\rangle)\ \mid\ \mathsf{mut}\ \ (\text{mut; }\langle v_*,v_\circ\rangle)$$

$$\mathit{const}\ ::=\ \text{same as COR}\qquad \mathit{op}\ ::=\ \text{same as COR}$$

$$X\ \ \text{(sort variable)}\qquad x,y\ \ \text{(logic variable)}\qquad f\ \ \text{(predicate variable)}$$

$$\mathsf{bool}\ :=\ \mathsf{unit}+\mathsf{unit}\qquad \mathsf{true}\ :=\ \mathsf{inj}_1\,()\qquad \mathsf{false}\ :=\ \mathsf{inj}_0\,()$$

Also, a CHC system is defined as a pair $(\mathbf{\Phi},\Xi)$ of a finite set of CHCs $\mathbf{\Phi}=\{\vec{\Phi}\}$ and a finite map Ξ from a predicate variable to a tuple of sorts (denoted by Ξ), specifying the sorts of the arguments for the predicate variable. Unlike the informal description in §1, we explicitly specify the sort information Ξ. For simplicity, we often omit the universal quantifier $\forall\vec{x}.$ of CHCs.

CHCs in this logic have a fairly restricted form, in comparison to informal CHCs used in §1. Every formula ϕ should be of form $f(\vec{t})$ and we do not have a category for constraints like $a<b$. Also, the head of each CHC should be of form $f(\vec{p})$, where p_i is a *pattern*, consisting only of variables and constructors, not having operators. Even in this restriction, we can express various predicates using the idea of pattern matching. For example, the equality relation Eq on a sort σ can be introduced in a CHC system by adding the following rule on Eq: $\forall\,x{:}\,\sigma.\ Eq(x,x)\ \Longleftarrow\ \top$ (precisely speaking, Eq is the equality relation in the *least* solution of the CHC system). This restriction helps to simplify our proof of the soundness and completeness later in §4.

In this logic, we have two special data types, a box container, whose value is $\langle t\rangle$ and whose sort is $\mathsf{box}\,\sigma$, and a mut container, whose value is $\langle t_*,t_\circ\rangle$ and whose sort is $\mathsf{mut}\,\sigma$. In our reduction, owning pointers and immutable references are modeled as a box container and mutable references are modeled as a mut container.

Sort System. The sort-giving judgment $\Delta\ \vdash\ t{:}\,\sigma$ (the term t has the sort σ under Δ) is defined as follows. Here, Δ is a finite map from variables to sorts.

$$\frac{\Delta(x)=\sigma}{\Delta\ \vdash\ x{:}\,\sigma}\qquad \frac{\Delta\ \vdash\ t{:}\,\sigma}{\Delta\ \vdash\ \langle t\rangle{:}\,\mathsf{box}\,\sigma}\qquad \frac{\Delta\ \vdash\ t_*,t_\circ{:}\,\sigma}{\Delta\ \vdash\ \langle t_*,t_\circ\rangle{:}\,\mathsf{mut}\,\sigma}\qquad \frac{\Delta\ \vdash\ t{:}\,\sigma_i}{\Delta\ \vdash\ \mathsf{inj}_i\,t{:}\,\sigma_0+\sigma_1}$$

$$\frac{\Delta\ \vdash\ t_0{:}\,\sigma_0,\ t_1{:}\,\sigma_1}{\Delta\ \vdash\ (t_0,t_1){:}\,\sigma_0\times\sigma_1}\qquad \Delta\ \vdash\ n{:}\,\mathsf{int}\qquad \Delta\ \vdash\ (){:}\,\mathsf{unit}\qquad \frac{\Delta\ \vdash\ t,t'{:}\,\mathsf{int}}{\Delta\ \vdash\ t\ \mathit{op}_\sigma\ t'{:}\,\sigma}\qquad \frac{\Delta\ \vdash\ t{:}\,\sigma\quad \sigma\sim\tau}{\Delta\ \vdash\ t{:}\,\tau}$$

$$\sigma\sim\tau\text{: the congruence on sorts induced by }\mu X.\sigma\sim\sigma[\mu X.\sigma/X]$$

We abbreviate $\varnothing\ \vdash\ t{:}\,\sigma$ as $\vdash\ t{:}\,\sigma$.

We introduce the well-sortedness judgments for a CHC system $\vdash\ (\mathbf{\Phi},\Xi){:}\,\mathsf{well\ sorted}$, for a CHC $\Xi\ \vdash\ \mathbf{\Phi}{:}\,\mathsf{well\ sorted}$ and for a formula $\Delta,\Xi\ \vdash\ \Phi{:}\,\mathsf{well\ sorted}$ and give them the following rules.

$$\frac{\text{for each }\Phi\in\mathbf{\Phi},\ \ \Xi\ \vdash\ \Phi{:}\,\mathsf{well\ sorted}}{\vdash\ (\mathbf{\Phi},\Xi){:}\,\mathsf{well\ sorted}}$$

$$\frac{\Delta=\{\overrightarrow{(x,\sigma)}\}\qquad \Delta,\Xi\ \vdash\ \check{\varphi}{:}\,\mathsf{well\ sorted}\qquad \text{for each }j,\ \Delta,\Xi\ \vdash\ \psi_j{:}\,\mathsf{well\ sorted}}{\Xi\ \vdash\ \forall\overrightarrow{x{:}\,\sigma}.\ \check{\varphi}\ \Longleftarrow\ \bigwedge\vec{\psi}{:}\ \ \mathsf{well\ sorted}}$$

$$\frac{\Xi(f)=(\vec{\sigma})\qquad \text{for each }i,\ \Delta\ \vdash\ t_i{:}\,\sigma_i}{\Delta,\Xi\ \vdash\ f(\vec{t}){:}\,\mathsf{well\ sorted}}$$

Semantics. An evaluation $\mathbf{I}$ is a finite map from variables to values. A predicate structure $\mathbf{M}$ is a finite map from predicate variables to predicates on values of some fixed sorts.

We define the sort-giving judgment on an evaluation $\vdash \mathbf{I} \colon \Delta$ as follows.

$$\frac{\text{for each } i,\ \vdash v_i \colon \sigma_i}{\vdash \{\overrightarrow{(x, v)}\} \colon \{\overrightarrow{(x, \sigma)}\}}$$

The interpretation of a term t into a value over an evaluation $\mathbf{I}$, denoted $[\![t]\!]_{\mathbf{I}}$, is defined as follows.

$$[\![x]\!]_{\mathbf{I}} := \mathbf{I}(x) \qquad [\![\langle t \rangle]\!]_{\mathbf{I}} := \langle [\![t]\!]_{\mathbf{I}} \rangle \qquad [\![\langle t_*, t_\circ \rangle]\!]_{\mathbf{I}} := \langle [\![t_*]\!]_{\mathbf{I}}, [\![t_\circ]\!]_{\mathbf{I}} \rangle \qquad [\![\mathsf{inj}_i\, t]\!]_{\mathbf{I}} := \mathsf{inj}_i [\![t]\!]_{\mathbf{I}}$$

$$[\![(t_0, t_1)]\!]_{\mathbf{I}} := ([\![t_0]\!]_{\mathbf{I}}, [\![t_1]\!]_{\mathbf{I}}) \qquad [\![const]\!]_{\mathbf{I}} := const \qquad [\![t\ op\ t']\!]_{\mathbf{I}} := [\![t]\!]_{\mathbf{I}}\, [\![op]\!]\, [\![t']\!]_{\mathbf{I}}$$

$$[\![op]\!] \colon \text{the binary operation on integers corresponding to } op$$

Although the definition is partial on op, the interpretation is defined for every well-sorted term (i.e., $[\![t]\!]_{\mathbf{I}}$ is defined if $\Delta \vdash t \colon \sigma$ holds for some Δ satisfying $\vdash \mathbf{I} \colon \Delta$ and some σ), which follows from straightforward induction.

The validity of a CHC $\mathbf{M} \models \Phi$ and the validity of a formula $\mathbf{M}, \mathbf{I} \models \varphi$ are defined as follows.

$$\frac{\text{for each } \mathbf{I} \text{ s.t. } \vdash \mathbf{I} \colon \{\overrightarrow{x \colon \sigma}\},\ \mathbf{M}, \mathbf{I} \models \check{\varphi} \text{ or } \mathbf{M}, \mathbf{I} \not\models \psi_i \text{ for some } i}{\mathbf{M} \models \forall\, \overrightarrow{x \colon \sigma}.\ \check{\varphi} \iff \bigwedge \vec{\psi}} \qquad \frac{\mathbf{M}(f)(\overrightarrow{[\![t]\!]_{\mathbf{I}}}) \text{ is true}}{\mathbf{M}, \mathbf{I} \models f(\vec{t})}$$

Finally, the validity of a CHC system $\mathbf{M} \models (\mathbf{\Phi}, \Xi)$ is defined as follows.

$$\frac{\begin{array}{c}\text{for each } (f, (\vec{\sigma})) \in \Xi,\ \mathbf{M}(f) \text{ is a predicate on values of sort } \vec{\sigma} \\ \operatorname{dom} \mathbf{M} = \operatorname{dom} \Xi \qquad \text{for each } \Phi \in \mathbf{\Phi},\ \mathbf{M} \models \Phi\end{array}}{\mathbf{M} \models (\mathbf{\Phi}, \Xi)}$$

We say that $\mathbf{M}$ is a *solution* to $(\mathbf{\Phi}, \Xi)$ if $\mathbf{M} \models (\mathbf{\Phi}, \Xi)$ holds. Every well-sorted CHC system $(\mathbf{\Phi}, \Xi)$ has a *least solution* with respect to the point-wise ordering, which can be proved based on the standard discussion [74]. We write the least solution of $(\mathbf{\Phi}, \Xi)$ as $\mathbf{M}^{\mathsf{least}}_{(\mathbf{\Phi}, \Xi)}$.

3.2 Our Reduction from Programs to CHCs

Now we formalize our reduction of Rust programs to CHC systems. We define the *CHC representation* $(\!|\Pi|\!)$ of a well-typed COR program Π, which is a CHC system that represents the input-output relations of the functions in Π.

We assign a predicate variable f_L to each program point (f, L) (e.g., each label L in each function f). Roughly speaking, the predicate f_L represents the input-output relation of the continuation from the program point L in the function f, where the inputs are the values of the local variables at (f, L) and the output is the return value of f.[9] For each f_L, we add one or two CHCs to the resulting CHC system, which represent the operation of the statement at (f, L). As explained in §1.2, in the resulting CHCs, we represent a mutable reference as $\langle v_*, v_\circ \rangle$, a pair of the current target value v_* and the final target value $v_\circ$, and do not explicitly model addresses and memory states.

Roughly speaking, our CHC representation is designed so that its least solution $\mathbf{M}^{\mathsf{least}}_{(\!|\Pi|\!)}$ satisfies the following property: for any values $\vec{v}, w$, the validity $\mathbf{M}^{\mathsf{least}}_{(\!|\Pi|\!)} \models f_{\mathsf{entry}}(\vec{v}, w)$ holds if and only if a function call $f(\vec{v})$ can return w in the program. Actually, since such values should be extracted from the heap memory in the operational semantics, the actual definition is a bit more involved. The formal description and the proof of this expected property are presented later in §3.3.

[9] When a local variable contains a mutable reference the meaning of the 'value' can be subtle because of the final target value in our model. Later in §4.2 and §4.3, we model each of the future target values as a *syntactic variable* in logic.

Preliminaries. We introduce some preliminary definitions and notions.

The sort corresponding to the type T, $(\!|T|\!)$, is defined as follows. Note that the information on lifetimes is all stripped off.

$$(\!|\check{P}\,T|\!) := \mathsf{box}\,(\!|T|\!) \qquad (\!|\mathsf{mut}_\alpha\,T|\!) := \mathsf{mut}\,(\!|T|\!)$$

$$(\!|T + T'|\!) := (\!|T|\!) + (\!|T'|\!) \qquad (\!|T \times T'|\!) := (\!|T|\!) \times (\!|T'|\!)$$

$$(\!|X|\!) := X \qquad (\!|\mu X.T|\!) = \mu X.(\!|T|\!) \qquad (\!|\mathsf{int}|\!) := \mathsf{int} \qquad (\!|\mathsf{unit}|\!) := \mathsf{unit}$$

We assume some *fixed linear order* on data variables and enumerate the elements of a data context, a stack frame, etc. in this order. Also, we fix a well-typed program Π as an implicit parameter.

The predicate signature of f_L, denoted by $\Xi_{f,L}$, is defined as $(\overrightarrow{(\!|T|\!)}, (\!|U|\!))$, where $\{\overrightarrow{x{:}^{\mathrm{a}}\,T}\}$ is the data context at (f, L) and U is the return type of f.

Our Reduction. Now we fully define our reduction.

In our reduction, for each program point (f, L), we generate a CHC or a pair of CHCs with the head of the form $f_L(\vec{p}, \mathrm{res})$, which models the computation performed by the statement. Here, res is a special variable that represents the result of the function (we put this variable at the last in the fixed linear order). We add just one CHC for a statement of the form I; $\mathsf{goto}\,L$ or $\mathsf{return}\,x$ and we add two CHCs for a match statement $\mathsf{match}\,{*x}\,\{\cdots\}$ (recall that we do not allow here disjunction in the body of each CHC, unlike informal description in §1).

For example, let us consider a labeled statement $\mathsf{L1}$: $\mathsf{let}\,{*y} = 3$; $\mathsf{goto}\,\mathsf{L2}$ in a function f, which allocates an integer memory cell. The CHC we generate for the statement is as follows, letting $\vec{x}$ be the local variables in $\mathsf{L1}$ (we omit the universal quantifier).

$$f_{\mathsf{L1}}(\vec{x}, \mathrm{res}) \iff f_{\mathsf{L2}}(\vec{x}, \langle 3\rangle, \mathrm{res})$$

Here, $\langle 3\rangle$ represents the value of y, i.e., the newly created owning pointer that has the integer data 3. This CHC can be read as a rewriting rule from left to right: the statement creates a new owning pointer $\langle 3\rangle$ and passes it with the carryover variables $\vec{x}$ to the next statement at $\mathsf{L2}$.

For another example, let us consider a labeled statement $\mathsf{L3}$: $\mathsf{let}\,{*z} = \mathsf{mutbor}_\alpha\,y$; $\mathsf{goto}\,\mathsf{L4}$ in f, which performs a mutable borrow. Assume that the data context at $\mathsf{L3}$ is $\{\overrightarrow{x{:}^{\mathrm{a}}\,T},\ y{:}\,\mathsf{own}\,T'\}$, which sets the data context at $\mathsf{L4}$ to $\{\overrightarrow{x{:}^{\mathrm{a}}\,T},\ y{:}^{\dagger\alpha}\,\mathsf{own}\,T',\ z{:}\,\mathsf{mut}_\alpha\,T\}$. The CHC we generate for this statement is as follows.

$$f_{\mathsf{L3}}(\vec{x}, \langle y_*\rangle, \mathrm{res}) \iff f_{\mathsf{L4}}(\vec{x}, \langle y_\circ\rangle, \langle y_*, y_\circ\rangle, \mathrm{res})$$

For convenience, we introduce the notation $\check{\varphi}_{f,L}$ for the pattern formula $f_L(\vec{x}, \mathrm{res})$, where $\vec{x}$ are the local variables at (f, L). (Note that we reuse data variables $\vec{x}$ of COR as logic variables.) For example, the CHC of the previous example can be written as follows.

$$\check{\varphi}_{f,\mathsf{L3}}[\langle y_*\rangle/y] \iff \check{\varphi}_{f,\mathsf{L4}}[\langle y_*, y_\circ\rangle/z,\ \langle y_\circ\rangle/y],$$

Now we define the *CHC representation* $(\!|\Pi|\!)$ of a well-typed program Π as follows.

$$(\!|\Pi|\!) := \big(\{\,\Phi \mid \Phi \text{ is in } (\!|L{:}\,S|\!)_{\Pi,\mathrm{name}\,F},\ F \text{ is in } \Pi,\ L{:}\,S \in \mathsf{LStmt}_F\,\},\ (\Xi_{\Pi,f,L})_{f_L \text{ s.t. } (f,L)\,\in\,\mathsf{FnLbl}_\Pi}\big)$$

Here, $(\!|L{:}\,S|\!)_{\Pi,f}$ is one CHC or a pair of CHCs we generate for the labeled statement $L{:}\,S$ in f in Π, which is defined by the following rules. For simplicity, we omit here universal quantifiers and Π. For some statements, depending on the pointer kinds of the input variables, we generate fairly different CHCs.

$$(\!|L\colon\ \mathsf{let}\, y = \mathsf{mutbor}_\alpha\, x;\ \mathsf{goto}\, L'|\!)_f \\ := \begin{cases} \check\varphi_{f,L}[\langle x_*\rangle/x] \Longleftarrow \check\varphi_{f,L'}[\langle x_*, x_\circ\rangle/y, \langle x_\circ\rangle/x] & (\mathsf{Ty}_{f,L}(x) = \mathsf{own}\, T) \\ \check\varphi_{f,L}[\langle x_*, x'_\circ\rangle/x] \Longleftarrow \check\varphi_{f,L'}[\langle x_*, x_\circ\rangle/y, \langle x_\circ, x'_\circ\rangle/x] & (\mathsf{Ty}_{f,L}(x) = \mathsf{mut}_\alpha\, T) \end{cases} \tag{Chc-Stmt-Mutbor}$$

$$(\!|L\colon\ \mathsf{drop}\, x;\ \mathsf{goto}\, L'|\!)_f \\ := \begin{cases} \check\varphi_{f,L} \Longleftarrow \check\varphi_{f,L'} & (\mathsf{Ty}_{f,L}(x) = \check{P}\, T) \\ \check\varphi_{f,L}[\langle x_*, x_*\rangle/x] \Longleftarrow \check\varphi_{f,L'} & (\mathsf{Ty}_{f,L}(x) = \mathsf{mut}_\alpha\, T) \end{cases} \tag{Chc-Stmt-Drop}$$

$$(\!|L\colon\ \mathsf{immut}\, x;\ \mathsf{goto}\, L'|\!)_f \\ := \check\varphi_{f,L}[\langle x_*, x_*\rangle/x] \Longleftarrow \check\varphi_{f,L'}[\langle x_*\rangle/x] \quad (\mathsf{Ty}_{f,L}(x) = \mathsf{mut}_\alpha\, T) \tag{Chc-Stmt-Immut}$$

$$(\!|L\colon\ \mathsf{swap}(*x, *y);\ \mathsf{goto}\, L'|\!)_f \\ := \begin{cases} \check\varphi_{f,L}[\langle x_*, x_\circ\rangle/x, \langle y_*\rangle/y] \Longleftarrow \check\varphi_{f,L'}[\langle y_*, x_\circ\rangle/x, \langle x_*\rangle/y] & (\mathsf{Ty}_{f,L}(y) = \mathsf{own}\, T) \\ \check\varphi_{f,L}[\langle x_*, x_\circ\rangle/x, \langle y_*, y_\circ\rangle/y] \Longleftarrow \check\varphi_{f,L'}[\langle y_*, x_\circ\rangle/x, \langle x_*, y_\circ\rangle/y] & (\mathsf{Ty}_{f,L}(y) = \mathsf{mut}_\alpha\, T) \end{cases}$$

$$(\!|L\colon\ \mathsf{let}\, *y = x;\ \mathsf{goto}\, L'|\!)_f \ := \ \check\varphi_{f,L} \Longleftarrow \check\varphi_{f,L'}[\langle x\rangle/y]$$

$$(\!|L\colon\ \mathsf{let}\, y = *x;\ \mathsf{goto}\, L'|\!)_f \\ := \begin{cases} \check\varphi_{f,L}[\langle x_*\rangle/x] \Longleftarrow \check\varphi_{f,L'}[x_*/y] & (\mathsf{Ty}_{f,L}(x) = \mathsf{own}\, P\, T) \\ \check\varphi_{f,L}[\langle\langle x_{**}\rangle\rangle/x] \Longleftarrow \check\varphi_{f,L'}[\langle x_{**}\rangle/y] & (\mathsf{Ty}_{f,L}(x) = \mathsf{immut}_\alpha\, \check{P}\, T) \\ \check\varphi_{f,L}[\langle\langle x_{**}, x_{*\circ}\rangle\rangle/x] \Longleftarrow \check\varphi_{f,L'}[\langle x_{**}\rangle/y] & (\mathsf{Ty}_{f,L}(x) = \mathsf{immut}_\alpha\, \mathsf{mut}_\beta\, T) \\ \check\varphi_{f,L}[\langle\langle x_{**}\rangle, \langle x_{\circ*}\rangle\rangle/x] \Longleftarrow \check\varphi_{f,L'}[\langle x_{**}, x_{\circ*}\rangle/y] & (\mathsf{Ty}_{f,L}(x) = \mathsf{mut}_\alpha\, \mathsf{own}\, T) \\ \check\varphi_{f,L}[\langle x_*, x_*\rangle/x] \Longleftarrow \check\varphi_{f,L'}[x_*/y] & (\mathsf{Ty}_{f,L}(x) = \mathsf{mut}_\alpha\, \mathsf{immut}_\beta\, T) \\ \check\varphi_{f,L}[\langle\langle x_{**}, x_{*\circ}\rangle, \langle x_{\circ*}, x_{*\circ}\rangle\rangle/x] \Longleftarrow \check\varphi_{f,L'}[\langle x_{**}, x_{\circ*}\rangle/y] & (\mathsf{Ty}_{f,L}(x) = \mathsf{mut}_\alpha\, \mathsf{mut}_\beta\, T) \end{cases} \tag{Chc-Stmt-Deref}$$

$$(\!|L\colon\ \mathsf{let}\, *y = \mathsf{copy}\, *x;\ \mathsf{goto}\, L'|\!)_f \ := \ \check\varphi_{f,L} \Longleftarrow \check\varphi_{f,L'}[x/y, x/x]$$

$$(\!|L\colon\ I;\ \mathsf{goto}\, L'|\!)_f \ := \ \check\varphi_{f,L} \Longleftarrow \check\varphi_{f,L'} \quad (I = x\ \mathsf{as}\ T,\ \mathsf{intro}\,\alpha,\ \mathsf{now}\,\alpha,\ \alpha \le \beta)$$

$$(\!|L\colon\ \mathsf{let}\, y = g\langle\cdots\rangle(\vec{x});\ \mathsf{goto}\, L'|\!)_f \ := \ \check\varphi_{f,L} \Longleftarrow g_{\mathsf{entry}}(\vec{x}, y) \land \check\varphi_{f,L'}[y/y] \tag{Chc-Stmt-Call}$$

$$(\!|L\colon\ \mathsf{return}\, x|\!)_f \ := \ \check\varphi_{f,L}[x/\mathsf{res}] \Longleftarrow \top \tag{Chc-Stmt-Return}$$

$$(\!|L\colon\ \mathsf{let}\, *y = \mathit{const};\ \mathsf{goto}\, L'|\!)_f \ := \ \check\varphi_{f,L} \Longleftarrow \check\varphi_{f,L'}[\langle \mathit{const}\rangle/y]$$

$$(\!|L\colon\ \mathsf{let}\, *y = *x_0\ \mathit{op}\, *x_1;\ \mathsf{goto}\, L'|\!)_f \ := \ \check\varphi_{f,L}[p_0/x_0, p_1/x_1] \Longleftarrow \check\varphi_{f,L'}[\langle x_{0*}\ \mathit{op}\ x_{1*}\rangle/y, p_0/x_0, p_1/x_1]$$

$$\text{where}\quad p_i := \begin{cases} \langle x_{i*}\rangle & (\mathsf{Ty}_{f,L}(x_i) = \check{P}\,\mathsf{int}) \\ \langle x_{i*}, x_{i\circ}\rangle & (\mathsf{Ty}_{f,L}(x_i) = \mathsf{mut}_\alpha\,\mathsf{int}) \end{cases}$$

$$(\!|L\colon\ \mathsf{let}\, *y = \mathsf{rand}();\ \mathsf{goto}\, L'|\!)_f \ := \ \check\varphi_{f,L} \Longleftarrow \check\varphi_{f,L'}[\langle y_*\rangle/y]$$

$$(\!|L\colon\ \mathsf{let}\, *y = \mathsf{inj}_i^{T_0+T_1}\, *x;\ \mathsf{goto}\, L'|\!)_f \ := \ \check\varphi_{f,L}[\langle x_*\rangle/x] \Longleftarrow \check\varphi_{f,L'}[\langle \mathsf{inj}_i\, x_*\rangle/y]$$

$$(\!|L\colon \mathsf{match}\ {*x}\ \{\ \overrightarrow{\mathsf{inj}\ {*y} \to \mathsf{goto}\ L'}\ \}|\!)_f$$
$$:= \begin{cases} (\check\varphi_{f,L}[\langle \mathsf{inj}_i\ x_{*!}\rangle/x] \Longleftarrow \check\varphi_{f,L'_i}[\langle x_{*!}\rangle/y_i])_{i=0,1} & (\mathsf{Ty}_{f,L}(x) = \check P\,(T_0+T_1)) \\ (\check\varphi_{f,L}[\langle \mathsf{inj}_i\ x_{*!}, \mathsf{inj}_i\ x_{\circ!}\rangle/x] \Longleftarrow \check\varphi_{f,L'_i}[\langle x_{*!}, x_{\circ!}\rangle/y_i])_{i=0,1} & (\mathsf{Ty}_{f,L}(x) = \mathsf{mut}_\alpha(T_0+T_1)) \end{cases}$$
(Chc-Stmt-Match)

$$(\!|L\colon \mathsf{let}\ {*y} = ({*x}, {*x'});\ \mathsf{goto}\ L'|\!)_f \ := \ \check\varphi_{f,L}[\langle x_*\rangle/x, \langle x'_*\rangle/x'] \Longleftarrow \check\varphi_{f,L'}[\langle (x_*, x'_*)\rangle/y]$$

$$(\!|L\colon \mathsf{let}\ ({*y}, {*y'}) = {*x};\ \mathsf{goto}\ L'|\!)_f$$
$$:= \begin{cases} \check\varphi_{f,L}[\langle (x_*, x'_*)\rangle/x] \Longleftarrow \check\varphi_{f,L'}[\langle x_*\rangle/y, \langle x'_*\rangle/y'] & (\mathsf{Ty}_{f,L}(x) = \check P\,(T\times T')) \\ \check\varphi_{f,L}[\langle (x_*, x'_*), (x_\circ, x'_\circ)\rangle/x] & \\ \quad \Longleftarrow \check\varphi_{f,L'}[\langle x_*, x_\circ\rangle/y, \langle x'_*, x'_\circ\rangle/y'] & (\mathsf{Ty}_{f,L}(x) = \mathsf{mut}_\alpha(T\times T')) \end{cases}$$
(CHC-Stmt-Pair-Split)

The important rule is Chc-Stmt-Mutbor, the rule for a mutable (re)borrow. The first and second cases respectively correspond to a borrow and a reborrow. In both cases, we take a fresh variable $x_\circ$ that represents the value of the target object at the deadline of the (re)borrow. Letting x_* be the current target value, we model the created mutable reference as $\langle x_*, x_\circ\rangle$, the pair of the current and final target values. After the (re)borrow, the (current) target value of the lender is set to $x_\circ$, which is valid because the lender gets frozen in the type system. In the case of reborrow, letting $\langle x_*, x'_\circ\rangle$ be the original value of the lender mutable reference, the new value of the lender is set to $\langle x_\circ, x'_\circ\rangle$, where the lender's own final target value is retained.

When a mutable reference x is released (the second case of Chc-Stmt-Drop), the final target value of x is set to the current target value x_*. We use pattern matching here instead of equality, unlike informal explanation in §1.2. A similar thing happens when we weaken a mutable reference into an immutable reference (Chc-Stmt-Immut).

The rule for dereference $\mathsf{let}\ y = {*x}$ Chc-Stmt-Deref is tricky. This instruction turns a pointer to a pointer x into a pointer to the inner target object y. We have six cases here depending on the type information, or the pointer kinds of the outer and inner pointers. Let us see each case more closely. (i) An owning pointer to a pointer is simply dereferenced into the inner pointer. (ii, iii) An immutable reference to a pointer is dereferenced into an immutable reference to the inner object. If the inner pointer of x is a mutable reference, we discard the final target value. (iv) Dereference of a mutable reference to an owning pointer can be regarded as a *subdivision* of the mutable reference. (v) Dereference of a mutable reference to an immutable reference yields an immutable reference, which weakens the update permission of the inner mutable reference into the read permission. Therefore, we constrain the value of x in a manner similar to Chc-Stmt-Immut.[10] (vi) Dereference of a mutable reference to a mutable reference can be regarded as a *subdivision* of the outer mutable reference, as in the fourth case. At the low level, the address of the outer mutable reference is fixed to the current one by this dereference. Therefore, in our CHC representation, we fix the final target value of the inner mutable reference to the current one $x_{*\circ}$. A subtle point is that, for a mutable reference to a mutable reference, we can destructively update the address of the inner mutable reference (we can see an example of such update later in §5.2, in the function named swap_dec).

In the second case of Chc-Stmt-Match and the second case of CHC-Stmt-Pair-Split, we perform *subdivision* of a mutable reference. Both the current and final target values are subdivided by the operations. For CHC-Stmt-Pair-Split, when the mutable reference x turns into y, x loses the update permission to the tag of the variant.

[10] We actually don't need to support the case (v), because we can perform $\mathsf{immut}\ x$ before $\mathsf{let}\ y = {*x}$.

For a function call, the CHC body has a conjunction (Chc-Stmt-Call). Recall that in the operational semantics we add one stack frame when we call a function. The conjunction in the CHC body actually introduces a behavior analogous to the stack frame addition in an algorithm called resolution, as seen in §4.1 and §4.3.

When we return from a function (Chc-Stmt-Return), we set the return value res for the return statement. Again, instead of writing res $= x$, we use pattern matching to constrain res to be equal to x.

Example 3 (CHC Representation). We present below the CHC representation of the program presented in Example 1, consisting of take-max and inc-max. We give variables the following fixed linear order: oc', $o1$, ma, mb, or, oa, ob.

$$\begin{aligned}
&\mathsf{take\text{-}max}_{\mathsf{entry}}(\langle ma_*, ma_\circ\rangle, \langle mb_*, mb_\circ\rangle, \mathsf{res})\\
&\quad\Longleftarrow\ \mathsf{take\text{-}max}_{\mathsf{L1}}(\langle ma_*, ma_\circ\rangle, \langle mb_*, mb_\circ\rangle, \langle ma_* \ \texttt{>=}\ mb_*\rangle, \mathsf{res})
\end{aligned}$$

$$\mathsf{take\text{-}max}_{\mathsf{L1}}(ma, mb, \langle \mathsf{inj}_1\ ord_{*!}\rangle, \mathsf{res}) \ \Longleftarrow\ \mathsf{take\text{-}max}_{\mathsf{L2}}(ma, mb, \langle ord_{*!}\rangle, \mathsf{res})$$

$$\mathsf{take\text{-}max}_{\mathsf{L1}}(ma, mb, \langle \mathsf{inj}_0\ ord_{*!}\rangle, \mathsf{res}) \ \Longleftarrow\ \mathsf{take\text{-}max}_{\mathsf{L5}}(ma, mb, \langle ord_{*!}\rangle, \mathsf{res})$$

$$\mathsf{take\text{-}max}_{\mathsf{L2}}(ma, mb, ou, \mathsf{res}) \ \Longleftarrow\ \mathsf{take\text{-}max}_{\mathsf{L3}}(ma, mb, \mathsf{res})$$

$$\mathsf{take\text{-}max}_{\mathsf{L3}}(ma, \langle mb_*, mb_*\rangle, \mathsf{res}) \ \Longleftarrow\ \mathsf{take\text{-}max}_{\mathsf{L4}}(ma, \mathsf{res})$$

$$\mathsf{take\text{-}max}_{\mathsf{L4}}(ma, ma) \ \Longleftarrow\ \top$$

$$\mathsf{take\text{-}max}_{\mathsf{L5}}(ma, mb, ou, \mathsf{res}) \ \Longleftarrow\ \mathsf{take\text{-}max}_{\mathsf{L6}}(ma, mb, \mathsf{res})$$

$$\mathsf{take\text{-}max}_{\mathsf{L6}}(\langle ma_*, ma_*\rangle, mb, \mathsf{res}) \ \Longleftarrow\ \mathsf{take\text{-}max}_{\mathsf{L7}}(mb, \mathsf{res})$$

$$\mathsf{take\text{-}max}_{\mathsf{L7}}(mb, mb) \ \Longleftarrow\ \top$$

$$\mathsf{inc\text{-}max}_{\mathsf{entry}}(oa, ob, \mathsf{res}) \ \Longleftarrow\ \mathsf{inc\text{-}max}_{\mathsf{L1}}(oa, ob, \mathsf{res})$$

$$\mathsf{inc\text{-}max}_{\mathsf{L1}}(\langle oa_*\rangle, ob, \mathsf{res}) \ \Longleftarrow\ \mathsf{inc\text{-}max}_{\mathsf{L2}}(\langle oa_*, oa_\circ\rangle, \langle oa_\circ\rangle, ob, \mathsf{res})$$

$$\mathsf{inc\text{-}max}_{\mathsf{L2}}(ma, oa, \langle ob_*\rangle, \mathsf{res}) \ \Longleftarrow\ \mathsf{inc\text{-}max}_{\mathsf{L3}}(ma, \langle ob_*, ob_\circ\rangle, oa, \langle ob_\circ\rangle, \mathsf{res})$$

$$\mathsf{inc\text{-}max}_{\mathsf{L3}}(ma, mb, oa, ob, \mathsf{res}) \ \Longleftarrow\ \mathsf{take\text{-}max}_{\mathsf{entry}}(ma, mb, mc) \land \mathsf{inc\text{-}max}_{\mathsf{L4}}(mc, oa, ob, \mathsf{res})$$

$$\mathsf{inc\text{-}max}_{\mathsf{L4}}(mc, oa, ob, \mathsf{res}) \ \Longleftarrow\ \mathsf{inc\text{-}max}_{\mathsf{L5}}(\langle 1\rangle, mc, oa, ob, \mathsf{res})$$

$$\mathsf{inc\text{-}max}_{\mathsf{L5}}(\langle o1_*\rangle, \langle mc_*, mc_\circ\rangle, oa, ob, \mathsf{res}) \ \Longleftarrow\ \mathsf{inc\text{-}max}_{\mathsf{L6}}(\langle mc_* + o1_*\rangle, \langle o1_*\rangle, \langle mc_*, mc_\circ\rangle, oa, ob, \mathsf{res})$$

$$\mathsf{inc\text{-}max}_{\mathsf{L6}}(oc', o1, mc, oa, ob, \mathsf{res}) \ \Longleftarrow\ \mathsf{inc\text{-}max}_{\mathsf{L7}}(oc', mc, oa, ob, \mathsf{res})$$

$$\mathsf{inc\text{-}max}_{\mathsf{L7}}(\langle oc'_*\rangle, \langle mc_*, mc_\circ\rangle, oa, ob, \mathsf{res}) \ \Longleftarrow\ \mathsf{inc\text{-}max}_{\mathsf{L8}}(\langle mc_*\rangle, \langle oc'_*, mc_\circ\rangle, oa, ob, \mathsf{res})$$

$$\mathsf{inc\text{-}max}_{\mathsf{L8}}(oc', mc, oa, ob, \mathsf{res}) \ \Longleftarrow\ \mathsf{inc\text{-}max}_{\mathsf{L9}}(mc, oa, ob, \mathsf{res})$$

$$\mathsf{inc\text{-}max}_{\mathsf{L9}}(\langle mc_*, mc_*\rangle, oa, ob, \mathsf{res}) \ \Longleftarrow\ \mathsf{inc\text{-}max}_{\mathsf{L10}}(oa, ob, \mathsf{res})$$

$$\mathsf{inc\text{-}max}_{\mathsf{L10}}(oa, ob, \mathsf{res}) \ \Longleftarrow\ \mathsf{inc\text{-}max}_{\mathsf{L11}}(oa, ob, \mathsf{res})$$

$$\mathsf{inc\text{-}max}_{\mathsf{L11}}(\langle oa_*\rangle, \langle ob_*\rangle, \mathsf{res}) \ \Longleftarrow\ \mathsf{inc\text{-}max}_{\mathsf{L12}}(\langle oa_* \ \texttt{!=}\ ob_*\rangle, \langle oa_*\rangle, \langle ob_*\rangle, \mathsf{res})$$

$$\mathsf{inc\text{-}max}_{\mathsf{L12}}(or, oa, ob, \mathsf{res}) \ \Longleftarrow\ \mathsf{inc\text{-}max}_{\mathsf{L13}}(or, ob, \mathsf{res})$$

$$\mathsf{inc\text{-}max}_{\mathsf{L13}}(or, ob, \mathsf{res}) \ \Longleftarrow\ \mathsf{inc\text{-}max}_{\mathsf{L14}}(or, \mathsf{res})$$

$$\mathsf{inc\text{-}max}_{\mathsf{L14}}(or, or) \impliedby \top$$

The essence of this CHC system is the same as what we informally presented in §1.2. Note that here we use *pattern matching* to eliminate equalities, unlike the informal description.

3.3 Soundness and Completeness of Our Reduction

Now we formally state the soundness and completeness of our reduction and outline the proof of it. The complete proof is presented in §4.

In order to formally state the soundness and completeness of our CHC representation with respect to the actual behavior in the operational semantics, we first define the judgment to extract structured values from the heap memory and also check the safety condition on the heap memory based on ownership. Then, using that, we define the *OS-based model* f_Π^{OS} of a function f, which is a predicate that describes the input-output relation of the function f with respect to its behavior in the operational semantics. Here, for simplicity, f is restricted to what we call a *simple* function, i.e., a function whose input/output types do not contain mutable references. Finally, we state the soundness and completeness theorem and outline the proof of it.

Notation. We use $\{\!|\cdots|\!\}$ (instead of $\{\cdots\}$) for multisets. $A \oplus B$ (or more generally $\bigoplus_\lambda A_\lambda$) denotes the multiset sum. For example, $\{\!|0, 1|\!\} \oplus \{\!|1|\!\} = \{\!|0, 1, 1|\!\} \neq \{\!|0, 1|\!\}$.

Basic Extraction-Examination Judgment. We build a mechanism for *extracting structured values* from the heap memory, which is a finite map from addresses to integers. Also, we formally describe the safety condition on the heap memory with respect to the type information, which is designed to ensure invariants on permission. Because we currently target only *simple* functions, we can ignore mutable references and ignore frozen variables. (Later in §4, we extend the judgments to actually handle them.)

We first introduce the notion of *weak abstract configuration.*

$$\text{(weak abstract configuration)}\ \check{\mathcal{C}} ::= [f, L]\,\check{\mathcal{F}}$$

$$\text{(weak abstract stack frame)}\ \check{\mathcal{F}} ::= \text{(a finite map from variables to values)}$$

A *weak abstract configuration* is similar to a concrete configuration in the operational semantics, but maps each variable to a value and gets rid of the heap memory. The configuration has only one stack frame, since we target only the initial and final states of a function call. We also introduce some auxiliary notions. An *access mode* D is an item either of the form update or read, representing the permission on the memory access. A *memory footprint* $\mathcal{M}$ is a multiset of items of form $a[D]$.

Now we introduce the two basic judgments for *structurally extracting the value from the heap memory*, $\mathbf{H} \vdash_D a{:}\,P\,T \blacktriangleright v \mid \mathcal{M}$ and $\mathbf{H} \vdash_D *a{:}\,T \blacktriangleright v \mid \mathcal{M}$. The former structurally extracts from the heap memory $\mathbf{H}$ the pointer object typed $P\,T$ of the address a as a value v, yielding a memory footprint $\mathcal{M}$, under the access mode D. The latter is similar to the former but extracts the object typed T stored *at* the address a. The two judgments are mutually inductively defined by the following rules.

$$\frac{\mathbf{H} \vdash_{D\cdot\check{P}} *a{:}\,T \blacktriangleright v \mid \mathcal{M}}{\mathbf{H} \vdash_D a{:}\,\check{P}\,T \blacktriangleright \langle v\rangle \mid \mathcal{M}} \qquad D\cdot\mathsf{own} := D \qquad D\cdot\mathsf{immut}_\beta := \mathsf{read}$$

$$\frac{\mathbf{H}(a) = a' \quad \mathbf{H} \vdash_D a'{:}\,P\,T \blacktriangleright v \mid \mathcal{M}}{\mathbf{H} \vdash_D *a{:}\,P\,T \blacktriangleright v \mid \mathcal{M} \oplus \{\!|a[D]|\!\}}$$

$$\frac{\mathbf{H}(a) = i \in \{0, 1\} \quad \mathbf{H} \vdash_D *(a+1){:}\,T_i \blacktriangleright v \mid \mathcal{M} \quad N = \max\{|T_{1-i}| - |T_i|, 0\}}{\mathbf{H} \vdash_D *a{:}\,T_0 + T_1 \blacktriangleright \mathsf{inj}_i\, v \mid \mathcal{M} \oplus \{\!|a[D]|\!\} \oplus \{\!|(a + 1 + |T_i| + k)[D] \mid 0 \le k < N|\!\}}$$

$$\dfrac{\mathbf{H} \vdash_D *a{:}\, T_0 \blacktriangleright v_0 \mid \mathcal{M}_0 \quad \mathbf{H} \vdash_D *(a + |T_0|){:}\, T_1 \blacktriangleright v_1 \mid \mathcal{M}_1}{\mathbf{H} \vdash_D *a{:}\, T_0 \times T_1 \blacktriangleright (v_0, v_1) \mid \mathcal{M}_0 \oplus \mathcal{M}_1}$$

$$\dfrac{\mathbf{H} \vdash_D *a{:}\, T[\mu X.T/X] \blacktriangleright v \mid \mathcal{M}}{\mathbf{H} \vdash_D *a{:}\, \mu X.T \blacktriangleright v \mid \mathcal{M}} \qquad \dfrac{\mathbf{H}(a) = n}{\mathbf{H} \vdash_D *a{:}\, \mathsf{int} \blacktriangleright n \mid \{\!|a[D]|\!\}} \qquad \mathbf{H} \vdash_D *a{:}\, \mathsf{unit} \blacktriangleright () \mid \varnothing$$

For example, the following judgments hold ($\spadesuit$, $\heartsuit$, $\diamondsuit$ and $\clubsuit$ can be any addresses).

$$\{(\spadesuit, 7), (\spadesuit + 1, 5)\} \vdash_{\mathsf{update}} \spadesuit{:}\, \mathsf{own}\,(\mathsf{int} \times \mathsf{int}) \blacktriangleright \langle(7,5)\rangle \mid \{\!|\spadesuit[\mathsf{update}],\ (\spadesuit + 1)[\mathsf{update}]|\!\}$$

$$\{(\heartsuit, \diamondsuit), (\heartsuit + 1, \diamondsuit), (\diamondsuit, 3)\} \vdash_{\mathsf{update}} \heartsuit{:}\, \mathsf{own}\,(\mathsf{immut}_\alpha\, \mathsf{int} \times \mathsf{immut}_\alpha\, \mathsf{int})$$
$$\blacktriangleright \langle\langle 3\rangle, \langle 3\rangle\rangle \mid \{\!|\heartsuit[\mathsf{update}], (\heartsuit + 1)[\mathsf{update}], \diamondsuit[\mathsf{read}], \diamondsuit[\mathsf{read}]|\!\}$$

Next, we introduce the judgment $\mathbf{H} \vdash \mathbf{F}{:}\, \mathbf{\Gamma} \blacktriangleright \check{\mathcal{F}} \mid \mathcal{M}$ for extracting values from a concrete stack frame as a weak abstract stack frame, which is defined by the following rule.

$$\dfrac{\text{for each } x{:}\, PT \in \mathbf{\Gamma},\ \mathbf{H} \vdash_{\mathsf{update}} \mathbf{F}(x){:}\, PT \blacktriangleright v_x \mid \mathcal{M}_x}{\mathbf{H} \vdash \mathbf{F}{:}\, \mathbf{\Gamma} \blacktriangleright \{(x, v_x) \mid x\} \mid \bigoplus_x \mathcal{M}_x}$$

Using this, we introduce the judgment $\Pi \vdash \mathbf{C} \blacktriangleright \check{\mathcal{C}} \mid \mathcal{M}$ for extracting values from a concrete configuration as a weak abstract configuration, which is defined by the following rule.

$$\dfrac{\mathbf{H} \vdash \mathbf{F}{:}\, \mathbf{\Gamma}_{\Pi,f,L} \blacktriangleright \check{\mathcal{F}} \mid \mathcal{M}}{\Pi \vdash [f, L]\,\mathbf{F} \mid \mathbf{H} \blacktriangleright [f, L]\,\check{\mathcal{F}} \mid \mathcal{M}}$$

$\mathbf{\Gamma}_{\Pi,f,L}$: the data context at the program point (f, L) in the program Π

We introduce the safety judgment on a memory footprint $\vdash \mathcal{M}{:}\,\mathsf{ok}$. It is defined through an auxiliary judgment $\vdash^a \mathcal{M}{:}\,\mathsf{ok}$ as follows.

$$\dfrac{\text{for each } a,\ \vdash^a \mathcal{M}{:}\,\mathsf{ok}}{\vdash \mathcal{M}{:}\,\mathsf{ok}} \qquad \dfrac{\mathcal{M}^a = \varnothing}{\vdash^a \mathcal{M}{:}\,\mathsf{ok}} \qquad \dfrac{\mathcal{M}^a = \{\!|a[\mathsf{update}]|\!\}}{\vdash^a \mathcal{M}{:}\,\mathsf{ok}} \qquad \dfrac{\mathcal{M}^a = \{\!|\overrightarrow{a[\mathsf{read}]}|\!\}}{\vdash^a \mathcal{M}{:}\,\mathsf{ok}}$$

$\mathcal{M}^a$: the multiset of items of the form $a[\cdots]$ in $\mathcal{M}$

Finally, we introduce the *basic extraction-examination judgment* $\Pi \vdash \mathbf{C} \blacktriangleright^{\mathsf{ok}} \check{\mathcal{C}}$. It is the judgment for extracting a weak abstract configuration from a concrete configuration and also examining the safety and is defined by the following rule.

$$\dfrac{\Pi \vdash \mathbf{C} \blacktriangleright \check{\mathcal{C}} \mid \mathcal{M} \qquad \vdash \mathcal{M}{:}\,\mathsf{ok}}{\Pi \vdash \mathbf{C} \blacktriangleright^{\mathsf{ok}} \check{\mathcal{C}}}$$

OS-based Model. Now we define the *OS-based model* f_Π^{OS} of each simple function f in a program Π. It is the predicate that describes the input-output relation of the function f with respect to its behavior in the *operational semantics* (abbreviated as OS). We say that a function is simple when it does not take mutable references in the input and output types. The OS-based model f_Π^{OS} is defined as the predicate on values of sorts $\overrightarrow{(\!|T|\!)}$, $(\!|U|\!)$ where $\Sigma_{\Pi,f} = \langle\cdots\rangle(\overrightarrow{x{:}\,T}) \to U$, given by the following rule.

$$\dfrac{\Pi \vdash \mathbf{C} \to \cdots \to \mathbf{C}'{:}\,\mathsf{end} \qquad \Pi \vdash \mathbf{C} \blacktriangleright^{\mathsf{ok}} [f, \mathsf{entry}]\{\overrightarrow{(x, v)}\} \qquad \Pi \vdash \mathbf{C}' \blacktriangleright^{\mathsf{ok}} [f, L]\{(y, w)\}}{f_\Pi^{\mathrm{OS}}(\vec{v}, w)}$$

Soundness and Completeness Theorem. Finally, the soundness and completeness of our reduction is simply stated as follows.

Theorem 1 (Soundness and Completeness of Our Reduction). *For any well-typed program Π and any function f in Π, $\mathbf{M}^{\text{least}}_{(\!|\Pi|\!)}(f_{\text{entry}})$ is equivalent to f^{OS}_{Π}.*

The complete proof of the theorem is presented in §4. We outline the proof below.

Outline of the Proof. We first introduce a deduction algorithm on CHCs called *SLDC resolution*, which is a variant of SLD resolution [43]. We show that SLDC resolution is complete with respect to the least model of the CHC system (Lemma 2).

Next, we extend the basic extraction-examination judgment to accept mutable references and frozen variables. The key idea is to model the final target value $a_\circ$ of each mutable reference as a syntactic *variable* in logic, which is semantically universally quantified. Roughly speaking, a mutable reference is modeled as a pair of the current target value and a unique logic variable, and a frozen variable is modeled as a value with some borrowed parts remaining to be logic variables.

Finally, we complete the proof by establishing a bisimulation between the operational semantics and SLDC resolution under our CHC representation (Theorem 4). A key point is that, at the moment we release a mutable reference, we specialize the logic variable for the mutable reference. □

3.4 Advanced Examples

Here we present two advanced examples of verifying pointer-manipulating Rust programs by our reduction. For readability, we write CHCs again in an informal style like §1.

Example 4. Let us consider the following Rust program, which is a variant of `just_rec` in §1.1.

```rust
fn choose<'a>(ma: &'a mut i32, mb: &'a mut i32) -> &'a mut i32 {
  if rand() { ma } else { mb }
}
fn linger_dec<'a>(ma: &'a mut i32) -> bool {
  *ma -= 1; if rand() { return true; }
  let mut b = rand(); let b0 = b;
  { let mb = &mut b; let r2 = linger_dec(choose(ma, mb)); }
  r2 && b0 >= b
}
```

Unlike `just_rec`, the function `linger_dec` can modify the local variable of an arbitrarily deep ancestor. Each recursive call to `linger_dec` can introduce a new lifetime for `mb`, so arbitrarily many layers of lifetimes can be yielded.

The Rust program above can be expressed in COR as follows.

```
fn choose ⟨α⟩ (ma: mut_α int, mb: mut_α int) → mut_α int {
  entry: goto L1/L2      L1: drop mb; return ma      L2: drop ma; return mb
}
fn linger-dec ⟨α⟩ (ma: mut_α int) → own bool {
  entry: ∗ma -= 1; goto L1/L2      L1: drop ma; let ∗otrue = true; return ∗otrue
  L2: let ∗ob = rand(); let ∗ob_0 = copy ∗ob; intro β; let mb = mutbor_β ob;
   let mc = choose⟨β⟩(ma, mb); let or′ = linger-dec⟨β⟩(mc);
   now β; let ∗or″ = ∗ob_0 >= ∗ob; let ∗or = ∗or′ && ∗or″; drop(ob_0, ob, or′, or″); return or
}
```

For brevity, we admitted here the following features: the non-deterministic branching statement goto L/L' (which jumps to either L or L'), the decrement instruction $*x$ -= 1, the true-value taking instruction let $*y$ = true, the boolean conjunction instruction let $*y = *x$ && $*x'$, and the multiple-variable release instruction drop$(\vec{x})$. These additional features can be expressed by composition of original features. Also, we omitted some labels.

Suppose we wish to verify that `linger_dec` never returns **false**. If we use, like $\mathit{JustRec}_+$ in §1.1, a predicate taking the memory states h, h' and the stack pointer sp, we have to discover the quantified invariant: $\forall\, i \le sp.\ h[i] \ge h'[i]$. In contrast, our approach reduces this verification problem to the following CHCs.

$$\begin{aligned}
&\mathit{Choose}(\langle a,a_\circ\rangle,\langle b,b_\circ\rangle,r) \impliedby (b_\circ = b \land r = \langle a,a_\circ\rangle) \lor (a_\circ = a \land r = \langle b,b_\circ\rangle)\\
&\mathit{LingerDec}(\langle a,a_\circ\rangle,r) \impliedby (\, a_\circ = a-1 \land r = \mathrm{true}\,) \lor\\
&\qquad (\,\exists b, b_\circ, mc, r'.\, \mathit{Choose}(\langle a-1,a_\circ\rangle,\langle b,b_\circ\rangle,mc) \land \mathit{LingerDec}(mc,r') \land r = (r' \,\&\&\, b >= b_\circ)\,)\\
&r = \mathrm{true} \impliedby \mathit{LingerDec}(\langle a,a_\circ\rangle,r)
\end{aligned}$$

This can be solved by many CHC solvers since it has a very simple solution like below.

$$\begin{aligned}
\mathit{Choose}(\langle a,a_\circ\rangle,\langle b,b_\circ\rangle,r) &:\Longleftrightarrow (b_\circ = b \land r = \langle a,a_\circ\rangle) \lor (a_\circ = a \land r = \langle b,b_\circ\rangle)\\
\mathit{LingerDec}(\langle a,a_\circ\rangle,r) &:\Longleftrightarrow r = \mathrm{true} \land a \ge a_\circ
\end{aligned}$$

Example 5. Combined with *recursive data types*, our method turns out to be more powerful. Let us consider the following Rust program that features a *singly linked list*.

```
enum List<T> { Cons(T, Box<List<T>>), Nil } use List::*;
fn take_some<'a>(mla: &'a mut List<i32>) -> &'a mut i32 {
  match mla {
    Cons(ma, mla2) => if rand() { ma } else { take_some(mla2) }
    Nil => loop {}
  }
}
fn sum(la: &List<i32>) -> i32 {
  match la { Cons(a, la2) => a + sum(la2), Nil => 0 }
}
fn inc_some(mut la: List<i32>) -> bool {
  let n = sum(&la); let mb = take_some(&mut la);
  *mb += 1; let m = sum(&la); m == n + 1
}
```

This program handles a singly linked list type `List<T>`, which is a common recursive data type. The function `take_some` takes a mutable reference to an integer list and returns a mutable reference to some element of the list. The function `sum` calculates the sum of the elements of a list. The function `inc_some` increments some element of the input list using a mutable reference and checks that the sum of the elements of the list has increased by 1.

The Rust program above can be expressed in COR as follows.

```
fn take-some ⟨$\alpha$⟩ ($mla$: mut$_\alpha$ list int) → mut$_\alpha$ int {
  entry: $mla$ as mut$_\alpha$(int × own list int + unit); match $*mla$ { inj$_0$ $*mala'$ → L1, inj$_1$ $*mu$ → L4 }
   L1: let ($*ma$, $*mla'$) = $*mala'$; goto L2/L3       L2: drop $mla'$; return $ma$
   L3: drop $ma$; let $ma'$ = take-some⟨$\alpha$⟩($mla'$); return $ma'$       L4: goto L4
}
fn sum ⟨α⟩ (rla: immut_α list int) → own bool {
  entry: rla as immut_α(int × own list int + unit); match ∗rla { inj_0 ∗rala′ → L1, inj_1 ∗ru → L2 }
  L1: let (∗ra, ∗rla′) = ∗rala′; let on = sum⟨α⟩(rla′); let ∗or = ∗ra + ∗on; drop(ra, on);
   return or        L2: drop ru; let ∗o0 = 0; return o0
}
fn drop-list(ola: own list int) → own unit {· · ·}
fn inc-some(ola: own list int) → own bool {
  entry: intro α; let rla = immutbor_α ola; let on = sum⟨α⟩(rla); now α;
   intro β; let mla = mutbor_β ola; let mb = take-some⟨β⟩(mla); ∗mb += 1; drop mb; now β;
   intro γ; let rla = immutbor_γ ola; let om = sum⟨γ⟩(rla); now γ;
  ∗on += 1; let or = ∗om == ∗on; let ou = drop-list(ola); drop(ou, om, on); return or
}
```

Here, list T is sugar for the recursive type $\mu X.\, T\times\text{own}\, X+\text{unit}$. We have omitted the implementation of the function drop-list, which releases the data of a list of integers. Also, we have admitted the no-op jump statement goto L, the immutable borrow instruction let y = immutbor$_\alpha$ x and the increment instruction $*x$ += 1.

Suppose we wish to verify that `inc_some` never returns `false`. Our method reduces this verification problem into the following system of CHCs.

$$\begin{aligned}
&\mathit{TakeSome}(\langle a :: la', la_\circ\rangle, r) \impliedby \exists a_\circ, la'_\circ.\ la_\circ = a_\circ :: la'_\circ \ \land \\
&\qquad\qquad \big(\,(\,la'_\circ = la' \ \land\ r = \langle a, a_\circ\rangle\,)\ \lor\ (\,a_\circ = a\ \land\ \mathit{TakeSome}(\langle la', la'_\circ\rangle, r)\,)\,\big)\\
&\mathit{TakeSome}(\langle \text{nil}, la_\circ\rangle, r) \impliedby \mathit{TakeSome}(\langle \text{nil}, la_\circ\rangle, r)\\
&\mathit{Sum}(\langle a :: la'\rangle, r) \impliedby \exists r'.\ \mathit{Sum}(\langle la'\rangle, r') \land r = a + r'\\
&\mathit{Sum}(\langle \text{nil}\rangle, r) \impliedby r = 0\\
&\mathit{IncSome}(la, r) \impliedby \exists n, la_\circ, y, y_\circ, m.\ \mathit{Sum}(\langle la\rangle, n) \land \mathit{TakeSome}(\langle la, la_\circ\rangle, \langle y, y_\circ\rangle)\\
&\qquad\qquad \land\ y_\circ = y + 1 \land \mathit{Sum}(\langle la_\circ\rangle, m) \land r = (m == n{+}1)\\
&r = \text{true} \impliedby \mathit{IncSome}(la, r)
\end{aligned}$$

Here, nil denotes the nil list and $t :: u$ denotes the cons list made of the head t and the tail u. In our formal logic introduced in §3.1, they are respectively expressed as $\text{inj}_0(x, \langle lx\rangle)$ and $\text{inj}_1()$. An important technique used above is *subdivision of a mutable reference* performed in the function `take_some`. In the function `take_some` the mutable reference `mla` can be subdivided into mutable references to the head and tail of the list, which is expressed in the first CHC by the constraint $la_\circ = a_\circ :: la'_\circ$.

We can give this CHC system a *very simple solution*, using an auxiliary recursive function sum defined by $\text{sum}(a :: la') := a + \text{sum}(la')$ and $\text{sum}(\text{nil}) := 0$.

$$\begin{aligned}
\mathit{TakeSome}(\langle la, la_\circ\rangle, \langle b, b_\circ\rangle) &:\iff b_\circ - b = \text{sum}(la_\circ) - \text{sum}(la)\\
\mathit{Sum}(\langle la\rangle, r) &:\iff r = \text{sum}(la)\\
\mathit{IncSome}(la, r) &:\iff r = \text{true}.
\end{aligned}$$

The validity of the solution can be checked *without induction* about sum; specifically, we can check the validity of each CHC just by unfolding sum at most once. Notably, we do not need auxiliary notions like index access on lists to express the solution, which makes our approach scalable to richer recursive data types like *trees*.

Notably, in our experiments reported in §5, the example presented above was *fully automatically and promptly* verified by our prototype verifier RustHorn, using HoIce [12, 13] as the back-end CHC solver. Our verifier also successfully verified the variant of this example for *trees* instead of lists, which indicates high scalability of our approach for recursive data types. Note still that the CHC solver HoIce adopts a rather heuristic approach to find solutions that handle recursive functions over recursive data types (details are presented in [12]). For example, for the CHCs for `inc_some`, HoIce found the recursive function sum by analyzing the CHCs for the predicate variable *Sum*. It remains to be seen how well our approach verifies Rust programs with mutable references and recursive data types in general, given also that CHC solving techniques are still evolving.

3.5 Discussions

We discuss here various topics about our idea.

Combining Our Reduction with Various Verification Techniques. Our idea can also be expressed as a reduction of a pointer-manipulating Rust program into a program of a *stateless functional programming language*, which allows us to use *various verification techniques* not limited to CHCs. Access to future information can be modeled using *non-determinism*. To model the target value $a_\circ$ at the end of the mutable borrow, we just *randomly guess* the value with non-determinism. At the time we actually release a mutable reference, we just *check* `a' = a` and cut off execution branches that do not pass the check.

For example, `take_max`/`inc_max` in §1.2 and Example 1 can be reduced into the following OCaml program.

```
let rec assume b = if b then () else assume b
let take_max (a, a') (b, b') =
  if a >= b then (assume (b' = b); (a, a'))
            else (assume (a' = a); (b, b'))
let inc_max a b =
  let a' = Random.int(0) in let b' = Random.int(0) in
  let (c, c') = take_max (a, a') (b, b') in
  assume (c' = c + 1); not (a' = b')
let main a b = assert (inc_max a b)
```

Here, the bindings `let a' = Random.int(0)` and `let b' = Random.int(0)` take the future target values with *random guesses*, and the assumption checks `assume (b' = b)`, `assume (a' = a)` and `assume (c' = c + 1)` model the check of the random guesses. The original problem "Does `inc_max` never return `false`?" on the Rust program is reduced to the problem "Does `main` never fail at the assertion?" on the OCaml representation above. Notably, MoCHi [41], a higher-order model checker for OCaml, successfully verified the safety property for the OCaml representation above. It also successfully and instantly verified a similar OCaml representation of the Rust program of `linger_dec` presented at Example 4.

This representation allows us to use various verification techniques for Rust programs, including model checking (higher-order, temporal, bounded, etc.), semi-automated verification (e.g., in Boogie [50]) and verification in proof assistants (e.g., Coq [16]). The verified properties can be not only partial correctness but also total correctness and liveness. Also, our reduction can be used with various *bug finding* techniques such as symbolic testing (because we get an *equivalent* representation of the Rust program, as the theorem 1 states). Further investigation in these directions is needed.

Verifying Higher-Order Programs. Rust supports *closures*, internally encoding them as the tuple of the function pointer and the captured objects, creating a fresh internal type for each closure. Our reduction can support such closures simply by desugaring them as the captured objects.

As an advanced feature, Rust support *trait objects*, which performs *dynamic dispatch*. Using a trait object, Rust can use a *boxed closure*, which is required to get the full expressivity of higher-order programming. If we use rich verification frameworks like higher-order CHCs [11], our reduction can still model Rust programs that operate boxed closures, using some tricks. In order to model a closure that captures mutable references, we can equip the model of a closure with the 'drop predicate', which expresses the constraint that we should add when we release the closure. In order to model a closure that updates objects it captures, we can equip the output of the closure the updated version of the closure (using some recursive type). We need further investigation on verifying Rust programs with boxed closures and trait objects.

Libraries with Unsafe Code. Although the subset discussed earlier is quite limited, we can easily apply our reduction to some Rust libraries. For example, the vector (dynamically allocated array) type `Vec<T>` [64] can simply be represented as a functional array. In particular, we can model `Vec::index_mut(self: &mut Vec<T>, idx: usize)-> &mut T`, a function that takes out a mutable reference to some element of a vector out of a mutable reference to a vector. Also, we can support mutable iterators on a vector. Similarly, we can also support data structures like a hash map `HashMap<K, V>` [60]. We can also support some concurrency libraries like `thread::spawn` [63].

However, Rust libraries like `RefCell<T>` [59] and `Mutex<T>` [62] impose challenges to our method, because they introduce *shared mutable states* (or more technically, *interior mutability*). A naive approach is to pass around the global memory state for such data types. Here, let us discuss how to support `RefCell<T>`, which is a memory cell that attains dynamic permission control by reference counting and allows us to build data structures with circular references. We can model each instance of `RefCell` as an index and pass around the *global array* that maps each index of a `RefCell<T>` instance to a pair of the body value and the reference counter. To take a mutable or immutable reference from `RefCell`, we check and update the counter and take out the value from the array. At the time we take a mutable reference $\langle a, a_\circ \rangle$ from a `RefCell<T>`, the body value in the global array should be updated into $a_\circ$. This precisely models `RefCell`, but handling indices and the global array can be costly. We can also think of separating the array into smaller parts by methods like region-based type systems and pointer analysis.

Even when we find a model for some Rust library, verifying the implementation of the library itself can be tough, since it usually relies a lot on *unsafe code*, which is Rust code without static permission control. RustBelt [34] mechanically proved (in the Coq proof assistant) memory safety of well-typed Rust programs supporting various Rust libraries, including those with interior mutability. We discuss this work more in §6. Matsushita [48] discusses how to extend RustBelt to verify *functional correctness* with respect to our reduction, but the proof is not mechanized yet.

One caveat about our verification method is that it *loses completeness* in the presence of *memory leaks*. A *memory leak* [55] is an act to throw away an object without successful cleanup, such as memory deallocation and lock release. Although a basic subset of Rust (including the features supported in COR) does not allow memory leaks, advanced libraries in Rust can cause memory leaks. For example, when we build a cyclic graph using interior mutability by `RefCell` [59] and reference-counting garbage collection by `Rc` [61], we can cause a memory leak by isolating some cycle. When a leaked object has a mutable reference, we can fail at determining the final target value of it, which makes our method incomplete. Still, we don't think this is a major problem, because our method is still sound and in general program behaviors with memory leaks are very hard to verify any way.

4 PROOF OF THE SOUNDNESS AND COMPLETENESS OF OUR REDUCTION

In this section, we give the complete proof of Theorem 1 stated in §3.3, the soundness and completeness of our reduction.

Clearly the tricky point is that our model of a mutable reference $\langle a, a_\circ \rangle$ has *future information*, namely the final target value $a_\circ$. Our proof gets around this by keeping all possibilities about the future and narrowing them in the course of execution. A key ingredient is *resolution*, a deduction algorithm over CHCs that can handle syntactic *variables* that are universally quantified over values. This is nice for encoding future possibilities. Our proof goes by building a *bisimulation* between execution in Rust and resolution over the CHCs obtained by our reduction, where the final target value of each mutable reference is modeled by a syntactic variable in logic.

In §4.1, we introduce a special sort of resolution called *SLDC resolution*. In §4.2, we extend the extraction-examination judgments introduced in §1 to model mutable references and frozen variables. In §4.3, we complete the proof of Theorem 1, the soundness and completeness of our reduction, by establishing a bisimulation between program execution and SLDC resolution (Theorem 4).

4.1 SLDC Resolution

We introduce a deduction algorithm on CHC systems, which we call *SLDC resolution (Selective Linear Definite clause Calculative resolution)*. It is a variant of SLD resolution [43] with calculative steps. SLDC resolution is designed to be complete with respect to the logic (Lemma 2). Interpreting each CHC as a deduction rule, resolution can generally be understood as a top-down construction of a proof tree, and this idea is related to computation. As we see later, SLDC resolution is designed to form bisimulation with execution in the operational semantics (Theorem 4).

SLDC resolution is described as a transition system on *resolutive configurations* $\mathcal{K}$, which are of the form $\vec{\check{\varphi}} \mid p$. In a process of transition, it also uses *resolutive pre-configurations* $\hat{\mathcal{K}}$, which are of the form $\vec{\varphi} \mid p$. Recall that $\check{\varphi}$ is a meta-variable for a pattern formula, which does not have integer operators *op*; on the other hand, φ is a meta-variable for a usual formula, which can have operators. The pattern p on the right side of a configuration/pre-configuration is used to track how variables are instantiated. Later, SLDC resolution is associated with execution in the operational semantics; the pattern formulas $\vec{\check{\varphi}}$ in a configuration/pre-configuration can be understood as a model of a *call stack*, and the pattern p records the final return value. Resolutive (pre-)configurations that are alpha-equivalent are considered identical.

The one-step transition relation judgment of SLDC resolution $(\Phi, \Xi) \vdash \mathcal{K} \to \mathcal{K}'$ is defined by the following non-deterministic transformation.

(1) $\mathcal{K}$ should have one or more pattern formulas on the left side. Let $\mathcal{K} = f(\vec{p}), \vec{\check{\varphi}} \mid q$. Take from Φ any CHC Φ whose head formula unifies with $f(\vec{p})$. Namely, Φ is of the form $\forall \cdots.\, f(\vec{p'}) \Longleftarrow \bigwedge \vec{\psi}$ and $\vec{p'}$ unifies with $\vec{p}$. Take the most general unifier (θ, θ') on $\vec{p}$ and $\vec{p'}$, such that $p_i\,\theta = p'_i\,\theta'$ holds for each i. Here, θ and θ' are finite maps from variables to patterns. Now we have a pre-configuration $\hat{\mathcal{K}} = \overrightarrow{\psi\,\theta'}, \overrightarrow{\check{\varphi}\,\theta} \mid q\,\theta$.

(2) Now we calculate and specialize $\hat{\mathcal{K}}$ until we remove all operators and all variables that appear only once (which we call *orphaned* variables) in $\mathcal{K}$. By that, we obtain a configuration $\mathcal{K}'$ out of the pre-configuration $\hat{\mathcal{K}}$. Then we judge that $(\Phi, \Xi) \vdash \mathcal{K} \to \mathcal{K}'$ holds.
More precisely, the calculation and specialization process repeats the following operations (the order of the operations can actually be freely chosen).
- We replace a term of the form $n\ op\ n'$ with the integer taken by $n\, [\![op]\!]\, n'$.

- An orphaned variable in the pre-configuration is replaced with any value of the suitable sort.[11]
- When in the pre-configuration there occurs a term of form $x\ \mathit{op}\ t$ or $t\ \mathit{op}\ x$ for some variable x, we globally replace x with any integer n.[12]

Lemma 2 (Completeness of SLDC Resolution). *For any CHC system (Φ, Ξ), for each predicate variable f taking one or more arguments, the predicate given in the least solution $\mathbf{M}^{\text{least}}_{(\Phi,\Xi)}(f)$ is equivalent to the predicate f^{SLDC} on values of the appropriate sorts defined by the following rule.*

$$\frac{(\Phi, \Xi)\ \vdash\ f(\vec{v}, x) \mid x \to \cdots \to \epsilon \mid p \quad p \text{ unifies to } w}{f^{\text{SLDC}}(\vec{v}, w)}$$

Here, 'p unifies to w' means that replacing variables in p with some values we obtain w.

Proof. Similar to the proof of completeness of SLD resolution [43]. □

Example 6 (Resolution Sequence in SLDC Resolution). Below is an example resolution sequence in SLDC resolution for the CHC system presented in Example 3, which represents the program introduced in Example 1. It corresponds to the example execution in the operational semantics presented in Example 2.

$$\begin{aligned}
&\text{inc-max}_{\text{entry}}(\langle 5\rangle, \langle 3\rangle, r) \mid r\\
&\to\ \text{inc-max}_{\text{L1}}(\langle 5\rangle, \langle 3\rangle, r) \mid r\\
&\to^+\ \text{inc-max}_{\text{L3}}(\langle 5, a_\circ\rangle, \langle 3, b_\circ\rangle, \langle a_\circ\rangle, \langle b_\circ\rangle, r) \mid r\\
&\to\ \text{take-max}_{\text{entry}}(\langle 5, a_\circ\rangle, \langle 3, b_\circ\rangle, mc),\ \text{inc-max}_{\text{L4}}(mc, \langle a_\circ\rangle, \langle b_\circ\rangle, r) \mid r\\
&\to\ \text{take-max}_{\text{L1}}(\langle \text{inj}_1()\rangle, \langle 5, a_\circ\rangle, \langle 3, b_\circ\rangle, mc),\ \text{inc-max}_{\text{L4}}(mc, \langle a_\circ\rangle, \langle b_\circ\rangle, r) \mid r\\
&\to^+\ \text{take-max}_{\text{L3}}(\langle 5, a_\circ\rangle, \langle 3, b_\circ\rangle, mc),\ \text{inc-max}_{\text{L4}}(mc, \langle a_\circ\rangle, \langle b_\circ\rangle, r) \mid r\\
&\to\ \text{take-max}_{\text{L4}}(\langle 5, a_\circ\rangle, mc),\ \text{inc-max}_{\text{L4}}(mc, \langle a_\circ\rangle, \langle 3\rangle, r) \mid r\\
&\to\ \text{inc-max}_{\text{L4}}(\langle 5, a_\circ\rangle, \langle a_\circ\rangle, \langle 3\rangle, r) \mid r\\
&\to\ \text{inc-max}_{\text{L5}}(\langle 1\rangle, \langle 5, a_\circ\rangle, \langle a_\circ\rangle, \langle 3\rangle, r) \mid r\\
&\to^+\ \text{inc-max}_{\text{L7}}(\langle 6\rangle, \langle 5, a_\circ\rangle, \langle a_\circ\rangle, \langle 3\rangle, r) \mid r\\
&\to^+\ \text{inc-max}_{\text{L9}}(\langle 6, a_\circ\rangle, \langle a_\circ\rangle, \langle 3\rangle, r) \mid r\\
&\to\ \text{inc-max}_{\text{L10}}(\langle 6\rangle, \langle 3\rangle, r) \mid r\\
&\to^+\ \text{inc-max}_{\text{L14}}(\langle \text{inj}_1()\rangle, r) \mid r\\
&\to\ \epsilon \mid \langle \text{inj}_1()\rangle
\end{aligned}$$

In the third line ($\text{inc-max}_{\text{L3}}$), the mutable references ma and mb are modeled respectively as $\langle 5, a_\circ\rangle$ and $\langle 3, b_\circ\rangle$, where $a_\circ$ and $b_\circ$ are *logic variables* freshly taken for the borrows. Here, the frozen variables oa and ob are respectively modeled as $\langle a_\circ\rangle$ and $\langle b_\circ\rangle$. In the seventh line ($\text{take-max}_{\text{L4}}$), the mutable reference mb has been thrown away, and now $b_\circ$ is specialized to 3, which makes ob modeled as a value $\langle 3\rangle$ without a logic variable. In the twelfth line ($\text{inc-max}_{\text{L10}}$), the variable $a_\circ$ has now been specialized to 6. Note that each logic variable is specialized *before* the deadline of the corresponding borrow and the timing is determined dynamically.

[11] We need this rule for the random value instruction let $*y = \text{rand}()$ in establishing the bisimulation of Theorem 4.
[12] We add this rule for the completeness lemma Lemma 2. Actually we do not need this rule for resolution of a CHC representation $(\!|\Pi|\!)$.

4.2 Extending the Basic Extraction-Examination Judgment

We extend the basic extraction-examination judgment $\Pi \vdash \mathbf{C} \blacktriangleright^{\mathrm{ok}} \check{C}$ defined in §3.3 to accept mutable references and frozen variable. The key idea is to model the final target value $a_\circ$ of each mutable reference as a *logic variable*, which is semantically universally quantified. To model each variable, we now use a value with logic variables, i.e., a *pattern* p. A mutable reference is modeled as a pair of the current target pattern and a logic variable uniquely assigned to the mutable reference. Each mutably borrowed part of a frozen variable is set to the logic variable of the mutable reference that borrows that part.

The new extraction-examination judgment is of the form $\Pi \vdash \mathbf{C} \triangleright^{\mathrm{ok}} C$, where C is an *abstract configuration*, which is an extension of a weak abstract configuration with logic variables. To formally describe this judgment, we introduce new judgments for collecting the global information on lifetime variables.

The safety judgment on a concrete configuration $\Pi \vdash \mathbf{C}\colon \mathrm{ok}$ is defined as $\exists C.\ \Pi \vdash \mathbf{C} \triangleright^{\mathrm{ok}} C$. Later we prove the progress and preservation properties for this safety condition for a well-typed program, which can be regarded as the proof of soundness of the type system.

Taking Global and Dynamic Information on Lifetime Variables. First, for a well-typed program Π and a concrete configuration $\mathbf{C}$, we construct the *global lifetime context* $\mathbf{A}_{\Pi,\mathbf{C}}$, which is the lifetime context for all the stack frames in $\mathbf{C}$. A local lifetime α in the i-th stack frame (indexed from the bottom) is named α^i. It also has the *global* elimination order, which is constructed based on the promises in each function call (i.e., the lifetime context given by the type system for each stack frame) and the hierarchy of stack frames (i.e., the property that $i \geq j$ implies $\alpha^i \leq \beta^j$). We also add to the global lifetime context the lifetime parameters in the base stack frame. Formally, $\mathbf{A}_{\Pi,\mathbf{C}}$ is defined as follows.

$$\mathbf{A}_{\Pi,\mathbf{C}} := \bigl(\{\alpha^i \mid i,\ \alpha \in A^!_i\},\ \{(\alpha^i,\beta^i) \mid i,\ (\alpha,\beta) \in \leq^!_i\} + \{(\alpha^i,\beta^j) \mid i \geq j,\ \alpha \in A^!_i,\ \beta \in A^!_j\}\bigr)$$

$$\begin{aligned}\text{where}\quad & \mathbf{C} := [f_n, L_n]\,\mathbf{F}_n;\ [f_{n-1}, L_{n-1}]\,x_{n-1}, \mathbf{F}_{n-1};\ \cdots\ [f_0, L_0]\,x_0, \mathbf{F}_0 \qquad \mathbf{A}_i := \mathbf{A}_{\Pi,f_i,L_i}\\ & A^!_{i+1} := |\mathbf{A}_{i+1}| - A_{\mathrm{ex}\,\Pi,f_{i+1},L_{i+1}}\ \ \text{(i.e., the set of local lifetimes)} \qquad A^!_0 := |\mathbf{A}_0|\\ & \leq^!_i := \{(\alpha,\beta) \mid (\alpha,\beta) \in \leq_{\mathbf{A}_i},\ \alpha,\beta \in A^!_i\}\\ & \quad \mathbf{A}_{\Pi,f,L}\colon \text{ the lifetime context assigned to } (f,L) \text{ in } \Pi \text{ by the type system}\end{aligned}$$

Also, for each stack frame, indexed i, we define the *lifetime substitution* $\mathbf{\Theta}_{\Pi,\mathbf{C},i}$. It maps each lifetime variable in the stack frame to the corresponding lifetime variable in the global lifetime context $\mathbf{A}_{\Pi,\mathbf{C}}$. For each local lifetime α in the stack frame, we just add the tag α^i. For each lifetime parameter, we should find the lifetime variable assigned to it. Therefore, formally, $\mathbf{\Theta}_{\Pi,\mathbf{C},i}$ is defined as follows.[13]

$$\mathbf{\Theta}_{\Pi,\mathbf{C},i+1} := \{(\alpha,\alpha^{i+1}) \mid \alpha \in A^!_{i+1}\} + \{\overrightarrow{(\beta_{i+1},\ \gamma_{i+1}\mathbf{\Theta}_{\Pi,\mathbf{C},i})}\} \qquad \mathbf{\Theta}_{\Pi,\mathbf{C},0} := \{(\alpha,\alpha^0) \mid \alpha \in A^!_0\}$$

$$\begin{aligned}\text{where}\quad & \mathbf{C} := [f_n, L_n]\,\mathbf{F}_n;\ [f_{n-1}, L_{n-1}]\,x_{n-1}, \mathbf{F}_{n-1};\ \cdots\ [f_0, L_0]\,x_0, \mathbf{F}_0 \qquad \mathbf{A}_i := \mathbf{A}_{\Pi,f_i,L_i}\\ & A^!_{i+1} := |\mathbf{A}_{i+1}| - A_{\mathrm{ex}\,\Pi,f_{i+1},L_{i+1}} \qquad A^!_0 := |\mathbf{A}_0|\\ & \Sigma_{\Pi,f_{i+1},L_{i+1}} := \langle \overrightarrow{\beta_{i+1}} \mid \cdots\rangle(\cdots) \to \cdots\\ & \text{the function call for each stack frame indexed } i < n \text{ is } \mathsf{let}\, x_i = f_{i+1}\langle\overrightarrow{\gamma_{i+1}}\rangle(\cdots)\end{aligned}$$

[13] For simplicity, we assume that for each non-top stack frame we can uniquely determine the label of the function call statement performed for creating the frame, which allows us to determine $\overrightarrow{\gamma_{i+1}}$. We can always satisfy this by inserting a fresh no-op labeled statement just after the function call.

Extracting Rich Information from the Heap Memory. First, we define an *abstract configuration* $\mathcal{C}$ and an *abstract stack frame* $\mathcal{F}$ as follows.

$$\text{(abstract configuration)}\ \ \mathcal{C}\ ::=\ [f_0, L_0]\,\mathcal{F}_0;\ [f_1, L_1]\,x_1, \mathcal{F}_1;\ \cdots\ [f_n, L_n]\,x_n, \mathcal{F}_n$$

$$\text{(abstract stack frame)}\ \ \mathcal{F}\ ::=\ \text{(a finite map from variables to patterns)}$$

They correspond to a concrete configuration **C** and a concrete stack frame **F**, but map each data variable to a *pattern*, which may contain *logic variables.* The use of the logic variables here is related to the notion of prophecy variables [1, 36, 73].

We also introduce some auxiliary notions. A *logic variable summary* $\mathcal{X}$ is a finite multiset of items of the form described below.

(item of a logic variable summary)

$$::=\ \uparrow_\alpha x\,[*a\colon T]\ \ \text{(the 'giver' on } x\text{, which promises to store an object typed } T \text{ at the address } a \text{ before the lifetime } \alpha)$$
$$\mid\ \downarrow^\alpha x\,[*a\colon T]\ \ \text{(the 'taker' on } x\text{, which expects to obtain an object typed } T \text{ at the address } a \text{ at the lifetime } \alpha)$$

A logic variable summary records how logic variables are used in extracting patterns from a concrete configuration. An *extended memory footprint* $\hat{\mathcal{M}}$ is a finite multiset of items of the form described below.

$$\text{(item of an extended memory footprint)}\ ::=\ a[\mathsf{update}^{\mathbf{a}}]\ \ \text{(update access to the address } a \text{ under the activeness } \mathbf{a})$$
$$\mid\ a[\mathsf{read}_\alpha]\ \ \text{(read access to the address } a \text{ allowed until the lifetime } \alpha)$$

The activeness $\mathbf{a}$ has been introduced for data contexts in §2.2; it is of the form actv (active) or $\dagger\beta$ (borrowed until the lifetime β). An extended memory footprint records the memory access employed in extracting the data from a concrete configuration. An *extended access mode* $\hat{D}$ is an item of the form either update or read_α.

Now we introduce the two basic *extraction* judgments, $\mathbf{H}\ \vdash^{\mathbf{a}}_{\hat{D}}\ a\colon P\,T \triangleright p \mid \mathcal{X}, \hat{\mathcal{M}}$ and $\mathbf{H}\ \vdash^{\mathbf{a}}_{\hat{D}}\ *a\colon T \triangleright p \mid \mathcal{X}, \hat{\mathcal{M}}$. The former structurally extracts from the heap memory **H** the pointer object typed $P\,T$ of the address a as a *pattern* p, yielding a logic variable summary $\mathcal{X}$ and the extended memory footprint $\mathcal{M}$, under the activeness $\mathbf{a}$ and the extended access mode $\hat{D}$. The latter is similar to the former but extracts the object typed T stored *at* the address a. They are defined by the following rules.

$$\frac{\mathbf{H}\ \vdash^{\mathbf{a}}_{\hat{D}\cdot\check{P}}\ *a\colon T \triangleright p \mid \mathcal{X}, \hat{\mathcal{M}}}{\mathbf{H}\ \vdash^{\mathbf{a}}_{\hat{D}}\ a\colon \check{P}\,T \triangleright \langle p\rangle \mid \mathcal{X}, \hat{\mathcal{M}}} \qquad \hat{D}\cdot\mathsf{own}\ :=\ \hat{D} \quad \mathsf{update}\cdot\mathsf{immut}_\beta\ :=\ \mathsf{read}_\beta \quad \mathsf{read}_\alpha\cdot\mathsf{immut}_\beta\ :=\ \mathsf{read}_\alpha$$

$$\frac{\mathbf{H}\ \vdash^{\mathbf{a}}_{\mathsf{update}}\ *a\colon T \triangleright p \mid \mathcal{X}, \hat{\mathcal{M}}}{\mathbf{H}\ \vdash^{\mathbf{a}}_{\mathsf{update}}\ a\colon \mathsf{mut}_\beta\,T \triangleright \langle p, x\rangle \mid \mathcal{X} \oplus \{\!|\uparrow_\beta x\,[*a\colon T]|\!\},\ \hat{\mathcal{M}}} \quad \text{(Extract-Mut-Update)}$$

$$\frac{\mathbf{H}\ \vdash^{\mathbf{a}}_{\mathsf{read}_\alpha}\ *a\colon T \triangleright p \mid \mathcal{X}, \hat{\mathcal{M}}}{\mathbf{H}\ \vdash^{\mathbf{a}}_{\mathsf{read}_\alpha}\ a\colon \mathsf{mut}_\beta\,T \triangleright \langle p, q\rangle \mid \mathcal{X}, \hat{\mathcal{M}}} \quad \text{(Extract-Mut-Read)}$$

$$a[\hat{D}^{\mathbf{a}}]\ :=\ \begin{cases} a[\mathsf{update}^{\mathbf{a}}] & (\hat{D} = \mathsf{update}) \\ a[\mathsf{read}_\beta] & (\hat{D} = \mathsf{read}_\beta) \end{cases}$$

$$\mathbf{H}\ \vdash^{\dagger\alpha}_{\hat{D}}\ *\,a\colon T \triangleright x \mid \{\!|\downarrow^\alpha x\,[*a\colon T]|\!\}, \varnothing \quad \text{(Extract-Take-Variable)}$$

$$\frac{\mathbf{H}(a) = a' \quad \mathbf{H} \vdash^{\mathbf{a}}_{\hat{D}} a' {:} P\,T \triangleright p \mid \mathcal{X}, \hat{\mathcal{M}}}{\mathbf{H} \vdash^{\mathbf{a}}_{\hat{D}} *a {:} P\,T \triangleright p \mid \mathcal{X}, \hat{\mathcal{M}} \oplus \{\!|a[\hat{D}^{\mathbf{a}}]|\!\}}$$

$$\frac{\mathbf{H}(a) = i \in \{0,1\} \quad \mathbf{H} \vdash^{\mathbf{a}}_{\hat{D}} *(a+1) {:} T_i \triangleright p \mid \mathcal{X}, \hat{\mathcal{M}} \quad N = \max\{|T_{1-i}| - |T_i|, 0\}}{\mathbf{H} \vdash^{\mathbf{a}}_{\hat{D}} *a {:} T_0 + T_1 \triangleright \mathsf{inj}_i\, p \mid \mathcal{X}, \hat{\mathcal{M}} \oplus \{\!|a[\hat{D}^{\mathbf{a}}]|\!\} \oplus \{\!|(a+1+|T_i|+k)[\hat{D}^{\mathbf{a}}] \mid 0 \le k < N|\!\}}$$

$$\frac{\mathbf{H} \vdash^{\mathbf{a}}_{\hat{D}} *a {:} T_0 \triangleright p_0 \mid \mathcal{X}_0, \hat{\mathcal{M}}_0 \quad \mathbf{H} \vdash^{\mathbf{a}}_{\hat{D}} *(a+|T_0|) {:} T_1 \triangleright p_1 \mid \mathcal{X}_1, \hat{\mathcal{M}}_1}{\mathbf{H} \vdash^{\mathbf{a}}_{\hat{D}} *a {:} T_0 \times T_1 \triangleright (p_0, p_1) \mid \mathcal{X}_0 \oplus \mathcal{X}_1,\ \hat{\mathcal{M}}_0 \oplus \hat{\mathcal{M}}_1}$$

$$\frac{\mathbf{H} \vdash^{\mathbf{a}}_{\hat{D}} *a {:} T[\mu X.T/X] \triangleright p \mid \mathcal{X}, \hat{\mathcal{M}}}{\mathbf{H} \vdash^{\mathbf{a}}_{\hat{D}} *a {:} \mu X.T \triangleright p \mid \mathcal{X}, \hat{\mathcal{M}}}$$

$$\frac{\mathbf{H}(a) = n}{\mathbf{H} \vdash^{\mathbf{a}}_{\hat{D}} *a {:} \mathsf{int} \triangleright n \mid \emptyset, \{\!|a[\hat{D}^{\mathbf{a}}]|\!\}} \qquad \mathbf{H} \vdash^{\mathbf{a}}_{\hat{D}} *a {:} \mathsf{unit} \triangleright () \mid \emptyset, \emptyset$$

The two judgments are extensions of the judgments $\mathbf{H} \vdash_D a {:} P\,T \blacktriangleright v \mid \mathcal{M}$ and $\mathbf{H} \vdash_D *a {:} T \blacktriangleright v \mid \mathcal{M}$ introduced in §3.3.

For the judgment $\mathbf{H} \vdash^{\mathbf{a}}_{\hat{D}} a {:} P\,T \triangleright p \mid \mathcal{X}, \hat{\mathcal{M}}$, we have two rules for extracting the data of a mutable reference, namely Extract-Mut-Update and Extract-Mut-Read. The former is the one used for update access; in this case, we use a logic variable x at the second argument of the mut container and record the variable into the logic variable summary. The latter is the one used for read access; in this case, we do not care about the second argument of the mut container.

Also, for the judgment $\mathbf{H} \vdash^{\mathbf{a}}_{\hat{D}} *a {:} T \triangleright p \mid \mathcal{X}, \hat{\mathcal{M}}$, we can stop exploring the heap memory and just return a logic variable using the rule Extract-Take-Variable. Although we impose here no special restriction on using this rule, the use of the rule is recorded in the logic variable summary $\mathcal{X}$ as the 'taker' item $\downarrow^{\alpha} x[*a {:} T]$. Later, in the safety condition on the logic variable summary, we require that the giver and the taker correspond one to one with some agreement conditions (Safe-Summary-Correspond).

Next, we introduce the judgment for extracting the data of a concrete stack frame into an abstract stack frame, $\mathbf{H}, \mathbf{\Theta} \vdash \mathbf{F} {:} \mathbf{\Gamma} \triangleright \mathcal{F} \mid \mathcal{X}, \hat{\mathcal{M}}$. Here, $\mathbf{\Theta}$ is a substitution on lifetime variables associated with the stack frame (soon later, $\mathbf{\Theta}_{\Pi,\mathbf{C},i}$ is assigned to $\mathbf{\Theta}$). It is defined by the following rule.

$$\frac{\text{for each } x {:}^{\mathbf{a}} T \in \mathbf{\Gamma},\ \mathbf{H} \vdash^{\mathbf{a}}_{\mathsf{update}} \mathbf{F}(x) {:}\ T\mathbf{\Theta} \triangleright p_x \mid \mathcal{X}_x, \hat{\mathcal{M}}_x}{\mathbf{H}, \mathbf{\Theta} \vdash \mathbf{F} {:} \mathbf{\Gamma} \triangleright \{(x, p_x) \mid x\} \mid \bigoplus_x \mathcal{X}_x,\ \bigoplus_x \hat{\mathcal{M}}_x}$$

It is an extension of the judgment $\mathbf{H} \vdash \mathbf{F} {:} \mathbf{\Gamma} \blacktriangleright \check{\mathcal{F}} \mid \mathcal{M}$ introduced in §3.3. Since the logic variable summary records the information of lifetime variables in types, we apply the lifetime substitution $\mathbf{\Theta}$ to the type T of each variable.

Now we define the judgment for extracting values from the concrete configuration as an abstract configuration, $\Pi \vdash \mathbf{C} \triangleright C \mid \mathcal{X}, \hat{\mathcal{M}}$, by the following rule.

$$\frac{\begin{array}{c}\mathbf{C} \ =\ [f_0, L_0]\,\mathbf{F}_0;\ [f_1, L_1]\,x_1, \mathbf{F}_1;\ \cdots;\ [f_n, L_n]\,x_n, \mathbf{F}_n \mid \mathbf{H} \\ \text{for each } i,\ \ \mathbf{H}, \mathbf{\Theta}_{\Pi,\mathbf{C},i} \vdash \mathbf{F}_i {:}\ \mathbf{\Gamma}_{\Pi,f_i,L_i} \triangleright \mathcal{F}_i \mid \mathcal{X}_i, \hat{\mathcal{M}}_i\end{array}}{\Pi \vdash \mathbf{C} \triangleright [f_0, L_0]\,\mathcal{F}_0;\ [f_1, L_1]\,x_1, \mathcal{F}_1;\ \cdots;\ [f_n, L_n]\,x_n, \mathcal{F}_n \mid \bigoplus_i \mathcal{X}_i, \bigoplus_i \hat{\mathcal{M}}_i}$$

It is an extension of the judgment $\Pi \vdash \mathbf{C} \blacktriangleright \check{C} \mid \mathcal{M}$ introduced in §3.3.

Examining the Safety. Now we define safety conditions.

First, we introduce the safety judgment on a logic variable summary $\mathbf{A} \vdash \mathcal{X} \colon \mathsf{ok}$. It uses the global lifetime context $\mathbf{A}$. It is defined by the following rules, using an auxiliary judgment $\mathbf{A} \vdash^{x} \mathcal{X} \colon \mathsf{ok}$.

$$\frac{\text{for each } x,\ \mathbf{A} \vdash^{x} \mathcal{X} \colon \mathsf{ok}}{\mathbf{A} \vdash \mathcal{X} \colon \mathsf{ok}} \qquad \frac{\mathcal{X}^{x} = \varnothing}{\mathbf{A} \vdash^{x} \mathcal{X} \colon \mathsf{ok}}$$

$$\frac{\mathcal{X}^{x} = \{\!|\, \uparrow_{\alpha} x[*a \colon T],\ \downarrow^{\beta} x[*a \colon T'] \,|\!\} \quad \mathbf{A} \vdash T \leq T',\ T' \leq T \quad \alpha \leq_{\mathbf{A}} \beta}{\mathbf{A} \vdash^{x} \mathcal{X} \colon \mathsf{ok}}$$

(Safe-Summary-Correspond)

$\mathcal{X}^{x}$: the multiset of the items of the form $\uparrow_{\cdots} x[\cdots]$ or $\downarrow^{\cdots} x[\cdots]$ in $\mathcal{X}$

For each logic variable x such that $\mathcal{X}^{x} \neq \varnothing$, we require that the logic variable summary $\mathcal{X}$ has exactly one giver and one taker of x, that they agree on the address and the type (up to equivalence), and that the lifetime of the giver is no later than the lifetime of the taker (Safe-Summary-Correspond).

Next, we introduce the safety judgment on an extended memory footprint $\mathbf{A} \vdash \hat{\mathcal{M}} \colon \mathsf{ok}$. It is defined by the following rules, using an auxiliary judgment $\mathbf{A} \vdash^{a} \hat{\mathcal{M}} \colon \mathsf{ok}$.

$$\frac{\text{for each } a,\ \mathbf{A} \vdash^{a} \hat{\mathcal{M}} \colon \mathsf{ok}}{\mathbf{A} \vdash \hat{\mathcal{M}} \colon \mathsf{ok}} \qquad \frac{\hat{\mathcal{M}}^{a} = \varnothing}{\mathbf{A} \vdash^{a} \hat{\mathcal{M}} \colon \mathsf{ok}} \qquad \frac{\hat{\mathcal{M}}^{a} = \{\!|\, a[\mathsf{update}^{\mathsf{actv}}] \,|\!\}}{\mathbf{A} \vdash^{a} \hat{\mathcal{M}} \colon \mathsf{ok}} \qquad \frac{\hat{\mathcal{M}}^{a} = \{\!|\, \overrightarrow{a[\mathsf{read}_{\beta}]} \,|\!\}}{\mathbf{A} \vdash^{a} \hat{\mathcal{M}} \colon \mathsf{ok}}$$

$$\frac{\hat{\mathcal{M}}^{a} = \{\!|\, a[\mathsf{update}^{\dagger\alpha}],\ \overrightarrow{a[\mathsf{read}_{\beta}]} \,|\!\} \quad \text{for each } i,\ \beta_i \leq_{\mathbf{A}} \alpha}{\mathbf{A} \vdash^{a} \hat{\mathcal{M}} \colon \mathsf{ok}}$$

(Safe-Footprint-Update-Read)

$\hat{\mathcal{M}}^{a}$: the multiset of items of the form $a[\cdots]$

This judgment is an extension of $\vdash \hat{\mathcal{M}} \colon \mathsf{ok}$ introduced in §3.3. We now have to deal with frozen access. Since a mutable (re)borrow completely masks the lender with a logic variable, even if we have $a[\mathsf{update}^{\mathsf{actv}}]$ that comes form a mutable reference, we do not have any other items of the form $a[\cdots]$, which keeps the situation simple. When we have shared references that comes from some lender, we need to see agreement between the frozen update access and the borrowed read access, which is performed by the rule Safe-Footprint-Update-Read.

Finally, we define the *extended extraction-examination judgment* $\Pi \vdash \mathbf{C} \triangleright^{\mathsf{ok}} C$ by the following rule.

$$\frac{\Pi \vdash \mathbf{C} \triangleright C \mid \mathcal{X}, \hat{\mathcal{M}} \quad \mathbf{A}_{\Pi,\mathbf{C}} \vdash \mathcal{X} \colon \mathsf{ok} \quad \mathbf{A}_{\Pi,\mathbf{C}} \vdash \hat{\mathcal{M}} \colon \mathsf{ok}}{\Pi \vdash \mathbf{C} \triangleright^{\mathsf{ok}} C}$$

Safety Condition on a Concrete Configuration. The safety judgment on a concrete configuration, $\Pi \vdash \mathbf{C} \colon \mathsf{ok}$, is defined simply as $\exists C.\, \Pi \vdash \mathbf{C} \triangleright^{\mathsf{ok}} C$.

For any well-typed program, we have the *progress* property on the safety condition.

Proposition 3 (Safety on a Concrete Configuration Ensures Progress). *For any Π and $\mathbf{C}$, if $\Pi \vdash \mathbf{C} \colon \mathsf{ok}$ holds and $\Pi \vdash \mathbf{C} \colon \mathsf{end}$ does not hold, then there exists some $\mathbf{C}'$ satisfying $\Pi \vdash \mathbf{C} \to \mathbf{C}'$.*

Proof. It can be easily proved by a straightforward case analysis. The safety condition simply ensures that the data stored in the heap memory has the expected forms. □

The *preservation* property on the safety condition also holds. It is later shown (Corollary 5) as a corollary of the bisimulation theorem (Theorem 4).

4.3 Bisimulation Between Execution and SLDC Resolution

Now we define the judgment, or relation, $\Pi \vdash \mathbf{C} \triangleright^{\mathrm{ok}} \mathcal{K}$, which links the world of the operational semantics and the world of SLDC resolution. We prove that the relation forms a *bisimulation* between execution in the operational semantics and SLDC resolution in our CHC representation (Theorem 4). Using this bisimulation, we complete the proof of Theorem 1, the soundness and completeness of our reduction. A key point is that, at the moment we release a mutable reference modeled as $\langle p, x_\circ \rangle$, we specialize the logic variable $x_\circ$ into the current target pattern p.

The judgment $\Pi \vdash \mathbf{C} \triangleright^{\mathrm{ok}} \mathcal{K}$, which translates a concrete configuration $\mathbf{C}$ into a logic configuration $\mathcal{K}$, is defined as follows.

$$\frac{\Pi \vdash \mathbf{C} \triangleright^{\mathrm{ok}} C \qquad C \,=\, [f_n, L_n]\,\{\overrightarrow{(x_n, p_n)}\};\ [f_{n-1}, L_{n-1}]\ y_n, \{\overrightarrow{(x_{n-1}, p_{n-1})}\};\ \cdots\ [f_0, L_0]\ y_1, \{\overrightarrow{(x_0, p_0)}\}}{\Pi \vdash \mathbf{C} \triangleright_{\mathrm{ok}} f_{n\,L_n}(\overrightarrow{p_n}, z_n),\ f_{n-1\,L_{n-1}}(z_n, \overrightarrow{p_{n-1}}, z_{n-1}),\ \cdots\ f_{0\,L_0}(z_1, \overrightarrow{p_0}, z_0) \mid z_0}$$

Here, $\mathcal{K}$ is designed as a resolutive configuration for our CHC representation of the program $(\!|\Pi|\!)$. For simplicity, we assumed here that the arguments of the predicate $f_{i\,L_i}$ are in the order of $y_{i+1}, \overrightarrow{x_i}$, res for each $i < n$. The variables $z_0, \ldots, z_n$ are fresh logic variables that are mutually distinct.

The relation $\Pi \vdash \mathbf{C} \triangleright^{\mathrm{ok}} \mathcal{K}$ forms a bisimulation between execution in the operational semantics and SLDC resolution in our CHC representation.

Theorem 4 (Bisimulation Between Execution and SLDC Resolution Under Our CHC Representation). *Assume that $\Pi \vdash \mathbf{C} \triangleright^{\mathrm{ok}} \mathcal{K}$ holds. For any $\mathbf{C}'$ satisfying $\Pi \vdash \mathbf{C} \to \mathbf{C}'$, there exists $\mathcal{K}'$ such that $(\!|\Pi|\!) \vdash \mathcal{K} \to \mathcal{K}'$ and $\Pi \vdash \mathbf{C}' \triangleright^{\mathrm{ok}} \mathcal{K}'$ hold. Likewise, for any $\mathcal{K}'$ satisfying $(\!|\Pi|\!) \vdash \mathcal{K} \to \mathcal{K}'$, there exists $\mathbf{C}'$ such that $\Pi \vdash \mathbf{C} \to \mathbf{C}'$ and $\Pi \vdash \mathbf{C}' \triangleright^{\mathrm{ok}} \mathcal{K}'$ hold.*

Proof. Taking a close look at each type of statements, we can find a correspondence between a transition on concrete configurations and the transition on resolutive configurations under our CHC representation. Therefore, we can choose $\mathcal{K}'$ based on $\mathbf{C}'$ and choose $\mathbf{C}'$ based on $\mathcal{K}'$ (we do not explicitly describe this choice here). The question is whether $\Pi \vdash \mathbf{C}' \triangleright^{\mathrm{ok}} \mathcal{K}'$ really holds. Let C' the abstract configuration associated with $\mathcal{K}'$. The property $\Pi \vdash \mathbf{C}' \triangleright^{\mathrm{ok}} \mathcal{K}'$ can be broken into (i) whether the extraction judgment $\Pi \vdash \mathbf{C}' \triangleright C' \mid \mathcal{X}', \hat{\mathcal{M}}'$ holds, (ii) whether the safety condition on the logic variable summary $\mathcal{X}'$ holds, and (iii) whether the safety condition on the extended memory footprint $\hat{\mathcal{M}}'$ holds. We can show $\Pi \vdash \mathbf{C}' \triangleright^{\mathrm{ok}} \mathcal{K}'$ under the assumptions by some case analyses. Below we give more detailed illustrations for some types of transitions.

Manipulation of Owning Pointers. Some transitions manipulates the target objects of some owning pointers. For example, the instruction $\mathsf{let}\ {*y} = ({*x}, {*x'})$ moves the memory sequences of x and x' to allocate the two at one consecutive memory region. Also, the swap instruction $\mathsf{swap}(x, y)$ destructively can update the target object of an ownership pointer.

In order to handle such operations, the type system and the extraction judgments give an important guarantee: the manipulated memory cells should always be accessed with the active update permission. The safety judgment on the extended memory footprint $\mathbf{A} \vdash \hat{\mathcal{M}} \colon \mathrm{ok}$ ensures that, when there is an active update access on an address, there is no other access on the address. Therefore, we can ensure that the transition updates only the expected part of the heap memory in the expected way and does not affect other unrelated memory cells. Note especially that the swap operation does not change the logic variable summary and the extended memory footprint.

Manipulation of Mutable References and Logic Variables. When a mutable reference is released, weakened (to an immutable reference), or subdivided by the transition, logic variables in the resolutive configuration and the abstract configuration are updated.

When a mutable reference modeled as a pattern $\langle p, x\rangle$ is released or weakened, the logic variable x is resolved into the pattern p. The lender of the target object of the mutable reference, which is still frozen under some lifetime in the type system, retrieves frozen update access to the object through extraction judgments. The safety on the logic variable summary provides an important guarantee: for each logic variable x concerned, there exist exactly one giver $\uparrow_\alpha x\,[*a\colon T]$ and one taker $\downarrow^\beta x\,[*a'\colon T']$, which agree on the address and types ($a = a'$ and $T \le T' \,\wedge\, T' \le T$).

When a mutable reference is subdivided, the situation is a bit more involved. For example, when we perform match on a mutable reference to a variant $\langle \mathsf{inj}_i\, p, x\rangle$, x is resolved into $\mathsf{inj}_i\, x_!$ with a newly taken logic variable $x_!$, and we get a new mutable reference $\langle p, x_!\rangle$.

We can check that, after each type of manipulation on mutable references and logic variables, $\Pi \vdash \mathbf{C}' \triangleright^{\mathrm{ok}} \mathcal{K}'$ holds.

Retyping. The retyping instruction x as T' can change the type of an active data variable x from the original type T to a new type T', if $\mathbf{A}_* \vdash T \le T'$ holds under the local lifetime context $\mathbf{A}_*$. By induction over the type T and the memory extraction for x, we can prove the following properties, under the global lifetime context $\mathbf{A}$, which extends the local lifetime context $\mathbf{A}_*$. (i) Every update on the extended memory footprint has the following form: an item $a[\mathrm{read}_\alpha]$ turns into $a[\mathrm{read}_\beta]$ for β satisfying $\beta \le_{\mathbf{A}} \alpha$ under the (global) lifetime context $\mathbf{A}$. (ii) Every update on the logic variable summary has the following form: an item $\uparrow_\alpha x\,[*a\colon T]$ turns into $\uparrow_\beta x\,[*a\colon T']$ for β and T satisfying $\beta \le_{\mathbf{A}} \alpha$ and $\mathbf{A}\colon T \le T',\, T' \le T$. Importantly, the type information in the logic variable summary remains unchanged up to *type equivalence*, because the mutable reference type $\mathsf{mut}_\alpha\, T$ is invariant over the body type T.

Elimination of a Local Lifetime Variable. When a local lifetime α is eliminated with the instruction now α, all the frozen variables in the data context tagged with $\dagger\alpha$ get reactivated. The type system ensures that there remains no reference associated with the lifetime α. Therefore, the extended memory footprint has no item of form $a[\mathrm{read}_\alpha]$, which ensures the safety condition on the extended memory footprint after the lifetime elimination. □

Using this bisimulation, we can show preservation of the safety condition on concrete configurations (although this is not directly linked to the proof of Theorem 1).

Corollary 5 (Safety on a Concrete Configuration is Preserved by Transition). *For any Π, $\mathbf{C}$ and $\mathbf{C}'$, if $\Pi \vdash \mathbf{C}\colon \mathrm{ok}$ and $\Pi \vdash \mathbf{C} \to \mathbf{C}'$ hold, then $\Pi \vdash \mathbf{C}'\colon \mathrm{ok}$ holds.*

Proof. It follows from Theorem 4, because the judgment $\Pi \vdash \mathbf{C}\colon \mathrm{ok}$ is equivalent to $\exists \mathcal{K}.\ \Pi \vdash \mathbf{C} \triangleright^{\mathrm{ok}} \mathcal{K}$. □

Before completing the proof of Theorem 1, we show a few simple lemmas.

Lemma 6 (Equivalence Between the Basic and Extended Extraction-Examination Judgments). *For any simple function f in a program Π, for any concrete configuration $\mathbf{C}$ of form $[f, L]\ \mathbf{F} \mid \mathbf{H}$, satisfying either $L = \mathsf{entry}$ or $\vdash \mathbf{C}\colon \mathrm{end}$, the following equivalence holds.*

$$\Pi \vdash \mathbf{C} \blacktriangleright^{\mathrm{ok}} \check{C} \iff \Pi \vdash \mathbf{C} \triangleright^{\mathrm{ok}} \check{C}$$

Proof. It can be proved by straightforward induction. □

Lemma 7 (Uniqueness on the Basic Extraction-Examination Judgments). *For any Π and $\mathbf{C}$, there exists at most one $\check{C}$ such that $\Pi \vdash \mathbf{C} \blacktriangleright^{\mathrm{ok}} \check{C}$ holds.*

Proof. Clear from the definition. □

Lemma 8 (Construction of a Concrete Configuration from a Weak Abstract Configuration). *For any program Π and any weak abstract configuration $\check{\mathcal{C}} = [f, L]\, \check{\mathcal{F}}$, if the function f is simple, the label L is* entry *or a label associated with a* return *statement, and $\check{\mathcal{F}}$ maps each variable to a value of the suitable sort, then there exists a concrete configuration* $\mathbf{C}$ *such that $\Pi \vdash \mathbf{C} \blacktriangleright^{\mathsf{ok}} \check{\mathcal{C}}$ holds.*

Proof. By straightforward construction. □

Now we complete the proof of Theorem 1, the soundness and completeness of our reduction.

Proof of Theorem 1. We show each direction of the implication.

Necessity ($f^{\mathrm{OS}}_{\Pi}(\vec{v}, w)$ implies $\mathbf{M}^{\mathsf{least}}_{(|\Pi|)}(f_{\mathsf{entry}})(\vec{v}, w)$). There exists a sequence of concrete configurations $\mathbf{C}_0, \dots, \mathbf{C}_n$ such that the following judgments hold.

$$\Pi \vdash \mathbf{C}_0 \to \cdots \to \mathbf{C}_n \colon \mathsf{end} \qquad \Pi \vdash \mathbf{C}_0 \blacktriangleright^{\mathsf{ok}} [f, \mathsf{entry}]\, \{\overrightarrow{(x, v)}\} \qquad \Pi \vdash \mathbf{C}_n \blacktriangleright^{\mathsf{ok}} [f, L]\, \{(y, w)\}$$

By Lemma 6, by setting $\mathcal{K}_0 = f_{\mathsf{entry}}(\vec{v}, z) \mid z$ and $\mathcal{K}' = f_L(w, z) \mid z$, the following judgments hold.

$$\Pi \vdash \mathbf{C}_0 \triangleright^{\mathsf{ok}} \mathcal{K}_0 \qquad \Pi \vdash \mathbf{C}_n \triangleright^{\mathsf{ok}} \mathcal{K}'$$

By Theorem 4, we have a sequence of resolutive configuration $\mathcal{K}_0, \dots, \mathcal{K}_n$ such that the following judgments hold.

$$(|\Pi|) \vdash \mathcal{K}_0 \to \cdots \to \mathcal{K}_n \qquad \Pi \vdash \mathbf{C}_n \triangleright^{\mathsf{ok}} \mathcal{K}_n$$

By Lemma 6 and Lemma 7, we have $\mathcal{K}_n = \mathcal{K}'$. Because $\Pi \vdash \mathbf{C}_n \colon \mathsf{end}$ holds, in the CHC representation $(|\Pi|)$ there is only one CHC whose head has f_L and the CHC has the form $f_L(y, y) \impliedby \top$. Thus $\Pi \vdash \mathcal{K}' \to \epsilon \mid w$ holds. Therefore, by Lemma 2, we have $\mathbf{M}^{\mathsf{least}}_{(|\Pi|)}(f)(\vec{v}, w)$.

Sufficiency ($\mathbf{M}^{\mathsf{least}}_{(|\Pi|)}(f_{\mathsf{entry}})(\vec{v}, w)$ implies $f^{\mathrm{OS}}_{\Pi}(\vec{v}, w)$). By Lemma 2, there exists a sequence of resolutive configurations $\mathcal{K}_0, \dots, \mathcal{K}_{n+1}$ such that the following properties hold.

$$(|\Pi|) \vdash \mathcal{K}_0 \to \cdots \to \mathcal{K}_{n+1} \qquad \mathcal{K}_0 = f_{\mathsf{entry}}(\vec{v}, z) \mid z \qquad \mathcal{K}_{n+1} = \epsilon \mid w$$

By the definition of the CHC representation $(|\Pi|)$, we can find that $\mathcal{K}_n$ is of the form $f_L(w, z) \mid z$, where the statement at the label L is a return statement. By Lemma 8, we can construct a concrete configuration $\mathbf{C}_0$ such that $\Pi \vdash \mathbf{C}_0 \triangleright^{\mathsf{ok}} \mathcal{K}_0$ holds. By Theorem 4, we also have a sequence of concrete configurations $\mathbf{C}_1, \dots, \mathbf{C}_n$ such that the following judgments hold.

$$\Pi \vdash \mathbf{C}_0 \to \mathbf{C}_1 \to \cdots \to \mathbf{C}_n \colon \mathsf{end} \qquad \Pi \vdash \mathbf{C}_n \triangleright^{\mathsf{ok}} \mathcal{K}_n$$

Thus we also have the following judgments.

$$\Pi \vdash \mathbf{C}_0 \blacktriangleright^{\mathsf{ok}} \{\overrightarrow{(x, v)}\} \qquad \Pi \vdash \mathbf{C}_n \blacktriangleright^{\mathsf{ok}} \{(y, w)\}$$

Therefore, $f^{\mathrm{OS}}_{\Pi}(\vec{v}, w)$ holds.

□

5 IMPLEMENTATION AND EVALUATION

We implemented a verification tool for Rust programs based on our reduction, *RustHorn*, and conducted preliminary evaluation experiments with small benchmarks, where we successfully confirmed the effectiveness of our approach. In this section we report on that.

5.1 Implementation of RustHorn

We implemented a prototype CHC-based verification tool *RustHorn*, which reduces Rust programs to CHCs by our method proposed in this paper. It is available at https://github.com/hopv/rust-horn.

It is written in Rust by ~2,500 lines of code. The tool supports the core features of Rust, including recursions and recursive types.

RustHorn analyzes the *MIR (Mid-level Intermediate Representation)* [54] of a Rust program, which is provided by the Rust compiler, and then generates CHCs by applying our reduction. The use of MIR enables our tool to support a broad range of Rust programs, with various kinds of syntax sugar. (An obstacle is that the implementation depends on a nightly version of the Rust compiler because the Rust compiler's internal representation is unstable.) RustHorn relies on the Rust compiler's borrow check and simply ignores the lifetime information, which is valid thanks to the nature of our method.

We briefly explain here MIR and RustHorn's algorithm. MIR models each Rust function as a set of simple instructions (called a *statement*) labeled with program points, like our calculus COR. In MIR, some sequentially executed instructions are packed into what is called a *basic block*. For efficiency, RustHorn introduces a predicate variable *for each basic block* rather than for each program point. RustHorn analyzes the set of local variables at the head of each basic block. Then for each basic block, it models the initial environment by symbolic values and performs a kind of *symbolic execution* to analyze the final environment of the block. In particular, in a MIR statement, we can directly access a (possibly deep) substructure of a local variable (called a *place*), which is not supported in COR but can easily be modeled in our reduction. Depending on the action taken at the end of the block (which is expressed by what is called a *terminator*), it adds some CHCs to the output. Before we jump to another basic block or return from the function, we clean up the local variables that will not be used any more and add the *equality constraint on the final target value of each mutable reference* contained in the variables.

This algorithm performs a more advanced reduction than our formalized reduction presented in §3. We believe that our soundness and completeness proof presented in §2 and §4 justifies the core idea of this advanced reduction, but direct proof on the advanced reduction remains to be a challenge.

5.2 Benchmarks and Experiments

In order to measure the performance of RustHorn and the existing CHC-based verifier for C, SeaHorn [25], we conducted preliminary experiments using the benchmarks listed in Table 1. Each benchmark program has one assertion to be verified, and is provided both in Rust and C. Most benchmark instances consist of a pair of *safe* and an *unsafe* programs that only differ from each other in the asserted property. We also wrote LOC (in Rust, skipping blank and comment lines) of each benchmark in the table. The column *Loop?* shows whether the verified program has loops (which include recursions, exclude vacuous loops like `loop {}`). The column *Mut?* shows whether the verified program uses mutable references. The benchmarks and experimental results are available at https://doi.org/10.5281/zenodo.4710723.

We conducted experiments on a commodity laptop (2.6GHz Intel Core i7 MacBook Pro with 16GB RAM). First we reduced each benchmark program into CHCs in the SMT-LIB 2 format using RustHorn and SeaHorn (version 10.0.0-rc0-86a31cf1) [25]. The time for the reduction was quite short (at most ~0.3 second for each program). After that, we measured the time of CHC solving by Spacer [42] in Z3 (version 4.8.10) [75] and HoIce (version 1.8.3)[14] [12, 13] for the generated CHCs. HoIce does not accept SeaHorn's outputs, because SeaHorn uses a different format and employs arrays for pointers. We have also prepared modified versions of some of the CHCs generated by SeaHorn, obtained by adding constraints on *address freshness* to improve accuracy of the model and reduce false alarms. Still, we could not make the modified versions for the benchmarks in the

[14] We used Z3 version 4.7.1 for the backend *SMT* solver of HoIce, in order to deal well with recursive data types.

						RustHorn		SeaHorn *w/Spacer*	
Group	*Instance*	*Safe?*	*LOC*	*Loop?*	*Mut?*	*w/Spacer*	*w/HoIce*	*as is*	*modified*
simple	01	safe	12	yes	no	<0.1	0.1	<0.1	
	04-recursive	safe	14	yes	no	0.5	timeout	0.8	
	05-recursive	unsafe	26	yes	no	<0.1	<0.1	<0.1	
	06-loop	safe	10	yes	no	timeout	0.1	timeout	
	hhk2008	safe	20	yes	no	timeout	47.9	<0.1	
	unique-scalar	unsafe	9	no	yes	<0.1	0.3	<0.1	
bmc	1	safe	46	no	no	0.2	<0.1	<0.1	
		unsafe				0.2	<0.1	<0.1	
	2	safe	15	yes	no	timeout	0.1	<0.1	
		unsafe				<0.1	0.1	<0.1	
	3	safe	35	yes	no	0.1	<0.1	<0.1	
		unsafe				<0.1	<0.1	<0.1	
	diamond-1	safe	55	no	no	0.1	<0.1	<0.1	
		unsafe				<0.1	<0.1	<0.1	
	diamond-2	safe	40	no	no	0.2	<0.1	<0.1	
		unsafe				0.1	<0.1	<0.1	
prusti	ackermann	safe	17	yes	no	<0.1	0.1	<0.1	
	ackermann-same	safe	26	yes	no	timeout	timeout	timeout	
	compress	safe	12	no	yes	<0.1	0.1	false alarm	
	borrows-align	safe	14	no	yes	<0.1	0.1	<0.1	
	account	safe	18	no	yes	<0.1	0.1	<0.1	
		unsafe				<0.1	0.2	<0.1	
	restore	safe	14	no	yes	<0.1	0.3	false alarm	
inc-max	base	safe	15	no	yes	<0.1	0.2	false alarm	<0.1
		unsafe				<0.1	0.2	<0.1	<0.1
	base3	safe	22	no	yes	<0.1	0.2	false alarm	
		unsafe				<0.1	0.2	<0.1	
	repeat	safe	22	yes	yes	0.1	timeout	false alarm	timeout
		unsafe				<0.1	0.5	<0.1	<0.1
	repeat3	safe	29	yes	yes	0.2	timeout	false alarm	
		unsafe				<0.1	1.4	<0.1	
swap-dec	base	safe	23	yes	yes	0.1	0.5	false alarm	<0.1
		unsafe				0.1	timeout	<0.1	0.1
	base3	safe	27	yes	yes	0.1	timeout	false alarm	<0.1
		unsafe				0.4	14.2	<0.1	0.1
	exact	safe	24	yes	yes	0.1	0.6	false alarm	0.2
		unsafe				<0.1	timeout	<0.1	<0.1
	exact3	safe	30	yes	yes	timeout	timeout	false alarm	0.2
		unsafe				0.1	2.2	<0.1	<0.1
swap2-dec	base	safe	24	yes	yes	0.4	0.8	false alarm	<0.1
		unsafe				1.2	timeout	<0.1	0.1
	base3	safe	32	yes	yes	1.8	timeout	false alarm	<0.1
		unsafe				67.0	39.0	<0.1	0.2
	exact	safe	25	yes	yes	1.0	1.0	false alarm	0.2
		unsafe				<0.1	timeout	<0.1	<0.1
	exact3	safe	35	yes	yes	timeout	timeout	false alarm	1.3
		unsafe				<0.1	6.2	<0.1	<0.1
just-rec	base	safe	14	yes	yes	<0.1	0.2	<0.1	
		unsafe				<0.1	0.2	<0.1	
linger-dec	base	safe	15	yes	yes	<0.1	0.2	false alarm	
		unsafe				<0.1	0.2	<0.1	
	base3	safe	29	yes	yes	<0.1	0.3	false alarm	
		unsafe				<0.1	31.9	<0.1	
	exact	safe	16	yes	yes	<0.1	0.3	false alarm	
		unsafe				<0.1	0.3	<0.1	
	exact3	safe	30	yes	yes	<0.1	0.4	false alarm	
		unsafe				<0.1	1.3	<0.1	
lists	append	safe	27	yes	yes	tool error	0.3	false alarm	
		unsafe				tool error	0.3	0.1	
	inc-all	safe	35	yes	yes	tool error	0.3	false alarm	
		unsafe				tool error	0.4	<0.1	
	inc-some	safe	32	yes	yes	tool error	0.3	timeout	
		unsafe				tool error	0.6	0.1	
	inc-some2	safe	34	yes	yes	tool error	timeout	timeout	
		unsafe				tool error	0.7	0.3	
trees	append	safe	35	yes	yes	tool error	0.4	false alarm	
		unsafe				tool error	0.3	<0.1	
	inc-all	safe	36	yes	yes	tool error	timeout	timeout	
		unsafe				tool error	0.2	0.1	
	inc-some	safe	34	yes	yes	tool error	0.4	timeout	
		unsafe				tool error	0.7	0.1	
	inc-some2	safe	36	yes	yes	tool error	tool error	timeout	
		unsafe				tool error	0.9	timeout	

Table 1. Benchmarks and experimental results on RustHorn and SeaHorn, with Spacer and HoIce.

groups linger-dec, lists and trees because address freshness check is quite hard to model. For inc-max/base3 and inc-max/repeat3 we could not make the modified versions because SeaHorn wrongly omitted all the memory manipulation in the CHC outputs for them, probably by inaccurate pointer analyses.

Below we explain our benchmarks more in detail.

The benchmarks in the groups simple and bmc were taken from those of SeaHorn (https://github.com/seahorn/seahorn/tree/master/test). They are originally provided in C and the Rust versions were written by us. The benchmarks in SeaHorn were chosen based upon the following criteria: they (i) consist only of features supported by the core of Rust (covered by RustHorn), (ii) follow Rust's permission discipline, and (iii) are small enough to be amenable for manual translation from C to Rust. We omitted SeaHorn tests that use arrays (such as simple/02_array) and function pointers (such as devirt/devirt_02), in light of (i). For an example of (ii), the following SeaHorn test dsa/test-1 was not included, because here the two pointers a and b can simultaneously point at the same object y with update permission.

```
void f(int* x, int* y){ *x = 1; *y = 2; }
void g(int* p, int* q, int* r, int* s) { f(p, q); f(r, s); }
int main(){
  int x, y, w, z; int* a = &x; int* b = &y; if (nd()) a = b;
  g(a, b, &w, &z); return x + y + w + z;
}
```

Also, in light of (iii), we omitted large SeaHorn tests, such as bmc/cdaudio_simpl1. The following benchmark simple/hhk2008 is a SeaHorn test that was adopted in our experiments. The key challenge of this program is to find out an invariant on the while loop.

```
int main() {
  int a = rand(), b = rand();
  if (!(a <= 1000000 && 0 <= b && b <= 1000000)) return 0;
  int res = a, cnt = b;
  while (cnt > 0) { cnt = cnt - 1; res = res + 1; }
  assert(res == a + b); return 0;
}
```

The benchmarks in the group prusti were taken from tests of Prusti [3], a semi-automated verification tool for Rust (available at https://github.com/viperproject/prusti-dev). We chose several small, interesting benchmarks from Prusti's tests. For example, restore features a mutable reference to a randomly chosen object.

```
struct T { val: i32 }
fn main() {
  let mut x = T { val: 11 };  let mut y = T { val: 22 };
  let z = if rand() { &mut x } else { &mut y };
  z.val += 33;  x.val += 44;  y.val += 44;
  assert!(x.val == 88 || x.val == 55);
  assert!(y.val == 66 || y.val == 99);
}
```

The benchmarks in the remaining seven groups were made by us featuring various use cases of mutable references. The benchmarks in the groups `inc-max`, `just-rec` and `linger-dec` are based on the examples in §1 and §3.4.

The group swap-dec consists of benchmark programs that perform repeated and involved updates via *mutable references to a mutable reference to an integer*. For example, below is the safe program of the instance swap-dec/base, with some details modified for readability. The verified property is that the function `test` always returns **`true`** whenever it terminates.

```
fn may_swap<T>(mx: &mut T, my: &mut T) {
  if rand() { swap(mx, my); }
}
fn swap_dec<'a>(mma: &mut &'a mut i32, mmb: &mut &'a mut i32) {
  may_swap(mma, mmb); if rand() { return; }
  **mma -= 1; **mmb -= 2; swap_dec(mma, mmb);
}
fn test(mut a: i32, mut b: i32) {
  let a0 = a; let mut ma = &mut a; let mut mb = &mut b;
  swap_dec(&mut ma, &mut mb); assert(a0 >= a);
}
```

The group swap2-dec is an advanced variant of swap-dec. It features a *mutable reference to a mutable reference to a mutable reference to an integer*. For example, the safe program of swap2-dec/base is as follows (may_swap is the same as above).

```
fn swap2_dec<'a>(mmma: &mut &'a mut &'a mut i32,
                 mmmb: &mut &'a mut &'a mut i32) {
  may_swap(mmma, mmmb); may_swap(*mmma, *mmmb);
  if rand() { return; }
  ***mmma -= 1; ***mmmb -= 2; swap2_dec(mmma, mmmb);
}
fn test(mut a: i32, mut b: i32) {
  let a0 = a; let mut ma = &mut a; let mut mb = &mut b;
  let mut mma = &mut ma; let mut mmb = &mut mb;
  swap2_dec(&mut mma, &mut mmb); assert(a0 >= a);
}
```

The instances labeled `repeat` in the group `inc-max` repeat the operation of the function `inc_max` `n` times, for some random number `n`. The instances labeled `exact` in the groups swap-dec, swap2-dec and `linger-dec` not only observe the decrease but also check the amount of the decrease. For example, in the safe program of swap-dec/exact, the last check is `a0 >= a && a0 - a <= 2 * n`, which has the condition `a0 - a <= 2 * n` unlike swap-dec/base.

The groups `lists` and `trees` feature destructive updates on recursive data structures (singly linked lists and binary trees) via mutable references. The instance `lists/inc-some` has appeared as Example 5. The safe program of the instance `lists/append` is as follows.

```
enum List<T> { Cons(T, Box<List<T>>), Nil } use List::*;
fn append(mla: &mut List<i32>, lb: List<i32>) { match mla {
    Cons(_, mla2) => { append(mla2, lb); }
    Nil => { *mla = lb; }
} }
fn sum(la: &List<i32>) -> i32 {
  match la { Cons(a, la2) => a + sum(la2), Nil => 0 }
}
fn test(la: List<i32>, lb: List<i32>) {
  let m = sum(&la); let n = sum(&lb);
  append(&mut la, lb); let r = sum(&la); assert(r == m + n);
}
```

The safe program of the instance `lists/inc-all` is as follows (`List` and `sum` are the same as above).

```
fn inc_all<'a>(mla: &'a mut List<i32>) { match mla {
    Cons(ma, mla2) => { *ma += 1; inc_all(mla2); },  Nil => {}
} }
fn length(la: &List<i32>) -> i32 {
  match la { Cons(a, la2) => 1 + length(la2), Nil => 0 }
}
fn test(mut la: List<i32>) {
  let n = sum(&la); let l = length(&la);
  inc_all(&mut la); let r = sum(&la); assert(r == n + l);
}
```

The instance `lists/inc-some2` is an advanced variant of `inc-some` of Example 5. Its program with the safe property is presented below (`List` and `sum` are the same as above). The function `test` takes mutable references to some two elements of the input list and performs increment on them.

```
fn take_some_rest<'a>(mla: &'a mut List<i32>) ->
  (&'a mut i32, &'a mut List<i32>) { match mla {
    Cons(ma, mla2) => { if rand() { (ma, mla2) }
                        else { take_some_rest(mla2) } }
    Nil => take_some_rest(mla)
} }
fn test(mut la: List<i32>) {
  let n = sum(&la); let (mb, mla2) = take_some_rest(&mut la);
  let (mc, _) = take_some_rest(mla2);
  *mb += 1; *mc += 1; let r = sum(&la); assert(r == n + 2);
}
```

Benchmarks in the group `trees` are analogous to those in the group `lists` but designed for binary trees instead of lists.

5.3 Experimental Results

Table 1 shows the results of the experiments. The columns for RustHorn and SeaHorn show the time for verification (in seconds) in the case the verification was successful. In the case the verification failed, the columns show one of the following failure labels. The label timeout means timeout over the time limit of 180 seconds. The label false alarm means a report of unsafety for a safe program. The label tool error means an error of the backend CHC solver; Spacer was unstable for recursive types in general and HoIce was unstable in some situations.

RustHorn, combined with either Spacer or HoIce, successfully verified all programs that were successfully verified by SeaHorn (without our modification). Also, RustHorn successfully verified various interesting programs that SeaHorn could not verify. The verification time of RustHorn largely matched that of SeaHorn. Although the benchmark set used for the experiments is small and more or less contrived, we believe that the experimental results already indicate effectiveness of our verification method. Experiments on larger, more realistic benchmark programs are left to future work.

The combination of RustHorn and HoIce succeeded in verifying many programs with *recursive data types* (`lists` and `trees`), including `lists/inc-some` and `trees/inc-some`. Still, it failed at some benchmarks such as `lists/inc-some2` and `trees/inc-all`. This is because HoIce, unlike Spacer, can find models defined with primitive recursive functions for recursive data types, as discussed in Example 5 in §3.4.

SeaHorn, without our modification, issued *false alarms* for many programs in the last seven benchmark groups, from `inc-max` to `trees`. This is due to SeaHorn's imprecise modeling of pointers and memory states, where *freshness of pointer addresses* is not fully specified. For our modified CHC outputs of SeaHorn, false alarms were not observed but verification timed out for one benchmark (although one timeout was observed), but we could not make modified versions for many of the benchmarks. For the last four groups from `just-rec` to `trees`, unboundedly many memory cells can be allocated, which imposes a fundamental challenge for the *array-based reduction* as discussed in §1.1. SeaHorn succeeded in verification for `just-rec` by analyzing absence of effective destructive updates and generating CHCs *without arrays*, but for all other benchmarks of the safe property SeaHorn failed because of imprecise representation. RustHorn succeeded in verifying most benchmark programs in these groups.

For the benchmarks in the groups `swap-dec` and `swap2-dec`, RustHorn performed somewhat inefficiently compared to SeaHorn with our modification. This is presumably because the benchmarks in the groups feature *nested mutable references*, which are modeled as a big value in our reduction. The group `swap-dec` features a mutable reference to a mutable reference to an integer, which is modeled as a pair of pairs of integers, which has four integers in total. The group `swap2-dec` features a mutable reference to a mutable reference to a mutable reference to an integer (threefold!), which is modeled as a value that has eight integers in total. For SeaHorn, the benchmarks in `swap-dec` and `swap2-dec` are quite easy because only a limited number of addresses (up to six addresses) are used in each program. This indicates a disadvantage of our approach compared to the array-based approach. Still, we believe that in real-world Rust programs we do not use such nested mutable references so often.

6 RELATED WORK

CHC-based Verification of Pointer-Manipulating Programs. SeaHorn [25] is a representative existing tool for CHC-based verification of pointer-manipulating programs. It basically represents the heap memory as an array. Although some pointer analyses [26] are used to optimize the array representation of the heap memory, their approach suffers from some pointer use cases that our approach can handle, as is examined by our experiments reported in §5. Still, their approach is significant in the context of automated verification, given that many real-world pointer-manipulating programs do not fit within Rust's permission control.

Another approach is taken by JayHorn [38, 39], which automatically verifies Java programs (possibly using object pointers) by reduction to CHCs. It represents store invariants using special predicates *pull* and *push*. Although this allows faster reasoning about the heap memory than the array-based approach, it can suffer from more false alarms. We conducted a small experiment for JayHorn (0.6-alpha) on some of the benchmarks of §5.2. JayHorn reported '`UNKNOWN`' (instead of

'SAFE' or 'UNSAFE') for even simple programs such as the programs of the instance `unique-scalar` in `simple` and the instance `basic` in `inc-max`.

Verification for Rust. Whereas we have presented the first CHC-based (fully automated) verification method specially designed for Rust, there are a number of studies on other types of verification for Rust.

RustBelt by Jung et al. [34] formally verified *safety* properties for Rust libraries with *unsafe* internal implementation, using manual reasoning in the higher-order separation logic Iris [35, 37], based upon higher-order concurrent separation logic, built on the Coq Proof Assistant [16]. They presented a formalized core of Rust, λ_{Rust}, which affected the language design of our calculus COR. Thanks to the power of Iris, their verification method is highly extensible for various Rust libraries with unsafe code, including those with interior mutability. Still, they verify only the safety property and do not cover functional correctness. Also, the automation of the verification in their approach is not well discussed.

Ullrich [72] translated a subset of Rust into a purely functional programming language to manually verify functional correctness of some tricky Rust programs using the Lean Theorem Prover [17]. Although this method eliminates pointers to get simple models like our approach, the applicability of this method is quite limited, because it deals with mutable references by simple static tracking of addresses based on lenses [22]. This method thus does not support even basic use cases such as dynamic selection of mutable references (e.g., `take_max` in §1.2) [71], which can be easily handled by our method. On the other hand, our approach covers arbitrary pointer operations supported in the safe core of Rust, as discussed in §3.

There are a series of studies [3, 19, 29] of methods of semi-automated verification of Rust programs using Viper [52], a verification platform based on separation logic with *fractional permission*. This approach can deal with advanced features such as unsafe code [29] and type traits [19] to some extent. In particular, Prusti by Astrauskas et al. [3] conducted semi-automated verification (manually providing pre/post-conditions and loop invariants) on many realistic examples. Also, they use special machinery called a pledge to model mutable borrows, which enables operations like `Vec::index_mut`. Still, this approach does not support some basic operations on mutable references, such as *split of mutable references*, unlike our RustHorn. We suppose that Viper's reasoning based on fractional permission does not naturally match Rust's lifetime-based permission control. On the other hand, our reduction of RustHorn is specially designed for Rust. As we discussed in §3.5, our reduction of Rust programs to CHCs can be applied to semi-automated verification where users can declare pre/post-conditions and loop invariants. This extension of our approach can work more nicely than their Viper-based approaches for a wide class of Rust programs.

There are also approaches based on bounded model checking [4, 46, 70] for verification of Rust programs with unsafe code. Our reduction can be applied to bounded model checking as discussed in §3.5.

Verification using Permission/Ownership. The notion of permission/ownership has been applied to a wide range of verification. It has been used for detecting race conditions in concurrent programs [8, 69] and analyzing the safety of memory allocation [68]. Separation logic based on permission is also studied well [7, 37, 52]. A simple notion of permission has also been used in some verification tools [5, 15, 23]. However, existing studies on permission-based verification are mostly based on fractional or counting permission, which is quite different from Rust's permission control.

Prophecy Variables. Our idea of considering a future value to represent a mutable reference is related to the notion of *prophecy variables* [1, 36, 73]. In particular, Jung et al. [36] presented a new program logic that supports prophecy variables on the separation logic Iris [37].

7 CONCLUSION

We proposed a novel method for CHC-based automated verification of Rust programs. The key idea is to model a mutable reference as a pair of the current target value and the target value *at the end of the borrow*. We formalized the method for a core language of Rust and proved its soundness and completeness. We implemented a prototype verification tool for a subset of Rust and confirmed the effectiveness of our approach through an experiment.

ACKNOWLEDGMENTS

This work was supported by JSPS KAKENHI Grant Numbers JP15H05706, JP16K16004, JP20H05703 and JP21J20459. We are grateful to the anonymous reviewers for insightful and helpful comments.

REFERENCES

[1] Martín Abadi and Leslie Lamport. 1991. The Existence of Refinement Mappings. *Theor. Comput. Sci.* 82, 2 (1991), 253–284. https://doi.org/10.1016/0304-3975(91)90224-P

[2] Francesco Alberti, Roberto Bruttomesso, Silvio Ghilardi, Silvio Ranise, and Natasha Sharygina. 2012. Lazy Abstraction with Interpolants for Arrays. In *Logic for Programming, Artificial Intelligence, and Reasoning - 18th International Conference, LPAR-18, Mérida, Venezuela, March 11-15, 2012. Proceedings (Lecture Notes in Computer Science, Vol. 7180)*, Nikolaj Bjørner and Andrei Voronkov (Eds.). Springer, 46–61. https://doi.org/10.1007/978-3-642-28717-6_7

[3] Vytautas Astrauskas, Peter Müller, Federico Poli, and Alexander J. Summers. 2018. Leveraging Rust Types for Modular Specification and Verification. (2018). https://doi.org/10.3929/ethz-b-000311092

[4] Marek S. Baranowski, Shaobo He, and Zvonimir Rakamaric. 2018. Verifying Rust Programs with SMACK, See [45], 528–535. https://doi.org/10.1007/978-3-030-01090-4_32

[5] Mike Barnett, Manuel Fähndrich, K. Rustan M. Leino, Peter Müller, Wolfram Schulte, and Herman Venter. 2011. Specification and Verification: The Spec# Experience. *Commun. ACM* 54, 6 (2011), 81–91. https://doi.org/10.1145/1953122.1953145

[6] Nikolaj Bjørner, Arie Gurfinkel, Kenneth L. McMillan, and Andrey Rybalchenko. 2015. Horn Clause Solvers for Program Verification. In *Fields of Logic and Computation II - Essays Dedicated to Yuri Gurevich on the Occasion of His 75th Birthday (Lecture Notes in Computer Science, Vol. 9300)*, Lev D. Beklemishev, Andreas Blass, Nachum Dershowitz, Bernd Finkbeiner, and Wolfram Schulte (Eds.). Springer, 24–51. https://doi.org/10.1007/978-3-319-23534-9_2

[7] Richard Bornat, Cristiano Calcagno, Peter W. O'Hearn, and Matthew J. Parkinson. 2005. Permission Accounting in Separation Logic. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, Jens Palsberg and Martín Abadi (Eds.). ACM, 259–270. https://doi.org/10.1145/1040305.1040327

[8] Chandrasekhar Boyapati, Robert Lee, and Martin C. Rinard. 2002. Ownership Types for Safe Programming: Preventing Data Races and Deadlocks. In *Proceedings of the 2002 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2002, Seattle, Washington, USA, November 4-8, 2002.*, Mamdouh Ibrahim and Satoshi Matsuoka (Eds.). ACM, 211–230. https://doi.org/10.1145/582419.582440

[9] John Boyland. 2003. Checking Interference with Fractional Permissions. In *Static Analysis, 10th International Symposium, SAS 2003, San Diego, CA, USA, June 11-13, 2003, Proceedings (Lecture Notes in Computer Science, Vol. 2694)*, Radhia Cousot (Ed.). Springer, 55–72. https://doi.org/10.1007/3-540-44898-5_4

[10] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. 2006. What's Decidable About Arrays?. In *Verification, Model Checking, and Abstract Interpretation, 7th International Conference, VMCAI 2006, Charleston, SC, USA, January 8-10, 2006, Proceedings (Lecture Notes in Computer Science, Vol. 3855)*, E. Allen Emerson and Kedar S. Namjoshi (Eds.). Springer, 427–442. https://doi.org/10.1007/11609773_28

[11] Toby Cathcart Burn, C.-H. Luke Ong, and Steven J. Ramsay. 2018. Higher-Order Constrained Horn Clauses for Verification. *PACMPL* 2, POPL (2018), 11:1–11:28. https://doi.org/10.1145/3158099

[12] Adrien Champion, Tomoya Chiba, Naoki Kobayashi, and Ryosuke Sato. 2018. ICE-Based Refinement Type Discovery for Higher-Order Functional Programs. In *Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 10805)*, Dirk Beyer and Marieke Huisman (Eds.). Springer, 365–384. https://doi.org/10.1007/978-3-319-89960-2_20

[13] Adrien Champion, Naoki Kobayashi, and Ryosuke Sato. 2018. HoIce: An ICE-Based Non-linear Horn Clause Solver. In *Programming Languages and Systems - 16th Asian Symposium, APLAS 2018, Wellington, New Zealand, December 2-6, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 11275)*, Sukyoung Ryu (Ed.). Springer, 146–156. https:

//doi.org/10.1007/978-3-030-02768-1_8

[14] David G. Clarke, John Potter, and James Noble. 1998. Ownership Types for Flexible Alias Protection. In *Proceedings of the 1998 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA '98), Vancouver, British Columbia, Canada, October 18-22, 1998.*, Bjørn N. Freeman-Benson and Craig Chambers (Eds.). ACM, 48–64. https://doi.org/10.1145/286936.286947

[15] Ernie Cohen, Markus Dahlweid, Mark A. Hillebrand, Dirk Leinenbach, Michal Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. 2009. VCC: A Practical System for Verifying Concurrent C. In *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5674)*, Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel (Eds.). Springer, 23–42. https://doi.org/10.1007/978-3-642-03359-9_2

[16] Coq Team. 2021. The Coq Proof Assistant. https://coq.inria.fr/

[17] Leonardo Mendonça de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. 2015. The Lean Theorem Prover (System Description). In *Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings (Lecture Notes in Computer Science, Vol. 9195)*, Amy P. Felty and Aart Middeldorp (Eds.). Springer, 378–388. https://doi.org/10.1007/978-3-319-21401-6_26

[18] Dropbox. 2020. *Rewriting the Heart of Our Sync Engine - Dropbox.* https://dropbox.tech/infrastructure/rewriting-the-heart-of-our-sync-engine

[19] Matthias Erdin. 2019. *Verification of Rust Generics, Typestates, and Traits.* Master's thesis. ETH Zürich.

[20] Grigory Fedyukovich, Samuel J. Kaufman, and Rastislav Bodík. 2017. Sampling Invariants from Frequency Distributions. In *2017 Formal Methods in Computer Aided Design, FMCAD 2017, Vienna, Austria, October 2-6, 2017*, Daryl Stewart and Georg Weissenbacher (Eds.). IEEE, 100–107. https://doi.org/10.23919/FMCAD.2017.8102247

[21] Grigory Fedyukovich, Sumanth Prabhu, Kumar Madhukar, and Aarti Gupta. 2019. Quantified Invariants via Syntax-Guided Synthesis. In *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 11561)*, Isil Dillig and Serdar Tasiran (Eds.). Springer, 259–277. https://doi.org/10.1007/978-3-030-25540-4_14

[22] J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. 2007. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Trans. Program. Lang. Syst.* 29, 3 (2007), 17. https://doi.org/10.1145/1232420.1232424

[23] Léon Gondelman. 2016. *Un système de types pragmatique pour la vérification déductive des programmes. (A Pragmatic Type System for Deductive Verification).* Ph.D. Dissertation. University of Paris-Saclay, France. https://tel.archives-ouvertes.fr/tel-01533090

[24] Sergey Grebenshchikov, Nuno P. Lopes, Corneliu Popeea, and Andrey Rybalchenko. 2012. Synthesizing Software Verifiers from Proof Rules. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, Jan Vitek, Haibo Lin, and Frank Tip (Eds.). ACM, 405–416. https://doi.org/10.1145/2254064.2254112

[25] Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A. Navas. 2015. The SeaHorn Verification Framework. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 9206)*, Daniel Kroening and Corina S. Pasareanu (Eds.). Springer, 343–361. https://doi.org/10.1007/978-3-319-21690-4_20

[26] Arie Gurfinkel and Jorge A. Navas. 2017. A Context-Sensitive Memory Model for Verification of C/C++ Programs. In *Static Analysis - 24th International Symposium, SAS 2017, New York, NY, USA, August 30 - September 1, 2017, Proceedings (Lecture Notes in Computer Science, Vol. 10422)*, Francesco Ranzato (Ed.). Springer, 148–168. https://doi.org/10.1007/978-3-319-66706-5_8

[27] Arie Gurfinkel, Sharon Shoham, and Yuri Meshman. 2016. SMT-Based Verification of Parameterized Systems. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, Thomas Zimmermann, Jane Cleland-Huang, and Zhendong Su (Eds.). ACM, 338–348. https://doi.org/10.1145/2950290.2950330

[28] Arie Gurfinkel, Sharon Shoham, and Yakir Vizel. 2018. Quantifiers on Demand, See [45], 248–266. https://doi.org/10.1007/978-3-030-01090-4_15

[29] Florian Hahn. 2016. *Rust2Viper: Building a Static Verifier for Rust.* Master's thesis. ETH Zürich. https://doi.org/10.3929/ethz-a-010669150

[30] Jochen Hoenicke, Rupak Majumdar, and Andreas Podelski. 2017. Thread Modularity at Many Levels: A Pearl in Compositional Verification. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 473–485. https://doi.org/10.1145/3009837

[31] Hossein Hojjat and Philipp Rümmer. 2018. The Eldarica Horn Solver. In *2018 Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA, October 30 - November 2, 2018*, Nikolaj Bjørner and Arie Gurfinkel (Eds.). IEEE,

1–7. https://doi.org/10.23919/FMCAD.2018.8603013

[32] Alfred Horn. 1951. On Sentences Which are True of Direct Unions of Algebras. *The Journal of Symbolic Logic* 16, 1 (1951), 14–21. http://www.jstor.org/stable/2268661

[33] Trevor Jim, J. Gregory Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. 2002. Cyclone: A Safe Dialect of C. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference, June 10-15, 2002, Monterey, California, USA*, Carla Schlatter Ellis (Ed.). USENIX, 275–288. http://www.usenix.org/publications/library/proceedings/usenix02/jim.html

[34] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2018. RustBelt: Securing the Foundations of the Rust Programming Language. *PACMPL* 2, POPL (2018), 66:1–66:34. https://doi.org/10.1145/3158154

[35] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.* 28 (2018), e20. https://doi.org/10.1017/S0956796818000151

[36] Ralf Jung, Rodolphe Lepigre, Gaurav Parthasarathy, Marianna Rapoport, Amin Timany, Derek Dreyer, and Bart Jacobs. 2020. The Future is Ours: Prophecy Variables in Separation Logic. *PACMPL* 4, POPL (2020), 45:1–45:32. https://doi.org/10.1145/3371113

[37] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, Sriram K. Rajamani and David Walker (Eds.). ACM, 637–650. https://doi.org/10.1145/2676726.2676980

[38] Temesghen Kahsai, Rody Kersten, Philipp Rümmer, and Martin Schäf. 2017. Quantified Heap Invariants for Object-Oriented Programs. In *LPAR-21, 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Maun, Botswana, May 7-12, 2017 (EPiC Series in Computing, Vol. 46)*, Thomas Eiter and David Sands (Eds.). EasyChair, 368–384.

[39] Temesghen Kahsai, Philipp Rümmer, Huascar Sanchez, and Martin Schäf. 2016. JayHorn: A Framework for Verifying Java programs. In *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 9779)*, Swarat Chaudhuri and Azadeh Farzan (Eds.). Springer, 352–358. https://doi.org/10.1007/978-3-319-41528-4_19

[40] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. 2018. Zeus: Analyzing Safety of Smart Contracts. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society.

[41] Naoki Kobayashi, Ryosuke Sato, and Hiroshi Unno. 2011. Predicate Abstraction and CEGAR for Higher-Order Model Checking. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, Mary W. Hall and David A. Padua (Eds.). ACM, 222–233. https://doi.org/10.1145/1993498.1993525

[42] Anvesh Komuravelli, Arie Gurfinkel, and Sagar Chaki. 2014. SMT-based Model Checking for Recursive Programs. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings (Lecture Notes in Computer Science, Vol. 8559)*, Armin Biere and Roderick Bloem (Eds.). Springer, 17–34. https://doi.org/10.1007/978-3-319-08867-9_2

[43] Robert A. Kowalski. 1974. Predicate Logic as Programming Language. In *Information Processing, Proceedings of the 6th IFIP Congress 1974, Stockholm, Sweden, August 5-10, 1974*, Jack L. Rosenfeld (Ed.). North-Holland, 569–574.

[44] Shuvendu K. Lahiri and Randal E. Bryant. 2004. Constructing Quantified Invariants via Predicate Abstraction. In *Verification, Model Checking, and Abstract Interpretation, 5th International Conference, VMCAI 2004, Venice, Italy, January 11-13, 2004, Proceedings (Lecture Notes in Computer Science, Vol. 2937)*, Bernhard Steffen and Giorgio Levi (Eds.). Springer, 267–281. https://doi.org/10.1007/978-3-540-24622-0_22

[45] Shuvendu K. Lahiri and Chao Wang (Eds.). 2018. *Automated Technology for Verification and Analysis - 16th International Symposium, ATVA 2018, Los Angeles, CA, USA, October 7-10, 2018, Proceedings*. Lecture Notes in Computer Science, Vol. 11138. Springer. https://doi.org/10.1007/978-3-030-01090-4

[46] Marcus Lindner, Jorge Aparicius, and Per Lindgren. 2018. No Panic! Verification of Rust Programs by Symbolic Execution. In *16th IEEE International Conference on Industrial Informatics, INDIN 2018, Porto, Portugal, July 18-20, 2018*. IEEE, 108–114. https://doi.org/10.1109/INDIN.2018.8471992

[47] Nicholas D. Matsakis and Felix S. Klock, II. 2014. The Rust Language. In *Proceedings of the 2014 ACM SIGAda annual conference on High integrity language technology, HILT 2014, Portland, Oregon, USA, October 18-21, 2014*, Michael Feldman and S. Tucker Taft (Eds.). ACM, 103–104. https://doi.org/10.1145/2663171.2663188

[48] Yusuke Matsushita. 2021. *Extensible Functional-Correctness Verification of Rust Programs by the Technique of Prophecy*. Master's thesis. University of Tokyo. http://www.kb.is.s.u-tokyo.ac.jp/~yskm24t/papers/master-thesis.pdf

[49] Yusuke Matsushita, Takeshi Tsukada, and Naoki Kobayashi. 2020. RustHorn: CHC-Based Verification for Rust Programs. In *Programming Languages and Systems - 29th European Symposium on Programming, ESOP 2020, Held as Part of the*

European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings (Lecture Notes in Computer Science, Vol. 12075), Peter Müller (Ed.). Springer, 484–514. https://doi.org/10.1007/978-3-030-44914-8_18

[50] Microsoft. 2021. Boogie: An Intermediate Verification Language. https://www.microsoft.com/en-us/research/project/boogie-an-intermediate-verification-language/

[51] Mozilla. 2021. *Rust language — Mozilla Research.* https://research.mozilla.org/rust/

[52] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. 2016. Viper: A Verification Infrastructure for Permission-Based Reasoning. In *Verification, Model Checking, and Abstract Interpretation - 17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January 17-19, 2016. Proceedings (Lecture Notes in Computer Science, Vol. 9583)*, Barbara Jobstmann and K. Rustan M. Leino (Eds.). Springer, 41–62. https://doi.org/10.1007/978-3-662-49122-5_2

[53] npm. 2019. *Rust Case Study: Community Makes Rust an Easy Choice for npm.* https://www.rust-lang.org/static/pdfs/Rust-npm-Whitepaper.pdf

[54] Rust Community. 2021. The MIR (Mid-level IR). https://rustc-dev-guide.rust-lang.org/mir/index.html

[55] Rust Community. 2021. Reference Cycles Can Leak Memory - The Rust Programming Language. https://doc.rust-lang.org/book/ch15-06-reference-cycles.html

[56] Rust Community. 2021. RFC 2025: Nested Method Calls. https://rust-lang.github.io/rfcs/2025-nested-method-calls.html

[57] Rust Community. 2021. RFC 2094: Non-lexical Lifetimes. https://rust-lang.github.io/rfcs/2094-nll.html

[58] Rust Community. 2021. Rust Programming Language. https://www.rust-lang.org/

[59] Rust Community. 2021. std::cell::RefCell - Rust. https://doc.rust-lang.org/std/cell/struct.RefCell.html

[60] Rust Community. 2021. std::collections::HashMap - Rust. https://doc.rust-lang.org/std/collections/struct.HashMap.html

[61] Rust Community. 2021. std::rc::Rc - Rust. https://doc.rust-lang.org/std/rc/struct.Rc.html

[62] Rust Community. 2021. std::sync::Mutex - Rust. https://doc.rust-lang.org/std/sync/struct.Mutex.html

[63] Rust Community. 2021. std::thread::spawn - Rust. https://doc.rust-lang.org/std/thread/fn.spawn.html

[64] Rust Community. 2021. std::vec::Vec - Rust. https://doc.rust-lang.org/std/vec/struct.Vec.html

[65] Rust Community. 2021. Two-phase borrows. https://rust-lang.github.io/rustc-guide/borrow_check/two_phase_borrows.html

[66] Ryosuke Sato, Naoki Iwayama, and Naoki Kobayashi. 2019. Combining Higher-Order Model Checking with Refinement Type Inference. In *Proceedings of the 2019 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM@POPL 2019, Cascais, Portugal, January 14-15, 2019*, Manuel V. Hermenegildo and Atsushi Igarashi (Eds.). ACM, 47–53. https://doi.org/10.1145/3294032.3294081

[67] Aaron Stump, Clark W. Barrett, David L. Dill, and Jeremy R. Levitt. 2001. A Decision Procedure for an Extensional Theory of Arrays. In *16th Annual IEEE Symposium on Logic in Computer Science, Boston, Massachusetts, USA, June 16-19, 2001, Proceedings*. IEEE Computer Society, 29–37. https://doi.org/10.1109/LICS.2001.932480

[68] Kohei Suenaga and Naoki Kobayashi. 2009. Fractional Ownerships for Safe Memory Deallocation. In *Programming Languages and Systems, 7th Asian Symposium, APLAS 2009, Seoul, Korea, December 14-16, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5904)*, Zhenjiang Hu (Ed.). Springer, 128–143. https://doi.org/10.1007/978-3-642-10672-9_11

[69] Tachio Terauchi. 2008. Checking Race Freedom via Linear Programming. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, Rajiv Gupta and Saman P. Amarasinghe (Eds.). ACM, 1–10. https://doi.org/10.1145/1375581.1375583

[70] John Toman, Stuart Pernsteiner, and Emina Torlak. 2015. CRUST: A Bounded Verifier for Rust. In *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*, Myra B. Cohen, Lars Grunske, and Michael Whalen (Eds.). IEEE Computer Society, 75–80. https://doi.org/10.1109/ASE.2015.77

[71] Sebastian Ullrich. 2016. Electrolysis Reference. http://kha.github.io/electrolysis/

[72] Sebastian Ullrich. 2016. *Simple Verification of Rust Programs via Functional Purification.* Master's thesis. Karlsruhe Institute of Technology.

[73] Viktor Vafeiadis. 2008. *Modular fine-grained concurrency verification.* Ph.D. Dissertation. University of Cambridge, UK. http://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.612221

[74] Maarten H. van Emden and Robert A. Kowalski. 1976. The Semantics of Predicate Logic as a Programming Language. *J. ACM* 23, 4 (1976), 733–742. https://doi.org/10.1145/321978.321991

[75] Z3 Team. 2021. The Z3 Theorem Prover. https://github.com/Z3Prover/z3