CHC-based Program Verification Exploiting Ownership Types
所有権型を利用した CHC ベースのプログラム検証

by

Yusuke Matsushita
松下 祐介

A Senior Thesis
卒業論文

Submitted to
the Department of Information Science
the Faculty of Science, the University of Tokyo
on February 28, 2019
in Partial Fulfillment of the Requirements
for the Degree of Bachelor of Science

Thesis Supervisor: Naoki Kobayashi　小林 直樹
Professor of Information Science

ABSTRACT

Reduction to constrained Horn clauses (CHCs) is a recently popular approach to automated program verification. One of the main challenges in this context is how to treat pointers and destructive updates. A conventional method used pointer analysis and array theory to address this issue; since resulting CHCs are rather complex, however, it suffered from a scalability problem. This thesis shows how to exploit ownership types, as provided in the Rust programming language, to improve the reduction of program verification problems to sets of CHCs. Our new translation does not call for pointer analysis or array theory in dealing with pointers controlled by ownership. Moreover, it outputs simpler CHCs than the conventional method does, and generally shows better verification performance when combined with an existing CHC solver. This study also formalizes the translation and describes a conjecture on the correctness.

論文要旨

制約付きホーン節 (CHC) への帰着は自動プログラム検証で近年人気のある手法である。これにおける主な困難の一つは、ポインタおよび破壊的更新の扱いである。従来手法はポインタ解析や配列理論を利用してこの問題に取り組んだが、出力される CHC が比較的複雑であるため、規模を大きくしにくい問題があった。この論文では、Rust プログラミング言語で提供されるような所有権型を利用することでプログラム検証問題の CHC 集合への帰着を改善する方法を示す。この新しい翻訳法は所有権で制御されたポインタを扱う上ではポインタ解析や配列理論を要さない。さらに、得られる CHC は従来手法よりも単純であり、既存の CHC ソルバとの組み合わせで概してより良い検証性能を示す。また、この研究では変換を形式化し、その正しさについての予想を記述する。

Acknowledgments

I would like to express my deep gratitude to my supervisor Professor Naoki Kobayashi and co-supervisor Associate Professor Takeshi Tsukada for patient guidance and encouragement during this research. I would also like to thank Mr. Masaki Hara, an enthusiastic contributor to the Rust community, for his great advice, especially in the early stages of the research.

Contents

Chapter 1

Introduction

1.1 Program Verification

Program verification is the act of proving/disproving that target programs satisfy particular requirements. In this paper, the term *verification* refers to creation of *strict, formal guarantees*. Methods like unit testing basically do not give rigid guarantees on the correctness, and thus are outside the scope. As illustrated below, a variety of (formal) program verification methods are conceived.

There exist many *automated* methods for program verification. This paper mainly focuses on automated verification using *constrained Horn clauses (CHCs)*. It embeds programs into a particular form of formulae in predicate logic; CHC-based verification is later explained in detail. Verification using model checking on *higher-order modal fixed-point logic (HFL)* [67, 32] shares the same spirit with CHC-based verification in that programs are translated into tractable logical entities. In addition, verification using model checking on *higher-order recursion schemes (HORSs)* [30, 38] is strongly related to verification using HFL [31]. Although such translation of programs into amenable entities gives clear insights and good heuristics, studies on these kinds of automated verification methods, especially for programs on infinite-state systems, are just in progress.[1] Let us see a few other automated methods for program verification. *Type checking* provides lightweight verification on programs, but the verification power is often quite limited. Some approaches, such as Checker Framework [46], extend the type system of existing mainstream languages. *Bounded model checking* searches some finite, bounded scope of inputs for bugs and errors; it is easy to use and gives a firm guarantee of the correctness for the bounded scope, but it usually requires a lot of computational resources and still does not give a general, complete guarantee.

Semi-automated methods for verification are also eagerly studied; such methods allow great flexibility at the cost of human involvement. Some approaches, such as ESC/Java2 [29], Why3 [61] and Spec# [59], make use of manual annotations on properties like preconditions and postconditions, and automatically check the correctness. Verification using *proof assistants*, such as Agda [44], Coq [47], Isabelle [48] and Lean [49], involves both machine calculation (for proof checking, proof search, etc.) and manual description of proof tactics; proof assistants are usually designed for general mathematical proofs, but they can easily be applied for program verification.

[1]Thinking of the famous Collatz conjecture, halting problems of some apparently simple programs turn out to be tremendously difficult.

1.1.1 CHC-based Program Verification

Reduction to *constrained Horn clauses (CHCs)*, formulae of a particular form in *predicate logic*, is a recently popular approach to automated program verification. [4] is one of the earliest proposals to use CHCs as a universal intermediate format for representing verification conditions; this design strategy is adopted by modern verification frameworks such as JayHorn [27, 28] and SeaHorn [54, 14], which are introduced later. As mentioned in [5], the connection between program logics and constrained Horn clauses in CHC-based verification can be seen as a descendant of *Floyd-Hoare logic* [11, 19]. This subsection gives a brief summary of CHC-based program verification. For more information, [5] discusses useful properties and techniques on CHC-based program verification.

Constrained Horn Clauses

A *constrained Horn clause* (CHC) is defined as follows. Into an ordinary first-order predicate logic, we newly introduce *uninterpreted* (atomic) predicates; let R represent an uninterpreted predicate. An extended formula $\hat{\varphi}$ is either a formula (without uninterpreted predicates) φ or an uninterpreted predicate applied to terms $R(t_1, \dots, t_n)$. A constrained Horn clause has form $\hat{\varphi}_0 \iff \hat{\varphi}_1 \wedge \cdots \wedge \hat{\varphi}_n$; free variables in a CHC are treated as universally quantified under appropriate scopes (e.g. within the set of natural numbers), which are not explicitly given in principle throughout the paper.

CHCs can be regularized into either of the following forms, where metavariables of form x^i_j stand for mutually distinct variables.

$$R_0(x^0_1, \dots, x^0_{m_0}) \Longleftarrow R_1(x^1_1, \dots, x^1_{m_1}) \wedge \cdots \wedge R_n(x^n_1, \dots, x^n_{m_n}) \wedge \varphi$$
$$\bot \Longleftarrow R_1(x^1_1, \dots, x^1_{m_1}) \wedge \cdots \wedge R_n(x^n_1, \dots, x^n_{m_n}) \wedge \varphi$$

Translating Programs to Sets of CHCs

In CHC-based verification, *functions* are represented as *uninterpreted predicates* describing relationship between inputs and outputs. *function definitions* (possibly recursive) are described as *sets of CHCs*.

Consider as an example the following recursively defined (total) function on natural numbers.

$$\mathtt{fact}(n) = \begin{cases} 1 & n = 0 \\ n \cdot \mathtt{fact}(n-1) & n > 0 \end{cases}$$

It is the well-known factorial function; in other words, $\mathtt{fact}(n) = n!$ holds. This function is translated into the following set of CHCs.

$$\begin{aligned} \mathrm{fact}(n, r) &\Longleftarrow n = 0 \wedge r = 1 \\ \mathrm{fact}(n, r) &\Longleftarrow n > 0 \wedge \mathrm{fact}(n-1, r') \wedge r = n \cdot r' \end{aligned}$$

Some predicates, such as $\mathtt{fact}(n) = r$ (i.e. $n! = r$), $\top$, $r > 0$ and $r > 0 \wedge r \geq n$ (where n and r are passed as arguments), satisfy these CHCs as $\mathrm{fact}(n, r)$; we call such predicates *fixed points* or *invariants*. In fact, $\mathtt{fact}(n) = r$ is the least of such predicates (under the pointwise preorder, with $\bot$ smaller than $\top$); thus we call it the *least fixed point*. In general, the least fixed point corresponds to the semantics of the function; this is the central principle in CHC-based program verification.

Infinite loops can also be tackled by this framework. Consider as another example the following *partial* function $\mathtt{id\text{-}or\text{-}loop}(n)$ on natural numbers which is recursively defined.

$$\texttt{id-or-loop}(n) = \begin{cases} n & n < 7 \\ \texttt{id-or-loop}(n+1) & n \geq 7 \end{cases}$$

This returns n for an input less than seven, and falls into an infinite loop for an input that is seven or more. The (partial) function is translated into the following set of CHCs.

$$\begin{aligned} \text{id-or-loop}(n, r) &\impliedby n < 7 \ \wedge \ r = n \\ \text{id-or-loop}(n, r) &\impliedby n \geq 7 \ \wedge \ \text{id-or-loop}(n+1, r) \end{aligned}$$

Predicate $n < 7 \ \wedge \ r = n$ is the least fixed point, and is equivalent to condition "$\texttt{id-or-loop}(n)$ (terminates and) returns r".

Verification of properties of form "$\varphi(x_1, \ldots, x_n, r)$ holds for any $x_1, \ldots, x_n, r$ such that $\texttt{f}(x_1, \ldots, x_n)$ returns r" can be reduced to *satisfiability* problems on CHCs, by translating the function definition of $\texttt{f}(x_1, \ldots, x_n)$ into CHCs for an uninterpreted predicate $\text{f}(x_1, \ldots, x_n)$ and adding CHCs of form $\varphi \impliedby \text{f}(x_1, \ldots, x_n, r)$.

If we want to *prove* $\texttt{fact}(n) = r \ \wedge \ n > 2 \implies n < r$ $(\forall n, r \in \mathbb{N})$, for example, it suffices to show that the following set of CHCs is *satisfiable*.

$$\begin{aligned} \text{fact}(n, r) &\impliedby n = 0 \ \wedge \ r = 1 \\ \text{fact}(n, r) &\impliedby n > 0 \ \wedge \ \text{fact}(n-1, r') \ \wedge \ r = n \cdot r' \\ n < r &\impliedby \text{fact}(n, r) \ \wedge \ n > 2 \end{aligned}$$

Satisfiability can be proved by *finding a fixed point* that is a *sufficient condition* of the property φ in question. In this example, every CHC will be true by setting $\text{fact}(n, r)$ to $n \leq r \ \wedge \ (n > 2 \implies n < r)$.

If we want to *disprove* $\texttt{fact}(n) = 1$ $(\forall n \in \mathbb{N})$, for another example, it suffices to show that the following set of CHCs is *unsatisfiable*.

$$\begin{aligned} \text{fact}(n, r) &\impliedby n = 0 \ \wedge \ r = 1 \\ \text{fact}(n, r) &\impliedby n > 0 \ \wedge \ \text{fact}(n-1, r') \ \wedge \ r = n \cdot r' \\ r = 1 &\impliedby \text{fact}(n, r) \end{aligned}$$

We can show unsatisfiability of a set of CHCs by *deriving contradiction* from the CHCs, which basically corresponds to *finding a counterexample*. In the example, we can derive $\text{fact}(0, 1)$ from the first CHC, $\text{fact}(1, 1)$ from the second CHC and $\text{fact}(0, 1)$, $\text{fact}(2, 2)$ from the second CHC and $\text{fact}(1, 1)$, and finally $\bot$ (contradiction) from the third CHC and $\text{fact}(2, 2)$; by this derivation unsatisfiability of the set of CHCs is proved. It amounts to finding a counterexample $\texttt{fact}(2) \neq 1$.

Solving CHCs

There exist a number of techniques for solving satisfiability problems on CHCs, which are well discussed in [5].

The fundamental technique is *resolution*. It generates a new CHC from two CHCs *sharing an uninterpreted predicate on the positive side and the negative side* (i.e. on the left-hand side and the right-hand side of $\impliedby$), respectively. Through resolution, the set of CHCs can be safely expanded. For example, resolution

$$\text{from} \quad \begin{aligned} R_0(x) &\impliedby R_*(x) \ \wedge \ \varphi_2 \\ R_*(x) &\impliedby R_1(x) \ \wedge \ \varphi_1 \end{aligned} \quad \text{yields} \quad R_0(x) \impliedby R_1(x) \ \wedge \ \varphi_1 \ \wedge \ \varphi_2.$$

This is a crucial technique, since *every correct variable-free formula* of form $R(t_1, \ldots, t_n)$ (an uninterpreted predicate applied to variable-free terms $t_1, \ldots, t_n$) can be obtained by resolution (with some minor adjustment).

The *unfold* transformation is a notable application of resolution. The transformation *erases a particular uninterpreted predicate* R_* from a set of CHCs, by replacing its negative appearance with its original preconditions (i.e. every negative appearance of $R_*(x_1, \ldots, x_n)$ is replaced with each B of the CHCs of form $R_*(x_1, \ldots, x_n) \iff B$, under regularization) and then removing the CHCs in which R_* appears positively; there is the limitation that the preconditions of R_* should not have R_*. For example, unfolding on R_* transforms

$$\begin{array}{rcl} R_*(x) & \Longleftarrow & R_1(x) \land \varphi_1 \\ R_*(x) & \Longleftarrow & R_2(x) \land \varphi_2 \\ R_0(x) & \Longleftarrow & R_*(x) \land \varphi_0 \end{array} \quad \text{into} \quad \begin{array}{rcl} R_0(x) & \Longleftarrow & R_1(x) \land \varphi_1 \land \varphi_0 \\ R_0(x) & \Longleftarrow & R_2(x) \land \varphi_2 \land \varphi_0 \end{array}.$$

A precise definition of the unfold transformation is given in [5]; several properties of the transformation are also discussed in the paper.

The cutting-edge multi-purpose theorem proving engine Z3 [62] embraces a sophisticated CHC solver Spacer [58, 34, 33]. Basic CHC solving techniques used in Z3 are discussed in [21, 20]; *property directed reachability* is the key concept in finding out fixed points and counterexamples.

1.1.2 Pointer Analysis

Pointer analysis collects information on which *pointers* can point to which *storage locations*, as a part of static program analysis. It is widely used as preprocessing for program verification. [18] gives a summary of techniques and problems on pointer analysis. The SeaHorn verification framework performs a field-, array-, and context-sensitive pointer analysis tailored for verification of low-level C/C++ programs [15], which is inspired by data structure analysis (DSA) [35]; a context-sensitive pointer analysis partitions memory into multiple regions exploiting the information on call paths.

1.1.3 Array Theory

Array theory is one of the theories (of predicate logic) designed for the system of *SMT*, or *satisfiability modulo theories*. In this theory, an *array* a is a mapping from a set of indices (typically, the set of integers) to a set of elements, equipped with *read* and *write* operations. The *read* operation takes out the value stored at index i in array a (written as $a[i]$). The *write* operation creates the array such that the value at index i is x and the values at the other indices are the same as array a (written as $a\{i \triangleleft x\}$). There are some non-trivial decidable classes of satisfiability problems in regard to array theory. More information on decidability about array theory can be found in [6, 16, 13]. The Spacer CHC solver in the Z3 engine can deal with arrays as in array theory.

1.1.4 Program Verification Tools

There are a number of program verification tools targeting real-world programs with pointers and destructive updates.

JayHorn [27, 28] is a CHC-based framework for verifying Java programs. It first takes Java bytecode, and exerts many bytecode-level transformations using Soot [42] to simplify exception handling, virtual methods, control flows; it then translates the bytecode into CHCs with a simple algorithm, and verify the generated CHCs with an existing CHC solver, such as Eldarica [41] and Spacer.

SeaHorn [54, 14] is a fully automated analysis framework for LLVM-based languages including C and C++, which also uses CHCs to describe verification conditions. It takes LLVM IR bitcode, and performs bitcode-level optimizations for verification. Then it emits verification conditions as CHCs from the optimized LLVM IR bitcode; it models heaps using clever techniques such as one presented in [15], as mentioned earlier. It finally performs verification on the generated CHCs using a CHC solver such as Spacer; it also assists the CHC solver by providing numerical invariants with abstract-interpretation-based static analyzer CRAB [55].

SMACK [57] is also a software verifier that takes LLVM IR. Unlike SeaHorn, however, it does not use CHCs for describing verification conditions. It translates IR into the Boogie intermediate verification language [45]; the tool for Boogie optionally infers some invariants, and finally generates verification conditions that are passed to an SMT solver (not a CHC solver).

1.2 Ownership Types

Ownership types refer to types controlled under a type system that employs an idea called *ownership*. Simply put, ownership can be explained as follows: multiple aliases to a resource may coexist, but only one of them can "own" the full right of operation on the resource; although the right is uncopyable, it can be "transferred" from one alias to another; in addition, some more advanced systems allow multiple aliases to share the read access.

There are many studies related to ownership types. The paper [8] introduces a formal type system for ownership types in an object-oriented programming language, as a "flexible mechanism to limit the visibility of object references and restrict access paths to objects". Vault [10] and Cyclone [23] are safer dialects of the C programming language that make use of region-based systems, which are related to ownership. L^3 [1], System F° [36], Alms [64] and Mezzo [2] are programming languages with substructural type systems that employ ownership.

Furthermore, modern C++ (especially since C++11) [22] provides ownership-related features such as smart pointers supporting the RAII (Resource acquisition is initialization) paradigm, and move semantics which enables transferring ownership combined with rvalue references; while these features do encourage safer coding practices, however, the type system of C++ still does not ensure ownership-based safety. Generally speaking, application of an academically studied ownership-based type system to a real-world programming language often results in making the language too restrictive for practical use.

The Rust programming language [52] is a new[2] programming language that aggressively supports ownership types; it is a systems programming language that allows *efficient low-level memory operations under strong compile-time safety guarantees*. Whereas ownership-based type checking is quite strict, `unsafe` code blocks loosen type checking and enable more flexible memory operations; and through encapsulation as a library, especially by the use of *traits* (analogous to interfaces of Java), the unsafe behaviors can be used by programmers in a predictable, controlled way. This *extensibility* of Rust makes the language flexible enough for producing efficient, practical code under great guarantees. Rust

[2]The first stable release (Rust 1.0) was released in 2015.

has been spreading its use, and it is used even for developing a full-featured operating system [50] and a modern, high-performance browser engine [56].

In this paper, we extensively use Rust as a representative example of a programming language with ownership types. In the following part of the paper, a number of Rust code examples are presented, but the syntax of Rust is roughly illustrated in comments so that those not familiar with Rust can understand the code. For more information on Rust, [53] and [51] give detailed explanations.

1.2.1 Introduction to Rust-style Ownership Types

This subsection briefly explains Rust-style ownership types.

Owning pointers. First of all, when a variable `x` is typed `T`, `x` has *ownership* on data of type `T`; for example, if we perform `let mut x : (i32,i32,i32)= (1,2,3);`, variable `x` of a tuple of three 32-bit integers `(i32,i32,i32)` will be initialized as data `(1,2,3)`.[3] When the data of type `T` is placed on the memory,[4] we can interpret `x` as an implicit pointer to the data. In this sense, we regard `x` as an *owning pointer*[5] to the data.

Move and copy. When we perform let binding `let mut y : T = x;`, the situation diverges by whether (data of) type `T` is copyable, indicated by `Copy` trait in Rust.[6] When `T` is copyable (e.g. a 32-bit integer), `y` is initialized as data copied from `x`; `x` and `y` refer to *different places* as owned pointers. When `T` is not copyable (e.g. a mutable reference, as explained later), a *move*, or transfer of ownership, takes place; `y` refers to the same data that `x` used to refer to, and `x` completely loses the power of ownership. To sum up, *only one, unique owning pointer* can refer to each data.

References, lifetimes and (re)borrowing. A *reference* is a *temporary* alias to data that some owning pointer already has control over. There are two types of references, mutable and immutable; a mutable reference can mutate the data that an owning pointer has, and an immutable reference can just read from the data. As explained later, any references can be created just by *borrowing* or *reborrowing*. Every reference is statically controlled under a *lifetime*, which is represented in Rust as a variable of form `'a` (starting with a single quote); for our purpose, a lifetime can be regarded as the *time limit* for releasing the temporary control (acquired by (re)borrowing) on the data.[7] `&'a mut T` is the type of a mutable reference, to data of type `T` controlled under lifetime `'a`, and `&'a T` is the type of an immutable reference. When variable `mx` has type `&'a mut (i32, i32,i32)`,[8] for example, you can set its data simply by `*x = (8,7,6);`.[9] While some reference, mutable or immutable, is active, the corresponding owning pointer gets *shadowed*. In addition, only two patterns are allowed on a presence of references to each resource: there is *only one mutable reference*, or there are *one*

[3]Here, `mut` in `let mut x` indicates that the data of `x` can be mutated like `x = (7,6,5);`. If this `mut` is eliminated, the data of `x` is not allowed to change.

[4]When the data is a 32-bit integer, it might be on some register in the hardware; when `T` is quite big, however, such as a tuple of twelve 32-bit integers, the data is very likely to be on the memory.

[5]This is not a very common term. RustBelt [25] calls it an owned pointer, but the point is that the pointer owns the data, rather than that the pointer is owned.

[6]In Rust's terminology, "copying" means shallow copying, like simple `memcpy` in C; deeper copying is called "cloning", and clonability is indicated by `Clone` trait.

[7]As explained later in another footnote, this is not the case for newly introduced non-lexical lifetimes.

[8]It of course means that `mx` has ownership on the address data of type `&'a mut (i32,i32,i32)`, but features like auto-(de)referencing in Rust make things much more complicated.

[9]You may wonder here what the type of `*x` is; it can be regarded as `(i32,i32,i32)`, but it does not behave like an owning pointer. Such things are just artfully and implicitly treated by Rust.

or more immutable references; thus *mutable references cannot be copied*. Lifetimes are thus used for statically imposing these kinds of *safety invariants*. Finally, let us see a little more on borrowing and reborrowing. References can be created by *borrowing* from owning pointers. Given an owning pointer `x` of type `T`, you can borrow a mutable reference from it by `let mx : &mut T = &mut x;` (you cannot specify the lifetime here; it is implicitly controlled by Rust's type system) and an immutable reference by `let ix : &T = &x;`. References can also be created by *reborrowing* from exiting mutable references. For example, you can reborrow from a mutable reference `mx` by `let mx' : &mut T = &mut *mx;`. We can thus say that a lifetime indicates the *end of borrowing or reborrowing*.

Lifetime abstraction. Some functions in Rust receive references as inputs and outputs. For example (as described later in Example A), you can write a function that takes two mutable references to (different) integer data and returns the reference with the larger integer value. In order to write these kinds of functions, we use *lifetime abstraction*; we just add *lifetime parameters* to functions, which are embodied into concrete lifetimes when the functions are called.

Box type and recursive types. Rust's `Box<T>`, Box type, works as an explicit *owning pointer* to some resource on the heap memory. You can take mutable or immutable references from `Box<T>`. When the variable owning `Box<T>` is dropped, the resource on the heap memory is released. Thus `Box<T>` works similarly to simple `T` itself, but Box type is so helpful on defining *recursive types*. A (singly linked) list type can be defined as `enum List<T> Nil, Cons(T, Box<List<T>>)`, where `enum` brings a tagged union (direct sum) and `<T>` introduces a type parameter `T`; the type cannot be defined as `enum List<T> Nil, Cons(T, List<T>)`, because it makes the size of the type *infinite*. Thus Box type serves as a kind of cushion for defining recursive types.

Splitting references. You can split a reference and obtain references to a smaller region of the memory. For example, when you have a mutable reference to a pair `mx : &'a mut (i32,bool)`, you can split it to `ma : &'a mut i32` and `mb : &'b mut bool` by pattern matching `let (ma, mb)= mx;`; after this, you cannot use `mx` anymore — `mx` is now split into `ma` and `mb`. This apparently simple feature of Rust gives great expressive power; as shown in Example C, for example, you can split a mutable reference to a (singly linked) list into a list of mutable references to each element.

1.3 This Research

CHC-based verification taking advantage of ownership types has not been studied well. This research achieves major progress in this area.

Chapter 2 gives an overview of our translation along with some examples. Chapter 3 formalizes our translation through a newly introduced formal language *Calculus of Ownership and Reference* (COR) corresponding to a basic subset of Rust, and describes a conjecture on the correctness of the translation. Chapter 4 shows the results of an experiment and discusses the verification performance achieved with our method.

Chapter 2

Overview of the Translation

This chapter provides an informal overview of our translation, with a number of illustrative examples. It also compares our method with the conventional address-based method.

2.1 Basic Ideas

When only owning pointers and immutable references are used and mutable references are not used, the situation is quite easy. For example, consider the following Rust code with a loop of destructive updates on owning pointers.

```
fn inc_loop<'a>(mut x: i32, iy: &'a i32) -> i32 {
  // fn stands for a function, and i32 is a 32-bit integer type;
  // 'mut x' indicates that x can be mutated;
  // the function finally returns x (the original value) + *iy
  let mut i: i32 = 0;
  while i < *iy { i += 1; x += 1; }
  x
    // Rust does not require 'return' when returning values;
    // it works like functional programming languages
}
fn check_inc_loop(x: i32, y: i32) -> i32 {
  let res = inc_loop(x, &y);
    // temporarily immutably borrow y,
    // and call inc_loop to get the result r;
    // the function terminates if y >= 0
  res - (x + y)
    // return the difference from the expected answer x + y
}
```

Naming the continuation from the while loop ‘while’, the program can be translated into the following set of CHCs. The immutable reference to an integer y, named iy, is represented as a box (1-tuple) containing y, and written as $\langle y \rangle$.

$$\begin{aligned}
\text{inc-loop}(x, iy, r) &\Longleftarrow \text{while}(0, x, iy, r) \\
\text{while}(i, x, \langle y \rangle, r) &\Longleftarrow i < y \ \wedge\ \text{while}(i+1, x+1, \langle y \rangle, r) \\
\text{while}(i, x, \langle y \rangle, r) &\Longleftarrow i \geq y \ \wedge\ r = x \\
\text{check-inc-loop}(x, y, r) &\Longleftarrow \text{inc-loop}(x, \langle y \rangle, res) \ \wedge\ r = res - (x + y)
\end{aligned}$$

Destructive updating on owning pointers just can be seen as rebinding or shadowing for variables, as provided in purely functional languages.

In the presence of *mutable references* (or equivalently *mutable borrowing*), however, this kind of treatment does not work appropriately. Actually, our translation treats mutable references in a novel manner. For example, consider the following

Rust code; `inc_max` takes two integer variables `x` and `y`, increment the larger one via a pointer obtained by `take_max`, and return the pair of the two values.

```
fn take_max<'a>(mx: &'a mut i32, my: &'a mut i32) -> &'a mut i32 {
  if(*mx >= *my) { mx } else { my }
    // return the larger side of mx and my
}
fn inc_max(mut x: i32, mut y: i32) -> (i32, i32) {
  // 'mut x' and 'mut y' mean that x and y can be mutated;
  // the annotations are needed for mutable borrowing
  {
    let mx = &mut x; let my = &mut y;
      // mutably borrow x as mx and y as my
    let mz = take_max(mx, my); *mz += 1;
      // take the larger side as mz and increment it;
      // thereby either x or y is updated
  } // end borrowing x as mx and y as my
  (x, y) // return the pair of x and y
}
```

This program is quite simple but somewhat difficult to translate into CHCs. If we try to represent mutable references as a box, we get stuck like this. (The symbol ? indicates a perplexing part.)

$$\begin{aligned}
\text{take-max}(\langle x\rangle, \langle y\rangle, r) &\impliedby x \geq y \ \land\ r = \langle x\rangle \\
\text{take-max}(\langle x\rangle, \langle y\rangle, r) &\impliedby x < y \ \land\ r = \langle y\rangle
\end{aligned}$$

$$\begin{aligned}
\text{inc-max}(x, y, r) \impliedby\ & \text{take-max}(\langle x\rangle, \langle y\rangle, \langle z\rangle) \\
& \land\ ? = z + 1 \ \land\ r = (?, ?)
\end{aligned}$$

The problem is how to obtain the values of updated versions of x and y. The value $l + 1$ should be the updated version of either x or y, but which? The traditional method used addresses to determine "which", but we conceived a simpler way to resolve it: we just need to tag to each mutable reference the *future, returned value*. The CHCs that our method yields are as follows.

$$\begin{aligned}
\text{take-max}(\langle x, x_*\rangle, \langle y, y_*\rangle, r) &\impliedby x \geq y \ \land\ y_* = y \ \land\ r = \langle x, x_*\rangle \\
\text{take-max}(\langle x, x_*\rangle, \langle y, y_*\rangle, r) &\impliedby x < y \ \land\ x_* = x \ \land\ r = \langle y, y_*\rangle
\end{aligned}$$

$$\begin{aligned}
\text{inc-max}(x, y, r) \impliedby\ & \text{take-max}(\langle x, x_*\rangle, \langle y, y_*\rangle, \langle z, z_*\rangle) \\
& \land\ z_* = z + 1 \ \land\ r = (x_*, y_*)
\end{aligned}$$

Each mutable reference mx is represented as a special pair $\langle x, x_*\rangle$ of the current value x and the *future value* x_* *that is returned at the end of the mutable borrowing*. Taking a future value may sound an astounding idea, but it is just what ordinary CHC-based verification does; the output of a function is represented as an argument of each uninterpreted predicate. In contrast to the conventional *address-based* method, our translation employs a *value-based* method.

Summary. Mutable borrowing triggers *switching of (full) ownership*, which is hard to manage with purely functional approaches. To deal with this situation, our method takes the *future value* that is planned to be passed when switching back the ownership (i.e. ending the mutable borrowing); each mutable reference keeps both the current value and the future value. Informally speaking, the future value is *symbolically* represented, and is passed around as the mutable reference travels around; and the value is determined on dropping the mutable reference (or giving up the ownership). This symbolical handling of future values is a natural extension of the policy of representing the output of a function as an argument of an uninterpreted predicate.

2.1.1 Operations on Mutable References Expressed in CHCs

This subsection gives an informal but detailed explanation of the method of expressing operations on mutable references in CHCs. These ideas are the basis of the formalization introduced in Chapter 3.

There are four types of basic operations on mutable references in Rust: (1) *creating* a mutable reference by (re)borrowing, (2) *updating* the referent of a mutable reference, (3) *returning* a mutable reference by dropping, and (4) *splitting* a mutable reference handling a memory area into mutable references handling subareas. In addition, splitting can be classified into three patterns: (4-a) getting mutable references to the elements of a tuple, (4-b) getting a mutable reference to the body of a tagged value, and (4-c) getting a mutable reference to the referent of a pointer.

In our translation, the four types of operations on mutable references are expressed in CHCs as follows.

(1) Creating To express *borrowing* of an owned value x, we take the return value x_* by existential quantification; the resulting mutable reference is $\langle x, x_* \rangle$, and the updated version of the owned value is x_*. Likewise, to express *reborrowing* of an existing mutable reference $\langle x, x_{**} \rangle$, we name the return value x_* to get a new mutable reference $\langle x, x_* \rangle$ along with the updated version of the existing mutable reference $\langle x_*, x_{**} \rangle$.

(2) Updating In order to express update of the value of a mutable reference $\langle x, x_{**} \rangle$ from x to x_*, we get $\langle x_*, x_{**} \rangle$ as the updated version of the mutable reference.

(3) Returning To express return of a mutable reference $\langle x, x_* \rangle$ by dropping, we just add constraint $x_* = x$.

(4) Splitting For (4-a), given a mutable reference to a tuple $\langle (x_1, \dots, x_n), p_* \rangle$, we obtain mutable references $\langle x_1, x_{1*} \rangle$, $\dots$, $\langle x_n, x_{n*} \rangle$ and add constraint $p_* = (x_{1*}, \dots, x_{n*})$. For (4-b), given a mutable reference to a tagged value $\langle A(y), x_* \rangle$ (where A is a data constructor, or a tag, of a tagged value), we obtain $\langle y, y_* \rangle$ taking a new value y_*, and append constraint $x_* = A(y_*)$. For (4-c), expressing dereference of a mutable reference to an owning pointer or an immutable reference is analogous to (4-b); to express dereference of a mutable reference to another mutable reference $\langle \langle y, y_{**} \rangle, x_* \rangle$, we get a new mutable reference $\langle y, y_* \rangle$ taking a new value y_*, and add constraint $x_* = \langle y_*, y_{**} \rangle$.

2.1.2 Extensions of the Translation

This subsection describes extensions of our translation, which is not formalized in Chapter 3.

Slices and Vectors

To extend our translation for *slices* (dynamically-sized views into a contiguous sequence in the memory; there are two types of slices, represented as `[T]`) and *vectors* (contiguous growable arrays, represented as `Vec<T>`), we need to use array theory to represent slices and vectors, but the technique of our translation is still applicable.

We can model both slices and vectors as an array a from natural numbers to values, equipped with the size information n; let us write slices and vectors as (a, n). Here we present some representative operations on slices and vectors.

Immutable access. $\mathrm{index}((a, n), i, x)$ (immutably access a slice or vector $\langle a, n \rangle$ at index i to obtain x; equivalent to Rust's `index` of `Index` trait) can be expressed in a CHC as follows.

$$\text{index}((a,n),i,x) \iff i < n \;\wedge\; x = a[i]$$

There are several ways to represent panics (exceptions), which here can be triggered by out-of-bounds access; in the CHC representation here, panics are expressed simply by returning no value; if $i \geq n$, then no x satisfies $\text{index}((a,n),i,x)$.

Mutable access. $\text{index-mut}\big(\langle(a,n),(a_*,n_*)\rangle,i,\langle x,x_*\rangle\big)$ (mutably access a mutable reference to a slice or vector $\langle(a,n),(a_*,n_*)\rangle$ at index i to obtain $\langle x,x_*\rangle$; equivalent to Rust's `index_mut` of `IndexMut` trait) is expressed in a CHC as follows.

$$\begin{aligned}\text{index-mut}\big(\langle(a,n),(a_*,n_*)\rangle,i,\langle x,x_*\rangle\big) \iff{}& i < n \;\wedge\; n_* = n \\ &\wedge\; x = a[i] \;\wedge\; a_* = a\{i \triangleleft x_*\}\end{aligned}$$

Note that when x_* is determined (by dropping the mutable reference $\langle x,x_*\rangle$), a_* is accordingly determined.

Pushing a value. $\text{push}\big(\langle(a,n),(a_*,n_*)\rangle,x\big)$ (push value x to a mutably referenced vector $\langle(a,n),(a_*,n_*)\rangle$; equivalent to Rust's `push`) is expressed in a CHC as follows.

$$\text{push}\big(\langle(a,n),(a_*,n_*)\rangle,x\big) \iff n_* = n+1 \;\wedge\; a_* = a\{n \triangleleft x\}$$

Swapping two elements. $\text{swap}\big(\langle(a,n),(a_*,n_*)\rangle,i,j\big)$ (swap two elements of index i and j on a mutably referenced slice or vector $\langle(a,n),(a_*,n_*)\rangle$; equivalent to Rust's `swap`) is expressed in a CHC as follows.

$$\begin{aligned}\text{swap}\big(\langle(a,n),(a_*,n_*)\rangle,i,j\big) \iff{}& i < n \;\wedge\; j < n \;\wedge\; n_* = n \\ &\wedge\; a_* = a\{i \triangleleft a[j]\}\{j \triangleleft a[i]\}\end{aligned}$$

Splitting. $\text{split-at-mut}\big(\langle(a,n),(a_*,n_*)\rangle,i,\langle(a^{\triangleleft},n^{\triangleleft}),(a^{\triangleleft}_*,n^{\triangleleft}_*)\rangle,\langle(a^{\triangleright},n^{\triangleright}),(a^{\triangleright}_*,n^{\triangleright}_*)\rangle\big)$ (split a mutably referenced slice $\langle(a,n),(a_*,n_*)\rangle$ at index i to obtain two mutable referenced slices $\langle(a^{\triangleleft},n^{\triangleleft}),(a^{\triangleleft}_*,n^{\triangleleft}_*)\rangle$ and $\langle(a^{\triangleright},n^{\triangleright}),(a^{\triangleright}_*,n^{\triangleright}_*)\rangle$; equivalent to Rust's `split_at_mut`) is expressed in a CHC as follows, extending array theory with a new primitive $\text{shift-left}(a,i)$ (the array obtained by cutting off the first i elements of array a).

$$\begin{aligned}&\text{split-at-mut}\big(\langle(a,n),(a_*,n_*)\rangle,i,\langle(a^{\triangleleft},n^{\triangleleft}),(a^{\triangleleft}_*,n^{\triangleleft}_*)\rangle,\langle(a^{\triangleright},n^{\triangleright}),(a^{\triangleright}_*,n^{\triangleright}_*)\rangle\big) \\ &\quad\iff i < n \;\wedge\; n_* = n \;\wedge\; n^{\triangleleft} = i \;\wedge\; n^{\triangleright} = n - i \\ &\qquad\wedge\; a^{\triangleleft} = a \;\wedge\; a^{\triangleleft}_* = a_* \\ &\qquad\wedge\; a^{\triangleright} = \text{shift-left}(a,i) \;\wedge\; a^{\triangleright}_* = \text{shift-left}(a_*,i)\end{aligned}$$

If you would like to use the usual array theory, you can define the following helper predicate $\text{copy-array}(a_\circ,i,a,j,n,a')$ (which roughly means $a' = a_\circ\{i \triangleleft a[j]\}\cdots\{i+(n-1-j) \triangleleft a[n-1]\}$), and then simply replace $a^{\triangleright} = \text{shift-left}(a,i)$ with $\text{copy-array}(a^{\triangleright}_\circ,0,a,i,n,a^{\triangleright})$ and do similarly for $a^{\triangleright}_*$ in the definition of split-at-mut.

$$\begin{aligned}\text{copy-array}(a_\circ,i,a,j,n,a') &\iff j = n \;\wedge\; a' = a_\circ \\ \text{copy-array}(a_\circ,i,a,j,n,a') &\iff j < n \\ &\qquad\wedge\; \text{copy-array}(a_\circ\{i \triangleleft a[j]\},i+1,a,j+1,n,a')\end{aligned}$$

RefCell

Rust's `RefCell`<T> is a sharable mutable container for which borrow rules are enforced by dynamic borrowing information: *no reference*, *one mutable reference*, or *n immutable references to the resource*. The internal value of type `T` cannot be accessed directly; you dynamically borrow (mutably or immutably) from an *immutable reference* (which is sharable!) to a RefCell instance. In short, the internal state of a RefCell instance can be mutated through shared references, which is called *interior mutability* in Rust. Mutable and immutable references to RefCell internal states are obtained through special *wrapper types* (`RefMut<T>` and `Ref<T>`, respectively), because static lifetimes are not relevant and when the *wrappers* are dropped they should update the borrowing information. Since Rust's simple static analysis is often too harsh for flexible memory operations, `RefCell<T>` greatly serves for extending description ability of Rust.

For CHC-based verification supporting interior mutability provided by Rust's `RefCell<T>`, we can mix the conventional address-based approach with our novel value-based approach.

The internal state of `RefCell<T>` is represented as a pair (x, c) of value x of type `T` and a counter c; the counter can be 0 (there is no reference), -1 (there is one mutable reference), or $n > 0$ (there are n immutable references). We give an imaginary address i to each RefCell internal state, and keep a pool of RefCell bodies as an array a from imaginary addresses i to RefCell bodies (x_i, c_i); in order to manage pools of RefCell bodies efficiently, pointer analysis, as used in SeaHorn [54, 14], can be applied here. We represent a RefCell instance simply as an address i; we represent an immutable reference *wrapper* to a RefCell internal state of address i as immut_i, and a mutable reference *wrapper* as mut_i. As indicated below, RefCell operations can be expressed as a function that additionally takes a mutable reference to a relevant pool of RefCell bodies $\langle a, a_* \rangle$ or an immutable reference to it $\langle a \rangle$.

Preparation for a access. $\text{pre-borrow}(i, \langle a, a_* \rangle, r)$ (obtain a immutable reference wrapper for address i, under (a mutable reference to) a RefCell pool $\langle a, a_* \rangle$; equivalent to Rust's `borrow` for `RefCell<T>`) is expressed as a CHC as follows.

$$\begin{aligned}\text{pre-borrow}(i, \langle a, a_* \rangle, r) \iff{} & (x, c) = a[i] \ \wedge\ c \geq 0 \\ & \wedge\ a_* = a\{i \triangleleft (x, c+1)\} \ \wedge\ r = \mathrm{immut}_i\end{aligned}$$

Similarly, $\text{pre-borrow-mut}(i, \langle a, a_* \rangle, r)$ (obtain a mutable reference wrapper for address i, under a mutable reference to a pool $\langle a, a_* \rangle$; corresponding to Rust's `borrow_mut` for `RefCell<T>`) is expressed as a CHC as follows.

$$\begin{aligned}\text{pre-borrow-mut}(i, \langle a, a_* \rangle, r) \iff{} & (x, 0) = a[i] \\ & \wedge\ a_* = a\{i \triangleleft (x, -1)\} \ \wedge\ r = \mathrm{mut}_i\end{aligned}$$

Genuine access. $\text{borrow}(\langle \mathrm{immut}_i \rangle, \langle a \rangle, r)$ (obtain a genuine immutable reference through (an immutable reference to) an immutable reference wrapper $\langle \mathrm{immut}_i \rangle$ under (an immutable reference to) a pool $\langle a \rangle$; corresponding to Rust's `borrow` for `Ref<T>`) is expressed as a CHC as follows.

$$\text{borrow}(\langle \mathrm{immut}_i \rangle, \langle a \rangle, r) \iff (x, c) = a[i] \ \wedge\ r = \langle x \rangle$$

Analogously, $\text{borrow-mut}(\langle \mathrm{mut}_i \rangle, \langle a, a_* \rangle, r)$ (obtain a genuine mutable reference through (a superficial mutable reference to) a mutable reference wrapper $\langle \mathrm{mut}_i, \mathrm{mut}_i \rangle$ under (a mutable reference to) a pool $\langle a, a_* \rangle$; corresponding to Rust's `borrow_mut` for `RefMut<T>`) is expressed as a CHC as follows.

$$\begin{aligned}\text{borrow}(\langle \mathrm{mut}_i, \mathrm{mut}_i \rangle, \langle a, a_* \rangle, r) \iff{} & (x, -1) = a[i] \\ & \wedge\ r = \langle x, x_* \rangle \ \wedge\ a_* = a\{i \triangleleft x_*\}\end{aligned}$$

Note that a_* contains the future value for address i; the dynamic counting on borrowing ensures the validity of this remedy.

Drop of a reference wrapper. drop-immut(immut$_i$, $\langle a, a_* \rangle$) (drop an immutable reference wrapper immut$_i$, under (a mutable reference to) a pool $\langle a, a_* \rangle$; corresponding to Rust's `drop` for `Ref<T>`) is expressed as a CHC as follows.

$$\text{drop-immut}(\text{immut}_i, \langle a, a_* \rangle) \impliedby (x, c) = a[i] \ \land \ a_* = a\{i \triangleleft (x, c-1)\}$$

Analogously, drop-mut(mut$_i$, $\langle a, a_* \rangle$) (drop a mutable reference wrapper mut$_i$, under (a mutable reference to) a pool $\langle a, a_* \rangle$; corresponding to Rust's `drop` for `RefMut<T>`) is expressed as a CHC as follows.

$$\text{drop-mut}(\text{mut}_i, \langle a, a_* \rangle) \impliedby (x, -1) = a[i] \ \land \ a_* = a\{i \triangleleft (x, 0)\}$$

Closures

Using the technique of *defunctionalization* [40, 9], a higher-order program using closures can be transformed into a *first-order program* without closures; imitating the way closures are treated in the real-world hardware, closures are simply transformed into concrete data structures, with some identifiers assigned to function pointers.

Rust's closures are a bit complex compared to other common languages, since ownership of the data in closures should be taken care of; Rust has three types for closures, `Fn`, `FnMut` and `FnOnce`. The defunctionalization technique, nevertheless, can deal with Rust's closures, since only the concrete representation of closures matters.

The defunctionalization strategy still has a *strong limitation*: the functions that can be dealt with as closures are limited to ones already appearing in the program. If you are interested in *higher-order properties* (e.g. a general property of a higher-order function that takes two closures and composes them as a function), you can use higher-order CHCs as proposed in [7].

2.2 Examples

To demonstrate our translation, we give four examples in this section.

Letters in variable names indicate types: `p` stands for a pointer `T*` in the C language; `m` stands for a mutable reference `&'a mut T`, `l` for a list `List<T>`, and `t` for a tree `Tree<T>` in Rust.

For Example C and Example D, which entails complex recursions, we discuss useful invariants on Section 4.2.

Example A: Selecting from Mutable References

We first revisit the very simple example that is previously mentioned; it showcases notable features of our translation. We here compare our method with the conventional address-based method.

Let us write the procedures in the C programming language.

```
int* take_max(int* px, int* py) {
  if(*px >= *py) return px; else return py;
    // return the larger side of mx and my
}
struct Pair { int x, y; }; // a pair of integers
Pair inc_max(int x, int y) {
  int* pz = take_max(&x, &y); *pz += 1;
    // take the larger side as pz and increment it
    // thereby either x or y is updated
  Pair res; res.x = x; res.y = y;
  return res; // return the pair of x and y
}
```

Translating this C code into CHCs is not very straightforward. In this situation, we can obtain the following set of CHCs by the address-based conventional method (as used in SeaHorn), which gives imaginary addresses 0 and 1 to x and y respectively. Each version of the memory state of x and y is represented as an array (written as a and a'); a° stands for an uninitialized array (which can be taken just by universal quantification).

$$\begin{aligned}
\text{take-max}(px, py, r, a) &\Longleftarrow a[px] \geq a[py] \ \land\ r = px \\
\text{take-max}(px, py, r, a) &\Longleftarrow a[px] < a[py] \ \land\ r = py
\end{aligned}$$

$$\begin{aligned}
\text{inc-max}(x, y, r) \Longleftarrow\ & a = a^\circ\{0 \triangleleft x\}\{1 \triangleleft y\} \ \land\ \text{take-max}(0, 1, pz, a) \\
& \land\ a' = a\{pz \triangleleft a[pz] + 1\} \\
& \land\ r = (a'[0], a'[1])
\end{aligned}$$

The same procedures can be written in Rust as follows; since this Rust code has already appeared in the previous part of the paper, the comments are omitted here.

```
fn take_max<'a>(mx: &'a mut i32, my: &'a mut i32) -> &'a mut i32 {
  if(*mx >= *my) { mx } else { my }
}
fn inc_max(mut x: i32, mut y: i32) -> (i32, i32) {
  {
    let mx = &mut x; let my = &mut y;
    let mz = take_max(mx, my); *mz += 1;
  }
  (x, y)
}
```

As previously described, the Rust code is translated into the following CHCs by our method. The point is that our method *does not use* imaginary addresses and arrays emulating memory.

$$\begin{aligned}
\text{take-max}(\langle x, x_* \rangle, \langle y, y_* \rangle, r) &\Longleftarrow x \geq y \ \land\ y_* = y \ \land\ r = \langle x, x_* \rangle \\
\text{take-max}(\langle x, x_* \rangle, \langle y, y_* \rangle, r) &\Longleftarrow x < y \ \land\ x_* = x \ \land\ r = \langle y, y_* \rangle
\end{aligned}$$

$$\begin{aligned}
\text{inc-max}(x, y, r) \Longleftarrow\ & \text{take-max}(\langle x, x_* \rangle, \langle y, y_* \rangle, \langle l, l_* \rangle) \\
& \land\ l_* = l + 1 \ \land\ r = (x_*, y_*)
\end{aligned}$$

By unfolding take-max and rearranging formulae a little, you will get the following set of CHCs on inc-max.

$$\begin{aligned}
\text{inc-max}(x, y, r) &\Longleftarrow x \geq y \ \land\ r = (x + 1, y) \\
\text{inc-max}(x, y, r) &\Longleftarrow x < y \ \land\ r = (x, y + 1)
\end{aligned}$$

Example B: Updating a Pair via Mutable References

This example goes a little further than Example A. The main focus is put on splitting of a mutable reference to structured data.

As the previous example, we first present the procedures in C code, which does not use ownership types. `inc_max_dec_min` updates a pair of integers (of type `struct Pair`) via a pair of pointers to integers (of type `struct PtrPair`) supplied by `take_max_min`; the larger value of the pair is incremented and the smaller decremented.

```
struct Pair { int x, y; }; // a pair of integers
struct PtrPair { int *px, *py; }; // a pair of pointers to integers
struct PtrPair take_max_min(struct Pair* pp) {
  struct PtrPair res;
  if(pp->x >= pp->y) { res.px = &(pp->x); res.py = &(pp->y); }
  else { res.px = &(pp->y); res.py = &(pp->x); }
    // pp->x is a shorthand for (*pp).x, and pp->y for (*pp).y;
    // res.px points to the larger side of *pp, and res.py the smaller
  return res;
}
struct Pair inc_max_dec_min(struct Pair p) {
  struct PtrPair q = take_max_min(&p);
  *(q.px) += 1; *(q.py) -= 1;
    // increment the referent of q.px, and decrement of q.py
  return p;
}
```

This C code is translated into the following set of CHCs by the conventional method. Each version of the memory state of the integer pair is expressed as an array (written as a, a' and a'').

$$\begin{aligned}
\text{take-max-min}(px, py, r.a) &\Longleftarrow a[px] \geq a[py] \ \wedge\ r = (px, py) \\
\text{take-max-min}(px, py, r.a) &\Longleftarrow a[px] < a[py] \ \wedge\ r = (py, px)
\end{aligned}$$

$$\begin{aligned}
\text{inc-max-dec-min}((x, y), r) \Longleftarrow\ & a = a^\circ\{0 \triangleleft x\}\{1 \triangleleft y\} \\
\wedge\ & \text{take-max-min}(0, 1, (pz, pw), a) \\
\wedge\ & a' = a\{pz \triangleleft a[pz] + 1\} \\
\wedge\ & a'' = a'\{pw \triangleleft a'[pw] - 1\} \ \wedge\ r = (a''[0], a''[1])
\end{aligned}$$

This program can be expressed in Rust as follows. The following Rust code contains the four types of operations on mutual references: creating, splitting, updating and returning.

```
fn take_max_min<'a>(mp: &'a mut (i32, i32))
    -> (&'a mut i32, &'a mut i32) {
  let (mx, my) = mp; // simply split the mutable reference mp
  if *mx >= *my { (mx, my) } else { (my, mx) }
    // put the larger side on the left
}
fn inc_max_dec_min(mut p: (i32, i32)) -> (i32, i32) {
  {
    let mp: &mut (i32, i32) = &mut p; // mutably borrow the pair p
    let (mz, mw) = take_max_min(mp);
      // split mp into the larger and the smaller sides
    *mz += 1; *mw -= 1;
      // increment the larger value, and decrement the smaller
  } // end borrowing p as mp
  p
}
```

By our translation, you will get a set of CHCs like this. Note that, in definition of take-max-min, a mutable reference to a pair of integers $\langle (x, y), (x_*, y_*) \rangle$ is split into two mutable references to an integer $\langle x, x_* \rangle$ and $\langle y, y_* \rangle$.

$$\begin{aligned}
\text{take-max-min}\big(\langle (x, y), p_* \rangle,\ r\big) &\Longleftarrow (x_*, y_*) = p_* \ \wedge\ x \geq y \ \wedge\ r = (\langle x, x_* \rangle, \langle y, y_* \rangle) \\
\text{take-max-min}\big(\langle (x, y), p_* \rangle,\ r\big) &\Longleftarrow (x_*, y_*) = p_* \ \wedge\ x < y \ \wedge\ r = (\langle y, y_* \rangle, \langle x, x_* \rangle)
\end{aligned}$$

$$\begin{aligned}
\text{inc-max-dec-min}(p, r) \Longleftarrow\ & \text{take-max-min}\big(\langle p, p_* \rangle,\ (\langle z, z_* \rangle, \langle w, w_* \rangle)\big) \\
\wedge\ & z_* = z + 1 \ \wedge\ w_* = w - 1 \ \wedge\ r = p_*
\end{aligned}$$

By unfolding take-max-min and rearranging formulae a little, you will get the following set of CHCs on inc-max-dec-min.

$$\text{inc-max-dec-min}\big((x,y),(x_*,y_*)\big) \impliedby x \ge y \ \land\ x_* = x+1 \ \land\ y_* = y-1$$
$$\text{inc-max-dec-min}\big((x,y),(x_*,y_*)\big) \impliedby x < y \ \land\ x_* = x-1 \ \land\ y_* = y+1$$

Example C: Update of a List of Integers via a List of Mutable References

The following Rust code uses a *(singly linked) list of mutable references* to update a list of integers. `sort_carve_list` takes a list of integers `lx` and decreases the i-th smallest element of `lx` by $i-1$ for each i. For that purpose, it takes out a list of mutable references to the elements of `lx` through `split_mut_list`, and sort the new list by `sort_list`.[1]

```
#![feature(box_syntax, box_patterns)]
enum List<T> { Nil, Cons(T, Box<List<T>>) }
  // recursive data type for a list;
  // Box<> is used for finiteness of the type size
fn split_mut_list<'a>(mlx: &'a mut List<i32>) -> List<&'a mut i32> {
  // given a mutable reference to a list,
  // split them into mutable references to its elements,
  // and wrap them in a list
  match mlx {
    List::Nil => List::Nil,
    List::Cons(mx, mlx2) => List::Cons(mx, box split_mut_list(mlx2))
  }
}
fn insert_list<'a>(mx: &'a mut i32, lmy: List<&'a mut i32>)
    -> List<&'a mut i32> {
  // helper function for sort_list;
  // insert mx into a sorted list lmx
  match lmy {
    List::Nil => List::Cons(mx, box List::Nil),
    List::Cons(my, box lmy2) =>
      if *mx <= *my { List::Cons(mx, box List::Cons(my, box lmy2)) }
      else { List::Cons(my, box insert_list(mx, lmy2)) }
  }
}
fn sort_list<'a>(lmx: List<&'a mut i32>) -> List<&'a mut i32> {
  // sort a list of mutable references by referent values
  match lmx {
    List::Nil => List::Nil,
    List::Cons(mx, box lmx2) => insert_list(mx, sort_list(lmx2))
  }
}
fn carve_list<'a>(i: i32, lmx: List<&'a mut i32>) {
  // decrease referents of list lmx by i, i+1, i+2, ... respectively
  match lmx {
    List::Nil => {}
    List::Cons(mx, box lmx2) => { *mx -= i; carve_list(i+1, lmx2) }
  }
}
fn sort_carve_list(mut lx: List<i32>) -> List<i32> {
  // decrease the i-th smallest element of lx by i-1 for each i
  carve_list(0, sort_list(split_mut_list(&mut lx)));
    // temporarily mutably borrow lx
  lx
}
```

This code is translated into the following set of CHCs by our translation. Operations on lists are written in the Prolog style: $[x_1, \dots, x_n]$ is an extensional

[1]Since apostrophe ’ cannot be used for variable names in Rust, 2 is added at the end instead.

notation (particularly [] is a nil), and $[x|lx]$ is a cons of head x and tail lx.

$$\begin{aligned}
\text{split-mut-list}(\langle [], lx_* \rangle, r) &\Longleftarrow lx_* = [] \;\land\; r = [] \\
\text{split-mut-list}(\langle [x|lx'], lx_* \rangle, r) &\Longleftarrow lx_* = [x_*|lx'_*] \\
&\quad \land\; \text{split-mut-list}(\langle lx', lx'_* \rangle, r') \\
&\quad \land\; r = [\langle x, x_* \rangle | r']
\end{aligned}$$

$$\begin{aligned}
\text{insert-list}(mx, [], r) &\Longleftarrow r = [mx] \\
\text{insert-list}(\langle x, x_* \rangle, [\langle y, y_* \rangle | lmy'], r) &\Longleftarrow x \geq y \;\land\; r = [\langle x, x_* \rangle | [\langle y, y_* \rangle | lmy']] \\
\text{insert-list}(\langle x, x_* \rangle, [\langle y, y_* \rangle | lmy'], r) &\Longleftarrow x < y \\
&\quad \land\; \text{insert-list}([\langle x, x_* \rangle | lmy'], r') \\
&\quad \land\; r = [my | r']
\end{aligned}$$

$$\begin{aligned}
\text{sort-list}([], r) &\Longleftarrow r = [] \\
\text{sort-list}([mx|lmx'], r) &\Longleftarrow \text{sort-list}(lmx', r') \;\land\; \text{insert-list}(mx, r', r)
\end{aligned}$$

$$\begin{aligned}
\text{carve-list}(i, []) &\Longleftarrow \top \\
\text{carve-list}(i, [\langle x, x_* \rangle | lmx']) &\Longleftarrow x_* = x - i \;\land\; \text{carve-list}(i + 1, lmx')
\end{aligned}$$

$$\begin{aligned}
\text{sort-carve-list}(lx, r) &\Longleftarrow \text{split-mut-list}(\langle lx, lx_* \rangle, r') \\
&\quad \land\; \text{sort-list}(r', r'') \;\land\; \text{carve-list}(r'') \;\land\; r = lx_*
\end{aligned}$$

Each element is separately treated by the pair representing the mutable reference. *Unboundedly many mutable references are scrambled* by sorting, but by our method updates on mutable references are naturally reflected into the value after borrowing.

With the use of clever, non-trivial techniques (for *automated* verification), the Rust code can also be expressed in CHCs as follows in an address-based manner: the list of mutable references are expressed as a *list of addresses*. For distinction, we give suffix '-a' to each uninterpreted predicate.

split-mut-list-a(i, lx, r) represents a function that receives a list of integers lx with an auxiliary argument i, and returns the tuple of the length of the list, the array that maps addresses to integers, and the list of addresses; elements of lx are given address $i, i + 1, i + 2, \ldots$ respectively. a° represents an uninitialized array (simply taken by universal quantification).

We use an additional uninterpreted predicate restore-list-a(i, l, a, r), which represents a function that takes the target index l and the array a with an auxiliary argument i, and obtains a list of integers with values $a[i], a[i + 1], \ldots, a[l - 1]$.

$$\begin{aligned}
\text{split-mut-list-a}(0, [], r) &\Longleftarrow r = (0, [], a^\circ) \\
\text{split-mut-list-a}(i, [x|lx], r) &\Longleftarrow \text{split-mut-list-a}(i + 1, lx, (l', lp', a')) \\
&\quad \land\; r = (l' + 1, [i|lp'], a'\{i \triangleleft x\})
\end{aligned}$$

$$\begin{aligned}
\text{insert-list-a}(p, [], r, a) &\Longleftarrow r = [p] \\
\text{insert-list-a}(p, [q|lq], r, a) &\Longleftarrow a[p] \leq a[q] \;\land\; r = [p|[q|lq]] \\
\text{insert-list-a}(p, [q|lq], r, a) &\Longleftarrow a[p] > a[q] \;\land\; \text{insert-list-a}(p, lq, r', a) \\
&\quad \land\; r = [q|r']
\end{aligned}$$

$$\begin{aligned}
\text{sort-list-a}([], r, a) &\Longleftarrow r = [] \\
\text{sort-list-a}([p|lp], r, a) &\Longleftarrow \text{sort-list-a}(lp, r', a) \;\land\; \text{insert-list-a}(p, r', r, a)
\end{aligned}$$

$$\begin{aligned}
\text{carve-list-a}(i, [], a, a_*) &\Longleftarrow a_* = a \\
\text{carve-list-a}(i, [p|lp], a, a_*) &\Longleftarrow \text{carve-list-a}(i + 1, a\{i \triangleleft a[p] - i\}, lp, a_*)
\end{aligned}$$

$$\begin{aligned}
\text{restore-list-a}(i,l,r,a) &\Longleftarrow i = l \ \wedge\ r = [\,] \\
\text{restore-list-a}(i,l,r,a) &\Longleftarrow i < l \ \wedge\ \text{restore-list-a}(i+1,l,r',a) \ \wedge\ r = [a[i]|r']
\end{aligned}$$

$$\begin{aligned}
\text{sort-carve-list-a}(lx,r) \Longleftarrow\ & \text{split-mut-list-a}(0,lx,(l,lp,a)) \\
\wedge\ & \text{sort-list-a}(lp,lq,a) \\
\wedge\ & \text{carve-list-a}(0,lq,a,a_*) \\
\wedge\ & \text{restore-list-a}(0,l,r,a_*)
\end{aligned}$$

Example D: Update of a Tree of Integers via a List of Mutable References

In this example, a *(binary) tree of integers* is updated using a list of mutable references. For these kinds of operations on trees, the traditional address-based translation is very likely to generate complicated CHCs, since it is rather hard to restore a tree of integers from an array simulating the memory state.

Consider the following Rust code. `carve_bfs_tree` visits elements on a tree by breadth first search and decreases them by $0, 1, 2, \ldots$ respectively; it uses `go_tree` as a helper recursive function. For breadth first search, it uses purely functional queue with operations `push_queue` and `pop_queue`. For simplicity of types, the queue operations are described under type polymorphism on the element type. Queue is implemented as a basic purely functional data structure known as "Banker's Queue", for which queue operations work in amortized linear time; for more information, see [37].

```
#![feature(box_syntax, box_patterns)]
enum List<T> { Nil, Cons(T, Box<List<T>>) }
  // recursive data type for a list
type Queue<T> = (List<T>, List<T>);
  // Banker's Queue;
  // ([x1,...,xm], [y1,...,yn]) represents a queue
  // of elements x1,...,xm,yn,...,y1
fn push_queue<T>(que: Queue<T>, z: T) -> Queue<T> {
  // push an element to the back of a queue
  let (lx, ly) = que; (lx, List::Cons(z, box ly))
}
fn reverse_list<T>(lx: List<T>, ly: List<T>) -> List<T> {
  // reverse_list([x1,...,xm],[y1,...,yn])
  // is equal to [xm,...,x1,y1,...,yn]
  match lx {
    List::Nil => ly,
    List::Cons(x, box lx2) => reverse_list(lx2, List::Cons(x, box ly))
  }
}
fn pop_queue<T>(que: Queue<T>) -> Option<(T, Queue<T>)> {
  // pop an element from the front of a queue
  // and return it with the remaining queue;
  // None is returned when que is empty
  let (lx, ly) = que;
  match lx {
    List::Nil => match reverse_list(ly, List::Nil) {
      List::Nil => None,
      List::Cons(y, box ly2) => Some((y, (ly2, List::Nil)))
    },
    List::Cons(x, box lx2) => Some((x, (lx2, ly)))
  }
}
enum Tree<T> { Leaf, Node(Box<Tree<T>>, T, Box<Tree<T>>) }
  // recursive data type for a tree
fn go_tree<'a>(i: i32, que: Queue<&'a mut Tree<i32>>) {
  // helper function for carve_bfs_tree;
  // progress breadth first search on a tree using a queue
  // and decrease elements by i, i+1, i+2, ... respectively
  match pop_queue(que) {
    None => {},
    Some((mtx, que2)) => match mtx {
      Tree::Leaf => go_tree(i, que2),
      Tree::Node(mty, mx, mtz) => {
        *mx -= i;
        go_tree(i + 1, push_queue(push_queue(que2, mty), mtz))
      }
    }
  }
}
fn carve_bfs_tree(mut tx: Tree<i32>) -> Tree<i32> {
  // visit elements on a tree by breadth first search
  // and decrease them by 0, 1, 2, ... respectively
  go_tree(0, push_queue((List::Nil, List::Nil), &mut tx));
    // temporarily mutably borrow tx
    // and start go_tree with a single-element queue
  tx
}
```

This code is translated into the following set of CHCs. Data constructors None, Some, Leaf and Node are used in the description; note that Rust's `Box<T>` type is ignored in representing trees.

$$\text{push-queue}\big((lx, ly), z, r\big) \iff r = (lx, [z, ly])$$

$$\begin{aligned}
\text{reverse-list}([], ly, r) &\iff r = ly \\
\text{reverse-list}([x|lx'], ly, r) &\iff \text{reverse-list}(lx', [x|ly], r)
\end{aligned}$$

$$\begin{aligned}
\text{pop-queue}\big(([], ly), r\big) &\iff \text{reverse-list}(ly, [], []) \;\wedge\; r = \text{None} \\
\text{pop-queue}\big(([], ly), r\big) &\iff \text{reverse-list}(ly, [], [y|ly']) \\
&\qquad \wedge\; r = \text{Some}\big((y, (ly', []))\big) \\
\text{pop-queue}\big(([x|lx'], ly), r\big) &\iff r = (x, (lx', ly))
\end{aligned}$$

$$\begin{aligned}
\text{go-tree}(i, que) &\iff \text{pop-queue}(que, \text{None}) \\
\text{go-tree}(i, que) &\iff \text{pop-queue}\big(que, \text{Some}(\langle \text{Leaf}, \text{Leaf}\rangle, que')\big) \\
&\qquad \wedge\; \text{go-tree}(i, que') \\
\text{go-tree}(i, que) &\iff \text{pop-queue}\big(que, \\
&\qquad\quad \text{Some}\big(\langle \text{Node}(ty, x, tz), \text{Node}(ty_*, x_*, tz_*)\rangle, que'\big)\big) \\
&\qquad \wedge\; x_* = x - i \\
&\qquad \wedge\; \text{push-queue}(que', \langle ty, ty_*\rangle, que'') \\
&\qquad \wedge\; \text{push-queue}(que'', \langle tz, tz_*\rangle, que''') \\
&\qquad \wedge\; \text{go-tree}(i+1, que''')
\end{aligned}$$

$$\text{carve-bfs-tree}(tx, r) \iff \text{go-tree}\big(0, \text{push-queue}\big(([], []), \langle tx, tx_*\rangle\big)\big) \;\wedge\; r = tx_*$$

Chapter 3

Formalization of the Translation

In this chapter, our translation is fully formalized under a new formal language *Calculus of Ownership and Reference* (COR) corresponding to a basic subset of Rust, and a conjecture on the correctness of the translation is presented.

3.1 Calculus of Ownership and Reference

Real-world Rust has many complex features. A number of formalizations of Rust have been proposed such as Patina [39] and λ_{Rust} [25]; however, they are still somewhat complex for our purpose.

We define here *Calculus of Ownership and Reference* (COR) to clarify our translation of programs with ownership types into sets of CHCs. COR corresponds to a basic subset of Rust, *including* ownership, (re)borrowing, lifetimes (along with lifetime polymorphism), **Box**<T>, tuples, sum types, recursive types, and basic arithmetic, *excluding* particularly uninitialized variables, slices and vectors, interior mutability, closures,[1] traits (analogous to Java's interfaces), concurrency, unsafe blocks, auto-(de)referencing, type polymorphism, type inference, lifetime inference and non-lexical lifetimes.[2]

COR is designed so that the timings of (re)borrowing and variable dropping are explicitly described, which clarifies our translation later. We provide syntax, examples, type checking, and operational semantics of COR in this section.

3.1.1 Syntax

Lifetime and Type[3][4]

$$\alpha, \beta \ ::= \ \text{(lifetime variable)}$$

$$\text{(lifetime)}\ \psi, \varphi \ ::= \ \alpha \ \mid \ \wedge(\psi_1, \ldots, \psi_n)\ \text{(earliest of lifetimes)}$$

$$\text{(reference authority)}\ B \ ::= \ \textbf{mut}\ \text{(mutable)} \ \mid \ \textbf{immut}\ \text{(immutable)}$$

$$\text{(pointer authority)}\ A \ ::= \ \textbf{own}\ \text{(owning pointer)} \ \mid \ B_\psi\ \text{(reference)}$$

$$X, Y \ ::= \ \text{(type variable)}$$

[1]As discussed in Section 2.1.2, we can extend our formalization for slices and vectors, RefCell, and closures. They are ignored here to keep the formalization clear and simple.

[2]With non-lexical lifetimes recently introduced in Rust, lifetimes should be regarded as a *set of program points*; thus switching of ownership may happen multiple times for each lifetime (or each (re)borrowing), which obfuscates our translation to CHCs.

[3]The symbol ψ comes from Ancient Greek ψυχή (life, soul, mind), the origin of Latin psychē.

[4]A lifetime $\wedge(\psi_1, \ldots, \psi_n)$ can be written as $\psi_1 \wedge \cdots \wedge \psi_n$. A type $+(T_1, \ldots, T_n)$ can be written as $T_1 + \cdots + T_n$ and $\sum_{i=1}^{n} T_i$, and $\times(T_1, \ldots, T_n)$ as $T_1 \times \cdots \times T_n$ and $\prod_{i=1}^{n} T_i$.

$$\text{(type) } T ::= X \mid \textbf{int} \text{ (integer)} \mid A\,T \text{ (pointer)} \mid \mu X.T \text{ (recursive type)}$$
$$\mid +(T_1, \dots, T_n) \text{ (sum)} \mid \times(T_1, \dots, T_n) \text{ (product)}$$

Lifetimes. In our terminology on COR, lifetimes are treated as a *time point* (*not* time range) that can indicate *the end of some (re)borrowing*; this is slightly different from how lifetimes are treated in the standard Rust terminology. Unlike Rust, lifetime variables are all explicitly named. As indicated later in type-checking rules, lifetime variables are introduced either by instruction **intro** α (as a *local lifetime variable*; local within a function) or by a function signature (as a *lifetime parameter*).

Pointers. An *owning pointer* **own** T represents full ownership of a region in the heap and corresponds to Rust's `box T`.[5] A *reference* $B_\psi\,T$ represents temporary access (valid until lifetime ψ) to a region in the heap that is owned by an owning pointer **own** T; there are two types of references, a *mutable reference* $\textbf{mut}_\psi\,T$ (which corresponds to Rust's `&'psi mut T`) and an *immutable reference* $\textbf{immut}_\psi\,T$ (which corresponds to Rust's `&'psi T`).

Recursive types. We call a type not used as a substructure of a larger type (e.g. variable types and function return types) a *complete type*. For any complete type, every appearance of type variables should be bound by μ, and also *guarded by a pointer type* (i.e. contained in a pointer type) for the sake of *finiteness and definiteness of type sizes*. Types such as $\mu X.\,\textbf{int} + X$ (which has an infinite size) and $\mu X.X$ (which has an indefinite size) are *prohibited*.

Convenient notation. The type $\times()$, or the *unit type*, can be written as $*$. In addition, we can write $* + *$, or the *boolean type*, as **bool**.

Instruction[6]

$$x, y, z ::= \text{((data) variable)}$$

$$f ::= \text{(function name)}$$

$$\text{(binary operation on integers) } o ::= + \mid - \mid \cdot \mid \cdots$$

$$\text{(binary relation on integers) } r ::= \leq \mid \geq \mid \cdots$$

$$\begin{aligned}
\text{(instruction) } I ::=\ & \textbf{let}\ y = n \text{ (creating an integer)} \\
\mid\ & \textbf{let}\ y = x\ o\ x' \text{ (binary operation on integers)} \\
\mid\ & \textbf{let}\ y = x\ r\ x' \text{ (binary relation on integers)} \\
\mid\ & \textbf{let}\ y = x \text{ (renaming)} \mid \textbf{let}\ y = \textbf{copy}\ x \text{ (copying)} \\
\mid\ & \textbf{let}\ y = \textbf{repl}\ x \text{ (replicating an immutable reference)} \\
\mid\ & \textbf{let}\ y = \textbf{mut bor}\ x\ \textbf{till}\ \psi \text{ ((re)borrowing a mutable reference)} \\
\mid\ & \textbf{let}\ y = \textbf{ref}\ x \text{ (creating a box)} \mid \textbf{let}\ y = \textbf{deref}\ x \text{ (dereferencing)} \\
\mid\ & \textbf{let}\ y = \textbf{inj}_i^T\ x \text{ (creating a tagged data)} \\
\mid\ & \textbf{let}\ y = (x_1, \dots, x_n) \text{ (creating a tuple)} \\
\mid\ & \textbf{let}\ (y_1, \dots, y_n) = x \text{ (splitting a tuple)} \\
\mid\ & \textbf{let}\ y = f\langle \psi_1, \dots, \psi_m \rangle (x_1, \dots, x_n) \text{ (function call)} \\
\mid\ & \textbf{swap}(x, y) \text{ (swapping referents of pointers)} \\
\mid\ & \textbf{drop}\ x \text{ (dropping a variable)} \\
\mid\ & \textbf{immut}\ x \text{ (altering a mutable reference into an immutable reference)}
\end{aligned}$$

[5]We simply assume that all non-address data is allocated in the heap and that the stack just traces addresses of variables.

[6]Variables appearing in an instruction should be mutually distinct (for simplicity of formal definitions).

$\mid\ x\ \mathbf{as}\ A\ T$ (retyping)
$\mid\ \mathbf{intro}\ \alpha$ (introducing a local lifetime variable)
$\mid\ \mathbf{now}\ \alpha$ (eliminating a local lifetime variable)

An instruction represents an atomic operation. In order to simplify type-checking rules, nested expressions (in usual languages) are decomposed into sequences of instructions. Here are a few notes on instructions.

Variables as pointers. Unlike Rust, every variable is a *pointer* (an owning pointer or a reference), but it *behaves as its referent* on the surface. In addition, arguments of a function call are *pointers*.[7] Reference instruction $\mathbf{let}\ y = \mathbf{ref}\ x$ creates an *owning pointer to a pointer*, and dereference instruction $\mathbf{let}\ y = \mathbf{deref}\ x$ dereferences a *pointer to a pointer*.

Let binding. As indicated later in type-checking rules, variables are introduced just by let-binding instructions (i.e. instructions of form $\mathbf{let} \cdots = \cdots$). Variables on the right-hand side of let-binding instructions are eliminated, except for $\mathbf{let}\ y = x\ o\ x'$, $\mathbf{let}\ y = x\ r\ x'$, $\mathbf{let}\ y = \mathbf{copy}\ x$ and $\mathbf{let}\ y = \mathbf{repl}\ x$. In particular, through a function call $\mathbf{let}\ y = f\langle\cdots\rangle(x_1, \ldots, x_n)$, the pointers $x_1, \ldots, x_n$ passed as arguments are taken away by the called function.

Drop. Variables are also eliminated by $\mathbf{drop}\ x$ (equivalent to Rust's `mem::drop(x)`). When dropping an owning pointer, the acquired data in the heap is released. Each reference should be dropped before the advent of the binding lifetime. Unlike Rust, the dropping of variables is explicitly stated.

Local lifetime variables. They are introduced by $\mathbf{intro}\ \alpha$ (α is introduced as a new local lifetime variable *strictly smaller* than any existing local lifetime variables) and eliminated by $\mathbf{now}\ \alpha$ before returning from a function. Local lifetime variables are strictly smaller than lifetime parameters, since lifetime parameters outlive the function.

Swap as a destructive update. Destructive updates are attained merely by $\mathbf{swap}(x, y)$ (equivalent to Rust's `mem::swap(x,y)`). As indicated later in type-checking rules, x should be a mutable reference, and y can be either an owning pointer or a mutable reference.[8] Swapping is adopted as a primitive instead of substitution mainly because substitution involves *dropping of the original value*; in addition, swapping is difficult to realize with substitution in a situation where uninitialized variables are not allowed.

Immutable references. An immutable reference y can be (re)borrowed by the following steps: (1) (re)borrow a mutable reference by $\mathbf{let}\ y = \mathbf{mut\ bor}\ x\ \mathbf{till}\ \psi$, and (2) alter it to an immutable reference by $\mathbf{immut}\ y$. An immutable reference can be replicated by $\mathbf{let}\ y = \mathbf{repl}\ x$,[9] which is different from instruction $\mathbf{let}\ y = \mathbf{copy}\ x$, which copies a referent on the memory and creates an owning pointer.

[7]This is similar to but slightly different from λ_{Rust}, where every variable is either a pointer or an *integer value*, and arguments of a function call are *owning pointers*.

[8]We do not need swap for two owning pointers, since the same effect is given by swapping variable names using renaming instruction $\mathbf{let}\ y = x$. Swap for a mutable reference and an owning pointer can also be excluded, since it can be reduced to swapping for two mutable references by temporarily borrowing an owning pointer.

[9]Replication instruction can be excluded, since the same effect can be achieved by the following sequential execution of instructions: $\mathbf{let}\ ox = \mathbf{ref}\ x$; $\mathbf{let}\ oy = \mathbf{copy}\ ox$; $\mathbf{let}\ x = \mathbf{deref}\ ox$; $\mathbf{let}\ y = \mathbf{deref}\ oy$.

Large Structures[10]

$$L \ ::= \ \text{(label)}$$

$$\text{(possibly claiming a variable)}\ \tilde{x}, \tilde{y} \ ::= \ x \ \text{(claiming)} \ \mid \ - \ \text{(not claiming)}$$

$$\begin{aligned}\text{(statement)}\ S \ ::= \ & I;\ \textbf{goto}\, L \ \text{(instruction)} \ \mid \ \textbf{return}\, x \ \text{(returning a value)} \\ \mid \ & \textbf{match}\, x\, \{\textbf{inj}_1 \tilde{y}_1 \colon \textbf{goto}\, L_1;\ \ldots;\ \textbf{inj}_n \tilde{y}_n \colon \textbf{goto}\, L_n\} \ \text{(branching by a tag)}\end{aligned}$$

$$\begin{aligned}\text{(function signature)}\ \Sigma \ ::= \ & \langle \alpha_1, \ldots, \alpha_m \mid \alpha_{a_1} \le \alpha_{b_1}, \ldots, \alpha_{a_{m'}} \le \alpha_{b_{m'}} \rangle \\ & (x_1 \colon A_1\, T_1, \ldots, x_n \colon A_n\, T_n) \to A'\, T'\end{aligned}$$

$$\text{(function)}\ F \ ::= \ \textbf{fn}\, f\, \Sigma\, \{\textbf{entry} \colon S_0\ L_1 \colon S_1\ \cdots\ L_n \colon S_n\}$$

$$\text{(program)}\ P \ ::= \ F_1 \cdots F_n$$

Labels. Control flows (such as sequential executions, branching and loops) are expressed solely with labels and goto directions for simplicity of formal description. Each label can be regarded as a *program point*, and also as *continuation* under a fixed context of typed variables.[11] Continuations behave similarly to functions, but we support continuations separately because functions work as a boundary of (re)borrowing, lifetime parameters and local lifetime variables.

Function signatures. In a function signature, the former part $\langle \alpha_1, \ldots, \alpha_m \mid \alpha_{a_1} \le \alpha_{b_1}, \ldots, \alpha_{a_{m'}} \le \alpha_{b_{m'}} \rangle$ introduces lifetime parameters along with constraints on them; and all lifetime variables appearing in the latter part $(x_1 \colon A_n\, T_n, \ldots, x_n \colon A_n\, T_n) \to A'\, T'$ should be introduced by the preceding part.

Function bodies. A function has one or more labeled statements in its body $\{\cdots\}$; the first one with label **entry** is the entry point.

3.1.2 Examples

Syntax Sugar on Labels, Instructions and Statements

The following syntax sugar on labels, instructions and statements is used for writing examples; it is not used for other formal descriptions but helps to understand COR a lot.

Labels of statements can be omitted when not referred to by **goto**. Sequential executions can be indicated by semicolons: $I_1; \cdots; I_n;\ S$ in place of a statement is a shorthand for $I_1; \textbf{goto}\, L_2 \quad L_2 \colon I_2; \textbf{goto}\, L_3 \ \cdots \ L_n \colon I_n; \textbf{goto}\, L_{n+1} \quad L_{n+1} \colon S$ (for some fresh labels $L_2, \ldots, L_{n+1}$). In addition, $\textbf{match}\, x\, \{\textbf{inj}_1 \hat{y}_1 \colon S_1 \ \cdots \ \textbf{inj}_n \hat{y}_n \colon S_n\}$ is a shorthand for $\textbf{match}\, x\, \{\textbf{inj}_1 \hat{y}_1 \colon \textbf{goto}\, L_1 \ \cdots \ \textbf{inj}_n \hat{y}_n \colon \textbf{goto}\, L_n\} \quad L_1 \colon S_1 \ \cdots \ L_n \colon S_n$ (for some fresh labels $L_1, \ldots, L_n$; here statements $S_1, \ldots, S_n$ may contain aforementioned sequential executions).

Examples

To demonstrate the syntax of our language, we translate functions written in Rust from Section 2.2 into COR, using the syntax sugar introduced above. In the following examples, List T is a shorthand for $\mu X. * + T \times \textbf{own}\, X$, and Cons T for $T \times \textbf{own}\, \text{List}\, T$. Letters in variable names indicate types: m stands for $\textbf{mut}_\alpha$, o for **own**, l for List, c for Cons, b for **bool**, and u for $*$.

[10] In a function, the names of lifetime parameters should be mutually distinct, and so should the names of labels. In a program, the names of functions should be mutually distinct.

[11] This kind of continuation corresponds to a continuation in λ_{Rust} introduced by `letcont`, though sequential executions are separately supported in λ_{Rust}.

The functions `take_max` and `inc_max` in Example A can be written in COR as follows.

```
$\textbf{fn}$ take-max$\langle \alpha | \rangle (mx\colon \textbf{mut}_\alpha\ \textbf{int}, my\colon \textbf{mut}_\alpha\ \textbf{int}) \to \textbf{mut}_\alpha\ \textbf{int}$ {
  $\textbf{let}\ ob = mx \geq my;$
  $\textbf{match}\ ob$ {
    $\textbf{inj}_0 -\colon\ \textbf{drop}\ ob;\ \textbf{drop}\ my;\ \textbf{return}\ mx$
    $\textbf{inj}_1 -\colon\ \textbf{drop}\ ob;\ \textbf{drop}\ mx;\ \textbf{return}\ my$
  }
}
$\textbf{fn}$ inc-max$\langle | \rangle (ox\colon \textbf{own int}, oy\colon \textbf{own int}) \to \textbf{own}\ (\textbf{int} \times \textbf{int})$ {
  $\textbf{intro}\ \alpha;\ \textbf{let}\ mx = \textbf{mut bor}\ ox\ \textbf{till}\ \alpha;\ \textbf{let}\ my = \textbf{mut bor}\ oy\ \textbf{till}\ \alpha;$
  $\textbf{let}\ ml = $ take-max$\langle \alpha \rangle (mx, my);$
  $\textbf{let}\ o1 = 1;\ \textbf{let}\ ol' = ml + o1;\ \textbf{drop}\ \ o1;\ \textbf{swap}(ml, ol');\ \textbf{drop}\ ol';$
  $\textbf{now}\ \alpha;\ \textbf{let}\ or = (ox, oy)\ \ \textbf{drop}\ ox;\ \textbf{drop}\ oy;\ \textbf{return}\ or$
}
```

Without sugar syntax, the function take-max looks like this.

```
$\textbf{fn}$ take-max$\langle \alpha | \rangle (mx\colon \textbf{mut}_\alpha\ \textbf{int}, my\colon \textbf{mut}_\alpha\ \textbf{int}) \to \textbf{mut}_\alpha\ \textbf{int}$ {
  $\textbf{entry}\colon \textbf{let}\ ob = mx \geq my; \textbf{goto}\ L^{\text{match}}$
  $L^{\text{match}}\colon \textbf{match}\ ob\ \{\ \textbf{inj}_0 -\colon\ \textbf{goto}\ L_0^x\ \ \textbf{inj}_1 -\colon\ \textbf{goto}\ L_0^y\ \}$
  $L_0^x\colon \textbf{drop}\ ob; \textbf{goto}\ L_1^x\ \ L_1^x\colon \textbf{drop}\ my; \textbf{goto}\ L_2^x\ \ L_2^x\colon \textbf{return}\ mx$
  $L_0^y\colon \textbf{drop}\ ob; \textbf{goto}\ L_1^y\ \ L_1^y\colon \textbf{drop}\ mx; \textbf{goto}\ L_2^y\ \ L_2^y\colon \textbf{return}\ my$
}
```

The function `split_mut_list` in Example C can be written in COR as follows.

```
$\textbf{fn}$ split-mut-list$\langle \alpha | \rangle (mlx\colon \textbf{mut}_\alpha\ \text{List}\ \textbf{int}) \to \textbf{own}\ \text{List}\ \textbf{mut}_\alpha\ \textbf{int}$ {
  $mlx\ \textbf{as}\ \textbf{mut}_\alpha (* + \text{Cons}\ \textbf{int});$
  $\textbf{match}\ mlx$ {
    $\textbf{inj}_0 -\colon\ \textbf{drop}\ mlx;$
      $\textbf{let}\ ou = ();\ \textbf{let}\ olmx = \textbf{inj}_0^{* + \text{Cons}\ \textbf{mut}_\alpha\ \textbf{int}} ou;\ olmx\ \textbf{as own}\ \text{List}\ \textbf{mut}_\alpha\ \textbf{int};$
      $\textbf{return}\ olmx$
    $\textbf{inj}_1\, mcx\colon$
      $\textbf{let}\ (mx, molx') = mcx;\ \textbf{let}\ omx = \textbf{ref}\ mx;$
      $\textbf{let}\ mlx' = \textbf{deref}\ molx';\ \textbf{let}\ olmx' = $ split-mut-list$\langle \alpha \rangle (mlx');$
      $\textbf{let}\ oolmx' = \textbf{ref}\ olmx';\ \textbf{let}\ ocmx = (omx, oolmx');$
      $\textbf{let}\ olmx = \textbf{inj}_1^{* + \text{Cons}\ \textbf{mut}_\alpha\ \textbf{int}} ocmx;\ olmx\ \textbf{as own}\ \text{List}\ \textbf{mut}_\alpha\ \textbf{int};$
      $\textbf{return}\ olmx$
  }
}
```

3.1.3 Type Checking

Contexts

The following five types of contexts are used to describe type-checking rules of COR.

The *function signature context* $\boldsymbol{\Sigma}$ of a program $P = F_1 \cdots F_n$ is the mapping that maps the function name of F_i to the function signature of F_i. It is used for type checking on function call instruction $\textbf{let}\ y = f \langle \psi_1, \ldots, \psi_m \rangle (x_1, \ldots, x_n)$.

The *lifetime parameter context* Ψ_{ex} of a function $f\langle\alpha_1,\ldots,\alpha_m|\cdots\rangle(\cdots)\{\cdots\}$ is the finite set of the lifetime parameters $\{\alpha_1,\ldots,\alpha_m\}$. It is used to distinguish local lifetime variables from lifetime parameters for type checking on lifetime-related instructions **intro** α and **now** α.[12]

A *lifetime context* $\boldsymbol{\Psi}$ is a finite preordered set of lifetime variables (Ψ, R) (where Ψ is the underlying set and R the preorder relation on Ψ). It represents the set of lifetime variables (local lifetime variables and lifetime parameters) with constraints on them provided at each program point (or label).

A *variable context* $\boldsymbol{\Gamma}$ is a finite set of elements of form $x{:}A\,T$ (x is *active* and has type $A\,T$) and of form $x{:}^{\psi} A\,T$ (x is *shadowed* until ψ and has type $A\,T$), satisfying the condition that all variable names in $\boldsymbol{\Gamma}$ are mutually distinct. A variable context represents the set of active and shadowed variables with their types given at each program point (or label).

A *label context* $\mathbf{L}$ of a function $F = \textbf{fn}\, f\,\Sigma\,\{L_1{:}\,S_1 \cdots L_n{:}\,S_n\}$ is a mapping that maps each label L_i to a pair $(\boldsymbol{\Psi}_i, \boldsymbol{\Gamma}_i)$ of a lifetime context $\boldsymbol{\Psi}_i$ and a variable context $\boldsymbol{\Gamma}_i$, satisfying the following condition: every lifetime variable appearing in $\boldsymbol{\Gamma}_i$ should be contained in $\boldsymbol{\Psi}_i$.

Subtyping Relation and Preorder on Authorities

To begin with, both the subtyping relation and the preorder on authorities are later defined using the *preorder on lifetimes* $\psi \le_{\boldsymbol{\Psi}} \varphi$ under lifetime context $\boldsymbol{\Psi} = (\Psi, R)$ defined by the following rules.

$$\frac{(\alpha,\beta)\in R}{\alpha \le_{\boldsymbol{\Psi}} \beta} \qquad \frac{\psi \le_{\boldsymbol{\Psi}} \varphi_i \ (i=1,\ldots,n)}{\psi \le_{\boldsymbol{\Psi}} \varphi_1 \wedge \cdots \wedge \varphi_n} \qquad \frac{\psi_i \le_{\boldsymbol{\Psi}} \varphi}{\psi_1 \wedge \cdots \wedge \psi_n \le_{\boldsymbol{\Psi}} \varphi}$$

Type reinterpretation instruction x **as** $A\,T$ is used for folding/unfolding recursive types and modifying lifetime bounds for reference types. Checking type reinterpretation involves the *subtyping relation* $T \le_{\boldsymbol{\Psi}} T'$ under lifetime context $\boldsymbol{\Psi}$ defined by the following rules. $T \sim_{\boldsymbol{\Psi}} T'$ is used to indicate that both $T \le_{\boldsymbol{\Psi}} T'$ and $T \ge_{\boldsymbol{\Psi}} T'$ hold.

$$T \le_{\boldsymbol{\Psi}} T \qquad \frac{T_1 \le_{\boldsymbol{\Psi}} T_2 \quad T_2 \le_{\boldsymbol{\Psi}} T_3}{T_1 \le_{\boldsymbol{\Psi}} T_3}$$

$$\frac{T \le_{\boldsymbol{\Psi}} T'}{\textbf{own}\, T \le_{\boldsymbol{\Psi}} \textbf{own}\, T'} \qquad \frac{T \le_{\boldsymbol{\Psi}} T'}{\textbf{immut}_{\psi}\, T \le_{\boldsymbol{\Psi}} \textbf{immut}_{\psi}\, T'}$$

$$\frac{T \sim_{\boldsymbol{\Psi}} T'}{\textbf{mut}_{\psi}\, T \sim_{\boldsymbol{\Psi}} \textbf{mut}_{\psi}\, T'} \qquad \frac{\psi \ge_{\boldsymbol{\Psi}} \varphi}{B_{\psi}\, T \le_{\boldsymbol{\Psi}} B_{\varphi}\, T}$$

$$\frac{T_i \le_{\boldsymbol{\Psi}} T'_i \ (i=1,\ldots,n)}{\sum_{i=1}^{n} T_i \le_{\boldsymbol{\Psi}} \sum_{i=1}^{n} T'_i} \qquad \frac{T_i \le_{\boldsymbol{\Psi}} T'_i \ (i=1,\ldots,n)}{\prod_{i=1}^{n} T_i \le_{\boldsymbol{\Psi}} \prod_{i=1}^{n} T'_i}$$

$$\frac{T \le_{\boldsymbol{\Psi}} T'}{\mu X.T \le_{\boldsymbol{\Psi}} \mu X.T'} \qquad \mu X.T \sim_{\boldsymbol{\Psi}} \mu Y.T[Y/X] \qquad \mu X.T \sim_{\boldsymbol{\Psi}} T[\mu X.T/X]$$

Subtyping rules on recursive types are quite restrained; for example, property $\mu X.\,\textbf{int} + \textbf{own}\, X \sim_{\boldsymbol{\Psi}} \mu X.\,\textbf{int} + \textbf{own}\,(\textbf{int} + \textbf{own}\, X)$ (or more generally, $\mu X.T \sim_{\boldsymbol{\Psi}} \mu X.T[T/X]$) cannot be derived from the rules above, but it can safely be admitted. In order to exert subtyping thoroughly, there are some remedies such as semantically regarding recursive types as regular (possibly infinite) trees; for our purpose, nevertheless, the presented subtyping rules are satisfactory.

[12] λ_{Rust} introduces *a lifetime for each function* (indicated by an archaic Greek letter digamma) to ensure that lifetime parameters live longer than local lifetime variables.

The *preorder on pointer authorities* $A \le_{\mathbf{\Psi}} A'$ under lifetime context $\mathbf{\Psi}$ is defined by the following rules. This is used to describe type checking on mutable (re)borrowing instruction $\mathbf{let}\ y = \mathbf{mut\ bor}\ x\ \mathbf{till}\ \psi$.

$$A \le_{\mathbf{\Psi}} \mathbf{own} \qquad \mathbf{immut}_\psi \le_{\mathbf{\Psi}} \mathbf{mut}_\varphi \qquad \frac{\psi \le_{\mathbf{\Psi}} \varphi}{B_\psi \le_{\mathbf{\Psi}} B_\varphi}$$

In this context, we also define $A \wedge A'$ by the following rules. It works as the minimum of pointer authorities with respect to preorder $\le_{\mathbf{\Psi}}$ under any lifetime context $\mathbf{\Psi}$, and is used for description of type checking on dereference instruction $\mathbf{let}\ y = \mathbf{deref}\ x$.

$$A \wedge \mathbf{own} = \mathbf{own} \wedge A = A \qquad \mathbf{immut}_\psi \wedge \mathbf{mut}_\varphi = \mathbf{mut}_\varphi \wedge \mathbf{immut}_\psi = \mathbf{immut}_\psi$$
$$B_\psi \wedge B_\varphi = B_{\psi \wedge \varphi}$$

Copyability

Type copyability $T\colon \mathbf{copy}$ (more precisely, copyability of data of type T) is defined by the following rules. It corresponds to Rust's **Copy** trait. In short, types containing owning pointers or mutable references out of the guard of immutable references are not copyable. This is used to describe type checking on copy instruction $\mathbf{let}\ y = \mathbf{copy}\ x$.

$$\mathbf{int}\colon \mathbf{copy} \qquad \mathbf{immut}_\psi\, T\colon \mathbf{copy}$$
$$\frac{T_i\colon \mathbf{copy}\ (i = 1, \ldots, n)}{\sum_{i=1}^n T_i\colon \mathbf{copy}} \qquad \frac{T_i\colon \mathbf{copy}\ (i = 1, \ldots, n)}{\prod_{i=1}^n T_i\colon \mathbf{copy}} \qquad \frac{T\colon \mathbf{copy}}{\mu X.T\colon \mathbf{copy}}$$

Note that the copyability checking rule for recursive types of form $\mu X.T$ is legitimate, since each appearance of X is guarded under some pointer type; while $\mu X.\ \mathbf{int} + \mathbf{immut}_\psi\ X$ is copyable, $\mu X.\ \mathbf{int} + \mathbf{own}\ X$ is not (it requires recursive copying for replication).

Type Checking on Instructions

The relation $\mathbf{\Sigma}; \Psi_{\mathrm{ex}}; \mathbf{\Psi}, \mathbf{\Gamma} \vdash I;\ \mathbf{\Psi}', \mathbf{\Gamma}'$ (given lifetime context $\mathbf{\Psi}$ and variable context $\mathbf{\Gamma}$, instruction I yields lifetime context $\mathbf{\Psi}'$ and variable context $\mathbf{\Gamma}'$, under function signature context $\mathbf{\Sigma}$ and lifetime parameter context Ψ_{ex}) is defined by the following rules.[13][14] $\mathbf{\Sigma}$ and Ψ_{ex} are omitted when not relevant.

$$\mathbf{\Psi}, \mathbf{\Gamma} \vdash \mathbf{let}\ x = n;\, \mathbf{\Psi}, \mathbf{\Gamma} + \{x\colon \mathbf{own\ int}\}$$

$$\frac{x\colon A\ \mathbf{int},\ x'\colon A'\ \mathbf{int} \in \mathbf{\Gamma}}{\mathbf{\Psi}, \mathbf{\Gamma} \vdash \mathbf{let}\ y = x\ o\ x';\ \mathbf{\Psi}, \mathbf{\Gamma} + \{y\colon \mathbf{own\ int}\}}$$

$$\frac{x\colon A\ \mathbf{int},\ x'\colon A'\ \mathbf{int} \in \mathbf{\Gamma}}{\mathbf{\Psi}, \mathbf{\Gamma} \vdash \mathbf{let}\ y = x\ r\ x';\ \mathbf{\Psi}, \mathbf{\Gamma} + \{y\colon \mathbf{own\ bool}\}}$$

$$\mathbf{\Psi}, \mathbf{\Gamma} + \{x\colon A\ T\} \vdash \mathbf{let}\ y = x;\ \mathbf{\Psi}, \mathbf{\Gamma} + \{y\colon A\ T\}$$

$$\frac{T\colon \mathbf{copy} \quad x\colon A\ T \in \mathbf{\Gamma}}{\mathbf{\Psi}, \mathbf{\Gamma} \vdash \mathbf{let}\ y = \mathbf{copy}\ x;\ \mathbf{\Psi}, \mathbf{\Gamma} + \{y\colon \mathbf{own}\ T\}}$$

---

[13] $<_{\mathbf{\Psi}}$ means that $\le_{\mathbf{\Psi}}$ holds but $\ge_{\mathbf{\Psi}}$ does not.

[14] For sets A and B, $A + B$ is a union $A \cup B$ that is defined just if $A \cap B \neq \emptyset$ holds, and $A - B$ is a set difference $A \setminus B$ that is defined just if $A \supseteq B$ holds; note that $A + B = C$ is equivalent to $A = C - B$ for sets A, B, C. For a binary relation R, R^+ stands for its transitive closure.

$$\frac{x\colon \mathbf{immut}_{\psi}\, T \in \boldsymbol{\Gamma}}{\boldsymbol{\Psi}, \boldsymbol{\Gamma} \vdash \mathbf{let}\, y = \mathbf{repl}\, x;\ \boldsymbol{\Psi}, \boldsymbol{\Gamma} + \{y\colon \mathbf{immut}_{\psi}\, T\}}$$

$$\frac{\mathbf{mut}_{\psi} <_{\boldsymbol{\Psi}} A}{\boldsymbol{\Psi}, \boldsymbol{\Gamma} + \{x\colon A\, T\} \vdash \mathbf{let}\, y = \mathbf{mut\,bor}\, x\ \mathbf{till}\ \psi;\ \boldsymbol{\Psi}, \boldsymbol{\Gamma} + \{x\colon^{\psi} A\, T,\ y\colon \mathbf{mut}_{\psi}\, T\}}$$

$$\boldsymbol{\Psi}, \boldsymbol{\Gamma} + \{x\colon A\, T\} \vdash \mathbf{let}\, y = \mathbf{ref}\, x;\ \boldsymbol{\Psi}, \boldsymbol{\Gamma} + \{y\colon \mathbf{own}\, A\, T\}$$

$$\boldsymbol{\Psi}, \boldsymbol{\Gamma} + \{x\colon A\, A'\, T\} \vdash \mathbf{let}\, y = \mathbf{deref}\, x;\ \boldsymbol{\Psi}, \boldsymbol{\Gamma} + \{y\colon A''\, T\} \text{ where } A'' = A \wedge A'$$

$$\boldsymbol{\Psi}, \boldsymbol{\Gamma} + \{x\colon \mathbf{own}\, T_i\} \vdash \mathbf{let}\, y = \mathbf{inj}_i^{\sum_{j=1}^{n} T_j}\, x;\ \boldsymbol{\Psi}, \boldsymbol{\Gamma} + \{y\colon \mathbf{own} \textstyle\sum_{j=1}^{n} T_j\}$$

$$\boldsymbol{\Psi}, \boldsymbol{\Gamma} + \{x_i\colon \mathbf{own}\, T_i \mid i = 1, \dots, n\} \vdash \mathbf{let}\, y = (x_1, \dots, x_n);\ \boldsymbol{\Psi}, \boldsymbol{\Gamma} + \{y\colon \mathbf{own} \textstyle\prod_{i=1}^{n} T_i\}$$

$$\boldsymbol{\Psi}, \boldsymbol{\Gamma} + \{x\colon \mathbf{own} \textstyle\prod_{i=1}^{n} T_i\} \vdash \mathbf{let}\, (y_1, \dots, y_n) = x;\ \boldsymbol{\Psi}, \boldsymbol{\Gamma} + \{y_i\colon \mathbf{own}\, T_i \mid i = 1, \dots, n\}$$

$$\frac{\begin{array}{l}\boldsymbol{\Sigma}(f) = \langle \alpha_1, \dots, \alpha_m \mid \alpha_{a_1} \le \alpha_{b_1}, \dots, \alpha_{a_{m'}} \le \alpha_{b_{m'}} \rangle \\ \qquad (z_1\colon A_1\, T_1, \dots, z_n\colon A_n\, T_n) \to A'\, T'\end{array} \quad \psi_{a_k} \le_{\boldsymbol{\Psi}} \psi_{b_k}\ (k = 1, \dots, m')}{\begin{array}{l}\boldsymbol{\Sigma}; \boldsymbol{\Psi}, \boldsymbol{\Gamma} + \{x_i\colon A_i\, T_i \mid i = 1, \dots, n\} \\ \qquad \vdash \mathbf{let}\, y = f\langle \psi_1, \dots, \psi_m \rangle(x_1, \dots, x_n);\ \boldsymbol{\Psi}, \boldsymbol{\Gamma} + \{y\colon A'\, T'\}\end{array}}$$

$$\frac{x\colon \mathbf{mut}_{\psi}\, T,\ x'\colon A\, T \in \boldsymbol{\Gamma} \quad A \text{ is } \mathbf{own} \text{ or has form } \mathbf{mut}_{\psi'}}{\boldsymbol{\Psi}, \boldsymbol{\Gamma} \vdash \mathbf{swap}(x, x');\ \boldsymbol{\Psi}, \boldsymbol{\Gamma}}$$

$$\frac{\text{if } A \text{ is } \mathbf{own}\text{, then } T \text{ does not contain } \mathbf{own} \text{ and } \mathbf{mut}}{\boldsymbol{\Psi}, \boldsymbol{\Gamma} + \{x\colon A\, T\} \vdash \mathbf{drop}\, x;\ \boldsymbol{\Psi}, \boldsymbol{\Gamma}}$$

$$\boldsymbol{\Psi}, \boldsymbol{\Gamma} + \{x\colon \mathbf{mut}_{\psi}\, T\} \vdash \mathbf{immut}\, x;\ \boldsymbol{\Psi}, \boldsymbol{\Gamma} + \{x\colon \mathbf{immut}_{\psi}\, T\}$$

$$\frac{A\, T \le_{\boldsymbol{\Psi}} A'\, T'}{\boldsymbol{\Psi}, \boldsymbol{\Gamma} + \{x\colon A\, T\} \vdash x\ \mathbf{as}\ A'\, T';\ \boldsymbol{\Psi}, \boldsymbol{\Gamma} + \{x\colon A'\, T'\}}$$

$$\Psi_{\mathrm{ex}}; (\Psi, R), \boldsymbol{\Gamma} \vdash \mathbf{intro}\, \alpha;\ \big(\Psi + \{\alpha\}, \big(R + \{\alpha\} \times (\{\alpha\} + \Psi)\big)\big), \boldsymbol{\Gamma}$$

$$\frac{\alpha \notin \Psi_{\mathrm{ex}} \quad \alpha \text{ is the smallest on } R \quad \text{types in } \boldsymbol{\Gamma} \text{ do not contain } \alpha}{\begin{array}{l}\Psi_{\mathrm{ex}}; (\Psi + \alpha, R), \boldsymbol{\Gamma} \vdash \mathbf{now}\, \alpha;\ \big(\Psi,\ R \cap (\Psi \times \Psi)\big), \boldsymbol{\Gamma}_{*} \\ \qquad \text{where } \boldsymbol{\Gamma}^{\dagger} = \{x\colon^{\psi} A\, T \mid \psi \text{ contains } \alpha\} \\ \qquad\quad \text{and } \boldsymbol{\Gamma}_{*} = \boldsymbol{\Gamma} - \boldsymbol{\Gamma}^{\dagger} + \{x\colon A\, T \mid x\colon^{\psi} A\, T \in \boldsymbol{\Gamma}^{\dagger}\}\end{array}}$$

Type checking for **drop**. The awkward precondition "if A is **own**, then T does not contain **own** and **mut**" in a rule for **drop** *can* be eliminated, but it is imposed here for the sake of simplicity of operational semantics and translation to CHCs; the rule "if A is **own**, then T does not contain **own**" simplifies operational semantics, and the rule "if A is **own**, then T does not contain **mut**" simplifies translation to CHCs.

Type checking for lifetime elimination. When local lifetime variable α is eliminated by instruction **now** α, there should not remain any variable, active or shadowed, that contains α in the type; it is guaranteed in type checking on programs by the restriction on label contexts.

Type Checking on Statements

The relation $\boldsymbol{\Sigma}; \Psi_{\mathrm{ex}}; \boldsymbol{\Psi}, \boldsymbol{\Gamma}; \mathbf{L}, A'\, T' \vdash S$ (given lifetime context $\boldsymbol{\Psi}$ and variable context $\boldsymbol{\Gamma}$, statement S jumps to labels typed by $\mathbf{L}$ or returns a value of type $A'\, T'$, under function signature context $\boldsymbol{\Sigma}$ and lifetime parameter context Ψ_{ex}) is defined by the following rules.

$$\frac{\mathbf{L}(L) = (\mathbf{\Psi}', \mathbf{\Gamma}') \quad \mathbf{\Sigma}; \Psi_{\text{ex}}; \mathbf{\Psi}, \mathbf{\Gamma} \vdash I;\ \mathbf{\Psi}', \mathbf{\Gamma}'}{\mathbf{\Sigma}; \Psi_{\text{ex}}; \mathbf{\Psi}, \mathbf{\Gamma}; \mathbf{L}, A'\, T' \vdash I;\ \mathbf{goto}\, L}$$

$$\mathbf{\Sigma}; \Psi_{\text{ex}}; (\Psi_{\text{ex}}, R), \{x\colon A'\, T'\}; \mathbf{L}, A'\, T' \vdash \mathbf{return}\, x$$

$$\frac{x\colon A \sum_{j=1}^{n} T_j \in \mathbf{\Gamma} \qquad \mathbf{L}(L_i) = \begin{cases} (\mathbf{\Psi}, \mathbf{\Gamma}) & (\tilde{y}_i = -) \\ \big(\mathbf{\Psi}, \mathbf{\Gamma} - \{x\colon A \sum_{j=1}^{n} T_j\} + \{y_i\colon A\, T_i\}\big) & (\tilde{y}_i = y_i) \end{cases} \ (i = 1, \dots, n)}{\mathbf{\Sigma}; \Psi_{\text{ex}}; \mathbf{\Psi}, \mathbf{\Gamma}; \mathbf{L}, A'\, T' \vdash \mathbf{match}\, x\, \{\mathbf{inj}_1\, \tilde{y}_1\colon \mathbf{goto}\, L_1;\ \dots;\ \mathbf{inj}_n\, \tilde{y}_n\colon \mathbf{goto}\, L_n\}}$$

Type checking for return. When escaping from a function with **return** x, there should not remain any local lifetime variable, and any variable but x. In particular, there remains no *shadowed* variable; thus, *(re)borrowing starts and ends within a function*, which means that for mutable (re)borrowing instruction **let** y = **mut bor** x **till** ψ, lifetime ψ should be equivalent to some *local lifetime variable*.

Type Checking on Functions and Programs

The relation $\mathbf{\Sigma}; \mathbf{L} \vdash F$ (function F is well-typed by label context $\mathbf{L}$ under function signature context $\mathbf{\Sigma}$) is defined by the following rule.[15]

$$\frac{\begin{gathered}\mathbf{\Sigma}; \Psi_{\text{ex}}; \mathbf{\Psi}, \mathbf{\Gamma}; \mathbf{L}, A'\, T' \vdash S_i \text{ where } (\mathbf{\Psi}, \mathbf{\Gamma}) = \mathbf{L}(L_i)\ (i = 1, \dots, l) \\ \mathbf{L}(\mathbf{entry}) = \big((\Psi_{\text{ex}}, R_{\text{ex}}), \{x_1\colon A_1\, T_1, \dots, x_n\colon A_n\, T_n\}\big)\end{gathered}}{\begin{gathered}\mathbf{\Sigma}; \mathbf{L} \vdash \mathbf{fn}\, f\, \langle \alpha_1, \dots, \alpha_m \mid \alpha_{a_1} \le \alpha_{b_1}, \dots, \alpha_{a_{m'}} \le \alpha_{b_{m'}} \rangle \\ (x_1\colon A_1\, T_1, \dots, x_n\colon A_n\, T_n)\{L_1\colon S_1 \cdots L_l\colon S_l\}\end{gathered}}$$

$$\text{where } \Psi_{\text{ex}} = \{\alpha_1, \dots, \alpha_m\} \text{ and } R_{\text{ex}} = \big(\mathrm{Id}_{\Psi_{\text{ex}}} \cup \{(\alpha_{a_j}, \alpha_{b_j}) \mid j = 1, \dots, m'\}\big)^+$$

For program $P = F_1 \cdots F_n$, the relation $(\mathbf{L}_f)_f \vdash P$ (program P is well-typed by an indexed family of label contexts $(\mathbf{L}_f)_f$ indexed by function names) is defined as the condition $\forall i \in \{1, \dots, n\}.\ \mathbf{\Sigma}; \mathbf{L}_{f_i} \vdash F_i$, where f_i is the name of F_i for each i, and $\mathbf{\Sigma}$ is the function signature context of P.

3.1.4 Operational Semantics

State Model: Stack and Heap

Mimicking the typical behavior of hardware, the state of execution is described by the state of a call stack (represented as a *stack* $\mathbf{S}$) and the state of heap memory (represented as a *heap* $\mathbf{H}$) in our operational semantics.

A *(stack) frame* $\mathbf{F}$ is a mapping that maps a variable name x to an address (natural number) a or an invalid value $\bot$, satisfying the condition that only a finite number of variable names x are mapped to a valid address; a frame represents the addresses of the (active or shadowed) variables in each function call.

A *stack* $\mathbf{S}$ has form $[f_0, L_0]\, \mathbf{F}_0; \tilde{\mathbf{S}}$, where a *pre-stack* $\tilde{\mathbf{S}}$ has form $[f_1, L_1]\, \langle \psi^1_1, \dots, \psi^1_{m_1} \rangle\, x_1, \mathbf{F}_1; \dots; [f_n, L_n]\, \langle \psi^n_1, \dots, \psi^n_{m_n} \rangle\, x_n, \mathbf{F}_n$; $(n \ge 0)$. A stack consists of accumulated layers of frames with additional information. The square-bracketed part $[f_i, L_i]$ indicates the program point with function name f_i and label L_i. The angle-bracketed part $\langle \psi^i_1, \dots, \psi^i_{m_i} \rangle$ indicates lifetime arguments passed to the function of the frame just above (it is not used in operational semantics but is

[15] $\mathrm{Id}_{\Psi_{\text{ex}}}$ is the identity relation on set Ψ_{ex}.

added for the sake of discussion on the safety invariant). The following part $x_i, \mathbf{F}_i$ means that the return value will be stored at x_i when the function of the frame just above returns.

A *heap* $\mathbf{H}$ is a mapping that maps an address (natural number) a to a integer n or an invalid value $\bot$, satisfying a condition that only a finite number of addresses a are mapped to a valid integer. Thus conceptually, the *memory cell* of each address accommodates one integer value.

$x_{\mathbf{F}}$ stands for the address stored for variable x in frame $\mathbf{F}$, and $*_{\mathbf{H}}a$ stands for the value stored at address a in heap $\mathbf{H}$. The frame $\mathbf{F}$ with the value at index x_i modified into a_i for each $i = 1, \dots, n$ is written as $\mathbf{F}\{x_1 \triangleleft a_1; \dots; x_n \triangleleft a_n\}$ (indices $x_1, \dots, x_n$ should be mutually distinct); we also use a similar notation for heaps. Moreover, we introduce a number of useful notations. $a \triangleleft_n v$ stands for a set of writes $a \triangleleft v$; $a + 1 \triangleleft v$; $\dots$; $a + n - 1 \triangleleft v$, and $a \blacktriangleleft_n^{\mathbf{H}} b$ for a set of writes $a \triangleleft \mathbf{H}[b]$; $a + 1 \triangleleft \mathbf{H}[b + 1]$; $\dots$; $a + n - 1 \triangleleft \mathbf{H}[b + n - 1]$. $\mathrm{new}_{\mathbf{H}}(n, a)$ means that $\mathbf{H}[a] = \mathbf{H}[a + 1] = \cdots = \mathbf{H}[a + n - 1] = \bot$ holds.

Type Size

Type size $\#T$ is recursively defined as follows; the type size of T represents how many memory cells are needed to store data of type T. The sizes for type variables are undefined; the size for any *complete type* (a type not used as a substructure of a larger type) is nevertheless defined since any type variable in the type is guarded under a pointer type.

$$\#\,\mathbf{int} \;:=\; 1 \quad \#A\,T \;:=\; 1 \quad \#\mu X.T \;:=\; \#T$$
$$\#\textstyle\sum_{i=1}^{n} T_i \;:=\; 1 + \max\{\#T_1, \dots, \#T_n\} \quad \#\textstyle\prod_{i=1}^{n} T_i \;:=\; \#T_1 + \cdots + \#T_n$$

Typed Instructions and Statements

In order to describe operational semantics, information on type sizes of variables is widely required (in addition, pointer authorities are relevant for dereference instruction **let** y = **deref** x and drop instruction **drop** x). For that purpose, we introduce *typed instructions and statements*. Typed instructions and statements are also used for description of translation to CHCs, because mutable references and other pointers (owning pointers or immutable references) should be differently treated.

The syntax of *typed instruction* $\hat{I}$ can be obtained by replacing x with $x\colon A\,T$; we often omit $:A\,T$ when type information is not relevant. In addition, the symbol $\hat{I}^\dagger$ indicates any typed instruction but function call instruction **let** $y = f\langle\cdots\rangle(x_1, \dots, x_n)$.

The syntax of *typed statement* $\hat{S}$ can be obtained by replacing I with $\hat{I}$, x with $x\colon A\,T$, and $\tilde{x}$ with $\tilde{x}\colon A\,T$; we often omit $:A\,T$ when type information is not relevant. We can obtain a typed statement by giving type information to a statement S with a variable context $\mathbf{\Gamma}$; we call it the *type modification* of S with $\mathbf{\Gamma}$ and write it as $\mathrm{TM}(S|\mathbf{\Gamma})$. Although it can be described with natural rules, we do not give the precise definition in the paper since it is a little cumbersome.

Operational Semantics on Instructions

The relation $\mathbf{F}/\mathbf{H} \vdash \hat{I}^{\dagger};\ \mathbf{F}'/\mathbf{H}'$ (given frame $\mathbf{F}$ with heap $\mathbf{H}$, typed instruction $I^{\dagger}$ yields frame $\mathbf{F}'$ with heap $\mathbf{H}'$) is defined by the following rules.

$$\frac{\mathrm{new}_{\mathbf{H}}(1, a)}{\mathbf{F}/\mathbf{H} \vdash \textbf{let}\, x = n;\ \mathbf{F}\{x \triangleleft a\}/\mathbf{H}\{a \triangleleft n\}}$$

$$\frac{\mathrm{new}_{\mathbf{H}}(1, a)}{\mathbf{F}/\mathbf{H} \vdash \textbf{let}\, y = x\ o\ x';\ \mathbf{F}\{x, x' \triangleleft \bot;\ y \triangleleft a\}/\mathbf{H}\{a \triangleleft *_{\mathbf{H}}x_{\mathbf{F}}\ o\ *_{\mathbf{H}}x'_{\mathbf{F}}\}}$$

$$\frac{\mathrm{new}_{\mathbf{H}}(1, a)}{\mathbf{F}/\mathbf{H} \vdash \textbf{let}\, y = x\ r\ x';\ \mathbf{F}\{x, x' \triangleleft \bot;\ y \triangleleft a\}/\mathbf{H}\{a \triangleleft *_{\mathbf{H}}x_{\mathbf{F}}\ r\ *_{\mathbf{H}}x'_{\mathbf{F}}\}}$$

$$\mathbf{F}/\mathbf{H} \vdash \textbf{let}\, y = x;\ \mathbf{F}\{x \triangleleft \bot;\ y \triangleleft x_{\mathbf{F}}\}/\mathbf{H}$$

$$\frac{\mathrm{new}_{\mathbf{H}}(\#T, a)}{\mathbf{F}/\mathbf{H} \vdash \textbf{let}\, y\colon \textbf{own}\, T = \textbf{copy}\, x;\ \mathbf{F}\{y \triangleleft a\}/\mathbf{H}\{a \blacktriangleleft^{\mathbf{H}}_{\#T} x_{\mathbf{F}}\}}$$

$$\mathbf{F}/\mathbf{H} \vdash \textbf{let}\, y = \textbf{repl}\, x;\ \mathbf{F}\{y \triangleleft x_{\mathbf{F}}\}/\mathbf{H}$$

$$\mathbf{F}/\mathbf{H} \vdash \textbf{let}\, y = \textbf{mut bor}\, x\ \textbf{till}\ \psi;\ \mathbf{F}\{y \triangleleft x_{\mathbf{F}}\}/\mathbf{H}$$

$$\frac{\mathrm{new}_{\mathbf{H}}(1, a)}{\mathbf{F}/\mathbf{H} \vdash \textbf{let}\, y = \textbf{ref}\, x\ \textbf{till}\ \psi;\ \mathbf{F}\{x \triangleleft \bot;\ y \triangleleft a\}/\mathbf{H}\{a \triangleleft x_{\mathbf{F}}\}}$$

$$\mathbf{F}/\mathbf{H} \vdash \textbf{let}\, y = \textbf{deref}\,(x\colon \textbf{own}\, A\, T)\ \textbf{till}\ \psi;\ \mathbf{F}\{x \triangleleft \bot;\ y \triangleleft *_{\mathbf{H}}x_{\mathbf{F}}\}/\mathbf{H}\{x_{\mathbf{F}} \triangleleft \bot\}$$

$$\mathbf{F}/\mathbf{H} \vdash \textbf{let}\, y = \textbf{deref}\,(x\colon B_{\varphi}\, A\, T)\ \textbf{till}\ \psi;\ \mathbf{F}\{x \triangleleft \bot;\ y \triangleleft *_{\mathbf{H}}x_{\mathbf{F}}\}/\mathbf{H}$$

$$\frac{\mathrm{new}_{\mathbf{H}}(\#T, a)}{\begin{array}{l}\mathbf{F}/\mathbf{H} \vdash\ \textbf{let}\, y = \textbf{inj}^{T}_{i}(x\colon \textbf{own}\, T'); \\ \qquad\quad \mathbf{F}\{x \triangleleft \bot;\ y \triangleleft a\}/\mathbf{H}\{a \triangleleft i;\ a + 1 \blacktriangleleft^{\mathbf{H}}_{\#T'} x_{\mathbf{F}};\ a + 1 + \#T' \triangleleft_{\#T - \#T' - 1} 0\}\end{array}}$$

$$\frac{\mathrm{new}_{\mathbf{H}}(\#T, a)}{\begin{array}{l}\mathbf{F}/\mathbf{H} \vdash\ \textbf{let}\, y\colon \textbf{own}\, T = (x_1\colon \textbf{own}\, T_1, \ldots, x_n\colon \textbf{own}\, T_n); \\ \qquad\quad \mathbf{F}\{x_1, \ldots, x_n \triangleleft \bot;\ y \triangleleft a\} \\ \qquad\quad /\ \mathbf{H}\{x_{i\mathbf{F}} \triangleleft_{\#T_i} \bot\ (i = 1, \ldots, n);\ a + \sum_{j=1}^{i-1} \#T_j \blacktriangleleft^{\mathbf{H}}_{\#T_i} x_{i\mathbf{F}}\ (i = 1, \ldots, n)\}\end{array}}$$

$$\mathbf{F}/\mathbf{H} \vdash \textbf{swap}(x\colon A\, T, y);\ \mathbf{F}/\mathbf{H}\{x_{\mathbf{F}} \blacktriangleleft^{\mathbf{H}}_{\#T} y_{\mathbf{F}};\ y_{\mathbf{F}} \blacktriangleleft^{\mathbf{H}}_{\#T} x_{\mathbf{F}}\}$$

$$\mathbf{F}/\mathbf{H} \vdash \textbf{drop}\,(x\colon \textbf{own}\, T);\ \mathbf{F}\{x \triangleleft \bot\}/\mathbf{H}\{x_{\mathbf{F}} \triangleleft_{\#T} \bot\}$$

$$\mathbf{F}/\mathbf{H} \vdash \textbf{drop}\,(x\colon B_{\psi}\, T);\ \mathbf{F}\{x \triangleleft \bot\}/\mathbf{H}$$

$$\frac{\hat{I}^{\dagger} = \textbf{immut}\, x,\ \ x\ \textbf{as}\ A\, T,\ \ \textbf{intro}\ \alpha,\ \ \textbf{now}\ \alpha}{\mathbf{F}/\mathbf{H} \vdash \hat{I}^{\dagger};\ \mathbf{F}/\mathbf{H}}$$

Operational semantics on injection instruction. For injection instruction $\textbf{let}\, y = \textbf{inj}^{\sum_{j=1}^{n} T_j}_{i} x$, zeros are padded when $1 + \#T_i$ is smaller than $\# \sum_{j=1}^{n} T_j$ (that is, $\#T_i$ is smaller than $\max_j \#T_j$).

Operational Semantics on Statements

The relation $\mathbf{S}/\mathbf{H} \vdash \hat{S};\ \left\{{}^{\mathbf{S}'}_{a}\right\}/\mathbf{H}'$ (given a stack $\mathbf{S}$ with a heap $\mathbf{H}$, typed statement $\hat{S}$ yields a new stack $\mathbf{S}'$ or an address a with a new heap $\mathbf{H}'$) is defined by the following rules.[16]

$$\frac{\mathbf{F}/\mathbf{H} \vdash \hat{I}^{\dagger};\ \mathbf{F}'/\mathbf{H}'}{[f,L]\,\mathbf{F};\tilde{\mathbf{S}}/\mathbf{H} \vdash \hat{I}^{\dagger};\mathbf{goto}\,L';\ [f,L']\,\mathbf{F}';\tilde{\mathbf{S}}/\mathbf{H}'}$$

$$\begin{aligned}&[f',L]\,\mathbf{F};\tilde{\mathbf{S}}/\mathbf{H} \vdash \mathbf{let}\,y = f\langle\psi_1,\dots,\psi_m\rangle(x_1,\dots,x_n);\mathbf{goto}\,L';\\ &\quad[f,\mathbf{entry}]\,\bot\{x_i \triangleleft x_{i\mathbf{F}}\ (i=1,\dots,n)\};\\ &\quad[f',L']\,\langle\psi_1,\dots,\psi_m\rangle\,y,\mathbf{F}\{x_i \triangleleft \bot\ (i=1,\dots,n)\};\tilde{\mathbf{S}}/\mathbf{H}\end{aligned}$$

$$[f,L]\,\mathbf{F};[f',L']\,\langle\cdots\rangle\,y,\mathbf{F}';\tilde{\mathbf{S}}/\mathbf{H} \vdash \mathbf{return}\,x;\ [f',L']\,\mathbf{F}'\{y \triangleleft x_{\mathbf{F}}\};\tilde{\mathbf{S}}/\mathbf{H}$$

$$[f,L]\,\mathbf{F};/\mathbf{H} \vdash \mathbf{return}\,x;\ x_{\mathbf{F}}/\mathbf{H}$$

$$\frac{*_{\mathbf{H}}x_{\mathbf{F}} = i}{[f,L]\,\mathbf{F};\tilde{\mathbf{S}}/\mathbf{H} \vdash \mathbf{match}\,x\,\{\cdots;\mathbf{inj}_i\!-\!:\mathbf{goto}\,L';\cdots\};\ [f,L']\,\mathbf{F};\tilde{\mathbf{S}}/\mathbf{H}}$$

$$\frac{*_{\mathbf{H}}x_{\mathbf{F}} = i}{\begin{aligned}&[f,L]\,\mathbf{F};\tilde{\mathbf{S}}/\mathbf{H} \vdash \mathbf{match}\,(x{:}\,B_{\psi}\,T)\,\{\cdots;\mathbf{inj}_i y{:}\,\mathbf{goto}\,L';\cdots\};\\ &\qquad\qquad[f,L']\,\mathbf{F}\{y \triangleleft x_{\mathbf{F}}+1\};\tilde{\mathbf{S}}/\mathbf{H}\end{aligned}}$$

$$\frac{*_{\mathbf{H}}x_{\mathbf{F}} = i}{\begin{aligned}&[f,L]\,\mathbf{F};\tilde{\mathbf{S}}/\mathbf{H} \vdash \mathbf{match}\,(x{:}\,\mathbf{own}\,T)\,\{\cdots;\mathbf{inj}_i y{:}\,\mathbf{goto}\,L';\cdots\};\\ &\qquad\qquad[f,L']\,\mathbf{F}\{y \triangleleft x_{\mathbf{F}}+1\};\tilde{\mathbf{S}}/\mathbf{H}\{x_{\mathbf{F}} \triangleleft \bot\}\end{aligned}}$$

Operational Semantics on Programs

The relation $P{:}\,(\mathbf{L}_f)_f \vdash \mathbf{S}/\mathbf{H} \to \left\{\begin{smallmatrix}\mathbf{S}'\\ a\end{smallmatrix}\right\}/\mathbf{H}'$ (under program P typed by $(\mathbf{L}_f)_f$, a stack $\mathbf{S}$ with a heap $\mathbf{H}$ changes by one step into a new stack $\mathbf{S}'$ or an address a with a new heap $\mathbf{H}'$) is defined by the following rule. Typed statement $\hat{S}^{P:(\mathbf{L}_f)_f}_{f_0,L}$ represents the type modification $\mathrm{TM}(S|\mathbf{\Gamma})$, where S is the statement labeled L in function f_0 in program P, and $\mathbf{\Gamma}$ is the variable context part of $\mathbf{L}_{f_0}(L)$.

$$\frac{\mathbf{S} = [f_0,L]\,\mathbf{F};\tilde{\mathbf{S}} \quad \mathbf{S}/\mathbf{H} \vdash \hat{S}^{P:(\mathbf{L}_f)_f}_{f_0,L};\ \left\{\begin{smallmatrix}\mathbf{S}'\\ a\end{smallmatrix}\right\}/\mathbf{H}'}{P{:}\,(\mathbf{L}_f)_f \vdash \mathbf{S}/\mathbf{H} \to \left\{\begin{smallmatrix}\mathbf{S}'\\ a\end{smallmatrix}\right\}/\mathbf{H}'}$$

Finally, the relation $P{:}\,(\mathbf{L}_f)_f \vdash \mathbf{S}/\mathbf{H} \twoheadrightarrow a/\mathbf{H}'$ (under program P typed by $(\mathbf{L}_f)_f$, a stack $\mathbf{S}$ with a heap $\mathbf{H}$ changes by one or more steps into an address a with a new heap $\mathbf{H}'$) is defined by the following rules.

$$\frac{P{:}\,(\mathbf{L}_f)_f \vdash \mathbf{S}/\mathbf{H} \to a/\mathbf{H}'}{P{:}\,(\mathbf{L}_f)_f \vdash \mathbf{S}/\mathbf{H} \twoheadrightarrow a/\mathbf{H}'} \qquad \frac{P{:}\,(\mathbf{L}_f)_f \vdash \mathbf{S}/\mathbf{H} \to \mathbf{S}'/\mathbf{H}' \quad P{:}\,(\mathbf{L}_f)_f \vdash \mathbf{S}'/\mathbf{H}' \twoheadrightarrow a/\mathbf{H}''}{P{:}\,(\mathbf{L}_f)_f \vdash \mathbf{S}/\mathbf{H} \twoheadrightarrow a/\mathbf{H}''}$$

3.2 Translation of COR Programs into Sets of CHCs

We formally describe our translation, setting COR as the source language; CHCs are defined within multi-sorted first-order predicate logic introduced here.

3.2.1 Multi-sorted First-order Predicate Logic

Syntax

$$X, Y \ ::= \ \text{(sort variable)}$$

[16]The empty frame (the frame that maps every variable x to $\bot$) is written as $\bot$.

$$
\begin{aligned}
\text{(sort) } S &::= X \mid \textbf{int} \text{ (integer)} \mid \textbf{box}\, S \text{ (box)} \mid \textbf{mut}\, S \text{ (mutable reference)} \\
&\mid \mu X.S \text{ (recursive sort)} \mid +(S_1, \dots, S_n) \text{ (sum)} \mid \times(S_1, \dots, S_n) \text{ (product)}
\end{aligned}
$$

$$x, y, a, b, r ::= \text{((data) variable)}$$

$$
\begin{aligned}
\text{(term) } t &::= x \mid x\colon S \text{ (variable with a sort annotation)} \mid n \text{ (integer)} \\
&\mid t \ o \ t' \text{ (binary operation on integers)} \\
&\mid t \ r \ t' \text{ (binary relation on integers)} \\
&\mid \langle t \rangle \text{ (box)} \mid \langle t, t' \rangle \text{ (mutable reference)} \\
&\mid \textbf{inj}_i^S t \text{ (tagged value)} \mid (t_1, \dots, t_n) \text{ (tuple)}
\end{aligned}
$$

$$
\begin{aligned}
\text{(basic formula) } \varphi &::= t = t' \text{ (equality)} \\
&\mid \varphi \wedge \varphi' \text{ (conjunction)} \mid \varphi \vee \varphi' \text{ (disjunction)} \\
&\mid \varphi \Rightarrow \varphi' \text{ (implication)} \mid \neg\varphi \text{ (negation)}
\end{aligned}
$$

$$f ::= \text{(uninterpreted predicate)}$$

$$\text{(extended formula) } \hat{\varphi} ::= \varphi \mid f(t_1, \dots, t_n)$$

$$\text{(CHC) } H ::= \hat{\varphi}_0 \Longleftarrow \hat{\varphi}_1 \wedge \cdots \wedge \hat{\varphi}_n$$

For a complete sort (a sort not used as a substructure of a larger sort), every appearance of sort variables in sorts should be bound by μ.

In a CHC, all variables are bound with universal quantification. Sort annotations are added to some variables so that sorts of variables are unambiguous (up to equivalence) within a CHC.

Just like COR, $*$ is a shorthand for $\times()$ and **bool** for $* + *$.

Sort Checking

Sort equivalence $S \sim S'$ is defined by the following rules.

$$
S \sim S \qquad \frac{S \sim S'}{S' \sim S} \qquad \frac{S_1 \sim S_2 \ \wedge \ S_2 \sim S_3}{S_1 \sim S_3}
$$

$$
\frac{S \sim S'}{\textbf{box}\, S \sim \textbf{box}\, S'} \qquad \frac{S \sim S'}{\textbf{mut}\, S \sim \textbf{mut}\, S'} \qquad \frac{S_i \sim S_i' \ (i = 1, \dots, n)}{\sum_{i=1}^n S_i \sim \sum_{i=1}^n S_i'} \qquad \frac{S_i \sim S_i' \ (i = 1, \dots, n)}{\prod_{i=1}^n S_i \sim \prod_{i=1}^n S_i'}
$$

$$
\frac{S \sim S'}{\mu X.S \sim \mu X.S'} \qquad \mu X.S \sim \mu Y.S[Y/X] \qquad \mu X.S \sim S[\mu X.S/X]
$$

Sort checking $t\colon S$ is defined by the following rules. Equivalent sorts behave similarly on sort checking.

$$
n\colon \textbf{int} \qquad \frac{t, t'\colon \textbf{int}}{t \ o \ t'\colon \textbf{int}} \qquad \frac{t, t'\colon \textbf{int}}{t \ r \ t'\colon \textbf{bool}} \qquad \frac{t\colon S}{\langle t \rangle\colon \textbf{box}\, S} \qquad \frac{t, t'\colon S}{\langle t, t' \rangle\colon \textbf{mut}\, S}
$$

$$
\frac{t\colon S_i}{\textbf{inj}_i^{\sum_{j=1}^n S_j} t\colon \sum_{j=1}^n S_j} \qquad \frac{t_i\colon S_i \ (i = 1, \dots, n)}{(t_1, \dots, t_n)\colon \prod_{i=1}^n S_i} \qquad \frac{\textbf{inj}_i^S t\colon S \ \wedge \ S \sim S'}{\textbf{inj}_i^{S'} t\colon S} \qquad \frac{t\colon S \ \wedge \ S \sim S'}{t\colon S'}
$$

3.2.2 Translation

Interpretation of Types

The *interpretation* $[\![T]\!]$ of type T as a sort is defined by the following rules. $A^\dagger$ indicates a pointer authority that is not one for mutable references (one for owning pointers or immutable references); this notation is used repeatedly hereinafter.

$$[\![X]\!] := X \quad [\![\mathbf{int}]\!] := \mathbf{int} \quad [\![A^\dagger\, T]\!] := \mathbf{box}\,[\![T]\!] \quad [\![\mathbf{mut}_\psi\, T]\!] := \mathbf{mut}\,[\![T]\!]$$
$$[\![\mu X.T]\!] := \mu X.[\![T]\!] \quad [\![\textstyle\sum_{i=1}^n T_i]\!] := \textstyle\sum_{i=1}^n [\![T_i]\!] \quad [\![\textstyle\prod_{i=1}^n T_i]\!] := \textstyle\prod_{i=1}^n [\![T_i]\!]$$

Interpretation of Typed Instructions

The *interpretation* $[\![\hat{I}]\!](\hat{\varphi})$ of typed instruction $\hat{I}$ under extended formula $\hat{\varphi}$ (representing the constraint of the succeeding label) as a conjunction of extended formulae is defined as follows.[17] For binary operation and relation instructions, $\mathrm{val}_A(a, x)$ stands for a formula $\langle a, a_* \rangle = x$ for some fresh a_* if $A = \mathbf{mut}_\psi$, and $\langle a \rangle = x$ otherwise.

$$[\![\mathbf{let}\, x = n]\!](\hat{\varphi}) := x = n \ \land\ \hat{\varphi}$$

$$\begin{aligned}
[\![\mathbf{let}\, y = (x\colon A\, \mathbf{int})\ o\ (x'\colon A'\, \mathbf{int})]\!](\hat{\varphi}) &:= \mathrm{val}_A(a, x) \ \land\ \mathrm{val}_{A'}(a', x') \\
&\quad \land\ y = \langle a\ o\ a' \rangle \ \land\ \hat{\varphi}
\end{aligned}$$

$$\begin{aligned}
[\![\mathbf{let}\, y = (x\colon A\, \mathbf{int})\ r\ (x'\colon A'\, \mathbf{int})]\!](\hat{\varphi}) &:= \mathrm{val}_A(a, x) \ \land\ \mathrm{val}_{A'}(a', x') \\
&\quad \land\ y = \langle a\ r\ a' \rangle \ \land\ \hat{\varphi}
\end{aligned}$$

$$[\![\mathbf{let}\, y = x]\!](\hat{\varphi}) = [\![\mathbf{let}\, y = \mathbf{copy}\, x]\!](\hat{\varphi}) = [\![\mathbf{let}\, y = \mathbf{repl}\, x]\!](\hat{\varphi}) := y = x \ \land\ \hat{\varphi}$$

$$\begin{aligned}
[\![\mathbf{let}\, y = \mathbf{mut\,bor}\,(x\colon \mathbf{own}\, T)\ \mathbf{till}\ \psi]\!](\hat{\varphi}) &:= \langle a \rangle = x \ \land\ y = \langle a, a_* \rangle \\
&\quad \land\ x_{\mathrm{new}} = \langle a_* \rangle \ \land\ \hat{\varphi}[x_{\mathrm{new}}/x]
\end{aligned}$$

$$\begin{aligned}
[\![\mathbf{let}\, y = \mathbf{mut\,bor}\,(x\colon \mathbf{mut}_\varphi\, T)\ \mathbf{till}\ \psi]\!](\hat{\varphi}) &:= \langle a, a_{**} \rangle = x \ \land\ y = \langle a, a_* \rangle \\
&\quad \land\ x_{\mathrm{new}} = \langle a_*, a_{**} \rangle \ \land\ \hat{\varphi}[x_{\mathrm{new}}/x]
\end{aligned}$$

$$[\![\mathbf{let}\, y = \mathbf{ref}\, x]\!](\hat{\varphi}) := y = \langle x \rangle \ \land\ \hat{\varphi}$$

$$[\![\mathbf{let}\, y = \mathbf{deref}\,(x\colon \mathbf{own}\, A\, T)]\!](\hat{\varphi}) := \langle y \rangle = x \ \land\ \hat{\varphi}$$

$$[\![\mathbf{let}\, y = \mathbf{deref}\,(x\colon \mathbf{mut}_\psi\, \mathbf{own}\, T)]\!](\hat{\varphi}) := \langle \langle a \rangle, \langle a_* \rangle \rangle = x \ \land\ y = \langle a, a_* \rangle \ \land\ \hat{\varphi}$$

$$\begin{aligned}
[\![\mathbf{let}\, y = \mathbf{deref}\,(x\colon \mathbf{mut}_\psi\, \mathbf{mut}_\varphi\, T)]\!](\hat{\varphi}) &:= \langle \langle a, a^\dagger \rangle, \langle a_*, a_*^\dagger \rangle \rangle = x \\
&\quad \land\ a_*^\dagger = a^\dagger \ \land\ y = \langle a, a_* \rangle \ \land\ \hat{\varphi}
\end{aligned}$$

$$\begin{aligned}
[\![\mathbf{let}\, y = \mathbf{deref}\,(x\colon \mathbf{mut}_\psi\, \mathbf{immut}_\varphi\, T)]\!](\hat{\varphi}) &:= \langle \langle a \rangle, \langle a_* \rangle \rangle = x \\
&\quad \land\ \langle a_* \rangle = \langle a \rangle \ \land\ y = \langle a \rangle \ \land\ \hat{\varphi}
\end{aligned}$$

$$[\![\mathbf{let}\, y = \mathbf{deref}\,(x\colon \mathbf{immut}_\psi\, A^\dagger\, T)]\!](\hat{\varphi}) := \langle \langle a \rangle \rangle = x \ \land\ y = \langle a \rangle \ \land\ \hat{\varphi}$$

$$[\![\mathbf{let}\, y = \mathbf{deref}\,(x\colon \mathbf{immut}_\psi\, \mathbf{mut}_\varphi\, T)]\!](\hat{\varphi}) := \langle \langle a, a_* \rangle \rangle = x \ \land\ y = \langle a \rangle \ \land\ \hat{\varphi}$$

$$[\![\mathbf{let}\, y = \mathbf{inj}_i^T\, x]\!](\hat{\varphi}) := \langle a \rangle = x \ \land\ y = \langle \mathbf{inj}_i^{[\![T]\!]} a \rangle \ \land\ \hat{\varphi}$$

$$\begin{aligned}
[\![\mathbf{let}\, y = (x_1, \ldots, x_n)]\!](\hat{\varphi}) &:= \langle a_1 \rangle = x_1 \ \land\ \cdots \ \land\ \langle a_n \rangle = x_n \\
&\quad \land\ y = \langle (a_1, \ldots, a_n) \rangle \ \land\ \hat{\varphi}
\end{aligned}$$

$$\begin{aligned}
[\![\mathbf{let}\, (y_1, \ldots, y_n) = x\colon A^\dagger\, T]\!](\hat{\varphi}) &:= \langle (a_1, \ldots, a_n) \rangle = x \\
&\quad \land\ y_1 = \langle a_1 \rangle \ \land\ \cdots \ \land\ y_n = \langle a_n \rangle \ \land\ \hat{\varphi}
\end{aligned}$$

$$\begin{aligned}
[\![\mathbf{let}\, (y_1, \ldots, y_n) = x\colon \mathbf{mut}_\psi\, T]\!](\hat{\varphi}) &:= \langle (a_1, \ldots, a_n), (a_{1*}, \ldots, a_{n*}) \rangle = x \\
&\quad \land\ y_1 = \langle a_1, a_{1*} \rangle \ \land\ \cdots \ \land\ y_n = \langle a_n, a_{n*} \rangle \\
&\quad \land\ \hat{\varphi}
\end{aligned}$$

$$[\![\mathbf{let}\, y = f\langle \cdots \rangle(x_1, \ldots, x_n)]\!](\hat{\varphi}) := f_{\mathrm{entry}}(x_1, \ldots, x_n, y) \ \land\ \hat{\varphi}$$

[17] Variables not appearing in $\hat{I}$ (such as a and x_{new}) are supposed to be fresh with respect to $\hat{\varphi}$.

$$\begin{aligned}
[\![\mathbf{swap}(x, y\colon \mathbf{own}\, T)]\!](\hat\varphi) \;&:=\; \langle a, a_*\rangle = x \;\land\; \langle b\rangle = y \\
&\quad\land\; x_{\text{new}} = \langle b, a_*\rangle \;\land\; y_{\text{new}} = \langle a\rangle \\
&\quad\land\; \hat\varphi[x_{\text{new}}/x,\, y_{\text{new}}/y]
\end{aligned}$$

$$\begin{aligned}
[\![\mathbf{swap}(x, y\colon \mathbf{mut}_\psi\, T)]\!](\hat\varphi) \;&:=\; \langle a, a_*\rangle = x \;\land\; \langle b, b_*\rangle = y \\
&\quad\land\; x_{\text{new}} = \langle b, a_*\rangle \;\land\; y_{\text{new}} = \langle a, b_*\rangle \\
&\quad\land\; \hat\varphi[x_{\text{new}}/x,\, y_{\text{new}}/y]
\end{aligned}$$

$$[\![\mathbf{drop}\,(x\colon A^\dagger\, T)]\!](\hat\varphi) \;:=\; \hat\varphi$$

$$[\![\mathbf{drop}\,(x\colon \mathbf{mut}_\psi\, T)]\!](\hat\varphi) \;:=\; \langle a, a_*\rangle = x \;\land\; a_* = a \;\land\; \hat\varphi$$

$$[\![\mathbf{immut}\, x]\!](\hat\varphi) \;:=\; \langle a, a_*\rangle = x \;\land\; a_* = a \;\land\; x_{\text{new}} = \langle a\rangle \;\land\; \hat\varphi[x_{\text{new}}/x]$$

$$[\![x \;\mathbf{as}\; A\, T]\!](\hat\varphi) = [\![\mathbf{intro}\, \alpha]\!](\hat\varphi) = [\![\mathbf{now}\, \alpha]\!](\hat\varphi) \;:=\; \hat\varphi$$

Interpretation of Typed Statements

The *interpretation* $[\![\hat S]\!]_{L_0, (\hat\varphi_L)_L}$ of typed statement $\hat S$ as a set of CHCs under a current label L_0 and an family of extended formulae $(\hat\varphi_L)_L$ indexed by labels is defined as follows.[18]

$$[\![\hat I; \mathbf{goto}\, L_1]\!]_{L_0, (\hat\varphi_L)_L} \;:=\; \{\, \hat\varphi_{L_0} \;\Longleftarrow\; [\![\hat I]\!](\hat\varphi_{L_1}) \,\}$$

$$[\![\mathbf{return}\, x]\!]_{L_0, (\hat\varphi_L)_L} \;:=\; \{\, \hat\varphi_{L_0} \;\Longleftarrow\; r = x \,\}$$

$$\begin{aligned}
&[\![\mathbf{match}\,(x\colon A^\dagger\, T)\,\{\mathbf{inj}_1 \tilde y_1\colon \mathbf{goto}\, L_1; \dots \mathbf{inj}_n \tilde y_n\colon \mathbf{goto}\, L_n\}]\!]_{L_0, (\hat\varphi_L)_L} \\
&\quad := \Big\{\, \hat\varphi_{L_0} \;\Longleftarrow\; \langle \mathbf{inj}_i^{[\![T]\!]} a\rangle = x \;\land\; y_i = \langle a\rangle \;\land\; \hat\varphi_{L_i} \;\Big|\; i = 1, \dots, n \text{ s.t. } \tilde y_i = y_i \,\Big\} \\
&\quad\;\cup \Big\{\, \hat\varphi_{L_0} \;\Longleftarrow\; \langle \mathbf{inj}_i^{[\![T]\!]} a\rangle = x \;\land\; \hat\varphi_{L_i} \;\Big|\; i = 1, \dots, n \text{ s.t. } \tilde y_i = - \,\Big\}
\end{aligned}$$

$$\begin{aligned}
&[\![\mathbf{match}\,(x\colon \mathbf{mut}_\psi\, T)\,\{\mathbf{inj}_1 \tilde y_1\colon \mathbf{goto}\, L_1; \dots \mathbf{inj}_n \tilde y_n\colon \mathbf{goto}\, L_n\}]\!]_{L_0, (\hat\varphi_L)_L} \\
&\quad := \Big\{\, \hat\varphi_{L_0} \;\Longleftarrow\; \langle \mathbf{inj}_i^{[\![T]\!]} a, b_*\rangle = x \;\land\; y_i = \langle a, a_*\rangle \;\land\; b_* = \mathbf{inj}_i^{[\![T]\!]} a_* \;\land\; \hat\varphi_{L_i} \\
&\qquad\quad \Big|\; i = 1, \dots, n \text{ s.t. } \tilde y_i = y_i \,\Big\} \\
&\quad\;\cup \Big\{\, \hat\varphi_{L_0} \;\Longleftarrow\; \langle \mathbf{inj}_i^{[\![T]\!]} a, b_*\rangle = x \;\land\; \hat\varphi_{L_i} \;\Big|\; i = 1, \dots, n \text{ s.t. } \tilde y_i = - \,\Big\}
\end{aligned}$$

Interpretation of Functions and Programs

The *interpretation* $[\![\mathbf{\Gamma}]\!]_{f,L}$ of variable context $\mathbf{\Gamma}$ under function name f and label L as an extended formula is defined as $f_L(x_1\colon [\![A_1\, T_1]\!], \dots, x_n\colon [\![A_n\, T_n]\!], r)$, where the variable-type pairs of the elements of $\mathbf{\Gamma}$ are enumerated as $(x_1, A_1\, T_1), \dots, (x_n, A_n\, T_n)$ using the (appropriately defined) lexicographic order on variable names.

The *interpretation* $[\![F\colon \mathbf{L}]\!]$ of function $F = \mathbf{fn}\, f\, \Sigma\, \{L_1\colon S_1 \;\cdots\; L_n\colon S_n\}$ typed by label context $\mathbf{L}$ as a set of CHCs is defined as $\bigcup_{i=1}^n [\![\hat S_i]\!]_{L_i, ([\![\mathbf{\Gamma}_L]\!]_{f,L})_L}$, where $\mathbf{\Gamma}_L$ is the variable context part of $\mathbf{L}(L)$, and typed statement $\hat S_i$ is statement S_i typed with $\mathbf{\Gamma}_{L_i}$.

Finally, we achieve the translation; the *interpretation* $[\![P\colon (\mathbf{L}_f)_f]\!]$ of program $P = F_1 \cdots F_n$ typed by $(\mathbf{L}_f)_f$ as a set of CHCs is defined as $\bigcup_{i=1}^n [\![F_i\colon \mathbf{L}_{f_i}]\!]$, where f_i is the function name of F_i for each i.

Further Simplification of CHCs

Allocating an uninterpreted predicate for each label usually results in too many uninterpreted predicates appearing in the output CHCs. Through *unfold-*

[18]In the rule for **return** x, variable r should be fresh with respect to $\hat\varphi_{L_0}$.

ing on the set of CHCs, however, the number of uninterpreted predicates will usually be greatly reduced.

3.3 Conjecture on the Correctness of the Translation

In this section, we give a conjecture on the correctness of the translation. We first introduce the *safety invariant* on a stack and a heap (with some additional information on types), which is preserved through computation. Then each *program point*, or label, of a function is given the *natural model*, which is defined in terms of computation from any stack and heap satisfying the safety invariant. Finally, we give a conjecture that the model is identical to the least fixed point specified by the CHCs that our translation generates.

3.3.1 Safety Invariant and the Preservation Lemma

Typed Stack

In order to manage the *ownership information* with regard to stacks, we define *typed stacks* here. The notable point is how to deal with *lifetime variables*; each function call can use lifetime parameters, which are embodied by local lifetime variables introduced in outer function calls. Thus on the creation of a typed stack, we give *discriminating indices* (e.g. $\alpha^{[i]}$ instead of α) to local lifetime variables of each function call, thereby get the *global lifetime context* of all local lifetime variables (and lifetime parameters of the base frame) in the stack, and embody all lifetime parameters (except those of the base frame) as local lifetime parameters.

A *typed frame* $\hat{\mathbf{F}}$ is a mapping that maps each variable name x to an invalid value $\bot$ or an item of form $a\colon A\,T$ (address a with *active* type $A\,T$) or $a\colon^{\psi} A\,T$ (address a with type $A\,T$ *shadowed* until ψ), satisfying the condition that only a finite number of variable names x are not mapped to an invalid value.

A *typed stack* $\hat{\mathbf{S}}/\mathbf{\Psi}$ consists of a *body* $\hat{\mathbf{S}}$ and a *global lifetime context* $\mathbf{\Psi}^*$ (a finite set of lifetime variables); the body $\hat{\mathbf{S}}$ has form $[f_0, L_0]\,\hat{\mathbf{F}}_0;\ [f_1, L_1]\,x_1, \hat{\mathbf{F}}_1;\ \dots;\ [f_n, L_n]\ x_n, \hat{\mathbf{F}}_n;$.

The *type modification* $\mathrm{TM}(\mathbf{F}\,|\,\mathbf{\Gamma}, y)$ of a frame $\mathbf{F}$ with a variable context $\mathbf{\Gamma}^*$ (which has undergone substitution on lifetime variables) is a typed frame defined as follows, under the restriction that the set of variables x such that $x_{\mathbf{F}} \neq \bot$ holds is equal to the set of variables x that appears in $\mathbf{\Gamma}^*$ and is not y.

$$x_{\mathrm{TM}(\mathbf{F}\,|\,\mathbf{\Gamma},y)} \ :=\ \begin{cases} \bot & (x_{\mathbf{F}} = \bot) \\ x_{\mathbf{F}}\colon A\,T & (x\colon A\,T \in \mathbf{\Gamma}) \\ x_{\mathbf{F}}\colon^{\psi} A\,T & (x\colon^{\psi} A\,T \in \mathbf{\Gamma}) \end{cases}$$

The *type modification* $\mathrm{TM}(\mathbf{S}\,|\,\mathbf{\Sigma}, (\mathbf{L}_f)_f)$ of a stack $\mathbf{S}$ with a function signature context $\mathbf{\Sigma}$ and an indexed family of label contexts $(\mathbf{L}_f)_f$ is a typed stack defined as follows. *To sum up*, each typed frame $\hat{\mathbf{F}}_i$ is obtained from the frame $\mathbf{F}_i$ with type modification by the variable context $\mathbf{\Gamma}_i$ under substitution on lifetime variables, and the global lifetime context $\mathbf{\Psi}^*_n$ is obtained by iteratively swelling a lifetime context from the base frame. θ or $[\psi_1/\alpha_1, \dots, \psi_k/\alpha_k]$ stands for a substitution on lifetime variables; $\theta\theta'$ stands for a composition of substitutions; $\psi\theta$ and $\mathbf{\Gamma}\theta$ stand for a result of a substitution.

$$\begin{aligned} &\mathrm{TM}\big([f_n, L_n]\,\mathbf{F}_n;\ [f_{n-1}, L_{n-1}]\,\langle\psi^{n-1}_1, \dots, \psi^{n-1}_{m_{n-1}}\rangle\, x_{n-1}, \mathbf{F}_{n-1};\\ &\qquad \dots;\ [f_0, L_0]\,\langle\psi^0_1, \dots, \psi^0_{m_0}\rangle\, x_0, \mathbf{F}_0;\ \big|\ \mathbf{\Sigma}, (\mathbf{L}_f)_f\big)\\ &\quad :=\ [f_n, L_n]\,\hat{\mathbf{F}}_n;\ [f_{n-1}, L_{n-1}]\,x_{n-1}, \hat{\mathbf{F}}_{n-1};\ \dots;\ [f_0, L_0]\,x_0, \hat{\mathbf{F}}_0;/\mathbf{\Psi}^*_n \end{aligned}$$

$$\begin{array}{rcl}
\multicolumn{3}{l}{\text{where } \mathbf{L}_{f_i}(L_i) = ((\Psi_i, R_i), \mathbf{\Gamma}_i),\ \ \mathbf{\Sigma}(f_i) = f_i\langle \alpha_1^{i,\mathrm{ex}}, \dots, \alpha_{m_i}^{i,\mathrm{ex}} \,|\, \cdots\rangle(\cdots)} \\
\text{and} \quad \Psi_{\mathrm{ex}}^i & := & \{\alpha_1^{i,\mathrm{ex}}, \dots, \alpha_{m_i}^{i,\mathrm{ex}}\} \\
(\Psi_i^\dagger, R_i^\dagger) & := & \big((\Psi_i - \Psi_{\mathrm{ex}}^i)^{[i]}, \big(R_i \cap (\Psi_i - \Psi_{\mathrm{ex}}^i) \times (\Psi_i - \Psi_{\mathrm{ex}}^i)\big)^{[i]}\big) \\
\mathbf{\Psi}_0^* = (\Psi_0^*, R_0^*) & := & (\Psi_0^{[0]}, R_0^{[0]}) \\
\text{where } \{\alpha_1, \dots, \alpha_k\}^{[i]} & := & \alpha_1^{[i]}, \dots, \alpha_k^{[i]} \\
\{(\alpha_1, \beta_1), \dots, (\alpha_k, \beta_k)\}^{[i]} & := & \{(\alpha_1^{[i]}, \beta_1^{[i]}), \dots, (\alpha_k^{[i]}, \beta_k^{[i]})\} \\
\mathbf{\Psi}_{i+1}^* = (\Psi_{i+1}^*, R_{i+1}^*) & := & (\Psi_i^* + \Psi_{i+1}^\dagger,\ R_i^* + R_{i+1}^\dagger + R_{i+1}^\dagger \times R_i^*) \\
\theta_0^{\mathrm{in}} = \theta_0 & := & [\alpha_1^{[0]}/\alpha_1, \dots, \alpha_k^{[0]}/\alpha_k] \text{ where } \Psi_0^* = \{\alpha_1^{[0]}, \dots, \alpha_k^{[0]}\} \\
\theta_{i+1}^{\mathrm{in}} & := & \theta_i^{\mathrm{in}}[\alpha_1^{[i]}/\alpha_1, \dots, \alpha_k^{[i]}/\alpha_k] \text{ where } \Psi_i^\dagger = \{\alpha_1^{[i]}, \dots, \alpha_k^{[i]}\} \\
\theta_{i+1} & := & \theta_{i+1}^{\mathrm{in}}[\psi_1^i \theta_i/\alpha_1^{i,\mathrm{ex}}, \dots, \psi_{m_i}^i \theta_i/\alpha_{m_i}^{i,\mathrm{ex}}] \\
\hat{\mathbf{F}}_i & := & \mathrm{TM}(\,\mathbf{F}_i \,|\, \mathbf{\Gamma}_i \theta_i,\ x_i\,)
\end{array}$$

We refine relation $P\colon(\mathbf{L}_f)_f \vdash \mathbf{S}/\mathbf{H} \to \left\{{}^{\mathbf{S}'}_{a}\right\}/\mathbf{H}'$ for typed stacks $P\colon(\mathbf{L}_f)_f \vdash \hat{\mathbf{S}}/\mathbf{\Psi}^*/\mathbf{H} \to \left\{{}^{\hat{\mathbf{S}}'/\mathbf{\Psi}^{*\prime}}_{a}\right\}/\mathbf{H}'$.[19]

Ownership Footprint

In our approach, the ownership information is summarized for each address as a *multiset of (pointer) authorities* possibly with shadowing information ("shadowed until lifetime ψ"); information such as "which variable has a control over which address" or "which address is borrowed from which variable" is ignored here. The summary of ownership information on all relevant addresses is here called an *ownership footprint*.

The *ownership footprint* $\mathrm{OF}_{\mathbf{H}}(a\colon A\,T)$ of an address a with type $A\,T$ under heap $\mathbf{H}$ is a finite multiset of elements of form $a'\colon A'$ (address a' is under a control of (pointer) authority A'). It is recursively defined as follows.[20] $\{\!|x_1, \dots, x_n|\!\}$ is an extensional representation of a multiset. Note that the ownership footprint is not defined for $\bot\colon A\,T$ (an invalid address); thus, if $\mathrm{OF}_{\mathbf{H}}(a\colon A\,T)$ is successfully defined, then the whole data structure of $a\colon A\,T$ can be traced without finding an invalid address.

$$\begin{array}{rcl}
\mathrm{OF}_{\mathbf{H}}(a\colon A\,\mathbf{int}) & := & \{\!|a\colon A|\!\} \ \text{ when } *_{\mathbf{H}}a \neq \bot \\
\mathrm{OF}_{\mathbf{H}}(a\colon A\,A'\,T) & := & \{\!|a\colon A|\!\} + \mathrm{OF}_{\mathbf{H}}(*_{\mathbf{H}}a\colon A''\,T) \ \text{ where } \ A'' = A \wedge A' \\
\mathrm{OF}_{\mathbf{H}}(a\colon A\, \sum_{i=1}^n T_i) & := & \{\!|a\colon A|\!\} + \mathrm{OF}_{\mathbf{H}}(a+1\colon A\,T_j) \\
 & & \quad + \{\!|(a+k)\colon A \,|\, k = 1 + \#T_j, \dots, \#\sum_{i=1}^n T_i - 1|\!\} \\
 & & \text{where } \ j = *_{\mathbf{H}}a \\
 & & \text{and } \ *_{\mathbf{H}}(a+k) = 0 \ \ (k = 1 + \#T_j, \dots, \#\sum_{i=1}^n T_i - 1) \\
\mathrm{OF}_{\mathbf{H}}(a\colon A\, \prod_{i=1}^n T_i) & := & \sum_{i=1}^n \mathrm{OF}_{\mathbf{H}}((a + \sum_{k=1}^{i-1} \#T_k)\colon A\,T) \\
\mathrm{OF}_{\mathbf{H}}(a\colon A\,\mu X.T) & := & \mathrm{OF}_{\mathbf{H}}(a\colon A\,T[\mu X.T/X])
\end{array}$$

Using this, the *ownership footprint* $\mathrm{OF}_{\mathbf{H}}(\hat{\mathbf{F}})$ of a typed frame $\hat{\mathbf{F}}$ under heap $\mathbf{H}$ is a finite multiset of elements of form $a\colon A$ (address a is under an *active* control of authority A) and $a\colon^{\psi} A$ (address a is under an *inactive* control of authority A,

[19]Information on lifetime arguments is omitted in typed stacks unlike stacks, but it is not relevant here.

[20]$+$ and $\sum$ indicate the sum or the disjoint union of multisets (that is, the multiset with the sum multiplicity given to each item).

which will be activated at lifetime ψ), is defined as follows.

$$\begin{aligned} \mathrm{OF}_{\mathbf{H}}(\hat{\mathbf{F}}) \ &:= \sum_{x_{\hat{\mathbf{F}}}=a:A\,T} \mathrm{OF}_{\mathbf{H}}(a\colon A\,T) + \sum_{x_{\hat{\mathbf{F}}}=a:^{\psi}A\,T} \big(\mathrm{OF}_{\mathbf{H}}(a\colon A\,T)\big)^{\psi} \\ \text{where } \{\!|a_1\colon A_1,\ldots,a_n\colon A_n|\!\}^{\psi} \ &:= \ \{\!|a_1\colon^{\psi} A_1,\ldots,a_n\colon^{\psi} A_n|\!\} \end{aligned}$$

Finally, the *ownership footprint* $\mathrm{OF}_{\mathbf{H}}(\hat{\mathbf{S}})$ of a body of a typed stack $\hat{\mathbf{S}}$ under heap $\mathbf{H}$ is a finite multiset of elements of form $a\colon A$ and $a\colon^{\psi} A$ defined as follows.

$$\mathrm{OF}_{\mathbf{H}}\big([f_0,L_0]\,\hat{\mathbf{F}}_0;[f_1,L_1]\,x_1,\hat{\mathbf{F}}_1;\ldots;[f_n,L_n]\,x_n,\hat{\mathbf{F}}_n;\big) \ := \ \sum_{i=1}^{n}\mathrm{OF}_{\mathbf{H}}(\hat{\mathbf{F}}_i)$$

Safety Invariant and the Preservation Lemma

The *safety invariant* of a typed stack $\hat{\mathbf{S}}/\mathbf{\Psi}^*$ with a heap $\mathbf{H}$ is the conjunction of the following conditions: (1) the ownership footprint $\mathrm{OF}_{\mathbf{H}}(\hat{\mathbf{S}})$ is defined, and (2) in regard to $\mathrm{OF}_{\mathbf{H}}(\hat{\mathbf{S}})$, (2a) for each address a, there should not be two or more elements of form either $a\colon \mathbf{own}$ or $a\colon \mathbf{mut}_{\psi}$; (2b) for each address a, if there is an element of form $a\colon \mathbf{immut}_{\psi}$, there should not be any element of form $a\colon \mathbf{own}$ or $a\colon \mathbf{mut}_{\psi'}$; (2c) for each address a, if there is an element of form $a\colon^{\psi} A$, there should not be any element of form $a\colon \mathbf{own}$; (2d) for each address a, if there is an element of form $a\colon^{\psi} \mathbf{own}$, there should not be any other element of form $a\colon^{\varphi} \mathbf{own}$; (2e) for each address a, and for any element of form $a\colon^{\psi} A$ and any other element of form $a\colon B_{\varphi}$ or $a\colon^{\psi'} B_{\varphi}$ satisfying $\psi <_{\mathbf{\Psi}^*} \varphi$, B should be $\mathbf{immut}$ and A should be of form $\mathbf{immut}_{\varphi'}$; and (2f) for each address a, and for any element of form $a\colon^{\psi} A$ and any other element of form $a\colon^{\psi'} A'$ such that $\psi \sim_{\mathbf{\Psi}^*} \psi'$,[21] A is of form $\mathbf{immut}_{\varphi}$ and A' also is of form $\mathbf{immut}_{\varphi'}$. The properties (1), (2a) and (2b) are of main interest; (2c), (2d), (2e) and (2f) support (2a) and (2b).

Finally, we achieve the preservation lemma of the safety invariant.

Lemma 1 *For any program P typed by $(\mathbf{L}_f)_f$, typed stacks $\hat{\mathbf{S}},\hat{\mathbf{S}}'$, and heaps $\mathbf{H},\mathbf{H}'$ such that $P\colon(\mathbf{L}_f)_f \vdash \hat{\mathbf{S}}/\mathbf{H} \to \hat{\mathbf{S}}'/\mathbf{H}'$ holds, if $\hat{\mathbf{S}}$ with $\mathbf{H}$ satisfies the safety invariant, then so does $\hat{\mathbf{S}}'$ with $\mathbf{H}'$.* ■

Proof Simply by checking rules of type checking and operational semantics for each type of instructions and statements. The crucial point is, mutual (re)borrowing (with instruction $\mathbf{let}\, y = \mathbf{mut\,bor}\, x\ \mathbf{till}\ \psi$), lifetime elimination (with instruction $\mathbf{now}\ \alpha$), alteration of mutable references into immutable references (with instruction $\mathbf{immut}\, x$), and replication of immutable references (with instructions $\mathbf{let}\, y = \mathbf{copy}\, x$ and $\mathbf{let}\, y = \mathbf{repl}\, x$) all preserve the safety invariant. □

3.3.2 Natural Model Built on Operational Semantics

The *natural model* of each program point, or label, of a function is defined here in terms of computation from a typed stack with a heap satisfying the safety invariant. This model may seem rather hard to understand for program points with shadowed variables or (mutable/immutable) references, but has a clear meaning for program points without them, which simply justifies the model.

[21] $\sim_{\mathbf{\Psi}^*}$ means that both $\leq_{\mathbf{\Psi}^*}$ and $\geq_{\mathbf{\Psi}^*}$ hold.

Traces

A finite sequence of pairs of a typed stack and a heap $((\hat{\mathbf{S}}_i/\mathbf{\Psi}^*_i, \mathbf{H}_i))_{i=1}^n$ $(n \geq 1)$ is said to be a *trace* $\mathcal{T}$ from label L of a function f in a program P typed $(\mathbf{L}_f)_f$, if (0) $\hat{\mathbf{S}}_1$ has form $[f, L]\,\hat{\mathbf{F}}$; (it has only one typed frame $\hat{\mathbf{F}}$ and there is no frame below it), (1) typed stack $\hat{\mathbf{S}}_1/\mathbf{\Psi}^*_1$ with heap $\mathbf{H}_1$ satisfies the safety invariant, (2) $P\colon (\mathbf{L}_f)_f \vdash \hat{\mathbf{S}}_i/\mathbf{H}_i \longrightarrow \hat{\mathbf{S}}_{i+1}/\mathbf{H}_{i+1}$ holds for $i = 1, \dots, n-1$, and (3) $P\colon (\mathbf{L}_f)_f \vdash \hat{\mathbf{S}}_n/\mathbf{H}_n \longrightarrow a/\mathbf{H}_n$ holds.

The *elimination moment* $\mathrm{EM}_{\mathcal{T}}(\psi)$ of lifetime ψ on a trace $\mathcal{T} = ((\hat{\mathbf{S}}_i/\mathbf{\Psi}^*_i, \mathbf{H}_i))_{i=1}^n$ is the smallest i such that some of the lifetime variables appearing in ψ is not contained in $\mathbf{\Psi}^*_i$.

The *completion moment* $\mathrm{CM}_{\mathcal{T}}(a)$ of address a on a trace $\mathcal{T} = ((\hat{\mathbf{S}}_i/\mathbf{\Psi}^*_i, \mathbf{H}_i))_{i=1}^n$ is an integer $1, \dots, n$ or infinity ∞ given as follows: from elements of form $a\colon^{\psi} A$ in $\mathrm{OF}_{\mathbf{H}_1}(\hat{\mathbf{S}}_1)$, take the one with the smallest ψ with respect to $\mathbf{\Psi}^*_1$, and return $\mathrm{EM}_{\mathcal{T}}(\psi)$; if such an element does not exist, return ∞.

Translating Concepts of Operational Semantics into Logic

The *exterior interpretation* $[\![*a\colon T]\!]^{\mathrm{ex}}_{OF,\mathbf{H}}$ of the data at address a typed T under ownership footprint OF and heap $\mathbf{H}$ as a term is recursively defined as follows. The variables of form $\mathbf{ret}_a$ are for special use.

$$[\![*a\colon \mathbf{int}]\!]^{\mathrm{ex}}_{OF,\mathbf{H}} \;:=\; \begin{cases} \mathbf{ret}_a & (a\colon \mathbf{mut}_\psi \in OF) \\ *_{\mathbf{H}}a & (\text{otherwise}) \end{cases}$$

$$[\![*a\colon A^{\dagger}\, T]\!]^{\mathrm{ex}}_{OF,\mathbf{H}} \;:=\; \begin{cases} \mathbf{ret}_a & (a\colon \mathbf{mut}_\psi \in OF) \\ \langle [\![*(*_{\mathbf{H}}a)\colon T]\!] \rangle & (\text{otherwise}) \end{cases}$$

$$[\![*a\colon \mathbf{mut}_\psi\, T]\!]^{\mathrm{ex}}_{OF,\mathbf{H}} \;:=\; \begin{cases} \mathbf{ret}_a & (a\colon \mathbf{mut}_\varphi \in OF) \\ \langle [\![*(*_{\mathbf{H}}a)\colon T]\!],\, \mathbf{ret}_{*_{\mathbf{H}}a} \rangle & (\text{otherwise}) \end{cases}$$

$$[\![*a\colon \textstyle\sum_{i=1}^n T_i]\!]^{\mathrm{ex}}_{OF,\mathbf{H}} \;:=\; \begin{cases} \mathbf{ret}_a & (a\colon \mathbf{mut}_\psi \in OF) \\ \mathbf{inj}^{[\![\sum_{i=1}^n T_i]\!]}_{*_{\mathbf{H}}a} [\![*(a+1)\colon T_{*_{\mathbf{H}}a}]\!]^{\mathrm{ex}}_{OF,\mathbf{H}} & (\text{otherwise}) \end{cases}$$

$$[\![*a\colon \textstyle\prod_{i=1}^n T_i]\!]^{\mathrm{ex}}_{OF,\mathbf{H}} \;:=\; \left([\![*(a + \textstyle\sum_{i=1}^{k-1} \#T_i)\colon T_k]\!]^{\mathrm{ex}}_{OF,\mathbf{H}}\right)_{k=1}^n$$

$$[\![*a\colon \mu X.T]\!]^{\mathrm{ex}}_{OF,\mathbf{H}} \;:=\; [\![*a\colon T[\mu X.T/X]]\!]^{\mathrm{ex}}_{OF,\mathbf{H}}$$

The *interpretation* $[\![a_1 \cdots a_{\#T}\colon T]\!]_{\mathcal{T}}$ of a sequence of integers $a_1 \cdots a_{\#T}$ with type T under trace $\mathcal{T} = ((\hat{\mathbf{S}}_i/\mathbf{\Psi}^*_i, \mathbf{H}_i))_{i=1}^n$ is recursively defined as follows.

$$[\![n\colon \mathbf{int}]\!]_{\mathcal{T}} \;:=\; n$$

$$[\![a\colon A^{\dagger}\, T]\!]_{\mathcal{T}} \;:=\; \langle [\![*_{\mathbf{H}_1}a \cdots *_{\mathbf{H}_1}(a + \#T - 1)\colon T]\!]_{\mathcal{T}} \rangle$$

$$[\![a\colon \mathbf{mut}_\psi\, T]\!]_{\mathcal{T}} \;:=\; \langle [\![*_{\mathbf{H}_1}a \cdots *_{\mathbf{H}_1}(a + \#T - 1)\colon T]\!]_{\mathcal{T}},\, x_* \rangle$$

$$\text{where } x_* = \begin{cases} [\![*_{\mathbf{H}_k}a \cdots *_{\mathbf{H}_k}(a + \#T - 1)\colon T]\!]_{((\hat{\mathbf{S}}_i/\mathbf{\Psi}^*_i, \mathbf{H}_i))_{i=k}^n} & (\mathrm{CM}_{\mathcal{T}}(a) = k) \\ [\![*a\colon T]\!]^{\mathrm{ex}}_{\mathrm{OF}(\hat{\mathbf{S}}),\mathbf{H}_n} & (\mathrm{CM}_{\mathcal{T}}(a) = \infty) \end{cases}$$

$$[\![j a_1 \cdots a_{\#T_j} 0 \cdots 0\colon \textstyle\sum_{i=1}^n T_i]\!]_{\mathcal{T}} \;:=\; \mathbf{inj}^{[\![\sum_{i=1}^n T_i]\!]}_j [\![a_1 \cdots a_{\#T_j}\colon T_j]\!]_{\mathcal{T}}$$

$$[\![a_1 \cdots a_{\sum_{i=1}^n \#T_i}\colon \textstyle\prod_{i=1}^n T_i]\!]_{\mathcal{T}} \;:=\; \left([\![a_{\sum_{k=1}^{j-1} \#T_k + 1} \cdots a_{\sum_{k=1}^{j} \#T_k}\colon T_j]\!]_{\mathcal{T}}\right)_{j=1}^n$$

$$[\![a_1 \cdots a_{\#\mu X.T}\colon \mu X.T]\!]_{\mathcal{T}} \;:=\; [\![a_1 \cdots a_{\#\mu X.T}\colon T[\mu X.T/X]]\!]_{\mathcal{T}}$$

Natural Model

For each label L in function f in a program P, we define its *natural model* as a predicate f_L defined as follows (the number and the sorts of the arguments are the same as the corresponding uninterpreted predicates of the CHCs).

For any variable-free terms $t_1, \dots, t_n$, the formula $f_L(t_1, \dots, t_n, r)$ is true if and only if there exist a trace $\mathcal{T} = ((\hat{\mathbf{S}}_i / \mathbf{\Psi}^*_i, \mathbf{H}_i))_{i=1}^m$ (from L in f) and a substitution θ (of variables of form $\mathbf{ret}_a$ into variable-free terms), such that $t_i = t_i^{\mathcal{T}} \theta$ holds for each i and $r = r_{\mathcal{T}} \theta$ holds, where $t_i^{\mathcal{T}}$ and $r_{\mathcal{T}}$ are defined as follows:

$$t_i^{\mathcal{T}} \;:=\; \begin{cases} [\![a \colon A\, T]\!]_{\mathcal{T}} & (x_{i\,\hat{\mathbf{F}}_1} = a \colon A\, T) \\ [\![a \colon A\, T]\!]_{((\hat{\mathbf{S}}_i / \mathbf{\Psi}^*_i, \mathbf{H}_i))_{i=\mathrm{EM}_{\mathcal{T}}(\psi)}^m} & (x_{i\,\hat{\mathbf{F}}_1} = a \colon^{\psi} A\, T) \end{cases}$$

$$r_{\mathcal{T}} \;:=\; [\![x^{\dagger}_{\hat{\mathbf{F}}_m}]\!]_{((\hat{\mathbf{S}}_m / \mathbf{\Psi}^*_m, \mathbf{H}_m))}$$

where $\hat{\mathbf{F}}_1$ satisfies $\hat{\mathbf{S}}_1 = [f, L]\, \hat{\mathbf{F}}_1$; holds and the variables mapped to a valid address in $\mathbf{F}_1$ are enumerated as $x_1, \dots, x_n$ in the lexicographic order; and $\hat{\mathbf{F}}_m$ satisfies $\hat{\mathbf{S}}_m = [f, L']\, \hat{\mathbf{F}}_m$; holds and the only variable mapped to a valid address is $x^{\dagger}$.

3.3.3 Conjecture on the Correctness

In this subsection, we give a conjecture that the natural model built on operational semantics is identical to the least fixed point specified by the CHCs that our translation generates; simply put, we describe the conjecture on the correctness of our translation.

Theorem 2 *For any label L in function f and any variable-free terms $t_1, \dots, t_n$ such that $f_L(t_1, \dots, t_n)$ holds in the natural model, the formula $f_L(t_1, \dots, t_n)$ can also be derived with resolution for the CHCs generated by our translation.* ■

Proof By straightforward induction on the length of a trace that supports $f_L(t_1, \dots, t_n)$ in the natural model. □

Conjecture 3 *Every CHC (i.e. formula of form $f_L(x_1, \dots, x_n, r) \impliedby \cdots$) generated by our translation is true under the natural model.* ■

With this conjecture proved, the least fixed points specified by the CHCs our translation generated will be proved to be identical to the natural model.

Chapter 4

Experiment and Discussion

This chapter shows the results of an experiment and discusses the verification performance achieved with our method.

4.1 Experiment on the Verification Performances

We compared the verification performances of two versions of CHCs, sort-carve-list (given by *our translation*) and sort-carve-list-a (given by a *clever, non-trivial address-based translation*), taken from Example C. The latter representation can be seen as an idealized form of the conventional, address-based method. We verified the following properties (for address-based versions, add '-a' appropriately) on the Z3 engine.[1]

(calc-1) sort-carve-list$([1],[1])$.
(calc-2) sort-carve-list$([3,1],[2,1])$.
(calc-3) sort-carve-list$([3,1,5],[2,1,3])$.
(calc-4) sort-carve-list$([7,1,3,5],[4,1,2,3])$.
(calc-5) sort-carve-list$([9,1,3,5,7],[5,1,2,3,4])$.
(back) $\exists lx.\,$sort-carve-list$(lx,[2,1,3])$.
(find) $\exists x,y.\,$sort-carve-list$([x,3,4],[y,3,2])$.
(size) $\neg\left(\exists x_1,x_2,x_3,x_4,y_1,y_2,y_3.\,\text{sort-carve-list}([x_1,x_2,x_3,x_4],[y_1,y_2,y_3])\right)$.
(single) $\forall x,y.\,\text{sort-carve-list}([x],[y]) \implies x=y$.
(double-1) $\forall x,y.\,x_1 \le x_2 \land \text{sort-carve-list}([x_1,x_2],[y_1,y_2]) \implies x_1=y_1$.
(double-2) $\forall x,y.\,x_1 > x_2 \land \text{sort-carve-list}([x],[y]) \implies x_2=y_2$.

We also wrote equivalent C code[2] and verified the properties with *SeaHorn*. Moreover, we wrote equivalent Java code and checked the properties for *JayHorn* (version 0.6, the latest version as of January 29, 2019); it unnaturally quickly finished some verification tasks, and the results turned out to be highly unreliable. With suspicion, we checked $[1]=[]$ and it was judged to be *true* by JayHorn (furthermore, $[1]=[2]$ was not judged in ten minutes). Thus we simply excluded JayHorn from the experiment table.

The experiment was carried out on MacBook Air 13-inch, Early 2015 with Processor 2.2 GHz Intel Core i7. For the Z3 engine, we used version 4.8.4, the latest stable release of Z3 as of January 29, 2019, and Spacer was used for the CHC solver. For SeaHorn, we used the docker version tagged 'latest' as of January 29, 2019.

Table shows the time spent on each verification problem by each method (our method, the address-based method, or SeaHorn); when the process did not end

[1]Properties are written in human-friendly forms. In order to verify (back), for example, we just need to add CHC $\bot \Longleftarrow$ sort-carve-list$(lx,[2,1,3])$ and check the unsatisfiability.

[2]Unlike Rust code, ownership is not explicitly guaranteed.

in three minutes, it is regarded as 'timeout'. For our method and the address-based method, the time was taken from ':time.spacer.solve' (the total time that Spacer CHC solver spent) printed with the configuration ':print-statistics true' on the Z3 engine. For SeaHorn, we measured the time for command `sea horn -solve *.bc` (excluded the preprocessing time of `sea fe *.c -o *.bc`); actually, SeaHorn fell into timeout for all problems except (calc-1).

In summary, *our method showed the best performance for every problem.*

	Time (s)		
Problem	Our Method	Address-based	SeaHorn
(calc-1)	**0.03**	0.05	1.55
(calc-2)	**0.07**	8.87	timeout
(calc-3)	**0.15**	1.74	timeout
(calc-4)	**0.27**	timeout	timeout
(calc-5)	**0.62**	timeout	timeout
(back)	**0.14**	0.32	timeout
(find)	**0.40**	timeout	timeout
(size)	**0.05**	0.26	timeout
(single)	**0.24**	0.36	timeout
(double-1)	**0.70**	timeout	timeout
(double-2)	**1.03**	timeout	timeout

Table: Results of the Experiment

4.2 Useful Invariants under CHCs Obtained by the Translation

With CHCs obtained by our translation, some kinds of non-trivial and general properties can be verified by giving quite simple invariants, as indicated in the following three cases. Since description of such properties and invariants requires (primitive) recursively defined relations and functions, common CHC solvers such as Spacer cannot find those invariants, but sophisticated CHC solving techniques may find the invariants; absence of array theory can possibly be an advantage in designing CHC solving techniques.

Case 1: Monotonicity of sort-carve-list

Under the CHCs given by Example C, suppose we want to verify that sort-carve-list(lx, ly) implies that lx and ly are of the same length and that an each element of lx is no less than the corresponding entry of ly. The property is expressed as the following additional CHC (query), employing a (primitive) recursively defined relation dec-list (the relation is *newly introduced out of CHCs*).

$$\begin{aligned} \text{dec-list}([], ly) &:\Longleftrightarrow\ ly = [] \\ \text{dec-list}([x|lx'], ly) &:\Longleftrightarrow\ \exists\, y, ly'.\ ly = [y|ly'] \ \wedge\ x \geq y \ \wedge\ \text{dec-list}(lx', ly') \end{aligned}$$

$$\text{dec-list}(lx, ly) \ \Longleftarrow\ \text{sort-carve-list}(lx, ly)$$

The new set of CHCs with this query can be satisfied by the following evaluation model using a recursively defined relation all-mut-dec. Note that we do not need induction on dec-list and all-mut-dec in order to check validity of the evaluation.

$$\begin{aligned} \text{all-mut-dec}([]) &:\Longleftrightarrow\ \top \\ \text{all-mut-dec}([\langle x, x_* \rangle | lmx']) &:\Longleftrightarrow\ x \geq x_* \ \wedge\ \text{all-mut-dec}(lmx') \end{aligned}$$

$$\begin{aligned}
\text{split-mut-list}(\langle lx, lx_* \rangle, lmx) &:\Longleftrightarrow \big(\text{dec-list}(lx, lx_*) \ \Longleftarrow\ \text{all-mut-dec}(lmx)\big) \\
\text{insert-list}(\langle x, x_* \rangle, lmy, lmz) &:\Longleftrightarrow \big(x \geq x_* \ \land\ \text{all-mut-dec}(lmy) \\
&\qquad\qquad \Longleftarrow\ \text{all-mut-dec}(lmz)\big) \\
\text{sort-list}(lmx, lmy) &:\Longleftrightarrow \big(\text{all-mut-dec}(lmx) \ \Longleftarrow\ \text{all-mut-dec}(lmy)\big) \\
\text{carve-list}(n, lmx) &:\Longleftrightarrow \big(n \geq 0 \ \Longrightarrow\ \text{all-mut-dec}(lmx)\big) \\
\text{sort-carve-list}(lx, ly) &:\Longleftrightarrow \text{dec-list}(lx, ly)
\end{aligned}$$

On the other hand, giving invariants to verify the property $\text{dec-list}(lx, ly) \Longleftarrow \text{sort-carve-list-a}(lx, ly)$ is not easy (in addition, it is probably impossible to avoid inductive discussion on dec-list), since we need to trace correspondence between arrays and lists.

Case 2: Monotonicity of carve-bfs-tree

Case 2 employs a technique similar to the one used in Case 1, but the situation is a little more complex. Under the CHCs given by Example D, let us verify that $\text{carve-bfs-tree}(tx, ty)$ implies that tx and ty are of the same form (i.e. equal modulo integer elements) and each integer element of tx is no less than the corresponding element of ty. The property is expressed as the following query employing a recursively defined relation dec-tree.

$$\begin{aligned}
\text{dec-tree}(\text{Leaf}, ty) &:\Longleftrightarrow ty = \text{Leaf} \\
\text{dec-tree}(\text{Node}(tx', x, tx''), ty) &:\Longleftrightarrow \exists\, ty', y, ty''. \\
&\qquad ty = \text{Node}(ty', y, ty'') \ \land\ x \geq y \\
&\quad \land\ \text{dec-tree}(tx', ty') \\
&\quad \land\ \text{dec-tree}(tx'', ty'')
\end{aligned}$$

$$\text{dec-tree}(tx, ty) \ \Longleftarrow\ \text{carve-bfs-tree}(tx, ty)$$

The new set of CHCs with the query can be satisfied by the following evaluation model described with a recursively defined relation all-dec-tree and a helper relation both-all-dec-tree.

$$\begin{aligned}
\text{all-dec-tree}([\,]) &:\Longleftrightarrow \top \\
\text{all-dec-tree}([\langle tx, tx_* \rangle | lmtx']) &:\Longleftrightarrow \text{dec-tree}(tx, tx_*) \ \land\ \text{all-dec-tree}(lmtx')
\end{aligned}$$

$$\text{both-all-dec-tree}((lmtx, lmty)) :\Longleftrightarrow \text{all-dec-tree}(lmtx) \ \land\ \text{all-dec-tree}(lmty)$$

$$\begin{aligned}
&\text{push-queue}(que, \langle tx, tx_* \rangle, que') :\Longleftrightarrow \\
&\quad \big(\text{both-all-dec-tree}(que) \ \land\ \text{dec-tree}(tx, tx_*) \ \Longleftrightarrow\ \text{both-all-dec-tree}(que')\big)
\end{aligned}$$

$$\begin{aligned}
\text{reverse-list}(lmtx, lmty, lmtz) &:\Longleftrightarrow \big(\text{all-dec-tree}(lmtx) \ \land\ \text{all-dec-tree}(lmty) \\
&\qquad \Longleftarrow\ \text{all-dec-tree}(lmtz)\big)
\end{aligned}$$

$$\begin{aligned}
\text{pop-queue}(que, \text{None}) &:\Longleftrightarrow \text{both-all-dec-tree}(que) \\
\text{pop-queue}\big(que, \text{Some}(\langle tx, tx_* \rangle, que')\big) &:\Longleftrightarrow \big(\text{both-all-dec-tree}(que) \\
&\qquad \Longleftarrow\ \text{dec-tree}(tx, tx_*) \\
&\qquad\quad \land\ \text{both-all-dec-tree}(que')\big) \\
\text{go-bfs-tree}(n, que) &:\Longleftrightarrow \big(n \geq 0 \ \Longrightarrow\ \text{both-all-dec-tree}(que)\big) \\
\text{carve-bfs-tree}(tx, ty) &:\Longleftrightarrow \text{dec-tree}(tx, ty)
\end{aligned}$$

Case 3: Total Decrease on sort-carve-list

Under the CHCs given by Example C, let us verify that $\text{sort-carve-list}(lx, ly)$ implies that the sum of lx minus the sum of $(n-1)n/2$ where n is the size (or length) of lx (and also of ly). The property is described as follows employing recursively defined functions size-list and sum-list and a helper function $\triangle(n)$ meaning the n-th triangular number (setting the zeroth and the first as 0, and the second as 1).

$$\begin{aligned}\text{size-list}([]) &:= 0\\ \text{size-list}([x|lx']) &:= 1 + \text{size-list}(lx')\end{aligned}$$

$$\begin{aligned}\text{sum-list}([]) &:= 0\\ \text{sum-list}([x|lx']) &:= x + \text{sum-list}(lx')\end{aligned}$$

$$\triangle(n) := (n-1)n/2$$

$$\text{sum-list}(lx) - \text{sum-list}(ly) = \triangle(\text{size-list}(lx)) \iff \text{sort-carve-list}(lx, ly)$$

The new set of CHCs with the query can be satisfied by the following evaluation model described with recursively defined functions cur-list and ret-list and a helper function $\triangle(k, n)$.[3]

$$\begin{aligned}\text{cur-list}([]) &:= []\\ \text{cur-list}([\langle x, x_* \rangle | lmx']) &:= [x | \text{cur-list}(lmx')]\end{aligned}$$

$$\begin{aligned}\text{ret-list}([]) &:= []\\ \text{ret-list}([\langle x, x_* \rangle | lmx']) &:= [x_* | \text{ret-list}(lmx')]\end{aligned}$$

$$\triangle(k, n) := (2k + n - 1)n/2$$

$$\text{split-mut-list}(\langle lx, lx_* \rangle, lmx) :\iff lx = \text{cur-list}(lmx) \ \wedge\ lx_* = \text{ret-list}(lmx)$$

$$\begin{aligned}&\text{insert-list}(\langle x, x_* \rangle, lmy, lmz)\\ &\quad :\iff x + \text{sum-list}(\text{cur-list}(lmy)) = \text{sum-list}(\text{cur-list}(lmz))\\ &\qquad \wedge\ x_* + \text{sum-list}(\text{ret-list}(lmy)) = \text{sum-list}(\text{ret-list}(lmz))\\ &\qquad \wedge\ 1 + \text{size-list}(\text{cur-list}(lmy)) = \text{size-list}(\text{cur-list}(lmz))\end{aligned}$$

$$\begin{aligned}&\text{sort-list}(lmx, lmy)\\ &\quad :\iff \text{sum-list}(\text{cur-list}(lmx)) = \text{sum-list}(\text{cur-list}(lmy))\\ &\qquad \wedge\ \text{sum-list}(\text{ret-list}(lmx)) = \text{sum-list}(\text{ret-list}(lmy))\\ &\qquad \wedge\ \text{size-list}(\text{cur-list}(lmx)) = \text{size-list}(\text{cur-list}(lmy))\end{aligned}$$

$$\begin{aligned}&\text{carve-list}(n, lmx)\\ &\quad :\iff \text{sum-list}(\text{cur-list}(lmx)) - \text{sum-list}(\text{ret-list}(lmx))\\ &\qquad\quad = \triangle\big(n, \text{size-list}(\text{cur-list}(lmx))\big)\end{aligned}$$

$$\text{sort-carve-list}(lx, ly) :\iff \text{sum-list}(lx) - \text{sum-list}(ly) = \triangle(\text{size-list}(lx))$$

[3]If we should replace size-list(cur-list(lmx)) with size-list(lmx), we would need an inductive discussion to prove that the former value equals the latter value.

Chapter 5

Related Work

CHC-based Verification in the Presence of Pointers and Destructive Updates

As introduced in Section 1.1.4, JayHorn [27, 28] and SeaHorn [54, 14] are CHC-based verification tools for programs with pointers and destructive updates. Judging from the results of Section 4.1, it seems that our method basically outperforms these existing verification tools for programs with ownership types, although design and implementation of a verification framework using our method is still a topic of future research.

Non-CHC-based Verification Techniques Regarding Ownership

Although CHC-based verification exploiting ownership types itself has not been studied well, there are a number of non-CHC-based verification techniques regarding ownership.

Suenaga and Kobayashi [43] propose a type system for a programming language with memory allocation/deallocation primitives like the C language; pointer types are augmented with fractional ownerships on the resources, which prevents errors such as double-frees and memory leaks. The type system admits a polynomial-time type inference algorithm, which eventually serves as an automated verification algorithm on memory-related errors.

CRUST [63] is a bounded model checker for detecting memory safety errors including violations of Rust's ownership discipline in a library using `unsafe` code blocks. It works fully automatically, requiring no additional manual annotations; test drivers are also automatically generated. It checks that calling a bounded number of library methods do not trigger memory safety errors. The approach successfully re-discovered some bugs from the developing versions of vector (`Vec<T>`) and slice (`[T]`) libraries. However, CRUST does not check safety of calling methods from multiple libraries in general.

Rust2Viper [17] verifies properties of annotated Rust programs using the Viper verification infrastructure [60]; thus it provides a semi-automated verification tool on Rust. Rust programs with special annotations (on purity, invariants, preconditions, postconditions, etc.) are translated into code of Silver, an intermediate language of Viper that supports reasoning about permissions. Ownership information of Rust is translated into permission information of Silver, which helps to reduce the amount of manual annotations drastically.

Electrolysis [65] translates Rust code (more precisely, Mid-level Intermediate Representation (MIR) of Rust code) into the purely functional language in the Lean theorem prover [49], and exert verification with the system of Lean. Although Electrolysis' reduction into a purely functional language partly contains a similar idea to our translation, the reduction imposes strong restrictions on the usage of mutable references; it tries to manage the situation with encoding mutable references with lenses [12] and executing static tracking of the reference relation, which prevents Electrolysis from supporting flexible operations on mutable references, including returning arbitrary mutable references from a function, nesting storing of mutable references, and mutable borrowing in loops [66]. Our translation can manage those types of operations on mutable references in a simple way.

Baranowski, He and Rakamarić [3] provide an automated verifier for Rust by extending the SMACK verifier [57], which was first designed for LLVM IR programs generated by the Clang C compiler. The tool supports a large class of features in Rust, but exerts only bounded verification on Rust programs in the paper; after complicated processes, bounded program verification problems are finally translated into logic formulae (without fixed-point operations), which are checked with existing SMT solvers.

RustBelt [25] provides the first formal (and machine-checked) safety proof for a formal language λ_{Rust} representing a realistic subset of Rust, including basic libraries `mem::swap`, `Rc`, `Cell` and `RefCell`, as well as concurrent libraries `thread::spawn`, `rayon::join`, `Arc`, `Mutex` and `RwLock`; the proof can be extended to check new Rust libraries that uses `unsafe` code blocks. The semantics of λ_{Rust} is described into a higher-order concurrent separation logic Iris [26], which is formalized on the Coq proof assistant [47]. To sum up, under the framework of RustBelt, semi-automated verification of memory safety of Rust libraries with `unsafe` code blocks can be performed with a logic Iris built on the Coq proof assistant.

Formalization of Rust

This paper proposes a formal language Calculus of Ownership and Reference (COR) in Chapter 3. The language represents a substantial subset of Rust, but there are also other formalizations of Rust.

Patina [39] is a formal language that models the safe subset of Rust; it aims to check that the novel ownership checking schemes of Rust successfully guarantees memory safety. Honestly following the structure of Rust, Patina employs rather complicated concepts such as lvalues and rvalues. Patina is also still not completely built up in the paper [39]; the expression and statement layers of Patina do not support operations on product types and sum types.

RustBelt introduces a formal language λ_{Rust} [25] representing a realistic subset of Rust; in fact, the language consists of two systems: the safe part, which is a target of substructural typing, and the unsafe part, which allows many more features and is not given any type system at all.[1] The operational semantics of the safe part is defined by embedding into the unsafe part; functions written in the unsafe part are exposed to the safe part with appropriately typed interfaces. This language design makes it possible to deal with a wide range of unsafe libraries. Basically COR resembles the safe part of λ_{Rust}, but is more simplified compared to the part; some differences of COR from λ_{Rust} are illustrated in footnotes in Chapter 3.

[1]In the RustBelt paper, mostly only the safe part is illustrated; the unsafe part is described in the technical appendix [24].

Chapter 6

Conclusion

We have presented a novel *value-based* method of CHC-based verification for programs with pointers and destructive updates controlled by ownership types. In our translation of programs into CHCs, each mutable reference is represented as a pair of the current value and the future value that is passed at the end of mutual borrowing, in contrast to the conventional *address-based* method which represents a pointer simply as an address. We have informally demonstrated that our translation is applicable for various programs of *Rust*, a programming language with ownership types. In addition, we have formalized our translation on a new formal language that represents a basic subset of Rust, and described a conjecture on the correctness of the translation; giving a solid proof on the correctness is a topic of future research. Finally, we have shown the advantage of our method on program verification by an experiment along with discussion on useful invariants.

Our method exploits both the features of CHCs and those of ownership types. CHC representation does not make an explicit distinction between inputs and outputs of functions; thus, in a sense, the logical description can be *free from the flow of calculation*. Ownership types ensure the *locality of effects of destructive updates*. It is very likely that our method can be combined with various existing CHC-based verification methods, and our method may also be fused with some non-CHC-based verification methods by providing the underlying representation with liberty from the computation flow.

Although we have formalized our method for our simplified formal language, design and implementation of a *new verification system for real-world Rust programs using our method* is an interesting topic of future research. Rust is aggressively used by more and more programmers these days, and thus the creation of a practical, efficient verification tool for Rust will surely contribute so much to real-world developers. Furthermore, with sophisticated pointer analysis, our method can possibly be applied to verification on C/C++ programs by cleverly tracking switching of ownership.

Bibliography

[1] Amal Ahmed, Matthew Fluet, and Greg Morrisett. L^3: A linear language with locations. *Fundamenta Informaticae*, pages 397–449, 2007.

[2] Thibaut Balabonski, François Pottier, and Jonathan Protzenko. The design and formalization of Mezzo, a permission-based programming language. *ACM Transactions on Programming Languages and Systems*, 2016.

[3] Marek Baranowski, Shaobo He, and Zvonimir Rakamarić. Verifying Rust programs with SMACK. In *Automated Technology for Verification and Analysis*, pages 528–535, 2018.

[4] Nikolaj Bjørner, Kenneth L. McMillan, and Andrey Rybalchenko. Program verification as satisfiability modulo theories. In *SMT@IJCAR*, pages 3–11, 2012.

[5] Nikolaj Bjørner, Arie Gurfinkel, Ken McMillan, and Andrey Rybalchenko. *Horn Clause Solvers for Program Verification*, pages 24–51. 2015.

[6] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. What's decidable about arrays? In *Verification, Model Checking, and Abstract Interpretation*, pages 427–442, 2006.

[7] Toby Cathcart Burn, C.-H. Luke Ong, and Steven J. Ramsay. Higher-order constrained Horn clauses for verification. In *Principles of Programming Languages*, pages 11:1–11:28, 2018.

[8] David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. In *Object-oriented Programming, Systems, Languages, and Applications*, pages 48–64, 1998.

[9] Olivier Danvy and Lasse R. Nielsen. Defunctionalization at work. In *Principles and Practice of Declarative Programming*, pages 162–174, 2001.

[10] Robert DeLine and Manuel Fähndrich. Enforcing high-level protocols in low-level software. In *Programming Language Design and Implementation*, pages 59–69, 2001.

[11] Robert W. Floyd. Assigning meanings to programs. *Mathematical Aspects of Computer Science, Proceedings of Symposia in Applied Mathematics*, 19:19–32, 1967.

[12] J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. Combinators for bi-directional tree transformations: A linguistic approach to the view update problem. In *Principles of Programming Languages*, pages 233–246, 2005.

[13] Carlo A. Furia. What's decidable about sequences? In *Automated Technology for Verification and Analysis*, pages 128–142, 2010.

[14] Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A. Navas. The SeaHorn verification framework. In *Computer Aided Verification*, pages 447–450, 2015.

[15] Arie Gurfinkel and Jorge A. Navas. A context-sensitive memory model for verification of C/C++ programs. In Francesco Ranzato, editor, *Static Analysis*, pages 148–168, 2017.

[16] Peter Habermehl, Radu Iosif, and Tomáš Vojnar. What else is decidable about integer arrays? In *Theory and Practice of Software*, pages 474–489, 2008.

[17] Florian Hahn. Rust2Viper: Building a static verifier for Rust. Master's thesis, ETH Zürich, 2016.

[18] Michael Hind. Pointer analysis: Haven't we solved this problem yet? In *Program Analysis for Software Tools and Engineering*, pages 54–61, 2001.

[19] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–580, 1969.

[20] Kryštof Hoder and Nikolaj Bjørner. Generalized property directed reachability. In *Theory and Applications of Satisfiability Testing*, pages 157–171, 2012.

[21] Kryštof Hoder, Nikolaj Bjørner, and Leonardo de Moura. μZ – an efficient engine for fixed points with constraints. In *Computer Aided Verification*, pages 457–462, 2011.

[22] ISO Working Group 21. The C++ standards committee. http://www.open-std.org/jtc1/sc22/wg21/, 2018.

[23] Trevor Jim, J. Greg Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of C. In *USENIX Annual Technical Conference*, pages 275–288, 2002.

[24] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. RustBelt: Securing the foundations of the Rust programming language – technical appendix and Coq development. https://plv.mpi-sws.org/rustbelt/popl18/, 2017.

[25] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. RustBelt: Securing the foundations of the Rust programming language. *Principles of Programming Languages*, 2018.

[26] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming*, 28, 2018.

[27] Temesghen Kahsai, Rody Kersten, Philipp Rümmer, Huascar Sanchez, and Martin Schäf. JayHorn: A framework for verifying Java programs. http://jayhorn.github.io/jayhorn/, 2019.

[28] Temesghen Kahsai, Philipp Rümmer, Huascar Sanchez, and Martin Schäf. JayHorn: A framework for verifying Java programs. In *Computer Aided Verification*, pages 352–358, 2016.

[29] Kind Software. https://kindsoftware.com/products/opensource/ESCJava2/, 2015.

[30] Teodor Knapik, Damian Niwinski, and Pawel Urzyczyn. Higher-order push-down trees are easy. In *Foundations of Software Science and Computation Structures*, pages 205–222, 2002.

[31] Naoki Kobayashi, Étienne Lozes, and Florian Bruse. On the relationship between higher-order recursion schemes and higher-order fixpoint logic. In *Principles of Programming Languages*, pages 246–259, 2017.

[32] Naoki Kobayashi, Takeshi Tsukada, and Keiichi Watanabe. Higher-order program verification via HFL model checking. In *Programming Languages and Systems*, pages 711–738, 2018.

[33] Anvesh Komuravelli, Arie Gurfinkel, and Sagar Chaki. SMT-based model checking for recursive programs. In *Computer Aided Verification*, pages 17–34, 2014.

[34] Anvesh Komuravelli, Arie Gurfinkel, Sagar Chaki, and Edmund M. Clarke. Automated abstraction in SMT-based unbounded software model checking. In *Computer Aided Verification*, pages 846–862, 2013.

[35] Chris Lattner and Vikram Adve. Automatic pool allocation: Improving performance by controlling data structure layout in the heap. In *Programming Language Design and Implementation*, pages 129–142, 2005.

[36] Karl Mazurak, Jianzhou Zhao, and Steve Zdancewic. Lightweight linear types in System F°. In *Types in Language Design and Implementation*, pages 77–88, 2010.

[37] Chris Okasaki. *Purely Functional Data Structures*. PhD thesis, Carnegie Mellon University, 1996.

[38] C.-H. Luke Ong. On model-checking trees generated by higher-order recursion schemes. In *Logic in Computer Science*, pages 81–90, 2006.

[39] Eric W. Reed. Patina : A formalization of the Rust programming language. Master's thesis, University of Washington, 2015.

[40] John Charles Reynolds. Definitional interpreters for higher-order programming languages. *Higher-Order and Symbolic Computation*, 11(4):363–397, 1998.

[41] Philipp Ruemmer. The Eldarica model checker. https://github.com/uuverifiers/eldarica, 2019.

[42] Sable Research Group. What is Soot? | soot. https://sable.github.io/soot/, 2019.

[43] Kohei Suenaga and Naoki Kobayashi. Fractional ownerships for safe memory deallocation. In *Asian Symposium on Programming Languages and Systems*, pages 128–143, 2009.

[44] The Agda Team. The Agda Wiki - Agda. https://wiki.portal.chalmers.se/agda/pmwiki.php, 2018.

[45] The Boogie Team. Boogie. https://github.com/boogie-org/boogie, 2019.

[46] The Cheker Framework Team. The Checker Framework. https://checkerframework.org/, 2018.

[47] The Coq Team. The Coq proof assistant. https://coq.inria.fr/, 2019.

[48] The Isabelle Team. Isabelle. https://isabelle.in.tum.de/, 2018.

[49] The Lean Team. Lean theorem prover. https://leanprover.github.io/, 2019.

[50] The Redox Team. Redox – your next(gen) OS. https://www.redox-os.org/, 2019.

[51] The Rust Team. Rust by example. https://doc.rust-lang.org/rust-by-example/, 2019.

[52] The Rust Team. Rust programming language. https://www.rust-lang.org, 2019.

[53] The Rust Team. The Rust programming language. https://doc.rust-lang.org/book/, 2019.

[54] The SeaHorn Team. SeaHorn | a verification framework. http://seahorn.github.io/, 2019.

[55] The SeaHorn Team. seahorn/crab: CoRnucopia of ABstractions: a language-agnostic library for abstract interpretation. https://github.com/seahorn/crab, 2019.

[56] The Servo Team. Servo, the parallel browser engine. https://servo.org/, 2019.

[57] The Smack Team. SMACK software verifier and verification toolchain. https://github.com/smackers/smack, 2018.

[58] The Spacer Team. Spacer. https://spacer.bitbucket.io, 2019.

[59] The Spec# Team. Spec# - Microsoft Research. https://www.microsoft.com/en-us/research/project/spec/, 2019.

[60] The Viper Team. http://www.pm.inf.ethz.ch/research/viper.html, 2019.

[61] The Why3 Team. Why3. http://why3.lri.fr/, 2018.

[62] The Z3 Team. The Z3 theorem prover. https://github.com/Z3Prover/z3, 2019.

[63] John Toman, Stuart Pernsteiner, and Emina Torlak. CRUST: A bounded verifier for Rust. In *Automated Software Engineering*, pages 75–80, 2015.

[64] Jesse A. Tov and Riccardo Pucella. Practical affine types. In *Principles of Programming Languages*, pages 447–458, 2011.

[65] Sebastian Ullrich. Simple verification of Rust programs via functional purification. Master's thesis, Karlsruhe Institute of Technology, 2016.

[66] Sebastian Ullrich. Electrolysis reference. http://kha.github.io/electrolysis/, 2019.

[67] Mahesh Viswanathan and Ramesh Viswanathan. A higher order modal fixed point logic. In *CONCUR 2004 - Concurrency Theory*, pages 512–528, 2004.