Extensible Functional-Correctness Verification of Rust Programs by the Technique of Prophecy

預言の技術による Rust プログラムの拡張可能な機能正当性検証

by

Yusuke Matsushita
松下 祐介

A Master's Thesis
修士論文

Submitted to
the Graduate School of the University of Tokyo
on February 24, 2021
in Partial Fulfillment of the Requirements
for the Degree of Master of Information Science and Technology
in Computer Science

Thesis Supervisor: Naoki Kobayashi　小林 直樹
Professor of Computer Science

ABSTRACT

Rust is a systems programming language that gives strong static guarantees on properties like memory and thread safety and has been widely used in recent years for high reliability and performance. In Rust, code is usually under a static check of resource usage under the ownership principle based on lifetimes. Also, we can robustly extend the expressivity of Rust by writing libraries with implementation free from the static check and interfaces following the ownership principle. Therefore, verifying safety and functional correctness of Rust programs in an extensible and scalable way is a key to developing reliable high-performance software. There is some existing work on verifying Rust programs. Jung et al. verified safety of Rust's type system and some Rust libraries in an extensible way using the higher-order separation logic Iris in the Coq Proof Assistant. Matsushita et al. proposed a method to automatically verify functional correctness of programs in a basic subset of Rust. However, it was still unclear how to verify functional correctness of Rust programs in an extensible and scalable way. In this thesis, we propose a novel extensible logical foundation to specify and verify functional correctness (and safety) of Rust programs, by extending the technique of prophecy used by Matsushita et al. We semantically model types and verification conditions in Rust in Iris, combining the work of Jung et al. with the technique of prophecy, to flexibly support new libraries and features of Rust. As a basis for that, we present a new formulation of prophecy that allows flexible operation. We plan to mechanize the results in Coq in the near future.

論文要旨

Rust はメモリ安全性やスレッド安全性などについて強い静的保証を与えるシステムプログラミング言語であり、高い信頼性と性能を求めて近年幅広く使われている。Rust では、コードは通常ライフタイムに基づく所有権原理のもとでリソースの使用を静的に検査されている。また、静的検査を受けない実装と所有権原理に従うインターフェースを持つライブラリを書くことで、Rust の表現力を堅牢に拡張することもできる。ゆえに、Rust プログラムの安全性および機能正当性を拡張可能かつスケーラブルな形で検証することが信頼できる高性能ソフトウェアを開発するための鍵となる。Rust プログラムの検証についてはいくつか既存研究がある。Jung らは Rust の型システムおよびいくつかの Rust ライブラリの安全性を定理証明支援系 Coq 上で高階分離論理 Iris を用いて拡張可能な形で検証した。松下らは Rust の基本的なサブセットのプログラムの機能正当性を自動検証する手法を提案した。しかしながら、Rust プログラムの機能正当性を検証するための拡張可能でスケーラブルな手法は知られていなかった。この論文では、松下らの用いた預言の技術を拡張し、Rust プログラムの機能正当性 (と安全性) を記述および検証するための、新しい拡張可能な論理的基盤を提案する。Jung らの成果と預言の技術を組み合わせて、Iris 上で Rust の型や検証条件を意味論的にモデル化し、Rust の新しいライブラリや機能を柔軟に取り入れられるようにする。このための基礎として、柔軟な操作を許す新しい預言の概念を提案する。近い将来に成果を Coq で機械化する予定である。

Acknowledgments

I express my sincere appreciation to my supervisor, Prof. Naoki Kobayashi. Over years he fostered my skills as a researcher, especially in presentation and communication, with great passion and patience. On this research he gave me great advice, too. I would also like to thank Prof. Takeshi Tsukada for his lively and warm advice on this research. Moreover, they both worked with me on RustHorn, an essential predecessor of this work.

This work emerged during my research internship at the RustBelt team of Max Planck Institute for Software Systems from September to December 2020. I would like to show my gratitude to people who gave me warm assistance during the internship. Prof. Derek Dreyer supervised me with patience to cultivate my research skills and guide the research direction; he encouraged me with a fabulous sense of humor and generously allowed me to use the name RustHornBelt for this research project. Jacques-Henri Jourdan gave me valuable comments in discussion and conceived some important ideas for this work; he also mechanized my new idea about consuming logical steps in Iris just a few days after I proposed it, which excited me a lot. Ralf Jung always encouraged me and gave me insightful advice; it was always my pleasure to interact with him. Xavier Denis collaborated closely with me on this work, with great passion for verifying Rust programs; he gave me advice on this thesis and discussed with me the future direction of this work. Rodolphe Lepigre took care of me continuously. Although the global COVID-19 pandemic prevented me from flying to Europe to meet them, I had amazing collaboration online.

Finally, I would like to thank my mother and father for supporting me from various aspects in my life. Thanks to their warm supports, over years, I have been able to enjoy academic activities.

Contents

Chapter 1

Introduction

The *Rust Programming Language* (Matsakis and Klock, 2014; Rust Community, 2021a) is a systems programming language that aims at a high level of performance, reliability and productivity. Like C/C++, Rust allows users to use low-level operations for high performance. At the same time, unlike common mainstream languages like C/C++ and Java, Rust gives a high-level safety guarantees by a strong static check on resource usage by its distinctive *ownership principle* based on *lifetimes*. Rust also allows unsafe code, i.e., code that does not get the static resource-usage check. We can robustly extend the expressivity of Rust by writing libraries with implementation with unsafe code and interfaces that follow the ownership principle. Rust has been widely used in recent years in industry and has attracted various academic interests. Therefore, *verification of safety and functional correctness of Rust programs* is significant for development of reliable high-performance software. In verification of Rust programs, *scalability and extensibility* is important, because Rust is widely used for large-scale software and the expressivity of Rust can be flexibly extended by libraries with unsafe code.

There are some existing studies on verification of Rust programs. *RustBelt* (Jung et al., 2018a) verified memory and thread *safety* of Rust's type system and some Rust libraries mechanically in the Coq Proof Assistant. Their proof is highly *extensible* thanks to a semantic approach based on the higher-order separation logic *Iris* (Jung et al., 2015, 2018c). *RustHorn* (Matsushita et al., 2020a) proposed a method to *automatically* verify *functional correctness* of Rust programs within a basic subset of features. Their verification method leverages the guarantees of Rust's ownership principle to give a clean logic model to a Rust program, which allows a fairly scalable verification of Rust programs. The logic model precisely tracks the effect of updates performed through *unique references* by the technique of *prophecy*, while it successfully omits explicit representation of the the heap memory and addresses. However, it was still unclear how to verify functional correctness of Rust programs in an extensible and scalable way.

In this thesis, we propose a novel extensible logical foundation to specify and verify functional correctness including safety, of Rust programs, using a prophecy-based clean logic model in the style of RustHorn and taking a semantic approach in the style of RustBelt. We name this research project *RustHornBelt*. As a basis of that, we present a new framework of prophecy on top of Iris; in this framework, prophecy lives only in the ghost state and allows flexible operations. We plan to mechanize our results in Coq in the near future, taking advantage of the strong Coq support of Iris.

In § 1.1, we introduce the Rust programming language, especially in terms of the type system. In §1.2, we introduce RustBelt. In §1.3, we introduce RustHorn and explain its idea, giving some detailed examples of verification in RustHorn. In §1.4, we give an overview of our work RustHornBelt.

1.1 The Rust Programming Language

The *Rust Programming Language* (Matsakis and Klock, 2014; Klabnik et al., 2018; Rust Community, 2021a) is a systems programming language aiming at a high level of performance, reliability and productivity.

As a fairly new language (Rust 1.0 was released in 2015), Rust has been designed on the basis of the lessons learned from existing programming languages and past development experiences. Like C/C++, Rust allows users to use *low-level operations* to achieve high memory and space efficiency. At the same time, Rust provides *high-level safety guarantees* by a strong static check on *resource usage* by the *ownership* principle based on *lifetimes*, which is introduced later in this section. Also, productivity of Rust is enhanced by various modern features, including pattern matching and lambda expressions, and also mature development tools.

Rust has been developed by a large, energetic community and has rapidly gained popularity. Rust is already used by software products such as Firefox (Mozilla, 2021), Dropbox (Dropbox, 2020) and npm (npm, 2019). Also, Rust is officially sponsored by Microsoft, Mozilla, Amazon and Google (Rust Community, 2021b).

Rust has attracted various academic interests. There have been various studies on verification of Rust programs (Toman et al., 2015; Hahn, 2016; Ullrich, 2016; Jung et al., 2018a; Lindner et al., 2018; Baranowski et al., 2018; Astrauskas et al., 2019; Dang et al., 2020; Matsushita et al., 2020a). Some studies worked on aliasing disciplines for Rust (Jung et al., 2020a), feature proposal on Rust (Fallin, 2020), and formalization of Rust (Reed, 2015; Weiss et al., 2019). Some studies investigated real-world Rust programs (Qin et al., 2020; Astrauskas et al., 2020). Some studies applied Rust to systems programming (Levy et al., 2015; Anderson et al., 2016; Balasubramanian et al., 2017; Levy et al., 2017; Lamowski et al., 2017; Ding et al., 2017; Almohri and Evans, 2018; Emmerich et al., 2019), concurrency (Jespersen et al., 2015), and cryptography (Mindermann et al., 2018).

In the remaining part of this section, we introduce and explain Rust's type system, especially in terms of the ownership principle based on lifetimes.

Possible Dangers of Pointer Use To understand the motivation of Rust's type system, let us first see how dangerous pointer manipulation can be without static checking on resource usage. In most mainstream programming languages such as C/C++ and Java, each pointer can read, modify and release its target resource without much restriction. This freedom can easily cause, however, unexpected errors.

A programmer can unintentionally use a pointer to a resource that has been implicitly released.

Example 1.1 (Dangerous Update After Implicit Release). For example, let us consider the following C++ code.

```
1 vector<int>* pv;
2 {  vector<int> v { 0, 1, 2 };  pv = &v;  }  // v is released
3 /* ... some operations ... */  pv->push_back(3); // dangerous!
```

Here, `vector<int>` is a type for an integer vector, i.e., an array of integers that can change in size. In the line 1, we declare an uninitialized pointer `pv` to an integer vector. In the line 2, a local scope is introduced by curly brackets and within it a new vector `v` is created with initial elements `0, 1, 2` and the pointer `pv` is set to `&v`. When we leave the local scope, the vector `v` is automatically released. After some operations, in the line 3, we try to append an element to the target vector of `pv`. Because the target of `pv` has actually been *invalidated*, this memory access is highly *dangerous*. If another

object is using the memory cells at the address pv, this memory access violates that object.

An internal memory block of an object can be released by update on the object, which may cause a tricky memory error.

Example 1.2 (Dangerous Update After Reallocation). Let us consider the following example.

```
1 vector<int> v { 0, 1 };  int* p = &v[0];
2 v.push_back(2);  // reallocation occurs
3 /* ... some operations ... */  *p = 10; // dangerous!
```

In the line 1, we create a vector of two elements 0 and 1 and store it to the variable v, and then we take a pointer p to the first element of the vector. In the line 2, we push a new element 2 to (the end of) the vector through v. Actually, at this point, the memory block for the elements of the vector gets *reallocated*.[1] After some operations, in the line 3, we try to update the target of the pointer p. However, because the memory cell at the address p is no longer managed by v, it may be now used by another object, so this memory access is highly *dangerous* and can cause *unexpected behavior*.

To sum up, when a resource is shared by multiple pointers, the programmer can easily cause unexpected behavior by updating the resource from some pointer. In practice, such memory safety errors occur relatively frequently. For example, Miller (2019) reported that about 70% of vulnerabilities in Microsoft in terms of CVE patches were related to memory safety in the years 2006-2018.

Rust's Ownership Principle Unlike languages like C/C++ and Java, Rust's type system guarantees *memory safety* by performing a strong static check on *resource usage* under the *ownership principle* based on *lifetimes*. This kind of type system that is aware of resource usage is called *substructural* (in a broad sense). Although substructural type systems with ownership have been actively studied for a long time (Wadler, 1990; Clarke et al., 1998; DeLine and Fähndrich, 2001; Jim et al., 2002; Fähndrich and DeLine, 2002; Fluet et al., 2006; Mazurak et al., 2010; Haller and Odersky, 2010; Zibin et al., 2010; Tov and Pucella, 2011; Ghica and Smith, 2014; Morris, 2016; Bernardy et al., 2018), the type system of Rust has some notable features. Now, before getting into Rust code examples, we give a high-level overview of Rust's ownership principle.

In Rust, an object can be pointed at by multiple pointers, but at each program point only one pointer can have *ownership* (also called *unique permission*) on the object, which permits writing to and reading from the object. While some pointer has ownership on an object, any other pointers do not have ownership on it. Ownership is structural; that is, ownership on an object includes ownership on any sub-objects of the object.

A key feature of Rust is *borrowing*. When a pointer a has ownership to some object, we can create a new pointer ua to the object and temporarily move the ownership on the object from a to ua during some period. The period for borrowing is called a *lifetime*. The newly created pointer ua is called a *unique reference*. Also, precisely, this type of borrowing is called a *unique borrow*. The lifetime is statically and globally managed by Rust's type system. While the lifetime is ongoing, the unique reference ua has the ownership on the object and the original owner a temporarily loses the ownership. At the moment when the lifetime ends, ua loses the ownership and a retrieves the ownership on the object. The key idea is that the unique reference ua and the original owner a *don't need to directly communicate with each other when the ownership is*

[1] To be precise, whether or not reallocation occurs depends on the capacity of the vector, but in common implementations the capacity of the vector is set exactly to n when we initialize a vector by an initializer list of the length n.

returned, because the static control based on the lifetime ensures that the ownership is never shared by multiple pointers. So we do not have to strictly keep `ua` until the lifetime ends. While the lifetime is ongoing, the borrowed ownership on the object can freely be passed around, subdivided, and thrown away. This is a key feature of Rust that realizes *modular* and *scalable* management of the ownership.

Rust also has the notion of *sharing permission*, which can be shared by multiple pointers but permits only reading from the object, not writing to it. Sharing permission is also structural like ownership. While a pointer has sharing permission on an object, any other pointer cannot have ownership on the object. Rust has a *shared borrow*, which is similar to a unique borrow but takes out sharing permission, not ownership (i.e., unique permission). When a pointer `a` has ownership on some object, we can create a new pointer `sa` called a *shared reference* and temporarily give it the sharing permission on the object under some lifetime. Because sharing permission is sharable, a shared reference can be *copied*, whereas a unique reference can't be copied. The original owner temporarily loses the ownership while the lifetime is ongoing and retrieves the ownership at the moment the lifetime ends. While the lifetime is ongoing, the borrowed sharing permission can be freely passed around, subdivided, and thrown away, just like a unique borrow.

Notably, a lifetime in Rust is roughly speaking a *set of program points* instead of just a lexical scope, which is important for flexibility of borrowing. This notion of a lifetime is called a *non-lexical lifetime*. The compiler of Rust *automatically infers* non-lexical lifetimes and checks the ownership principle with some elaborate algorithm (Rust Community, 2020). This machinery is dubbed a *borrow checker*. Also, a new algorithm for the borrow checker is under development (Matsakis, 2018, 2020; Rust Community, 2021c).

The ownership principle of Rust can be too restrictive for the purpose of guaranteeing safety. Still, the Rust compiler exploits the guarantees of the ownership principle for *optimization*, which is studied in depth by Jung et al. (2020a). Also, Rust allows users to write *unsafe code*, which is code that does not get the static resource-usage check based on the ownership principle; we can robustly extend the expressivity of Rust by appropriately encapsulating unsafe code. We explain unsafe code more in depth later in this section.

Examples of Unique Borrows Let us see how we can use unique borrows. You can try the Rust compiler online at the Rust Playground `https://play.rust-lang.org/`.

For a very simple example, we can use a unique borrow as follows.[2]

```
1 let mut n: i32 = 3;
2 let p: &mut i32 = &mut n;
3 *p = *p + 4;
4 print!("{}", n);  // 7
```

The type `i32` represents a (32-bit) integer. In the line 1, we get an integer variable `n` with the initial value `3`. The keyword `mut` in `let mut` allows us to modify the data of `n`. The type `&mut T` represents a *unique reference* (also called a mutable reference) to an object typed `T`, which have temporarily acquired the ownership on its target object. In the line 2, by `&mut n`, we perform a *unique borrow* (also called a mutable borrow) to take a unique reference `p` to the integer data of `n`.[3] Then in the line 3, we increment

[2] If we want to make it a complete Rust program, we just need to put the code inside the entry point function `fn main() { ... }`.

[3] In Rust, the keyword `mut`, standing for 'mutable', is used for two different contexts. The use of `mut`

the integer data by 4 through the unique reference p. In the line 4, ending the unique borrow, we access the integer data through n and know that the value is now set to 7. In Rust, each reference and borrow is associated with some lifetime but the lifetime is *inferred by the compiler* and the programmer cannot explicitly specify it (except when we handle polymorphism over lifetimes, which is explained later).

Now let us see how Rust's type system prevents the two dangerous pointer usages unregulated in C++ which were discussed above.

Example 1.3 (Update After Release Prevented by Rust). The first C++ code snippet of Example 1.1 can be translated into the following Rust code.

```
1 let pv: &mut Vec<i32>;
2 {  let mut v: Vec<i32> = vec![0, 1, 2];  pv = &mut v;  }
3 /* ... some operations ... */  pv.push(3);  // dangerous!
```

The type `Vec<T>` represents a vector of the element type `T`, which corresponds to C++'s `vector<T>`. In the line 2, we first construct a vector by `vec![0, 1, 2]` and let the variable `v` own it. Later in the line 2, we uniquely borrow the vector `&mut v` to get a unique reference, which is named `pv`. The Rust compiler correctly judges this Rust code dangerous, emitting an error message like below.

```
error: `v` does not live long enough
* |   {  ...  pv = &mut v;  }  ...
  |               ^^^^^^   - `v` dropped here while still borrowed
  |               borrowed value does not live long enough
* |   ...  pv.push(3);
  |        -- borrow later used here
```

While the lifetime is ongoing, the alias `v` temporarily loses the ownership. Because `pv` is used in the line 3 by the push `pv.push(3)`, Rust judges that the lifetime is alive until the end of the line 3. When the variable `v` gets out of the lexical scope of curly braces, Rust tries to release (or *drop*) the vector through `v` but notices that this violates the ownership principle with regard to the lifetime. This is the reason why the Rust compiler emit the error message like shown above. Note that we remove the curly brackets of the line 2 as follows, the code passes the type check.

```
1 let pv: &mut Vec<i32>;
2 let mut v: Vec<i32> = vec![0, 1, 2];  pv = &mut v;
3 pv.push(3);
```

Example 1.4 (Update After Reallocation Prevented by Rust). The second C++ code snippet of Example 1.2 is translated into the following Rust code.

```
1 let mut v: Vec<i32> = vec![0, 1];  let p: &mut i32 = &mut v[0];
2 v.push(2);  // reallocation occurs
3 /* ... some operations ... */  *p = 10;  // dangerous!
```

Here, the operation `&mut v[0]` is syntax sugar of `Vec::index_mut(&mut v, 0)`[4] and `v.push(2)` is syntax sugar of `Vec::push(&mut v, 2)`. Again the Rust compiler correctly judges this program dangerous and emits an error message like below.

```
error: cannot borrow `v` as mutable more than once at a time
* |   ...  let p: &mut i32 = &mut v[0];
  |                           - first mutable borrow occurs here
* |   v.push(2);  ...
  |   ^ second mutable borrow occurs here
* |   ...  *p = 10;  ...
  |        ------- first borrow later used here
```

in `let mut` allows the object directly owned by the variable to be modified. In the meanwhile, the use of `mut` in `&mut` indicates a unique borrow/reference rather than a shared borrow/reference.

[4] We need a declaration such as `use std::ops::IndexMut;` to use the name `index_mut`.

In the line 1, the operation `&mut v[0]`, which stands for `Vec::index_mut(&mut v, 0)`, uniquely borrows `v` to take a unique reference and then *subdivide* it into a unique reference to the head element of the vector by calling the function `Vec::index_mut`. The resulting unique reference, typed `&mut i32`, is passed to the variable `p`. Since we use `p` in the line 3 by `*p = 10`, the lifetime of this unique borrow should be continued until the line 3. However, in the line 2, the operation `v.push(2)`, which stands for `Vec::push(&mut v, 2)`, uniquely borrows `v` and performs the push operation by calling the function `Vec::push`. To perform this second unique borrow, we need the ownership of `v` here, but it should be temporarily lost until the lifetime of the first borrow ends. This is why the compiler emits the error message like above. If we swap the order of the updates like shown below, the Rust code becomes safe and passes the static check of Rust.

```
1 let mut v: Vec<i32> = vec![0, 1];  let p: &mut i32 = &mut v[0];
2 *p = 10;  // the lifetime of p ends here
3 v.push(2);  // reallocation occurs but it's ok
```

Here, the Rust compiler infers that the lifetime of `p` or the unique borrow of `&mut v[0]` ends at the line 2. Note that the Rust code passes the type check even if we change `&mut v[0]` into `&mut v[10]`; the method `index_mut` performs dynamic bounds checking and we simply get a dynamic error by this change.

In Rust, we can also create a new reference by borrowing from an existing unique reference. This operation is called a *reborrow*.[5]

Example 1.5 (Reborrowing). For example, let us consider the following Rust code.

```
1 let mut v = vec![0];  let pv = &mut v;
2 pv.push(1);  pv.push(2);
```

It passes Rust's type check and works correctly. It seems fairly natural but actually is made possible by reborrowing. The push operation `pv.push(1)` actually calls the function `Vec::push`, which *consumes* the input unique reference. In order to perform the second push `pv.push(2)` after the first push `pv.push(1)`, we cannot consume `pv` by the first push. Therefore, Rust implicitly performs *reborrowing*. For each push on `pv`, Rust takes a new unique reference `&mut *pv` that lives only during the function call of `Vec::push`. We can rewrite the Rust code above into the following code with explicit reborrowing.[6]

```
1 let mut v = vec![0];  let pv = &mut v;
2 Vec::push(&mut *pv, 1);  Vec::push(&mut *pv, 2);
```

Examples of Shared Borrows Let us see how we can use shared borrows.

We can copy shared references and use multiple shared references at the same time. For a simple example, the following Rust code is valid.

```
1 let n: i32 = 7;
2 let p: &i32 = &n;  let q: &i32 = p;
3 print!("{} and ", *p);  print!("{}", *q);  // 7 and 7
```

[5] We do not need to think about reborrowing from a shared reference because shared references can be copied.

[6] We can also write `Vec::push(pv, 1);` instead of `Vec::push(&mut *pv, 1);` but still Rust implicitly performs reborrowing of `pv`.

The type `&T` (*without* the keyword `mut`) represents a *shared reference* (also called an immutable reference) to an object typed `T`, which have temporarily acquired *sharing permission* on the target object. In the line 2, we first perform a *shared borrow* (also called an immutable borrow) `&n` to get a shared reference `p` to the integer data of `n`; then we *copy* the address of `p` to get a shared reference `q`. In the line 3, we read the integer value from `p` and then from `q`. This makes the lifetimes of `p` and `q` *overlap* but it is still *valid* because sharing permission can be shared. At a low level (without optimization), unique and shared borrows/references are the same operation/object, but for simplicity of the type system Rust expects that programmers *explicitly annotate* whether the borrow/reference is unique or shared by the notations `&mut` and `&`.

Rust forbids release of an object while we are using a shared reference to the object.

Example 1.6 (Read After Release Prevented by Rust). For example, the following Rust code is correctly rejected by the Rust compiler.

```
1 let pv: &Vec<i32>;
2 {  let v: Vec<i32> = vec![0, 1, 2];  pv = &v;  }
3 /* ... */  print!("{}", pv[0]);  // dangerous!
```

The type `Vec<i32>` represents a vector of shared references to integer data. In the line 2, we construct the vector `v` and then take a shared reference `pv` to the vector `v`. When we leave from the inner scope, the vector `v` is automatically released. In the line 3, by `pv[0]`, we try to access the head element of the vector through `pv`, where `pv[0]` is syntax sugar for `*Vec::index(&pv, 0)`.[7] However, it is dangerous because it can cause invalid memory access, including *dereference of a null pointer*. When we perform `pv[0]`, we first read the address data of the vector and then dereference it; however, since the vector has already been released, the memory cell that used to store the address now may have an invalid address, possibly the null value. This Rust code does violate Rust's ownership principle and the compiler emits an error message like the following.

```
error: `v` does not live long enough
* |   {  ...  pv = &v;  }  ...
  |               ^^   - `v` dropped here while still borrowed
  |               borrowed value does not live long enough
* |   ...  print!("{}", pv[0]);  ...
  |                     -- borrow later used here
```

Rust also forbids update of an object while we are using a shared reference to the object.

Example 1.7 (Read After Reallocation Prevented by Rust). For example, the following Rust code is correctly rejected by the Rust compiler.

```
1 let mut v: Vec<&i32> = vec![&0, &1];  let p: &&i32 = &v[0];
2 v.push(&2);  // reallocation occurs
3 /* ... */  print!("{}", **p);  // dangerous!
```

In the line 1, we first crate a vector `v` of shared references to integer data, of the type `Vec<&i32>` and the initial value `vec![&0, &1]`, and then take a shared reference `p` to the head element of the vector, which itself is a shared reference to integer data. Here, by performing `&0`, we statically allocate integer data `0` and take an address of it. In the line 2, we push a new element `&2` to the vector `v`, which causes reallocation. In the line 3, after some operations, we try to access the inner integer data of `**p`. However it is dangerous, because the data at `p` is no longer managed by the vector and thus `*p` can be an invalid address, possibly the null value. This code violates Rust's ownership principle and the compiler emits an error message like the following.

[7] We need declaration like `use std::ops::Index;` to use the method `index`, just like `index_mut`.

```
error: cannot borrow `v` as mutable because it is also borrowed as
↪  immutable
* |   ...  let p: &&i32 = &v[0];
  |                        - immutable borrow occurs here
* |   v.push(&2);  ...
  |   ^^^^^^^^^^ mutable borrow occurs here
* |   ...  print!("{}", **p);  ...
  |                     --- immutable borrow later used here
```

In Rust, a unique reference under a shared reference can only have the sharing permission instead of the ownership (unique permission).

Example 1.8 (A Unique Reference Under a Shared Reference). For example, the following Rust code is correctly rejected by the Rust compiler.

```
1 let mut v: Vec<&i32> = vec![&0, &1];
2 let pv: &mut Vec<&i32> = &mut v;
3 let ppv: & &mut Vec<&i32> = &pv;  let q: &&i32 = &ppv[0];
4 ppv.push(2);  // reallocation occurs
5 /* ... */ print!("{}", **q);  // dangerous!
```

In the line 1, we create a vector v of shared references to integer data. Then in the line 2, we take a unique reference pv to the vector. In the line 3, we take a shared reference ppv to the unique reference pv and then take a shared reference qv to the head element of the vector using ppv. In the line 4, we perform the push operation `ppv.push(2)` through ppv, which causes reallocation. In the line 5, after some operation, we try to read the inner integer data of q. However, it is dangerous because the address at `*q` may be now invalid due to the reallocation. This code violates Rust's ownership principle and the compiler emits an error message like the following.

```
error: cannot borrow `**ppv` as mutable, as it is behind a `&`
↪  reference
* |   let ppv: ... = &pv;  ...
  |                   --- help: consider changing this to be a mutable
  ↪  reference: `&mut pv`
* |   ppv.push(2);  ...
  |   ^^^ `ppv` is a `&` reference, so the data it refers to cannot be
  ↪  borrowed as mutable
```

Polymorphism over Lifetimes In Rust, a function can be *polymorphic over lifetimes*, which is an important feature for modularity of borrowing.

For example, the method `Vec::push` used above has the following function signature.[8]

```
1 fn Vec::push<'a, T>(pv: &'a mut Vec<T>, a: T) -> ()
```

The keyword `fn` stands for 'function'. The method `Vec::push` inputs a unique reference to a vector `pv: &'a mut Vec<T>` and a value `a: T` and appends the value to the vector, possibly performing reallocation. The output type is the unit type `()`. The function is polymorphic over the *lifetime* `'a` of the reference pv, as well as the element type T.

For another example, the method `Vec::index_mut`, which is used for index access on a vector `&mut pv[i]`, has the following signature.

```
1 fn Vec::index_mut<'a,T>(pv: &'a mut Vec<T>, i:usize) -> &'a mut T
```

[8] Although Rust uses a special notation for methods, we use here a notation for usual functions.

This method outputs the unique reference to the `i`-th element of the vector of the type `&'a mut T` by *subdividing* the input unique reference to a vector `pv: &'a mut Vec<T>`. The type `usize` represents an unsigned 64-bit integer (if we are using the 64-bit architecture). We subdivide the input reference instead of performing a new borrow. Therefore, the lifetime of the output reference `&'a mut T` has the same lifetime `'a` as the input reference `&'a mut Vec<T>`.

Variant Types Rust supports *variant types*, unlike common imperative languages like C/C++ and Java.

For example, the following type `Result<T, E>` is commonly used.

```
enum Result<T, E> {  Ok(T), Err(E)  }
```

This type is parametrized over the main result type `T` and the error type `E`. A value of the type `Result<T, E>` is either a successful value `Ok(v)` with a body value `v: T` or an error value `Err(e)` with an error value `e: E`.

We can also make recursive types using variant types. For example, the singly linked list type can be defined as follows.

```
enum List<T> {  Nil, Cons(T, Box<List<T>>)  }
```

Here, the *box pointer type* `Box<U>` represents a pointer to an object typed `T` with the ownership on the object and the right to free (or deallocate) the memory block for the object. A value of the type `List<T>` is either the nil `Nil` or the cons value `Cons(v, l)` of the head value `v: T` and the boxed tail list `l: Box<List<T>>`. Using recursion, we can write various functions over the list type. In particular, we can write interesting functions by combining lists with unique references.

For example, we can write the following function `inc_all`.[9]

```
fn inc_all<'a>(pla: &'a mut List<i32>) -> () {
  match pla {
    Nil =>  {},
    Cons(pa, pla2) =>  {  *pa += 1;  inc_all(pla2);  }
  }
}
```

It inputs a unique reference to a list `pla` and destructively increments all elements of the list. The pattern `Cons(pa, pla2)` *splits* the unique reference `pla: &'a mut List<i32>` into a unique reference to the head `pa: &'a mut i32` and a unique reference to the tail `pla2: &'a mut List<i32>`.

For another example, we can write the following function `split_unq`.

```
fn split_unq<'a>(pla: &'a mut List<i32>) -> List<&'a mut i32> {
  match pla {
    Nil =>  Nil,
    Cons(px, plx2) =>  Cons(px, Box::new(split_unq(plx2)))
  }
}
```

It inputs a unique reference to an integer list `pla: &'a mut List<i32>` and outputs a list of unique references to each element of the integer list.

[9] We need a declaration like `use List::*;` to use the constructors `Nil` and `Cons` (without the qualifier `List::`).

Unsafe Code In Rust, we can also write *unsafe code* (Klabnik et al., 2018, §19.1), which is code that does not take the static check on resource usage. If we want to make a piece of code unsafe, we can just put it into the unsafe code block `unsafe { ... }`. In unsafe code, we can use the *mutable raw pointer type* `*mut T` and the *immutable raw pointer type* `*const T`, which represent a pointer to an object typed `T` with or without the ability to update its target object. In unsafe code, we can actively use any number of mutable and immutable raw pointers to one object *without restriction*. Also, in unsafe code, we can freely perform conversion between (unique or shared) references and raw pointers.

We can robustly extend the expressivity of Rust using libraries that have *internal implementation with unsafe code* and *interfaces that follow the ownership principle*. For example, the internal implementation of the vector type `Vec<T>` depends on unsafe code but we can use the vector type with *safety guarantees* as long as we access vectors through the safe interfaces like `Vec::push` and `Vec::index_mut`. Although Rust's type system is quite restrictive by the strict enforcement of the ownership principle, we can robustly extend the expressivity of Rust in this way.

Interior Mutability Using unsafe code, we can write a type whose internal data structure can be modified through *shared references*. Such a property on a type is called *interior mutability* (Klabnik et al., 2018, §15.5) (Jung, 2020, §8.6). Rust provides a bunch of robust libraries that offer interior mutability.

For a simple example, Rust has the cell type `Cell<T>`. It can be used as follows.

```
let a: &Cell<i32> = & Cell::new(3);  let b: &Cell<i32> = a;
a.set(7);  print!("{}", b.get());  // 7
```

Here, we make a new integer cell with the initial value `3` by `Cell::new(3)`. We take a shared reference `a` to the cell, which can be *copied* to `b`. By `a.set(7)`, which is syntax sugar for `Cell::set(a, 7)`, we can set the internal value of the cell to `7`. By `b.get()`, which is syntax sugar for `Cell::get(b)`, we can get the internal value of the cell, which is now set to `7`. The methods `Cell::set` and `Cell::get` have the following function signatures.

```
fn Cell::set<'a, T>(c: &'a Cell<T>, d: T) -> ()
fn Cell::get<'a, T>(c: &'a Cell<T>) -> T
```

Any shared reference to a cell has the right to call `set` and `get`. Actually, the call of `Cell::get` requires that the type `T` is *copyable*, i.e., an object typed `T` can be copied. Types like the integer type `i32` and the shared reference type `&T` are copyable but types like the vector type `Vec<T>` and the mutable reference type `&mut T` are not copyable. Rust manages a property like copyability on a type using the machinery called a *trait* (Klabnik et al., 2018, §10.2), which is comparable to Java's interface and Haskell's type class.

For an advanced example, Rust also supports the mutex type `Mutex<T>`. A mutex can be shared by multiple threads and be modified by exclusively taking the lock.

1.2 RustBelt — A Semantic Model of Rust Types

As explained in §1.1, Rust guarantees memory and thread safety using a static resource-usage check by its distinctive lifetime-based ownership principle. In fact, many common errors in imperative programming (e.g., Example 1.1 and Example 1.2) are prevented by Rust's type system. Also, the expressivity of Rust can be robustly extended using libraries have internal implementation with unsafe code and interfaces that follow the ownership principle.

The Rust community has been making a lot of efforts to enhance and guarantee the safety of Rust's type system and libraries through tests, bug reports and human thoughts. Still, safety of Rust is highly tricky. In particular, when we have *multiple* libraries with unsafe internal implementation, it is challenging to guarantee that *any combination of method calls* on these libraries does not cause unexpected behaviors. In fact, Rust has suffered from a number of bugs caused by interaction of multiple libraries (Ben-Yehuda, 2015; Borowski, 2019).

Therefore, it is significant to give *formal safety guarantees* to Rust's type system and libraries in an *extensible* way, being open to new libraries and features of Rust. This need gave rise to the work *RustBelt* (Jung et al., 2018a,b). They gave the first *mechanized* proof of *memory and thread safety* of a core of Rust's type system and some key standard libraries of Rust, including ones with interior mutability such as `Cell`, `RefCell`, `Mutex` and `RwLock`. They found a tricky bug of a real-world Rust library (Jung, 2017) in the process of the formal verification. Their proof is mechanized in the Coq Proof Assistant (Coq Community, 2021), making full use of the higher-order separation logic *Iris* (Jung et al., 2015, 2018c), which is provided as a library in Coq (Krebbers et al., 2017, 2018). Iris is a general-purpose separation logic that attains outstanding extensibility by *impredicativity* or higher-order ghost states (Iris is explained more in depth in Chapter 2).

The core of their proof is construction of *semantic models of Rust types*, which consist of *predicates in Iris*. Each syntactic typing rule is semantically interpreted in Iris and gets a separate soundness proof. Also, each library method with unsafe implementation and typed interfaces is likewise semantically interpreted in Iris and separately verified. In this way, the safety proof of RustBelt is highly *extensible*. This semantic approach stands in contrast to common *syntactic* type soundness proof based on progress and preservation (Wright and Felleisen, 1994); although the syntactic approach seems handy, it is in general non-trivial how we can modify the syntactic soundness proof when we add new execution rules and typing rules.

As an essential basis for their semantic approach, they designed the *lifetime logic* on top of Iris in order to flexibly discuss borrowing of resources based on lifetimes. In the lifetime logic, we can directly borrow Iris propositions instead of some fixed set of resources, which is made possible by the impredicativity of Iris.

1.3 RustHorn — Verifying Rust Programs by Prophecy

How can we verify *functional correctness* of Rust programs leveraging the guarantees given by Rust's ownership principle? Let us first focus on a basic subset of Rust without unsafe code. *Without unique references*, we can model Rust programs just as functional programs. But what about *with unique references*? When we perform a unique borrow and create a unique reference in Rust, in the verification model we can pass the current value of the target object to the unique reference. When we update the target object of the unique reference, in the model we can just accordingly update the current target value of the unique reference. The problem is, after the lifetime of the borrow ends, how we can *tell* the original owner the *new value* of the object, reflecting updates that have been performed by the unique reference.

RustHorn (Matsushita et al., 2020a) proposed a very simple solution to this question: we use a *prophecy variable* on the value at the end of the lifetime of the unique borrow.[10] Roughly speaking a *prophecy variable* is a variable that peeks out some value that will

[10] The idea of using prophecy for verifying Rust programs was first proposed by the senior thesis of the author (Matsushita, 2019). The author completed the RustHorn paper (Matsushita et al., 2020a) in collaboration with Takeshi Tsukada and Naoki Kobayashi, the supervisor of this thesis.

be observed in the *future*. More specifically, the verification model handles a unique borrow as follows. First we have some object `a: T`, which is modeled as a value v. When we create a unique reference `ua: &mut T` by uniquely borrowing `a`, we create a *prophecy variable* x, which prophesy the value of the borrowed object at the end of the lifetime of the unique borrow. We model the unique reference `ua` as (v, x), a pair of the current value v of the target object and the prophecy variable x. Also, until the lifetime of the borrow ends, we model `a` as the prophecy variable x. When we update the target object of `ua`, in the model we update the current target value v of the pair (v, x) into the new target value v', obtaining the model (v', x). And importantly, at the point we throw away (or release) the unique reference `ua` modeled (v, x), we *resolve* the prophecy variable x into the target value v (i.e., assign v to x), obtaining the equality information $x = v$ as a postcondition. This method also supports advanced usage of unique references like *reborrowing* and *subdivision* (see Example 1.10). After the lifetime of the borrow ends, the original owner `a` retrieves the ownership, now being able to read and update its object. We know that `a` has the value x, because x has been resolved to the value v. In this way, we can translate any Rust program (within a basic subset of Rust) into a logic model *without* explicit representation of the heap memory and addresses, just like programs of functional programming languages. In a sense, this prophecy-based technique gives us a general way to view a basic subset of Rust as a functional programming language.

RustHorn (Matsushita et al., 2020a) demonstrated this prophecy-based method particularly in the context of automated verification based on *constrained Horn clauses (CHCs)* (Grebenshchikov et al., 2012; Bjørner et al., 2015). Roughly speaking, a CHC representation of a program is a set of logical formulas of some format called CHCs, which describe constraints on *predicate variables* that represent the input-output relation of some function or continuation. The program verification problem is reduced to the satisfiability problem on CHCs, which can be automatically solved by existing CHC solvers (Komuravelli et al., 2014; Fedyukovich et al., 2017; Hojjat and Rümmer, 2018; Champion et al., 2018) in a highly *scalable* way. RustHorn (Matsushita et al., 2020a) implemented an automated verifier that translates Rust programs into CHCs using the idea of prophecy for unique borrows and then calls the backend CHC solvers Spacer (Komuravelli et al., 2014) and HoIce (Champion et al., 2018). They also performed experiments on benchmarks using the verifier, including comparison with SeaHorn (Gurfinkel et al., 2015), an existing CHC-based verification platform for the C Programming Language. RustHorn succeeded in automated verification of fairly complicated Rust programs.

In principle, the prophecy-based translation of RustHorn can also be applied to many Rust libraries including vectors `Vec`. Still it cannot handle libraries with *interior mutability* in a clean way in general, because mutable states are actually *shared* under interior mutability, unlike ownership by unique references.

Below we show two examples of verification in RustHorn with detailed explanation.

Example 1.9 (Dynamic Decision of the Address of a Unique Reference). For example, let us consider the following Rust code.

```
fn take_max<'a>(ua: &'a mut int, ub: &'a mut int)-> &'a mut int {
  if *ua >= *ub { ua } else { ub }
}
fn test(mut a: int, mut b: int) {
  let uc: &mut int = take_max(&mut a, &mut b);  *uc += 1;
  assert!(a != b);
}
```

For simplicity of verification, we assume here that we have an unbounded integer type

`int`. The function `take_max` takes two unique references to integer `ua` and `ub` and returns the one with the greater target value. This function is quite simple, but the point is that the *address* of the unique reference returned by it is determined by a *dynamic* condition `*ua >= *ub`. The function `test` performs a simple test of `take_max`. It inputs two integer objects `a` and `b`. It uniquely borrows the two and passes them to `take_max` to take a unique reference `uc`. Then it increments the target value of `uc`. Finally it checks that the values of `a` and `b` are different. We want to verify that the assertion of `test` always succeeds, regardless of the inputs. That property holds because the increment on `uc` makes the difference between `a` and `b` exactly one bigger. However, it is tricky because we need to ensure that `uc` points to the greater of `a` and `b`, not only either `a` or `b`.

RustHorn translates the Rust program (or verification problem) above into the following two CHCs.

$$\begin{aligned} &\mathit{take\_max}((a, a_\circ), (b, b_\circ), r) \iff \\ &\quad \big(a \geq b \wedge b_\circ = b \wedge r = (a, a_\circ)\big) \vee \big(a < b \wedge a_\circ = a \wedge r = (b, b_\circ)\big) \\ &a_\circ \neq b_\circ \iff \mathit{take\_max}((a, a_\circ), (b, b_\circ), (c, c_\circ)) \wedge c_\circ = c + 1 \end{aligned}$$

Each CHC is universally quantified over the free variables. The predicate logic for CHCs is implicitly multi-sorted and each variable has some sort.

The first CHC characterizes the function `take_max` through the predicate variable *take_max*, which represents the input-output relation of `take_max`. The unique reference `ua` is modeled as $(a, a_\circ)$, the current target integer a and the *prophecy* target integer $a_\circ$. We do similarly for `ub`. Suppose $a \geq b$ holds. We *release* the unique reference $(b, b_\circ)$ and thus *resolve* the prophecy variable $b_\circ$ to b, obtaining the equality $b_\circ = b$. We also set the return value r to $(a, a_\circ)$. When $a < b$ holds, we do a similar thing swapping and $(a, a_\circ)$ and $(b, b_\circ)$.

The second CHC describes the function `test`. The inputs `a` and `b` correspond to the variables a and b. When we perform a unique borrow `&mut a`, we take a new *prophecy variable* $a_\circ$, which represents the value of `a` just after the end of the borrow, and model the created unique reference as $(a, a_\circ)$ and `a` now as $a_\circ$. We do similarly for `&mut b`. The call of `take_max` is modeled as the use of the predicate variable *take_max*. The return value `uc` is modeled again as $(c, c_\circ)$. We increment the target integer of `uc`, which updates the model of `uc` into $(c + 1, c_\circ)$. Then we *release* `uc` and thus *resolve* the prophecy variable $c_\circ$ into $c + 1$, obtaining the equality $c_\circ = c + 1$. Finally, the assertion `assert!(a != b)` is modeled as the condition $a_\circ \neq b_\circ$. The variables $a_\circ$ and $b_\circ$ perfectly reflect the values of `a` and `b` at this point.

The system of two CHCs can be satisfied by the following solution.

$$\mathit{take\_max}((a, a_\circ), (b, b_\circ), (c, c_\circ)) \quad := \quad c_\circ = c + 1 \Rightarrow a_\circ \neq b_\circ$$

In the experiment of RustHorn (Matsushita et al., 2020a, §4), this problem was automatically and instantly solved.

We can also represent the Rust program as the following functional program (we use OCaml here).

```
let take_max (a, a') (b, b') = if a >= b
    then ( assume (b' = b); (a, a') )
    else ( assume (a' = a); (b, b') )
let test a b =
    let a' = Random.int(0) in  let b' = Random.int(0) in
    let (c, c') = take_max (a, a') (b, b') in
```

```
    assume (c' = c + 1);
    assert (a <> b)
```

Here, the function `assume` is defined as follows.

```
let assume b = if b then () else assume b
```

This function falls into an infinite loop when `b` is false. Since we are solving the *safety problem*, where non-termination is considered success, `assume b` simply introduces the postcondition that `b` is true.

Combining the technique of prophecy with recursive types, we can perform interesting verification.

Example 1.10 (Update of a List via a Unique Reference). Let us consider the following recursive function on the list type `List<T>` (defined in §1.1).

```
fn take_some<'a>(ula: &'a mut List<int>) -> &'a mut int {
  match ula {
    Nil =>  take_some(ula),
    Cons(ua, ula2) =>  if rand() { ua } else { take_some(ula2) }
  }
}
```

The function `take_some` inputs a unique reference to an integer list `ula` and outputs a unique reference to some non-deterministically chosen element of the list. Here, we assume some non-deterministic function `fn rand() -> bool`. When the target list of `ula` is cons, we split `ula` into unique references on the head `ua: &'a mut int` and the tail `ula2: &'a mut List<int>` and either return `ua` or recursively call `take_some` on `ula2`. When the target list is nil, we fall into an infinite loop. We can write the following test function on `take_some`.

```
fn test(mut la: List<int>) {
  let n = sum(&la);  let ua = take_some(&mut la);
  *ua += 1;  assert!(sum(&la) == n + 1);
}
```

The function `test` first saves the initial sum of the list `la`, then uniquely borrows `la` and passes it to `take_some` to get a unique reference `ua`, increments the target of `ua`, and finally asserts that the sum of the list has increased by one. Here, the function `sum` is defined as follows.

```
fn sum<'a>(sla: &'a List<int>) -> int {
  match sla { Nil => 0, Cons(sa, sla2) => *sa + sum(sla2) }
}
```

RustHorn translates this Rust program into the following CHCs.

$$take\_some((\mathsf{nil}, la_\circ), r) \iff take\_some((\mathsf{nil}, la_\circ), r)$$

$$take\_some((a :: la', la_\circ), r) \iff la_\circ = a_\circ :: la'_\circ \ \wedge \ \big(\big(la'_\circ = la' \ \wedge \ r = (a, a_\circ)\big) \vee \big(a'_\circ = a' \ \wedge \ take\_some((la', la'_\circ), r)\big)\big)$$

$$sum(\mathsf{nil}, r) \iff r = 0$$

$$sum(a :: la, r) \iff sum(la, r') \ \wedge \ r = a + r'$$

$$n' = n + 1 \iff sum(la, n) \ \wedge \ take\_some((la, la_\circ), (a, a_\circ)) \ \wedge \ a_\circ = a + 1 \ \wedge \ sum(la_\circ, n')$$

The first and second CHCs characterize the function `take_some`, respectively for the cases where the target list of `ula` is nil or cons. The first CHC is a tautology. The second CHC is interesting. Because we destruct `ula` into `Cons(ua, ula2)`, we accordingly *subdivide* the prophecy variable $la_\circ$ into two prophecy variables $a_\circ, la_\circ$, constraining $la_\circ$ to $a_\circ :: la'_\circ$. Now `ua` is modeled as $(a, a_\circ)$ and `ula2` is modeled as $(la, la_\circ)$. In general, when we subdivide a unique reference, we also accordingly subdivide its prophecy variable, which is an essential technique in this prophecy-based approach. Now what we do is simple. Depending on the non-deterministic boolean `rand()`, we either (i) release `ula2`, obtaining $la'_\circ = la'$, and return `ua` or (ii) release `ua`, obtaining $a'_\circ = a'$, and recursively call `take_some(ula2)`. The third and fourth CHCs characterize the function `sum`. A shared reference is represented simply as the value of its target object. The fifth CHC characterizes the test function. This system of CHCs has the following simple solution.

$$\mathit{take\_some}((la, la_\circ), (a, a_\circ)) \quad := \quad a_\circ = a + 1 \implies \mathsf{sum}\ la_\circ = \mathsf{sum}\ la + 1$$

$$\mathit{sum}(la, r) \quad := \quad r = \mathsf{sum}\ la$$

Here, the function $\mathsf{sum} : \mathsf{List}\,\mathbb{Z} \to \mathbb{Z}$ is defined inductively as follows.

$$\mathsf{sum}\ \mathsf{nil} \ := \ 0 \qquad\qquad \mathsf{sum}\,(a :: la) \ := \ a + \mathsf{sum}\ la$$

In the experiment (Matsushita et al., 2020a, §4), the verifier of RustHorn succeeded in *automated verification* of this Rust program. This is because HoIce (Champion et al., 2018), one of the CHC backend solvers, can find and handle an inductive function like sum from the CHCs of *sum*. Once we have sum, the solution to *take_some* presented above can be easily found from the last CHC. The validity of this solution can be easily checked using the equality $\mathsf{sum}\,(a :: la) = a + \mathsf{sum}\ la$.

As these examples show, using the idea of prophecy, we can give a clean logic model to a fairly wide class of Rust programs, which makes verification highly *scalable*. Although the paper of RustHorn focused on CHC-based automated verification, by the technique of prophecy, we can also verify Rust programs using existing semi-automated verification platforms for functional programming languages such as F* (Swamy et al., 2016) and Why3 (Filliâtre and Paskevich, 2013).

The correctness of this prophecy-based method is non-trivial. The RustHorn paper (Matsushita et al., 2020a,b) presented the proof of *soundness and completeness* of the reduction from Rust programs to CHCs. It was achieved mainly by constructing a bisimulation between the execution of Rust programs and some deduction algorithm on CHCs. On the side of CHCs, prophecy variables of unique references are represented simply as syntactic variables, which can be specialized later in the deduction algorithm.

1.4 Our Work, RustHornBelt — Unifying RustHorn and RustBelt

As introduced in § 1.2, *RustBelt* (Jung et al., 2018a) verified memory and thread *safety* of the core type and some basic libraries of Rust in an *extensible* way. Still, RustBelt did not verified functional correctness. In particular, RustBelt models unique reference as a kind of invariant saying that *some* object of the type is stored at the target address and does not track precise information about the object. As introduced in § 1.3, *RustHorn* (Matsushita et al., 2020a) presented a new method of verifying *functional correctness* of a wide class of Rust programs in a fairly scalable way leveraging the guarantees of Rust's ownership principle. The point is that for each unique borrow we can take a *prophecy variable* on the value of the target object just after the end of the borrow and model the unique reference as a pair of the current target value and the prophecy variable

to track the final target value when the unique reference is released. Although this prophecy-based modeling is expected to be applicable to a wide class of Rust libraries including `Vec` and `HashMap`, we did not have a logical foundation for proving that in an *extensible* way. Although Matsushita et al. (2020a,b) presented a proof of soundness and completeness of their prophecy-based reduction, their proof depends heavily on syntactic structures and has rather little extensibility (this aspect is discussed more in depth in § 3.1 and § 6.5). It was unclear how we can unify the approaches of RustHorn and RustBelt.

In this thesis, we propose a novel *extensible* logical foundation to specify and verify functional correctness of Rust programs, using a prophecy-based logic model in the style of RustHorn and taking a semantic approach in the style of RustBelt. We name this research project *RustHornBelt*, as it combines the approaches of RustHorn and RustBelt. We build a new formalization of prophecy in Iris (Chapter 3), where the information about prophecy lives only in the ghost state and we can perform an operation that we call dependent resolution. We also develop a low-level foundation for verifying imperative programs in Iris (Chapter 4), with a new technique for spending an unbounded number of logical steps at one physical step. We give new semantic models or *semantic types* to program types in Rust (Chapter 5), extending the semantic approach of RustBelt with refinement of a type by a pure value that is parametrized over assignments on prophecy variables, which allows us to use the *prophecy*-based logic model in the style of RustHorn. Finally, in order to formalize and verify type-based translation of Rust programs into logic models, we introduce and semantically model a type system for Rust that manipulates a logic model, which we dub the *refined type system* (Chapter 6). Thanks to the semantic approach, we can verify each deduction rule of the type system *separately*, which brings about *extensibility*, allowing us to flexibly add new features and libraries of Rust.

The remaining part of this thesis is structured as follows. In Chapter 2, we introduce the higher-order separation logic Iris as preliminaries for the following chapters. In Chapter 3, we present a new platform for prophecy. In Chapter 4, we present a low-level foundation for verification of imperative programming. In Chapter 5, we define our notion of the semantic type and model various Rust types. In Chapter 6, we introduce and semantically model our refined type system for Rust, proving some deduction rules in detail and giving some examples of verification. In Chapter 7, we present the conclusion of the paper and discuss future work.

This work emerged during my research internship at the RustBelt team of Max Planck Institute for Software Systems from September to December 2020. Nevertheless, virtually all the core ideas of this thesis were conceived by the author unless otherwise noted, although I received valuable advice about the direction of this research. It is worth noting here that Jacques-Henri Jourdan conceived the idea of using cumulative time receipts for swelling persistent time receipts (§ 4.2; CumuTime-Swell-PersTime), which was an essential technique for completing the formalization of this thesis.

Chapter 2

Preliminaries on the Higher-Order Separation Logic Iris

A *separation logic* is a logic for scalable and modular reasoning about the resources, which was introduced by O'Hearn et al. (2001) and Reynolds (2002). Various separation logics have been designed so far. The separation logic *Iris* (Jung et al., 2015, 2018c) is a general-purpose separation logic that attains outstanding extensibility by *impredicativity* (or higher-order ghost states), which is made possible by *step indexing* (Appel and McAllester, 2001; Birkedal et al., 2011). Also, Iris is *mechanized* in the Coq Proof Assistant with sophisticated tactics and proof modes (Krebbers et al., 2017, 2018).

Iris is applied to verification of various contexts; to name a few, C11's weak memory model (Kaiser et al., 2017), refinement on fine-grained concurrency (Frumin et al., 2018), the ST monad (Timany et al., 2018), persistency of the Intel-x86 Architecture (Raad et al., 2020), low-level sandboxing (Sammler et al., 2020a), session types (Hinrichsen et al., 2020), distributed databases (Gondelman et al., 2021), effect handlers (de Vilhena and Pottier, 2021), and gradual typing (Jacobs et al., 2021). And notably, Iris is a vital foundation for RustBelt (Jung et al., 2018a). Our work RustHornBelt, which is based on the technique of RustBelt, is also built on top of the separation logic Iris.

In this chapter, we briefly describe how we can *use* the higher-order separation logic Iris at a fairly high level. In § 2.1, we introduce basic connectives and deduction rules of Iris. In § 2.2, we introduce the notion of the *resource algebra* and explain how we can customize Iris using resource algebras. In § 2.3, we describe how we can the *lifetime logic*, which was built on top of Iris by Jung et al. (2018a) to discuss borrowing of resources under lifetimes in Rust. This chapter serves as technical preliminaries for the following chapters Chapter 3, Chapter 4, Chapter 5 and Chapter 6.

We still do not introduce any program logic in this chapter. The weakest precondition (and the Hoare triple) for imperative programs can be *defined* using the existing connectives of Iris. Because our definition of the weakest precondition is a part of our contribution, it is described later in § 4.2.

2.1 Basic Connectives and Deduction Rules of Iris

In this section, we introduce and describe basic logical connectives and deduction rules of Iris.

Before introducing the connectives, we explicitly write down here the precedence on the connectives: (i, the strongest) $\Box, \triangleright, \diamond, \Rightarrow\!\!\!\mid_{\mathcal{E}}$ (and similar connectives); (ii) $*, \wedge, \vee$; (iii) $-\!\!*, \Rightarrow, \mathrel{=\!\!\!*}_{\mathcal{E}}, \Rrightarrow_{\mathcal{E}}$ (and similar connectives); (iv, the weakest) $\forall\, x.\,, \exists\, x.\,$.

Iris Proposition An *Iris proposition* $P, Q, R : \mathsf{IProp}$ is roughly speaking a *predicate over resources*. More precisely, it is a *step-indexed, monotone* predicate over *valid* resources. Also, a resource for an Iris proposition is called a *ghost state*. The formal

model of Iris propositions is well described in (Jung et al., 2018c, §4). In this thesis, we often call an Iris proposition just a proposition.

We call a proposition of metalogic a *pure proposition* and represent it by variables ϕ, ψ, to distinguish it from an Iris proposition. We write Prop for the type of pure propositions.

Sequent Iris has a *sequent* $P \vdash Q$, which is a (metalogic) binary relation on Iris propositions $P, Q : \mathsf{IProp}$. It satisfies reflexivity, transitivity and anti-symmetry.

$$\textsc{Seq-Refl}\quad P \vdash P \qquad \textsc{Seq-Trans}\quad \frac{P \vdash Q \qquad Q \vdash R}{P \vdash R} \qquad \textsc{Seq-AntiSym}\quad \frac{P \vdash Q \qquad Q \vdash P}{P = Q}$$

Basic Connectives Iris has the basic connectives of predicate logic: the truth True, falsehood False, conjunction $P \wedge Q$, universal quantification $\forall\, x\colon T.\, P_x$, disjunction $P \vee Q$, existential quantification $\exists\, x\colon T.\, P_x$, and implication $P \Rightarrow Q$. These connectives are defined based on Kripke semantics of intuitionistic logic and thus satisfy *all tautologies of intuitionistic logic*. In particular, the following properties hold.

$$\textsc{True-Intro}\quad P \vdash \mathsf{True} \qquad \textsc{False-Elim}\quad \mathsf{False} \vdash P \qquad \textsc{And-Univ}\quad \frac{R \vdash P \qquad R \vdash Q}{R \;\vdash\; P \wedge Q} \qquad \textsc{All-Univ}\quad \frac{\forall\, x\colon T.\, \big(Q \vdash P_x\big)}{Q \;\vdash\; (\forall\, x\colon T.\, P_x)}$$

$$\textsc{Or-Univ}\quad \frac{P \vdash R \qquad Q \vdash R}{P \vee Q \;\vdash\; R} \qquad \textsc{Exist-Univ}\quad \frac{\forall\, x\colon T.\, \big(P_x \vdash Q\big)}{(\exists\, x\colon T.\, P_x) \;\vdash\; Q} \qquad \textsc{Imp-Univ}\quad \frac{P \wedge Q \;\vdash\; R}{P \;\vdash\; Q \Rightarrow R}$$

Note that we can 'throw away' any proposition P by the rule True-Intro. A separation logic that satisfies this rule is called *affine*.

We can also promote a pure proposition ϕ: Prop into an Iris proposition. This promotion preserves basic connectives — truth, falsehood, conjunction, universal quantification, disjunction, existential quantification and implication.

We also introduce the following shorthand.

$$P \Leftrightarrow Q \quad :=\quad (P \Rightarrow Q) \;\wedge\; (Q \Rightarrow P)$$

$$\exists\, x \text{ s.t. } \phi_x.\; P \quad :=\quad \exists\, x.\; \phi_x \wedge P \qquad\qquad \forall\, x \text{ s.t. } \phi_x.\; P \quad :=\quad \forall\, x.\; \phi_x \Rightarrow P$$

We say that an Iris proposition P is a tautology if it is equal to True.

Separating Conjunction and Magic Wand The *separating conjunction* $P * Q$ of propositions P and Q represents a resource that can be written as a composition of some resource satisfying P and some resource satisfying Q.

The separating conjunction is associative, commutative and unital (on True).

$$\textsc{Sep-Assoc}\quad (P * Q) * R \;=\; P * (Q * R) \qquad \textsc{Sep-Comm}\quad P * Q \;=\; Q * P \qquad \textsc{Sep-Unit}\quad P * \mathsf{True} \;=\; P$$

Also, we can compose sequents in parallel according to separating conjunction, which is a very important property.

$$\textsc{Sep-Parallel}\quad \frac{P \vdash Q \qquad P' \vdash Q'}{P * P' \;\vdash\; Q * Q'}$$

We can derive the following rules.

Sep-Lose
$P * Q \vdash P$

Sep-And
$P * Q \vdash P \wedge Q$

Sep-All
$P * \forall x.\, Q_x \vdash \forall x.\, P * Q_x$

Sep-Frame
$$\frac{P \vdash Q}{P * R \vdash Q * R}$$

The separating conjunction has the right adjoint, which is written as $P \mathrel{-\!\!*} Q$ and called the *magic wand*.

Wand-Univ
$$\frac{P * Q \vdash R}{\dfrac{}{P \vdash Q \mathrel{-\!\!*} R}}$$

A magic wand $P \mathrel{-\!\!*} Q$ from P to Q can be understood as a resource such that if we compose it with any resource satisfying P then we get a resource satisfying Q. Given that the implication is the right adjoint of the (plain) conjunction, roughly speaking, the magic wand is to the separating conjunction what the implication is to the conjunction.

We can derive the following rules.

Imp-Wand
$P \Rightarrow Q \vdash P \mathrel{-\!\!*} Q$

Wand-Apply
$P * (P \mathrel{-\!\!*} Q) \vdash Q$

Wand-Curry
$P * Q \mathrel{-\!\!*} R = P \mathrel{-\!\!*} (Q \mathrel{-\!\!*} R)$

Wand-Id
$\mathsf{True} \vdash P \mathrel{-\!\!*} P$

Wand-Comp
$(P \mathrel{-\!\!*} Q) * (Q \mathrel{-\!\!*} R) \vdash P \mathrel{-\!\!*} R$

Wand-Parallel
$(P \mathrel{-\!\!*} Q) * (P' \mathrel{-\!\!*} Q') \vdash P * P' \mathrel{-\!\!*} Q * Q'$

Since the separating conjunction is a left adjoint of the magic wand, by the 'left adjoints preserve colimits' law, we have the following.

Sep-Exist
$P * \exists x.\, Q_x = \exists x.\, P * Q_x$

Persistence Modality Iris has the persistence modality $\Box P$. Intuitively, $\Box P$ means that P holds in a 'lightweight' way.

The persistence modality satisfies the following property.

Pers-And-Sep
$Q \wedge \Box P = Q * \Box P$

The proposition $Q \wedge \Box P$ can be understood as a resource that satisfies both Q and $\Box P$. By this rule, we can *separate out* a resource that satisfies $\Box P$ from that resource (note that $Q \wedge \Box P$ is equivalent to $(Q \wedge \Box P) \wedge \Box P$).

The persistence modality forms a comonad.

Pers-Mono
$$\frac{P \vdash Q}{\Box P \vdash \Box Q}$$

Pers-Elim
$\Box P \vdash P$

Pers-Idem
$\Box P = \Box\Box P$

The persistence modality commutes with the separating conjunction, conjunction, universal quantification, disjunction, existential quantification, and pure-proposition promotion.

Pers-Comm-Sep
$\Box(P * Q) = \Box P * \Box Q$

Pers-Comm-And
$\Box(P \wedge Q) = \Box P \wedge \Box Q$

Pers-Comm-All
$\Box(\forall x.\, P_x) = \forall x.\, \Box P_x$

Pers-Comm-Or
$\Box(P \vee Q) = \Box P \vee \Box Q$

Pers-Comm-Exist
$\Box(\exists x.\, P_x) = \exists x.\, \Box P_x$

Pers-Comm-Pure
$\Box \phi = \phi$

Also, under the persistence modality, the separating conjunction and the conjunction coincide and so do the magic wand and the implication.

PERS-EQ-SEP-AND
$$\Box(P * Q) \;=\; \Box(P \wedge Q)$$

PERS-EQ-WAND-IMP
$$\Box(P \mathrel{-\!\!*} Q) \;=\; \Box(P \Rightarrow Q)$$

We say that P is *persistent* and write $\mathsf{persistent}(P)$ when it is equal to $\Box P$.

$$\mathsf{persistent}(P) \quad := \quad P \;=\; \Box P$$

Persistent propositions satisfy the following properties.

PERSISTENT-EQ-SEP-AND
$$\frac{\mathsf{persistent}(P)}{P * Q \;=\; P \wedge Q}$$

PERSISTENT-EQ-WAND-IMP
$$\frac{\mathsf{persistent}(P)}{P \mathrel{-\!\!*} Q \;=\; P \Rightarrow Q}$$

PERSISTENT-INTRO-PERS
$$\frac{\mathsf{persistent}(P) \qquad P \vdash Q}{P \;\vdash\; \Box Q}$$

PERSISTENT-DUP
$$\frac{\mathsf{persistent}(P)}{P \;=\; P * P}$$

PERSISTENT-RETAIN
$$\frac{\mathsf{persistent}(P) \qquad Q \vdash P}{Q \;\vdash\; Q * P}$$

The rule PERSISTENT-INTRO-PERS says that if we can prove Q out of a persistent assumption, we can also prove $\Box Q$.

We have the following lemmas for finding persistent propositions.

PERSISTENT-PERS
$$\mathsf{persistent}(\Box P)$$

PERSISTENT-PURE
$$\mathsf{persistent}(\phi)$$

PERSISTENT-SEP
$$\frac{\mathsf{persistent}(P) \qquad \mathsf{persistent}(Q)}{\mathsf{persistent}(P * Q)}$$

PERSISTENT-AND
$$\frac{\mathsf{persistent}(P) \qquad \mathsf{persistent}(Q)}{\mathsf{persistent}(P \wedge Q)}$$

PERSISTENT-OR
$$\frac{\mathsf{persistent}(P) \qquad \mathsf{persistent}(Q)}{\mathsf{persistent}(P \vee Q)}$$

PERSISTENT-ALL
$$\frac{\forall x.\ \mathsf{persistent}(P_x)}{\mathsf{persistent}(\forall x.\, P_x)}$$

PERSISTENT-EXIST
$$\frac{\forall x.\ \mathsf{persistent}(P_x)}{\mathsf{persistent}(\exists x.\, P_x)}$$

Step Indexing Iris is a *step-indexed* logic. Step indexing allows us to introduce *general recursion* and *impredicativity*, which is an essential source of expressivity and extensibility of Iris. Later we introduce the *later modality* $\triangleright P$, which is an important connective for step indexing. Before introducing the later modality, we briefly introduce *step indexing* in Iris. See Jung et al. (2018c, §4) for details.

A *step index* is any natural number n (in the setting of Iris). Intuitively, the world is *stratified* over step indices and the world gets more *clearer* as the step index grows. Formally, a step-indexed type (also known as an ordered family of equivalences or OFE) is a type equipped with the *step-indexed equality* $(\overset{n}{=})_n$, which is a family of equivalence relations indexed by step indices which is anti-monotone over step indices and whose limit coincides with the equality. A function between step-indexed types is said to be *non-expansive* if it preserves the step-indexed equality. Also, a function f between step-indexed types is said to be *contractive* if it further satisfies $f\,x \overset{0}{=} f\,y$ and $x \overset{n}{=} y \Rightarrow f\,x \overset{n+1}{=} f\,y$. We have the following fixed point theorem on contractive functions, which is very useful in practice. Taking the fixed point by contractiveness can be understood as *coinduction over the step indices*.

Theorem 2.1 (Banach's Fixed Point Theorem)**.** *Any contractive function $f : T \to T$ over the step-indexed type T has a* unique fixed point*, i.e., has a unique solution x for the equation $x = f\,x$.*

A step-indexed proposition X: SProp is a predicate over step indices that is anti-monotone over step indices. The type SProp is equipped with the step-indexed equality $X \overset{n}{=} Y$, which is defined as $\forall\, m \le n.\ X\,m = Y\,m$, the equality at the step indices $0, \ldots, n$. An Iris proposition P: IProp is a step-indexed predicate over resources, with some restrictions and quotients. The type IProp is also equipped with the step-indexed equality in a manner similar to SProp.

Later Modality Iris has the *later modality* $\triangleright P$. Intuitively, $\triangleright P$ means that P holds *one step later*. More precisely, $\triangleright P$ holds at the step index 0 regardless of P and holds at the step index $n+1$ if P holds at the step index n.

The later modality $\triangleright$ is important for recursion and impredicativity because it is a *contractive* function over IProp. Other logical connectives are also designed to be *non-expansive*. So roughly speaking, a function over IProp composed by logical connectives of Iris is non-expansive *if all occurrences of the argument are under the later modality*. As we mentioned earlier, contractiveness allows us to take a *unique fixed point*, which is very useful for recursive definitions.

The later modality is monotone and can be introduced.

$$\textsc{Later-Mono}\ \frac{P \vdash Q}{\triangleright P \vdash \triangleright Q} \qquad \textsc{Later-Intro}\ P \vdash \triangleright P$$

The later modality is not idempotent because it just says 'one step later' instead of 'any steps later'.

Also, we have the following rule for *Löb induction*.

$$\textsc{Löb}\ \triangleright P \Rrightarrow P \;=\; P$$

It means that whenever we prove P we can assume $\triangleright P$. This is a strong deduction rule when we deal with recursive definition with the later modality.

The later modality also commutes with the truth, separating conjunction, conjunction, universal quantification, disjunction, inhabited existential quantification, implication, and persistent modality.

$$\textsc{Later-Comm-True}\ \triangleright \mathsf{True} \;=\; \mathsf{True} \qquad \textsc{Later-Comm-Sep}\ \triangleright(P * Q) \;=\; \triangleright P * \triangleright Q \qquad \textsc{Later-Comm-And}\ \triangleright(P \wedge Q) \;=\; \triangleright P \wedge \triangleright Q$$

$$\textsc{Later-Comm-All}\ \triangleright(\forall x.\, P_x) \;=\; \forall x.\, \triangleright P_x \qquad \textsc{Later-Comm-Or}\ \triangleright(P \vee Q) \;=\; \triangleright P \vee \triangleright Q \qquad \textsc{Later-Comm-InhExist}\ \frac{\exists \_ : T.\, \mathsf{True}}{\triangleright(\exists x : T.\, P_x) \;=\; \exists x : T.\, \triangleright P_x}$$

$$\textsc{Later-Comm-Pers}\ \triangleright \Box P \;=\; \Box \triangleright P$$

The following also holds because of Later-Comm-Pers.

$$\textsc{Persistent-Later}\ \frac{\mathsf{persistent}(P)}{\mathsf{persistent}(\triangleright P)}$$

In order to help discussion about the later modality, we introduce the following *except-0 modality*.

$$\diamond P \;:=\; \triangleright \mathsf{False} \lor P$$

It means that P holds at least except at the step index 0. It forms a monad.

$$\textsc{Except0-Mono}\ \frac{P \vdash Q}{\diamond P \vdash \diamond Q} \qquad \textsc{Except0-Intro}\ P \vdash \diamond P \qquad \textsc{Except0-Idem}\ \diamond\diamond P = \diamond P$$

It also satisfies the following properties.

$$\textsc{Except0-Later}\ \diamond\triangleright P = \triangleright P \qquad \textsc{Except0-Comm-Sep}\ \diamond(P * Q) = \diamond P * \diamond Q \qquad \textsc{Except0-Comm-And}\ \diamond(P \land Q) = \diamond P \land \diamond Q$$

$$\textsc{Except0-Comm-All}\ \diamond(\forall x.\, P_x) = \forall x.\, \diamond P_x \qquad \textsc{Except0-Comm-Or}\ \diamond(P \lor Q) = \diamond P \lor \diamond Q \qquad \textsc{Except0-Comm-Exist}\ \diamond(\exists x.\, P_x) = \exists x.\, \diamond P_x$$

$$\textsc{Except0-Comm-Pers}\ \diamond\Box P = \Box\diamond P$$

We say that P is *timeless* and write $\mathsf{timeless}(P)$ when $\triangleright P$ is equal to $\diamond P$.

$$\mathsf{timeless}(P) \;:=\; \triangleright P = \diamond P$$

If we peek into the model, $\mathsf{timeless}(P)$ turns out to be equivalent to saying that P is constant over the step indices.

We have the following lemmas for finding timeless propositions.

$$\textsc{Timeless-Pure}\ \mathsf{timeless}(\phi) \qquad \textsc{Timeless-Sep}\ \frac{\mathsf{timeless}(P) \qquad \mathsf{timeless}(Q)}{\mathsf{timeless}(P * Q)} \qquad \textsc{Timeless-And}\ \frac{\mathsf{timeless}(P) \qquad \mathsf{timeless}(Q)}{\mathsf{timeless}(P \land Q)}$$

$$\textsc{Timeless-Or}\ \frac{\mathsf{timeless}(P) \qquad \mathsf{timeless}(Q)}{\mathsf{timeless}(P \lor Q)} \qquad \textsc{Timeless-All}\ \frac{\forall x.\ \mathsf{timeless}(P_x)}{\mathsf{timeless}(\forall x.\, P_x)} \qquad \textsc{Timeless-Exist}\ \frac{\forall x.\ \mathsf{timeless}(P_x)}{\mathsf{timeless}(\exists x.\, P_x)}$$

$$\textsc{Timeless-Wand}\ \frac{\mathsf{timeless}(P)}{\mathsf{timeless}(Q \mathrel{-\!\!*} P)} \qquad \textsc{Timeless-Imp}\ \frac{\mathsf{timeless}(P)}{\mathsf{timeless}(Q \Rightarrow P)}$$

Fancy Update Modality Iris also has the *fancy update modality*.

First, we introduce some notions. We use some infinite type InvName and call its elements an *invariant name* and represent them by ι. Internally in the model of Iris, an invariant name is used for naming an *Iris proposition used for an invariant*. A *mask* $\mathcal{E}$: $\mathbb{P}\,\mathsf{InvName}$ is a set of invariant names. A *namespace* $\mathcal{N}$ is an *infinite* set of invariant names.[1] We write $\top$ for the universal set of the type InvName.

Iris has the *fancy update modality* ${}^{\mathcal{E}}\!\Rrightarrow^{\mathcal{E}'} P$ for masks $\mathcal{E}, \mathcal{E}'$. Roughly speaking, ${}^{\mathcal{E}}\!\Rrightarrow^{\mathcal{E}'} P$ represents a resource such that it can be updated into a resource satisfying P in a frame-preserving way; before the update we *own* the invariant names in $\mathcal{E}$, which can be used for the update, and after the update we *own* the invariant names in $\mathcal{E}'$. Actually, the fancy update modality is defined in Iris combining simpler connectives, which is described in (Jung et al., 2018c, §7).

[1] This formulation is a bit different from the actual Coq implementation in Iris.

The fancy update modality forms an indexed monad.

FUpd-Mono
$$\frac{P \vdash Q}{\mathcal{E}\Rrightarrow^{\mathcal{E}'} P \vdash \mathcal{E}\Rrightarrow^{\mathcal{E}'} Q}$$

FUpd-Intro
$$P \vdash \mathcal{E}\Rrightarrow^{\mathcal{E}} P$$

FUpd-Join
$$\mathcal{E}\Rrightarrow^{\mathcal{E}'} \mathcal{E}'\Rrightarrow^{\mathcal{E}''} P \vdash \mathcal{E}\Rrightarrow^{\mathcal{E}''} P$$

We can regard the fancy update modality as some *computation*, as is often the case for various monads in functional programming.

We also have the following stronger variant of FUpd-Intro.[2]

FUpd-Bounce
$$P \vdash \mathcal{E}+\mathcal{E}'\Rrightarrow^{\mathcal{E}} \mathcal{E}\Rrightarrow^{\mathcal{E}+\mathcal{E}'} P$$

The fancy update modality also satisfies the following *frame rule*.

FUpd-Frame
$$Q * \mathcal{E}\Rrightarrow^{\mathcal{E}'} P \vdash \mathcal{E}\Rrightarrow^{\mathcal{E}'} (Q * P)$$

The fancy update modality can clear away the except-0 modality.

FUpd-Except0
$$\diamond \mathcal{E}\Rrightarrow^{\mathcal{E}'} \diamond P = \mathcal{E}\Rrightarrow^{\mathcal{E}'} P$$

We can also derive the following rules.

FUpd-Merge
$$\mathcal{E}\Rrightarrow^{\mathcal{E}'} P * \mathcal{E}'\Rrightarrow^{\mathcal{E}''} Q \vdash \mathcal{E}\Rrightarrow^{\mathcal{E}''} (P * Q)$$

FUpd-Wkn-Mask
$$\mathcal{E}\Rrightarrow^{\mathcal{E}'} P \vdash \mathcal{E}+\mathcal{E}''\Rrightarrow^{\mathcal{E}'+\mathcal{E}''} P$$

FUpd-Merge follows from FUpd-Frame and FUpd-Join. FUpd-Wkn-Mask follows from FUpd-Bounce, FUpd-Frame and FUpd-Join.

The fancy update does *not* commute with any of the separating conjunction, magic wand, conjunction, universal quantification, disjunction, existential quantification, implication, persistence modality, and later modality.

We also use the following shorthand for propositions P, Q and masks $\mathcal{E}, \mathcal{E}'$.

$$P \mathrel{{}^{\mathcal{E}}\!\Rrightarrow\!\!\!*^{\mathcal{E}'}} Q \;:=\; P \mathrel{-\!\!*} \mathcal{E}\Rrightarrow^{\mathcal{E}'} Q \qquad P \mathrel{{}^{\mathcal{E}}\!\Rrightarrow^{\mathcal{E}'}} Q \;:=\; \Box\left(P \mathrel{{}^{\mathcal{E}}\!\Rrightarrow\!\!\!*^{\mathcal{E}'}} Q\right)$$

We call $P \mathrel{{}^{\mathcal{E}}\!\Rrightarrow\!\!\!*^{\mathcal{E}'}} Q$ a *view shift* from P to Q, under the input and output masks $\mathcal{E}, \mathcal{E}'$. Intuitively, $P \mathrel{{}^{\mathcal{E}}\!\Rrightarrow\!\!\!*^{\mathcal{E}'}} Q$ means that, by consuming a resource satisfying P and performing an update that inputs the mask $\mathcal{E}$ and outputs the mask $\mathcal{E}'$, we get a resource satisfying Q. The proposition $P \mathrel{{}^{\mathcal{E}}\!\Rrightarrow^{\mathcal{E}'}} Q$ means that the view shift $P \mathrel{{}^{\mathcal{E}}\!\Rrightarrow\!\!\!*^{\mathcal{E}'}} Q$ holds persistently; the symbol looks more like the implication than the magic wand, which is justified by the fact that under the persistent modality the implication and the magic wand coincide (Pers-Eq-Wand-Imp).

We can derive the following properties on the view shift.

VShift-Apply
$$P * (P \mathrel{{}^{\mathcal{E}}\!\Rrightarrow\!\!\!*^{\mathcal{E}'}} Q) \vdash \mathcal{E}\Rrightarrow^{\mathcal{E}'} Q$$

VShift-Id
$$\mathsf{True} \vdash P \mathrel{{}^{\mathcal{E}}\!\Rrightarrow\!\!\!*^{\mathcal{E}}} P$$

VShift-Comp
$$(P \mathrel{{}^{\mathcal{E}}\!\Rrightarrow\!\!\!*^{\mathcal{E}'}} Q) * (Q \mathrel{{}^{\mathcal{E}'}\!\Rrightarrow\!\!\!*^{\mathcal{E}''}} R) \vdash P \mathrel{{}^{\mathcal{E}}\!\Rrightarrow\!\!\!*^{\mathcal{E}''}} R$$

VShift-Merge
$$(P \mathrel{{}^{\mathcal{E}}\!\Rrightarrow\!\!\!*^{\mathcal{E}'}} Q) * (P' \mathrel{{}^{\mathcal{E}'}\!\Rrightarrow\!\!\!*^{\mathcal{E}''}} Q') \vdash P * P' \mathrel{{}^{\mathcal{E}}\!\Rrightarrow\!\!\!*^{\mathcal{E}''}} Q * Q'$$

[2] For sets A, B, we write $A + B$ for the disjoint union of them, i.e., the union $A \cup B$ with the precondition that the argument sets are disjoint $A \cap B = \emptyset$.

VSHIFT-WKN-MASK
$$P \ {}^{\mathcal{E}}\!\Rrightarrow\!\!\!*^{\mathcal{E}'} \ Q \ \vdash \ P \ {}^{\mathcal{E}+\mathcal{E}''}\!\Rrightarrow\!\!\!*^{\mathcal{E}'+\mathcal{E}''} \ Q$$

VSHIFT-EXCEPT0
$$P \ {}^{\mathcal{E}}\!\Rrightarrow\!\!\!*^{\mathcal{E}'} \ \diamond Q \ = \ P \ {}^{\mathcal{E}}\!\Rrightarrow\!\!\!*^{\mathcal{E}'} \ Q \ = \ \diamond P \ {}^{\mathcal{E}}\!\Rrightarrow\!\!\!*^{\mathcal{E}'} \ Q$$

For common use cases, the input and output masks for the fancy update are the same. So we introduce the following shorthand.

$$\Rrightarrow_{\mathcal{E}} P \ := \ {}^{\mathcal{E}}\!\Rrightarrow^{\mathcal{E}} P \qquad P \Rrightarrow\!\!*_{\mathcal{E}} \ Q \ := \ P \ {}^{\mathcal{E}}\!\Rrightarrow\!\!\!*^{\mathcal{E}} \ Q \qquad P \Rightarrow_{\mathcal{E}} \ Q \ := \ P \ {}^{\mathcal{E}}\!\Rightarrow^{\mathcal{E}} \ Q$$

We also introduce the following *step-taking* fancy update modality.

$$\Rrightarrow\!\!\!\Rightarrow_{\mathcal{E}} P \ := \ {}^{\mathcal{E}}\!\Rrightarrow^{\varnothing} \triangleright {}^{\varnothing}\!\Rrightarrow^{\mathcal{E}} P$$

It means that we get P after performing an update consuming the mask $\mathcal{E}$, spending one step index, and performing an update that returns the mask $\mathcal{E}$. Informally, when we have $\Rrightarrow\!\!\!\Rightarrow_{\mathcal{E}} P$, we say that after one *logical step* under the mask $\mathcal{E}$ we get P. We also introduce the following shorthand.

$$P \Rrightarrow\!\!\!\Rightarrow\!\!*_{\mathcal{E}} \ Q \ := \ P \mathrel{-\!\!*} \Rrightarrow\!\!\!\Rightarrow_{\mathcal{E}} Q \qquad P \Rrightarrow\!\!\!\Rightarrow_{\mathcal{E}} \ Q \ := \ \Box \left(P \Rrightarrow\!\!\!\Rightarrow\!\!*_{\mathcal{E}} \ Q \right)$$

Moreover, we also introduce the following shorthand for repeating logical steps.

$$\Rrightarrow\!\!\!\Rightarrow^{0}_{\mathcal{E}} P \ := \ \Rrightarrow_{\mathcal{E}} P \qquad \Rrightarrow\!\!\!\Rightarrow^{n+1}_{\mathcal{E}} P \ := \ \Rrightarrow\!\!\!\Rightarrow_{\mathcal{E}} \Rrightarrow\!\!\!\Rightarrow^{n}_{\mathcal{E}} P$$

$$P \Rrightarrow\!\!\!\Rightarrow\!\!*^{n}_{\mathcal{E}} \ Q \ := \ P \mathrel{-\!\!*} \Rrightarrow\!\!\!\Rightarrow^{n}_{\mathcal{E}} Q \qquad P \Rrightarrow\!\!\!\Rightarrow^{n}_{\mathcal{E}} \ Q \ := \ \Box \left(P \Rrightarrow\!\!\!\Rightarrow\!\!*^{n}_{\mathcal{E}} \ Q \right)$$

Note that $\Rrightarrow\!\!\!\Rightarrow^{0}_{\mathcal{E}} P$ is defined as $\Rrightarrow_{\mathcal{E}} P$ instead of P. Note also that $\Rrightarrow\!\!\!\Rightarrow^{1}_{\mathcal{E}} P$, $P \Rrightarrow\!\!\!\Rightarrow\!\!*^{1}_{\mathcal{E}} \ Q$ and $P \Rrightarrow\!\!\!\Rightarrow^{1}_{\mathcal{E}} \ Q$ are equal to $\Rrightarrow\!\!\!\Rightarrow_{\mathcal{E}} P$, $P \Rrightarrow\!\!\!\Rightarrow\!\!*_{\mathcal{E}} \ Q$ and $P \Rrightarrow\!\!\!\Rightarrow_{\mathcal{E}} \ Q$, respectively.

We can *derive* the following properties on the step-taking fancy update modality.

STFUPD-MONO
$$\frac{P \vdash Q}{\Rrightarrow\!\!\!\Rightarrow^{n}_{\mathcal{E}} P \ \vdash \ \Rrightarrow\!\!\!\Rightarrow^{n}_{\mathcal{E}} Q}$$

LATER-STFUPD
$$\triangleright^{n} P \ \vdash \ \Rrightarrow\!\!\!\Rightarrow^{n}_{\mathcal{E}} P$$

STFUPD-FUPD
$$\Rrightarrow_{\mathcal{E}} \Rrightarrow\!\!\!\Rightarrow^{n}_{\mathcal{E}} \Rrightarrow_{\mathcal{E}} P \ = \ \Rrightarrow\!\!\!\Rightarrow^{n}_{\mathcal{E}} P$$

STFUPD-WKN-MASK
$$\Rrightarrow\!\!\!\Rightarrow^{n}_{\mathcal{E}} P \ \vdash \ \Rrightarrow\!\!\!\Rightarrow^{n}_{\mathcal{E}+\mathcal{E}'} P$$

STFUPD-FRAME
$$Q \ * \ \Rrightarrow\!\!\!\Rightarrow^{n}_{\mathcal{E}} P \ \vdash \ \Rrightarrow\!\!\!\Rightarrow^{n}_{\mathcal{E}} (Q * P)$$

STFUPD-MERGE
$$\Rrightarrow\!\!\!\Rightarrow^{n}_{\mathcal{E}} P \ * \ \Rrightarrow\!\!\!\Rightarrow^{n}_{\mathcal{E}} Q \ \vdash \ \Rrightarrow\!\!\!\Rightarrow^{n}_{\mathcal{E}} (P * Q)$$

STFUPD-ADD-NSTEP
$$\Rrightarrow\!\!\!\Rightarrow^{n}_{\mathcal{E}} \Rrightarrow\!\!\!\Rightarrow^{m}_{\mathcal{E}} P \ = \ \Rrightarrow\!\!\!\Rightarrow^{n+m}_{\mathcal{E}} P$$

Note that step-taking updates can be merged *in parallel* by the rule STFUPD-MERGE, which follows from FUPD-MERGE and LATER-COMM-SEP. The rule STFUPD-WKN-MASK follows from FUPD-BOUNCE, FUPD-FRAME, LATER-INTRO and LATER-COMM-SEP.

The step-taking fancy update modality satisfies the following *soundness* theorem.

Theorem 2.2 (Soundness of the Step-Taking Fancy Update Modality)**.** *For any pure proposition* ϕ: Prop, *if* $\Rrightarrow\!\!\!\Rightarrow^{n}_{\mathcal{E}} \phi$ *is a tautology for some mask* $\mathcal{E}$ *and natural number* n*, then* ϕ *holds.*

Invariant In Iris, we can store an Iris proposition $\triangleright P$ to the world as an *invariant* and share it by the *invariant token* $\boxed{P}^{\mathcal{E}}$ which is *persistent*. The mask $\mathcal{E}$ is what we need to consume to take out the Iris proposition $\triangleright P$.

Formally, the *invariant token* $\boxed{P}^{\mathcal{E}}$ is defined as the following *persistent view shift*.

$$\boxed{P}^{\mathcal{E}} \ := \ \mathsf{True} \ {}^{\mathcal{E}}\!\Rrightarrow^{\varnothing} \triangleright P \ * \ \left(\triangleright P \ {}^{\varnothing}\!\Rrightarrow\!\!\!*^{\mathcal{E}} \ \mathsf{True} \right)$$

By consuming the mask $\mathcal{E}$, we can take out $\triangleright P$, and after we put back $\triangleright P$ we can retrieve the mask $\mathcal{E}$.

We can get an invariant token by the following rule.

$$\textsc{Inv-Intro} \quad \triangleright P \vdash \Rrightarrow_{\varnothing} \boxed{P}^{\mathcal{N}}$$

We can use any namespace $\mathcal{N}$, which is an infinite set of invariant names. Internally, we choose some unused invariant name ι from $\mathcal{N}$ (always only a finite number of invariant names are used), associate ι with P, and store $\triangleright P$ to some globally shared place; when we temporarily take out $\triangleright P$, we temporarily deposit the ownership on the invariant name ι to that shared place. Actually, in order to achieve this operation, the fancy update modality ${}^{\mathcal{E}}\Rrightarrow^{\mathcal{E}'}$ is equipped with the masks $\mathcal{E}, \mathcal{E}'$.

Later in § 3.3, we use the mechanism of the invariant to formalize the prophecy resolution (Proph-Resolve).

2.2 Customizing Iris by Resource Algebras

We can customize Iris by registering new *resource algebras* to the global resource. A resource algebra (RA) is a variant of a partial commutative monoid (PCM). In this section, we define the RA, introduce various RAs, and describe how we can customize Iris through RAs.

Resource Algebra A *resource algebra* (RA) Ra is a quintuple of (i) the underlying type $|Ra|$: Type, (ii) the *composition* operation $\cdot : |Ra| \to |Ra| \to |Ra|$, (iii) the *unit* $\varepsilon : |Ra|$, (iv) the *validity* predicate $\checkmark : |Ra| \to \mathrm{Prop}$, and (v) the *core* of each item $|-| : |Ra| \to |Ra|$, satisfying the following properties.[3]

$$\textsc{RA-Assoc}\ (a \cdot b) \cdot c = a \cdot (b \cdot c) \qquad \textsc{RA-Comm}\ a \cdot b = b \cdot a \qquad \textsc{RA-Unit}\ a \cdot \varepsilon = a \qquad \textsc{RA-Valid-Unit}\ a \cdot \varepsilon = a$$

$$\textsc{RA-Valid-Mono}\ \frac{a \preccurlyeq b \qquad \checkmark b}{\checkmark a} \qquad \textsc{RA-Core-Unit}\ a \cdot |a| = a \qquad \textsc{RA-Core-Idem}\ ||a|| = |a| \qquad \textsc{RA-Core-Mono}\ \frac{a \preccurlyeq b}{|a| \preccurlyeq |b|}$$

$$\text{where} \quad a \preccurlyeq b \ := \ \exists c.\, a \cdot b = c$$

The composition is associative, commutative and unital (RA-Assoc, RA-Comm, RA-Unit). The validity predicate holds for the unit and anti-monotone over composition (RA-Valid-Unit, RA-Valid-Mono). The core $|a|$ of an item a can be understood as what works like a unit in the presence of a (RA-Core-Unit). The core is idempotent and monotone over composition (RA-Core-Idem, RA-Core-Mono).

RA is similar to a partial commutative monoid (PCM) but different from that in that (i) RA uses a total composition operation and separately discusses validity, while PCM has a partial composition operation, and (ii) RA has the core operation, which PCM does not have.

We introduce the following relation $a \rightsquigarrow B$ for an item a: $|Ra|$ and a set of items B: $\mathbb{P}|Ra|$.

$$a \rightsquigarrow B \quad := \quad \forall c \text{ s.t. } \checkmark(a \cdot c).\ \exists b \in B.\ \checkmark(b \cdot c)$$

It means that we can safely update a into some element of the set B without losing validity for any *frame* c, i.e., the remaining part of the global resource. This kind of update is called *frame-preserving update*.

[3] This algebra is actually called a *unital* RA in Iris. Iris allows an RA to lack the unit and the totality of the core operation. In this thesis, we only use unital RAs for simplicity.

Various RAs Now we introduce various RAs, which are used later in the paper.

We call a positive rational number no greater than 1 a *fraction*. The type of fractions $\mathbb{Q}_{(0,1]}$ is defined as the subtype $\{\, a\colon \mathbb{Q} \mid 0 < a \le 1 \,\}$.[4] We use the letter q to represent a fraction.

The *fraction RA* $\textsc{frac}$ is defined as follows.

$$a\colon |\textsc{frac}| \;::=\; 0 \mid q \;\; (q\colon \mathbb{Q}_{(0,1]}) \mid \lightning \qquad a \cdot b \;:=\; \begin{cases} a+b & a, b \neq \lightning,\ a+b \le 1 \\ \lightning & \text{otherwise} \end{cases}$$

$$\varepsilon \;:=\; 0 \qquad \checkmark a \;:=\; a \neq \lightning \qquad |a| \;:=\; 0$$

An item of $\textsc{frac}$ is (i) 0, (ii) a fraction $q\colon \mathbb{Q}_{(0,1]}$, or (iii) $\lightning$, the invalid element.

The *exclusive RA* $\textsc{ex}(T)$ on the type T is defined as follows.

$$a\colon |\textsc{ex}(T)| \;::=\; \varepsilon \mid \mathsf{ex}\, x \;\; (x\colon T) \mid \lightning \qquad a \cdot b \;:=\; \begin{cases} a & b = \varepsilon \\ b & a = \varepsilon \\ \lightning & \text{otherwise} \end{cases}$$

$$\varepsilon \;:=\; \varepsilon \qquad \checkmark a \;:=\; a \neq \lightning \qquad |a| \;:=\; \varepsilon$$

An item of $\textsc{ex}(T)$ is (i) ε, the unit, (ii) $\mathsf{ex}\, x$, witnessing $x\colon T$ exclusively, or (iii) $\lightning$, invalid. The exclusive RA admits the following frame-preserving update for any $x, y\colon T$.

$$\mathsf{ex}\, x \rightsquigarrow \{\mathsf{ex}\, y\}$$

The *fractional ownership RA* $\textsc{fracown}(T)$ on the type T is defined as follows.

$$a\colon |\textsc{fracown}(T)| \;::=\; \varepsilon \mid \mathsf{fown}_q\, x \;\; (q\colon \mathbb{Q}_{(0,1]},\ x\colon T) \mid \lightning$$

$$a \cdot b \;:=\; \begin{cases} a & b = \varepsilon \\ b & a = \varepsilon \\ \mathsf{fown}_{q+q'}\, x & a = \mathsf{fown}_q\, x,\ b = \mathsf{fown}_{q'}\, x,\ q + q' \le 1 \\ \lightning & \text{otherwise} \end{cases}$$

$$\varepsilon \;:=\; \varepsilon \qquad \checkmark a \;:=\; a \neq \lightning \qquad |a| \;:=\; \varepsilon$$

An item of $\textsc{fracown}(T)$ is (i) ε, the unit, (ii) $\mathsf{fown}_q\, x$, witnessing $x\colon T$ with the fraction q, or (iii) $\lightning$, invalid. When we have $\mathsf{fown}_q\, x$ and $\mathsf{fown}_{q'}\, y$ in a valid way, the sum of fractions $q + q'$ should be no more than 1 and x and y should coincide. This fractional ownership RA admits the following frame-preserving update for any $x, y\colon T$, just like the exclusive RA.

$$\mathsf{fown}_1\, x \rightsquigarrow \{\mathsf{fown}_1\, y\}$$

Also note that we have the following rules on the fraction RA.

$$\frac{\checkmark\big(\mathsf{fown}_q\, x \cdot \mathsf{fown}_{q'}\, y\big)}{x = y \;\wedge\; q + q' \le 1} \qquad \frac{q + q' \le 1}{\mathsf{fown}_{q+q'}\, x \;=\; \mathsf{fown}_q\, x \cdot \mathsf{fown}_{q'}\, x}$$

The *product RA* $\prod_i Cmra_\lambda$ of an indexed family of RAs $(Cmra_i)_i$ is defined pointwise as follows.

$$\Big|\prod_i Cmra_i\Big| \;:=\; \prod_i |Cmra_i|$$

$$a \cdot b \;:=\; \lambda i.\, a\, i \cdot b\, i \qquad \varepsilon \;:=\; \lambda i.\, \varepsilon \qquad \checkmark a \;:=\; \forall i.\ \checkmark(a\, i) \qquad |a| \;:=\; \lambda i.\, |a\, i|$$

[4] In this thesis, we use the notation $\{\, x\colon T \mid \phi_x \,\}$ for the subtype of T: Type that consists of any value $x\colon T$ satisfying the condition ϕ_x. We implicitly promote a value of the type $\{\, x\colon T \mid \phi_x \,\}$ into the type T.

We can combine the frame-preserving updates of the component RAs in the product RA as follows.

$$\frac{\forall i.\ \ a\, i \rightsquigarrow B_i}{a \ \rightsquigarrow\ \prod_i B_i}$$

The *finite-support map RA* $T \xrightarrow{\text{fin}} Ra$ on the type T and the RA Ra is defined as follows.

$$|T \xrightarrow{\text{fin}} Ra| \ := \ \left\{ f : T \to |Ra| \ / \ \{ x \mid f\, x \neq \varepsilon \} \text{ is finite} \right\} \qquad a \cdot b \ := \ \lambda x.\, a\, x \cdot b\, x$$

$$\varepsilon \ := \ \lambda x.\, \varepsilon \qquad \checkmark a \ := \ \forall x.\ \checkmark (a\, x) \qquad |a| \ := \ \lambda x.\, |a\, x|$$

An item of $T \xrightarrow{\text{fin}} Ra$ is a function $f : T \to |Ra|$ such that $f\, x$ is non-unit for only a finite number of x. The operations on this RA are defined pointwise. For $x : T$ and $a : Ra$, we write $[x \leftarrow a]$ for the element of $T \xrightarrow{\text{fin}} Ra$ that returns a if the input is x and returns ε otherwise.[5] The finite-support map RA admits the following frame-preserving update, which 'allocates' a valid element b of the target RA Ra to some fresh place $x : T$ ($a\{x \leftarrow b\}$ denotes the map a with the output for x replaced with b).

$$\frac{\checkmark b}{a \ \rightsquigarrow\ \left\{ a\{x \leftarrow b\} \mid x : T,\ a\, x = \varepsilon \right\}}$$

The RA also admits the following frame-preserving update for modifying the item at one place.

$$\frac{a\, x \rightsquigarrow B}{a \ \rightsquigarrow\ \left\{ a\{x \leftarrow b\} \mid b \in B \right\}}$$

Global Camera We can also define a step-indexed version of a resource algebra, which is called a *camera*. The definition of the camera can be found in (Jung et al., 2018c, §4.4); in this thesis we omit it. Any resource algebra can be promoted into a *discrete* camera. We use *Cmra* to represent a camera.

An Iris proposition is parametrized over some camera, which we call the *global camera* and write as GlobCmra in this thesis. The global camera GlobCmra is constructed as follows for a finite set of *component cameras* CompCmras.[6]

$$\mathsf{GlobCmra} \ := \ \prod_{Cmra \in \mathsf{CompCmras}} \left(\mathsf{GhostName} \xrightarrow{\text{fin}} Cmra \right)$$

Here, GhostName is some infinite type, whose elements are called *ghost names* and are represented by γ. The type of Iris propositions IProp is defined based on this global camera GlobCmra. Actually, Iris has a mechanism for *impredicativity*, which allows GlobCmra to depend on IProp, using the fixed point theorem on locally contractive profunctors. For the purpose of this thesis, we don't need to get deeply into this mechanism.

A proof in Iris can be *extensible* by keeping it parametrized over the set of component cameras CompCmras, assuming only that the set contains the cameras we need. We benefit a lot from going *open-world*.

In RustHornBelt, we register our RAs to the global camera, i.e., add them to the set CompCmras — PROPH in §3.3, HEAP and TIME in §4.2, and UNQ in §5.3.

[5] For some T, we need some classical axioms to construct $[x \leftarrow a]$.

[6] The actual Coq implementation of Iris uses a finite sequence of component cameras possibly with duplication in order to avoid equality on cameras.

Customizing Iris by Cameras/RAs Iris has the connective $\mathsf{Own}(a)$ for *owning* the resource a: |GlobCmra| in the global camera. It satisfies the following properties.[7]

OWN-COMP
$$\mathsf{Own}(a) * \mathsf{Own}(b) = \mathsf{Own}(a \cdot b)$$

OWN-UNIT
$$\mathsf{True} = \mathsf{Own}(\varepsilon)$$

OWN-CORE-PERS
$$\mathsf{persistent}(\mathsf{Own}(|a|))$$

OWN-VALID
$$\mathsf{Own}(a) \vdash \checkmark a$$

OWN-UPD
$$\frac{a \rightsquigarrow B}{\mathsf{Own}(a) \vdash \Rightarrow_{\varnothing} \exists\, b \in B.\ \mathsf{Own}(b)}$$

We can merge and split Own according to the composition of the camera (OWN-COMP). We always own the unit (OWN-UNIT). Owning the core of any item is persistent (OWN-CORE-PERS). Owning an item ensures that the item is valid (OWN-VALID). When the frame-preserving update $a \rightsquigarrow B$ holds, consuming $\mathsf{Own}(a)$ we get $\mathsf{Own}(b)$ for some b in B (OWN-UPD).

Importantly, the frame-preserving update $a \rightsquigarrow B$ we can use for OWN-UPD can be extended by registering new cameras to the global camera.

Usually, instead of directly using Own, we use the connective $\lceil a \rceil^{\gamma}_{Cmra}$, which owns the resource a: $|Cmra|$ at the specific ghost name γ and the camera $Cmra$, where $Cmra$ is in the set of component cameras CompCmras.

$$\lceil a \rceil^{\gamma}_{Cmra} \quad := \quad \left[Cmra \leftarrow [\gamma \leftarrow a] \right]$$

It satisfies the following properties, which follows from the properties on Own.

GOWN-COMP
$$\lceil a \rceil^{\gamma}_{Cmra} * \lceil b \rceil^{\gamma}_{Cmra} = \lceil a \cdot b \rceil^{\gamma}_{Cmra}$$

GOWN-UNIT
$$\mathsf{True} = \lceil \varepsilon \rceil^{\gamma}_{Cmra}$$

GOWN-CORE-PERS
$$\mathsf{persistent}(\lceil |a| \rceil^{\gamma}_{Cmra})$$

GOWN-VALID
$$\lceil a \rceil^{\gamma}_{Cmra} \vdash \checkmark a$$

GOWN-TIMELESS
$$\frac{Cmra \text{ is a discrete camera}}{\mathsf{timeless}(\lceil a \rceil^{\gamma}_{Cmra})}$$

GOWN-ALLOC
$$\frac{\checkmark a}{\mathsf{True} \vdash \Rightarrow_{\varnothing} \exists\, \gamma.\ \lceil a \rceil^{\gamma}_{Cmra}}$$

GOWN-UPD
$$\frac{a \rightsquigarrow B}{\lceil a \rceil^{\gamma}_{Cmra} \vdash \Rightarrow_{\varnothing} \exists\, b \in B.\ \lceil b \rceil^{\gamma}_{Cmra}}$$

GOWN-UPD-TWO
$$\frac{(a, a') \rightsquigarrow B \qquad Cmra \neq Cmra'}{\lceil a \rceil^{\gamma}_{Cmra} * \lceil a' \rceil^{\gamma'}_{Cmra'} \vdash \Rightarrow_{\varnothing} \exists\, (b, b') \in B.\ \left(\lceil b \rceil^{\gamma}_{Cmra} * \lceil b' \rceil^{\gamma'}_{Cmra'} \right)}$$

GOWN-ALLOC says that a valid element a can be allocated to some ghost name γ. GOWN-UPD-TWO can discuss interaction of two different cameras $Cmra, Cmra'$ registered to the global camera; the frame preserving update $(a, a') \rightsquigarrow B$ is discussed in the product camera $Cmra \times Cmra'$.

2.3 Lifetime Logic — Borrowing Propositions Under Lifetimes

The *lifetime logic* was introduced by Jung et al. (2018a,b) to discuss borrowing of resources under *lifetimes* in Rust. The lifetime logic is formalized as a group of definitions and lemmas in the separation logic Iris with Coq mechanization. Using the *impredicativity* of the Iris, the lifetime logic allows us to *borrow any Iris proposition P*, which is very important for *extensibility* of the proofs with regard to borrowing. Our work, RustHornBelt, depends a lot on the lifetime logic (Chapter 5 and Chapter 6).

[7] The validity predicate $\checkmark a$ used in OWN-VALID is actually step-indexed.

The Iris model of the lifetime logic is highly complicated and subtle, which is well described and explained by Jung (2020, Chapter 11). However, if we just *use* the lifetime logic, we just need to understand the deduction rules of the lifetime logic.

In this chapter, we describe how we can use the lifetime logic. To aid understanding, the formulas and names of the deduction rules are a bit modified from those of (Jung et al., 2018b; Jung, 2020).

Presenting Deduction Rules in Iris A deduction rule like the following for an *Iris* proposition P means that P is a tautology.

$$\frac{}{P}\ \textsc{Some-Simple-Rule}$$

A deduction rule like the following for *Iris* propositions $\vec{Q}, P$ means that $Q_0 \land Q_1 \land \cdots \land Q_{n-1} \Rrightarrow P$ is a tautology.

$$\frac{Q_0 \qquad Q_1 \qquad \cdots \qquad Q_{n-1}}{P}\ \textsc{Some-Rule-With-Assumptions}$$

To aid understanding, we only use *persistent propositions* for the assumptions $\vec{Q}$.

Sometimes we may hide some common assumptions that are persistent. For the lifetime logic, we hide the *lifetime context* $\mathsf{Ctx}_{\mathsf{lft}}$, which is an invariant token on some complicated proposition.

Lifetimes A *lifetime* α, β is an object of some type Lft.

We have the *intersection* operation over lifetimes $\alpha \sqcap \beta$, which is associative, commutative, and unital under the *eternity* lifetime ∞.

$$(\alpha \sqcap \alpha') \sqcap \alpha'' = \alpha \sqcap (\alpha' \sqcap \alpha'') \qquad \alpha \sqcap \beta = \beta \sqcap \alpha \qquad \infty \sqcap \alpha = \alpha$$

Lifetime Tokens and Dead-Lifetime Tokens In order to manage the information about the liveness of lifetimes, we use the lifetime token $[\alpha]_q$ and the dead lifetime token $[\dagger\alpha]$.

The lifetime token $[\alpha]_q$ witnesses with the fraction q that the lifetime α is still alive. The dead-lifetime token $[\dagger\alpha]$ persistently witnesses that the lifetime α has been dead.

Both types of tokens are timeless. A dead-lifetime token is persistent.

$$\mathsf{timeless}([\alpha]_q) \qquad \mathsf{persistent}([\dagger\alpha]) \qquad \mathsf{timeless}([\dagger\alpha])$$

We can introduce a new lifetime α by the following rule.

$$\textsc{Lft-Intro}\quad \mathsf{True} \Rrightarrow_{\mathcal{N}_{\mathsf{lft}}} \exists\, \alpha.\ [\alpha]_1 * \left([\alpha]_1 \mathrel{=\!\!\!\!\!\!\Rightarrow}_{\mathcal{N}_{\mathsf{lft}}} [\dagger\alpha] \right)$$

We get the full lifetime token $[\alpha]_1$ and the step-taking view shift $[\alpha]_1 \mathrel{=\!\!\!\!\!\!\Rightarrow}_{\mathcal{N}_{\mathsf{lft}}} [\dagger\alpha]$. The view shift allows us to end the lifetime one step after we consume the full lifetime token $[\alpha]_1$.

A lifetime token can be split and merged according to the fraction.

$$\textsc{LftToken-Frac}\quad [\alpha]_{q+q'} \Leftrightarrow [\alpha]_q * [\alpha]_{q'}$$

A lifetime cannot be alive and dead at the same time.

$$\textsc{LftToken-DeadLft}\ \frac{[\dagger\alpha] \qquad [\alpha]_q}{\text{False}}$$

The intersection operation and the null lifetime interact with the lifetime and dead-lifetime tokens as follows.

$$\textsc{LftToken-Intrsct}\ [\alpha \sqcap \beta]_q \Leftrightarrow [\alpha]_q * [\beta]_q \qquad \textsc{DeadLft-Intrsct}\ [\dagger(\alpha \sqcap \beta)]_q \Leftrightarrow [\dagger\alpha] \lor [\dagger\beta]$$

$$\textsc{LftToken-Eter}\ [\infty]_q \qquad \textsc{DeadLft-Eter}\ [\dagger\infty] \Rrightarrow \text{False}$$

Lifetime Inclusion We define *inclusion between lifetimes* $\alpha \sqsubseteq \beta$ as follows.

$$\alpha \sqsubseteq \beta \;:=\; \left(\forall q.\ [\alpha]_q \Rrightarrow_{\mathcal{N}_{\text{lft}}} \exists q'.\ [\beta]_{q'} * \left([\beta]_{q'} \mathrel{-\!\!*}_{\mathcal{N}_{\text{lft}}} [\alpha]_q\right)\right) \land \left([\dagger\beta] \Rrightarrow_{\mathcal{N}_{\text{lft}}} [\dagger\alpha]\right)$$

It means that (i) out of a fractional lifetime token on α of any fraction q, we can temporarily take out a fractional lifetime token on β of some fraction q', and (ii) if we have the dead lifetime token on β, we can make the dead lifetime token on α. Intuitively, it means that the lifetime β lives longer than α. We say the lifetime β *outlives* the lifetime α when $\alpha \sqsubseteq \beta$ holds.

We can use lifetime inclusion by the following rules.

$$\textsc{LftIncl-LftToken}\ \frac{\alpha \sqsubseteq \beta}{[\alpha]_q \Rrightarrow_{\mathcal{N}_{\text{lft}}} \exists q'.\ [\beta]_{q'} * \left([\beta]_{q'} \mathrel{-\!\!*}_{\mathcal{N}_{\text{lft}}} [\alpha]_q\right)} \qquad \textsc{LftIncl-DeadLft}\ \frac{\beta \sqsubseteq \alpha \qquad [\dagger\alpha]}{\text{True} \Rrightarrow_{\mathcal{N}_{\text{lft}}} [\dagger\beta]}$$

We can *derive* the following rules for deducing lifetime inclusion.

$$\textsc{LftIncl-Refl}\ \alpha \sqsubseteq \alpha \qquad \textsc{LftIncl-Trans}\ \frac{\alpha \sqsubseteq \alpha' \qquad \alpha' \sqsubseteq \alpha''}{\alpha \sqsubseteq \alpha''}$$

$$\textsc{LftIncl-Intrsct-L}\ \alpha \sqcap \beta \sqsubseteq \alpha \qquad \textsc{LftIncl-Intrsct-R}\ \frac{\alpha \sqsubseteq \beta \qquad \alpha \sqsubseteq \beta'}{\alpha \sqsubseteq \beta \sqcap \beta'} \qquad \textsc{LftIncl-Eter}\ \alpha \sqsubseteq \infty$$

Full Borrows A *full borrow* $\&^{\alpha}_{\text{full}} P$ fully borrows the *Iris proposition* P: IProp under the lifetime α. It corresponds to a unique reference in Rust.

For soundness of impredicativity, the full borrow owns $\triangleright P$ instead of just P, just like invariants.

We can introduce a full borrow by the following rule.

$$\textsc{FullBor-Intro}\ \triangleright P \Rrightarrow_{\mathcal{N}_{\text{lft}}} \&^{\alpha}_{\text{full}} P * \left([\dagger\alpha] \mathrel{-\!\!*}_{\mathcal{N}_{\text{lft}}} \triangleright P\right)$$

By consuming $\triangleright P$, we get a full borrow $\&^{\alpha}_{\text{full}} P$ and the view shift $[\dagger\alpha] \mathrel{-\!\!*}_{\mathcal{N}_{\text{lft}}} \triangleright P$. Using the view shift, we can retrieve $\triangleright P$ after the lifetime α ends.

We can temporarily access $\triangleright P$ out of a full borrow $\&^{\alpha}_{\text{full}} P$ with help of a fractional lifetime token $[\alpha]_q$.

$$\textsc{FullBor-Access}\ \&^{\alpha}_{\text{full}} P * [\alpha]_q \Rrightarrow_{\mathcal{N}_{\text{lft}}} \triangleright P * \left(\triangleright P \mathrel{-\!\!*}_{\mathcal{N}_{\text{lft}}} \&^{\alpha}_{\text{full}} P * [\alpha]_q\right)$$

The fractional lifetime token $[\alpha]_q$ is returned after we put back $\triangleright P$.

We can also *reborrow* a full borrow $\&^{\beta}_{\text{full}} P$ from a full borrow $\&^{\alpha}_{\text{full}} P$ instead of $\triangleright P$.

FullBor-Rebor
$$\frac{\beta \sqsubseteq \alpha}{\&^{\alpha}_{\text{full}} P \;\Rrightarrow_{\mathcal{N}_{\text{lft}}}\; \&^{\beta}_{\text{full}} P \;*\; \left([\dagger\beta] \;\mathrel{\text{-}\!\!*}\!\!\Rrightarrow_{\mathcal{N}_{\text{lft}}}\; \&^{\alpha}_{\text{full}} P\right)}$$

It corresponds to reborrowing a unique reference out of a unique reference in Rust.

We can modify the lifetime and the proposition of a full borrow.

FullBor-Mono-Lft
$$\frac{\beta \sqsubseteq \alpha}{\&^{\alpha}_{\text{full}} P \;\Rightarrow\; \&^{\beta}_{\text{full}} P}$$

FullBor-Iff
$$\frac{\triangleright \Box (P \Leftrightarrow Q)}{\&^{\alpha}_{\text{full}} P \;\Leftrightarrow\; \&^{\alpha}_{\text{full}} Q}$$

Note that a version of FullBor-Iff with $\Leftrightarrow$ replaced with one-side implication $\Rightarrow$ does not hold, because the target proposition P of a full borrow $\&^{\alpha}_{\text{full}} P$ represents both what we get from and what we have to put into the full borrow.

We also have various operations on full borrows.

We can split and merge full borrows according to separating conjunction.

FullBor-Split
$$\&^{\alpha}_{\text{full}} (P * Q) \;\Rrightarrow_{\mathcal{N}_{\text{lft}}}\; \&^{\alpha}_{\text{full}} P \;*\; \&^{\alpha}_{\text{full}} Q$$

FullBor-Merge
$$\&^{\alpha}_{\text{full}} P \;*\; \&^{\alpha}_{\text{full}} Q \;\Rrightarrow_{\mathcal{N}_{\text{lft}}}\; \&^{\alpha}_{\text{full}} (P * Q)$$

We can freeze an internal parameter inside the target proposition of a full borrow, as long as the type T of the variable is inhabited.

FullBor-Freeze
$$\frac{\exists \_ : T.\ \mathsf{True}}{\&^{\alpha}_{\text{full}} (\exists\, a : T.\, P_a) \;\Rrightarrow_{\mathcal{N}_{\text{lft}}}\; \exists\, a : T.\ \&^{\alpha}_{\text{full}} P_a}$$

A later modality inside a full borrow can be removed in one logical step.

FullBor-Unlater
$$\&^{\alpha}_{\text{full}} (\triangleright P) \;\mathrel{\text{=}\!\!\Rrightarrow}_{\mathcal{N}_{\text{lft}}}\; \&^{\alpha}_{\text{full}} P$$

A full borrow of a full borrow can be unnested into a full borrow with the intersection lifetime in one logical step.

FullBor-Unnest
$$\&^{\alpha}_{\text{full}} \&^{\beta}_{\text{full}} P \;\mathrel{\text{=}\!\!\Rrightarrow}_{\mathcal{N}_{\text{lft}}}\; \&^{\alpha \sqcap \beta}_{\text{full}} P$$

We have the following strong rule for *subdividing* a full borrow $\&^{\alpha}_{\text{full}} P$ into a new full borrow $\&^{\alpha}_{\text{full}} Q$.

FullBor-Subdiv
$$\&^{\alpha}_{\text{full}} P \;*\; [\alpha]_q \;\Rrightarrow_{\mathcal{N}_{\text{lft}}}\; \triangleright P \;*\; \forall\, Q.\ \left(\triangleright Q \;*\; (\triangleright Q \;\mathrel{\text{-}\!\!*}\!\!\Rrightarrow_{\varnothing} \triangleright P) \;\mathrel{\text{-}\!\!*}\!\!\Rrightarrow_{\mathcal{N}_{\text{lft}}}\; \&^{\alpha}_{\text{full}} Q \;*\; [\alpha]_q\right)$$

We first consume the full borrow $\&^{\alpha}_{\text{full}} P$ with help of a fractional lifetime token $[\alpha]_q$. We access the content $\triangleright P$. Then we can decide the Iris proposition Q, possibly depending on internal parameters of P. We have to construct $\triangleright Q$ and also a view shift $\triangleright Q \mathrel{\text{-}\!\!*}\!\!\Rrightarrow_{\varnothing} \triangleright P$ that turns $\triangleright Q$ into $\triangleright P$ under the empty mask $\varnothing$. By consuming them, we finally get the new subdivided full borrow $\&^{\alpha}_{\text{full}} Q$ and also get back $[\alpha]_q$.

Fractured Borrows A *fractured borrow* $\&^{\alpha}_{\mathrm{frac}}\,\Phi$ is a *persistent* proposition that borrows an Iris proposition parametrized over a fraction $\Phi\colon \mathbb{Q}_{(0,1]} \to \mathsf{IProp}$.

$$\mathsf{persistent}(\&^{\alpha}_{\mathrm{frac}}\,\Phi)$$

We can make a fractured borrow of Φ out of a full borrow of $\Phi(1)$.

$$\textsc{FullBor-FracBor}\quad \&^{\alpha}_{\mathrm{full}}\,\Phi(1) \;\Rrightarrow_{\mathcal{N}_{\mathsf{lft}}}\; \&^{\alpha}_{\mathrm{frac}}\,\Phi$$

With help of a lifetime token, we can temporarily access the content of a fractured borrow $\&^{\alpha}_{\mathrm{frac}}\,\Phi$, getting a proposition $\triangleright\,\Phi(q)$ for *some* fraction q.

$$\textsc{FracBor-Access}\quad \frac{\&^{\alpha}_{\mathrm{frac}}\,\Phi \qquad \Box\left(\forall\, q_0, q_1.\ \Phi(q_0+q_1) \;\Leftrightarrow\; \Phi(q_0) * \Phi(q_1)\right)}{[\alpha]_{q'} \;\Rrightarrow_{\mathcal{N}_{\mathsf{lft}}}\; \exists\, q.\ \ \triangleright\,\Phi(q) * \left(\triangleright\,\Phi(q) \;\Rrightarrow\!\!*_{\mathcal{N}_{\mathsf{lft}}}\; [\alpha]_{q'}\right)}$$

The second assumption persistently witnesses that Φ can be split and merged according to the fraction.

We can modify the lifetime and the target predicate of fractured borrows.

$$\textsc{FracBor-Mono-Lft}\quad \frac{\&^{\alpha}_{\mathrm{frac}}\,\Phi \qquad \beta \sqsubseteq \alpha}{\&^{\beta}_{\mathrm{frac}}\,\Phi}$$

$$\textsc{FracBor-Iff}\quad \frac{\triangleright\,\Box\left(\forall\, q.\ \Phi(q) \;\Leftrightarrow\; \Phi'(q)\right)}{\&^{\alpha}_{\mathrm{frac}}\,\Phi \;\Leftrightarrow\; \&^{\alpha}_{\mathrm{frac}}\,\Phi'}$$

Also, we can *dynamically create lifetime inclusion* $\alpha \sqsubseteq \beta$ by introducing a fractured borrow of a fractional lifetime token of β under the lifetime α.

$$\textsc{FracBor-LftIncl}\quad \frac{\&^{\alpha}_{\mathrm{frac}}\left(\lambda q'.\,[\beta]_{q\cdot q'}\right)}{\alpha \sqsubseteq \beta}$$

Note that we can create the fractured borrow $\&^{\alpha}_{\mathrm{frac}}\left(\lambda q'.\,[\beta]_{q\cdot q'}\right)$ by the following view shift, which can be derived from FullBor-Intro and FullBor-FracBor.

$$[\beta]_q \;\Rrightarrow_{\mathcal{N}_{\mathsf{lft}}}\; \&^{\alpha}_{\mathrm{frac}}\left(\lambda q'.\,[\beta]_{q\cdot q'}\right) * \left([\dagger\alpha] \;\Rrightarrow\!\!*_{\mathcal{N}_{\mathsf{lft}}}\; [\beta]_q\right)$$

Chapter 3

A New Formulation of Prophecy

In program verification, our reasoning usually goes forward, along the execution of the program. For some advanced reasoning, however, we need to know and use some information about *what happens in the future* (i.e., events occurring later in the execution). For example, we sometimes need to know in advance what non-deterministic choices will be made in the future. Also, even if the verified program is deterministic, we can often avoid elaborate analysis of the program by peeking out information about the future. To address that need, *prophecy variables* are employed. A prophecy variable is a special variable that conveys information about the future for verification.[1] Roughly speaking, we use a prophecy variable in the following way. We create a prophecy variable x at some point. From then on, we can use the value of x for verification to refer to some information about the future. Later at a certain point, after some program execution, we construct some value v based on information we have at the moment, and we *resolve* x to v, i.e., we finally assign to x the value v.

The idea of prophecy variables has been long used in various contexts. Abadi and Lamport (1988, 1991) introduced the idea of prophecy variables for a new technique of verifying refinement between state machines. Later, prophecy variables were used by Sezgin et al. (2010) to verify non-interference on concurrent programs and by Cook and Koskinen (2011) and Lamport and Merz (2017) to automatically verify temporal properties of transition systems. Jung et al. (2020b) presented a separation logic with prophecy variables by extending Iris, which was used by them to verify logical atomicity of the RDCSS operation and by de Vilhena et al. (2020) to verify that the local generic solver computes the least fixed point. And as we discussed in §1.3, *RustHorn* by Matsushita et al. (2020a) is a notable application of prophecy variables for verification of imperative programs in Rust.

For our work RustHornBelt, aiming at unifying RustHorn and RustBelt, we newly built a new flexible framework of prophecy in the *separation logic Iris*, which is different in many aspects from existing work. Prophecy variables live only in the *ghost state* and allows resolution of a prophecy variable to a value that *depends on other prophecy variables*. In this chapter, we introduce this new framework of prophecy. In §3.1, we explain the motivation of our framework in light of what we need for RustHornBelt. In §3.2, we present a high-level overview of our framework. In §3.3, we present the formulation of our framework. In §3.4, we discuss related work.

[1] Some literature uses the term prophecy variable only for a form of *auxiliary variable*, i.e., a variable employed in *ghost code* that was added to the original program for verification. In this thesis, we use the term prophecy variable in a broader sense to refer to any kind of variable that describes something about the future for verification.

3.1 Motivation

What kind of framework of prophecy do we need for our work RustHornBelt? To model Rust in the style of RustHorn on top of the separation logic Iris, we should be able to resolve the prophecy variable of a unique reference into a *value described in the ghost state* modeling the target object of the unique references. Moreover, when we release a unique reference to some object that itself contains other unique references, which we call a *nested unique reference*, we have to resolve its prophecy variable into a value depending on *other prophecy variables* (later we show a detailed example of this), which we call *dependent resolution*. Also, in order to tackle Rust's elaborate lifetime-based ownership principle in an *extensible* way, we need to build our framework of prophecy in an extensible and powerful separation logic like *Iris*. In particular, it will be great if we can to reuse the *lifetime logic* (introduced in §2.3), which is built on top of Iris.

However, none of the existing formulations of prophecy variables were sufficiently flexible and powerful for our project RustHornBelt. The framework by Jung et al. (2020b) targets a OCaml-like *value-based* language, and requires us to add to the original program some ghost code manipulating prophecy variables, in order to observe *physical values* in the execution. This is far from sufficient for modeling Rust's unique references with prophecies. The proof of soundness and completeness of RustHorn by Matsushita et al. (2020b) tackles this difficulty by focusing on a formalization of a *safe* subset of Rust (not allowing unsafe code) and building a bisimulation between execution of a Rust program and some deduction algorithm (named SLDC resolution) on the CHCs, or *logic formulas*, generated by RustHorn as a model of the program. In the proof, prophecy variables of unique references are encoded as universally quantified *syntactic variables* in a logic formula, which can be specialized later in the deduction algorithm. The use of syntactic variables is justified by soundness and completeness of the deduction algorithm (with respect to the least-fixed-point model of the CHCs). In this way, we can support dependent resolution in a natural way. Although this proof works well in that setting, assuming that verified Rust programs do not have unsafe code, it is rather *fragile*. For the proof, we have to carefully design the transition systems of the formalization of Rust and the deduction algorithm, as well as the CHCs generated for a Rust program, and we also have to depend heavily on syntactic structures of logic formulas. So it was unclear how to extend the proof to support Rust programs with unsafe code for in general.

This is the reason why we decided to try to design a new formulation of prophecy that is sufficient for our work RustHornBelt. Let us discuss more in detail the features we want for our new formulation of prophecy through some examples of Rust code.

To begin with, consider the following very simple code.

```
1 let mut a : i32 = 1;   let ua : &mut i32 = &mut a;  /* (i) */
2 *ua += 2;  /* (ii) */  print!("{}", a); // 3
```

By the unique borrow `&mut a`, we introduce a new prophecy variable `x` that represents the value of `a` at the end of the borrow.[2] At (i), `ua` is modeled as $(1, x)$, a pair of the current value 1 and the prophecy variable x, so that x can be resolved to an appropriate value when `ua` is released. Also, at (i), `a` is modeled as x so that we can access the value of `a` when we use `a` again. By the operation `*ua += 2`, the model of `ua` is updated into $(3, x)$. Then at (ii), we release `ua` and we resolve the prophecy variable x to the value 3. Now we can end the borrow and retrieve `a`. As we know that x has been resolved to 3, at last we successfully know that `a` has the value 3.

[2] In this thesis, hereinafter, we usually use x, y, z for prophecy variables and a, b, c for program variables.

When we handle *nested unique references* (i.e., unique references to an object containing unique references), the situation can be very tricky and we need *dependent resolution.*

Example 3.1 (Nested Unique Reference). Let us consider the following Rust code.

```
let mut a = 0;   let mut b = 1;
let mut uc = &mut a;   *uc += 2;   let ub = &mut b;  /* (i) */
let uuc = &mut uc;  /* (ii) */  *uuc = ub;  /* (iii) */
*uc += 3;  /* (iv) */  print!("{}, {}", a, b); // 2, 4
```

Here, `a` and `b` are typed `i32`, `uc` and `ub` are typed `&mut i32`, and `uuc` is typed `&mut &mut i32` — `uuc` is a nested unique reference. We omit lifetime information since it is not relevant to the discussion. Let x, y and z be the prophecy variables for the unique borrows `uc = &mut a`, `ub = &mut b` and `uuc = &mut uc`. At (i), `uc` is modeled as $(2, x)$, `ub` as $(1, y)$, `a` as x, and `b` as y. At (ii), after the borrow for `uuc`, `uuc` is modeled $((2, x), z)$ and the model of `uc` is updated into z.

At (iii), after the update `*uuc = ub`, the model of `uuc` is updated into $((1, y), z)$. Since $(2, x)$ is lost here, we resolve x to 2. Now, we release the nested unique reference `uuc`, modeled as $((1, y), z)$, and thus resolve z to $(1, y)$ — *resolution of a prophecy variable z to a value depending on another prophecy variable y.* This is an instance of *dependent resolution.* Through this resolution, `uc` is now modeled as $(1, y)$.

At (iv), the model of `uc` is updated into $(4, y)$. Here we release `uc` and resolve y to 4. Now, since x and y have been resolved to 2 and 4, we know that `a` and `b` have the values 2 and 4 at last.

If we allow dependent resolution without restriction, we can accidentally introduce circular dependencies between prophecies, causing unsoundness. For example, if we can resolve x to $y + 1$ and y to $x + 1$, we fall into a contradiction. Naturally, we can use the following simple and intuitive restriction: a prophecy variable can only be resolved into a value depending on *other unresolved prophecy variables.* This is a good idea in modeling Rust, because resolution dependencies can be determined in a dynamic way in Rust.

Example 3.2 (Dynamically Determined Resolution Dependency). For example, suppose the following code, which uses a tricky recursive type `MNat` featuring the unique-reference type:

```
enum MNat<'a> { MZero, MSucc( &'a mut MNat<'a> ) }   use MNat::*;
let mut a = MZero;   let mut b = MZero;
let ua = &mut a;   let ub = &mut b;
if rand() { *ua = MSucc(ub); } else { *ub = MSucc(ua); }
print!("{}, {}", a, b);
```

Here, we assume some non-deterministic function `fn rand() -> bool`. Let x and y be the prophecy variables for `ua` and `ub`. Now `ua` is modeled as (MZero, x) and `ub` is modeled as (MZero, y). If `rand()` returns true, `*ua` is updated into $\mathsf{MSucc}(ub)$ and `ua` is released; thereby the prophecy variable x is resolved into $\mathsf{MSucc}((\mathsf{MZero}, y))$, making x depend on y. Otherwise, we do exactly the other way around; y is resolved into $\mathsf{MSucc}((\mathsf{MZero}, x))$, making y depend on x. Importantly, whether x depends on y or y depends on x is determined by a *dynamic condition* `rand()`. In particular, the lifetime information is insufficient to capture this kind of dependency, as both `ua` and `ub` have the same lifetime.

3.2 Overview of Our Formulation of Prophecy

The difficulty with prophecy is that we *don't know* what happens in the future in a usual sense, especially when the program is non-deterministic. Then how can we model prophecy variables *within the ghost state*?

Intuitively, it seems natural to model prophecies in the following way. Every time we take a new prophecy variable x, the current world *ramifies* into many *possible worlds* with different values assigned to x. When we resolve x to some value v, we *prune* all the possible worlds that assign to x any value different from v. The soundness of this model is guaranteed by the fact that we never lose all the possible worlds, i.e., the constraint over possible worlds is always satisfiable. We can even perform dependent resolution as long as we can ensure satisfiability of the constraint. Although this approach works, ramification and pruning of possible worlds seems to be a *global* operation which conflicts with the *modular, local reasoning of separation logic*.

Instead, we can take a slightly different approach. First, we fix a *closed set of prophecy variables*, which is infinite. Each time we want to introduce a new prophecy variable, we can choose a fresh prophecy variable from this set. Next, we introduce the notion of a *prophecy assignment* π, a map that assigns to each prophecy variable a value of some expected type, representing a possible world. The key idea is that we *retain a set of possible worlds* that remain valid (i.e., have not yet been pruned) in each Hoare triple, instead of choosing one specific possible world for each Hoare triple like Jung et al. (2020b) did. We can share information about the conditions that the valid possible worlds satisfy. Note that, since we have a closed set of possible worlds, we never ramify possible worlds more and only prune them. For prophecy-based reasoning, we pervasively use a *value parametrized over a possible world / prophecy assignment*, which we call a *π-parametrized value* and represent by a letter with a hat like $\hat{v}, \hat{w}$. A π-parametrized value can be regarded as a value depending on values of some prophecy variables. This framework supports *dependent resolution*: we can resolve a prophecy variable x to a π-parametrized value $\hat{v}$ depending only on other unresolved prophecy variables; after the resolution, we know that $\pi\, x = \hat{v}\, \pi$ holds for any valid prophecy assignment π.

We can naturally adapt this idea to the modular reasoning of separation logic, especially Iris. In order to manage resolution of prophecy variables, we use a *prophecy token* $[x]_q$, which observes with a fraction q that the prophecy variable x has not yet been resolved. When we introduce a fresh prophecy variable x, we get a full prophecy token $[x]_1$. We also use a persistent token $\langle \pi.\ \phi_\pi \rangle$ called a *prophecy observation*, which means that a pure proposition ϕ_π holds for any valid prophecy assignment π. In the notation $\langle \pi.\ \phi_\pi \rangle$, the first part $\pi.$ binds the variable π and the second part ϕ_π describes a condition on π. Now, dependent resolution is supported in the following way: we can resolve x into a π-parametrized value $\hat{v}$ to get a prophecy observation $\langle \pi.\ \pi\, x = \hat{v}\, \pi \rangle$, by consuming the full prophecy token $[x]_1$ and temporarily using a partial prophecy token $[y]_q$ (which ensures that y is unresolved and different from x) for every dependency y of $\hat{v}$. We can modify the prophecy observations acquired through resolution by merging (i.e., getting $\langle \pi.\ \phi_\pi \wedge \psi_\pi \rangle$ out of $\langle \pi.\ \phi_\pi \rangle$ and $\langle \pi.\ \psi_\pi \rangle$) and weakening (i.e., getting $\langle \pi.\ \phi'_\pi \rangle$ out of $\langle \pi.\ \phi_\pi \rangle$ and $\forall\, \pi.\, \phi_\pi \Rightarrow \phi'_\pi$). We use the following lemma to link a prophecy observation to the 'reality': when we have $\langle \pi.\ \hat{\phi}\, \pi \rangle$, there exists some prophecy assignment π_* satisfying $\hat{\phi}\, \pi_*$. This lemma intuitively means that there always remains some possible world that has not been pruned. Notably, this formulation of prophecies is completely independent of how we define the weakest precondition predicate or the Hoare triple.

This very simple formulation of prophecies is flexible enough to model unique references in Rust. Each variable of Rust is now modeled as a *π-parametrized value*. Each

unique reference is equipped with the full prophecy token $[x]_1$ of its prophecy variable x.

Example 3.3 (Nested Unique Reference, Revisited). To see how our framework works, let us reconsider the Rust program of Example 3.1, an example of using a nested unique reference.

```
1 let mut a = 0;   let mut b = 1;
2 let mut uc = &mut a;   *uc += 2;   let ub = &mut b;  /* (i) */
3 let uuc = &mut uc;  /* (ii) */  *uuc = ub;  /* (iii) */
4 *uc += 3;  /* (iv) */  print!("{}, {}", a, b); // 2, 4
```

On the unique borrows `&mut a` and `&mut b`, we create full prophecy tokens $[x]_1$ and $[y]_1$ for fresh prophecy variables x and y, which are given to the unique references `uc` and `ub`. At (i), `uc` is modeled as a π-parametrized value $\lambda\pi.(2, \pi\, x)$, `ub` as $\lambda\pi.(1, \pi\, y)$, `a` as $\lambda\pi.\pi\, x$ and `b` as $\lambda\pi.\pi\, y$.

At (ii), letting z be the prophecy variable for the borrow `&mut uc`, `uc` is modeled as $\lambda\pi.\pi\, z$ and `uuc` as $\lambda\pi.((2, \pi\, x), \pi\, z)$. Through the update `*uuc = ub`, consuming the full prophecy token $[x]_1$ previously owned by `*uuc`, we *resolve* x into 2 to get a prophecy observation $\langle\pi.\ \pi\, x = 2\rangle$.

At (iii), `uuc` is modeled as $\lambda\pi.((1, \pi\, y), \pi\, z)$, and we release `uuc` here. We *resolve* z into $(1, \pi\, y)$ by consuming $[z]_1$ owned by `uuc` and temporarily accessing $[y]_1$ owned by `*uuc`, and get $\langle\pi.\ \pi\, z = (1, \pi\, y)\rangle$. This is *dependent resolution*. We temporarily use $[y]_1$ to ensure that y has not been resolved.

When we access again `uc` in the line 4, we know that `uc` is modeled as *some* π-parametrized value $\hat{v}$ satisfying $\langle\pi.\ \hat{v}\ \pi = \pi\, z\rangle$. The type of `uc` tells us that $\hat{v}$ is of the form $\lambda\pi.(n, \pi\, y')$ for some integer n and prophecy variable y'. We get $\langle\pi.\ (n, \pi\, y') = (1, \pi\, y)\rangle$ from the two previously introduced prophecy observations, which entails $n = 1$ and $\langle\pi.\ \pi\, y' = \pi\, y\rangle$. Here, we got $n = 1$ from $\langle\pi.\ n = 1\rangle$. Although we don't obtain $y = y'$ here, it suffices to just have $\langle\pi.\ \pi\, y' = \pi\, y\rangle$ for verification.

At (iv), after the update `*uc += 3`, `uc` is modeled as $\lambda\pi.(4, \pi\, y')$, and by releasing `uc` we get $\langle\pi.\ \pi\, y' = 3\rangle$. Now, from the prophecy observations we have acquired, we get $\langle\pi.\ \pi\, x = 2 \land \pi\, y = 4\rangle$. Therefore, we successfully know that the values of `a` and `b` are set to 2 and 4 at last.

3.3 Technical Details

Now we discuss technical details of our formulation of prophecy variables described in the previous section. Although the idea was originally conceived for our project RustHornBelt, this formulation is designed so that it can be employed for a general purpose. We assume that we work on a dependent type system like Coq, but our notation does not strictly follow the style of Coq.

Basic Concepts A *prophecy variable* x, y, z is modeled as an object of the record type

$$\mathsf{ProphVar} \ := \ (\mathsf{discr}\colon \mathbb{N},\ \mathsf{type}\colon \mathsf{InhType}),$$

consisting of (discr) the discriminator number and (type) the type of a value assigned to the variable. Here, InhType denotes the type of *inhabited* types and is defined as the subtype $\{\, T\colon \mathsf{Type} \ /\ \exists\ \_\colon T.\ \mathsf{True}\,\}$.[3] Notably, for the value type of a prophecy variable,

[3] Note that Coq prevents unsoundness about a 'type of types' by having a *hierarchy of universes*. The type Type actually has an implicit parameter i representing some universe, being of the form Type_i. For any i, the type Type_i belongs to the type Type_{i+1}, but not to Type_i. In the definition of InhType, Type is in fact Type_i for some universe level i.

we can use any inhabited type; we can use especially a predicate type $T \to \mathsf{Prop}$, which is essential for modeling Rust's function type in RustHornBelt (see §6.1.4).

A *prophecy assignment* π is an object of the dependent function type

$$\mathsf{ProphAsn} \;:=\; (x\colon \mathsf{ProphVar}) \to x.\mathsf{type},$$

which assigns to each prophecy variable some value of the expected type. A prophecy assignment can be regarded as a representation of a possible world. Since for any prophecy variable x the type $x.\mathsf{type}$ is inhabited, it is guaranteed that there exists some prophecy assignment (i.e., $\exists\, \pi\colon \mathsf{ProphAsn}.\, \mathsf{True}$ holds),[4] which is a vital property for this framework (and is later used for the proof of Theorem 3.1).

A *π-parametrized value* $\hat{v}, \hat{w}$ over a type T is a map of the type $\mathsf{ProphAsn} \to T$, which is an important machinery in this framework. In particular, later in the formulation of RustHornBelt, each object in Rust is associated with a π-parametrized value (see Chapter 5).

A key notion about π-parametrized values is the relation $\mathsf{Dep}(\hat{v}, X)$, meaning that a π-parametrized value $\hat{v}\colon \mathsf{ProphAsn} \to T$ depends only on values assigned to the prophecy variables in the set $X\colon \mathbb{P}\,\mathsf{ProphVar}$. For example, the following holds.

$$\mathsf{Dep}\big(\,\lambda\pi.\ \mathsf{inj}_0((1, \pi\, x), \pi\, y),\ \{x, y\}\,\big)$$

Let us define this relation. First, we introduce an equivalence relation on prophecy assignments $\pi \equiv_X \pi'$, meaning that π and π' assign exactly the same value to every prophecy variable in the set X. It is simply defined as follows.

$$\pi \equiv_X \pi' \;:=\; \forall\, x \in X.\ \pi\, x = \pi'\, x$$

Each equivalence class on $(\equiv_X)$ is characterized exactly by what values are assigned to the prophecy variables in X. Therefore, $\mathsf{Dep}(\hat{v}, X)$ can be defined as the property that $\hat{v}$ stays constant within each equivalence class on $(\equiv_X)$, which is formally described as follows.

$$\mathsf{Dep}(\hat{v}, X) \;:=\; \forall\, \pi, \pi' \text{ s.t. } \pi \equiv_X \pi'.\ \hat{v}\, \pi = \hat{v}\, \pi'$$

We have the following lemmas on the relation $\mathsf{Dep}(\hat{v}, X)$.

$$\textsc{Dep-One}\quad \mathsf{Dep}(\lambda\pi.\, \pi\, x,\ \{x\})$$

$$\textsc{Dep-Construct}\quad \frac{\forall\, i.\ \mathsf{Dep}(\hat{v}_i, X_i) \qquad f \text{ does not depend on } \pi}{\mathsf{Dep}\big(\lambda\pi.\, f(\overrightarrow{\hat{v}\,\pi}),\ \bigcup_i X_i\big)}$$

$$\textsc{Dep-Destruct}\quad \frac{\mathsf{Dep}\big(\lambda\pi.\, f(\overrightarrow{\hat{v}\,\pi}),\ X\big) \qquad f \text{ is injective}}{\mathsf{Dep}(\hat{v}_i, X)}$$

The basic π-parametrized value $\lambda\pi.\, \pi\, x$ depends only on x (Dep-One). Dep-Construct says that, when we have π-parametrized values $\vec{\hat{v}}$ depending only on $\vec{X}$ respectively, the π-parametrized value constructed by applying a function f (which does not depend on π) to them depends only on the union of the sets $\vec{X}$. Conversely, Dep-Destruct says that, if the π-parametrized value constructed by applying an injective function f (which does not depend on π) to $\vec{\hat{v}}$ depends only on X, then each $\hat{v}_i$ depends only on X.

[4] In Coq, we use the functional choice axiom to prove the existence.

Prophecy Log To manage resolution of prophecy variables, we introduce the notion of a *prophecy log* $\mathcal{P}$. It is a list $[(x_0, \hat{v}_0), \ldots, (x_{n-1}, \hat{v}_{n-1})]$ which records the resolutions that have been performed in reverse chronological order; each item $(x_i, \hat{v}_i)$ means that a prophecy variable x_i has been resolved to a π-parametrized value $\hat{v}_i$. When we resolve a prophecy variable x to a π-parametrized value $\hat{v}$, we update the prophecy log $\mathcal{P}$ into $(x, \hat{v}) :: \mathcal{P}$. Formally, each item of a prophecy log is of the dependent pair type

$$\mathsf{ProphLogItem} \;:=\; (x\colon \mathsf{ProphVar}) \times (\mathsf{ProphAsn} \to x.\mathsf{type})$$

and a prophecy log has the type

$$\mathsf{ProphLog} \;:=\; \mathsf{List}\ \mathsf{ProphLogItem}.$$

The set of resolutions in a prophecy log $\mathcal{P}$ forms a constraint over prophecy assignments. We use the following relation $\pi \triangleleft \mathcal{P}$ to describe the constraint.

$$\pi \triangleleft \mathcal{P} \;:=\; \forall\, (x, \hat{v})\ \mathsf{in}\ \mathcal{P}.\ \pi\, x = \hat{v}\, \pi$$

It is essential to always make sure that the prophecy log $\mathcal{P}$ is satisfiable, i.e., there exists some prophecy assignment π_* that satisfies $\pi_* \triangleleft \mathcal{P}$. We can achieve that by ensuring the following property: each time we resolve a prophecy variable x to a π-parametrized value $\hat{v}$, x has never been resolved before that and $\hat{v}$ depends only on other unresolved prophecy variables. Formally, we model this idea as the following *validity* condition $\checkmark\, \mathcal{P}$ over a prophecy log $\mathcal{P}$.

$$\checkmark\big((x, \hat{v}) :: \mathcal{P}\big) \;:=\; x \in \mathsf{unres}(\mathcal{P}) \;\wedge\; \mathsf{Dep}\big(\hat{v}, \mathsf{unres}(\mathcal{P}) - \{x\}\big) \;\wedge\; \checkmark\, \mathcal{P} \qquad \checkmark\, \mathsf{nil} := \mathsf{True}$$

Here, $\mathsf{unres}(\mathcal{P})$ denotes the set of prophecy variables that have not been resolved by $\mathcal{P}$, i.e., the complement of $\{\, x \mid \exists\, \hat{v}.\ (x, \hat{v})\ \mathsf{in}\ \mathcal{P}\,\}$. As we intended, we achieve the following theorem.

Theorem 3.1 (Satisfiability of a Valid Prophecy Log)**.** *For any prophecy log $\mathcal{P}_*$ that satisfies $\checkmark\, \mathcal{P}_*$, there exists a prophecy assignment π_* that satisfies $\pi_* \triangleleft \mathcal{P}_*$.*

Proof. We introduce the following function $\mathsf{modify}_{\mathcal{P}}\, \pi$ that modifies a prophecy assignment π based on the information of $\mathcal{P}$.

$$\mathsf{modify}_{(x,\hat{v})::\mathcal{P}}\, \pi \;:=\; \mathsf{modify}_{\mathcal{P}}\big(\pi\{x \leftarrow \hat{v}\, \pi\}\big) \qquad\qquad \mathsf{modify}_{\mathsf{nil}}\, \pi \;:=\; \pi$$

Here, $\pi\{x \leftarrow \hat{v}\, \pi\}$ is a prophecy assignment that assigns $\hat{v}\, \pi$ to x and $\pi\, y$ to any $y \neq x$. We inductively show $\mathsf{modify}_{\mathcal{P}}\, \pi \equiv_{\mathsf{unres}(\mathcal{P})} \pi$ for any $\mathcal{P}$. Using that, we inductively show $\mathsf{modify}_{\mathcal{P}}\, \pi \triangleleft \mathcal{P}$ for any $\mathcal{P}$ satisfying $\checkmark\, \mathcal{P}$. By taking some prophecy assignment π_0, we set $\pi_* := \mathsf{modify}_{\mathcal{P}_*}(\pi_0)$.[5] □

Machinery for Prophecy on Iris Now, using the notions formulated so far, we build the machinery for prophecy upon the separation logic Iris.

First, we introduce the following RA PROPH, named the *prophecy RA*.[6]

$$|\textsc{proph}| \;:=\; |\textsc{ex}(\mathsf{ProphLog})| \times (\mathsf{ProphAsn} \to \mathsf{Prop}) \times |\mathsf{ProphVar} \xrightarrow{\mathsf{fin}} \textsc{frac}|$$

$$(a, \hat{\phi}, f) \cdot (b, \hat{\psi}, g) \;:=\; \big(a \cdot b,\ \lambda \pi.\ \hat{\phi}\, \pi \wedge \hat{\psi}\, \pi,\ f \cdot g\big)$$

[5] We need classical axioms to discuss equality over Type.

[6] We can use $\mathbb{N} \xrightarrow{\mathsf{fin}} \mathsf{InhType} \to \textsc{frac}$ instead of $\mathsf{ProphVar} \xrightarrow{\mathsf{fin}} \textsc{frac}$ for this RA to simplify formalization about finiteness.

$$\varepsilon \;:=\; (\varepsilon,\ \lambda\pi.\mathsf{True},\ \varepsilon) \qquad\qquad |(a,\hat\phi,f)| \;:=\; (\varepsilon,\hat\phi,\varepsilon)$$

$$\begin{aligned}\checkmark(a,\hat\phi,f) \;:=\;& \checkmark a \;\wedge\; (\exists\pi.\,\hat\phi\,\pi) \;\wedge\; \checkmark f \;\wedge\\ & \forall\mathcal{P}\text{ s.t. }\mathsf{ex}\,\mathcal{P}=a.\ \checkmark\mathcal{P}\;\wedge\;\big(\forall\pi\triangleleft\mathcal{P}.\,\hat\phi\,\pi\big)\;\wedge\;\big(\forall x\notin\mathsf{unres}(\mathcal{P}).\ f\,x=0\big)\end{aligned}$$

An item $(a,\hat\phi,f)$ of this RA consists of (i) an item a: $|\textsc{ex}(\mathsf{ProphLog})|$ (later used for the *prophecy context*), (ii) a known condition on valid prophecy assignments $\hat\phi$: $\mathsf{ProphAsn}\to\mathsf{Prop}$ (later used for *prophecy observations*), and (iii) a finite-support map from prophecy variables to extended fractions f: $|\mathsf{ProphVar}\xrightarrow{\text{fin}}\textsc{frac}|$ that manages unresolved prophecy variables (later used for *prophecy tokens*).

For definition, we piggyback the unit elements ε and the compositions $(\cdot)$ of the RAs $\textsc{ex}(\mathsf{ProphLog})$ and $\mathsf{ProphVar}\xrightarrow{\text{fin}}\textsc{frac}$. The composition of the prophecy RA is defined component-wise; for the $\mathsf{ProphAsn}\to\mathsf{Prop}$ part, we take the (pointwise) logical conjunction. Since logical conjunction is idempotent, the core $|(a,\hat\phi,f)|$ can be set to $(\varepsilon,\hat\phi,\varepsilon)$ (later this makes prophecy observations persistent).

The validity predicate $\checkmark$ of this RA is important. An item $(\mathsf{ex}\,\mathcal{P},\hat\phi,f)$ is valid when (i) $\mathcal{P}$ is valid, (ii) $\hat\phi\,\pi$ holds for any π satisfying $\mathcal{P}$ (which entails satisfiability of $\hat\phi$ by Theorem 3.1), and (iii) f is valid and $f\,x=0$ holds for any x resolved by $\mathcal{P}$. An item $(\varepsilon,\hat\phi,f)$ is valid when $\hat\phi$ is satisfiable and f is valid.

Now we register the prophecy RA $\textsc{proph}$ as a component of the global camera and take some fresh namespace $\mathcal{N}_{\mathsf{proph}}$. We define the following *prophecy context* $\mathsf{Ctx}_{\mathsf{proph}}$.

$$\mathsf{Ctx}_{\mathsf{proph}} \;:=\; \boxed{\exists\mathcal{P}.\ \boxed{(\mathsf{ex}\,\mathcal{P},\ \lambda\pi.\mathsf{True},\ \varepsilon)}^{\gamma_{\mathsf{proph}}}_{\textsc{proph}}}^{\mathcal{N}_{\mathsf{proph}}}$$

It is a persistent token that refers to an invariant in $\mathcal{N}_{\mathsf{proph}}$ that governs the prophecy log $\mathcal{P}$ at a ghost name γ_{proph}. We need this context to resolve prophecy variables (see the lemma Proph-Resolve). The ghost name γ_{proph} is chosen when we introduce the context $\mathsf{Ctx}_{\mathsf{proph}}$ by the following lemma.

Proph-Ctx-Intro
$$\mathsf{True} \;\Rrightarrow_{\varnothing}\; \exists\gamma_{\mathsf{proph}}.\ \mathsf{Ctx}_{\mathsf{proph}}$$

Since $\mathsf{Ctx}_{\mathsf{proph}}$ is persistent, later on we implicitly assume $\mathsf{Ctx}_{\mathsf{proph}}$.

After γ_{proph} is fixed, we can define the following *prophecy observation* $\langle\pi.\ \phi_\pi\rangle$ and *prophecy token* $[x]_q$.

$$\langle\pi.\ \phi_\pi\rangle \;:=\; \boxed{(\varepsilon,\ \lambda\pi.\phi_\pi,\ \varepsilon)}^{\gamma_{\mathsf{proph}}}_{\textsc{proph}} \qquad\qquad [x]_q \;:=\; \boxed{(\varepsilon,\ \lambda\pi.\mathsf{True},\ [x\leftarrow q])}^{\gamma_{\mathsf{proph}}}_{\textsc{proph}}$$

A prophecy observation $\langle\pi.\ \phi_\pi\rangle$ persistently witnesses that the pure proposition ϕ_π holds for any valid prophecy assignment π. A prophecy token $[x]_q$ witnesses with the fraction q: $\mathbb{Q}_{(0,1]}$ that the prophecy variable x has not been resolved. We also use the shorthand $[X]_q := \ast_{x\in X}[x]_q$ for X: $\mathbb{P}_{\text{fin}}\,\mathsf{ProphVar}$.

The following properties hold.

$$\mathsf{persistent}(\langle\pi.\ \phi_\pi\rangle) \qquad \mathsf{timeless}(\langle\pi.\ \phi_\pi\rangle) \qquad \mathsf{timeless}([x]_q)$$

ProphToken-Intro
$$\frac{\exists\,\_ : T.\,\mathsf{True}}{\mathsf{True}\;\Rrightarrow_{\varnothing}\;\exists x\text{ s.t. }x.\mathsf{type}=T.\ [x]_1}$$

ProphToken-Frac
$$[x]_{q+q'} \;\Leftrightarrow\; [x]_q \ast [x]_{q'}$$

Proph-Resolve
$$\frac{\mathsf{Dep}(\hat v, Y)}{[x]_1 \ast [Y]_q \;\Rrightarrow_{\mathcal{N}_{\mathsf{proph}}}\; \langle\pi.\ \pi\,x=\hat v\,\pi\rangle \ast [Y]_q}$$

ProphObs-Merge
$$\frac{\langle\pi.\ \phi_\pi\rangle \qquad \langle\pi.\ \psi_\pi\rangle}{\langle\pi.\ \phi_\pi\wedge\psi_\pi\rangle}$$

$$\textsc{ProphObs-Wkn}\ \frac{\forall\, \pi.\ \phi_\pi \Rrightarrow \psi_\pi \qquad \langle \pi.\ \phi_\pi \rangle}{\langle \pi.\ \psi_\pi \rangle} \qquad \textsc{ProphObs-Sat}\ \frac{\langle \pi.\ \hat{\phi}\, \pi \rangle}{\exists\, \pi_*.\ \hat{\phi}\, \pi_*} \qquad \textsc{ProphObs-Fact}\ \frac{\langle \pi.\ \phi \rangle}{\phi}$$

ProphToken-Intro says that we can always take a fresh prophecy variable x of any expected (inhabited) type T and get a full prophecy token $[x]_1$. A prophecy token can be split and merged according to the fraction (ProphToken-Frac). Proph-Resolve says that, by consuming a full prophecy token of x, as long as we have a partial prophecy token on the prophecy variables that a π-parametrized value $\hat{v}$ may depend on, we can resolve x to $\hat{v}$ and obtain a prophecy observation $\pi\, x = \hat{v}\, \pi$ (note that we assume here that $\hat{v}$ has the type ProphAsn $\to$ x.type). Prophecy observations can be merged (ProphObs-Merge) and weakened (ProphObs-Wkn). ProphObs-Sat says that a prophecy observation $\langle \pi.\ \hat{\phi}\, \pi \rangle$ implies that there exists a prophecy assignment π_* satisfying the predicate $\hat{\phi}$. ProphObs-Fact says that having a prophecy observation $\langle \pi.\ \phi \rangle$, where ϕ does not depend on π, is the same thing as knowing that ϕ holds.

Proof of ProphToken-Intro. Because only a finite number of prophecy variables have been *used* in the global resource, i.e., resolved by the prophecy log or included in the support of the map ProphVar $\xrightarrow{\text{fin}}$ frac, we can always find a prophecy variable that has not been used yet. □

Proof of Proph-Resolve. We update the prophecy log in the invariant from $\mathcal{P}$ to $(x, \hat{v}) :: \mathcal{P}$. The validity condition on the prophecy log is retained, because the prophecy tokens $[x]_1$ and $[Y]_q$ ensure that x has not been resolved, that each y in Y has not been resolved, and that x is not included in Y. Also, by consuming the full token $[x]_1$, we are allowed to switch the state of x from unresolved to resolved. Because the constraint on a prophecy assignment π denoted by the prophecy log just changes from $\pi \triangleleft \mathcal{P}$ to $\pi\, x = \hat{v}\, \pi \ \wedge\ \pi \triangleleft \mathcal{P}$, we can obtain the prophecy observation $\langle \pi.\ \pi\, x = \hat{v}\, \pi \rangle$. □

Proof of ProphObs-Sat. By the definition of the validity predicate ✓ of proph. □

Proof of ProphObs-Fact. The forward implication follows by ProphObs-Sat. The backward implication follows by the definition of ε in proph. □

3.4 Related Work

The idea of prophecy variables was first introduced in the theoretical work by Abadi and Lamport (1988, 1991) as the 'mirror image' of history variables (auxiliary variables describing the past events), for a new technique of verifying refinement between state machines. In general, we can verify that a state machine S_1 refines S_2 (i.e., the observable behaviors of S_1 are a subset of those of S_2) by finding a refinement mapping from S_1 to S_2 (i.e., a behavior-preserving map from S_1's states to S_2's states). We can get a more powerful verification method by using history and prophecy variables. A state machine S_1^* obtained by adding history and prophecy variables to S_1 has the same traces with S_1, and existence of a refinement mapping from S_1^* to S_2 entails refinement of S_2 by S_1. Actually, this new method is complete: under some conditions, if S_1 refines S_2, there exist a state machine S_1^*, obtained by adding some history and prophecy variables to S_1, and a refinement mapping from S_1^* to S_2. Here, prophecies are employed to know in advance what non-deterministic choices S_2 makes in the future. This completeness theorem is the main theoretical result of their work. Note that in their verification method they make a modified state machine S_1^* with prophecy variables, which can be understood as addition of some prophecy-related ghost code to programs.

The studies of Vafeiadis (2008, §5.3.3), Zhang et al. (2012) and Jung et al. (2020b) employed prophecy variables for verification in separation logic. At a high level, all of them basically took the same approach. They added to the program some *ghost code* that introduces prophecy variables as *physical* variables and resolves them by information observed *physically*. They performed Hoare-style verification over the modified program with the prophecy-handling ghost code, letting each Hoare triple choose *one specific possible world* on prophecies (in contrast to our approach of retaining a *set of possible worlds*). Also, they separately proved the *erasure theorem*, saying that wished properties on the original program can be ensured by verifying the modified program with ghost code. As a key use case, all of them targeted verification of linearizability of the RDCSS (restricted double-compare single-swap) operation proposed by Harris et al. (2002). We say a concurrent operation is *linearizable* if it has a single physical step called the *linearization point* where the relevant part of the *ghost state* is *atomically* updated, even though the operation may not be physically atomic, i.e., may take multiple physical steps. Verifying linearizability of the RDCSS operation naturally requires prophecies because the linearization point depends on information about the future (specifically, whether the current thread wins the race to perform some completion operation). Because atomicity over the ghost state, or *logical atomicity*, is technically subtle, their approach of adding prophecy-handling ghost code is fairly reasonable. In fact, Jung et al. (2020b) successfully verified a very strong version of linearizability for the RDCSS operation. Still, we did not take their approach for our project RustHornBelt, because resolution of the prophecy variables of unique references actually depends on information about the *ghost state*, rather than the physical state, in the presence of unsafe code. If we were to take their approach for our purpose, we would have to prove our version of the *erasure theorem* with regard to the ghost state, which seems highly non-trivial.

The idea of retaining a set of possibilities about the future has been employed in the context of separation logic by Turon et al. (2013), in the name of *speculation*. They used speculation to verify linearizability (via contextual refinement) of a simplified version of the RDCSS operation, called conditional increment. Later, Liang and Feng (2013) extended the idea to verify the original version of RDCSS. Both studies took basically the same approach, which is quite different from ours. Each separation-logic proposition P is parametrized over a *set of speculative states* Σ, which is non-empty and *finite*. They use the *speculative choice* $P \oplus Q$ over propositions P and Q, which says that the set of speculative states Σ is the union of some Σ_0 satisfying P and Σ_1 satisfying Q. When they perform some speculation about the future, where each case corresponds to $P_0, \ldots, P_n$, they obtain $P_0 \oplus \cdots \oplus P_n$. When an expression e can turn P into P' and Q into Q', e can turn $P \oplus Q$ into $P' \oplus Q'$. When they have $P \oplus Q$ and (physically) observe that the speculation of P is correct, they can get P out of $P \oplus Q$. They successfully used the mechanism to do case analysis for verifying linearizability of conditional increment or RDCSS. However, in their frameworks they could not do speculation on an unbounded integer value, for example, which introduces an *infinite* number of possibilities, especially because of the nature of the speculative choice ($\oplus$) operator. Therefore, their frameworks were unsuitable for our project RustHornBelt.

Matsushita et al. (2020b) proved soundness and completeness of RustHorn's reduction from Rust programs to CHCs by establishing bisimulation between the execution of a Rust program and the course of some deduction algorithm, named SLDC resolution, on the CHCs generated by RustHorn. SLDC resolution was specially designed for this work, extending the idea of SLD resolution proposed by Kowalski (1974). In general, a resolution[7] algorithm over CHCs can be understood as *top-down* construction (i.e.,

[7] *Resolution* over CHCs is not quite related to *resolution* over prophecy variables.

construction from the goal/root) of a proof tree on CHCs, where each CHC is regarded as a deduction rule. In the SLDC resolution algorithm, intermediate states can contain some *syntactic logic variables*, which can model the *prophecy variables* of unique references in Rust. Because these logic variables are universally quantified semantically, each intermediate state of SLDC resolution virtually handles a *set of possible worlds* that is parametrized over values assigned to the variables. Their proof is rather fragile, because the formalized execution of Rust programs and the SLDC resolution algorithm are carefully *designed* to form bisimulation and SLDC resolution depends heavily on *syntactic* structures of logic formulas. Also, their proof omits detailed discussion about locality of the effects of various Rust operations, without formal machinery like separation logic. Our approach to prophecies for RustHornBelt can be understood as a *semantic* reformulation of their syntactic approach built upon the *mechanized* separation logic Iris.

Chapter 4

A Low-level Foundation for Verification

Before building the verification platform specifically for Rust, we introduce a simple lambda calculus for imperative programming and build a low-level program logic on the calculus on top of the separation logic Iris. The calculus supports heap manipulation, concurrency and non-determinism. Although our program logic serves as a low-level foundation for verifying Rust programs later in Chapter 5 and Chapter 6, the logic is designed for a general purpose.

Although the construction of the program logic in Iris largely follows the standard technique, we present a new technique for spending an *unbounded number of logical steps* for each physical step in the *finite*-step-indexed logic Iris, which is technically important for our work.

In § 4.1, we present our lambda calculus for imperative programming. In § 4.2, we present our program logic for the calculus, with the new technique for spending an unbounded number of logical steps at a time. In § 4.3 we discuss some related work.

4.1 A Lambda Calculus for Imperative Programming

We introduce a simple *lambda calculus* for imperative programming, which models the *unsafe* part of Rust. To ease verification, we use this *substitution-based* calculus, just like RustBelt (Jung et al., 2018a) did (see also (Jung, 2020, §7.3)).[1] Advanced operations can be represented as *syntax sugar* in this calculus. The so-called *safe part* of Rust can be understood as some subset of this calculus.

Syntax An *address* $\mathfrak{l}$[2] is an object of the record type

$$\mathsf{Addr} \;:=\; (\mathsf{block}\colon \mathbb{Z},\ \mathsf{cell}\colon \mathbb{Z}),$$

consisting of (block) the id of the *memory block* and (cell) the index of a *memory cell*. (These notions get clearer when we formalize the heap memory later in the paragraph Operational Semantics.) We write $\mathsf{addr}(n, i)$ for the address whose block field equals n and whose cell field equals i. For $\mathfrak{l}\colon \mathsf{Addr}$ and $n\colon \mathbb{Z}$, we define $\mathfrak{l} + n$ as follows.

$$\mathfrak{l} + n \;:=\; \mathsf{addr}(\mathfrak{l}.\mathsf{block},\ \mathfrak{l}.\mathsf{cell} + n)$$

A *cell value* $\mathfrak{a}, \mathfrak{b}, \mathfrak{c}\colon \mathsf{CellVal}$ is a value stored in a single memory cell. An *expression* $e\colon \mathsf{Expr}$ is a structural program that may perform some imperative operations. Cell values and expressions are mutually inductively defined as follows.

[1] To be precise, this deviates from the behavior the real-world systems (e.g., a function definition is stored in the text segment of the program memory instead of carried around as a value). Still, we believe that this is a reasonable simplification.

[2] The letter 'l' stands for 'location'.

$$\text{(cell value)}\ \mathfrak{a}, \mathfrak{b}, \mathfrak{c} : \mathsf{CellVal} \ ::= \ n \ \text{(integer; } n\colon \mathbb{Z}) \mid \mathfrak{l} \ \text{(address)} \mid \mathfrak{f} \ \text{(function)} \mid \lightning \ \text{(invalid)}$$

$$\text{(function value)}\ \mathfrak{f} : \mathsf{FnVal} \ ::= \ \mathsf{fn}\, f(\vec{a})\{e\}$$

$$\begin{aligned}\text{(expression)}\ e : \mathsf{Expr} \ ::= \ & a \ \text{(variable)} \mid \mathfrak{a} \ \text{(value)} \\ \mid \ & e(\vec{e'}) \ \text{(function call)} \mid \mathsf{alloc}\, e \ \text{(allocate)} \mid \mathsf{free}\, e \ \text{(free)} \\ \mid \ & *e \ \text{(load)} \mid e \leftarrow e' \ \text{(store)} \mid e.e' \ \text{(address shift)} \\ \mid \ & \mathsf{case}\ e\ \mathsf{of}\ \{ 0 \to e', 1 \to e'' \} \ \text{(conditional branching)} \\ \mid \ & e \parallel e' \ \text{(concurrent execution)} \\ \mid \ & e \ \mathit{iop}\ e' \ \text{(integer operation)} \mid e \ \mathit{irel}\ e' \ \text{(integer relation)} \\ \mid \ & \mathsf{ndint} \ \text{(non-deterministic integer)}\end{aligned}$$

$$\text{(program variable)}\ a, b, c, f : \mathsf{Var}$$

$$\mathit{iop} \ ::= \ + \mid - \mid \times \mid \cdots \qquad\qquad \mathit{irel} \ ::= \ \leq \mid < \mid = \mid \cdots$$

A cell value $\mathfrak{a}, \mathfrak{b}, \mathfrak{c}$ can be an integer n, an address $\mathfrak{l}$, a function value $\mathfrak{f}$, or the invalid value $\lightning$. We use variables $\bar{\mathfrak{a}}, \bar{\mathfrak{b}}, \bar{\mathfrak{c}}$ for a list of cell values. The invalid value cannot be used in a meaningful way. A function value $\mathfrak{f} = \mathsf{fn}\, f(\vec{a})\{e\}$ consists of (i) the variable f binding the function itself, which is used for recursion in e, (ii) the parameter variables $\vec{a}$, and (iii) the function body e.

We describe here the meaning of the expression primitives. By $e(\vec{e'})$, we call a function e with arguments $\vec{e'}$. By $\mathsf{alloc}\, e$, we allocate a new memory block of the size e. By $\mathsf{free}\, e$, we free the memory block starting at the address e. By $*e$, we load the value of the memory cell at the address e. By $e \leftarrow e'$, we update the value of the memory cell at the address e into the value e'. By $e.e'$, we get the address that is ahead of the address e by the integer e'. By $\mathsf{case}\ e\ \mathsf{of}\ \{ 0 \to e', 1 \to e'' \}$, we conditionally branch by the value of e, executing e' if the value is 0 and e'' when if 1. By $e \parallel e'$, we execute the expressions e and e' concurrently; the expression finally returns the cell value of e after both e and e' terminates. We calculate a binary integer operation (that inputs and outputs an integer) by $e\ \mathit{iop}\ e'$ and a binary integer relation by $e\ \mathit{irel}\ e'$. By ndint, we get a non-deterministic integer value.

We write $\mathit{irel}_{\mathsf{bool}}$ for the function of the type $\mathbb{Z} \to \mathbb{Z} \to \mathbb{B}$ that corresponds to the binary integer relation irel.

We use the following shorthand for cell values.

$$\mathsf{tt} \ := \ 1 \qquad\qquad \mathsf{ff} \ := \ 0 \qquad\qquad \mathsf{fn}(\vec{a})\{e\} \ := \ \mathsf{fn}\,\_(\vec{a})\{e\}$$

We use the underscore $\_$ for a special program variable from which we never take out the value (i.e., $\_$ is never used as a variable expression).

We use the following shorthand for expressions.

$$\mathsf{let}\ a = e\ \mathsf{in}\ e' \ := \ (\mathsf{fn}(a)\{e'\})(e) \qquad\qquad e;\, e' \ := \ \mathsf{let}\ \_ = e\ \mathsf{in}\ e'$$

$$\mathsf{if}\ e\ \{e'\}\ \mathsf{else}\ \{e''\} \ := \ \mathsf{case}\ e\ \mathsf{of}\ \{ 0 \to e'', 1 \to e' \} \qquad \mathsf{if}\ e\ \{e'\} \ := \ \mathsf{if}\ e\ \{e'\}\ \mathsf{else}\ \{\lightning\}$$

$$\begin{aligned}&\mathsf{for}\ a \leftarrow e\,..\,e'\ \{e''\} \ := \\ &\quad \big(\mathsf{fn}\, \mathit{for}(b, c, f)\big\{\, \mathsf{if}\ b \geq c\ \{\lightning\}\ \mathsf{else}\ \{f(b);\, \mathit{for}(b+1, c, f)\}\,\big\}\big)(e, e', \mathsf{fn}(a)\{e''\})\end{aligned}$$

$$e \leftarrow^{*}_{e''} e' \ := \ \big(\mathsf{fn}(a, b, c)\big\{\, \mathsf{for}\ i \leftarrow 0\,..\,c\{a.i \leftarrow *(b.i)\}\,\big\}\big)(e, e', e'')$$

For simplicity, let binding $\mathsf{let}\ a = e\ \mathsf{in}\ e'$ is defined using a function call. The for loop $\mathsf{for}\ a \leftarrow e\,..\,e'\ \{e''\}$ iteratively performs e'' setting a to each integer value from the integer

e to the integer e' exclusive. The sequential store $e \leftarrow^{*}_{e''} e'$ copies e'' consecutive cell values from the source address e' to the target address e.

Operational Semantics A *heap* (or heap memory) H is an object of the type

$$\mathsf{Heap} \;:=\; \mathbb{Z} \xrightarrow{\text{fin}} \mathsf{List}\,\mathsf{CellVal},$$

a finite map from *memory block* ids to a list of cell values, representing the sequence of *memory cells* for the memory block.

For a heap H, we define the address domain $\mathsf{Dom}\,H \colon \mathbb{P}_{\text{fin}}\,\mathsf{Addr}$ as follows.

$$\mathsf{Dom}\,H \;:=\; \{\; \mathfrak{I} \colon \mathsf{Addr} \mid \mathfrak{I}.\mathsf{block} \in \mathsf{dom}\,H \;\wedge\; 0 \le \mathfrak{I}.\mathsf{cell} < \mathsf{len}\,H[\mathfrak{I}.\mathsf{block}] \;\}$$

Here, $\mathsf{dom}\,H \colon \mathbb{P}_{\text{fin}}\,\mathbb{N}$ denotes the domain of H as a finite map over natural numbers. For an address $\mathfrak{I}$ in $\mathsf{Dom}\,H$, we introduce the following shorthand.

$$H[\mathfrak{I}] \;:=\; H[\mathfrak{I}.\mathsf{block}][\mathfrak{I}.\mathsf{cell}] \qquad H\{\mathfrak{I} \leftarrow \mathfrak{a}\} \;:=\; H\big\{\mathfrak{I}.\mathsf{block} \leftarrow H[\mathfrak{I}.\mathsf{block}]\{\mathfrak{I}.\mathsf{cell} \leftarrow \mathfrak{a}\}\big\}$$

Here, $H[\mathfrak{I}]$ denotes the cell value at the address $\mathfrak{I}$ and $H\{\mathfrak{I} \leftarrow \mathfrak{a}\}$ denotes the heap with the value of the memory cell at $\mathfrak{I}$ updated into $\mathfrak{a}$.

Also, for a heap H and a memory block id n, $H - n$ denotes the heap obtained from H by erasing the memory block at n.

The *evaluation context* $K \colon \mathsf{EvCtx}$ is defined as follows.

$$\begin{aligned} K : \mathsf{EvCtx} \;::=\; & \bullet \mid e(\vec{\mathfrak{a}}, K, \vec{e}) \mid K(\vec{\mathfrak{a}}) \mid \mathsf{alloc}\,K \mid \mathsf{free}\,K \\ & \mid *K \mid e \leftarrow K \mid K \leftarrow \mathfrak{a} \mid K.e \mid \mathfrak{a}.K \mid \mathsf{case}\,K\,\mathsf{of}\,\{\,0 \to e,\, 1 \to e'\,\} \\ & \mid K \parallel e \mid e \parallel K \mid K \mathbin{\mathit{iop}} e \mid \mathfrak{a} \mathbin{\mathit{iop}} K \mid K \mathbin{\mathit{irel}} e \mid \mathfrak{a} \mathbin{\mathit{irel}} K \end{aligned}$$

For the concurrent execution $e \parallel e'$, we can choose whether to reduce e or e' for each reduction step. In the store expression $e \leftarrow e'$, the source expression e' is evaluated earlier than the target expression e.

Now the reduction relation $(e, H) \to (e', H')$ is defined inductively by the following rules.[3]

$$\frac{(e,\, H) \;\to\; (e',\, H)}{(K[e],\, H) \;\to\; (K[e'],\, H)} \qquad \frac{\mathfrak{f} \;=\; \mathsf{fn}\,f(a_0, \ldots, a_{n-1})\{e\}}{\big(\mathfrak{f}(\mathfrak{a}_0, \ldots, \mathfrak{a}_{n-1}),\, H\big) \;\to\; \big(e[\mathfrak{f}/f, \overrightarrow{\mathfrak{a}/a}],\, H\big)}$$

$$\frac{m \notin \mathsf{dom}\,H \qquad H' = H\{m \leftarrow \tilde{\mathfrak{a}}\}}{\big(\mathsf{alloc}\,n,\, H\big) \;\to\; \big(\mathsf{addr}(m, 0),\, H'\big)} \qquad \frac{\mathfrak{I}.\mathsf{cell} = 0 \qquad H' = H - \mathfrak{I}.\mathsf{block}}{(\mathsf{free}\,\mathfrak{I},\, H) \;\to\; (\lightning,\, H')}$$

$$\frac{\mathfrak{I} \in \mathsf{Dom}\,H}{(*\mathfrak{I},\, H) \;\to\; (H[\mathfrak{I}],\, H)} \qquad \frac{\mathfrak{I} \in \mathsf{Dom}\,H}{(\mathfrak{I} \leftarrow \mathfrak{a},\, H) \;\to\; (\lightning,\, H\{\mathfrak{I} \leftarrow \mathfrak{a}\})} \qquad \frac{\mathfrak{I}' = \mathfrak{I} + n}{(\mathfrak{I}.n,\, H) \;\to\; (\mathfrak{I}',\, H)}$$

$$\frac{i = 0 \;\vee\; i = 1}{(\mathsf{case}\;i\;\mathsf{of}\;\{\,0 \to e_0,\, 1 \to e_1\,\},\, H) \;\to\; (e_i,\, H)} \qquad (\mathfrak{a} \parallel \mathfrak{b},\, H) \;\to\; (\mathfrak{a},\, H)$$

$$\frac{l = m \mathbin{\mathit{iop}} n}{(m \mathbin{\mathit{iop}} n,\, H) \;\to\; (l,\, H)} \qquad \frac{bl = m \mathbin{\mathit{irel}_{\mathsf{bool}}} n}{(m \mathbin{\mathit{irel}} n,\, H) \;\to\; (bl,\, H)} \qquad (\mathsf{ndint},\, H) \;\to\; (n,\, H)$$

Note that the reduction rules expect some *preconditions* on the form of the expression and the heap. When the preconditions are not satisfied, the reduction gets *stuck*. Also note that we cannot perform any reduction when the expression e is simply a cell value.

[3] In the rule on *irel*, the Boolean value *bl* is interpreted as 1 or 0 as a cell value.

For convenience, we introduce the following set $\mathsf{Next}(e, H)$, which is the set of configurations to which the configuration (e, H) can reduce.

$$\mathsf{Next}(e, H) \;:=\; \{\, (e', H') \mid (e, H) \to (e', H') \,\}$$

Also, we introduce the following pure predicate $\mathsf{red}(e, H)$, which means that the configuration (e, H) is reducible.

$$\mathsf{red}(e, H) \;:=\; \mathsf{Next}(e, H) \neq \emptyset$$

4.2 Verification in Iris

We model the weakest precondition and the Hoare triple for the imperative lambda calculus presented in the previous section.

We introduce a simple but new mechanism that enables us to *spend unboundedly many logical steps for each physical step* in Iris, in a sense. It is an important technique in Iris, because Iris employs *finite* step indexing; in Iris, the proposition $\exists\, n.\ \Rrightarrow^n_{\mathcal{E}} P$ is equivalent to True for any P, because at each step index i we can set n to i, which makes all the 'visible part' at the step index True. More specifically, our machinery enables spending $\mathsf{step}(n) + 1$ logical steps for the n-th physical step, for some fixed monotone function step, e.g., $\lambda n.n$. This comes from the observation that (i) typically the number of logical steps we need to spend at once is proportional to the *depth* or complexity of the manipulated object and that (ii) the depth of the object lower-bounds the number of program steps we have spent so far. In order to manage how many physical steps have passed, we introduce a new *time receipt RA*, extending the idea of Mével et al. (2019). This machinery is used later in RustHornBelt to temporarily take out prophecy tokens deep inside an object for dependent resolution (see SemTy-Own-ProphToken and SWkn-UnqRef-Release).

Heap RA In order to manage information about the heap, we introduce the following *heap RA* HEAP.

$$|\textsc{heap}| \;:=\; |\textsc{ex}(\mathsf{Heap})| \times |\mathbb{Z} \to \textsc{ex}(\mathbb{N})| \times |\mathsf{Addr} \to \textsc{fracown}(\mathsf{CellVal})|$$

$$(a, f, h) \cdot (b, g, h') \;:=\; (a \cdot b,\ f \cdot g,\ h \cdot h') \qquad \varepsilon \;:=\; (\varepsilon, \varepsilon, \varepsilon) \qquad |(a, f, h)| \;:=\; (\varepsilon, \varepsilon, \varepsilon)$$

$$\begin{aligned} \checkmark\,(a, f, h) \;:=\; & \checkmark a \;\wedge\; \checkmark f \;\wedge\; \checkmark h \;\wedge\; \forall H \text{ s.t. } \mathsf{ex}\, H = a. \\ & \big(\forall m, n \text{ s.t. } \mathsf{ex}\, n = f\, m.\ m \in \mathsf{dom}\, H \;\wedge\; \mathsf{len}\, H[m] = n\big) \;\wedge \\ & \big(\forall \mathfrak{l}, \mathfrak{a}, q \text{ s.t. } \mathsf{fown}_q\, \mathfrak{a} = h\, \mathfrak{l}.\ \mathfrak{l} \in \mathsf{Dom}\, H \;\wedge\; H[\mathfrak{l}] = \mathfrak{a}\big) \end{aligned}$$

We register the heap RA HEAP to the global camera. Using some ghost name γ_{heap}, we introduce the following *exclusive heap token* $\mathsf{Ex}(H)$, *free-right token* $\mathsf{Free}_n(\mathfrak{l})$ and *maps-to token* $\mathfrak{l} \overset{q}{\mapsto} \mathfrak{a}$.

$$\mathsf{Ex}(H) \;:=\; \boxed{(\mathsf{ex}\, H, \varepsilon, \varepsilon)}^{\gamma_{\mathsf{heap}}}_{\textsc{heap}} \qquad \mathsf{Free}_n(\mathfrak{l}) \;:=\; \mathfrak{l}.\mathsf{cell} = 0 \;*\; \boxed{(\varepsilon, [\mathfrak{l}.\mathsf{block} \leftarrow n], \varepsilon)}^{\gamma_{\mathsf{heap}}}_{\textsc{heap}}$$

$$\mathfrak{l} \overset{q}{\mapsto} \mathfrak{a} \;:=\; \boxed{(\varepsilon, \varepsilon, [\mathfrak{l} \leftarrow \mathsf{fown}_q\, \mathfrak{a}])}^{\gamma_{\mathsf{heap}}}_{\textsc{heap}}$$

An exclusive heap token $\mathsf{Ex}(H)$ registers the heap H. A free-right token $\mathsf{Free}_n(\mathfrak{l})$ owns the right to free the memory block starting at $\mathfrak{l}$ of the size n. A points-to token $\mathfrak{l} \overset{q}{\mapsto} \mathfrak{a}$ witnesses with the fraction q that the value of the memory cell at $\mathfrak{l}$ is $\mathfrak{a}$. We also introduce the following sequential points-to token $\mathfrak{l} \overset{q}{\mapsto} \bar{\mathfrak{a}}$, which witnesses with the fraction q that the cell values of the $\mathsf{len}\, \bar{\mathfrak{a}}$ consecutive memory cells starting at $\mathfrak{l}$ are $\bar{\mathfrak{a}}$.

$$\mathfrak{l} \overset{q}{\mapsto} \bar{\mathfrak{a}} \;:=\; *_{i < \mathsf{len}\, \bar{\mathfrak{a}}}\ \mathfrak{l} + i \overset{q}{\mapsto} \bar{\mathfrak{a}}[i]$$

We also use the following shorthand.

$$\mathfrak{l} \overset{q}{\mapsto} \_ \;:=\; \exists\, \mathfrak{a}.\ \mathfrak{l} \overset{q}{\mapsto} \mathfrak{a} \qquad\qquad \mathfrak{l} \overset{q}{\mapsto} \_^n \;:=\; \exists\, \bar{\mathfrak{a}} \text{ s.t. } \mathsf{len}\,\bar{\mathfrak{a}} = n.\ \mathfrak{l} \overset{q}{\mapsto} \bar{\mathfrak{a}}$$

The ghost name γ_{heap} is fixed when we introduce the exclusive heap token $\mathsf{Ex}(H)$ using the following rule.

ExHeap-Intro
$$\mathsf{True} \;\Rrightarrow_{\varnothing}\; \exists\, \gamma_{\mathsf{heap}}.\ \mathsf{Ex}(H)$$

The following properties hold.

$$\mathsf{timeless}\big(\mathsf{Ex}(H)\big) \qquad \mathsf{timeless}\big(\mathsf{Free}_n(\mathfrak{l})\big) \qquad \mathsf{timeless}\big(\mathfrak{l} \overset{q}{\mapsto} \mathfrak{a}\big)$$

MapsTo-Frac
$$\mathfrak{l} \overset{q+q'}{\mapsto} \mathfrak{a} \;\Leftrightarrow\; \mathfrak{l} \overset{q}{\mapsto} \mathfrak{a} \;*\; \mathfrak{l} \overset{q'}{\mapsto} \mathfrak{a}$$

MapsTo-Agree
$$\mathfrak{l} \overset{q}{\mapsto} \mathfrak{a} \;*\; \mathfrak{l} \overset{q'}{\mapsto} \mathfrak{a}' \;\Rightarrow\; \mathfrak{a} = \mathfrak{a}'$$

Heap-Alloc
$$\frac{m \notin \mathsf{dom}\, H \qquad \mathsf{len}\,\bar{\mathfrak{a}} = n \qquad \mathfrak{l} = \mathsf{addr}(m, 0)}{\mathsf{Ex}(H) \;\Rrightarrow_{\varnothing}\; \mathsf{Free}_n(\mathfrak{l}) \;*\; \mathfrak{l} \overset{1}{\mapsto} \bar{\mathfrak{a}} \;*\; \mathsf{Ex}(H\{m \leftarrow \bar{\mathfrak{a}}\})}$$

Heap-Free
$$\mathsf{Free}_n(\mathfrak{l}) \;*\; \mathfrak{l} \overset{1}{\mapsto} \_^n \;*\; \mathsf{Ex}(H) \;\Rrightarrow_{\varnothing}\; \mathsf{Ex}(H - \mathfrak{l}.\mathsf{block})$$

Heap-Load
$$\mathfrak{l} \overset{q}{\mapsto} \mathfrak{a} \;*\; \mathsf{Ex}(H) \;\Rightarrow\; \mathfrak{l} \in \mathsf{Dom}\, H \;\wedge\; H[\mathfrak{l}] = \mathfrak{a}$$

Heap-Store
$$\mathfrak{l} \overset{1}{\mapsto} \mathfrak{a} \;*\; \mathsf{Ex}(H) \;\Rrightarrow_{\varnothing}\; \mathfrak{l} \overset{1}{\mapsto} \mathfrak{b} \;*\; \mathsf{Ex}(H\{\mathfrak{l} \leftarrow \mathfrak{b}\})$$

Points-to tokens can be split and merged according to the fraction (MapsTo-Frac). Two points-to tokens on the same address should agree on the cell value (MapsTo-Agree). Heap-Alloc says that, when we allocate in the heap a new memory block of the length n and the cell values $\bar{\mathfrak{a}}$, letting $\mathfrak{l}$ be the head address of the memory block, we get a free-right token $\mathsf{Free}_n(\mathfrak{l})$ and a full sequential points-to token $\mathfrak{l} \overset{1}{\mapsto} \bar{\mathfrak{a}}$. Heap-Free says that, by consuming a free-right token on an address $\mathfrak{l}$ and the size n and a full points-to token on the n consecutive memory cells starting at $\mathfrak{l}$, we can free the memory block of $\mathfrak{l}$. Heap-Load says that, when we have a fractional points-to token $\mathfrak{l} \overset{q}{\mapsto} \mathfrak{a}$, we know that there exists a memory cell at $\mathfrak{l}$ storing $\mathfrak{a}$. Heap-Store says that, by consuming a full points-to token $\mathfrak{l} \overset{1}{\mapsto} \mathfrak{a}$, we can update the value of the memory cell at $\mathfrak{l}$ into any value $\mathfrak{b}$ and obtain a new full points-to token $\mathfrak{l} \overset{1}{\mapsto} \mathfrak{b}$.

Time Receipt RA In order to manage information about *how many physical steps have passed*, we introduce the following *time receipt RA* TIME.

$$|\textsc{time}| \;:=\; |\textsc{ex}(\mathbb{N})| \times \mathbb{N} \times \mathbb{N}$$

$$(a, m, n) \cdot (b, m', n') \;:=\; (a \cdot b,\ m + m',\ \max\{n, n'\}) \qquad \varepsilon \;:=\; (\varepsilon, 0, 0)$$

$$|(a, m, n)| \;:=\; (\varepsilon, 0, n) \qquad \checkmark(a, m, n) \;:=\; \checkmark a \;\wedge\; \forall\, l \text{ s.t. } \mathsf{ex}\, l = a.\ l \geq m + n$$

We register the time receipt RA TIME to the global camera. Using some ghost name γ_{time}, we introduce the following *exclusive time receipt* $\blacklozenge\, n$, *cumulative time receipt* $\blacklozenge\!\!\blacklozenge\, n$, and *persistent time receipt* $\lozenge\, n$.[4]

$$\blacklozenge\, n \;:=\; \boxed{(\mathsf{ex}\, n, 0, 0)}^{\gamma_{\mathsf{time}}}_{\textsc{time}} \qquad \blacklozenge\!\!\blacklozenge\, n \;:=\; \boxed{(\varepsilon, n, 0)}^{\gamma_{\mathsf{time}}}_{\textsc{time}} \qquad \lozenge\, n \;:=\; \boxed{(\varepsilon, 0, n)}^{\gamma_{\mathsf{time}}}_{\textsc{time}}$$

[4] The symbols represent sandglasses.

An exclusive time receipt $⧗ n$ registers the number n of physical steps that have passed. When we have a cumulative time receipt $⧗ m$ and a persistent time receipt $⧖ n$, we witness that at least $m + n$ physical steps have passed. Cumulative time receipts follow the addition law $⧗ m * ⧗ n \Leftrightarrow ⧗(m + n)$ but are not persistent. Persistent time receipts are persistent but do not follow such an addition law. We fix the ghost name γ_{time} when we introduce the exclusive time receipt by the following rule (typically n is set to 0).

$$\frac{}{\mathsf{True} \Rrightarrow_{\varnothing} \exists \gamma_{\mathsf{time}}.\ ⧗ n}\ \textsc{ExTime-Intro}$$

The following properties hold.

$$\mathsf{timeless}(⧗ n) \qquad \mathsf{timeless}(⧖ n) \qquad \mathsf{persistent}(⧖ n) \qquad \mathsf{timeless}(⧖ n)$$

$$⧗ n \Rrightarrow_{\varnothing} ⧗(n+1) * ⧗ 1\ \textsc{ExTime-Increment-CumuTime} \qquad ⧗ m * ⧗ n \Leftrightarrow ⧗(m+n)\ \textsc{CumuTime-Add} \qquad ⧖ 0\ \textsc{PersTime-0}$$

$$\frac{⧖ n}{⧗ m \Rrightarrow_{\varnothing} ⧖(m+n)}\ \textsc{CumuTime-Swell-PersTime} \qquad \frac{⧖ m}{⧗ n \Rightarrow n \geq m}\ \textsc{PersTime-Bound-ExTime}$$

ExTime-Increment-CumuTime says that, when we increment the number of the exclusive time receipt, we get a cumulative time receipt of one step $⧗ 1$. Cumulative time receipts follow the addition law CumuTime-Add. A persistent receipt of zero step $⧖ 0$ can be freely obtained (PersTime-0). The rule CumuTime-Swell-PersTime is interesting and important; by consuming a cumulative time receipt $⧗ m$, we can *swell* a persistent time receipt $⧖ n$ into $⧖(m + n)$.[5] PersTime-Bound-ExTime says that the number m of a persistent time receipt $⧖ m$ lower-bounds the number n of the exclusive time receipt $⧗ n$.

Weakest Precondition and Hoare Triple We fix some *monotone* function over natural numbers step: $\mathbb{N} \to \mathbb{N}$. For RustHornBelt, we particularly set $\mathsf{step}(n) = 2n$.

Now we define the *weakest precondition* $\mathsf{wp}_{\mathcal{E}}\, e\, \{\Phi\}$, where Φ is the postcondition over the returned cell value (typed CellVal → IProp).

$$\begin{aligned}\mathsf{wp}_{\mathcal{E}}\, e\, \{\Phi\} \ :=\ & (\exists\, \mathfrak{a}\ \text{s.t.}\ \mathfrak{a} = e.\ \Rrightarrow_{\mathcal{E}} \Phi\, \mathfrak{a}) \ \vee\ \big((\forall\, \mathfrak{a}.\ \mathfrak{a} \neq e) \ \wedge \\ & \forall H, n.\ \mathsf{Ex}(H) * ⧗ n \ \Rrightarrow\!\!\!*^{\mathsf{step}(n)+1}_{\mathcal{E}}\ \mathsf{red}(e, H) \ * \\ & \quad \forall (e', H') \in \mathsf{Next}(e, H).\ \Rrightarrow_{\mathcal{E}} \big(\mathsf{Ex}(H') * ⧗(n{+}1) * \mathsf{wp}_{\mathcal{E}}\, e'\, \{\Phi\} \big) \big)\end{aligned}$$

This recursive equation over $\mathsf{wp}_{\mathcal{E}} \cdots \{\Phi\}$ has the unique solution due to the *contractiveness* introduced by the step-taking view shift $\Rrightarrow\!\!\!*^{\mathsf{step}(n)+1}_{\mathcal{E}}$ (by Banach's fixed point theorem Theorem 2.1).

This definition of the weakest precondition $\mathsf{wp}_{\mathcal{E}}\, e\, \{\Phi\}$ can be understood as follows. When the given expression e is the form of some value $\mathfrak{a}$, the postcondition $\Phi\, \mathfrak{a}$ should be satisfied after a fancy update ($\Rrightarrow_{\mathcal{E}}$). When e is not a value, and when the heap is H and it is the n-th physical step, having the tokens $\mathsf{Ex}(H)$ and $⧗ n$, after fancy update *with* $\mathsf{step}(n)+1$ *logical steps* ($\Rrightarrow\!\!\!*^{\mathsf{step}(n)+1}_{\mathcal{E}}$), we know that the configuration (e, H) is not stuck, and for any next possible configuration (e', H'), along with the tokens $\mathsf{Ex}(H')$ and $⧗(n{+}1)$ we get the weakest precondition $\mathsf{wp}_{\mathcal{E}}\, e'\, \{\Phi\}$, which gives us the postcondition Φ after executing e'.

[5] This use of a cumulative time receipt was conceived by Jacques-Henri Jourdan.

We introduce the following *super fancy update modality* $\Rrightarrow^{\#}_{\mathcal{E}} P$.

$$\Rrightarrow^{\#}_{\mathcal{E}} P \;:=\; \Rrightarrow_{\mathcal{E}} \left(\exists\, n.\; \boxtimes n \ast \Rrightarrow^{\mathsf{step}(n)}_{\mathcal{E}} P\right)$$

It means that, after one logical step, for some n such that we know that n physical steps have passed by $\boxtimes n$, we can get P after $\mathsf{step}(n)$ logical steps, under the mask $\mathcal{E}$. Note that $\exists\, n.\; \boxtimes n \ast \Rrightarrow^{\mathsf{step}(n)}_{\mathcal{E}} P$ is *not* equal to True, unlike $\exists\, n.\; \Rrightarrow^{\mathsf{step}(n)}_{\mathcal{E}} P$, because the persistent time receipt $\boxtimes n$ prevents us from taking unboundedly large n as the step index grows.

We can construct a proposition under a super fancy update modality using the following properties.

SpFUpd-Def
$$\Rrightarrow_{\mathcal{E}} \left(\exists\, n.\; \boxtimes n \ast \Rrightarrow^{\mathsf{step}(n)}_{\mathcal{E}} P\right) \;\Leftrightarrow\; \Rrightarrow^{\#}_{\mathcal{E}} P$$

SpFUpd-Zero
$$\Rrightarrow^{\mathsf{step}(0)+1}_{\mathcal{E}} P \;\Rightarrow\; \Rrightarrow^{\#}_{\mathcal{E}} P$$

SpFUpd-Merge
$$\Rrightarrow^{\#}_{\mathcal{E}} P \ast \Rrightarrow^{\#}_{\mathcal{E}} Q \;\Rightarrow\; \Rrightarrow^{\#}_{\mathcal{E}} (P \ast Q)$$

We also introduce the following shorthand.

$$P \mathrel{\Rrightarrow\!\!\!\ast^{\#}_{\mathcal{E}}} Q \;:=\; P \mathrel{-\!\!\ast} \Rrightarrow^{\#}_{\mathcal{E}} Q \qquad\qquad P \Rrightarrow^{\#}_{\mathcal{E}} Q \;:=\; \Box\left(P \mathrel{\Rrightarrow\!\!\!\ast^{\#}_{\mathcal{E}}} Q\right)$$

We can strip off the super fancy update modality using the following property.

ExTime-SpFUpd-CumuTime
$$\Rrightarrow^{\#}_{\mathcal{E}} P \ast \blacktriangleright n \;\Rrightarrow^{\mathsf{step}(n)+1}_{\mathcal{E}}\; P \ast \boxtimes 1 \ast \blacktriangleright(n+1)$$

It means that, spending $\mathsf{step}(n)+1$ logical steps under the mask $\mathcal{E}$, if we update $\blacktriangleright n$ into $\blacktriangleright(n+1)$, we can strip off one super fancy update modality $\Rrightarrow^{\#}_{\mathcal{E}}$ and get a cumulative time receipt $\boxtimes 1$.

Proof. Assume that we have $\Rrightarrow^{\#}_{\mathcal{E}} P$ and $\blacktriangleright n$. We can decompose $\Rrightarrow^{\#}_{\mathcal{E}} P$ into $\boxtimes m$ and $\Rrightarrow^{\mathsf{step}(m)+1}_{\mathcal{E}} P$ for some m. By PersTime-Bound-ExTime we know $n \geq m$. Since $\mathsf{step}(n) + 1 \geq \mathsf{step}(m) + 1$ holds by monotonicity of step, we can update $\Rrightarrow^{\mathsf{step}(m)+1}_{\mathcal{E}} P$ into P in $\mathsf{step}(n)+1$ logical steps. Also, by ExTime-Increment-CumuTime, we can update $\blacktriangleright n$ into $\boxtimes 1$ and $\blacktriangleright(n+1)$. □

By ExTime-SpFUpd-CumuTime, we can prove the following basic lemma for discussing the weakest precondition over an operation that takes one reduction step.

Wp-Step
$$\frac{\mathsf{Ex}(H) \ast P \;\Rrightarrow_{\mathcal{E}}\; \mathsf{red}(e,H) \ast \forall\,(e',H') \in \mathsf{Next}(e,H).\; \models_{\mathcal{E}} \left(\mathsf{Ex}(H') \ast \mathsf{wp}_{\mathcal{E}}\, e'\,\{\Phi\}\right)}{\Rrightarrow^{\#}_{\mathcal{E}} P \;\Rightarrow\; \mathsf{wp}_{\mathcal{E}}\, e\,\{\lambda\mathfrak{a}.\; \boxtimes 1 \ast \Phi\,\mathfrak{a}\}}$$

In one physical step, we can strip off the super fancy later modality and gain a cumulative time receipt $\boxtimes 1$.

We also have the following lemmas for the weakest precondition on an evaluation context and a value expression.

Wp-EvCtx
$$\mathsf{wp}_{\mathcal{E}}\, e\,\{\lambda\mathfrak{a}.\; \mathsf{wp}_{\mathcal{E}}\, K[\mathfrak{a}]\,\{\Phi\}\} \;\Rightarrow\; \mathsf{wp}_{\mathcal{E}}\, K[e]\,\{\Phi\}$$

Wp-Val
$$\Phi\,\mathfrak{a} \;\Leftrightarrow\; \mathsf{wp}_{\mathcal{E}}\, \mathfrak{a}\,\{\Phi\}$$

We can modify the postcondition part of the weakest precondition using a view shift.

Wp-VShift
$$(\forall\,\mathfrak{a}.\; \Phi\,\mathfrak{a} \mathrel{\Rrightarrow\!\!\!\ast_{\mathcal{E}}} \Psi\,\mathfrak{a}) \ast \mathsf{wp}_{\mathcal{E}}\, e\,\{\Phi\} \;\Rightarrow\; \mathsf{wp}_{\mathcal{E}}\, e\,\{\Psi\}$$

We can prove this rule using Löb induction (Löb).

The weakest precondition satisfies the following *adequacy theorem.*

Theorem 4.1 (Adequacy of the Weakest Precondition). *Let ϕ be a pure predicate on a cell value. If*

$$\Rrightarrow_{\mathcal{E}} \exists \gamma_{\mathsf{heap}}, \gamma_{\mathsf{time}}.\ \mathsf{Ex}(H_0) * \blacklozenge k * \mathsf{wp}_{\mathcal{E}}\, e_0\, \{\phi\}$$

is a tautology, for any reduction sequence $(e_0, H_0) \to (e_1, H_1) \to \cdots \to (e_n, H_n)$ such that $\neg\mathsf{red}(e_n, H_n)$ holds, e_n is a cell value satisfying ϕ.

Here, the tokens $\mathsf{Ex}(H_0)$ and $\blacklozenge k$ depend on the ghost names γ_{heap} and γ_{time}. We can use the fancy update $\Rrightarrow_{\mathcal{E}}$ and the existential quantification $\exists \gamma_{\mathsf{heap}}, \gamma_{\mathsf{time}}.$ for introducing the tokens $\mathsf{Ex}(H_0)$ and $\blacklozenge k$ by the rules ExHeap-Intro and ExTime-Intro.

Proof. By the definition of the weakest precondition, the following holds for each reduction step $(e_i, H_i) \to (e_{i+1}, H_{i+1})$.

$$\mathsf{Ex}(H_i) * \blacklozenge(k+i) * \mathsf{wp}_{\mathcal{E}}\, e_i\, \{\phi\} \Rrightarrow^{\mathsf{step}(k+i)+1}_{\mathcal{E}} \mathsf{Ex}(H_{i+1}) * \blacklozenge(k+i+1) * \mathsf{wp}_{\mathcal{E}}\, e_{i+1}\, \{\phi\}$$

Therefore, using the assumption tautology $\mathsf{Ex}(H_0) * \blacklozenge k * \mathsf{wp}_{\mathcal{E}}\, e_0\, \{\phi\}$, we get the following tautology.

$$\Rrightarrow^{\sum_{i=0}^{n-1}(\mathsf{step}(k+i)+1)}_{\mathcal{E}} \exists \gamma_{\mathsf{heap}}, \gamma_{\mathsf{time}}.\ \left(\mathsf{Ex}(H_n) * \blacklozenge(k+n) * \mathsf{wp}_{\mathcal{E}}\, e_n\, \{\phi\}\right)$$

Also, since $\neg\mathsf{red}(e_n, H_n)$ holds, when we have the tokens $\mathsf{Ex}(H_n)$ and $\blacklozenge(k+n)$, it turns out after $\mathsf{step}(k+n)+1$ logical steps that the weakest precondition cannot take the right-hand disjunct. So we have the following tautology.

$$(\exists \gamma_{\mathsf{heap}}, \gamma_{\mathsf{time}}.\ \mathsf{Ex}(H_n) * \blacklozenge(k+n) * \mathsf{wp}_{\mathcal{E}}\, e_n\, \{\phi\}) \Rrightarrow^{\mathsf{step}(k+n)+1}_{\mathcal{E}} \exists\, \mathfrak{a}\ \text{s.t.}\ \mathfrak{a} = e_n.\ \phi\, \mathfrak{a}$$

Combining the two tautologies, we have the following tautology.

$$\Rrightarrow^{\sum_{i=0}^{n}(\mathsf{step}(k+i)+1)}_{\mathcal{E}} \exists\, \mathfrak{a}\ \text{s.t.}\ \mathfrak{a} = e_n.\ \phi\, \mathfrak{a}$$

By the soundness theorem on the step-taking fancy update modality Theorem 2.2, we finally get $\exists\, \mathfrak{a}$ s.t. $\mathfrak{a} = e_n.\ \phi\, \mathfrak{a}$. □

For verification, it is more convenient to use the *Hoare triple* $\{P\}\ e\ \{\mathfrak{a}.\, Q_{\mathfrak{a}}\}_{\mathcal{E}}$ than directly use the weakest precondition. It is simply defined as follows.

$$\{P\}\ e\ \{\mathfrak{a}.\, Q_{\mathfrak{a}}\}_{\mathcal{E}} \ :=\ P \Rrightarrow_{\mathcal{E}} \mathsf{wp}_{\mathcal{E}}\, e\, \{\lambda \mathfrak{a}.\, Q\}$$

Note that the Hoare triple is persistent. Also, we introduce the following shorthand for the case where the returned value is ignored by the postcondition.

$$\{P\}\ e\ \{Q\}_{\mathcal{E}} \ :=\ \{P\}\ e\ \{\_.\, Q\}_{\mathcal{E}}$$

We have the following structural rules on the Hoare triple.

Hoare-Frame
$$\frac{\{P\}\ e\ \{\mathfrak{a}.\, Q_{\mathfrak{a}}\}_{\mathcal{E}}}{\{P * R\}\ e\ \{\mathfrak{a}.\, Q_{\mathfrak{a}} * R\}_{\mathcal{E}}}$$

Hoare-VShift
$$\frac{P' \Rrightarrow_{\mathcal{E}} P \qquad \{P\}\ e\ \{\mathfrak{a}.\, Q_{\mathfrak{a}}\}_{\mathcal{E}} \qquad \forall\, \mathfrak{a}.\, Q_{\mathfrak{a}} \Rrightarrow_{\mathcal{E}} Q'_{\mathfrak{a}}}{\{P'\}\ e\ \{\mathfrak{a}.\, Q'_{\mathfrak{a}}\}_{\mathcal{E}}}$$

Hoare-Ctx-In
$$\frac{\mathsf{persistent}(P) \qquad P \qquad \{P * Q\}\ e\ \{\mathfrak{a}.\, R_{\mathfrak{a}}\}_{\mathcal{E}}}{\{Q\}\ e\ \{\mathfrak{a}.\, R_{\mathfrak{a}}\}_{\mathcal{E}}}$$

Hoare-Ctx-Out
$$\frac{P \Rrightarrow_{\mathcal{E}} \{Q\}\ e\ \{\mathfrak{a}.\, R_{\mathfrak{a}}\}_{\mathcal{E}}}{\{P * Q\}\ e\ \{\mathfrak{a}.\, R_{\mathfrak{a}}\}_{\mathcal{E}}}$$

Hoare-False
$$\{\mathsf{False}\}\ e\ \{\mathfrak{a}.\ P\}_{\mathcal{E}}$$

Hoare-Disj
$$\frac{\{P\}\ e\ \{\mathfrak{a}.\ R_{\mathfrak{a}}\}_{\mathcal{E}} \qquad \{Q\}\ e\ \{\mathfrak{a}.\ R_{\mathfrak{a}}\}_{\mathcal{E}}}{\{P \lor Q\}\ e\ \{\mathfrak{a}.\ R_{\mathfrak{a}}\}_{\mathcal{E}}}$$

Hoare-Exist
$$\frac{\forall x.\ \{P_x\}\ e\ \{\mathfrak{a}.\ Q_{\mathfrak{a}}\}_{\mathcal{E}}}{\{\exists x.\ P_x\}\ e\ \{\mathfrak{a}.\ Q_{\mathfrak{a}}\}_{\mathcal{E}}}$$

Hoare-Mono-Mask
$$\frac{\{P\}\ e\ \{\mathfrak{a}.\ Q_{\mathfrak{a}}\}_{\mathcal{E}} \qquad \mathcal{E} \subseteq \mathcal{E}'}{\{P\}\ e\ \{\mathfrak{a}.\ Q_{\mathfrak{a}}\}_{\mathcal{E}'}}$$

Hoare-Frame is an important rule that removes a proposition R that is shared under *separating conjunction* by the precondition and postcondition, which is called a *frame*. Hoare-VShift modifies the precondition and postcondition of the Hoare triple using view shifts. Both Hoare-Frame and Hoare-VShift follow from Wp-VShift. Hoare-Ctx-In adds persistent knowledge P to the precondition. Hoare-Ctx-Out takes out a part P of the precondition into an assumption of the Hoare triple. Hoare-False, Hoare-Disj and Hoare-Exist eliminate false, disjunction and existential quantification in the precondition. Hoare-Mono-Mask weakens the mask.

We also have the following rules for an evaluation context and a value expression.

Hoare-EvCtx
$$\frac{\{P\}\ e\ \{\mathfrak{a}.\ Q_{\mathfrak{a}}\}_{\mathcal{E}} \qquad \forall \mathfrak{a}.\ \{Q_{\mathfrak{a}}\}\ K[\mathfrak{a}]\ \{\mathfrak{b}.\ R_{\mathfrak{b}}\}_{\mathcal{E}}}{\{P\}\ K[e]\ \{\mathfrak{b}.\ R_{\mathfrak{b}}\}_{\mathcal{E}}}$$

Hoare-Val
$$\frac{P \Rrightarrow_{\mathcal{E}} Q_{\mathfrak{a}}}{\{P\}\ \mathfrak{a}\ \{\mathfrak{a}'.\ Q_{\mathfrak{a}'}\}_{\mathcal{E}}}$$

Hoare-EvCtx follows from Wp-EvCtx and Hoare-Val follows from Wp-Val.

We have the following rule for concurrent execution.

Hoare-Concur
$$\frac{\{P\}\ e\ \{\mathfrak{a}.\ Q_{\mathfrak{a}}\}_{\mathcal{E}} \qquad \{P'\}\ e'\ \{Q'\}_{\mathcal{E}}}{\{P * P'\}\ e \parallel e'\ \{\mathfrak{a}.\ Q_{\mathfrak{a}} * Q'\}_{\mathcal{E}}}$$

This rule separatingly conjoins the preconditions and postconditions of the assumption Hoare triples on e and e'. It is a standard rule of *concurrent separation logic* (O'Hearn, 2007).

Using the lemma Wp-Step, we can prove the following Hoare-triple rules on basic operations.

Hoare-FnCall
$$\frac{\mathfrak{f} = \mathsf{fn}\, f(a_0, \ldots, a_{n-1})\{e\} \qquad \{P * \blacktriangleright 1\}\ e[\mathfrak{f}/f, \overrightarrow{\mathfrak{a}/a}]\ \{\mathfrak{c}.\ Q_{\mathfrak{c}}\}_{\mathcal{E}}}{\{\Rrightarrow^{\#}_{\mathcal{E}} P\}\ \mathfrak{f}(\mathfrak{a}_0, \ldots, \mathfrak{a}_{n-1})\ \{\mathfrak{c}.\ Q_{\mathfrak{c}}\}_{\mathcal{E}}}$$

Hoare-Alloc
$$\{\Rrightarrow^{\#}_{\mathcal{E}} P\}\ \mathsf{alloc}\, n\ \{\mathfrak{a}.\ \exists \mathfrak{l}\ \text{s.t.}\ \mathfrak{l} = \mathfrak{a}.\ \mathsf{Free}_n(\mathfrak{l}) * \mathfrak{l} \overset{1}{\mapsto} \_^n * \blacktriangleright 1 * P\}_{\mathcal{E}}$$

Hoare-Free
$$\{\Rrightarrow^{\#}_{\mathcal{E}} (\mathsf{Free}_n(\mathfrak{l}) * \mathfrak{l} \overset{1}{\mapsto} \_^n * P)\}\ \mathsf{free}\, \mathfrak{l}\ \{\blacktriangleright 1 * P\}_{\mathcal{E}}$$

Hoare-Load
$$\{\Rrightarrow^{\#}_{\mathcal{E}} (\mathfrak{l} \overset{q}{\mapsto} \mathfrak{a} * P)\}\ *\mathfrak{l}\ \{\mathfrak{a}'.\ \mathfrak{a}' = \mathfrak{a} * \mathfrak{l} \overset{q}{\mapsto} \mathfrak{a} * \blacktriangleright 1 * P\}_{\mathcal{E}}$$

Hoare-Store
$$\{\Rrightarrow^{\#}_{\mathcal{E}} (\mathfrak{l} \overset{1}{\mapsto} \_ * P)\}\ \mathfrak{l} \leftarrow \mathfrak{a}\ \{\mathfrak{l} \overset{1}{\mapsto} \mathfrak{a} * \blacktriangleright 1 * P\}_{\mathcal{E}}$$

Hoare-AddrShift
$$\{\Rrightarrow^{\#}_{\mathcal{E}} P\}\ \mathfrak{l}.n\ \{\mathfrak{a}.\ \mathfrak{a} = \mathfrak{l} + n * \blacktriangleright 1 * P\}_{\mathcal{E}}$$

HOARE-CASE
$$\frac{i = 0 \lor i = 1 \qquad \left\{ P * \text{⧗}1 \right\}\ e_i\ \left\{ \mathfrak{a}.\ Q_{\mathfrak{a}} \right\}_{\mathcal{E}}}{\left\{ \Rrightarrow^{\#}_{\mathcal{E}} P \right\}\ \mathsf{case}\ i\ \mathsf{of}\ \{ 0 \to e_0,\ 1 \to e_1 \}\ \left\{ \mathfrak{a}.\ Q_{\mathfrak{a}} \right\}_{\mathcal{E}}}$$

HOARE-INTOP
$$\left\{ \Rrightarrow^{\#}_{\mathcal{E}} P \right\}\ m\ iop\ n\ \left\{ \mathfrak{a}.\ \mathfrak{a} = m\ iop\ n * \text{⧗}1 * P \right\}_{\mathcal{E}}$$

HOARE-INTREL
$$\left\{ \Rrightarrow^{\#}_{\mathcal{E}} P \right\}\ m\ irel\ n\ \left\{ \mathfrak{a}.\ \mathfrak{a} = m\ irel\ n * \text{⧗}1 * P \right\}_{\mathcal{E}}$$

HOARE-NDINT
$$\left\{ \Rrightarrow^{\#}_{\mathcal{E}} P \right\}\ \mathsf{ndint}\ \left\{ \mathfrak{a}.\ \exists n.\ \mathfrak{a} = n * \text{⧗}1 * P \right\}_{\mathcal{E}}$$

For each physical step, we can strip off the super fancy update modality $\Rrightarrow^{\#}_{\mathcal{E}}$ and obtain a cumulative time receipt ⧗1. The rules HOARE-ALLOC, HOARE-FREE, HOARE-LOAD and HOARE-STORE follow particularly from the rules HEAP-ALLOC, HEAP-FREE, HEAP-LOAD and HEAP-STORE, respectively.

Using the rules above, we can derive the following rules.

HOARE-LET
$$\frac{\left\{ P \right\}\ e\ \left\{ \mathfrak{a}.\ \Rrightarrow^{\#}_{\mathcal{E}} Q_{\mathfrak{a}} \right\}_{\mathcal{E}} \qquad \forall \mathfrak{a}.\ \left\{ Q_{\mathfrak{a}} * \text{⧗}1 \right\}\ e'[\mathfrak{a}/a]\ \left\{ \mathfrak{b}.\ R_{\mathfrak{b}} \right\}_{\mathcal{E}}}{\left\{ P \right\}\ \mathsf{let}\ a = e\ \mathsf{in}\ e'\ \left\{ \mathfrak{b}.\ R_{\mathfrak{b}} \right\}_{\mathcal{E}}}$$

HOARE-SEQ
$$\frac{\left\{ P \right\}\ e\ \left\{ \Rrightarrow^{\#}_{\mathcal{E}} Q \right\}_{\mathcal{E}} \qquad \left\{ Q * \text{⧗}1 \right\}\ e'\ \left\{ \mathfrak{a}.\ R_{\mathfrak{a}} \right\}_{\mathcal{E}}}{\left\{ P \right\}\ e; e'\ \left\{ \mathfrak{a}.\ R_{\mathfrak{a}} \right\}_{\mathcal{E}}}$$

HOARE-IF
$$\frac{\left\{ P * \text{⧗}1 \right\}\ e_{bl}\ \left\{ \mathfrak{a}.\ Q_{\mathfrak{a}} \right\}_{\mathcal{E}}}{\left\{ \Rrightarrow^{\#}_{\mathcal{E}} P \right\}\ \mathsf{if}\ bl\ \{e_{\mathsf{tt}}\}\ \mathsf{else}\ \{e_{\mathsf{ff}}\}\ \left\{ \mathfrak{a}.\ Q_{\mathfrak{a}} \right\}_{\mathcal{E}}}$$

HOARE-SEQCOPY
$$\frac{\mathsf{len}\ \bar{\mathfrak{a}} = \mathsf{len}\ \bar{\mathfrak{b}} = n}{\left\{ \Rrightarrow^{\#}_{\mathcal{E}}{}^{n+1} (\mathfrak{l} \overset{1}{\mapsto} \bar{\mathfrak{a}} * \mathfrak{l}' \overset{q}{\mapsto} \bar{\mathfrak{b}} * P) \right\}\ \mathfrak{l} \leftarrow^{*}_{n} \mathfrak{l}'\ \left\{ \mathfrak{l} \overset{1}{\mapsto} \bar{\mathfrak{b}} * \mathfrak{l}' \overset{q}{\mapsto} \bar{\mathfrak{b}} * \text{⧗}(n+1) * P \right\}_{\mathcal{E}}}$$

4.3 Related Work

Time Receipts The idea of *time receipts* came from Mével et al. (2019). A time receipt *lower*-bounds the number of physical steps that have elapsed. It was conceived as a dual of a *time credit* (Atkey, 2011), which *upper*-bounds the number of physical steps that have elapsed and is typically used for verifying an upper bound on the execution time of a program. Mével et al. (2019) used time receipts to ensure that undesirable events like *integer overflows* cannot occur within N physical steps, where N is a *global constant* that is typically set to a very large number like 2^{63}. Unlike their use of time receipts, we use time receipts to ensure that we do not need to consume an undesirably large number of logical steps *with respect to* the number of physical steps that have elapsed. Also, our time receipt RA combines the *cumulative* and *persistent* time receipts through the *swelling* rule CUMUTIME-SWELL-PERSTIME, which is a new idea.

Transfinite Iris A recent emerging study by Spies et al. (2020) introduces a variant of Iris called *Transfinite Iris*, which incorporates *transfinite* step indexing (i.e., machinery where a step index can be a transfinite ordinal) into Iris, which currently uses *finite* step

indexing (i.e., machinery where a step index is a natural number or finite ordinal). In Transfinite Iris, we can easily spend any finite number of logical steps for each physical step. This is due to the *existential property* of Transfinite Iris, which says that if $\exists\, x\colon T.\ P_x$ is a tautology then P_x is a tautology for some $x\colon T$, as long as the cardinality of T is not too large. In particular, $\exists\, n\colon \mathbb{N}.\ \Rrightarrow^n_{\mathcal{E}} P$ is not equivalent to True in Transfinite Iris, unlike Iris. They also provide Coq mechanization of Transfinite Iris. However, in Transfinite Iris we lose many useful deduction rules of the original Iris, including commutativity of the later modality over the inhabited existential quantifier Later-Comm-InhExist and the separating conjunction Later-Comm-Sep. Because of this, it remains unclear how we can model the *lifetime logic* in Transfinite Iris, which prevents us from using Transfinite Iris for RustHornBelt.

Chapter 5

A New Design of Semantic Types for Rust

Semantically modeling the guarantees of program types is one approach to proving correctness (e.g., reducibility, memory safety) of a type system in an *extensible* way. We call such a semantic model of a program type a *semantic type*. For a simple example, if we target a simple functional programming language instead of Rust, we can semantically model each program type as the *set of values* that the type accepts. In general, this semantic approach is more extensible to new features than common syntactic approaches, such as type soundness proof by progress and preservation.

When we target Rust, one challenge is that an object of a program type can have some *ownership* or *permission* on resources. For example, an object of `Box<i32>` is a pointer that owns a memory block of an integer value with permission to update and read the integer data and release the memory block. More subtly, a *unique reference* of the type `&'a mut i32` can update and read its target integer data *only while the lifetime `'a` is alive*. This is exactly the challenge that *RustBelt* (Jung et al., 2018a) solved. They model the guarantees of program types as propositions in the separation logic Iris (Chapter 2). In order to handle borrows based on lifetimes, they built the lifetime logic (§ 2.3) on Iris.

For our work RustHornBelt, we have another challenge. We want to verify *functional correctness* of Rust programs, not only safety, unlike RustBelt. Moreover, we want to model each object of Rust as a *pure value* without irrelevant low-level information, even using *prophecy* to model unique references. For example, an object of the box pointer type `Box<i32>` should be modeled simply as its target integer value, without the information about the address. Furthermore, a unique reference `&'a mut i32` should be modeled as a pair of two integers, the current target value and the final target value that is prophesied.

For this purpose, we extend RustBelt's approach using our new formulation of prophecy in Iris introduced in Chapter 3. A key idea is to assign to each Rust object a *π-parametrized pure value*, a pure value that is parametrized over assignments on prophecy variables, as we roughly discussed in § 3.2.

In this chapter, we introduce our new design of semantic types for Rust. In § 5.1, we give an overview and a formal definition of our notion of the semantic type. In § 5.2, we present the semantic types for basic program types in Rust. In § 5.3, we present the semantic type for the *unique reference type* in Rust, introducing some additional machinery on prophecy. In § 5.4, we present the semantic type for the general recursive type.

Defining semantic types itself does not conclude the functional correctness proof for Rust programs. Later in Chapter 6, we introduce a type system that handles logic models, which we dub a *refined type system*, for Rust programs. It uses a typing judgment emitting a logic model, dubbed a refined typing judgment, which is modeled as persistent Iris propositions using the Iris predicates of the semantics types we define.

Related work of the combination of this chapter and Chapter 6 is discussed in §6.5. The function type is defined later in § 6.2.4, because it is modeled using the refined typing judgment.

5.1 Our Notion of the Semantic Type

Now we introduce our new notion of the semantic type for RustHornBelt, extending the approach of RustBelt (Jung et al., 2018a) with our formulation of prophecy introduced in Chapter 3.

First, a semantic type for a Rust program type has an *Iris* predicate *Own* describing the *ownership* of the object of that type, which we call the *ownership predicate*. The predicate *Own* has the type $\mathit{LowVal} \to (\mathsf{ProphAsn} \to \mathit{PureVal}) \to \mathsf{IProp}$. Here, *LowVal* and *PureVal* are types of low-level values and pure values designed for the program type. The *Own* predicate takes as an argument a *π-parametrized pure value* $\hat{v}$ instead of just a pure value, in order to handle prophecy in the style described in §3.2.

For example, let us discuss the semantic type for the box pointer type `Box<i32>`. We can set the type *LowVal* to the address type and set the type *PureVal* to the integer type $\mathbb{Z}$. The ownership predicate $\mathit{Own}\, \mathfrak{l}\, \hat{v}$ for the box pointer type is set to $\exists\, n$ s.t. $\hat{v} = \mathsf{const}\, n$. $\mathfrak{l} \overset{1}{\mapsto} n$, saying that $\hat{v}$ is a constant function over some integer n and the object fully owns the address $\mathfrak{l}$ with the value n.

For another example, let us roughly discuss the semantic type for the the unique reference type `&'a mut i32`. We can set *LowVal* again to the address type and and set *PureVal* to the *pair type of two integers*, in the style of RustHorn. Defining the ownership predicate $\mathit{Own}\, \mathfrak{l}\, \hat{v}$ for `&'a mut i32` is fairly challenging. Roughly speaking, it contains existential quantification like $\exists\, x$ s.t. $(.1) \circ \hat{v} = \lambda \pi.\, \pi\, x.\ \cdots$, checking that there is some prophecy variable x that represents the final target value on $\hat{v}$, and the full prophecy token $[x]_1$ is in some sense owned by the unique reference. Later this part is elaborated in §5.3.

The story does not end here. In order to deal with shared references `&'a T` in general, we equip the semantic type with another predicate *Share*: $\mathsf{Lft} \to \mathit{LowVal} \to \mathit{PureVal} \to \mathsf{IProp}$, representing the condition to *share* an object of the program type with some low-level and pure values. We need this because in Rust a program type works quite differently under the ownership (unique permission) and the sharing permission. For example, a shared reference to a unique reference `&'b &'a mut i32` is copyable but only has the sharing permission on the inner integer data (as explained by Example 1.8), unlike a bare unique reference `&'a mut i32`. The idea of using two predicates *Own* and *Share* for unique and sharing permissions comes from RustBelt.

Our actual formulation is a bit different from the explanation above. As a technical workaround to mitigate *later modalities* in Iris, we add to *Own* and *Share* an extra natural-number parameter dubbed the *depth*. Also, we use symbolic notation $\bar{\mathfrak{a}} \blacktriangleleft \tau\, \{\hat{v}\}_d$, which corresponds to something like $\mathit{Own}_\tau\, \bar{\mathfrak{a}}\, \hat{v}\, d$ in the explanation above, and write $\bar{\mathfrak{a}} \triangleleft^\alpha \tau\, \{\hat{v}\}_d$ for the *Share* counterpart.

For simplicity and flexibility, in this thesis, we formalize program types only in terms of *semantic* types, not introducing any fixed *syntax* of program types. We can do so particularly because later (in Chapter 6) we formulate our deductive type system for Rust only in semantic terms, without introducing any syntactic structures about deduction.

Formal Definition of a Semantic Type We write $\mathsf{List}_n\, \mathsf{CellVal}$ for the subtype $\{\, \bar{\mathfrak{a}} : \mathsf{List}\, \mathsf{CellVal} \mid \mathsf{len}\, \bar{\mathfrak{a}} = n \,\}$, i.e., the type of a list of cell values that has the length n.

A *semantic type* τ: SemTy is a quintuple

$$\begin{aligned}&(\ \lfloor\tau\rfloor : \mathsf{Type},\quad |\tau| : \mathbb{N},\quad -\sqsubseteq\tau : \mathsf{Lft}\to\mathsf{IProp},\\ &\quad -\blacktriangleleft\tau\{-\}_- : \mathsf{List}_{|\tau|}\,\mathsf{CellVal}\to(\mathsf{ProphAsn}\to\lfloor\tau\rfloor)\to\mathbb{N}\to\mathsf{IProp}\\ &\quad -\triangleleft^{-}\tau\{-\}_- : \mathsf{List}_{|\tau|}\,\mathsf{CellVal}\to\mathsf{Lft}\to(\mathsf{ProphAsn}\to\lfloor\tau\rfloor)\to\mathbb{N}\to\mathsf{IProp}\ )\end{aligned}$$

that satisfies the following properties.

SemTy-Outlv-Pers
$\mathsf{persistent}(\alpha\sqsubseteq\tau)$

SemTy-Outlv-Mono
$$\frac{\alpha\sqsubseteq\tau\qquad\beta\sqsubseteq\alpha}{\beta\sqsubseteq\tau}$$

SemTy-Own-Mono
$$\frac{d\le d'}{\bar{\mathfrak{a}}\blacktriangleleft\tau\{\hat{v}\}_d\ \Rrightarrow\ \bar{\mathfrak{a}}\blacktriangleleft\tau\{\hat{v}\}_{d'}}$$

SemTy-Shr-Pers
$\mathsf{persistent}(\bar{\mathfrak{a}}\triangleleft^{\alpha}\tau\{\hat{v}\}_d)$

SemTy-Shr-Mono
$$\frac{\bar{\mathfrak{a}}\triangleleft^{\alpha}\tau\{\hat{v}\}_d\qquad\beta\sqsubseteq\alpha\qquad d\le d'}{\bar{\mathfrak{a}}\triangleleft^{\beta}\tau\{\hat{v}\}_{d'}}$$

SemTy-Own-Shr
$$\frac{\alpha\sqsubseteq\tau\qquad\alpha\sqsubseteq\beta}{\&^{\beta}_{\mathsf{full}}(\bar{\mathfrak{a}}\blacktriangleleft\tau\{\hat{v}\}_d)\ *\ [\alpha]_q\ \Rrightarrow^{d}_{\mathcal{N}_{\mathsf{lft}}}\ \bar{\mathfrak{a}}\triangleleft^{\beta}\tau\{\hat{v}\}_d\ *\ [\alpha]_q}$$

SemTy-Own-ProphToken
$$\frac{\alpha\sqsubseteq\tau}{\begin{array}{l}\bar{\mathfrak{a}}\blacktriangleleft\tau\{\hat{v}\}_d\ *\ [\alpha]_q\ \Rrightarrow^{d}_{\mathcal{N}_{\mathsf{lft}}}\\ \quad\exists X\ \text{s.t.}\ \mathsf{Dep}(\hat{v},X).\ \exists q'.\ [X]_{q'}\ *\ \left([X]_{q'}\ \Rrsquigarrow_{\mathcal{N}_{\mathsf{lft}}}\ \bar{\mathfrak{a}}\blacktriangleleft\tau\{\hat{v}\}_d\ *\ [\alpha]_q\right)\end{array}}$$

SemTy-Shr-ProphToken
$$\frac{\alpha\sqsubseteq\tau\qquad\alpha\sqsubseteq\beta}{\begin{array}{l}\bar{\mathfrak{a}}\triangleleft^{\beta}\tau\{\hat{v}\}_d\ *\ [\alpha]_q\ \Rrightarrow^{d}_{\mathcal{N}_{\mathsf{lft}}}\\ \quad\exists X\ \text{s.t.}\ \mathsf{Dep}(\hat{v},X).\ \exists q'.\ [X]_{q'}\ *\ \left([X]_{q'}\ \Rrsquigarrow_{\mathcal{N}_{\mathsf{lft}}}\ \bar{\mathfrak{a}}\triangleleft^{\beta}\tau\{\hat{v}\}_d\ *\ [\alpha]_q\right)\end{array}}$$

A semantic type τ consists of: (i) the *pure type* $\lfloor\tau\rfloor$, which is used for values modeling data of τ; (ii) the *size* $|\tau|$: int, which is the number of memory cells used for data typed τ at the shallowest level; (iii) the *outliving predicate* $\alpha\sqsubseteq\tau$: IProp for α: Lft, which is the persistent condition for the type τ to outlive the lifetime α; (iv) the *ownership predicate* $\bar{\mathfrak{a}}\blacktriangleleft\tau\{\hat{v}\}_d$, which is the condition to own data typed τ that has the list of cell values $\bar{\mathfrak{a}}$ at the low level, is modeled as the π-parametrized value $\hat{v}$, and has the *depth* d; and (v) the *sharing predicate* $\bar{\mathfrak{a}}\triangleleft^{\alpha}\tau\{\hat{v}\}_d$, which is the persistent condition to share data typed τ associated with $\bar{\mathfrak{a}}$, $\hat{v}$ and d until the lifetime α ends.

We equip the semantic type with an outliving predicate $\alpha\sqsubseteq\tau$ instead of directly giving the lifetime of the type. It enables us to use *later modalities*, which facilitates definition of the recursive type (§5.4).[1]

The outliving predicate $\alpha\sqsubseteq\tau$ should be persistent (SemTy-Outlv-Pers) and anti-monotone over α (SemTy-Outlv-Mono). The ownership predicate $\bar{\mathfrak{a}}\blacktriangleleft\tau\{\hat{v}\}_d$ should be monotone over d (SemTy-Own-Mono). The sharing predicate $\bar{\mathfrak{a}}\triangleleft^{\alpha}\tau\{\hat{v}\}_d$ should be persistent (SemTy-Shr-Pers) as well as anti-monotone over α and monotone over d (SemTy-Shr-Mono). SemTy-Own-Shr says that we should be able to transform a full borrow of the ownership predicate $\&^{\beta}_{\mathsf{full}}\bar{\mathfrak{a}}\blacktriangleleft\tau\{\hat{v}\}_d$ into a sharing predicate $\bar{\mathfrak{a}}\triangleleft^{\beta}\tau\{\hat{v}\}_d$ taking d logical steps, by temporarily depositing a partial token on $\alpha\sqcap\beta$. SemTy-Own-ProphToken says that, out of the ownership predicate $\bar{\mathfrak{a}}\blacktriangleleft\tau\{\hat{v}\}_d$, we should be able to temporarily take a partial prophecy token $[X]_{q'}$ on prophecy variables X that $\hat{v}$ depends on, taking d logical steps, by temporarily depositing a partial token on α. SemTy-Shr-ProphToken is a similar rule for a sharing predicate. Note that the depth of an

[1] This idea was conceived by Jacques-Henri Jourdan.

object determines how many logical steps we can use for SemTy-Own-Shr, SemTy-Own-ProphToken and SemTy-Shr-ProphToken.

5.2 Basic Semantic Types

In this section, we introduce a number of basic semantic types.

We introduce notation for list slicing. For a list v and i such that $0 \le i < \mathrm{len}\, v$, $v[..i]$ and $v[i..]$ denote $[v[0], \ldots, v[i-1]]$ and $[v[i], \ldots, v[\mathrm{len}\, v - 1]]$ respectively. Also, for a list v and i, j such that $0 \le i \le j < \mathrm{len}\, v$, $v[i..j]$ denotes $[v[i], \ldots, v[j-1]]$.

Integer Type The (unbounded) *integer type* int is an idealization of Rust's bounded integer types like `i32`. Note that in our low-level formulation one memory cell can contain any integer, idealizing the real-world hardware. We define int as the following semantic type.

$$\lfloor \mathsf{int} \rfloor := \mathbb{Z} \qquad |\mathsf{int}| := 1 \qquad \alpha \sqsubseteq \mathsf{int} := \mathsf{True}$$

$$[\mathfrak{a}] \blacktriangleleft \mathsf{int}\,\{\hat{v}\}_d = [\mathfrak{a}] \triangleleft^{\alpha} \mathsf{int}\,\{\hat{v}\}_d := \exists\, n \text{ s.t. } n = \mathfrak{a}.\ \hat{v} = \mathsf{const}\, n$$

We use an integer of the mathematical type $\mathbb{Z}$ to model an object of the program type int. At the low level, int occupies one memory cell that has an integer value n. Any lifetime outlives int. The π-parametrized integer value for int should be a constant function that returns the expected integer n. Any depth is permitted for int.

Boolean Type The *boolean type* bool, which corresponds to Rust's `bool`, is defined as follows, in a similar way to the integer type.

$$\lfloor \mathsf{bool} \rfloor := \mathbb{B} \qquad |\mathsf{bool}| := 1 \qquad \alpha \sqsubseteq \mathsf{bool} := \mathsf{True}$$

$$[\mathfrak{a}] \blacktriangleleft \mathsf{bool}\,\{\hat{v}\}_d = [\mathfrak{a}] \triangleleft^{\alpha} \mathsf{bool}\,\{\hat{v}\}_d := \exists\, bl \text{ s.t. } bl = \mathfrak{a}.\ \hat{v} = \mathsf{const}\, bl$$

To model boolean data, we use the mathematical boolean type $\mathbb{B}$, which has two values tt (true) and ff (false).

Invalid-Data Type We introduce the *invalid-data type* $\lightning_n$ of the size n.

$$\lfloor \lightning_n \rfloor := \mathsf{Unit} \qquad |\lightning_n| := n \qquad \alpha \sqsubseteq \lightning_n := \mathsf{True}$$

$$\bar{\mathfrak{a}} \blacktriangleleft \lightning_n\,\{\hat{v}\}_d = \bar{\mathfrak{a}} \triangleleft^{\alpha} \lightning_n\,\{\hat{v}\}_d := \mathsf{True}$$

The π-parametrized value $\hat{v}$ used here has the type $\mathsf{ProphAsn} \to \mathsf{Unit}$, which has only one value $\mathsf{const}\,()$. In modeling Rust, we can use the invalid-data type $\lightning_n$ to model data invalidated after a move operation and also to model uninitialized data obtained by memory allocation.[2] In particular, an object of a type τ can be invalidated into an invalid object of the type $\lightning_{|\tau|}$.

Box Pointer Type The *box pointer type* box τ, which corresponds to `Box<T>` in Rust, represents a pointer that targets an object of the type τ and has *full ownership* on the target object, i.e., can read, update and delete it. It can be defined as follows.

[2] This is actually a simplification of the actual behavior of the Rust compiler, which handles validity flags on each variable apart from type information.

$$\lfloor \mathrm{box}\,\tau \rfloor := \lfloor \tau \rfloor \qquad |\mathrm{box}\,\tau| := 1 \qquad \alpha \sqsubseteq \mathrm{box}\,\tau := \triangleright\,(\alpha \sqsubseteq \tau)$$

$$[\mathfrak{a}] \blacktriangleleft \mathrm{box}\,\tau\,\{\hat{v}\}_d := d > 0 * \exists \mathfrak{l}\ \text{s.t.}\ \mathfrak{l} = \mathfrak{a}.\ \exists \bar{\mathfrak{b}}.\ \mathfrak{l} \overset{1}{\mapsto} \bar{\mathfrak{b}} * \mathsf{Free}_{|\tau|}(\mathfrak{l}) * \triangleright\,(\bar{\mathfrak{b}} \blacktriangleleft \tau\,\{\hat{v}\}_{d-1})$$

$$[\mathfrak{a}] \blacktriangleleft^{\alpha} \mathrm{box}\,\tau\,\{\hat{v}\}_d := d > 0 * \exists \mathfrak{l}\ \text{s.t.}\ \mathfrak{l} = \mathfrak{a}.\ \exists \bar{\mathfrak{b}}.\ *_i\, \&^{\alpha}_{\mathsf{frac}}(\lambda q.\, \mathfrak{l} + i \overset{q}{\mapsto} \bar{\mathfrak{b}}[i]) * \triangleright\,(\bar{\mathfrak{b}} \blacktriangleleft^{\alpha} \tau\,\{\hat{v}\}_{d-1})$$

The value for a box pointer is set to the value for its target object, without the address information. The outliving predicate of box τ is $\triangleright\,(\alpha \sqsubseteq \tau)$, the outliving predicate of the target type under the later modality.

The ownership predicate of box τ consists mainly of (i) the *full points-to token* $\mathfrak{l} \overset{1}{\mapsto} \bar{\mathfrak{b}}$, which owns $|\tau|$ consecutive memory cells starting at $\mathfrak{l}$ with the cell values $\bar{\mathfrak{b}}$, (ii) the *free-right token* $\mathsf{Free}_{|\tau|}(\mathfrak{l})$, which owns the right to free the memory block of the size $|\tau|$ starting at $\mathfrak{l}$, and (iii) $\triangleright\,(\bar{\mathfrak{b}} \blacktriangleleft \tau\,\{\hat{v}\}_{d-1})$, the ownership predicate on the target object under the later modality.

On the other hand, the sharing predicate consists mainly of (i) *fractured borrows* on the points-to tokens to each address $*_i\, \&^{\alpha}_{\mathsf{frac}}(\lambda q.\, \mathfrak{l} + i \overset{q}{\mapsto} \bar{\mathfrak{b}}[i])$,[3] from which we can temporarily take out $\mathfrak{l} \overset{q}{\mapsto} \bar{\mathfrak{b}}$ for some q while the lifetime α is alive, and (ii) the target sharing predicate under the later modality $\triangleright\,(\bar{\mathfrak{b}} \blacktriangleleft^{\alpha} \tau\,\{\hat{v}\}_{d-1})$.

The *contractiveness* introduced by the later modality in the outliving, owning and sharing predicates is useful in defining *recursive types* with self reference under box (e.g., the standard singly linked list type; see §5.4). The depth of a box pointer is set to 1 plus the depth of its target object, which gives us one logical step to strip off the later modality on the outliving, ownership and sharing predicate in proving SemTy-Own-Shr, SemTy-Own-ProphToken and SemTy-Shr-ProphToken of box τ. Here we elaborate the proof of SemTy-Own-Shr of box τ.

Proof of SemTy-Own-Shr of box τ. Assume that the input is a full borrow under α of the ownership predicate of a box pointer, $\&^{\alpha}_{\mathsf{full}}[\mathfrak{a}] \blacktriangleleft \mathrm{box}\,\tau\,\{\hat{v}\}_d$. We freeze the inner parameters $\mathfrak{l}, \bar{\mathfrak{b}}$ of the full borrow (FullBor-Freeze) and then split it (FullBor-Split) to get two full borrows, (i) a full borrow of the points-to token $P := \&^{\alpha}_{\mathsf{full}}(\mathfrak{l} \overset{1}{\mapsto} \bar{\mathfrak{b}})$ and (ii) a full borrow of the target ownership predicate under later $Q := \&^{\alpha}_{\mathsf{full}}(\triangleright\,\bar{\mathfrak{b}} \blacktriangleleft^{\alpha} \tau\,\{\hat{v}\}_{d-1})$. In one logical step, we strip off the later modality of the full borrow Q (FullBor-Unlater) and also the outliving predicate on the target object. Using SemTy-Own-Shr on the target type τ in $d-1$ logical steps, we get the target sharing predicate. The full borrow P can be transformed into fractured borrows $*_i\, \&^{\alpha}_{\mathsf{frac}}(\lambda q.\, \mathfrak{l} + i \overset{q}{\mapsto} \bar{\mathfrak{b}}[i])$ (by FullBor-Split and FullBor-FracBor). □

Shared Reference Type The *shared reference type* $\&^{\alpha}_{\mathsf{shr}}\,\tau$, which corresponds to `&'a T` in Rust, can be defined fairly easily using the sharing predicate of τ.

$$\lfloor \&^{\alpha}_{\mathsf{shr}}\,\tau \rfloor := \lfloor \tau \rfloor \qquad |\&^{\alpha}_{\mathsf{shr}}\,\tau| := 1 \qquad \beta \sqsubseteq \&^{\alpha}_{\mathsf{shr}}\,\tau := \beta \sqsubseteq \alpha \wedge \triangleright\,(\beta \sqsubseteq \tau)$$

$$[\mathfrak{a}] \blacktriangleleft \&^{\alpha}_{\mathsf{shr}}\,\tau\,\{\hat{v}\}_d = [\mathfrak{a}] \blacktriangleleft^{\beta} \&^{\alpha}_{\mathsf{shr}}\,\tau\,\{\hat{v}\}_d := d > 0 * \exists \mathfrak{l}\ \text{s.t.}\ \mathfrak{l} = \mathfrak{a}.\ \exists \bar{\mathfrak{b}}.\ *_i\, \&^{\alpha}_{\mathsf{frac}}(\lambda q.\, \mathfrak{l} + i \overset{q}{\mapsto} \bar{\mathfrak{b}}[i]) * \triangleright\,(\bar{\mathfrak{b}} \blacktriangleleft^{\alpha} \tau\,\{\hat{v}\}_{d-1})$$

The ownership and sharing predicates of $\&^{\alpha}_{\mathsf{shr}}\,\tau$ is the same as the sharing predicate of box τ with the lifetime α. Interestingly, we can ignore the lifetime argument β of the sharing predicate; we can still prove SemTy-Own-Shr because the ownership predicate

[3] We need to use $*_i\, \&^{\alpha}_{\mathsf{frac}}(\lambda q.\, \mathfrak{l} + i \overset{q}{\mapsto} \bar{\mathfrak{b}}[i])$ instead of simpler $\&^{\alpha}_{\mathsf{frac}}(\lambda q.\, \mathfrak{l} \overset{q}{\mapsto} \bar{\mathfrak{b}})$ because the lifetime logic does not have the split rule like FullBor-Split for fractured borrows.

is already persistent. Note that SemTy-Own-PropHToken of $\&^{\alpha}_{\mathsf{shr}}\,\tau$ can be proved using SemTy-Shr-PropHToken on the target type τ.

The later modality in the outliving, ownership and sharing predicates are for contractiveness, just as in the box pointer type box τ.

Plain Reference Type To aid verification, we introduce the following *plain reference type* $\&_q\,\tau$ for a type τ and a fraction q.

$$\lfloor \&_q\,\tau \rfloor \coloneqq \lfloor \tau \rfloor \qquad |\&_q\,\tau| \coloneqq 1 \qquad \alpha \sqsubseteq \&_q\,\tau \coloneqq \triangleright\big(\alpha \sqsubseteq \tau\big)$$

$$[\mathfrak{a}] \blacktriangleleft \&_q\,\tau\,\{\hat{v}\}_d \coloneqq d > 0 \ast \exists \mathfrak{l}\ \text{s.t.}\ \mathfrak{l} = \mathfrak{a}.\ \exists \bar{\mathfrak{b}}.\ \mathfrak{l} \overset{q}{\mapsto} \bar{\mathfrak{b}} \ast \triangleright\big(\bar{\mathfrak{b}} \blacktriangleleft \tau\,\{\hat{v}\}_{d-1}\big)$$

$$[\mathfrak{a}] \triangleleft^{\alpha} \&_q\,\tau\,\{\hat{v}\}_d \coloneqq \mathsf{True}$$

It is a reference to an object typed τ. The pure value of a plain reference is set to the pure value of its target object. The ownership predicate of a plain reference mainly consists of (i) the points-to token to the target values of the fraction q, $\mathfrak{l} \overset{q}{\mapsto} \bar{\mathfrak{b}}$, and (ii) the ownership predicate on the target object under later, $\triangleright\big(\bar{\mathfrak{b}} \blacktriangleleft \tau\,\{\hat{v}\}_{d-1}\big)$.

A plain reference type is intended to be used at the top level of a type and not inside other types. Since we do not need to use the plain reference type under a shared reference, the sharing predicate of the type is set to the trivial proposition True.

The later modality in the outliving, ownership and sharing predicates helps compatibility with other pointer types such as the box pointer type box τ.

We call a plain reference with the fraction 1 of the type $\&_1\,\tau$ a *full plain reference* and call a plain reference with any fraction q a *fractional plain reference*.

Pair Type The *pair type* $\tau \times \tau'$ of the semantic types τ and τ', which corresponds to Rust's `(T, T')`, is defined as follows.

$$\lfloor \tau \times \tau' \rfloor \coloneqq \lfloor \tau \rfloor \times \lfloor \tau' \rfloor \qquad |\tau \times \tau'| \coloneqq |\tau| + |\tau'| \qquad \alpha \sqsubseteq \tau \times \tau' \coloneqq \alpha \sqsubseteq \tau \wedge \alpha \sqsubseteq \tau'$$

$$\bar{\mathfrak{a}} \blacktriangleleft \tau \times \tau'\,\{\hat{v}\}_d \coloneqq \bar{\mathfrak{a}}[..|\tau|] \blacktriangleleft \tau\,\{(.0) \circ \hat{v}\}_d \ast \bar{\mathfrak{a}}[|\tau|..] \blacktriangleleft \tau'\,\{(.1) \circ \hat{v}\}_d$$

$$\bar{\mathfrak{a}} \triangleleft^{\alpha} \tau \times \tau'\,\{\hat{v}\}_d \coloneqq \bar{\mathfrak{a}}[..|\tau|] \triangleleft^{\alpha} \tau\,\{(.0) \circ \hat{v}\}_d \ast \bar{\mathfrak{a}}[|\tau|..] \triangleleft^{\alpha} \tau'\,\{(.1) \circ \hat{v}\}_d$$

For $i = 0, 1$, $(.i)$ denotes the function $\lambda v.\, v.i$ and thus $(.i) \circ \hat{v}$ is equal to $\lambda \pi.\, (\hat{v}\,\pi).i$.

We piggyback the components ($\lfloor \tau \rfloor$, $\alpha \sqsubseteq \tau$, etc.) of τ and τ' to construct the semantic type $\tau \times \tau'$. The list of cell values for data typed $\tau \times \tau'$ is the concatenation of the list for τ and the list for τ'. The value for data typed $\tau \times \tau'$ is set to the pair of the value for $\lfloor \tau \rfloor$ and the value for $\lfloor \tau' \rfloor$ (for each π). The outliving predicate of $\tau \times \tau'$ is the conjunction of those of τ and τ'. The ownership predicate for $\tau \times \tau'$ is defined as the separating conjunction of those for τ and τ', sharing the same depth. The sharing predicate is defined in a similar way. We can prove the required properties (SemTy-Outlv-Pers, etc.) of $\tau \times \tau'$ by using those of τ and τ'.

Variant Type The *variant type* $\tau_0 + \tau_1$, modeling Rust's `enum` type (e.g., `Result<T0, T1>`), is defined as follows.

$$\lfloor \tau_0 + \tau_1 \rfloor \coloneqq \lfloor \tau_0 \rfloor + \lfloor \tau_1 \rfloor \qquad |\tau_0 + \tau_1| \coloneqq 1 + \max\{|\tau_0|, |\tau_1|\}$$

$$\alpha \sqsubseteq \tau_0 + \tau_1 \coloneqq \alpha \sqsubseteq \tau_0 \wedge \alpha \sqsubseteq \tau_1$$

$$\bar{\mathfrak{a}} \blacktriangleleft \tau_0 + \tau_1\,\{\hat{v}\}_d \coloneqq \exists i, \hat{w}\ \text{s.t.}\ \bar{\mathfrak{a}}[0] = i \wedge \hat{v} = \mathsf{inj}_i \circ \hat{w}.\ \bar{\mathfrak{a}}[1\,..\,1+|\tau_i|] \blacktriangleleft \tau_i\,\{\hat{w}\}_d$$

$$\bar{\mathfrak{a}} \triangleleft^{\alpha} \tau_0 + \tau_1\,\{\hat{v}\}_d \coloneqq \exists i, \hat{w}\ \text{s.t.}\ \bar{\mathfrak{a}}[0] = i \wedge \hat{v} = \mathsf{inj}_i \circ \hat{w}.\ \bar{\mathfrak{a}}[1\,..\,1+|\tau_i|] \triangleleft^{\alpha} \tau_i\,\{\hat{w}\}_d$$

Here, i is either 0 or 1 and $\hat{w}$ is a π-parametrized value of the type $\mathsf{ProphAsn} \to \lfloor \tau_i \rfloor$ (depending on i). Note that $\mathsf{inj}_i \circ \hat{w}$ is equal to $\lambda \pi.\, \mathsf{inj}_i(\hat{w}\, \pi)$.

Vector Type The *vector type* $\mathsf{vec}\, \tau$, modeling Rust's `Vec<T>`, represents a dynamically allocated, growable array of objects typed τ. It is defined as follows, extending the definition of the box pointer type $\mathsf{box}\, \tau$.

$$\lfloor \mathsf{vec}\, \tau \rfloor := \mathsf{List} \lfloor \tau \rfloor \qquad |\mathsf{vec}\, \tau| := 3 \qquad \alpha \sqsubseteq \mathsf{vec}\, \tau := \triangleright \left(\alpha \sqsubseteq \tau \right)$$

$$\begin{aligned} \bar{\mathfrak{a}} \blacktriangleleft \mathsf{vec}\, \tau\, \{\hat{v}\}_d := {} & d > 0 * \exists \mathfrak{l}, \mathit{cap}, \mathit{len} \text{ s.t. } [\mathfrak{l}, \mathit{cap}, \mathit{len}] = \bar{\mathfrak{a}} \wedge \mathit{cap} \geq \mathit{len}. \\ & \exists \bar{\mathfrak{b}} \text{ s.t. } \mathsf{len}\, \bar{\mathfrak{b}} = \mathit{cap} \cdot |\tau|.\ \mathfrak{l} \overset{1}{\mapsto} \bar{\mathfrak{b}} * \mathsf{Free}_{\mathit{cap} \cdot |\tau|}(\mathfrak{l}) * (\forall \pi.\ \mathsf{len}(\hat{v}\, \pi) = \mathit{len}) * \\ & \ast_{i < \mathit{len}} \triangleright \left(\bar{\mathfrak{b}}[i \cdot |\tau| \mathinner{..} (i+1) \cdot |\tau|] \blacktriangleleft \tau\, \{\lambda \pi.\, (\hat{v}\, \pi)[i]\}_{d-1} \right) \end{aligned}$$

$$\begin{aligned} \bar{\mathfrak{a}} \triangleleft^{\alpha} \mathsf{vec}\, \tau\, \{\hat{v}\}_d := {} & d > 0 * \exists \mathfrak{l}, \mathit{cap}, \mathit{len} \text{ s.t. } [\mathfrak{l}, \mathit{cap}, \mathit{len}] = \bar{\mathfrak{a}} \wedge \mathit{cap} \geq \mathit{len}. \\ & \exists \bar{\mathfrak{b}} \text{ s.t. } \mathsf{len}\, \bar{\mathfrak{b}} = \mathit{cap} \cdot |\tau|. \quad \ast_i \&^{\alpha}_{\mathsf{frac}} \left(\lambda q.\, \mathfrak{l} + i \overset{q}{\mapsto} \bar{\mathfrak{b}}[i] \right) * (\forall \pi.\ \mathsf{len}(\hat{v}\, \pi) = \mathit{len}) * \\ & \ast_{i < \mathit{len}} \triangleright \left(\bar{\mathfrak{b}}[i \cdot |\tau| \mathinner{..} (i+1) \cdot |\tau|] \triangleleft^{\alpha} \tau\, \{\lambda \pi.\, (\hat{v}\, \pi)[i]\}_{d-1} \right) \end{aligned}$$

At the low level, a vector consists of three cell values — the head location of data $\mathfrak{l}$, the capacity size *cap*, and the content length *len*. The value for $\mathsf{vec}\, \tau$ is set to the list of *len* elements.

The ownership predicate of the vector type consists mainly of (i) the full points-to token $\mathfrak{l} \overset{1}{\mapsto} \bar{\mathfrak{b}}$ that owns $\mathit{cap} \cdot |\tau|$ (= $\mathsf{len}\, \bar{\mathfrak{b}}$) consecutive memory cells starting at $\mathfrak{l}$, (ii) the free-right token $\mathsf{Free}_{\mathit{cap} \cdot |\tau|}(\mathfrak{l})$ on the memory block of the size $\mathit{cap} \cdot |\tau|$ starting at $\mathfrak{l}$, and (iii) the (separating) conjunction of the ownership predicate for the *len* elements. The sharing predicate is obtained by some modification to the ownership predicate, in an analogous way to the box pointer type.

The later modality in the outliving, ownership and sharing predicates are for contractiveness, just as in the box pointer type $\mathsf{box}\, \tau$.

5.3 Modeling the Unique Reference Type with Prophecy

The unique reference type $\&^{\alpha}_{\mathsf{unq}}\, \tau$ is the key semantic type in our framework RustHornBelt. A unique reference is modeled as the π-parametrized value $\lambda \pi.\, (\hat{v}\, \pi, \pi\, x)$, where $\hat{v}$ models the current target value and x is the prophecy variable for the borrow.

We should design the semantic type $\&^{\alpha}_{\mathsf{unq}}\, \tau$ so that we can support various operations on unique references. First, when we release a unique reference modeled $\lambda \pi.\, (\hat{v}\, \pi, \pi\, x)$, we should be able to resolve the prophecy variable x to $\hat{v}$. On the other hand, we should also be able to throw away a unique reference *without* resolving the prophecy variable, letting the borrowee resolve the prophecy variable when it reclaims the borrowed object; we need this feature because Rust's type system allows an object to be *leaked* out of the static control, especially when we have circular references (Klabnik et al., 2018, §15.6). Moreover, we should be able to subdivide a unique reference in various ways (e.g., take $\&^{\alpha}_{\mathsf{unq}}\, \tau$ out of $\&^{\alpha}_{\mathsf{unq}}(\tau \times \tau')$) and also support unique reborrows.

We already have the *lifetime logic* (introduced in § 2.3) developed by Jung et al. (2018a), which supports various operations for borrows. We set up some machinery on the side of prophecy.

Prophecy Equalizer First, we introduce the following predicate dubbed a *prophecy equalizer* $\mathsf{PE}(\hat{w}, \hat{v})$ for $\hat{w}, \hat{v}$: $\mathsf{ProphAsn} \to T$ on some type T.

$$\mathsf{PE}(\hat{w}, \hat{v}) := \forall X \text{ s.t. } \mathsf{Dep}(\hat{v}, X).\ \forall q.\ [X]_q \Rrightarrow \ast_{\mathcal{N}_{\mathsf{proph}}} \langle \pi.\ \hat{w}\, \pi = \hat{v}\, \pi \rangle * [X]_q$$

It can be explained as a *delayed* prophecy observation $\langle \pi.\ \hat{w}\,\pi = \hat{v}\,\pi \rangle$; the acquisition of the prophecy observation is delayed until we get a partial prophecy token $[X]_q$ on the dependency X of $\hat{v}$. This notion is useful for supporting *flexible timing of resolution*; we need it for RustHornBelt, because the prophecy variable of a unique borrow can be resolved by the unique reference and also by the borrowee.

We can use a prophecy equalizer in the following way.

$$\textsc{ProphEqz-Use}\quad \frac{\mathsf{Dep}(\hat{v}, X)}{\mathsf{PE}(\hat{w}, \hat{v}) \ast [X]_q \Rrightarrow_{\mathcal{N}_{\mathsf{proph}}} \langle \pi.\ \hat{w}\,\pi = \hat{v}\,\pi \rangle \ast [X]_q}$$

We can construct prophecy equalizers using the following lemmas.

$$\textsc{ProphEqz-ProphToken}\quad [x]_1 \Rightarrow \mathsf{PE}\big(\lambda \pi.\, \pi\, x,\ \hat{v}\big)$$

$$\textsc{ProphEqz-ProphObs}\quad \frac{\langle \pi.\ \hat{w}\,\pi = \hat{v}\,\pi \rangle}{\mathsf{PE}(\hat{w}, \hat{v})}$$

$$\textsc{ProphEqz-Modify}\quad \frac{\langle \pi.\ \hat{w}'\,\pi = \hat{w}\,\pi \rangle}{\mathsf{PE}(\hat{w}, \hat{v}) \Rightarrow \mathsf{PE}(\hat{w}', \hat{v})}$$

$$\textsc{ProphEqz-Transform}\quad \frac{f \text{ is injective}}{\ast_{i<n}\, \mathsf{PE}(\hat{w}_i, \hat{v}_i) \Rightarrow \mathsf{PE}\big(\lambda \pi. f(\hat{w}_0\,\pi, \ldots, \hat{w}_{n-1}\,\pi),\ \lambda \pi. f(\hat{v}_0\,\pi, \ldots, \hat{v}_{n-1}\,\pi)\big)}$$

ProphEqz-ProphToken says that, out of a full prophecy token $[x]_1$, we can create a prophecy equalizer for $\lambda \pi.\, \pi\, x$ and any π-parametrized value $\hat{v}$; it follows from Proph-Resolve. ProphEqz-ProphObs says that a prophecy equalizer can be trivially obtained from a prophecy observation on the expected equality. ProphEqz-Modify allows us to modify the first argument of a prophecy equalizer using a prophecy observation. ProphEqz-Transform says that we can get a new prophecy equalizer out of prophecy equalizers, where both for the first and second arguments we construct the new value for the output by applying the same injective function f (which does not depend on π) to the values of the inputs.

Proof of ProphEqz-Transform. We satisfy the dependency precondition of each of the input prophecy equalizers by Dep-Destruct. Also, we split the partial prophecy token into n pieces to feed the prophecy equalizers. Then we compose the view shifts of them in parallel. Finally, we merge the prophecy observations and the partial prophecy tokens that were returned by them. □

Value Observer and Prophecy Control Now we introduce extra machinery for the unique reference type using the prophecy equalizer. First, we register the following RA UNQ to the global camera.

$$\textsc{unq} \ :=\ (x\colon \mathsf{ProphVar}) \xrightarrow{\mathsf{fin}} \textsc{fracown}\big((\mathsf{ProphAsn} \to x.\mathsf{type}) \times \mathbb{N}\big)$$

In this RA, we assign to a finite number of prophecy variables a value of the type $(\mathsf{ProphAsn} \to x.\mathsf{type}) \times \mathbb{N}$, under fractional ownership by FRACOWN. A value assigned to a prophecy variable is a pair $(\hat{v}, d)$ of a π-parametrized pure value $\hat{v}$ and a depth d. Using this RA, taking some fixed ghost name γ_{unq}, we introduce the following predicates, a *value observer* $\mathsf{VO}_x(\hat{v}, d)$ and a *prophecy control* $\mathsf{PC}(x, \hat{v}, d)$.

$$\mathsf{VO}_x(\hat{v}, d) \ :=\ \boxed{[x \leftarrow \mathsf{fown}_{1/2}(\hat{v}, d)]}^{\gamma_{\mathsf{unq}}}_{\mathrm{UNQ}}$$

$$\begin{aligned}\mathsf{PC}(x, \hat{v}, d) \ :=\ &\Big(\boxed{[x \leftarrow \mathsf{fown}_{1/2}(\hat{v}, d)]}^{\gamma_{\mathsf{unq}}}_{\mathrm{UNQ}} \ast [x]_1\Big) \ \vee \\ &\Big((\exists\, a.\, \boxed{[x \leftarrow \mathsf{fown}_1\, a]}^{\gamma_{\mathsf{unq}}}_{\mathrm{UNQ}}) \ast \mathsf{PE}(\lambda \pi.\, \pi\, x, \hat{v})\Big)\end{aligned}$$

A value observer $\mathsf{VO}_x(\hat{v}, d)$ witnesses with half ownership that $(\hat{v}, d)$ is assigned to x by the UNQ RA at γ_{unq}. A prophecy control $\mathsf{PC}(x, \hat{v}, d)$ either (i) witnesses with half ownership that $(\hat{v}, d)$ is assigned to x and owns the full prophecy token $[x]_1$ or (ii) has full ownership on the value assigned to x and owns a prophecy equalizer on $\lambda\pi.\, \pi\, x$ and $\hat{v}$. They satisfy the following properties.

$$\mathsf{timeless}\big(\mathsf{VO}_x(\hat{v}, d)\big) \qquad \textsc{ValObs-ProphCtrl-Timeless}\quad \mathsf{timeless}\big(\mathsf{VO}_x(\hat{v}, d) \mathrel{*} \mathsf{PC}(x, \hat{v}, d)\big)$$

$$\textsc{ValObs-ProphCtrl-Intro}\quad \mathsf{True} \Rrightarrow_{\varnothing} \exists x \text{ s.t. } x.\mathsf{type} = T.\ \mathsf{VO}_x(\hat{v}, d) \mathrel{*} \mathsf{PC}(x, \hat{v}, d)$$

$$\textsc{ValObs-ProphCtrl-Agree}\quad \mathsf{VO}_x(\hat{v}, d) \mathrel{*} \mathsf{PC}(x, \hat{v}', d') \Rightarrow \hat{v} = \hat{v}' \wedge d = d'$$

$$\textsc{ValObs-ProphCtrl-Update}\quad \mathsf{VO}_x(\hat{v}, d) \mathrel{*} \mathsf{PC}(x, \hat{v}, d) \Rrightarrow_{\mathcal{N}_{\mathsf{proph}}} \mathsf{VO}_x(\hat{v}', d') \mathrel{*} \mathsf{PC}(x, \hat{v}', d')$$

$$\textsc{ValObs-ProphCtrl-ProphToken}\quad \mathsf{VO}_x(\hat{v}, d) \mathrel{*} \mathsf{PC}(x, \hat{v}, d) \Rightarrow \mathsf{VO}_x(\hat{v}, d) \mathrel{*} [x]_1 \mathrel{*} \big([x]_1 \mathrel{-\!\!*} \mathsf{PC}(x, \hat{v}, d)\big)$$

$$\textsc{ValObs-ProphCtrl-Resolve}\quad \frac{\mathsf{Dep}(\hat{v}, Y)}{\mathsf{VO}_x(\hat{v}, d) \mathrel{*} \mathsf{PC}(x, \hat{v}, d) \mathrel{*} [Y]_q \Rrightarrow_{\mathcal{N}_{\mathsf{proph}}} \langle \pi.\, \pi\, x = \hat{v}\, \pi \rangle \mathrel{*} \mathsf{PC}(x, \hat{v}, d) \mathrel{*} [Y]_q}$$

$$\textsc{ValObs-ProphCtrl-Preresolve}\quad \frac{\mathsf{Dep}(\hat{w}, Y)}{\mathsf{VO}_x(\hat{v}, d) \mathrel{*} \mathsf{PC}(x, \hat{v}, d) \mathrel{*} [Y]_q \Rrightarrow_{\mathcal{N}_{\mathsf{proph}}} \langle \pi.\, \pi\, x = \hat{w}\, \pi \rangle \mathrel{*} [Y]_q \mathrel{*} \big(\forall \hat{v}', d'.\ \mathsf{PE}(\hat{w}, \hat{v}') \mathrel{-\!\!*} \mathsf{PC}(x, \hat{v}', d')\big)}$$

$$\textsc{ProphCtrl-PropheQz}\quad \mathsf{PC}(x, \hat{v}, d) \Rightarrow \mathsf{PE}(\lambda\pi.\, \pi\, x,\ \hat{v})$$

A value observer is timeless. A prophecy control itself is not timeless because of the view shift of the prophecy equalizer. Still, the separating conjunction of a prophecy control and a value observer is timeless (ValObs-ProphCtrl-Timeless).

ValObs-ProphCtrl-Intro says that, taking a fresh prophecy variable x, we can create a value observer $\mathsf{VO}_x(\hat{v}, d)$ and a prophecy control $\mathsf{PC}(x, \hat{v}, d)$. When we have a value observer and a prophecy control on the same prophecy variable, the values of the two agree (ValObs-ProphCtrl-Agree) and the values can be simultaneously updated (ValObs-ProphCtrl-Update). We can temporarily take out the full prophecy token out of a prophecy control as long as we have a value observer on the same prophecy variable (ValObs-ProphCtrl-ProphToken).

ValObs-ProphCtrl-Resolve says that, consuming a prophecy control and a value observer on a prophecy variable x, we can resolve x to a value $\hat{v}$ and reclaim a prophecy control with the value $\hat{v}$, with help of a partial prophecy token on a dependency Y of $\hat{v}$. We use this rule to resolve the prophecy variable when we release a unique reference (see SWkn-UnqRef-Release in § 6.3.3). ValObs-ProphCtrl-Preresolve is an advanced variant of the previous rule; in this lemma, we first resolve x to some value $\hat{w}$ and, after we get a prophecy equalizer $\mathsf{PE}(\hat{w}, \hat{v}')$ for some $\hat{v}'$, we achieve a prophecy control with the value $\hat{v}'$ and any depth d'. We use this rule for subdividing a unique reference (e.g., Rfn-Split-UnqRef-Pair-R in §6.3.4). ProphCtrl-PropheQz says that we can obtain a prophecy equalizer out of a prophecy control. We use this rule to reclaim a borrowed variable (see Wkn-UnqBor-PlnRef in §6.3.1 and SWkn-End-LocalLft-Reclaim in §6.1.2).

Proof of VALOBS-PROPHCTRL-TIMELESS. The left-hand disjunct of the prophecy control, the separating conjunction of $\boxed{[x \leftarrow \mathsf{fown}_{1/2}(\hat{v}, d)]}^{\gamma_{\mathsf{unq}}}_{\mathrm{UNQ}}$ and $[x]_1$, is timeless. The prophecy control cannot take the right-hand disjunct, because that side contains the full ownership token $\boxed{[x \leftarrow \mathsf{fown}_1(\hat{v}, d)]}^{\gamma_{\mathsf{unq}}}_{\mathrm{UNQ}}$, which is timeless and conflicts with the half ownership token of the value observer. □

Proof of VALOBS-PROPHCTRL-INTRO. We take a fresh prophecy variable x of the type T that has not been used *both* by the PROPH RA at γ_{proph} and by the UNQ RA at γ_{unq}. We obtain $[x]_1$ just like PROPHTOKEN-INTRO. We know that the type T is inhabited because we have $\hat{v}\colon \mathsf{ProphAsn} \to T$. We newly assign $(\hat{v}, d)$ to x in the UNQ RA to get a full ownership token $\boxed{[x \leftarrow \mathsf{fown}_1(\hat{v}, d)]}^{\gamma_{\mathsf{unq}}}_{\mathrm{UNQ}}$, which can be halved into two half ownership tokens. Therefore, we finally obtain $\mathsf{VO}_x(\hat{v}, d)$ and the right-hand disjunct of $\mathsf{PC}(x, \hat{v}, d)$. (We can use the rule GOWN-UPD-TWO for this update.) □

Proof of PROPH-RESOLVE. By agreement on the UNQ RA, since we have a value observer $\mathsf{VO}_x(\hat{v}, d)$, we know that the input prophecy control $\mathsf{PC}(x, \hat{v}, d)$ takes the *left-hand* disjunct. By consuming the full prophecy token $[x]_1$ of the prophecy control, by PROPH-RESOLVE, we resolve x into $\hat{v}$ to obtain a prophecy observation $\langle \pi.\ \pi\, x = \hat{v}\, \pi \rangle$. We obtain a prophecy control $\mathsf{PC}(x, \hat{v}, d)$ by constructing the *right-hand* disjunct: by consuming the half ownership token of the value observer, in the UNQ RA we change the value assigned to x into none and get the half ownership token for that; we get a prophecy equalizer from the prophecy observation we have just obtained by PROPHEQZ-PROPHOBS. □

Proof of VALOBS-PROPHCTRL-PRERESOLVE. First, like VALOBS-PROPHCTRL-RESOLVE, we consume the full prophecy token $[x]_1$ of the prophecy control to obtain a prophecy observation $\langle \pi.\ \pi\, x = \hat{w}\, \pi \rangle$. Also, out of two half ownership tokens of the value observer and the left-hand disjunct of the prophecy control, we get a full ownership token for the right-hand disjunct of a prophecy control. After we get a prophecy equalizer $\mathsf{PE}(\hat{w}, \hat{v}')$, by PROPHEQZ-MODIFY, conjoining it with the prophecy observation we have just obtained, we get a prophecy equalizer $\mathsf{PE}(\lambda\pi.\, \pi\, x, \hat{v}')$. Therefore, we can construct a prophecy control $\mathsf{PC}(x, \hat{v}', d')$. □

Proof of PROPHCTRL-PROPHEQZ. When the input prophecy control takes the left-hand disjunct, by PROPHEQZ-PROPHTOKEN we turn the full prophecy token $[x]_1$ into a prophecy equalizer. When the prophecy control takes the right-hand disjunct, we can just use the prophecy equalizer it has. □

Unique Reference Type Finally we define the semantic type of the unique reference type $\&^{\alpha}_{\mathsf{unq}}\, \tau$.

$$\lfloor \&^{\alpha}_{\mathsf{unq}}\, \tau \rfloor := \lfloor \tau \rfloor \times \lfloor \tau \rfloor \qquad |\&^{\alpha}_{\mathsf{unq}}\, \tau| := 1 \qquad \beta \sqsubseteq \&^{\alpha}_{\mathsf{unq}}\, \tau := \beta \sqsubseteq \alpha \wedge \triangleright\, \beta \sqsubseteq \tau$$

$$[\mathfrak{a}] \blacktriangleleft \&^{\alpha}_{\mathsf{unq}}\, \tau\, \{\hat{v}\}_d := d > 0 * \exists \mathfrak{l} \text{ s.t. } \mathfrak{l} = \mathfrak{a}.\ \exists x \text{ s.t. } (.1) \circ \hat{v} = \lambda\pi.\, \pi\, x.$$
$$\mathsf{VO}_x((.0) \circ \hat{v}, d-1) * \&^{\alpha}_{\mathsf{full}} \left(\exists \bar{\mathfrak{b}}, \hat{v}', d'.\ \mathfrak{l} \overset{1}{\mapsto} \bar{\mathfrak{b}} * \bar{\mathfrak{b}} \blacktriangleleft \tau\, \{\hat{v}'\}_{d'} * \rhd(d'+1) * \mathsf{PC}(x, \hat{v}', d')\right)$$

$$[\mathfrak{a}] \triangleleft^{\beta} \&^{\alpha}_{\mathsf{unq}}\, \tau\, \{\hat{v}\}_d := d > 0 * \exists \mathfrak{l} \text{ s.t. } \mathfrak{l} = \mathfrak{a}.\ \exists \bar{\mathfrak{b}}.\ \exists x \text{ s.t. } \mathrm{Dep}((.1) \circ \hat{v}, \{x\}).$$
$$\&^{\beta \sqcap \alpha}_{\mathsf{frac}}(\lambda q.\, [x]_q) * \circledast_i \&^{\beta \sqcap \alpha}_{\mathsf{frac}}(\lambda q.\, \mathfrak{l} + i \overset{q}{\mapsto} \bar{\mathfrak{b}}[i]) * \triangleright \left(\bar{\mathfrak{b}} \triangleleft^{\beta \sqcap \alpha} \tau\, \{(.0) \circ \hat{v}\}_{d-1}\right)$$

The ownership predicate $[\mathfrak{a}] \blacktriangleleft \&^{\alpha}_{\mathsf{unq}}\, \tau\, \{\hat{v}\}_d$ of $\&^{\alpha}_{\mathsf{unq}}\, \tau$ is quite tricky. The inner predicate of the full borrow is *existentially quantified* over the cell values $\bar{\mathfrak{b}}$, the π-parametrized value $\hat{v}'$ and the depth d' of the target object, because a unique reference

should be able to update the target object. In order to fix the inner parameters $\hat{v}'$ and d' to the expected values $(.0) \circ \hat{v}$ and $d-1$, the ownership predicate has a value observer $\mathrm{VO}_x((.0) \circ \hat{v}, d-1)$. The inner predicate of the full borrow has the points-to token $\mathfrak{l} \overset{1}{\mapsto} \bar{\mathfrak{b}}$, the ownership predicate on the target $\bar{\mathfrak{b}} \blacktriangleleft \tau\,\{\hat{v}'\}_{d'}$, the persistent time receipt $⧖(d'+1)$, and the prophecy control $\mathrm{PC}(x, \hat{v}', d')$. These objects are passed to the borrowee when the borrowee that the unique reference originates from reclaims the object. The borrowee obtains a prophecy observation $\langle \pi.\, \pi\, x = \hat{v}'\, \pi \rangle$ by consuming the returned prophecy control by ProphCtrl-ProphEqz and ProphEqz-Use. The full borrow introduces contractiveness and thus we set the depth d to 1 plus the depth of the target object.

In the sharing predicate of $\&^{\alpha}_{\mathrm{unq}}\,\tau$, the π-parametrized pure value and the depth of the target are simply fixed to $(.0) \circ \hat{v}$ and $d-1$, and the cell values $\bar{\mathfrak{b}}$ are existentially quantified outside borrows; the sharing predicate has fractured borrows on the prophecy token $[x]_q$ and on the memory ownership $\mathfrak{l} \overset{q}{\mapsto} \bar{\mathfrak{b}}$ as well as the sharing predicate on the target object. The value $(.1) \circ \hat{v}$ can be anything as long as it depends only on the prophecy variable x.

Proof of SemTy-Own-Shr of $\&^{\alpha}_{\mathrm{unq}}\,\tau$. Assume that the input is a full borrow under the lifetime β of the ownership predicate of $\&^{\alpha}_{\mathrm{unq}}\,\tau$, i.e., $\&^{\beta}_{\mathrm{full}}\big([\mathfrak{a}] \blacktriangleleft \&^{\alpha}_{\mathrm{unq}}\,\tau\,\{\hat{v}\}_d\big)$. After freeze by FullBor-Freeze and split by FullBor-Split, we obtain a full borrow of the full borrow of the ownership predicate of the form $\&^{\beta}_{\mathrm{full}}\,\&^{\alpha}_{\mathrm{full}}\cdots$, which we unnest it into $\&^{\beta\sqcap\alpha}_{\mathrm{full}}\cdots$ in one logical step by FullBor-Unnest. We freeze its inner parameters $\bar{\mathfrak{b}}$, $\hat{v}'$ and d' by FullBor-Freeze and split it by FullBor-Split to get full borrows of the points-to token $\&^{\beta\sqcap\alpha}_{\mathrm{full}}(\mathfrak{l} \overset{1}{\mapsto} \bar{\mathfrak{b}})$, of the ownership predicate of the target object $\&^{\beta\sqcap\alpha}_{\mathrm{full}}(\bar{\mathfrak{b}} \blacktriangleleft \tau\,\{\hat{v}'\}_{d'})$, and of the prophecy control $\&^{\beta\sqcap\alpha}_{\mathrm{full}}\,\mathrm{PC}(x, \hat{v}', d')$. We also have a full borrow on the value observer $\&^{\beta}_{\mathrm{full}}\,\mathrm{VO}_x((.0)\circ\hat{v},\ d-1)$. By agreement of the value observer and the prophecy control (ValObs-ProphCtrl-Agree), $\hat{v}'$ and d' turn out to be equal to $(.0) \circ \hat{v}$ and $d-1$.

We merge the full borrows on the value observer and the prophecy control by FullBor-Merge and then turn it into a full borrow on the full prophecy token $\&^{\beta\sqcap\alpha}_{\mathrm{full}}[x]_1$ by FullBor-Subdiv and ValObs-ProphCtrl-ProphToken, which is turned into a fractured borrow $\&^{\beta\sqcap\alpha}_{\mathrm{frac}}(\lambda q.\, [x]_q)$ by FullBor-FracBor. Also, the full borrow on the points-to token is turned into a fractured borrow $\ast_i\,\&^{\beta\sqcap\alpha}_{\mathrm{frac}}\big(\lambda q.\, \mathfrak{l} + i \overset{q}{\mapsto} \bar{\mathfrak{b}}[i]\big)$ by FullBor-Split FullBor-FracBor. The full borrow on the ownership predicate of the target object is transformed into the sharing predicate $\bar{\mathfrak{b}} \triangleleft^{\beta\sqcap\alpha} \tau\,\{(.0)\circ\hat{v}\}_{d-1}$ in $d-1$ logical steps by SemTy-Own-Shr on the target type τ. □

Proof of SemTy-Own-ProphToken of $\&^{\alpha}_{\mathrm{unq}}\,\tau$. We temporarily access the content of the full borrow of the ownership predicate of $\&^{\alpha}_{\mathrm{unq}}\,\tau$ by FullBor-Access, stripping off the later modality using one logical step. From the prophecy control $\mathrm{PC}(x, \hat{v}', d')$ we can temporarily get the full token $[x]_1$ by ValObs-ProphCtrl-ProphToken, with help of the value observer. Using $d-1$ logical steps, we obtain partial prophecy tokens on dependencies of $\hat{v}'$ out of the resource of the target $\bar{\mathfrak{b}} \blacktriangleleft \tau\,\{\hat{v}'\}_{d-1}$ by SemTy-Own-ProphToken of τ. □

Proof of SemTy-Shr-ProphToken on $\&^{\alpha}_{\mathrm{unq}}\,\tau$. Accessing the content of the fractured borrow $\&^{\beta\sqcap\alpha}_{\mathrm{frac}}(\lambda q.\, [x]_q)$ by FracBor-Access, we temporarily get $[x]_q$ for some q. We strip off the later modality of the target resource in one logical step and obtain expected partial prophecy tokens out of the resource $\bar{\mathfrak{b}} \triangleleft^{\beta\sqcap\alpha} \tau\,\{(.0)\circ\hat{v}\}_{d-1}$ in $d-1$ logical steps by SemTy-Shr-ProphToken on τ. □

5.4 Recursive Type

In this section, we introduce the *recursive type*. We support *recursion by contractiveness*, which includes contravariant and invariant recursion as well as covariant recursion.

We write $\mathsf{SemTy}_{T,n}$ for the subtype $\{\, \tau\colon \mathsf{SemTy} \mid \lfloor\tau\rfloor = T \land |\tau| = n \,\}$ for the semantics types of the pure type T and the size n.

The *recursive type* $\mathrm{rec}_{T,n}\,\sigma$ is defined for a pure type $T\colon \mathsf{Type}$, a size $n\colon \mathbb{N}$, and a *contractive* function over semantic types $\sigma\colon \mathsf{SemTy}_{T,n} \to \mathsf{SemTy}_{T,n}$. We introduce some notions for defining and using the recursive type.

Injection Type For convenience, we introduce the following semantic type $\mathsf{in}(\tau, f)$, which we dub the *injection type*, for any semantic type τ, pure type $T\colon \mathsf{Type}$ and injective function $f\colon \lfloor\tau\rfloor \to T$.

$$\lfloor \mathsf{in}(\tau,f)\rfloor \;:=\; T \qquad |\mathsf{in}(\tau,f)| \;:=\; |\tau| \qquad \alpha \sqsubseteq \mathsf{in}(\tau,f) \;:=\; \alpha \sqsubseteq \tau$$

$$\bar{\mathfrak{a}} \blacktriangleleft \mathsf{in}(\tau,f)\,\{\hat{v}\}_d \;:=\; \exists \hat{v}' \text{ s.t. } \hat{v} = f \circ \hat{v}'.\ \ \bar{\mathfrak{a}} \blacktriangleleft \tau\,\{\hat{v}'\}_d$$

$$\bar{\mathfrak{a}} \triangleleft^{\alpha} \mathsf{in}(\tau,f)\,\{\hat{v}\}_d \;:=\; \exists \hat{v}' \text{ s.t. } \hat{v} = f \circ \hat{v}'.\ \ \bar{\mathfrak{a}} \triangleleft^{\alpha} \tau\,\{\hat{v}'\}_d$$

SemTy-Own-ProphToken and SemTy-Shr-ProphToken on $\mathsf{in}(\tau, f)$ follow from the corresponding properties on τ, the injectivity of f, and Dep-Destruct.

The injection type satisfies the following rules on *type equality*.

TyEq-InjTy-Id
$\tau \;=\; \mathsf{in}(\tau, \mathsf{id})$

TyEq-InjTy-Comp
$\mathsf{in}(\tau,\, g \circ f) \;=\; \mathsf{in}(\mathsf{in}(\tau, f), g)$

TyEq-InjTy-Comp-Id
$$\frac{g \circ f = \mathsf{id}}{\tau \;=\; \mathsf{in}(\mathsf{in}(\tau, f), g)}$$

Non-expansiveness and Contractiveness We say a function over semantic types $\sigma\colon \mathsf{SemTy}_{T,n} \to \mathsf{SemTy}_{U,m}$ (for some T, n, U, m) is *non-expansive* and write $\mathsf{Nonex}(\sigma)$ if it is non-expansive as a function over the triple of the outliving, ownership and sharing predicates. We say σ is *contractive* and write $\mathsf{Contr}(\sigma)$ if it is contractive from that perspective.

We have the following rules on non-expansiveness and contractiveness.

Nonex-Id
$\mathsf{Nonex}(\mathsf{id})$

Contr-Const
$\mathsf{Contr}(\mathsf{const}\,\tau)$

Contr-Nonex
$$\frac{\mathsf{Contr}(\sigma)}{\mathsf{Nonex}(\sigma)}$$

Contr-BoxPtr
$$\frac{\mathsf{Nonex}(\sigma)}{\mathsf{Contr}(\mathsf{box} \circ \sigma)}$$

Contr-ShrRef
$$\frac{\mathsf{Nonex}(\sigma)}{\mathsf{Contr}(\&^{\alpha}_{\mathsf{shr}} \circ \sigma)}$$

Contr-UnqRef
$$\frac{\mathsf{Nonex}(\sigma)}{\mathsf{Contr}(\&^{\alpha}_{\mathsf{unq}} \circ \sigma)}$$

Contr-Vec
$$\frac{\mathsf{Nonex}(\sigma)}{\mathsf{Contr}(\mathsf{vec} \circ \sigma)}$$

Nonex-Pair
$$\frac{\forall\, i.\ \mathsf{Nonex}(\sigma_i)}{\mathsf{Nonex}(\lambda\tau.\ \sigma_0\,\tau \times \sigma_1\,\tau)}$$

Contr-Pair
$$\frac{\forall\, i.\ \mathsf{Contr}(\sigma_i)}{\mathsf{Contr}(\lambda\tau.\ \sigma_0\,\tau \times \sigma_1\,\tau)}$$

Nonex-Vrnt
$$\frac{\forall\, i.\ \mathsf{Nonex}(\sigma_i)}{\mathsf{Nonex}(\lambda\tau.\ \sigma_0\,\tau + \sigma_1\,\tau)}$$

Contr-Vrnt
$$\frac{\forall\, i.\ \mathsf{Contr}(\sigma_i)}{\mathsf{Contr}(\lambda\tau.\ \sigma_0\,\tau + \sigma_1\,\tau)}$$

Nonex-InjTy
$$\frac{\mathsf{Nonex}(\sigma)}{\mathsf{Nonex}(\lambda\tau.\ \mathsf{in}(\sigma\,\tau, f))}$$

Contr-InjTy
$$\frac{\mathsf{Contr}(\sigma)}{\mathsf{Contr}(\lambda\tau.\ \mathsf{in}(\sigma\,\tau, f))}$$

The pointer types $\mathsf{box}\,\tau$, $\&^{\alpha}_{\mathsf{shr}}\,\tau$, $\&^{\alpha}_{\mathsf{unq}}\,\tau$ and $\mathsf{vec}\,\tau$ are all contractive. For the unique reference type $\&^{\alpha}_{\mathsf{unq}}\,\tau$, the contractiveness comes from the contractiveness of the full borrow $\&^{\alpha}_{\mathsf{full}}$, as well as the later modality in the outliving and sharing predicates.

Recursive Type We define the *recursive type* $\mathsf{rec}_{T,n}\,\sigma$ for a pure type $T\colon \mathsf{Type}$, a size $n\colon \mathbb{N}$, and a *contractive* function over semantic types $\sigma\colon \mathsf{SemTy}_{T,n} \to \mathsf{SemTy}_{T,n}$ as the semantic type that satisfies the following recursive equation.

$$\mathsf{rec}_{T,n}\,\sigma \;:=\; \sigma\,(\mathsf{rec}_{T,n}\,\sigma)$$

The existence and uniqueness of the solution is ensured by the contractiveness of σ (by Banach's fixed point theorem Theorem 2.1). Note that $\mathsf{rec}_{T,n}\,\sigma$ is *equal* to $\sigma\,(\mathsf{rec}_{T,n}\,\sigma)$.

We introduce the following notation for convenience.

$$\mathsf{rec}_{T,n}(\sigma, f) \;:=\; \mathsf{rec}_{T,n}\left(\lambda\tau.\,\mathsf{in}(\sigma\,\tau, f)\right)$$

Also, we omit the size (i.e., use the notation $\mathsf{rec}_T\,\sigma$ and $\mathsf{rec}_T(\sigma, f)$) when the size information is clear.

The recursive type satisfies the following properties on *type equality*.

TyEq-Rec
$\mathsf{rec}_{T,n}\,\sigma \;=\; \sigma\,(\mathsf{rec}_{T,n}\,\sigma)$

TyEq-Rec-InjTy
$\mathsf{rec}_{T,n}(\sigma, f) \;=\; \mathsf{in}\big(\sigma\,(\mathsf{rec}_{T,n}(\sigma, f)),\, f\big)$

The recursive type satisfies the following non-expansiveness and contractiveness rules.

Nonex-Rec
$$\frac{\forall\,\tau'.\ \mathsf{Nonex}(\lambda\tau.\,\sigma\,\tau\,\tau')}{\mathsf{Nonex}\big(\lambda\tau.\ \mathsf{rec}_{T,n}(\sigma\,\tau)\big)}$$

Contr-Rec
$$\frac{\forall\,\tau'.\ \mathsf{Contr}(\lambda\tau.\,\sigma\,\tau\,\tau')}{\mathsf{Contr}\big(\lambda\tau.\ \mathsf{rec}_{T,n}(\sigma\,\tau)\big)}$$

Example Recursive Types We introduce and discuss a few example recursive types.

Example 5.1 (List Type). The semantic type of the *singly linked list type* $\mathsf{list}\,\tau$ can be defined as the following recursive type, using the list type $\mathsf{List}\,T$ of the metalogic.

$$\mathsf{list}\,\tau \;:=\; \mathsf{rec}_{\mathsf{List}\lfloor\tau\rfloor}\left(\lambda\tau'.\,\not{\mathsf{z}}_0 + \tau \times \mathsf{box}\,\tau',\ \mathsf{in}_{\mathsf{List}}\right)$$

Here, we define $\mathsf{in}_{\mathsf{List}}\colon \mathsf{Unit} + T \times \mathsf{List}\,T \to \mathsf{List}\,T$ as follows.

$$\mathsf{in}_{\mathsf{List}}(\mathsf{inj}_0()) \;:=\; \mathsf{nil} \qquad \mathsf{in}_{\mathsf{List}}(\mathsf{inj}_1(v, w)) \;:=\; v :: w$$

The contractiveness condition is satisfied because the self reference is under the box pointer box. The function $\mathsf{in}_{\mathsf{List}}$ has the following inverse $\mathsf{out}_{\mathsf{List}}$.

$$\mathsf{out}_{\mathsf{List}}\,\mathsf{nil} \;:=\; \mathsf{inj}_0() \qquad \mathsf{out}_{\mathsf{List}}(v :: w) \;:=\; \mathsf{inj}_1(v, w)$$

The following properties hold on $\mathsf{list}\,\tau$.

TyEq-List
$$\mathsf{list}\,\tau \;=\; \mathsf{in}\left(\not{\mathsf{z}}_0 + \tau \times \mathsf{box}\,\mathsf{list}\,\tau,\ \mathsf{in}_{\mathsf{List}}\right)$$

$$\lfloor\mathsf{list}\,\tau\rfloor \;=\; \mathsf{List}\,\lfloor\tau\rfloor \qquad |\mathsf{list}\,\tau| \;=\; |\tau| + 2 \qquad \alpha \sqsubseteq \mathsf{list}\,\tau \Leftrightarrow \alpha \sqsubseteq \tau$$

$$\bar{\mathfrak{a}} \blacktriangleleft \mathsf{list}\,\tau\,\{\hat{v}\}_d \;\Leftrightarrow\; \bar{\mathfrak{a}} \blacktriangleleft \not{\mathsf{z}}_0 + \tau \times \mathsf{box}\,\mathsf{list}\,\tau\,\{\mathsf{out}_{\mathsf{List}} \circ \hat{v}\}_d$$

$$\bar{\mathfrak{a}} \triangleleft^{\alpha} \mathsf{list}\,\tau\,\{\hat{v}\}_d \;\Leftrightarrow\; \bar{\mathfrak{a}} \triangleleft^{\alpha} \not{\mathsf{z}}_0 + \tau \times \mathsf{box}\,\mathsf{list}\,\tau\,\{\mathsf{out}_{\mathsf{List}} \circ \hat{v}\}_d$$

The property $\alpha \sqsubseteq \mathsf{list}\,\tau \Leftrightarrow \alpha \sqsubseteq \tau$ on the outliving predicate holds by Löb induction: let P be $\alpha \sqsubseteq \mathsf{list}\,\tau$; by definition, P is equivalent to $\alpha \sqsubseteq \tau \wedge \triangleright P$; so $\alpha \sqsubseteq \tau$ is equivalent to $\triangleright P \Rightarrow P$, which is equivalent to P by Löb induction (Löb).

Example 5.2 (Recursion Under the Unique Reference Type). We can also construct a recursive type with self reference under the unique reference type. For example, we have the following irregular Peano number type mnat_α whose self reference is under a unique reference instead of a box pointer.

$$\mathsf{mnat}_\alpha \;:=\; \mathsf{rec}_{\mathsf{BNat}}\left(\lambda\tau.\, \not{\iota}_0 + \&^{\alpha}_{\mathsf{unq}}\,\tau,\ \mathsf{in}_{\mathsf{BNat}}\right)$$

Here, the metalogic type BNat is the following inductive binary tree type without node data, having the data constructor $\mathsf{in}_{\mathsf{BNat}}$.

$$\mathsf{inductive}\ \ \mathsf{BNat} \;:=\; \mathsf{in}_{\mathsf{BNat}} : \mathsf{BNat} + \mathsf{BNat} \times \mathsf{BNat} \longrightarrow \mathsf{BNat}$$

Note that the unique reference type $\&^{\alpha}_{\mathsf{unq}}\,\tau$ is *not monotone* over τ, which makes it impossible to use the least fixed point of *covariant induction* to model recursive types like mnat_α.

Example 5.3 (Recursion Under the Recursive Type). We can construct a recursion type with *self reference under the recursive type*. For example, we can construct the following *rose tree* type $\mathsf{rose}\,\tau$.

$$\mathsf{rose}\,\tau \;:=\; \mathsf{rec}_{\mathsf{Rose}\lfloor\tau\rfloor}\left(\lambda\tau'.\, \tau \times \mathsf{box}\,\mathsf{list}\,\tau',\ \mathsf{in}_{\mathsf{Rose}}\right)$$

Here, the metalogic type $\mathsf{Rose}\,T$ is the following inductive type with the data constructor $\mathsf{in}_{\mathsf{Rose}}$.

$$\mathsf{inductive}\ \ \mathsf{Rose}\,T \;:=\; \mathsf{in}_{\mathsf{Rose}} : T \times \mathsf{List}\,\mathsf{Rose}\,T \longrightarrow \mathsf{Rose}\,T$$

The contractiveness of $\lambda\tau'.\, \tau \times \mathsf{box}\,\mathsf{list}\,\tau'$ follows from Nonex-InjTy and Nonex-Rec and Contr-BoxPtr.

Chapter 6

A Refined Type System for Verifying Rust Programs

In order to formalize and verify type-based translation of Rust programs into clean logic models in the style of RustHorn, we introduce and semantically model a type system for Rust that handles logic models for functional correctness, which we dub the *refined type system*. In the refined type system, we mainly use a typing judgment equipped with a logic model, which we call a *refined typing judgment*. The refined type system is designed as a composable and flexible deduction system. Although the typing rules are highly non-deterministic, once we get a deduction tree for typing, we can compute the logic model in a straightforward way. Thanks to the *semantic approach* built on the separation logic Iris, we can prove soundness of the refined type system just by checking each deduction rule semantically, instead of checking the global behavior of the system.

For simplicity and flexibility, we formalize the refined typing judgment *only semantically* as a persistent predicate and treat each deduction rule as a lemma on them, without introducing any syntactic structures for reasoning. We also do not provide algorithms for type checking. In modeling the refined typing judgments, we use *semantic types* (Chapter 5) to model type information, *lifetime logic* (Jung et al., 2018a) (explained in § 2.3) to handle lifetime information, and *prophecy observations* $\langle \pi.\ \phi_\pi \rangle$ to describe the precondition and postcondition handling prophecy information.

The logic model of each operation is formalized as a *(backward) predicate transformer*, a function that maps each postcondition to some precondition, which is a standard technique used by various verification tools such as F* (Swamy et al., 2016). For example, for the following expression of incrementing the target integer of a unique reference and then releasing the unique reference, we get the following refined typing judgment, where the last part is the predicate transformer (Example 6.3).

$$\alpha \Vdash \mathfrak{a} : \&^{\alpha}_{\mathsf{unq}} \mathsf{int} \vdash \mathfrak{a} \leftarrow *\mathfrak{a} + 1;\ \lightning \Vdash\!\Vdash \lambda \mathit{post}, ((n, n_\circ)).\ n_\circ = n + 1 \implies \mathit{post}\,()$$

Although the *input-output relation* is also a common representation for the postcondition used for verification methods like reduction to constrained Horn clauses (Grebenshchikov et al., 2012; Bjørner et al., 2015), we can lift an input-output relation into a predicate transformer in a natural way, as we see later in §6.1.2. We can translate Rust programs into CHCs in a fairly straightforward way using our refined type system, although we do not present a specific algorithm for that in this thesis for brevity.

In §6.1, we introduce the judgments used for the refined type system and basic rules on them. We also introduce the function type in § 6.1.4. In § 6.2, we introduce refined typing rules for basic types (excluding the unique reference type). In §6.3, we introduce refined typing rules for the unique reference type, giving detailed proofs for some of them. In § 6.4, we present some examples of verifying functional correctness of Rust programs using the refined typing rules. In § 6.5 we discuss some related work.

6.1 Judgments for the Refined Type System

We introduce the judgments used for the refined type system — the refined typing judgment and the weakening, super weakening, copyability, subtyping, and access judgments. We also introduce some basic rules on the judgments. These judgments are *semantically* modeled as persistent predicates in the separation logic Iris. We also introduce here the function type, which depends on the refined typing judgment.

6.1.1 Refined Typing Judgment and Its Variants

Variants of the Ownership Predicate We introduce the following variant of the ownership predicate which *hides the depth information*.

$$\bar{\mathfrak{a}} \blacktriangleleft \tau\,\{\hat{v}\} \;:=\; \exists d.\, \bar{\mathfrak{a}} \blacktriangleleft \tau\,\{\hat{v}\}_d \ast \mathord{\text{⧖}} d$$

Here, we can use any depth d for the ownership predicate $\bar{\mathfrak{a}} \blacktriangleleft \tau\,\{\hat{v}\}_d$ as long as we have a persistent time receipt $\mathord{\text{⧖}} d$, which witnesses that at least d physical steps have been passed.

We also introduce the following predicate that represents the *resource for an object which is borrowed under the lifetime α*.

$$\bar{\mathfrak{a}} \blacktriangleleft^{\dagger\alpha} \tau\,\{\hat{v}\} \;:=\; [\dagger\alpha] \Rrightarrow\!\!\ast_{\mathcal{N}_{\mathsf{lft}}} \exists \hat{v}'.\, \bar{\mathfrak{a}} \blacktriangleleft \tau\,\{\hat{v}'\} \ast \triangleright\, \mathsf{PE}(\hat{v}, \hat{v}')$$

After we know that the lifetime α has ended, with update in one logical step, we get the resource of the ownership predicate with some value $\hat{v}'$ and a prophecy equalizer under later $\triangleright\, \mathsf{PE}(\hat{v}, \hat{v}')$. If we can temporarily access a relevant partial prophecy token by SemTy-Own-ProphToken, consuming the prophecy equalizer we can get a prophecy observation $\langle \pi.\, \hat{v}\,\pi = \hat{v}'\,\pi \rangle$ by PropheQz-Use.

We introduce the notion of an *activity mode act*: Act, which is of the form ! (unborrowed) or $\dagger\alpha$ (borrowed under the lifetime α: Lft). We define the following variant of the ownership predicate which is parametrized over an activity mode.

$$\bar{\mathfrak{a}} \blacktriangleleft^{act} \tau\,\{\hat{v}\} \;:=\; \begin{cases} \bar{\mathfrak{a}} \blacktriangleleft \tau\,\{\hat{v}\} & (act = !) \\ \bar{\mathfrak{a}} \blacktriangleleft^{\dagger\alpha} \tau\,\{\hat{v}\} & (act = \dagger\alpha) \end{cases}$$

Type and Lifetime Contexts A *type context* Γ is a sequence of items of the form $\mathfrak{a} :^{act} \tau$, which represents an object of a cell value $\mathfrak{a}$: CellVal, an activity mode *act*: Act, and a type τ: SemTy_1. Here, SemTy_1 denotes the subtype $\{\, \tau\colon \mathsf{SemTy} \;/\; |\tau| = 1 \,\}$, the type for a semantic type whose size is 1. For an item of a type context, we just write $\mathfrak{a}\colon \tau$ for $\mathfrak{a} :^{!} \tau$ (the case $act = !$). Roughly speaking, a type context represents a sequence of 'local variables', but each 'local variable' is not given an explicit name.

A *lifetime context* Δ is simply a sequence of lifetimes $\vec{\alpha}$, each of which represents a *local lifetime*, a lifetime such that we know it is alive and we have the right to end it. The Iris proposition $\lceil \Delta \rceil$ for a lifetime context Δ is defined as follows.

$$\lceil \vec{\alpha} \rceil \;:=\; \ast_i \left([\alpha_i]_1 \ast ([\alpha_i]_1 \Rrightarrow_{\mathcal{N}_{\mathsf{lft}}} [\dagger\alpha_i]) \right)$$

For each lifetime α_i, it has (i) the full token $[\alpha_i]_1$ and (ii) the step-taking view shift $[\alpha_i]_1 \Rrightarrow_{\mathcal{N}_{\mathsf{lft}}} [\dagger\alpha_i]$ which can end the lifetime α_i by consuming the full token (see also the rule Lft-Intro).

For a lifetime α and a lifetime context $\Delta = \vec{\beta}$, we write $\sqcap(\alpha, \Delta)$ for the intersection of $\alpha, \vec{\beta}$.

Refined Typing Judgment Now we introduce the *refined typing judgment*, which has the following form.

$$\alpha \mid \Delta; \Gamma \vdash e : \tau \mid \Delta'; \Gamma' \mid \mathit{pre}$$

It specifies the effect of executing the expression e. The input objects, i.e., the objects we have before the execution of e, are represented by Γ. The immediate output object of the expression e has the type τ. The remaining output objects are represented by Γ'.

The lifetime contexts Δ, Δ' represent the set of local lifetimes we have before and after the execution of e. The lifetime α at the head is the lifetime that remains alive during the execution of e aside from the local lifetimes. Unlike RustBelt (Jung et al., 2018a,b), we omit information about the outliving order $\alpha \sqsubseteq \beta$ over lifetimes from this judgment; for reasoning about the outliving order, we piggyback the inference rules of the lifetime logic and Iris.

The last part *pre* represents the specification of the operation in terms of a *predicate transformer*, whose first argument is the postcondition on the pure values of the output objects and whose second argument is the tuple of the pure values of the input objects. If we have $\Gamma = \overrightarrow{\mathfrak{a} :^{\mathit{act}} \tau'}$ and $\Gamma' = \overrightarrow{\mathfrak{b} :^{\mathit{act}} \tau''}$, the predicate transformer *pre* has the following type.

$$\mathit{pre} : \left(\times(\lfloor \tau \rfloor, \overrightarrow{\lfloor \tau'' \rfloor}) \to \mathsf{Prop}\right) \to \times(\overrightarrow{\lfloor \tau' \rfloor}) \to \mathsf{Prop}$$

Here, $\times(T_0, \dots, T_{n-1})$ denotes the tuple type $T_0 \times \cdots \times T_{n-1}$.

The refined typing judgment is defined as the following persistent predicate.

$$\begin{aligned}
&\alpha \mid \Delta; \overrightarrow{\mathfrak{a} :^{\mathit{act}} \tau'} \vdash e : \tau \mid \Delta'; \overrightarrow{\mathfrak{b} :^{\mathit{act}'} \tau''} \mid \mathit{pre} \quad := \quad \forall\, \mathit{post}, q.\\
&\left\{ \exists \vec{\hat{v}}.\ \left\langle \pi.\ \mathit{pre}\, \mathit{post}\, (\overrightarrow{\hat{v}\, \pi}) \right\rangle * \mathop{\Large *}_i \left([\mathfrak{a}_i] \blacktriangleleft^{\mathit{act}_i} \tau_i' \{\hat{v}_i\} \right) * \lfloor \Delta \rfloor * [\alpha]_q \right\}\ e\ \left\{ \mathfrak{c}.\ \exists \hat{w}, \overrightarrow{\hat{v}'}.\right.\\
&\qquad \left. \left\langle \pi.\ \mathit{post}\, (\hat{w}\, \pi, \overrightarrow{\hat{v}'\, \pi}) \right\rangle * [\mathfrak{c}] \blacktriangleleft \tau \{\hat{w}\} * \mathop{\Large *}_j \left([\mathfrak{b}_j] \blacktriangleleft^{\mathit{act}'_j} \tau_j'' \{\hat{v}_j'\} \right) * \lfloor \Delta' \rfloor * [\alpha]_q \right\}_{\top}
\end{aligned}$$

The Hoare triple is universally quantified over the postcondition *post*, as well as the fraction q for the lifetime α. The input objects have the π-*parametrized* pure values $\hat{v}_i$; we know that they satisfy the precondition *pre post* for *any valid* π through the *prophecy observation* $\left\langle \pi.\ \mathit{pre}\, \mathit{post}\, (\overrightarrow{\hat{v}\, \pi}) \right\rangle$. The output object have the π-parametrized pure values $\hat{w}, \overrightarrow{\hat{v}'}$, where $\hat{w}$ is for the immediate output and $\overrightarrow{\hat{v}'}$ are for the rest; we know that they satisfy the postcondition for *any valid* π through the prophecy observation $\left\langle \pi.\ \mathit{post}\, (\hat{w}\, \pi, \overrightarrow{\hat{v}'\, \pi}) \right\rangle$.

We also use the following variant of the refined typing judgment that ignores the immediate output of the expression e.

$$\begin{aligned}
&\alpha \mid \Delta; \Gamma \vdash e \mid \Delta'; \Gamma' \mid \mathit{pre} \quad :=\\
&\qquad \alpha \mid \Delta; \Gamma \vdash e : \lightning_1 \mid \Delta'; \Gamma' \mid \mathit{pre} \circ \left(\lambda \mathit{post}, (\vec{v}).\ \mathit{post}\, ((), \vec{v})\right)
\end{aligned}$$

Here, the function $\lambda \mathit{post}, (\vec{v}).\ \mathit{post}\, ((), \vec{v})$ modifies the postcondition.

For a refined typing judgment, we can omit the parts 'Δ;' and 'Δ';' if the input and output lifetime contexts Δ, Δ' are empty, and we can omit the first part $\alpha \mid$ if α is ∞.

The refined typing judgment satisfies the following adequacy theorem on integers.

Theorem 6.1 (Adequacy of the Refined Typing Judgment on Integers)**.** *The following is a tautology for any expression e, predicate transformer pre and postcondition post.*

$$\begin{aligned}
&\overrightarrow{\mathfrak{a} : \mathsf{int}} \vdash e : \mathsf{int} \mid\mid \mathit{pre} \quad \Rightarrow\\
&\qquad \left\{ \exists \vec{m}\ \mathit{s.t.}\ \forall i.\, m_i = \mathfrak{a}_i.\ \mathit{pre}\, \mathit{post}\, (\vec{m}) \right\}\ e\ \left\{ \mathfrak{b}.\ \exists n\ \mathit{s.t.}\ n = \mathfrak{b}.\ \mathit{post}\, n \right\}_{\top}
\end{aligned}$$

Proof. By the definition of the integer type int (§5.2) and the rule ProphObs-Fact. □

Weakening and Super Weakening Judgments We also introduce the following *weakening judgment*, which amounts to the refined typing judgment *without execution*.

$$\alpha \mid \Delta;\, \Gamma \vdash \Delta';\, \Gamma' \mid \mathit{pre}$$

We use a *predicate transformer*, instead of just a function over pure values, which allows us to impose a *precondition* on the weakening operation; it is later effectively used in the rules like Wkn-PlnRef-Vrnt-Triple. The weakening judgment is defined as follows, in a way analogous to the definition of the refined typing judgment.

$$\begin{aligned}\alpha \mid \Delta;\, \overrightarrow{\mathfrak{a} :^{\mathit{act}} \tau'} \vdash \Delta';\, \overrightarrow{\mathfrak{b} :^{\mathit{act}'} \tau''} \mid \mathit{pre} \quad := \quad & \forall\, \mathit{post},\, q.\\ & \left(\exists \vec{\hat{v}}.\ \left\langle \pi.\ \mathit{pre}\, \mathit{post}\, (\overrightarrow{\hat{v}\, \pi}) \right\rangle * \mathbin{\ast}_i \left([\mathfrak{a}_i] \blacktriangleleft^{\mathit{act}_i} \tau_i\, \{\hat{v}_i\}\right) * \lceil \Delta \rceil * [\alpha]_q\right) \quad \Rrightarrow_{\top}\\ & \quad \exists\, \hat{w}, \vec{\hat{v}'}.\ \left\langle \pi.\ \mathit{post}\, (\hat{w}\, \pi, \overrightarrow{\hat{v}'\, \pi}) \right\rangle * \mathbin{\ast}_j \left([\mathfrak{b}_j] \blacktriangleleft^{\mathit{act}'_j} \tau''_j\, \{\hat{v}'_j\}\right) * \lceil \Delta' \rceil * [\alpha]_q\end{aligned}$$

We also introduce the following *super weakening judgment*.

$$\alpha \mid \Delta;\, \Gamma \vdash^{\#} \Delta';\, \Gamma' \mid \mathit{pre}$$

It is defined as follows.

$$\begin{aligned}\alpha \mid \Delta;\, \overrightarrow{\mathfrak{a} :^{\mathit{act}} \tau'} \vdash \Delta';\, \overrightarrow{\mathfrak{b} :^{\mathit{act}'} \tau''} \mid \mathit{pre} \quad := \quad & \forall\, \mathit{post},\, q.\\ & \left(\exists \vec{\hat{v}}.\ \left\langle \pi.\ \mathit{pre}\, \mathit{post}\, (\overrightarrow{\hat{v}\, \pi}) \right\rangle * \mathbin{\ast}_i \left([\mathfrak{a}_i] \blacktriangleleft^{\mathit{act}_i} \tau_i\, \{\hat{v}_i\}\right) * \lceil \Delta \rceil * [\alpha]_q\right) \quad \Rrightarrow^{\#}_{\top}\\ & \quad \exists\, \hat{w}, \vec{\hat{v}'}.\ \left\langle \pi.\ \mathit{post}\, (\hat{w}\, \pi, \overrightarrow{\hat{v}'\, \pi}) \right\rangle * \mathbin{\ast}_j \left([\mathfrak{b}_j] \blacktriangleleft^{\mathit{act}'_j} \tau''_j\, \{\hat{v}'_j\}\right) * \lceil \Delta' \rceil * [\alpha]_q\end{aligned}$$

Unlike the weakening judgment, we can use a *super fancy update*. We can perform a super weakening judgment on let binding (Rfn-Let) and sequential execution (Rfn-Seq).

For weakening and super weakening judgments, we can omit the parts 'Δ;' and 'Δ';' if the input and output lifetime contexts Δ, Δ' are empty, and we can omit the first part $\alpha \mid$ if α is ∞.

6.1.2 Basic Deduction Rules

Structural Rules We can modify a refined typing judgment both on the input and output sides using weakening judgments.

Rfn-Wkn
$$\frac{\alpha \mid \Delta_0;\, \Gamma_0 \vdash e : \tau \mid \Delta_1;\, \Gamma_1 \mid \mathit{pre} \qquad \alpha \mid \Delta'_0;\, \Gamma'_0 \vdash \Delta_0;\, \Gamma_0 \mid \mathit{pre}_0 \qquad \forall\, \mathfrak{a}.\ \alpha \mid \Delta_1;\, \Gamma_1 \vdash \Delta'_1;\, \Gamma'_1 \mid \mathit{pre}_1}{\alpha \mid \Delta'_0;\, \Gamma'_0 \vdash e : \tau \mid \Delta'_1;\, \Gamma'_1 \mid \mathit{pre}_0 \circ \mathit{pre} \circ \mathit{pre}_1}$$

We compose the predicate transformers of the three judgments. Likewise, we can modify a super weakening judgment by weakening judgments.

SWkn-Wkn
$$\frac{\alpha \mid \Delta_0;\, \Gamma_0 \vdash^{\#} \Delta_1;\, \Gamma_1 \mid \mathit{pre} \qquad \alpha \mid \Delta'_0;\, \Gamma'_0 \vdash \Delta_0;\, \Gamma_0 \mid \mathit{pre}_0 \qquad \forall\, \mathfrak{a}.\ \alpha \mid \Delta_1;\, \Gamma_1 \vdash \Delta'_1;\, \Gamma'_1 \mid \mathit{pre}_1}{\alpha \mid \Delta'_0;\, \Gamma'_0 \vdash^{\#} \Delta'_1;\, \Gamma'_1 \mid \mathit{pre}_0 \circ \mathit{pre} \circ \mathit{pre}_1}$$

Also, weakening judgments can be composed.

Wkn-Comp
$$\frac{\alpha \mid \Delta;\, \Gamma \vdash \Delta';\, \Gamma' \mid \mathit{pre} \qquad \alpha \mid \Delta';\, \Gamma' \vdash \Delta'';\, \Gamma'' \mid \mathit{pre}'}{\alpha \mid \Delta;\, \Gamma \vdash \Delta'';\, \Gamma'' \mid \mathit{pre} \circ \mathit{pre}'}$$

We have the *frame rules* on the refined typing, weakening and super weakening judgments.

RFN-FRAME
$$\frac{\alpha \mid \Delta; \Gamma \vdash e: \tau \mid \Delta'; \Gamma' \mid pre}{\alpha \mid \Delta; \mathfrak{a} :^{act} \tau, \Gamma \vdash e: \tau \mid \Delta; \mathfrak{a} :^{act} \tau, \Gamma' \mid \lambda post, (v, \vec{w}).\ pre\left(\lambda(w', \vec{w''}).\ post\,(w', v, \vec{w''})\right)(\vec{w})}$$

WKN-FRAME
$$\frac{\alpha \mid \Delta; \Gamma \vdash \Delta'; \Gamma' \mid pre}{\alpha \mid \Delta; \mathfrak{a} :^{act} \tau, \Gamma \vdash \Delta'; \mathfrak{a} :^{act} \tau, \Gamma' \mid \lambda post, (v, \vec{w}).\ pre\left(\lambda(\vec{w'}).\ post\,(v, \vec{w'})\right)(\vec{w})}$$

SWKN-FRAME
$$\frac{\alpha \mid \Delta; \Gamma \vdash \Delta'; \Gamma' \mid pre}{\alpha \mid \Delta; \mathfrak{a} :^{act} \tau, \Gamma \vdash^{\#} \Delta'; \mathfrak{a} :^{act} \tau, \Gamma' \mid \lambda post, (v, \vec{w}).\ pre\left(\lambda(\vec{w'}).\ post\,(v, \vec{w'})\right)(\vec{w})}$$

We can move a local lifetime α to the head lifetime if we do not end α during the execution.

RFN-LFT-LOCAL-HEAD
$$\frac{\beta \sqcap \alpha \mid \Delta; \Gamma \vdash e: \tau \mid \Delta'; \Gamma' \mid pre}{\alpha \mid \beta, \Delta; \Gamma \vdash e: \tau \mid \beta, \Delta'; \Gamma' \mid pre}$$

WKN-LFT-LOCAL-HEAD
$$\frac{\beta \sqcap \alpha \mid \Delta; \Gamma \vdash \Delta'; \Gamma' \mid pre}{\alpha \mid \beta, \Delta; \Gamma \vdash \beta, \Delta'; \Gamma' \mid pre}$$

SWKN-LFT-LOCAL-HEAD
$$\frac{\beta \sqcap \alpha \mid \Delta; \Gamma \vdash^{\#} \Delta'; \Gamma' \mid pre}{\alpha \mid \beta, \Delta; \Gamma \vdash^{\#} \beta, \Delta'; \Gamma' \mid pre}$$

We can modify the head lifetime and the predicate transformer.

RFN-MONO
$$\frac{\alpha \mid \Delta; \Gamma \vdash e: \tau \mid \Delta'; \Gamma' \mid pre \qquad pre \dot{\Rightarrow} pre' \qquad \beta \sqsubseteq \alpha}{\beta \mid \Delta; \Gamma \vdash e: \tau \mid \Delta'; \Gamma' \mid pre'}$$

WKN-MONO
$$\frac{\alpha \mid \Delta; \Gamma \vdash \Delta'; \Gamma' \mid pre \qquad pre \dot{\Rightarrow} pre' \qquad \beta \sqsubseteq \alpha}{\beta \mid \Delta; \Gamma \vdash \Delta'; \Gamma' \mid pre'}$$

SWKN-MONO
$$\frac{\alpha \mid \Delta; \Gamma \vdash^{\#} \Delta'; \Gamma' \mid pre \qquad pre \dot{\Rightarrow} pre' \qquad \beta \sqsubseteq \alpha}{\beta \mid \Delta; \Gamma \vdash^{\#} \Delta'; \Gamma' \mid pre'}$$

Here, the relation $pre' \dot{\Rightarrow} pre$ denotes the following pointwise implication.

$$pre' \dot{\Rightarrow} pre \quad := \quad \forall\, post, (\vec{v}).\ pre'\, post\,(\vec{v}) \Rightarrow pre\, post\,(\vec{v})$$

We can freely permute the type context, accordingly modifying the predicate transformer, by using the following rule as well as WKN-FRAME and WKN-COMP a number of times.

WKN-SWAP
$$\mathfrak{a} :^{act} \tau,\ \mathfrak{b} :^{act'} \tau',\ \Gamma \vdash \mathfrak{b} :^{act'} \tau',\ \mathfrak{a} :^{act} \tau,\ \Gamma \mid \lambda post, (v, v', \vec{w}).\ post\,(v', v, \vec{w})$$

We can also permute the lifetime context.

WKN-LFT-PERMUTE
$$\frac{\Delta' \text{ is a permutation of } \Delta}{\Delta; \vdash \Delta'; \mid \mathsf{id}}$$

We can throw away an object in the type context.

$$\textsc{Wkn-Leak}\quad \mathfrak{a} :^{\mathit{acct}} \tau \vdash \mathsf{I}\ \lambda \mathit{post}, (v).\ \mathit{post}\ ()$$

We can modify the lifetime limit of a borrowed object in the type context.

$$\textsc{Wkn-Mono-BorLft}\quad \frac{\alpha \sqsubseteq \beta}{\mathfrak{a} :^{\dagger\alpha} \tau \vdash \mathfrak{a} :^{\dagger\beta} \tau \mathrel{\mathsf{I}} \mathsf{id}}$$

We have the reflexive super weakening judgment.

$$\textsc{SWkn-Refl}\quad \alpha \mathrel{\mathsf{I}} \Delta; \Gamma \vdash^{\#} \Delta; \Gamma \mathrel{\mathsf{I}} \mathsf{id}$$

Binding We have the following binding rule based on an evaluation context.

$$\textsc{Rfn-EvCtx}\quad \frac{\alpha \mathrel{\mathsf{I}} \Delta; \Gamma \vdash e : \tau \mathrel{\mathsf{I}} \Delta'; \Gamma' \mathrel{\mathsf{I}} \mathit{pre} \qquad \forall \mathfrak{a}.\ \alpha \mathrel{\mathsf{I}} \Delta'; \mathfrak{a} : \tau, \Gamma' \vdash K[\mathfrak{a}] : \tau' \mathrel{\mathsf{I}} \Delta''; \Gamma'' \mathrel{\mathsf{I}} \mathit{pre}'}{\alpha \mathrel{\mathsf{I}} \Delta; \Gamma \vdash K[e] : \tau' \mathrel{\mathsf{I}} \Delta''; \Gamma'' \mathrel{\mathsf{I}} \mathit{pre} \circ \mathit{pre}'}$$

The cell value we get from e is *universally quantified* under the name $\mathfrak{a}$. Note that the output type context Γ'' cannot depend on the value $\mathfrak{a}$. The resulting predicate transformer $\mathit{pre} \circ \mathit{pre}'$ is simply the composite of the given predicate transformers $\mathit{pre}, \mathit{pre}'$.

Based on Rfn-EvCtx, we can introduce the following binding rule for let binding $\mathsf{let}\ a = e\ \mathsf{in}\ e'$.

$$\textsc{Rfn-Let}\quad \frac{\begin{array}{l}\alpha \mathrel{\mathsf{I}} \Delta; \Gamma \vdash e : \tau \mathrel{\mathsf{I}} \Delta'; \Gamma' \mathrel{\mathsf{I}} \mathit{pre} \qquad \forall \mathfrak{a}.\ \alpha \mathrel{\mathsf{I}} \Delta'; \mathfrak{a} : \tau, \Gamma' \vdash^{\#} \Delta^+; \Gamma^+_{\mathfrak{a}} \mathrel{\mathsf{I}} \mathit{pre}^+ \\ \quad \Box \left(P \Rightarrow \forall \mathfrak{a}.\ \alpha \mathrel{\mathsf{I}} \Delta^+; \Gamma^+_{\mathfrak{a}} \vdash e'[\mathfrak{a}/a] : \tau' \mathrel{\mathsf{I}} \Delta''; \Gamma'' \mathrel{\mathsf{I}} \mathit{pre}' \right) \qquad \triangleright P\end{array}}{\alpha \mathrel{\mathsf{I}} \Delta; \Gamma \vdash \mathsf{let}\ a = e\ \mathsf{in}\ e' : \tau' \mathrel{\mathsf{I}} \Delta''; \Gamma'' \mathrel{\mathsf{I}} \mathit{pre} \circ \mathit{pre}^+ \circ \mathit{pre}'}$$

We replace the program variable a in the expression e' with the evaluation result $\mathfrak{a}$ of e. Using the physical step for let binding, we can perform the *super fancy update* of a *super weakening judgment*. We can also strip off the later modality on any (persistent) assumption P, which can be used for strengthening the outliving assumptions on a function (e.g., strengthening $\alpha \sqsubseteq \mathsf{box}\,\tau$ into $\alpha \sqsubseteq \tau$).

Since sequential execution $e; e'$ is defined as $\mathsf{let}\ \_ = e\ \mathsf{in}\ e'$, we can derive the following rule for sequential execution from Rfn-Let.

$$\textsc{Rfn-Seq}\quad \frac{\begin{array}{l}\alpha \mathrel{\mathsf{I}} \Delta; \Gamma \vdash e \mathrel{\mathsf{I}} \Delta'; \Gamma' \mathrel{\mathsf{I}} \mathit{pre} \qquad \alpha \mathrel{\mathsf{I}} \Delta'; \Gamma' \vdash^{\#} \Delta^+; \Gamma^+ \mathrel{\mathsf{I}} \mathit{pre}^+ \\ \quad \Box \left(P \Rightarrow \alpha \mathrel{\mathsf{I}} \Delta^+; \Gamma^+ \vdash e' : \tau' \mathrel{\mathsf{I}} \Delta''; \Gamma'' \mathrel{\mathsf{I}} \mathit{pre}' \right) \qquad \triangleright P\end{array}}{\alpha \mathrel{\mathsf{I}} \Delta; \Gamma \vdash e; e' : \tau' \mathrel{\mathsf{I}} \Delta''; \Gamma'' \mathrel{\mathsf{I}} \mathit{pre} \circ \mathit{pre}^+ \circ \mathit{pre}'}$$

Concurrency We have the following refined typing rule for the concurrent execution $e \parallel e'$.

$$\textsc{Rfn-Concur}\quad \frac{\alpha \mathrel{\mathsf{I}} \Delta; \Gamma \vdash e : \tau \mathrel{\mathsf{I}} \Delta_+; \Gamma_+ \mathrel{\mathsf{I}} \mathit{pre} \qquad \alpha \mathrel{\mathsf{I}} \Delta'; \Gamma' \vdash e' \mathrel{\mathsf{I}} \Delta'_+; \Gamma'_+ \mathrel{\mathsf{I}} \mathit{pre}'}{\begin{array}{l}\alpha \mathrel{\mathsf{I}} \Delta, \Delta'; \Gamma, \Gamma' \vdash e \parallel e' : \tau \mathrel{\mathsf{I}} \Delta_+, \Delta'_+; \Gamma_+, \Gamma'_+ \mathrel{\mathsf{I}} \lambda \mathit{post}, (\vec{v}, \vec{v'}). \\ \quad \mathit{pre}\left(\lambda(\vec{w}).\ \forall \vec{w'}.\ \mathit{post}\,(\vec{w}, \vec{w'})\right)(\vec{v}) \ \wedge\ \mathit{pre}'\left(\lambda(\vec{w'}).\ \forall \vec{w}.\ \mathit{post}\,(\vec{w}, \vec{w'})\right)(\vec{v'})\end{array}}$$

When we have the postcondition post on the outputs of both e and e', we can let the postcondition on the outputs of e be $\lambda(\vec{w}).\ \forall \vec{w'}.\ \mathit{post}\,(\vec{w}, \vec{w'})$ and of e' be $\lambda(\vec{w'}).\ \forall \vec{w}.\ \mathit{post}\,(\vec{w}, \vec{w'})$. We can prove this rule using Hoare-Concur.

Manipulating Local Lifetimes We have the following rule for *introducing a new local lifetime.*

$$\textsc{Rfn-Intro-LocalLft}\quad \frac{\forall\, \alpha.\ \ \beta \mid \alpha, \Delta; \Gamma \vdash e : \tau \mid \Delta'; \Gamma' \mid \mathit{pre}}{\beta \mid \Delta; \Gamma \vdash e : \tau \mid \Delta'; \Gamma' \mid \mathit{pre}}$$

This rule follows from Lft-Intro, which acquires a full lifetime token $[\alpha]_1$ and a step-taking view shift $[\alpha]_1 \Rrightarrow_{\top} [\dagger\alpha]$ for *some* lifetime α. To eliminate existential quantification over α, we need the universal quantifier over the lifetime α in the assumption of the rule Rfn-Intro-LocalLft. The lifetime and type contexts $\Delta, \Delta', \Gamma, \Gamma'$ and the type τ cannot depend on the lifetime α; in particular, Δ' cannot contain α.

We can borrow objects using local lifetimes. Later we introduce the rules for borrowing (e.g., SWkn-ShrBor-BoxPtr, Wkn-UnqBor-BoxPtr, Wkn-UnqBor-UnqRef).

For a local lifetime α, we have a full lifetime token $[\alpha]_1$ and a *step-taking* view shift $[\alpha]_1 \Rrightarrow_{\mathcal{N}_{\mathsf{lft}}} [\dagger\alpha]$.

After we get $[\dagger\alpha]$, we can reclaim a borrowed object using the following lemma.

$$\textsc{Raw-Reclaim}\quad \frac{[\dagger\alpha] \qquad \beta \sqsubseteq \tau}{[\mathfrak{a}] \blacktriangleleft^{\dagger\alpha} \tau\,\{\hat{v}\} \ast [\beta]_q \ \Rrightarrow_{\mathcal{N}_{\mathsf{lft}}} \exists\, d.\ \ \mathord{\text{⧖}} d \ \ast \ \Rrightarrow^{d+1}_{\mathcal{N}_{\mathsf{lft}}+\mathcal{N}_{\mathsf{proph}}} \big(\exists\, \hat{v}'.\ [\mathfrak{a}] \blacktriangleleft \tau\,\{\hat{v}'\} \ast \langle \pi.\, \hat{v}\,\pi = \hat{v}'\,\pi \rangle\big)}$$

Although the π-parametrized value changes from $\hat{v}$ to $\hat{v}'$, the prophecy observation $\langle \pi.\, \hat{v}\,\pi = \hat{v}'\,\pi \rangle$ tells us that $\hat{v}$ and $\hat{v}'$ are the same for valid prophecy assignments.

Proof. From $[\dagger\alpha]$ and $\alpha' \sqsubseteq \alpha$, we obtain the dead-lifetime token $[\dagger\alpha']$ by LftIncl-DeadLft. Using it, we turn the borrowed ownership predicate $\bar{\mathfrak{a}} \blacktriangleleft^{\dagger\alpha'} \tau\,\{\hat{v}\}$ (defined in § 6.1.1) into an ownership predicate $\bar{\mathfrak{a}} \blacktriangleleft \tau\,\{\hat{v}'\}$ and a prophecy equalizer under later $\triangleright\, \mathsf{PE}(\hat{v}, \hat{v}')$, for some π-parametrized value $\hat{v}'$. The ownership predicate can be decomposed into $\bar{\mathfrak{a}} \blacktriangleleft \tau\,\{\hat{v}'\}_d$ and a persistent time receipt $\mathord{\text{⧖}} d$ for some depth d. We spend one logical step to strip off the later modality on the prophecy equalizer $\mathsf{PE}(\hat{v}, \hat{v}')$.

By SemTy-Own-ProphToken, we temporarily take out from the ownership predicate a partial prophecy token on some dependency X of $\hat{v}'$ *in d logical steps*, with help of a partial lifetime token on τ. By consuming the prophecy equalizer with help of the partial prophecy token by ProphEqz-Use, we get a prophecy observation $\langle \pi.\, \hat{v}\,\pi = \hat{v}'\,\pi \rangle$. □

Using Raw-Reclaim, we can prove the following refined typing rule for ending a local lifetime α and reclaiming objects borrowed under α.

$$\textsc{SWkn-End-LocalLft-Reclaim}\quad \frac{\forall\, i.\ \ \beta \sqsubseteq \tau_i}{\beta \mid \alpha;\ \overrightarrow{\mathfrak{a} :^{\dagger\alpha} \tau} \vdash^{\#} \overrightarrow{\mathfrak{a} : \tau} \mid \mathsf{id}}$$

Input-Output Relations and Predicate Transformers An *input-output relation* is a common representation of the *postcondition* of functions and programs, which is used in verification methods like reduction to constrained Horn clauses (Grebenshchikov et al., 2012; Bjørner et al., 2015).

We can lift an input-output relation $R\colon T \to U \to \mathsf{Prop}$ (where T and U respectively represent the input and the output) into the following (backward) predicate transformer $\lceil R \rceil : (U \to \mathsf{Prop}) \to T \to \mathsf{Prop}$.

$$\lceil R \rceil \quad := \quad \lambda\, \mathit{post}, v.\ \forall\, w.\ R\, v\, w \Rightarrow \mathit{post}\, w$$

Various operations on predicate transformers correspond to natural operations on input-output relations through the lift function.

$$\lceil R \rceil \circ \lceil R' \rceil \;=\; \lceil\, \lambda u, w.\ \exists v.\ R\,u\,v \,\wedge\, R'\,v\,w \,\rceil$$

$$\lambda post, v.\ post\,(f\,v) \;=\; \lceil\, \lambda v, w.\ w = f\,v \,\rceil$$

$$\lambda post, v.\ \phi\,v \,\Rightarrow\, \lceil R \rceil\, post\, v \;=\; \lceil\, \lambda v, w.\ \phi\,v \,\wedge\, R\,v\,w \,\rceil$$

$$\lambda post, v.\ \lceil R \rceil\, post\, v \,\wedge\, \lceil R' \rceil\, post\, v \;=\; \lceil\, \lambda v, w.\ R\,v\,w \,\vee\, R'\,v\,w \,\rceil$$

Composition of predicate transformers amounts to relational composition of input-output relations. Passing a value transformed by a function f to the postcondition in the predicate transformer amounts to the relational graph of f used as the input-output relation. Adding an assumption $\phi\,v$ on the input v in the predicate transformer amounts to adding the postcondition $\phi\,v$ in the input-output relation. Conjunction of predicate transformers amounts to disjunction of input-output relations.

6.1.3 Other Judgments

Copyability Judgment We introduce the *copyability judgment* τ copy, which is defined as the following persistent predicate.

$$\tau \text{ copy} \quad :=\quad \forall\, \bar{\mathfrak{a}}, \hat{v}, d, \alpha.\ \square\left(\bar{\mathfrak{a}} \blacktriangleleft \tau\,\{\hat{v}\}_d \;\Leftrightarrow\; \bar{\mathfrak{a}} \triangleleft^{\alpha} \tau\,\{\hat{v}\}_d\right)$$

It simply means that the the ownership and sharing predicates of the type τ are equivalent.

By the persistence of the sharing predicate, it allows us to duplicate the ownership predicate, i.e., the following lemma holds.

Raw-Copy

$$\frac{\tau \text{ copy}}{\bar{\mathfrak{a}} \blacktriangleleft \tau\,\{\hat{v}\}_d \;\Rightarrow\; \bar{\mathfrak{a}} \blacktriangleleft \tau\,\{\hat{v}\}_d \,*\, \bar{\mathfrak{a}} \blacktriangleleft \tau\,\{\hat{v}\}_d}$$

Using this lemma, we can prove the following refined typing rule for copying an unborrowed object of a copyable type.

Wkn-Copy

$$\frac{\tau \text{ copy}}{\mathfrak{a}\colon \tau \;\vdash\; \mathfrak{a}\colon \tau,\ \mathfrak{a}\colon \tau \;\vdash\; \lambda post, (v).\ post\,(v, v)}$$

It follows from Raw-Copy.

The integer type int, the invalid-data type $\lightning_n$ and the shared reference type $\&^{\alpha}_{\mathsf{shr}}\,\tau$ are copyable (Copy-Int, Copy-Invalid, Copy-ShrRef). We have also structural copyability rules (Copy-Pair, Copy-Vrnt, Copy-InjTy).

Types like the box pointer type box τ, the vector type vec τ and the unique reference type $\&^{\alpha}_{\mathsf{unq}}\,\tau$ are not copyable.

Subtyping Judgment We introduce the notion of an *access mode acc*: Acc, which is either own (ownership) or shr(α) (sharing under the lifetime α).

The *subtyping judgment* has the following form.

$$\tau \sqsubseteq^{acc} \tau' \;\vdash\; f$$

We transform an object typed τ into an object typed τ' using the *value transformer* f: $\lfloor \tau \rfloor \to \lfloor \tau' \rfloor$. When we apply subtyping, letting v be the pure value of the input object,

the pure value of the output object is $f\,v$. We also require that the types τ, τ' should have the same size (i.e., satisfy $|\tau| = |\tau'|$).

Now the subtyping judgment is defined as the follows.

$$\tau \sqsubseteq^{\mathsf{own}} \tau' \mid f \quad := \quad \forall\, \bar{\mathfrak{a}}, \hat{v}, d.\ \Box \left(\bar{\mathfrak{a}} \blacktriangleleft \tau\, \{\hat{v}\}_d \implies \bar{\mathfrak{a}} \blacktriangleleft \tau'\, \{f \circ \hat{v}\}_d \right)$$

$$\tau \sqsubseteq^{\mathsf{shr}(\alpha)} \tau' \mid f \quad := \quad \forall\, \bar{\mathfrak{a}}, \hat{v}, d.\ \Box \left(\bar{\mathfrak{a}} \blacktriangleleft^{\alpha} \tau\, \{\hat{v}\}_d \implies \bar{\mathfrak{a}} \blacktriangleleft^{\alpha} \tau'\, \{f \circ \hat{v}\}_d \right)$$

When the access mode *acc* is own, we work on the ownership predicate, and wheen it is $\mathsf{shr}(\alpha)$, we work on the sharing predicate with the lifetime set to α.

By applying the subtyping judgment, we can modify the type and value of an object in the type context.

$$\textsc{Wkn-Subty}\ \frac{\tau \sqsubseteq^{\mathsf{own}} \tau' \mid f}{\mathfrak{a} :^{\mathit{act}} \tau \vdash \mathfrak{a} :^{\mathit{act}} \tau' \mid \lambda \mathit{post}, (v).\ \mathit{post}\ (f\ v)}$$

Any type is a subtype of itself under the identity transformer.

$$\textsc{Subty-Id}\ \tau \sqsubseteq^{\mathit{acc}} \tau \mid \mathsf{id}$$

We can compose subtyping judgments, accordingly composing the value transformers.

$$\textsc{Subty-Comp}\ \frac{\tau \sqsubseteq^{\mathit{acc}} \tau' \mid f \qquad \tau' \sqsubseteq^{\mathit{acc}} \tau'' \mid g}{\tau \sqsubseteq^{\mathit{acc}} \tau'' \mid g \circ f}$$

Access Judgment We introduce the following *access judgment.*

$$\alpha \mid \mathfrak{a} :\, {}^{\tau_+}\!/_{\tau'_+} \vdash e : q.\ {}^{\tau(q)}\!/_{\tau'(q)} \mid \mathit{get};\ \mathit{set}$$

The meaning of the expression is as follows. First we have an object of the cell value $\mathfrak{a}$ and the type τ_+. Out of this object, by performing the expression e, we can take out a *sub-object* of the type τ_q for some fraction q. (The part 'q.' binds the variable q.) The fraction can be defined by the *plain reference type* introduced above. We can freely update the sub-object. After we get back the sub-object with a new type $\tau'(q)$, we retrieve the original object with the updated type τ'_+. The initial value of the sub-object is specified by the *getter* function $\mathit{get}\colon \lfloor \tau_+ \rfloor \to \lfloor \tau(1) \rfloor$, which takes the pure value of the original object. The value of the retrieved object is specified by the *setter* function $\mathit{set}\colon \lfloor \tau_+ \rfloor \to \lfloor \tau'(1) \rfloor \to \lfloor \tau'_+ \rfloor$, which takes the pure value of the original object and the returned sub-object. The use of the getter and setter functions here is closely related to the notion of *functional lenses* used in functional programming (Foster et al., 2007).

Here, we assume that all the types $\tau^+, \tau^{+\prime}, \tau(q), \tau'(q)$ have the size 1 (regardless of q) and that the pure types $\lfloor \tau(q) \rfloor$ and $\lfloor \tau'(q) \rfloor$ are constant over the fraction q. The lifetime α is ensured to be alive during this access.

The access judgment is defined as the following persistent predicate.

$$\begin{aligned} &\alpha \mid \mathfrak{a} :\, {}^{\tau_+}\!/_{\tau'_+} \vdash e : q.\ {}^{\tau(q)}\!/_{\tau'(q)} \mid \mathit{get};\ \mathit{set} \quad := \quad \forall\, q'. \\ &\quad \left\{ [\mathfrak{a}] \blacktriangleleft \tau_+\, \{\hat{v}\} \ *\ [\alpha]_{q'} \right\}\ e\ \left\{ \mathfrak{b}.\ \exists q.\ \ [\mathfrak{b}] \blacktriangleleft \tau(q)\, \{\lambda \pi.\, \mathit{get}\ (\hat{v}\ \pi)\} \ * \right. \\ &\qquad \left. \forall\, \hat{w}.\ \left([\mathfrak{b}] \blacktriangleleft \tau'(q)\, \{\hat{w}\} \ \ \Rrightarrow_{\top} \ \ [\mathfrak{a}] \blacktriangleleft \tau'_+\, \{\lambda \pi.\, \mathit{set}\ (\hat{v}\ \pi)\ (\hat{w}\ \pi)\} \ *\ [\alpha]_{q'} \right) \right\}_{\top} \end{aligned}$$

We introduce the following shorthand on the access judgment. We can omit the part '$\alpha \mid$' when the lifetime α is ∞. We can omit the part 'q.' if the types $\tau(q)$ and $\tau'(q)$ do not depend on the fraction q.

We can temporarily take out a full plain reference out of a box pointer and a unique reference (Acc-BoxPtr-PlnRef, Acc-UnqRef-PlnRef) and a fractional plain reference out of a shared reference (Acc-ShrRef-PlnRef). We can also subdivide plain references (e.g., Acc-Deref-PlnRef-BoxPtr, Acc-PlnRef-Pair-L).

By the following rule, we can use an access judgment whose expression e is just the original address $\mathfrak{a}$.

Rfn-Acc-Same

$$\frac{\alpha \mid \mathfrak{a} \colon {}^{\tau_+}\!/_{\tau'_+} \vdash \mathfrak{a} : q.\, {}^{\tau(q)}\!/_{\tau'(q)} \mid \mathit{get};\ \mathit{set} \qquad \forall\, q.\ \ \alpha \mid \Delta;\ \mathfrak{a} \colon \tau(q),\ \Gamma \vdash e_* \colon \tau_* \mid \Delta';\ \mathfrak{a} \colon \tau'(q),\ \Gamma' \mid \mathit{pre}}{\alpha \mid \Delta;\ \mathfrak{a} \colon \tau_+,\ \Gamma \vdash e_* \colon \tau_* \mid \Delta';\ \mathfrak{a} \colon \tau'_+,\ \Gamma' \mid \lambda \mathit{post}, (v, \vec{w}).\ \mathit{pre}\left(\lambda (v', \vec{w'}).\ \mathit{post}\ (\mathit{set}\ v\ v', \vec{w'})\right)\ (\mathit{get}\ v, \vec{w})}$$

The assumption refined typing judgment is universally quantified over the fraction q and has $\mathfrak{a} \colon \tau(q)$ in the inputs and $\mathfrak{a} \colon \tau'(q)$ in the outputs. The predicate transformer *pre* of the assumption judgment is modified using the getter and setter functions. Note that we can use the reverse direction of the rule Hoare-Val to prove this rule.

The following rule is a variant of Rfn-Acc-Same for an access judgment whose expression may not be just the original value.

Rfn-Acc-Expr

$$\frac{\alpha \mid \mathfrak{a} \colon {}^{\tau_+}\!/_{\tau'_+} \vdash e : q.\, {}^{\tau(q)}\!/_{\tau'(q)} \mid \mathit{get};\ \mathit{set} \qquad \forall\, \mathfrak{b}, q.\ \ \alpha \mid \Delta;\ \mathfrak{b} \colon \tau(q),\ \Gamma \vdash K_*[\mathfrak{b}] \colon \tau_* \mid \Delta';\ \mathfrak{b} \colon \tau'(q),\ \Gamma' \mid \mathit{pre}}{\alpha \mid \Delta;\ \mathfrak{a} \colon \tau_+,\ \Gamma \vdash K_*[e] \colon \tau_* \mid \Delta';\ \mathfrak{a} \colon \tau'_+,\ \Gamma' \mid \lambda \mathit{post}, (v, \vec{w}).\ \mathit{pre}\left(\lambda (v', \vec{w'}).\ \mathit{post}\ (\mathit{set}\ v\ v', \vec{w'})\right)\ (\mathit{get}\ v, \vec{w})}$$

We have the following reflexive access judgment.

Acc-Id

$$\mathfrak{a} \colon {}^{\tau}\!/_{\tau'} \vdash \mathfrak{a} \colon {}^{\tau}\!/_{\tau'} \mid \mathrm{id};\ \lambda v, w.\, w$$

We can modify the head lifetime of the access judgment.

Acc-Mono-Lft

$$\frac{\alpha \mid \mathfrak{a} \colon {}^{\tau_+}\!/_{\tau'_+} \vdash \mathfrak{a} : q.\, {}^{\tau(q)}\!/_{\tau'(q)} \mid \mathit{get};\ \mathit{set} \qquad \beta \sqsubseteq \alpha}{\beta \mid \mathfrak{a} \colon {}^{\tau_+}\!/_{\tau'_+} \vdash \mathfrak{a} : q.\, {}^{\tau(q)}\!/_{\tau'(q)} \mid \mathit{get};\ \mathit{set}}$$

We can compose access judgments.

Acc-Comp

$$\frac{\alpha \mid \mathfrak{a} \colon {}^{\tau_{++}}\!/_{\tau'_{++}} \vdash e : q.\, {}^{\tau_+(q)}\!/_{\tau'_+(q)} \mid \mathit{get};\ \mathit{set} \qquad \forall\, \mathfrak{b}, q.\ \ \alpha \mid \mathfrak{b} \colon {}^{\tau_+(q)}\!/_{\tau'_+(q)} \vdash K[\mathfrak{b}] : {}^{\tau(q)}\!/_{\tau'(q)} \mid \mathit{get}';\ \mathit{set}'}{\alpha \mid \mathfrak{a} \colon {}^{\tau_{++}}\!/_{\tau'_{++}} \vdash K[e] : q.\, {}^{\tau(q)}\!/_{\tau'(q)} \mid \mathit{get}' \circ \mathit{get};\ \lambda v, w.\ \mathit{set}\ v\ (\mathit{set}'\ (\mathit{get}'\ v)\ w)}$$

Acc-Comp-Var

$$\frac{\alpha \mid \mathfrak{a} \colon {}^{\tau_{++}}\!/_{\tau'_{++}} \vdash e : q.\, {}^{\tau_+(q)}\!/_{\tau'_+(q)} \mid \mathit{get};\ \mathit{set} \qquad \forall\, \mathfrak{b}, q.\ \ \alpha \mid \mathfrak{b} \colon {}^{\tau_+(q)}\!/_{\tau'_+(q)} \vdash K[\mathfrak{b}] : q'.\, {}^{\tau(q')}\!/_{\tau'(q')} \mid \mathit{get}';\ \mathit{set}'}{\alpha \mid \mathfrak{a} \colon {}^{\tau_{++}}\!/_{\tau'_{++}} \vdash K[e] : q'.\, {}^{\tau(q')}\!/_{\tau'(q')} \mid \mathit{get}' \circ \mathit{get};\ \lambda v, w.\ \mathit{set}\ v\ (\mathit{set}'\ (\mathit{get}'\ v)\ w)}$$

The getter and setter functions are composed in the standard way. In Acc-Comp the second access judgment keeps the fraction q given by the first access judgment, whereas in Acc-Comp-Var the second access judgment introduces a new fraction q'.

6.1.4 Function Type

Now that the refined typing judgment is defined, we can introduce the *function type*. It has the following form.

$$\forall\, \vec{\alpha}.\ \mathsf{fn}(\overrightarrow{\tau_{\vec{\alpha}}}) \to \tau'_{\vec{\alpha}}$$

The function type is universally quantified over the lifetime parameters $\vec{\alpha}$. We omit the assumptions on inclusion among these lifetimes; we virtually don't lose expressivity because we can use conjunction over lifetimes. The input types are $\overrightarrow{\tau_{\vec{\alpha}}}$ and the output type is $\tau'_{\vec{\alpha}}$, which depends on the lifetime parameters $\vec{\alpha}$. The input and output types should have the size 1 regardless of $\vec{\alpha}$ and the pure types of the input and output types should be constant over $\vec{\alpha}$.

The function type is modeled as follows.

$$\lfloor \forall\, \vec{\alpha}.\ \mathsf{fn}(\overrightarrow{\tau_{\vec{\alpha}}}) \to \tau'_{\vec{\alpha}} \rfloor \ :=\ \big(\times(\lfloor \tau'_{\vec{\infty}} \rfloor) \to \mathsf{Prop}\big) \to \times(\overrightarrow{\lfloor \tau_{\vec{\infty}} \rfloor}) \to \mathsf{Prop}$$

$$|\forall\, \vec{\alpha}.\ \mathsf{fn}(\overrightarrow{\tau_{\vec{\alpha}}}) \to \tau'_{\vec{\alpha}}| \ :=\ 1 \qquad\qquad \beta \sqsubseteq \forall\, \vec{\alpha}.\ \mathsf{fn}(\overrightarrow{\tau_{\vec{\alpha}}}) \to \tau'_{\vec{\alpha}} \ :=\ \mathsf{True}$$

$$\begin{aligned}[\mathfrak{a}] \blacktriangleleft \forall\, \vec{\alpha}.\ \mathsf{fn}(\overrightarrow{\tau_{\vec{\alpha}}}) \to \tau'_{\vec{\alpha}}\ \{\hat{v}\}_d \ =\ [\mathfrak{a}] \lhd^{\alpha'} \forall\, \vec{\alpha}.\ \mathsf{fn}(\overrightarrow{\tau_{\vec{\alpha}}}) \to \tau'_{\vec{\alpha}} \ :=\ \\ \exists\, f, \vec{b}, e.\ \mathsf{let}\ \mathfrak{f} = \mathsf{fn}\, f(\vec{b})\{e\}\ \mathsf{in}\ \mathfrak{a} = \mathfrak{f} \ast \exists\, \mathit{pre}\ \text{s.t.}\ \hat{v} = \mathsf{const}\ \mathit{pre}. \\ \forall\, \vec{\alpha}, \vec{\mathfrak{b}}, \beta.\ \triangleright \Box \big(\beta \sqsubseteq (\overrightarrow{\tau_{\vec{\alpha}}}) \ \Rightarrow\ \beta \,\mathrel{|}\, \overrightarrow{\mathfrak{b}\colon \tau_{\vec{\alpha}}} \vdash e[\mathfrak{f}/f, \overrightarrow{\mathfrak{b}/b}] \colon \tau'_{\vec{\alpha}} \,\mathrel{|\!|}\, \mathit{pre}\big)\end{aligned}$$

Here, we introduce the following shorthand.

$$\alpha \sqsubseteq (\vec{\tau}) \ :=\ \forall\, i.\ \alpha \sqsubseteq \tau_i$$

The π-parametrized pure value for a function is a constant function on a *predicate transformer pre*. The cell value of a function is some function value $\mathfrak{f}$. For any lifetime parameters $\vec{\alpha}$, a function call $\mathfrak{f}(\vec{\mathfrak{b}})$ on any inputs $\vec{\mathfrak{b}}$ typed $\overrightarrow{\tau_{\vec{\alpha}}}$ should output an object typed $\tau'_{\vec{\alpha}}$ under some lifetime β and the predicate transformer *pre*, which is specified by the refined typing judgment. The lifetime β is ensured to be outlived by all the input types. The refined typing judgment in the ownership and sharing predicates is under the later modality for *contractiveness* over the input and output types $\overrightarrow{\tau_{\vec{\alpha}}}, \tau'_{\vec{\alpha}}$. We can strip off the later modality at the physical step of the function call (Hoare-FnCall).

The function type satisfies the following contractiveness rule.

$$\textsc{Contr-Fn}\quad \frac{\forall\, i, \vec{\alpha}.\ \mathsf{Nonex}(\sigma_{\vec{\alpha},i}) \qquad \forall\, \vec{\alpha}.\ \mathsf{Nonex}(\sigma'_{\vec{\alpha}})}{\mathsf{Contr}\big(\lambda \tau.\ \forall\, \vec{\alpha}.\ \mathsf{fn}(\overrightarrow{\sigma_{\vec{\alpha}}\, \tau}) \to \sigma'_{\vec{\alpha}}\, \tau\big)}$$

For the function type $\forall\, \vec{\alpha}.\ \mathsf{fn}(\overrightarrow{\tau_{\vec{\alpha}}}) \to \tau'_{\vec{\alpha}}$, we also use the following shorthand. If the sequence of lifetime parameters $\vec{\alpha}$ is empty, we can omit the part '$\forall\, \vec{\alpha}.\,$' of the function type. If the subscript lifetime $\beta_{\vec{\alpha}}$ is constantly ∞, we can omit it. Also, if the output type $\tau'_{\vec{\alpha}}$ is $\lightning_1$, we can omit the part '$\to \tau'_{\vec{\alpha}}$'.

6.2 Specifying and Verifying Operations on Basic Types

We introduce refined typing rules on the judgments defined in §6.1 for the semantic types defined in §5.2, §5.4 and §6.1.4.

Ths refined typing rules for the unique reference type are introduced later in §6.3.

6.2.1 Value Types

Integer Type We have the following refined typing rules on the integer type int (§5.2).

$$\textsc{Rfn-Val-Int}\quad \vdash n : \mathsf{int} \mid\mid \lambda \mathit{post}, (). \ \mathit{post}\,(n)$$

$$\textsc{Rfn-IntOp}\quad \mathfrak{a}: \mathsf{int},\ \mathfrak{b}: \mathsf{int} \vdash \mathfrak{a}\ \mathit{iop}\ \mathfrak{b} : \mathsf{int} \mid\mid \lambda \mathit{post}, (m, n). \ \mathit{post}\,(m\ \mathit{iop}\ n)$$

$$\textsc{Rfn-IntRel}\quad \mathfrak{a}: \mathsf{int},\ \mathfrak{b}: \mathsf{int} \vdash \mathfrak{a}\ \mathit{irel}\ \mathfrak{b} : \mathsf{bool} \mid\mid \lambda \mathit{post}, (m, n). \ \mathit{post}\,(m\ \mathit{irel}_{\mathsf{bool}}\ n)$$

$$\textsc{Rfn-NdInt}\quad \vdash \mathsf{ndint} : \mathsf{int} \mid\mid \lambda \mathit{post}, (). \ \forall n.\ \mathit{post}\,(n) \qquad\qquad \textsc{Copy-Int}\quad \mathsf{int}\ \mathsf{copy}$$

Rfn-Val-Int, Rfn-IntOp, Rfn-IntRel, and Rfn-NdInt specify an expression on a constant integer, an integer operation, an integer relation, and a non-deterministic integer (*iop* can be $+, -, \times$ and *irel* can be $\leq, <, =$, for example). Copy-Int enables copying of an integer.

Here, we elaborate the proof of Rfn-IntOp.

Proof of Rfn-IntOp. It suffices to prove that the following is a tautology.

$$\left\{ \exists \hat{v}, \hat{v}'. \ \left\langle \pi.\ \mathit{post}\,(\hat{v}\,\pi\ \mathit{iop}\ \hat{v}'\,\pi) \right\rangle \ *\ [\mathfrak{a}] \blacktriangleleft \mathsf{int}\,\{\hat{v}\} \ *\ [\mathfrak{b}] \blacktriangleleft \mathsf{int}\,\{\hat{v}'\} \right\}$$
$$\mathfrak{a}\ \mathit{iop}\ \mathfrak{b} \qquad \left\{ \mathfrak{c}. \ \exists \hat{w}. \ \left\langle \pi.\ \mathit{post}\,(\hat{w}\,\pi) \right\rangle \ *\ [\mathfrak{c}] \blacktriangleleft \mathsf{int}\,\{\hat{w}\} \right\}$$

By the model of the integer type, the cell values $\mathfrak{a}$ and $\mathfrak{b}$ should be some integers m and n. Also, the π-parametrized pure values $\hat{v}$ and $\hat{v}'$ should be const m and const n. So by Hoare-IntOp, we can execute $\mathfrak{a}\ \mathit{iop}\ \mathfrak{b}$ and obtain a cell value $\mathfrak{c}$, which is equal to the integer $l := m\ \mathit{iop}\ n$. We can set $\hat{w}$ to const l. Now we have $[\mathfrak{c}] \blacktriangleleft \mathsf{int}\,\{\hat{w}\}$ by the model of the integer type. We also get $\left\langle \pi.\ \mathit{post}\,(\hat{w}\,\pi) \right\rangle$, because it is equivalent to $\left\langle \pi.\ \mathit{post}\,(\hat{v}\,\pi\ \mathit{iop}\ \hat{v}'\,\pi) \right\rangle$ by PropObs-Wkn. □

Boolean Type We have the following rules on the boolean type bool (§5.2).

$$\textsc{Rfn-Val-Bool}\quad \vdash \mathit{bl}: \mathsf{bool} \mid\mid \lambda \mathit{post}, (). \ \mathit{post}\,(\mathit{bl}) \qquad\qquad \textsc{Copy-Bool}\quad \mathsf{bool}\ \mathsf{copy}$$

$$\textsc{Rfn-If}\quad \frac{\forall\, \mathit{bl}.\ \ \alpha \mid \Delta; \Gamma \vdash e_{\mathit{bl}}: \tau \mid \Delta'; \Gamma' \mid \mathit{pre}_{\mathit{bl}}}{\alpha \mid \Delta;\ \mathfrak{a}: \mathsf{bool},\ \Gamma \vdash \mathsf{if}\ \mathfrak{a}\ \{e_{\mathsf{tt}}\}\ \mathsf{else}\ \{e_{\mathsf{ff}}\} : \tau \mid \Delta';\ \Gamma' \mid \lambda \mathit{post}, (\mathit{bl}, \vec{v}). \ \mathit{pre}_{\mathit{bl}}\ \mathit{post}\,(\vec{v})}$$

Rfn-Val-Bool specifies a boolean-value expression. Copy-Bool enables copying of a boolean value. Rfn-If specifies conditional branching by a boolean value; note that the resulting predicate transformer is equivalent to the following.

$$\lambda \mathit{post}, (\mathit{bl}, \vec{v}). \ (\mathit{bl} = \mathsf{tt} \implies \mathit{pre}_{\mathsf{tt}}\ \mathit{post}\,(\vec{v})) \wedge (\mathit{bl} = \mathsf{ff} \implies \mathit{pre}_{\mathsf{ff}}\ \mathit{post}\,(\vec{v}))$$

We can *reinterpret* a boolean value as an integer value.

$$\textsc{Subty-Bool-Int}\quad \mathsf{bool} \sqsubseteq^{\mathit{acc}} \mathsf{int} \mid \lambda \mathit{bl}. \ \mathsf{if}\ \mathit{bl}\,\{1\}\ \mathsf{else}\ \{0\}$$

Invalid-Data Type We have the following rules on the invalid-data type $\lightning_n$ (§5.2).

$$\textsc{Copy-Invalid}\quad \lightning_n \text{ copy} \qquad\qquad \textsc{Subty-Invalid}\quad \tau \sqsubseteq^{acc} \lightning_{|\tau|} \mid \lambda v.\, ()$$

Copy-Int enables copying of invalid data. Subty-Invalid invalidates an object of any type, turning the pure value into () regardless of the original pure value v.

6.2.2 Basic Pointer Types

Box Pointer Type Now we introduce refined typing rules for the box pointer type $\text{box}\,\tau$ (§5.2).

By allocating a new memory block of the size n, we can create a box pointer to uninitialized, invalid data of the size n.

$$\textsc{Rfn-Alloc}\quad \vdash \text{alloc}\, n : \text{box}\, \lightning_n \mid\mid \lambda post, (). \, post\, (())$$

We can free the memory block of a box pointer.

$$\textsc{Rfn-Free}\quad \mathfrak{a} : \text{box}\,\tau \vdash \text{free}\,\mathfrak{a} \mid\mid \lambda post\,(v).\, post\,()$$

We can temporarily take out a *full plain reference* from a box pointer. We may change the target type when we put back a full reference, as long as the size of the target type does not change.

$$\textsc{Acc-BoxPtr-PlnRef}\quad \frac{|\tau| = |\tau'|}{\mathfrak{a} :\, {}^{\text{box}\,\tau}/_{\text{box}\,\tau'} \vdash \mathfrak{a} :\, {}^{\&_1\,\tau}/_{\&_1\,\tau'} \mid \text{id};\ \lambda v, w.\, w}$$

It follows from the following lemma.

$$\textsc{Raw-BoxPtr-PlnRef}\quad \frac{|\tau| = |\tau'|}{\begin{array}{l}[\mathfrak{a}] \blacktriangleleft \text{box}\,\tau\,\{\hat{v}\} \quad \Longrightarrow \\ \quad [\mathfrak{a}] \blacktriangleleft \&_1\,\tau\,\{\hat{v}\} \ * \ \forall \hat{v}'.\, \left([\mathfrak{a}] \blacktriangleleft \&_1\,\tau'\,\{\hat{v}'\} \mathrel{-\!\!*} [\mathfrak{a}] \blacktriangleleft \text{box}\,\tau'\,\{\hat{v}'\}\right)\end{array}}$$

We can also subdivide a plain reference to a box pointer.

$$\textsc{Acc-Deref-PlnRef-BoxPtr}\quad \frac{|\tau| = |\tau'|}{\mathfrak{a} :\, {}^{\&_q\,\text{box}\,\tau}/_{\&_q\,\text{box}\,\tau'} \vdash *\mathfrak{a} :\, {}^{\&_1\,\tau}/_{\&_1\,\tau'} \mid \text{id};\ \lambda v, w.\, w}$$

It follows from the following lemma.

$$\textsc{Raw-Subdiv-PlnRef-BoxPtr}\quad \frac{|\tau| = |\tau'|}{\begin{array}{l}[\mathfrak{l}] \blacktriangleleft \&_q\,\text{box}\,\tau\,\{\hat{v}\} \ \Rrightarrow_{\varnothing} \ \exists \mathfrak{b}.\, \mathfrak{l} \overset{q}{\mapsto} \mathfrak{b} \ * \ [\mathfrak{b}] \blacktriangleleft \&_1\,\tau\,\{\hat{v}\} \ * \\ \quad \forall \hat{v}', d'.\, \left(\text{⧗}1 \ * \ \mathfrak{l} \overset{q}{\mapsto} \mathfrak{b} \ * \ [\mathfrak{b}] \blacktriangleleft \&_1\,\tau\,\{\hat{v}'\} \mathrel{-\!\!*} [\mathfrak{l}] \blacktriangleleft \&_q\,\text{box}\,\tau\,\{\hat{v}\}\right)\end{array}}$$

We use the cumulative time receipt ⧗1 to swell the persistent time receipt of the new target object (CumuTime-Swell-PersTime).

The box pointer type admits the following subtyping rule.

$$\textsc{Subty-BoxPtr}\quad \frac{\triangleright\left(\tau \sqsubseteq^{acc} \tau' \mid f\right)}{\text{box}\,\tau \sqsubseteq^{acc} \text{box}\,\tau' \mid f}$$

The subtyping assumption on the target types can be *under the later modality*. This is important for proving subtyping on *recursive types* with self reference under the box pointer type; see also Recursive Type of this section.

Shared Reference Type We introduce refined typing rules for the shared reference type $\&^{\alpha}_{\mathsf{shr}}\,\tau$ (§5.2).

We can create a shared reference by borrowing a full plain reference.

SWkn-ShrBor-PlnRef
$$\mathfrak{a}\colon \&_1\,\tau \;\vdash^{\#}\; \mathfrak{a}\colon \&^{\alpha}_{\mathsf{shr}}\,\tau,\ \mathfrak{a} :^{\dagger\alpha} \&_1\,\tau \;\mid\; \lambda\, post, (v).\ post\,(v, v)$$

It follows from the following lemma.

Raw-ShrBor-PlnRef
$$[\mathfrak{l}] \blacktriangleleft \&_1\,\tau\,\{\hat{v}\}_{d+1} \;\Rrightarrow^{d}_{\mathcal{N}_{\mathsf{lft}}}\; [\mathfrak{l}] \blacktriangleleft \&^{\alpha}_{\mathsf{shr}}\,\tau\,\{\hat{v}\}_{d+1} \;*\; [\mathfrak{l}] \blacktriangleleft^{\dagger\alpha} \&_1\,\tau\,\{\hat{v}\}$$

The ownership predicate under borrow is defined in §6.1.1.

Proof of Raw-ShrBor-PlnRef. The plain reference consists of a full points-to token $\mathfrak{l} \overset{1}{\mapsto} \bar{\mathfrak{b}}$ and a target object under the later modality $\triangleright\,(\,\bar{\mathfrak{b}} \blacktriangleleft \tau\,\{\hat{v}\}_d\,)$.

We fully borrow the points-to token (FullBor-Intro). We split the full borrow $\&^{\alpha}_{\mathsf{full}}(\mathfrak{l} \overset{1}{\mapsto} \bar{\mathfrak{b}})$ into $*_i\, \&^{\alpha}_{\mathsf{full}}(\mathfrak{l} + i \overset{1}{\mapsto} \bar{\mathfrak{b}}[i])$ (FullBor-Split) and turn them into fractured borrows $*_i\, \&^{\alpha}_{\mathsf{frac}}(\lambda q.\ \mathfrak{l} + i \overset{q}{\mapsto} \bar{\mathfrak{b}}[i])$ (FullBor-FracBor). We also get a view shift $[\dagger\alpha] \Rrightarrow *_{\mathcal{N}_{\mathsf{lft}}} \mathfrak{l} \overset{q}{\mapsto} \bar{\mathfrak{b}}$.

We also fully borrow the target object under later (FullBor-Intro). We can turn the resulting full borrow $\&^{\alpha}_{\mathsf{full}}(\bar{\mathfrak{b}} \blacktriangleleft \tau\,\{\hat{v}\}_d)$ into the sharing predicate on the target object $\bar{\mathfrak{b}} \triangleleft^{\alpha} \tau\,\{\hat{v}\}_d$ in d logical steps by SemTy-Own-Shr on τ. We also get a view shift $[\dagger\alpha] \Rrightarrow *_{\mathcal{N}_{\mathsf{lft}}} \triangleright\,(\,\bar{\mathfrak{b}} \blacktriangleleft \tau\,\{\hat{v}\}_d\,)$.

Combining the fractured borrows and the sharing predicate, we get a shared reference $[\mathfrak{l}] \blacktriangleleft \&^{\alpha}_{\mathsf{shr}}\,\tau\,\{\hat{v}\}_{d+1}$. Combining the two view shifts with a trivial prophecy equalizer $\mathsf{PE}(\hat{v}, \hat{v})$ (made by ProphEqz-ProphObs and ProphObs-Fact), we get a plain reference under borrow $\mathfrak{l} \blacktriangleleft^{\dagger\alpha} \&_1\,\tau\,\{\hat{v}\}_{d+1}$. □

We can create a shared reference also by borrowing a box pointer.

SWkn-ShrBor-BoxPtr
$$\mathfrak{a}\colon \mathsf{box}\,\tau \;\vdash^{\#}\; \mathfrak{a}\colon \&^{\alpha}_{\mathsf{shr}}\,\tau,\ \mathfrak{a} :^{\dagger\alpha} \mathsf{box}\,\tau \;\mid\; \lambda\, post, (v).\ post\,(v, v)$$

It follows from Raw-ShrBor-PlnRef and Raw-BoxPtr-PlnRef.

The shared reference type is copyable, unlike the box pointer type and the unique reference type.

Copy-ShrRef
$$\&^{\alpha}_{\mathsf{shr}}\,\tau\ \mathsf{copy}$$

We can temporarily take out a *fractional plain reference* from an shared reference if the target type is copyable.

Acc-ShrRef-PlnRef
$$\frac{\tau\ \mathsf{copy} \qquad \tau'\ \mathsf{copy} \qquad |\tau| = |\tau'|}{\alpha \mid \mathfrak{a}\colon {}^{\&^{\alpha}_{\mathsf{shr}}\,\tau}/_{\&^{\alpha}_{\mathsf{shr}}\,\tau'} \;\vdash\; \mathfrak{a}\colon q.\ {}^{\&_q\,\tau}/_{\&_q\,\tau'} \;\mid\; \mathsf{id};\ \lambda v, w.\, w}$$

It follows from the following lemma.

Raw-ShrRef-PlnRef
$$\frac{\tau\ \mathsf{copy} \qquad \tau'\ \mathsf{copy} \qquad |\tau| = |\tau'|}{\begin{array}{l}[\mathfrak{a}] \blacktriangleleft \&^{\alpha}_{\mathsf{shr}}\,\tau\,\{\hat{v}\} \;*\; [\alpha]_{q'} \;\Rrightarrow_{\mathcal{N}_{\mathsf{lft}}} \exists\, q. \\ \quad [\mathfrak{a}] \blacktriangleleft \&_q\,\tau\,\{\hat{v}\} \;*\; \forall\, \hat{v}'.\,(\,[\mathfrak{a}] \blacktriangleleft \&_q\,\tau'\,\{\hat{v}'\} \Rrightarrow *_{\mathcal{N}_{\mathsf{lft}}} [\mathfrak{a}] \blacktriangleleft \&^{\alpha}_{\mathsf{shr}}\,\tau'\,\{\hat{v}'\} \;*\; [\alpha]_{q'}\,)\end{array}}$$

Proof of Raw-ShrRef-PlnRef. The cell value $\mathfrak{a}$ should be some address $\mathfrak{l}$ and the depth d should be positive.

Let $\bar{\mathfrak{b}}$ the target cell values of the shared reference. Out of the fractured borrows of the shared reference, we can temporarily take out a fractional points-to token $\mathfrak{l} \overset{q}{\mapsto} \bar{\mathfrak{b}}$ for some fraction q by FracBor-Access, using a fractional lifetime token $[\alpha]_{q'}$. By τ copy, we can turn the target sharing predicate under later $\triangleright \left(\bar{\mathfrak{b}} \triangleleft^{\alpha} \tau \, \{\hat{v}\}_{d-1} \right)$ into the target ownership predicate under later $\triangleright \left(\bar{\mathfrak{b}} \blacktriangleleft \tau \, \{\hat{v}\}_{d-1} \right)$. Therefore we can construct a plain reference of some fraction q, $[\mathfrak{a}] \blacktriangleleft \&_q \tau \, \{\hat{v}\}_d$.

Assume that we get back a plain reference $[\mathfrak{a}] \blacktriangleleft \&_q \tau' \, \{\hat{v}'\}_{d'}$. We get back a fractional points-to token of the fraction q, $\mathfrak{l} \overset{q}{\mapsto} \bar{\mathfrak{b}}'$, and thus get back the lifetime token $[\alpha]_{q'}$. By τ' copy, the target ownership predicate under later $\triangleright \left(\bar{\mathfrak{b}}' \blacktriangleleft \tau \, \{\hat{v}'\}_{d'-1} \right)$ of the plain reference can be turned into the target sharing predicate. So we can reconstruct the shared reference $[\mathfrak{a}] \blacktriangleleft \&^{\alpha}_{\mathsf{shr}} \tau' \, \{\hat{v}'\}_{d'}$. □

Combining Acc-ShrRef-PlnRef and Rfn-Deref-PlnRef-Copy, we can derive the following dereference rule on a shared reference to a shared reference. The lifetime of the resulting reference can be that of the *inner* shared reference β.

$$\textsc{Rfn-Deref-ShrRef-ShrRef}$$
$$\alpha \lhd \mathfrak{a} \colon \&^{\alpha}_{\mathsf{shr}} \&^{\beta}_{\mathsf{shr}} \tau \vdash *\mathfrak{a} : \&^{\beta}_{\mathsf{shr}} \tau \dashv\vdash \mathsf{id}$$

We can also subdivide a plain reference to an shared reference.

$$\textsc{Acc-Deref-PlnRef-ShrRef} \quad \frac{|\tau| = |\tau'| \qquad \tau \text{ copy} \qquad \tau' \text{ copy}}{\alpha \lhd \mathfrak{a} \colon {}^{\&_q \&^{\alpha}_{\mathsf{shr}} \tau}\!/_{\&_q \&^{\alpha}_{\mathsf{shr}} \tau'} \vdash *\mathfrak{a} \colon q'. \, {}^{\&_{q'} \tau}\!/_{\&_{q'} \tau'} \dashv \mathsf{id};\ \lambda v, w. \, w}$$

It can be proved using Raw-ShrRef-PlnRef. Note that we get a cumulative time receipt $⧗1$ by the physical step of the load operation.

The shared reference type admits the following subtyping rule.

$$\textsc{Subty-ShrRef} \quad \frac{\triangleright \left(\tau \sqsubseteq^{\mathsf{shr}(\alpha)} \tau' \dashv f \right) \qquad \beta \sqsubseteq \alpha}{\&^{\alpha}_{\mathsf{shr}} \tau \sqsubseteq^{acc} \&^{\beta}_{\mathsf{shr}} \tau' \dashv f}$$

It modifies the target type of the shared reference type; the subtyping assumption on the target type is discussed on the sharing access mode and can be under the later modality like Subty-BoxPtr.

We can turn a box pointer into a shared reference under the access mode $\mathsf{shr}(\alpha)$.

$$\textsc{Subty-Shr-BoxPtr-ShrRef}$$
$$\mathsf{box}\, \tau \sqsubseteq^{\mathsf{shr}(\alpha)} \&^{\alpha}_{\mathsf{shr}} \tau \dashv \mathsf{id}$$

Combining this with Rfn-Deref-ShrRef-ShrRef, we can derive the following dereference rule.

$$\textsc{Rfn-Deref-ShrRef-BoxPtr} \quad \frac{\beta \sqsubseteq \alpha}{\beta \lhd \mathfrak{a} \colon \&^{\alpha}_{\mathsf{shr}} \mathsf{box}\, \tau \vdash *\mathfrak{a} : \&^{\alpha}_{\mathsf{shr}} \tau \dashv\vdash \mathsf{id}}$$

Plain Reference Type We introduce refined typing rules for the plain reference type $\&_q \tau$ (§5.2). We read and update objects through plain references, using access rules like Acc-BoxPtr-PlnRef and Acc-ShrRef-PlnRef.

By dereferencing a plain reference to a size-1 type τ, we can move out the target object. If the target type is copyable, the plain reference can retain the target object.

RFN-DEREF-PLNREF-MOVE
$$\mathfrak{a}\colon \&_q\, \tau \;\vdash\; *\mathfrak{a} : \tau \mid \mathfrak{a}\colon \&_q\, \lightning_{|\tau|} \mid \lambda\, post, (v).\ post\, (v, ())$$

RFN-DEREF-PLNREF-COPY
$$\frac{\tau \text{ copy}}{\mathfrak{a}\colon \&_q\, \tau \;\vdash\; *\mathfrak{a} : \tau \mid \mathfrak{a}\colon \&_q\, \tau \mid \lambda\, post, (v).\ post\, (v, v)}$$

We can strip off the later modality on the target object of a plain reference, using the physical step of the load operation.

We can use the following rule when the target type τ, τ' has the size 1.

RFN-STORE-PLNREF-VAL
$$\frac{|\tau| = |\tau'| = 1}{\mathfrak{a}\colon \&_1\, \tau,\ \mathfrak{b}\colon \tau' \;\vdash\; \mathfrak{a} \leftarrow \mathfrak{b} \mid \mathfrak{a}\colon \&_1\, \tau' \mid \lambda\, post, (v, w).\ post\, (w)}$$

When the new target object $\mathfrak{b}$ has the depth d, the updated plain reference $\mathfrak{a}$ should have the depth $d + 1$ and thus have the persistent time receipt $\boxtimes(d + 1)$. We make $\boxtimes(d+1)$ out of the persistent time receipt $\boxtimes d$ of $\mathfrak{b}$ and the cumulative time receipt $\hourglass 1$ obtained by the physical step of the store operation, using CUMUTIME-SWELL-PERSTIME.

We can also move or copy an object from a plain reference to a full plain reference.

RFN-STORE-PLNREF-PLNREF-MOVE
$$\frac{|\tau'| = |\tau|}{\mathfrak{a}\colon \&_1\, \tau',\ \mathfrak{b}\colon \&_q\, \tau \mid \mathfrak{a} \leftarrow^*_{|\tau|} \mathfrak{b} \mid \mathfrak{a}\colon \&_1\, \tau,\ \mathfrak{b}\colon \&_q\, \lightning_{|\tau|} \mid \lambda\, post, (w, v).\ post\, (v, ())}$$

RFN-STORE-PLNREF-PLNREF-COPY
$$\frac{|\tau'| = |\tau| \qquad \tau \text{ copy}}{\mathfrak{a}\colon \&_1\, \tau',\ \mathfrak{b}\colon \&_q\, \tau \mid \mathfrak{a} \leftarrow^*_{|\tau|} \mathfrak{b} \mid \mathfrak{a}\colon \&_1\, \tau,\ \mathfrak{b}\colon \&_q\, \tau \mid \lambda\, post, (w, v).\ post\, (v, v)}$$

The plain reference type satisfies the following structural subtyping rule.

SUBTY-PLNREF
$$\frac{\tau \sqsubseteq^{\mathsf{own}} \tau' \mid f}{\&_q\, \tau \sqsubseteq^{\mathsf{own}} \&_q\, \tau' \mid f}$$

6.2.3 Constructive Types

Pair Type We introduce refined typing rules for the pair type $\tau \times \tau'$ (§5.2).

When we have a plain reference to a pair, we can take out a plain reference to each component of the pair.

ACC-PLNREF-PAIR-L
$$\frac{|\tau_0| = |\tau_0'|}{\mathfrak{a}\colon {}^{\&_q(\tau_0 \times \tau_1)}/_{\&_q(\tau_0' \times \tau_1)} \;\vdash\; \mathfrak{a}\colon {}^{\&_q \tau_0}/_{\&_q \tau_0'} \mid \lambda v.\, v.0;\ \lambda v, w.\, (w, v.1)}$$

ACC-PLNREF-PAIR-R
$$\mathfrak{a}\colon {}^{\&_q(\tau_0 \times \tau_1)}/_{\&_q(\tau_0 \times \tau_1')} \;\vdash\; \mathfrak{a}.|\tau_0|\colon {}^{\&_q \tau_1}/_{\&_q \tau_1'} \mid \lambda v.\, v.1;\ \lambda v, w.\, (v.0, w)$$

We can also split a plain reference to a pair into plain references to each element of the pair.

RFN-SPLIT-PLNREF-PAIR-R
$$\frac{|\tau_0| = |\tau_0'|}{\begin{array}{l}\forall\, \mathfrak{b}.\ \ \alpha \mid \Delta;\ \mathfrak{a}\colon \&_q\, \tau_0,\ \mathfrak{b}\colon \&_q\, \tau_1,\ \Gamma \;\vdash\; K[\mathfrak{b}]\colon \tau \mid \Delta';\ \mathfrak{a}\colon \&_q\, \tau_0',\ \mathfrak{b}\colon \&_q\, \tau_1',\ \Gamma' \mid \mathit{pre} \\ \hline \alpha \mid \Delta;\ \mathfrak{a}\colon \&_q(\tau_0 \times \tau_1),\ \Gamma \;\vdash\; K[\mathfrak{a}.|\tau_0|]\colon \tau \mid \Delta';\ \mathfrak{a}\colon \&_q(\tau_0' \times \tau_1'),\ \Gamma' \mid \\ \quad \lambda\, post, ((v_0, v_1), \vec{w}).\ \mathit{pre}\left(\lambda (v_0', v_1', \vec{w}').\ post\, ((v_0', v_1'), \vec{w}')\right)(v_0, v_1, \vec{w})\end{array}}$$

These rules follow from the following simple lemma.

Raw-Split-Merge-PlnRef-Pair
$$[\mathrm{I}] \blacktriangleleft \&_q(\tau_0 \times \tau_1)\,\{\lambda\pi.\,(\hat{v}_0\,\pi, \hat{v}_1\,\pi)\}_d \;\Leftrightarrow\; [\mathrm{I}] \blacktriangleleft \&_q\,\tau_0\,\{\hat{v}_0\}_d \;*\; [\mathrm{I} + |\tau_0|] \blacktriangleleft \&_q\,\tau_1\,\{\hat{v}_1\}_d$$

From a shared reference to a pair, we can take shared references to each component of the pair.

Subty-Subdiv-ShrRef-Pair-L
$$\&^{\alpha}_{\mathsf{shr}}(\tau \times \tau') \sqsubseteq \&^{\alpha}_{\mathsf{shr}}\,\tau \;\vdash\; \lambda v.\,v.0$$

Rfn-Subdiv-ShrRef-Pair-R
$$\mathfrak{a} : \&^{\alpha}_{\mathsf{shr}}(\tau \times \tau') \;\vdash\; \mathfrak{a}.|\tau| : \&^{\alpha}_{\mathsf{shr}}\,\tau' \;\vdash\!\vdash\; \lambda post, (v).\; post\,(v.1)$$

We have the following structural rules on copyability and subtyping on the pair type.

Copy-Pair
$$\frac{\tau \text{ copy} \qquad \tau' \text{ copy}}{\tau \times \tau' \text{ copy}}$$

Subty-Pair
$$\frac{\tau_0 \sqsubseteq^{acc} \tau_0' \vdash f \qquad \tau_1 \sqsubseteq^{acc} \tau_1' \vdash g}{\tau_0 \times \tau_1 \sqsubseteq^{acc} \tau_0' \times \tau_1' \vdash f \times g}$$

Here, the function $f \times g$ is defined as follows.

$$f \times g \;:=\; \lambda(v, w).\,(f\,v, g\,w)$$

We can also use the following subtyping rules for *reinterpreting* a nested pair in terms of associativity.

Subty-Assoc-Pair-L
$$(\tau \times \tau') \times \tau'' \simeq \tau \times (\tau' \times \tau'') \;\vdash\; \lambda((v, v'), v'').\,(v, (v', v''))$$

Subty-Assoc-Pair-R
$$\tau \times (\tau' \times \tau'') \times \tau'' \simeq (\tau \times \tau') \;\vdash\; \lambda(v, (v', v'')).\,((v, v'), v'')$$

Using these rules, we can simplify the address shift operation for data access, i.e., we can write l instead of $e.m.n$ where $l = m + n$.

We have the following subtyping rule for splitting an invalid object.

Subty-Split-Invalid
$$\lightning_{m+n} \sqsubseteq^{acc} \lightning_m \times \lightning_n \;\vdash\; \mathsf{const}\,((), ())$$

Variant Type We introduce refined typing rules for the variant type $\tau + \tau'$ (§5.2).

We can convert between a plain reference to a variant and a plain reference to a triple of the tag, body and invalid object.

Wkn-PlnRef-Vrnt-Triple
$$\mathfrak{a} : \&_q(\tau_0 + \tau_1) \;\vdash\; \mathfrak{a} : \&_q\big((\mathsf{int} \times \tau_i) \times \lightning_{|\tau_0+\tau_1|-1-|\tau_i|}\big) \;\vdash\!\vdash\; \lambda post, (v).\; \exists w \text{ s.t. } v = \mathsf{inj}_i\,w.\; post\,(((i, w), ()))$$

Wkn-PlnRef-Triple-Vrnt
$$\mathfrak{a} : \&_q\big((\mathsf{int} \times \tau_i) \times \lightning_{|\tau_0+\tau_1|-1-|\tau_i|}\big) \;\vdash\; \mathfrak{a} : \&_q(\tau_0 + \tau_1) \;\vdash\!\vdash\; \lambda post, \big(((n, v), ())\big).\; n = i \;\wedge\; post\,(\mathsf{inj}_i\,v)$$

In the rule Wkn-PlnRef-Vrnt-Triple, the predicate transformer requires the precondition $\exists w.\, v = \mathsf{inj}_i\,w$ (i.e., the tag of v is i) for any postcondition *post*, which ensures that the cell value at the address $\mathfrak{a}$ is i (by the definition of the variant type and PropObs-Sat). In the rule Wkn-PlnRef-Triple-Vrnt, we have the precondition that the tag number n has the value i. These rules allow us to read and update a variant object using the rules for pairs.

Also, we also have the following rule for subdividing a shared reference to a variant into a shared reference to the body object.

Rfn-Subdiv-ShrRef-Vrnt

$$\mathfrak{a}\colon \&^{\alpha}_{\mathsf{shr}}(\tau_0 + \tau_1) \vdash \mathfrak{a}.1 : \&^{\alpha}_{\mathsf{shr}}\,\tau_i \mid\mid \lambda \mathit{post}, (v).\ \exists\, w \text{ s.t. } \mathsf{inj}_i\, w = v.\ \ \mathit{post}\,(w)$$

Like Wkn-PlnRef-Vrnt-Triple, the predicate transformer ensures the precondition $\exists\, w.\ \mathsf{inj}_i\, w = v$ for any postcondition *post*.

We have the following rule for conditional branching on the tag of a variant object.

Rfn-Case-Vrnt-Access

$$\frac{\sqcap(\beta, \Delta) \mid \mathfrak{a}\colon {}^{\tau_+}\!/_{\tau_+} \vdash e : q.\ {}^{\&_{fq}(\tau_0+\tau_1)}\!/_{\&_{fq}(\tau_0+\tau_1)} \mid \mathit{get};\ \mathit{set} \qquad \forall\, i.\ \ \beta \mid \Delta;\ \mathfrak{a}\colon \tau_+,\ \Gamma \vdash e_i \colon \tau' \mid \Delta'; \Gamma' \mid \mathit{pre}_i}{\begin{array}{l}\beta \mid \Delta;\ \mathfrak{a}\colon \tau_+,\ \Gamma \vdash \mathsf{case}\ {*e}\ \mathsf{of}\ \{\, 0 \to e_0,\ 1 \to e_1 \,\} : \tau' \mid \Delta'; \Gamma' \mid \\ \quad \lambda \mathit{post}, (v, \vec{w}).\ \forall\, i.\ (\exists\, v'.\ \mathsf{inj}_i\, v' = \mathit{get}\, v) \implies \mathit{pre}_i\ \mathit{post}\ (\mathit{set}\, v\ (\mathit{get}\, v), \vec{w})\end{array}}$$

The access judgment says that, by performing e, we can access some plain reference to a variant object inside $\mathfrak{a}\colon \tau_+$ with the getter function *get*. We use a function on the fraction variable f, which is usually set to id or const 1. The value $\mathit{set}\, v\ (\mathit{get}\, v)$ is usually equal to v. If we go to the branch of i (0 or 1), we get the information that the variant sub-object v has the tag i as a postcondition.

We have the following structural rules on copyability and subtyping on the variant type.

Copy-Vrnt

$$\frac{\tau \text{ copy} \qquad \tau' \text{ copy}}{\tau + \tau' \text{ copy}}$$

Subty-Vrnt

$$\frac{\tau_0 \sqsubseteq^{\mathit{acc}} \tau_0' \mid f \qquad \tau_1 \sqsubseteq^{\mathit{acc}} \tau_1' \mid g}{\tau_0 + \tau_1 \sqsubseteq^{\mathit{acc}} \tau_0' + \tau_1' \mid f + g}$$

Here, the function $f + g$ is defined as follows.

$$f + g \ :=\ \lambda v.\ \mathsf{case}\ v\ \mathsf{of}\ \{\, \mathsf{inj}_0\, w_0 \to f\, w_0,\ \mathsf{inj}_1\, w_1 \to g\, w_1 \,\}$$

Injection Type We introduce refined typing rules for the injection type $\mathsf{in}\ (\tau, f)$ (§5.4).

We can subdivide a plain reference to an injection type by the following rule.

Wkn-Subdiv-PlnRef-InjTy

$$\mathfrak{a}\colon \&_q\, \mathsf{in}(\tau, f) \vdash \mathfrak{a}\colon \&_q\, \tau \mid \lambda \mathit{post}, (v).\ \forall\, w \text{ s.t. } f\, w = v.\ \ \mathit{post}\,(w)$$

Also, we can subdivide a shared reference to an injection type by the following rule.

Wkn-Subdiv-ShrRef-InjTy

$$\mathfrak{a}\colon \&^{\alpha}_{\mathsf{shr}}\, \mathsf{in}(\tau, f) \vdash \mathfrak{a}\colon \&^{\alpha}_{\mathsf{shr}}\, \tau \mid \lambda \mathit{post}, (v).\ \forall\, w \text{ s.t. } f\, w = v.\ \ \mathit{post}\,(w)$$

The injection type satisfies the following structural copyability rule.

Copy-InjTy

$$\frac{\tau \text{ copy}}{\mathsf{in}(\tau, f) \text{ copy}}$$

We have the following subtyping rules between τ and the injection type $\mathsf{in}(\tau, f)$.

Subty-Ty-InjTy

$$\tau \sqsubseteq^{\mathit{acc}} \mathsf{in}(\tau, f) \mid f$$

Subty-InjTy-Ty

$$\frac{g \circ f = \mathsf{id}}{\mathsf{in}(\tau, f) \sqsubseteq^{\mathit{acc}} \tau \mid g}$$

Combining the two rules, we can derive the following structural subtyping rule.

Subty-InjTy-InjTy

$$\frac{\tau \sqsubseteq^{\mathit{acc}} \tau' \mid h \qquad g \circ f = \mathsf{id}}{\mathsf{in}(\tau, f) \sqsubseteq^{\mathit{acc}} \mathsf{in}(\tau', f') \mid f' \circ h \circ g}$$

Recursive Type We introduce refined typing rules for the recursive type $\mathrm{rec}(\sigma, f)$ (§5.4).

We can prove subtyping on recursive types using *Löb induction* (Löb).

Example 6.1 (Subtyping on the List Type). For example, for the list type $\mathsf{list}\,\tau$ defined in Example 5.1, we can *derive* the following subtyping rule.

$$\frac{\tau \sqsubseteq^{acc} \tau' \mid f}{\mathsf{list}\,\tau \sqsubseteq^{acc} \mathsf{list}\,\tau' \mid \mathsf{map}\,f}$$

We apply the value transformer f to each element of the list by the function $\mathsf{map}\,f$.

Proof. Assume $\tau \sqsubseteq^{acc} \tau' \mid f$. By Löb induction (Löb), we can also assume $\triangleright \left(\mathsf{list}\,\tau \sqsubseteq^{acc} \mathsf{list}\,\tau' \mid \mathsf{map}\,f\right)$.

Unfolding $\mathsf{list}\,\tau$ by TyEq-List, by Subty-InjTy-InjTy, Subty-Vrnt, Subty-Pair and Subty-BoxPtr, we have the following subtyping judgment.

$$\mathsf{list}\,\tau \sqsubseteq^{acc} \mathsf{list}\,\tau' \mid \mathsf{in}_{\mathsf{List}} \circ (\mathsf{id} + f \times \mathsf{map}\,f) \circ \mathsf{out}_{\mathsf{List}}$$

The value transformer of this judgment is equal to $\mathsf{map}\,f$. Therefore, we obtain the expected judgment. □

6.2.4 Function Type

We introduce refined typing rules for the function type (§6.1.4). We also describe how we can specify and verify the while loop, based on the technique for recursive functions.

We introduce the following shorthand for specifying a constant value $\mathfrak{a}$.

$$\mathfrak{a}\colon \tau \mid v \quad := \quad \vdash \mathfrak{a} : \tau \mid\mid \lambda post, (). \; post\,(v)$$

Function Type We can specify a non-recursive function by the following rule.

Rfn-Fn-Nonrec

$$\frac{\forall \vec{\alpha}, \vec{\mathfrak{a}}, \beta. \; \square\left(\beta \sqsubseteq (\overrightarrow{\tau_\alpha}) \;\Rightarrow\; \beta \mid \overrightarrow{\mathfrak{a}\colon \tau_\alpha} \vdash e[\overrightarrow{\mathfrak{a}/a}]\colon \tau'_\alpha \mid pre\right)}{\mathsf{fn}(\vec{a})\{e\} : \forall \vec{\alpha}. \; \mathsf{fn}(\overrightarrow{\tau_\alpha}) \to \tau'_\alpha \mid pre}$$

If we want to set the pure value of the function to the predicate transformer *pre*, we just need to specify the body expression e (under arbitrary substitution) with the predicate transformer *pre*, using the refined typing judgment.

Extending Rfn-Fn-Nonrec, we get the following rule for specifying a *recursive* function.

Rfn-Fn-Rec

$$\frac{\begin{array}{c}\tau_+ \;=\; \forall \vec{\alpha}. \; \mathsf{fn}(\overrightarrow{\tau_\alpha}) \to \tau'_\alpha \\ \forall \vec{\alpha}, \mathfrak{b}, \vec{\mathfrak{a}}, \beta. \; \square\left(\beta \sqsubseteq (\overrightarrow{\tau_\alpha}) \;\Rightarrow\; \beta \mid \mathfrak{b}\colon \tau_+, \overrightarrow{\mathfrak{a}\colon \tau_\alpha} \vdash e[\mathfrak{b}/f, \overrightarrow{\mathfrak{a}/a}]\colon \tau'_\alpha \mid pre'\right) \\ pre \;\dot{\Rightarrow}\; \lambda post, (\vec{v}).\; pre'\; post\; (pre, \vec{v})\end{array}}{\mathsf{fn}\, f(\vec{a})\{e\} : \tau_+ \mid pre}$$

We want to prove that the function $\mathfrak{f}$ satisfies the predicate transformer *pre*. First, we specify the body expression e under arbitrary substitution with a predicate transformer *pre*′. When we verify the substituted expression, we can use the assumption that $\mathfrak{f}$ satisfies *pre*, because we can use that assumption under later by Löb induction (Löb) and strip off the later on the physical step of the function call (Hoare-FnCall). Therefore, it suffices to prove that *pre* is stronger than *pre*′ with the function pure value set to *pre*, i.e., $\lambda post, (\vec{v}).\; pre'\; post\; (pre, \vec{v})$. If *pre*′ satisfies monotonicity condition on the function

pure value, this is equivalent to the condition that *pre* is stronger than the *greatest fixed point* of pre' with a modified argument order $\lambda pre_*, post, (\vec{v}).\, pre'\, post\, (pre_*, \vec{v})$ (whose existence is ensured by Tarski's fixed point theorem).

We can specialize the lifetime parameters of the function type by the following subtyping rule.

Subty-Fn-Lft
$$\forall \vec{\alpha}.\ \mathsf{fn}(\overrightarrow{\tau_{\vec{\alpha}}}) \to \tau'_{\vec{\alpha}} \ \sqsubseteq\ \forall \vec{\beta}.\ \mathsf{fn}(\overrightarrow{\tau_{\overrightarrow{\alpha'_{\vec{\beta}}}}}) \to \tau'_{\overrightarrow{\alpha'_{\vec{\beta}}}} \ \mid\ \mathsf{id}$$

Here, each of the old lifetime parameters α_i is specialized into some lifetime $\alpha'_{\vec{\beta},i}$ that depends on the new lifetime parameters $\vec{\beta}$. In particular, we can use an empty sequence for $\vec{\beta}$.

A function call is specified by the following rule (we can specialize the lifetime parameters of the function beforehand by Subty-Fn-Lft).

Rfn-Fn-Call
$$\frac{\alpha \ \sqsubseteq\ (\vec{\tau})}{\alpha \mid \mathfrak{b}\colon \mathsf{fn}(\vec{\tau}) \to \tau',\ \overrightarrow{\mathfrak{a}\colon \tau} \vdash \mathfrak{b}(\vec{\mathfrak{a}})\colon \tau' \dashv\vdash \lambda post, (pre, \vec{v}).\ pre\ post\ (\vec{v})}$$

We can use any lifetime arguments $\vec{\beta}$ for the function call. The precondition required for the function call is the pure value of the function *pre* (predicate transformer) applied to the postcondition on the output *post* and the arguments $\vec{v}$. Note that we can derive the following rule from Rfn-Fn-Call and Rfn-EvCtx.

Rfn-Fn-Call-Const
$$\frac{\mathfrak{f}\colon \mathsf{fn}(\vec{\tau}) \to \tau' \qquad \alpha \ \sqsubseteq\ (\vec{\tau})}{\alpha \mid \overrightarrow{\mathfrak{a}\colon \tau} \vdash \mathfrak{f}(\vec{\mathfrak{a}})\colon \tau' \dashv\vdash \lambda post, (pre, \vec{v}).\ pre\ post\ (\vec{v})}$$

A function is copyable.

Copy-Fn
$$\forall \vec{\alpha}.\ \mathsf{fn}(\overrightarrow{\tau_{\vec{\alpha}}}) \to \tau'_{\vec{\alpha}}\ \mathsf{copy}$$

The function type satisfies the following structural subtyping rule.

Subty-Fn-Ty
$$\frac{\forall i, \vec{\alpha}.\ \triangleright\left(\tau'_{\vec{\alpha},i} \sqsubseteq^{\mathsf{own}} \tau_{\vec{\alpha},i} \mid f_i\right) \qquad \forall \vec{\alpha}.\ \triangleright\left(\tau^{+}_{\vec{\alpha}} \sqsubseteq^{\mathsf{own}} \tau^{+\prime}_{\vec{\alpha}} \mid g\right)}{\forall \vec{\alpha}.\ \mathsf{fn}(\overrightarrow{\tau_{\vec{\alpha}}}) \to \tau^{+}_{\vec{\alpha}} \ \sqsubseteq\ \forall \vec{\alpha}.\ \mathsf{fn}(\overrightarrow{\tau'_{\vec{\alpha}}}) \to \tau^{+\prime}_{\vec{\alpha}} \ \mid\ \lambda pre, post, (\vec{v}).\ pre\ (post \circ g)\ (\overrightarrow{f\ v})}$$

We can modify the input and output objects of the function by subtyping. Like Subty-BoxPtr, the assumption subtyping judgments on the input and output types are under the later modality.

While Loop The while loop $\mathsf{while}\, e\, \{e'\}$ is defined as follows.

$$\mathsf{while}\, e\, \{e'\} \quad := \quad \left(\mathsf{fn}\ while()\left\{\ \mathsf{if}\ e\ \{e';\ while()\}\ \right\}\right)()$$

We have the following refined typing rule on the while loop.

Rfn-While
$$\frac{\begin{array}{c}\alpha \mid \Delta; \Gamma \vdash e : \mathsf{bool} \mid \Delta; \Gamma \mid pre \qquad \alpha \mid \Delta; \Gamma \vdash e' \mid \Delta; \Gamma \mid pre' \\ pre_+ \ \dot{\Rightarrow}\ \lambda post, (\vec{v}).\ pre\left(\lambda(bl, \vec{v'}).\ \mathsf{if}\ bl\ \{pre'\ (pre_+\ post)\ (\vec{v'})\}\ \mathsf{else}\ \{pre_+\ post\ (\vec{v'})\}\right)(\vec{v})\end{array}}{\alpha \mid \Delta; \Gamma \vdash \mathsf{while}\, e\, \{e'\} \mid \Delta; \Gamma \mid pre_+}$$

Just like Rfn-Fn-Rec, we can use the predicate transformer pre_+ if pre_+ is stronger than the predicate transformer of the while loop with the predicate transformer for the recursive call set to pre_+.

We can freely add new control flows to this refined type system like this.

6.2.5 Vector Type

We introduce refined typing rules for the vector type $\mathsf{vec}\,\tau$ (§5.2).

We introduce the following function $\mathsf{new}_{\mathsf{vec}}$ for newly allocating an empty vector of some capacity.

$$\mathsf{new}_{\mathsf{vec}} \quad := \quad \mathsf{fn}(a)\,\{\,\mathsf{let}\ b = \mathsf{alloc}\,3\ \mathsf{in}\ b \leftarrow \mathsf{alloc}(a \times |\tau|);\ b.1 \leftarrow a;\ b.2 \leftarrow 0;\ b\,\}$$

It satisfies the following specification.

RFN-NEW-VEC
$$\mathsf{new}_{\mathsf{vec}} : \mathsf{fn}(\mathsf{int}) \to \mathsf{box}\,\mathsf{vec}\,\tau \ \mid\ \lambda \mathit{post}, (n).\ \mathit{post}\,\mathsf{nil}$$

We introduce the following function for taking out of the reference to a vector a a reference to the i-th element of the vector.

$$\mathsf{idx}^{\tau}_{\mathsf{vec}} \quad := \quad \mathsf{fn}(a, i)\,\big(\,(*a).(i \times |\tau|)\,\big)$$

Note that this function does not have *dynamic bounds checking* on the index input i.

The function $\mathsf{idx}^{\tau}_{\mathsf{vec}}$ satisfies the following specification for shared references.

RFN-IDX-VEC-SHRREF
$$\mathsf{idx}^{\tau}_{\mathsf{vec}} : \forall\,\alpha.\ \mathsf{fn}(\&^{\alpha}_{\mathsf{shr}}\,\mathsf{vec}\,\tau, \mathsf{int}) \to \&^{\alpha}_{\mathsf{shr}}\,\tau \ \vdash\ \lambda \mathit{post}, (v, i).\ \ 0 \le i < \mathsf{len}\,v \ \wedge\ \mathit{post}\,(v[i])$$

We can safely perform $\mathsf{idx}^{\tau}_{\mathsf{vec}}$ *without dynamic bounds checking*, because the precondition ensures that the index i is in the suitable range (the part $0 \le i < \mathsf{len}\,v$).

Update operations on the vector type are performed through *unique references* and discussed later in §6.3.6.

We have the following structural subtyping rule on the vector type.

SUBTY-VEC
$$\frac{\triangleright\,(\tau \sqsubseteq^{\mathit{acc}} \tau' \mid f)}{\mathsf{vec}\,\tau \sqsubseteq^{\mathit{acc}} \mathsf{vec}\,\tau' \mid \mathsf{map}\,f}$$

Like SUBTY-BOXPTR, the subtyping assumption on the target types can be under the later modality. We apply the value transformer f to each element of the vector by the function $\mathsf{map}\,f$. The function $\mathsf{map}\colon (T \to U) \to \mathsf{List}\,T \to \mathsf{List}\,U$ is defined as follows.

$$\mathsf{map}\,f\,\mathsf{nil} \ := \ \mathsf{nil} \qquad\qquad \mathsf{map}\,f\,(v :: w) \ := \ f\,v :: \mathsf{map}\,f\,w$$

6.3 Specifying and Verifying Operations on Unique References

Now we introduce refined typing rules on the unique reference type $\&^{\alpha}_{\mathsf{unq}}\,\tau$ (§5.3). We present detailed proofs on some of them, which contain interesting interactions of full borrow and prophecy.

6.3.1 Unique Borrow and Reborrow

We can borrow a full plain reference to get a unique reference by the following rule.

WKN-UNQBOR-PLNREF
$$\mathfrak{a} \colon \&_1\,\tau \ \vdash\ \mathfrak{a} \colon \&^{\alpha}_{\mathsf{unq}}\,\tau,\ \mathfrak{a} :^{\dagger\alpha} \&_1\,\tau \ \mid\ \lambda \mathit{post}, (v).\ \forall\, v_\circ.\ \mathit{post}\,((v, v_\circ), v_\circ)$$

We take a prophecy value $v_\circ$ for the unique borrow and set the values of the unique reference and the borrowed plain reference to $(v, v_\circ)$ and $v_\circ$, respectively. When we reclaim the plain reference after the lifetime α ends (SWKN-END-LOCALLFT-RECLAIM), it is ensured that $v_\circ$ has been resolved to the new value of the plain reference.

Wkn-UnqBor-PlnRef follows from the following lemma.

Raw-UnqBor-PlnRef
$$[\mathfrak{l}] \blacktriangleleft \&_1\, \tau\,\{\hat{v}\} \;\; \Rrightarrow_{\mathcal{N}_{\mathsf{lft}}+\mathcal{N}_{\mathsf{proph}}} \exists x.\; [\mathfrak{l}] \blacktriangleleft \&^{\alpha}_{\mathsf{unq}}\, \tau\,\{\lambda\pi.\,(\hat{v}\,\pi, \pi\, x)\} \ast [\mathfrak{l}] \blacktriangleleft^{\dagger\alpha} \&_1\, \tau\,\{\lambda\pi.\,\pi\, x\}$$

Proof of Raw-UnqBor-PlnRef. The plain reference is decomposed into (i) a full points-to token $\mathfrak{l} \overset{1}{\mapsto} \bar{\mathfrak{b}}$, (ii) the target object under later $\triangleright\, \bar{\mathfrak{b}} \blacktriangleleft \tau\,\{\hat{v}\}_d$, and (iii) a persistent time receipt $⧖(d+1)$, for some target cell values $\bar{\mathfrak{b}}$ and depth d.

By ValObs-ProphCtrl-Intro, for a fresh prophecy variable x satisfying $x.\mathsf{type} = \lfloor\tau\rfloor$, we get a value observer $\mathsf{VO}_x(\hat{v}, d)$ and a prophecy control $\mathsf{PC}(\hat{v}, d)$. We name the following proposition P, which is a separating conjunction of a full points-to token, a target object, a persistent time receipt and a prophecy control that is existentially quantified over the list of cell values, target value and depth.

$$P \;:=\; \exists \bar{\mathfrak{b}}', \hat{v}', d'.\; \mathfrak{l} \overset{1}{\mapsto} \bar{\mathfrak{b}}' \ast \bar{\mathfrak{b}}' \blacktriangleleft \tau\,\{\hat{v}'\}_{d'} \ast ⧖(d'+1) \ast \mathsf{PC}(x, \hat{v}', d')$$

We make $\triangleright P$ consuming the full points-to token, the target object *under later*, and the prophecy control, which is then turned by FullBor-Intro into a full borrow $\&^{\alpha}_{\mathsf{full}} P$ and a view shift $[\dagger\alpha] \Rrightarrow\!\!\ast_{\mathcal{N}_{\mathsf{lft}}} \triangleright P$. We turn the separating conjunction of the full borrow and the value observer into a unique reference $[\mathfrak{l}] \blacktriangleleft \&^{\alpha}_{\mathsf{unq}}\, \tau\,\{\lambda\pi.\,(\hat{v}\,\pi, \pi\, x)\}$.

Now let us construct the full plain reference under borrow $[\mathfrak{l}] \blacktriangleleft^{\dagger\alpha} \&_1\, \tau\,\{\lambda\pi.\,\pi\, x\}$ (defined in §6.1.1). Now assume we have $[\dagger\alpha]$. Consuming the view shift $[\dagger\alpha] \Rrightarrow\!\!\ast_{\mathcal{N}_{\mathsf{lft}}} \triangleright P$, we get $\triangleright P$. By consuming $\triangleright P$, we get a new full reference $[\mathfrak{l}] \blacktriangleleft \&_1\, \tau\,\{\hat{v}'\}$ and a prophecy control under later $\triangleright\, \mathsf{PC}(x, \hat{v}', d')$. The latter can be transformed into a prophecy equalizer under later $\triangleright\, \mathsf{PE}(\lambda\pi.\,\pi\, x, \hat{v}')$ by ProphCtrl-ProphEqz. Therefore, we finally get $[\mathfrak{l}] \blacktriangleleft^{\dagger\alpha} \&_1\, \tau\,\{\lambda\pi.\,\pi\, x\}$. □

Note that we can borrow only some component of a (possibly nested) pair by combining Wkn-UnqBor-PlnRef and Rfn-Split-UnqRef-Pair-R.

We can uniquely borrow a box pointer in an indirect way by combining Acc-BoxPtr-PlnRef, Rfn-Acc-Same and Wkn-UnqBor-PlnRef. Still, we can also have the following rule for uniquely borrowing a box pointer in a direct way.

Wkn-UnqBor-BoxPtr
$$\mathfrak{a} \colon \mathsf{box}\,\tau \;\vdash\; \mathfrak{a} \colon \&^{\alpha}_{\mathsf{unq}}\, \tau,\; \mathfrak{a} :^{\dagger\alpha} \mathsf{box}\,\tau \;\mathrel{⫞}\; \lambda\mathit{post}, (v).\; \forall v_\circ.\; \mathit{post}\,((v, v_\circ), v_\circ)$$

This rule follows from Raw-UnqBor-PlnRef and Raw-BoxPtr-PlnRef.

We can also create a unique reference by *reborrowing* from an existing *unique reference*.

Wkn-UnqBor-UnqRef
$$\mathfrak{a} \colon \&^{\beta}_{\mathsf{unq}}\, \tau \;\vdash\; \mathfrak{a} \colon \&^{\alpha}_{\mathsf{unq}}\, \tau,\; \mathfrak{a} :^{\dagger\alpha} \&^{\beta}_{\mathsf{unq}}\, \tau \;\mathrel{⫞}\; \lambda\mathit{post}, ((v, v'_\circ)).\; \forall v'_\circ.\; \mathit{post}\,((v, v_\circ), (v_\circ, v'_\circ))$$

Like Wkn-UnqBor-BoxPtr, we take a prophecy value $v_\circ$ for this borrow. The unique reference under borrow $\mathfrak{l} :^{\dagger\alpha} \&^{\beta}_{\mathsf{unq}}\, \tau$ takes the value $(v_\circ, v'_\circ)$, where the current-value part is set to the new prophecy value $v_\circ$ and the prophecy-value part remains the same as before the reborrow $v'_\circ$.

Wkn-UnqBor-UnqRef follows from the following lemma.

Raw-UnqBor-UnqRef
$$\frac{\alpha \sqsubseteq \beta}{[\mathfrak{l}] \blacktriangleleft \&^{\beta}_{\mathsf{unq}}\, \tau\,\{\lambda\pi.\,(\hat{v}\,\pi, \pi\, y)\} \;\; \Rrightarrow_{\mathcal{N}_{\mathsf{lft}}+\mathcal{N}_{\mathsf{proph}}} \exists x.\; [\mathfrak{l}] \blacktriangleleft \&^{\alpha}_{\mathsf{unq}}\, \tau\,\{\lambda\pi.\,(\hat{v}\,\pi, \pi\, x)\} \ast [\mathfrak{l}] \blacktriangleleft^{\dagger\alpha} \&^{\beta}_{\mathsf{unq}}\, \tau\,\{\lambda\pi.\,(\pi\, x, \pi\, y)\}}$$

Proof of Raw-UnqBor-UnqRef. We name the following proposition P'.

$$P' \;:=\; \exists \bar{\mathfrak{b}}'', \hat{v}'', d''.\ \mathfrak{l} \overset{1}{\mapsto} \bar{\mathfrak{b}}'' \ast \bar{\mathfrak{b}}'' \blacktriangleleft \tau\{\hat{v}''\}_{d''} \ast \boxtimes(d''+1) \ast \mathsf{PC}(y, \hat{v}'', d'')$$

Out of the unique reference, we get a value observer $\mathsf{VO}_y(\hat{v}, d)$, a full borrow $\&^{\beta}_{\mathsf{full}} P'$ and a persistent time receipt $\boxtimes(d+1)$ for some depth d.

The full borrow of the existing unique reference $\&^{\beta}_{\mathsf{full}} P'$ is turned by FullBor-Rebor into a new full borrow $\&^{\alpha}_{\mathsf{full}} P'$ and a view shift $[\alpha] \Rrightarrow\!\!\!\ast_{\mathcal{N}_{\mathsf{lft}}} \&^{\beta}_{\mathsf{full}} P'$. By ValObs-ProphCtrl-Intro, for a fresh prophecy variable x satisfying $x.\mathsf{type} = \lfloor \tau \rfloor$, we get a value observer $\mathsf{VO}_x(\hat{v}, d)$ and a prophecy control $\mathsf{PC}(x, \hat{v}, d)$. We name the following proposition Q.

$$Q \;:=\; \exists \hat{v}', d'.\ \mathsf{VO}_y(\hat{v}', d') \ast \boxtimes(d'+1) \ast \mathsf{PC}(x, \hat{v}', d')$$

We can create Q out of the value observer $\mathsf{VO}_y(\hat{v}, d)$ of the existing unique reference, the prophecy control we created $\mathsf{PC}(x, \hat{v}, d)$ and the persistent time receipt $\boxtimes(d+1)$. Then Q is turned into a full borrow $\&^{\alpha}_{\mathsf{full}} Q$ and a view shift $[\alpha] \Rrightarrow\!\!\!\ast_{\mathcal{N}_{\mathsf{lft}}} \triangleright Q$ (FullBor-Intro).

The two obtained view shifts are merged into $[\alpha] \Rrightarrow\!\!\!\ast_{\mathcal{N}_{\mathsf{lft}}} \&^{\beta}_{\mathsf{full}} P' \ast \triangleright Q$, which turns into the following.

$$[\alpha] \;\Rrightarrow\!\!\!\ast_{\mathcal{N}_{\mathsf{lft}}}\; \exists \hat{v}', d'.\ [\mathfrak{l}] \blacktriangleleft \&^{\beta}_{\mathsf{unq}} \tau\{\lambda\pi.(\hat{v}'\,\pi, \pi\, y)\}_{d'+1} \ast \boxtimes(d'+1) \ast \triangleright \mathsf{PC}(x, \hat{v}', d')$$

By ProphCtrl-ProphEqz, we can turn a prophecy control under later $\triangleright \mathsf{PC}(x, \hat{v}', d')$ into a prophecy equalizer under later $\triangleright \mathsf{PE}(\lambda\pi.\,\pi\, x, \hat{v}')$. Therefore, we get a unique reference under borrow $[\mathfrak{l}] \blacktriangleleft^{\dagger\alpha} \&^{\beta}_{\mathsf{unq}} \tau$.

The two obtained full borrows are merged by FullBor-Merge into $\&^{\alpha}_{\mathsf{full}}(P' \ast Q)$. We temporarily access the content $\triangleright (P' \ast Q)$ of the full borrow by FullBor-Subdiv, with help of a fractional lifetime token on α. The values $\hat{v}'', d''$ inside P' turn out to be equal to $\hat{v}', d'$ inside Q by agreement of the prophecy control of P' and the value observer of Q (ValObs-ProphCtrl-Agree). Out of $\triangleright (P' \ast Q)$, aside from the value observer and the prophecy control, we can take out $\triangleright P$, where P is defined as follows.

$$P \;:=\; \exists \bar{\mathfrak{b}}', \hat{v}', d'.\ \mathfrak{l} \overset{1}{\mapsto} \bar{\mathfrak{b}}' \ast \bar{\mathfrak{b}}' \blacktriangleleft \tau\{\hat{v}'\}_{d'} \ast \boxtimes(d'+1) \ast \mathsf{PC}(x, \hat{v}', d')$$

After we get back $\triangleright P$, by simultaneously updating the arguments of the value observer and the prophecy control (ValObs-ProphCtrl-Update) into the new internal values $\hat{v}', d'$ of P, we can reconstruct $\triangleright (P' \ast Q)$.

Therefore, by the power of FullBor-Subdiv, we finally get a new full borrow $\&^{\alpha}_{\mathsf{full}} P$. Combining the it with the value observer that we created $\mathsf{VO}_x(\hat{v}, d)$, we get the expected unique reference $[\mathfrak{l}] \blacktriangleleft \&^{\alpha}_{\mathsf{unq}} \tau\{\lambda\pi.(\hat{v}\,\pi, \pi\, x)\}$. □

6.3.2 Access on Unique References

We can temporarily take out a full plain reference from a unique reference.

Acc-UnqRef-PlnRef
$$\alpha \vdash \mathfrak{a} : {}^{\&^{\alpha}_{\mathsf{unq}}\tau}/_{\&^{\alpha}_{\mathsf{unq}}\tau} \vdash \mathfrak{a} : {}^{\&_1\tau}/_{\&_1\tau} \dashv \lambda(v, v_\circ).v;\ \lambda(v, v_\circ), w.(w, v_\circ)$$

The prophecy value $v_\circ$ is retained through this access. Unlike Acc-BoxPtr-PlnRef, the target type should be retained.

Acc-UnqRef-PlnRef follows from the following lemma.

Raw-UnqRef-PlnRef
$$[\mathfrak{l}] \blacktriangleleft \&^{\alpha}_{\mathsf{unq}} \tau\{\lambda\pi.(\hat{v}\,\pi, \pi\, x)\} \ast [\alpha]_q \;\Rrightarrow_{\mathcal{N}_{\mathsf{lft}}}\; [\mathfrak{l}] \blacktriangleleft \&_1 \tau\{\hat{v}\} \ast \forall \hat{w}.\ \big([\mathfrak{l}] \blacktriangleleft \&_1 \tau\{\hat{w}\} \;\Rrightarrow\!\!\!\ast_{\mathcal{N}_{\mathsf{lft}}}\; [\mathfrak{l}] \blacktriangleleft \&^{\alpha}_{\mathsf{unq}} \tau\{\lambda\pi.(\hat{w}\,\pi, \pi\, x)\} \ast [\alpha]_q\big)$$

Proof of Raw-UnqRef-PlnRef. We temporarily access the content of the full borrow of the unique reference using a partial lifetime token on α (FullBor-Access). By agreement of the value observer and the prophecy control (ValObs-ProphCtrl-Agree), the target value inside the full borrow turns out to be $\hat{v}$. Therefore, we can take out the full plain reference $[\mathfrak{l}] \blacktriangleleft \&_1\, \tau\, \{\hat{v}\}$.

Assume that we have got back $[\mathfrak{l}] \blacktriangleleft \&_1\, \tau\, \{\hat{w}\}$. We simultaneously update the arguments of the value observer and the prophecy control (ValObs-ProphCtrl-Update) into the new pure value $\hat{w}$ and depth. The full plain reference has the persistent time receipt we need for the full borrow and the unique reference itself. Therefore, we can close the full borrow, getting back $[\alpha]_q$, and reconstruct the unique reference with the new target value $[\mathfrak{l}] \blacktriangleleft \&^{\alpha}_{\mathsf{unq}}\, \tau\, \{\lambda \pi.\, (\hat{w}\, \pi, \pi\, x)\}$. □

We can also subdivide a plain reference to a unique reference.

Acc-Deref-PlnRef-UnqRef
$$\alpha \mid \mathfrak{a}\colon {}^{\&_q \&^{\alpha}_{\mathsf{unq}} \tau}/_{\&_q \&^{\alpha}_{\mathsf{unq}} \tau} \vdash *\mathfrak{a}\colon {}^{\&_1 \tau}/_{\&_1 \tau} \mid \lambda(v, v_\circ).\, v;\ \lambda(v, v_\circ), w.\, (w, v_\circ)$$

It can be proved using Raw-UnqRef-PlnRef.

Example 6.2 (Increment on a Unique Reference). For a simple example, we can increment the target integer of a unique reference.

$$\alpha \mid \mathfrak{a}\colon \&^{\alpha}_{\mathsf{unq}}\, \mathsf{int} \vdash \mathfrak{a} \leftarrow *\mathfrak{a} + 1 \mid \mathfrak{a}\colon \&^{\alpha}_{\mathsf{unq}}\, \mathsf{int} \mid \lambda post, ((n, n_\circ)).\ post\, ((n+1, n_\circ))$$

Proof. Let $(n, n_\circ)$ be the original pure value of the unique reference.

For the load $*\mathfrak{a}$, we temporarily get a plain reference by Acc-UnqRef-PlnRef and acquire an integer value by Rfn-Deref-PlnRef-Copy. We increment the integer value and bind it to some cell value $\mathfrak{b}$, which has the pure value $n + 1$.

Then for the store $\mathfrak{a} \leftarrow \mathfrak{b}$, we temporarily get a full plain reference again by Acc-UnqRef-PlnRef. Finally the pure value of the unique reference is updated into $(n + 1, n_\circ)$. □

6.3.3 Resolution of Unique References

When we release a unique reference, we can resolve its prophecy value into its current value.

SWkn-UnqRef-Release
$$\frac{\beta \sqsubseteq \&^{\alpha}_{\mathsf{unq}}\, \tau}{\beta \mid \mathfrak{a}\colon \&^{\alpha}_{\mathsf{unq}}\, \tau \vdash^{\#} \mid \lambda post, ((v, v_\circ)).\ v_\circ = v \implies post\, ()}$$

We get the information $v_\circ = v$ that the prophecy value $v_\circ$ is equal to the current value v in the postcondition. Note that we need to ensure that both the lifetime α and the lifetime of the type τ are alive.

SWkn-UnqRef-Release follows from the following lemma.

Raw-UnqRef-Release
$$\frac{\beta \sqsubseteq \&^{\alpha}_{\mathsf{unq}}\, \tau}{[\mathfrak{l}] \blacktriangleleft \&^{\alpha}_{\mathsf{unq}}\, \tau\, \{\lambda \pi.\, (\hat{v}\, \pi, \pi\, x)\}_{d+1} * [\beta]_q \ \Rrightarrow^{d}_{\mathcal{N}_{\mathsf{lft}} + \mathcal{N}_{\mathsf{proph}}} \ \langle \pi.\, \pi\, x = \hat{v}\, \pi \rangle * [\beta]_q}$$

Proof of Raw-UnqRef-Release. Out of the unique reference, we obtain a value observer $\mathrm{VO}_x(\hat{v}, d)$ (for some depth d) and a full borrow $\&^{\alpha}_{\mathsf{full}}\, P$, where P is defined as follows.

$$P \ :=\ \exists \bar{\mathfrak{b}}, \hat{v}', d'.\ \mathfrak{l} \mapsto^{1} \bar{\mathfrak{b}} * \bar{\mathfrak{b}} \blacktriangleleft \tau\, \{\hat{v}'\}_{d'} * \boxtimes(d'+1) * \mathrm{PC}(x, \hat{v}', d')$$

By FullBor-Access, we temporarily access $\triangleright P$ using a partial lifetime token on α. We open the internal values $\bar{\mathfrak{b}}, \hat{v}', d'$ of P. We get $\triangleright \mathrm{PC}(x, \hat{v}', d')$, whose later modality can be removed because we also have the value observer $\mathrm{VO}_x(\hat{v}, d)$ (ValObs-ProphCtrl-Timeless). By agreement with the value observer (ValObs-ProphCtrl-Agree), we know $\hat{v}' = \hat{v}$ and $d' = d$. We also get the target object $\bar{\mathfrak{b}} \blacktriangleleft \tau\,\{\hat{v}\}_d$.

From the target object, we temporarily take out a partial prophecy token $[Y]_q$ on a dependency Y of the target value $\hat{v}$ in d logical steps, with help of a partial lifetime token on τ (SemTy-Own-ProphToken). By ValObs-ProphCtrl-Resolve, consuming the value observer and the prophecy control, with help of the partial prophecy token $[Y]_q$, we get a prophecy observation $\langle \pi.\ \pi\, x = \hat{v}\, \pi \rangle$ and also a new prophecy control. Using the prophecy control, we can reconstruct $\triangleright P$, which is put back to the full borrow $\&^{\alpha}_{\mathsf{full}} P$. The prophecy observation $\langle \pi.\ \pi\, x = \hat{v}\, \pi \rangle$ serves as the postcondition $v_\circ = v$ for the unique reference $(v, v_\circ)$. □

Example 6.3 (Release of a Unique Reference after Increment). If we increment the target value of a unique reference $(n, n_\circ)$ and release the unique reference after that, we get the postcondition that the prophecy value $n_\circ$ is equal to $n + 1$.

$$\alpha \mid \mathfrak{a}\colon \&^{\alpha}_{\mathsf{unq}}\,\mathsf{int} \vdash \mathfrak{a} \leftarrow *\mathfrak{a} + 1;\ \lightning \mid\mid \lambda \mathit{post}, ((n, n_\circ)).\ n_\circ = n + 1 \Rightarrow \mathit{post}\,()$$

Proof. From SWkn-UnqRef-Release and the refined typing judgment of Example 6.2. We can use the super weakening judgment for SWkn-UnqRef-Release using the additional part '; $\lightning$' by Rfn-Seq. □

Also, when we turn a unique reference into a shared reference, we can resolve the prophecy value.

Rfn-UnqRef-ShrRef

$$\frac{\beta \sqsubseteq \&^{\alpha}_{\mathsf{unq}}\,\tau}{\beta \mid \mathfrak{a}\colon \&^{\alpha}_{\mathsf{unq}}\,\tau \vdash^{\#} \mathfrak{a}\colon \&^{\alpha}_{\mathsf{shr}}\,\tau \mid \lambda \mathit{post}, ((v, v_\circ)).\ v_\circ = v \Rightarrow \mathit{post}\,(v)}$$

It follows from the following lemma.

Raw-UnqRef-ShrRef

$$\frac{\beta \sqsubseteq \&^{\alpha}_{\mathsf{unq}}\,\tau}{[\mathfrak{a}] \blacktriangleleft \&^{\alpha}_{\mathsf{unq}}\,\tau\,\{\lambda \pi.\,(\hat{v}\,\pi, \pi\, x)\}_{d+1} * [\beta]_q \ \Rrightarrow^{2d}_{\mathcal{N}_{\mathsf{lft}}+\mathcal{N}_{\mathsf{proph}}} \langle \pi.\ \pi\, x = \hat{v}\, \pi \rangle * [\mathfrak{a}] \blacktriangleleft \&^{\alpha}_{\mathsf{shr}}\,\tau\,\{\hat{v}\}_{d+1} * [\beta]_q}$$

Proof of Raw-UnqRef-ShrRef. Just like the proof of Raw-UnqRef-Release, we consume the value observer and get a prophecy observation $\langle \pi.\ \pi\, x = \hat{v}\, \pi \rangle$, by temporarily accessing the content of the full borrow and also temporarily accessing the partial prophecy tokens of the target object spending d logical steps.

We can freeze the inner values (the cell values $\bar{\mathfrak{b}}$, pure value $\hat{v}'$ and depth d') of the full borrow (FullBor-Freeze). Temporarily accessing the full borrow, we know $\hat{v}' = \hat{v}$ and $d' = d$ by agreement of ValObs-ProphCtrl-Agree. Splitting the full borrow (FullBor-Split), we get full borrows of the full points-to token and the target ownership predicate. The full borrow of the full points-to token is turned into a fractured borrow of a fractional points-to token (FullBor-FracBor). The full borrow of the target ownership predicate is turned into the sharing predicate on the target object in d logical steps by SemTy-Own-Shr. Therefore, we finally get the expected shared reference. □

6.3.4 Subdivision of Unique References

By consuming a unique reference to an object, we can take unique references to some parts of the object. We call this operation *subdivision* of a unique reference.

A unique reference to a box pointer can be dereferenced into a unique reference without changing the pure value. In this simple case, we can just reuse the prophecy variable of the original unique reference.

Rfn-Deref-UnqRef-BoxPtr
$$\alpha \mid \mathfrak{a} \colon \&^{\alpha}_{\mathsf{unq}} \operatorname{box} \tau \vdash *\mathfrak{a} : \&^{\alpha}_{\mathsf{unq}} \tau \mid\mid \mathsf{id}$$

It follows from the following lemma.

Raw-Deref-UnqRef-BoxPtr
$$[\mathfrak{l}] \blacktriangleleft \&^{\alpha}_{\mathsf{unq}} \operatorname{box} \tau \, \{\lambda \pi. (\hat{v}\, \pi, \pi\, x)\}_{d+1} * [\alpha]_q \Rrightarrow_{\mathcal{N}_{\mathsf{Lft}}} \exists \mathfrak{l}'.$$
$$\mathfrak{l} \overset{1}{\mapsto} \mathfrak{l}' * \left(\mathfrak{l} \overset{1}{\mapsto} \mathfrak{l}' * \hourglass 1 \Rrightarrow_{\mathcal{N}_{\mathsf{lft}}} [\mathfrak{l}'] \blacktriangleleft \&^{\alpha}_{\mathsf{unq}} \tau \, \{\lambda \pi. (\hat{v}\, \pi, \pi\, x)\}_{d} * [\alpha]_q\right)$$

Proof of Raw-Deref-UnqRef-BoxPtr. The given unique reference has a full borrow $\&^{\alpha}_{\mathsf{full}} P$, where P can be written as follows.

$$P \;:=\; \exists \mathfrak{l}', \hat{v}', d'.\ \mathfrak{l} \overset{1}{\mapsto} \mathfrak{l}' * [\mathfrak{l}'] \blacktriangleleft \operatorname{box} \tau \, \{\hat{v}'\}_{d'+1} * \divideontimes(d'+2) * \mathsf{PC}(x, \hat{v}', d'+1)$$

We temporarily access the content $\triangleright P$ of the full borrow by FullBor-Subdiv, with help of a fractional lifetime token on α. In one logical step, we strip off the later modality and get P. We open the variables $\mathfrak{l}', \hat{v}', d'$ inside P. By agreement (ValObs-ProphCtrl-Agree), we know $\hat{v}' = \hat{v}$ and $d' = d - 1$. We temporarily take out $\mathfrak{l} \overset{1}{\mapsto} \mathfrak{l}'$.

Assume that we have retrieved $\mathfrak{l} \overset{1}{\mapsto} \mathfrak{l}'$ and also obtained a cumulative time receipt $\hourglass 1$. From the target box pointer $[\mathfrak{l}'] \blacktriangleleft \operatorname{box} \tau \, \{\hat{v}'\}_{d'+1}$, we can take out the full points-to token to the inner target $\mathfrak{l}' \overset{1}{\mapsto} \bar{\mathfrak{b}}$ and the inner target object under later $\triangleright\ \bar{\mathfrak{b}} \blacktriangleleft \operatorname{box} \tau \, \{\hat{v}'\}_{d'+1}$, for some $\bar{\mathfrak{b}}$. We also update the depth of the value observer and prophecy control from d into $d - 1$ (ValObs-ProphCtrl-Update). Now we can take out $\triangleright Q$, where we define Q as follows.

$$Q \;:=\; \exists \bar{\mathfrak{b}}, \hat{v}', d'.\ \mathfrak{l}' \overset{1}{\mapsto} \bar{\mathfrak{b}} * \triangleright \left(\bar{\mathfrak{b}} \blacktriangleleft \tau \, \{\hat{v}'\}_{d'}\right) * \divideontimes(d'+1) * \mathsf{PC}(x, \hat{v}', d')$$

After we get back $\triangleright Q$, we can reconstruct $\triangleright P$ in the following way. We update the pure value and depth of the value observer and prophecy control into $\hat{v}'$ and $d' + 1$, based on the variables $\hat{v}', d'$ inside Q. We bump up the persistent time receipt $\divideontimes(d'+1)$ of Q into the persistent time receipt $\divideontimes(d'+2)$ required by P using $\hourglass 1$ (CumuTime-SwellPersTime).

Therefore, by the power of FullBor-Subdiv, we get the full borrow $\&^{\alpha}_{\mathsf{full}} Q$. Combining it with the value observer $\mathsf{VO}_x(\hat{v}, d)$, we finally get the expected unique reference $[\mathfrak{l}'] \blacktriangleleft \&^{\alpha}_{\mathsf{unq}} \tau \, \{\hat{v}\}_{d+1}$. □

A unique reference to a pair can be subdivided into unique references to the elements of the pair.

Rfn-Split-UnqRef-Pair-R
$$\alpha \mid \mathfrak{a} \colon \&^{\alpha}_{\mathsf{unq}} (\tau_0 \times \tau_1) \vdash \mathfrak{a}.|\tau_0| : \&^{\alpha}_{\mathsf{unq}} \tau_1 \mid \mathfrak{a} \colon \&^{\alpha}_{\mathsf{unq}} \tau_0 \mid$$
$$\lambda \mathit{post}, ((v, v_\circ)).\ \mathit{post}\, ((v.1, v_\circ.1), (v.0, v_\circ.0))$$

In order to take two unique references of the addresses $\mathfrak{a}$ and $\mathfrak{a}.|\tau|$, we use a slightly tricky way here.

Rfn-Split-UnqRef-Pair-R follows from the following lemma.

Raw-Split-UnqRef-Pair
$$[\mathfrak{l}] \blacktriangleleft \&^{\alpha}_{\mathsf{unq}} (\tau_0 \times \tau_1) \, \{\lambda \pi. ((\hat{v}\, \pi), \pi\, x)\}_d * [\alpha]_q \Rrightarrow_{\mathcal{N}_{\mathsf{lft}} + \mathcal{N}_{\mathsf{proph}}} \exists y_0, y_1.$$
$$\langle \pi.\ \pi\, x = (\pi\, y_0, \pi\, y_1) \rangle * \mathbin{⅋}_i \left([\mathfrak{l} + i \cdot |\tau_0|] \blacktriangleleft \&^{\alpha}_{\mathsf{unq}} \tau_i \, \{\lambda \pi. ((\hat{v}\, \pi).i, \pi\, y_i)\}_d\right) * [\alpha]_q$$

We take new prophecy variables y_0, y_1 for the new unique references and *resolve* the original prophecy variable x into $\lambda\pi.\,(\pi\, y_0, \pi\, y_1)$.

Proof of Raw-Split-UnqRef-Pair. The given unique reference consists of a value observer $\mathsf{VO}_x(\lambda\pi.\,(\hat{v}\,\pi), d-1)$ and a full borrow $\&^\alpha_{\mathsf{full}}\, P$, where we define P as follows.

$$P \;:=\; \exists\, \bar{\mathfrak{b}}_0, \bar{\mathfrak{b}}_1, \hat{v}', d'.\; \ast_i \left(\mathfrak{l} + i \cdot |\tau_0| \overset{1}{\mapsto} \bar{\mathfrak{b}}_i \;\ast\; \bar{\mathfrak{b}}_i \blacktriangleleft \tau_i\, \{(.i) \circ \hat{v}'\}_{d'} \right) \;\ast\; \textstyle\bigtriangledown(d'+1) \;\ast\; \mathsf{PC}(x, \lambda\pi.\,(\hat{v}'\,\pi), d')$$

We temporarily access the content $\triangleright P$ of the full borrow using FullBor-Subdiv. We open the variables $\bar{\mathfrak{b}}_0, \bar{\mathfrak{b}}_1, \hat{v}', d'$ inside P. By agreement (ValObs-ProphCtrl-Agree), we know $\hat{v}' = \hat{v}$ and $d' = d-1$.

For each i, taking a fresh prophecy variable y_i, we create a value observer $\mathsf{VO}_{y_i}((.i) \circ \hat{v}, d-1)$ and a prophecy control $\mathsf{PC}(y_i, (.i) \circ \hat{v}, d-1)$ (ValObs-ProphCtrl-Intro). Now, by ValObs-ProphCtrl-Preresolve, consuming the value observer and prophecy control of the given unique reference, with help of the prophecy tokens $[y_i]_1$ (taken by ValObs-ProphCtrl-ProphToken), we acquire the prophecy observation $\langle \pi.\, \pi\, x = (\pi\, y_0, \pi\, y_1) \rangle$ and the following proposition R.

$$R \;:=\; \forall\, \hat{v}', d'.\;\; \mathsf{PE}\big(\lambda\pi.\,(\pi\, y_0, \pi\, y_1), \hat{v}'\big) \;\mathrel{-\!\!\ast}\; \mathsf{PC}(x, \hat{v}', d')$$

Using the remaining parts of $\triangleright P$ and the prophecy controls we created, we get $\triangleright\,(Q_0 \;\ast\; Q_1)$, where Q_i is defined as follows for $i = 0, 1$.

$$Q_i \;:=\; \exists\, \bar{\mathfrak{b}}_i, \hat{v}_i', d_i'.\;\; \mathfrak{l} + i \cdot |\tau_0| \overset{1}{\mapsto} \bar{\mathfrak{b}}_i \;\ast\; \bar{\mathfrak{b}}_i \blacktriangleleft \tau_i\, \{\hat{v}_i'\}_{d_i'} \;\ast\; \textstyle\bigtriangledown(d_i'+1) \;\ast\; \mathsf{PC}(y_i, \hat{v}_i', d_i')$$

Assume that we have retrieved $\triangleright\,(Q_0 \;\ast\; Q_1)$. We can reconstruct $\triangleright P$ in the following way. We open the variables $\bar{\mathfrak{b}}_i, \hat{v}_i', d_i'$ inside Q_i for $i = 0, 1$. For each i, we turn the prophecy control on y_i (under later) $\triangleright\, \mathsf{PC}(y_i, \hat{v}_i', d_i')$ owned by $\triangleright Q_i$ into a prophecy equalizer $\triangleright\, \mathsf{PE}(\lambda\pi.\,\pi\, y_i, \hat{v}_i')$ by ProphCtrl-ProphEqz. We then merge the two prophecy equalizers into $\triangleright\, \mathsf{PE}(\lambda\pi.\,(\pi\, y_0, \pi\, y_1), \lambda\pi.\,(\hat{v}_0'\,\pi, \hat{v}_1'\,\pi))$ by ProphEqz-Transform. By applying it to R, we get the prophecy control on x, $\triangleright\, \mathsf{PC}(x, \lambda\pi.\,(\hat{v}_0'\,\pi, \hat{v}_1'\,\pi), d')$, letting d' be $\max\{d_0', d_1'\}$. Therefore, separatingly conjoining this with the remaining parts of Q_0, Q_1, we can reconstruct $\triangleright P$.

Therefore, by the power of FullBor-Subdiv, we get the full borrow $\&^\alpha_{\mathsf{full}}(Q_0 \;\ast\; Q_1)$, which can be split into $\&^\alpha_{\mathsf{full}}\, Q_0$ and $\&^\alpha_{\mathsf{full}}\, Q_1$ (FullBor-Split). Combining the full borrows with the value observers we have created, we finally get the expected unique references. □

A unique reference to a variant can be subdivided into a unique reference to the body of the variant.

Rfn-Subdiv-UnqRef-Vrnt
$$\alpha \;\vdash\; \mathfrak{a}\colon \&^\alpha_{\mathsf{unq}}(\tau_0 + \tau_1) \;\vdash\; \mathfrak{a}.1\colon \&^\alpha_{\mathsf{unq}}\, \tau_i \;\vdash\; \mathrm{I}$$
$$\lambda\, post, ((v, v_\circ)).\;\; \exists\, w \text{ s.t. } \mathsf{inj}_i\, w = v.\;\; \forall\, w_\circ \text{ s.t. } \mathsf{inj}_i\, w_\circ = v_\circ.\;\; post\,((w, w_\circ))$$

It follows from the following lemma.

Raw-Subdiv-UnqRef-Vrnt
$$[\mathfrak{l}] \blacktriangleleft \&^\alpha_{\mathsf{unq}}(\tau_0 + \tau_1)\, \{\lambda\pi.\,(\mathsf{inj}_i(\hat{v}\,\pi), \pi\, x)\}_d \;\ast\; [\alpha]_q \;\;\Rrightarrow_{\mathcal{N}_{\mathsf{lft}}+\mathcal{N}_{\mathsf{proph}}}\; \exists\, y.$$
$$\langle \pi.\, \pi\, x = \mathsf{inj}_i(\pi\, y) \rangle \;\ast\; [\mathfrak{l}+1] \blacktriangleleft \&^\alpha_{\mathsf{unq}}\, \tau\, \{\lambda\pi.\,(\hat{v}\,\pi, \pi\, y)\}_d \;\ast\; [\alpha]_q$$

We take a new prophecy variable y for the new unique reference and *resolve* the original prophecy variable x into $\lambda\pi.\, \mathsf{inj}_i(\pi\, y)$.

Proof of Raw-Subdiv-UnqRef-Vrnt. Similar to the proof of Raw-Split-UnqRef-Pair. Note that we use the injectivity of inj_i when we apply ProphEqz-Transform. □

Also, a unique reference to an injection type can be subdivided as follows, in a similar way to Rfn-Subdiv-UnqRef-Vrnt.

Wkn-Subdiv-UnqRef-InjTy
$$\alpha \mid \mathfrak{a}\colon \&^{\alpha}_{\mathsf{unq}}\, \mathsf{in}(\tau, f) \vdash \mathfrak{a}\colon \&^{\alpha}_{\mathsf{unq}}\, \tau \mid \\ \lambda post, ((v, v_\circ)).\ \forall w \text{ s.t. } f\, w = v.\ \forall w_\circ \text{ s.t. } f\, w_\circ = v_\circ.\ post\, ((w, w_\circ))$$

Note that we can use universal quantification instead of existential quantification in the part '$\forall w$ s.t. $f\, w = v$.', unlike Rfn-Subdiv-UnqRef-Vrnt. It follows from the following lemma.

Raw-Subdiv-UnqRef-InjTy
$$[\mathfrak{l}] \blacktriangleleft \&^{\alpha}_{\mathsf{unq}}\, \mathsf{in}(\tau, f)\, \{\lambda\pi.(f\, (\hat{v}\, \pi), \pi\, x)\}_d \ast [\alpha]_q \quad \Rrightarrow_{\mathcal{N}_{\mathsf{lft}}+\mathcal{N}_{\mathsf{proph}}} \exists y. \\ \langle \pi.\ \pi\, x = f\, (\pi\, y)\rangle \ast [\mathfrak{l}] \blacktriangleleft \&^{\alpha}_{\mathsf{unq}}\, \tau\, \{\lambda\pi.(\hat{v}\, \pi, \pi\, y)\}_d \ast [\alpha]_q$$

Proof of Raw-Subdiv-UnqRef-InjTy. Just analogous to the proof of Raw-Subdiv-UnqRef-Vrnt. Note that the function f is injective. □

The rule Wkn-Subdiv-UnqRef-InjTy is useful for modifying a unique reference to a recursive type.

A unique reference to a unique reference (which we call a double unique reference) can be dereferenced into a unique reference.

Rfn-Deref-UnqRef-UnqRef
$$\alpha \sqcap \beta \mid \mathfrak{a}\colon \&^{\alpha}_{\mathsf{unq}}\, \&^{\beta}_{\mathsf{unq}}\, \tau \vdash *\mathfrak{a}\colon \&^{\alpha\sqcap\beta}_{\mathsf{unq}}\, \tau \mid \mid \\ \lambda post, (((v, v^\circ), (v_\circ, v^\circ_\circ))).\ v^\circ_\circ = v^\circ \Rightarrow post\, ((v, v_\circ))$$

We handle prophecy values in an interesting way. The current and prophecy values $v, v_\circ$ of the resulting unique reference are the current values of the current and prophecy values of the given double unique reference, respectively. We get the postcondition $v^\circ_\circ = v^\circ$ that the prophecy values $v^\circ, v^\circ_\circ$ of the current and prophecy values are equal.

Raw-Deref-UnqRef-UnqRef follows from the following lemma.

Raw-Deref-UnqRef-UnqRef
$$[\mathfrak{l}] \blacktriangleleft \&^{\alpha}_{\mathsf{unq}}\, \&^{\beta}_{\mathsf{unq}}\, \tau\, \{\lambda\pi.((\hat{v}\, \pi, \pi\, y), \pi\, x)\}_{d+1} \ast [\alpha \sqcap \beta]_q \quad \Rrightarrow\!\!\!\!\Rrightarrow_{\mathcal{N}_{\mathsf{lft}}+\mathcal{N}_{\mathsf{proph}}} \exists \mathfrak{l}', z. \\ \mathfrak{l} \overset{1}{\mapsto} \mathfrak{l}' \ast \langle \pi.\ \pi\, x = (\pi\, z, \pi\, y)\rangle \ast \\ \left(\mathfrak{l} \overset{1}{\mapsto} \mathfrak{l}' \ast \mathbin{⧗} 1 \Rrightarrow_{\mathcal{N}_{\mathsf{lft}}} [\mathfrak{l}'] \blacktriangleleft \&^{\alpha\sqcap\beta}_{\mathsf{unq}}\, \tau\, \{\lambda\pi.(\hat{v}\, \pi, \pi\, z)\}_d \ast [\alpha \sqcap \beta]_q\right)$$

Proof of Raw-Deref-UnqRef-UnqRef. The given double unique reference is decomposed into the value observer $\mathsf{VO}_x(\lambda\pi.(\hat{v}\, \pi, \pi\, y), d)$ and the full borrow that can be written as follows.

$$\&^{\alpha}_{\mathsf{full}} \left(\exists \mathfrak{l}', \hat{v}', y, d'.\ \mathfrak{l} \overset{1}{\mapsto} \mathfrak{l}' \ast [\mathfrak{l}'] \blacktriangleleft \&^{\beta}_{\mathsf{unq}}\, \tau\, \{\lambda\pi.(\hat{v}'\, \pi, \pi\, y)\}_{d'+1} \ast \right. \\ \left. ⧖(d'+2) \ast \mathsf{PC}(y, \hat{v}', d'+1)\right)$$

We freeze the inner variables $\mathfrak{l}', y$ of the full borrow (FullBor-Freeze) and split the full borrow (by FullBor-Split) into the following two.

$$\&^{\alpha}_{\mathsf{full}} \left(\exists \hat{v}', d'.\ \mathfrak{l} \overset{1}{\mapsto} \mathfrak{l}' \ast \mathsf{VO}_y(\hat{v}', d') \ast \mathsf{PC}(x, \lambda\pi.(\hat{v}'\, \pi, \pi\, y), d'+1) \ast ⧖(d'+2)\right) \\ \&^{\alpha}_{\mathsf{full}}\, \&^{\beta}_{\mathsf{full}} \left(\exists \bar{\mathfrak{b}}, \hat{v}', d'.\ \mathfrak{l} \overset{1}{\mapsto} \bar{\mathfrak{b}} \ast \bar{\mathfrak{b}} \blacktriangleleft \tau\, \{\hat{v}'\}_{d'} \ast ⧖(d'+1) \ast \mathsf{PC}(y, \hat{v}', d')\right)$$

We unnest the latter in one logical step (FullBor-Unnest) and then merge it with the former (FullBor-Merge) to get the full borrow $\&_{\mathsf{full}}^{\alpha \sqcap \beta} P$, where we define P as follows.

$$\begin{aligned} P \;:=\; & \exists\, \bar{\mathfrak{b}}, \hat{v}', d'.\ \mathfrak{l} \overset{1}{\mapsto} \mathfrak{l}' \;*\; \mathfrak{l}' \overset{1}{\mapsto} \bar{\mathfrak{b}} \;*\; \bar{\mathfrak{b}} \blacktriangleleft \tau\,\{\hat{v}'\}_{d'} \;*\; \textstyle\bigtimes(d'+2) \\ & \mathsf{VO}_y(\hat{v}', d') \;*\; \mathsf{PC}(y, \hat{v}', d') \;*\; \mathsf{PC}(x, \lambda\pi.(\hat{v}'\,\pi, \pi\, y), d'+1) \end{aligned}$$

We access the content $\triangleright P$ of this full borrow by FullBor-Subdiv. We can temporarily take out the points-to token $\mathfrak{l} \overset{1}{\mapsto} \mathfrak{l}'$.

Assume we have retrieved $\mathfrak{l} \overset{1}{\mapsto} \mathfrak{l}'$ and also obtained a cumulative time receipt $\textstyle\bigtimes 1$. By agreement of the value observer and prophecy control on x (ValObs-ProphCtrl-Agree), we know that d is positive. Taking a fresh prophecy variable z, we create a value observer $\mathsf{VO}_z(\hat{v}, d-1)$ and a prophecy control $\mathsf{PC}(z, \hat{v}, d-1)$. By ValObs-ProphCtrl-Preresolve, consuming the value observer and prophecy control of the given double unique reference, with help of prophecy tokens $[y]_1$ and $[z]_1$ (taken by ValObs-ProphCtrl-PropToken), we obtain the prophecy observation $\langle \pi.\ \pi\, x = (\pi\, z, \pi\, y) \rangle$ and the following proposition R.

$$R \;:=\; \forall\, \hat{v}', d'.\ \mathsf{PE}\big(\lambda\pi.(\pi\, z, \pi\, y), \hat{v}'\big) \mathrel{-\!\!*} \mathsf{PC}(x, \hat{v}', d')$$

From the remaining parts of $\triangleright P$ and the created prophecy control $\mathsf{PC}(z, \hat{v}, d-1)$, we can take out $\triangleright Q$, where we define Q as follows.

$$Q \;:=\; \exists\, \bar{\mathfrak{b}}, \hat{v}', d'.\ \mathfrak{l}' \overset{1}{\mapsto} \bar{\mathfrak{b}} \;*\; \bar{\mathfrak{b}} \blacktriangleleft \tau\,\{\hat{v}'\}_{d'} \;*\; \textstyle\bigtimes(d'+1) \;*\; \mathsf{PC}(z, \hat{v}', d')$$

When we get back $\triangleright Q$, we can reconstruct $\triangleright P$ in the following way. We open the variables $\bar{\mathfrak{b}}, \hat{v}', d'$ of Q. We turn the prophecy control on z (under later) $\triangleright \mathsf{PC}(z, \hat{v}', d')$ of $\triangleright Q$ into the prophecy equalizer $\triangleright \mathsf{PE}(\lambda\pi.\,\pi\, z, \hat{v}')$ (ProphCtrl-ProphEqz) and then turn it into $\triangleright \mathsf{PE}(\lambda\pi.(\pi\, z, \pi\, y), \lambda\pi.(\hat{v}'\,\pi, \pi\, y))$ (ProphEqz-Transform); applying it to R, we get the prophecy control on x, $\triangleright \mathsf{PC}(x, \hat{v}, d+2)$. We update the arguments of the value observer and prophecy control on y into $\hat{v}'$ and d' (ValObs-ProphCtrl-Update). We bump up the persistent time receipt $\textstyle\bigtimes(d'+1)$ of Q into $\textstyle\bigtimes(d'+2)$ required by P by consuming $\textstyle\bigtimes 1$ (CumuTime-Swell-PersTime).

Therefore, by the power of FullBor-Subdiv, we get the full borrow $\&_{\mathsf{full}}^{\alpha \sqcap \beta} Q$. Combining it with the value observer that we created $\mathsf{VO}_z(\hat{v}, d-1)$, we finally get the expected unique reference $\&_{\mathsf{unq}}^{\alpha \sqcap \beta} \tau\,\{\lambda\pi.(\hat{v}\,\pi, \pi\, z)\}_d$. □

6.3.5 Subtyping on Unique References

We have the following structural subtyping rule on the unique reference type under the access mode own.

$$\textsc{Subty-Own-UnqRef}\quad \frac{\triangleright\big(\tau \sqsubseteq^{\mathsf{own}} \tau' \mid \mathsf{id}\big) \qquad \triangleright\big(\tau' \sqsubseteq^{\mathsf{own}} \tau \mid \mathsf{id}\big) \qquad \beta \sqsubseteq \alpha}{\&_{\mathsf{unq}}^{\alpha} \tau \sqsubseteq^{\mathsf{own}} \&_{\mathsf{unq}}^{\beta} \tau' \mid \mathsf{id}}$$

Again, the subtyping assumptions on the target types can be under the later modality.

Proof. From FullBor-Iff and FullBor-Mono-Lft. □

Using the later modality in the assumption of Subty-Own-UnqRef, with Löb induction (Löb), we can prove subtyping rules (under the access mode own) for recursive types with self reference under the unique reference type.

Example 6.4 (Subtyping for Recursion Under Unique References). For example, we can derive the following subtyping rule on the type mnat_α (defined in Example 5.2), a Peano number type with self reference under the unique reference type.

$$\frac{\alpha \sqsubseteq \beta \qquad \beta \sqsubseteq \alpha}{\mathsf{mnat}_\alpha \sqsubseteq^{\mathsf{own}} \mathsf{mnat}_\beta \mid \mathsf{id}}$$

Proof. Assume $\alpha \sqsubseteq \beta$ and $\beta \sqsubseteq \alpha$. We prove both $P := \mathsf{mnat}_\alpha \sqsubseteq^{\mathsf{own}} \mathsf{mnat}_\beta \mid \mathsf{id}$ and $Q := \mathsf{mnat}_\beta \sqsubseteq^{\mathsf{own}} \mathsf{mnat}_\alpha \mid \mathsf{id}$. By Löb induction (Löb), we can also assume $\triangleright(P \land Q)$, which entails $\triangleright P$ and $\triangleright Q$.

By Subty-Own-UnqRef, we can prove P from $\triangleright P$, $\triangleright Q$ and $\beta \sqsubseteq \alpha$. Similarly we can prove Q. Therefore, we have proved both P and Q. □

Under the sharing access mode, we have the following more liberated subtyping rule on the unique reference type.

Subty-Shr-UnqRef
$$\frac{\triangleright\left(\tau \sqsubseteq^{\mathsf{shr}(\alpha \sqcap \beta)} \tau' \mid f\right) \qquad \alpha' \sqsubseteq \alpha}{\&^{\alpha}_{\mathsf{unq}}\, \tau \sqsubseteq^{\mathsf{shr}(\beta)} \&^{\alpha'}_{\mathsf{unq}}\, \tau' \mid \lambda(v, v_\circ).\, (f\, v, g\, v_\circ)}$$

We can modify the prophecy part $v_\circ$ *using any function* g, which follows from Dep-Construct (see the model of the unique reference in §5.3).

Also, a unique reference can be transformed into a shared reference under the sharing access mode.

Subty-Shr-UnqRef-ShrRef
$$\&^{\beta}_{\mathsf{unq}}\, \tau \sqsubseteq^{\mathsf{shr}(\alpha)} \&^{\alpha \sqcap \beta}_{\mathsf{shr}}\, \tau \mid \lambda(v, v_\circ).\, v$$

Combining this with Rfn-Deref-ShrRef-ShrRef, we can derive the following dereference rule.

Rfn-Deref-ShrRef-UnqRef
$$\frac{\beta \sqsubseteq \alpha}{\beta \mid \mathfrak{a} \colon \&^{\alpha}_{\mathsf{shr}}\, \&^{\alpha'}_{\mathsf{unq}}\, \tau \vdash *\mathfrak{a} : \&^{\alpha \sqcap \alpha'}_{\mathsf{shr}}\, \tau \mid\mid \lambda post, ((v, v_\circ)).\, post\,(v)}$$

6.3.6 Manipulating Vectors via Unique References

Now we introduce refined typing rules for combination of the vector type $\mathsf{vec}\,\tau$ (§5.2) and the unique reference type $\&^{\alpha}_{\mathsf{unq}}\,\tau$.

A function that takes a unique reference to a vector, updates the vector, and just releases the unique reference can be verified using Raw-UnqRef-PlnRef and Raw-UnqRef-Release. Although we need to work on the model of the vector type, we need no other rules on unique references.

For example, we can introduce the following function $\mathsf{pop}^{\tau}_{\mathsf{vec}}$ that takes a unique reference to a vector a and moves out the last element from the vector making an new box pointer.

$$\begin{aligned}\mathsf{pop}^{\tau}_{\mathsf{vec}} \;:=\; & \mathsf{fn}(a)\,\{\, \mathsf{let}\ b = \mathsf{alloc}\,|\tau|\ \mathsf{in} \\ & \quad \mathsf{let}\ l = *(a.2) - 1\ \mathsf{in}\ \ a.2 \leftarrow l;\ b \leftarrow^{*}_{|\tau|} (*a).(l \times |\tau|);\ b\,\}\end{aligned}$$

This function satisfies the following specification.

Rfn-Pop-Vec
$$\mathsf{pop}^{\tau}_{\mathsf{vec}} : \forall\, \alpha.\ \mathsf{fn}(\&^{\alpha}_{\mathsf{unq}}\, \mathsf{vec}\,\tau) \to \mathsf{box}\,\tau \mid \lambda post, ((v, v_\circ)).\ \exists v', v''\ \text{s.t.}\ v' +\!\!+ [v''] = v.\ v_\circ = v' \implies post\,(v'')$$

Note that the precondition requires that the vector is non-empty. In order to prove this specification, we simply take out a full plain reference from the input unique reference by Raw-UnqRef-PlnRef, verify the operations based on the model of the type, and then resolve the prophecy value by Raw-UnqRef-Release.

Also, we can introduce the following function $\mathsf{push}^{\tau}_{\mathsf{vec}}$ that takes a unique reference to a vector a and an ownership b and pushes the target object of b to the end of the vector.

$$\begin{aligned}&\mathsf{push}^{\tau}_{\mathsf{vec}} \;:=\; \mathsf{fn}(a,b)\,\big\{\,\mathsf{let}\ c = *(a.1)\ \mathsf{in}\ \mathsf{let}\ l = *(a.2)\ \mathsf{in}\\ &\quad \mathsf{if}\ c = l\ \big\{\,\mathsf{let}\ a' = \mathsf{alloc}\,((c+1)\times|\tau|)\ \mathsf{in}\ a' \leftarrow^{*}_{c\times|\tau|} *a;\ a \leftarrow a';\ a.1 \leftarrow c+1\,\big\};\\ &\quad (*a).(l\times|\tau|) \leftarrow^{*}_{|\tau|} b;\ \mathsf{free}\,b;\ a.2 \leftarrow l+1\,\big\}\end{aligned}$$

If we have no room in the capacity $c = l$, we increment the capacity size and reallocate the memory block. We move the target object to the end address of the vector, free the memory block of b, and increment the length of the vector. The function $\mathsf{push}^{\tau}_{\mathsf{vec}}$ satisfies the following specification.

Rfn-Push-Vec

$$\mathsf{push}^{\tau}_{\mathsf{vec}} : \forall\alpha.\ \mathsf{fn}(\&^{\alpha}_{\mathsf{unq}}\,\mathsf{vec}\,\tau,\ \mathsf{box}\,\tau)\ \shortmid\ \lambda post, ((v, v_\circ), v').\ v_\circ = v +\!\!+ [v'] \implies post\,()$$

Again, the rules we need about the unique reference are just Raw-UnqRef-PlnRef and Raw-UnqRef-Release, although we need to work on the big expression of $\mathsf{push}^{\tau}_{\mathsf{vec}}$ and the model of the vector type.

We can also *subdivide* a unique reference to a vector into unique references to each element of the vector, which requires some new lemma on unique references. We can introduce the following lemma for that subdivision.

Raw-Split-UnqRef-Vec

$$\begin{aligned}&\mathfrak{l} \blacktriangleleft \&^{\alpha}_{\mathsf{unq}}\,\mathsf{vec}\,\tau\,\{\lambda\pi.([\overrightarrow{\hat{v}\,\pi}], \pi\,x)\}_{d+1} \ast [\alpha]_q \;\Rrightarrow_{\mathcal{N}_{\mathsf{lft}}+\mathcal{N}_{\mathsf{proph}}} \exists\,\mathfrak{l}', \vec{y}.\\ &\quad \mathfrak{l} \overset{1}{\mapsto} \mathfrak{l}' \ast \langle \pi.\ \pi\,x = [\overrightarrow{\pi\,y}]\rangle \ast\\ &\quad \big(\mathfrak{l} \overset{1}{\mapsto} \mathfrak{l}' \ast \text{⧗}1 \Rrightarrow_{\mathcal{N}_{\mathsf{lft}}} \textstyle\bigast_i \big([\mathfrak{l}' + i\cdot|\tau|] \blacktriangleleft \&^{\alpha}_{\mathsf{unq}}\,\tau\,\{\lambda\pi.(\hat{v}_i\,\pi, \pi\,y_i)\}_d\big) \ast [\alpha]_q\big)\end{aligned}$$

Proof. Similar to Raw-Deref-UnqRef-BoxPtr and Raw-Split-UnqRef-Pair. □

Combining Raw-Split-UnqRef-Vec and Raw-UnqRef-Release, we can prove the following lemma for taking out of a unique reference to a vector a unique reference to a specific element of the vector. The prophecy variables of the remaining unique references are resolved.

Raw-Vec-Idx-UnqRef

$$\dfrac{\vec{\hat{v}} = \hat{v}_0, \ldots, \hat{v}_{n-1} \qquad 0 \le i < n \qquad \beta \sqsubseteq \&^{\alpha}_{\mathsf{unq}}\,\mathsf{vec}\,\tau}{\begin{aligned}&\mathfrak{l} \blacktriangleleft \&^{\alpha}_{\mathsf{unq}}\,\mathsf{vec}\,\tau\,\{\lambda\pi.([\overrightarrow{\hat{v}\,\pi}, \pi\,x])\}_{d+1} \ast [\alpha]_q \;\Rrightarrow_{\mathcal{N}_{\mathsf{lft}}+\mathcal{N}_{\mathsf{proph}}} \exists\,\mathfrak{l}'.\ \ \mathfrak{l} \overset{1}{\mapsto} \mathfrak{l}' \ast\\ &\big(\mathfrak{l} \overset{1}{\mapsto} \mathfrak{l}' \ast \text{⧗}1 \Rrightarrow\!\!\ast^{d}_{\mathcal{N}_{\mathsf{lft}}} \exists y.\\ &\quad \langle \pi.\ \pi\,x = [\overrightarrow{\hat{v}\,\pi}]\{i \leftarrow \pi\,y\}\rangle \ast [\mathfrak{l}' + i\cdot|\tau|] \blacktriangleleft \&^{\alpha}_{\mathsf{unq}}\,\tau\,\{\lambda\pi.(\hat{v}_i\,\pi, \pi\,y)\}_d \ast [\alpha]_q\big)\end{aligned}}$$

Note that we can obtain $\beta \sqsubseteq \alpha$ out of $\beta \sqsubseteq \&^{\alpha}_{\mathsf{unq}}\,\mathsf{vec}\,\tau$ in two logical steps. This lemma leads to the following refined typing rule for the function $\mathsf{idx}^{\tau}_{\mathsf{vec}}$ (defined in §6.2.5).

Rfn-Idx-Vec-UnqRef

$$\begin{aligned}&\mathsf{idx}^{\tau}_{\mathsf{vec}} : \forall\alpha.\ \mathsf{fn}(\&^{\alpha}_{\mathsf{unq}}\,\mathsf{vec}\,\tau, \mathsf{int}) \to \&^{\alpha}_{\mathsf{unq}}\,\tau\ \shortmid\\ &\quad \lambda post, ((v, v_\circ), i).\ \ 0 \le i < \mathsf{len}\,v\ \land\ \forall w_\circ.\ v_\circ = v\{i \leftarrow w_\circ\} \implies post\,((v[i], w_\circ))\end{aligned}$$

The precondition ensures that the index i is within the bound of the current list v ($0 \le i < \mathsf{len}\,v$). We take a prophecy value $w_\circ$ for the newly taken unique reference to the

i-th element. The new unique reference has the value $(v[i], w_\circ)$. We set the prophecy value $v_\circ$ on the vector to the current value v with the i-th element updated to $w_\circ$.

We can also write a following *higher-order function* that inputs a reference a to a vector and a function f and calls f on a reference to each element of the vector.

$$\mathsf{iter}^\tau_{\mathsf{vec}} \;:=\; \mathsf{fn}(a, f)\,\{\,\mathsf{let}\ a' = *a\ \mathsf{in}\ \mathsf{for}\ i \leftarrow 0\,..\,*(a.2)\,\{\,f(a'.(i \times |\tau|))\,\}\,\}$$

Using Raw-Split-UnqRef-Vec and Raw-UnqRef-Release, we can prove the following refined typing rule on this function.

Rfn-Iter-Vec-UnqRef
$$\mathsf{iter}^\tau_{\mathsf{vec}} : \forall\,\alpha.\ \mathsf{fn}\big(\&^\alpha_{\mathsf{unq}}\,\mathsf{vec}\,\tau,\ \mathsf{fn}(\&^\alpha_{\mathsf{unq}}\,\tau)\big)\ \mathsf{I}\ \lambda\, post, ((v, v_\circ), pre).\ \mathsf{len}\,v_\circ = \mathsf{len}\,v \implies \mathsf{foldr}\,\big(\lambda w, post', (()).\ pre\ post'\,(w)\big)\,(\mathsf{zip}\,v\,v_\circ)\ post\,(())$$

By Raw-Split-UnqRef-Vec, we get the postcondition $\mathsf{len}\,v_\circ = \mathsf{len}\,v$ and have the unique references $(v[i], v_\circ[i])$ (for $0 \le i < \mathsf{len}\,v$). In order to compose the predicate transformers on the unique reference, we use here $\mathsf{foldr}\colon (T \to U \to U) \to \mathsf{List}\,T \to U \to U$ and $\mathsf{zip}\colon \mathsf{List}\,T \to \mathsf{List}\,T \to \mathsf{List}(T \times T)$, which are defined as follows.

$$\mathsf{foldr}\,f\,\mathsf{nil}\,a \;:=\; a \qquad \mathsf{foldr}\,f\,(v :: w)\,a \;:=\; f\,v\,(\mathsf{foldr}\,f\,w\,a)$$

$$\mathsf{zip}\,\mathsf{nil}\,w \;=\; \mathsf{zip}\,v\,\mathsf{nil} \;:=\; \mathsf{nil} \qquad \mathsf{zip}\,(v :: v')\,(w :: w') \;:=\; (v, w) :: \mathsf{zip}\,v'\,w'$$

6.4 Examples of Verification by the Refined Type System

In this section, we verify some Rust programs using the refined typing rules introduced in the previous sections, §6.1, §6.2 and §6.3.

Example 6.5 (Dynamic Decision of the Address of a Unique Reference). The verification problem Example 1.9 verified by RustHorn can also be verified in our refined type system.

The function `take_max` can be formalized as the following function take_max.

$$\mathsf{take\_max} \;:=\; \mathsf{fn}(a, b)\,\{\,\mathsf{if}\ *a \ge *b\ \{\lightning;\ a\}\ \mathsf{else}\ \{\lightning;\ b\}\,\}$$

It takes two unique references to an integer a, b and returns the one with the greater value, releasing the other. The point is that the returned unique reference (a or b) is determined by a dynamic condition $*a \ge *b$. This function satisfies the following specification.

$$\mathsf{take\_max} : \forall\,\alpha.\ \mathsf{fn}(\&^\alpha_{\mathsf{unq}}\,\mathsf{int}, \&^\alpha_{\mathsf{unq}}\,\mathsf{int}) \to \&^\alpha_{\mathsf{unq}}\,\mathsf{int}\ \mathsf{I}\ \lambda\, post, ((n, n_\circ), (m, m_\circ)).\ \big(n \ge m \implies m_\circ = m \implies post\,((n, n_\circ))\big) \wedge \big(n < m \implies n_\circ = n \implies post\,((m, m_\circ))\big)$$

It follows from Acc-UnqRef-PlnRef, Rfn-Deref-PlnRef-Copy, Rfn-IntRel, Rfn-If, and SWkn-UnqRef-Release.

The test function `test` can be formalized as the following function inc_max.

$$\mathsf{inc\_max} \;:=\; \mathsf{fn}(a, b)\,\{\,\mathsf{let}\ a' = \mathsf{alloc}\,a\ \mathsf{in}\ \mathsf{let}\ b' = \mathsf{alloc}\,b\ \mathsf{in}\ a' \leftarrow a;\ b' \leftarrow b;\ \mathsf{let}\ c' = \mathsf{take\_max}(a', b')\ \mathsf{in}\ c' \leftarrow *c' + 1;\ \lightning;\ \mathsf{let}\ r = *a' \ne *b'\ \mathsf{in}\ \mathsf{free}\,a';\ \mathsf{free}\,b';\ r\,\}$$

It allocates two integer values a, b, uniquely borrows the box pointers, increments the target of the unique reference with the larger value taken by take_max, and checks that the two integer values have become different. We can verify that this function always return tt, i.e., the following specification holds.

$$\mathsf{inc\_max} : \mathsf{fn}(\mathsf{int}, \mathsf{int}) \to \mathsf{bool}\ \mathsf{I}\ \lambda\, post, (m, n).\ post\,(\mathsf{tt})$$

The verification goes as follows. By Rfn-Alloc, Rfn-Intro-LocalLft, Wkn-UnqBor-BoxPtr, Rfn-Fn-Call, SWkn-UnqRef-Release, SWkn-End-LocalLft-Reclaim, and Rfn-Free, we first obtain the following predicate transformer.

$$\lambda\, post, (n, m).\ \forall\, n_\circ, m_\circ.\ \big(n \geq m \Rightarrow m_\circ = m \Rightarrow n_\circ = n + 1 \Rightarrow post\,(n_\circ \neq m_\circ)\big)\ \wedge \\ \big(n < m \Rightarrow n_\circ = n \Rightarrow m_\circ = m + 1 \Rightarrow post\,(n_\circ \neq m_\circ)\big)$$

Now we can easily check that this is equivalent to $\lambda\, post, (m, n).\ post\,(\mathrm{tt})$.

Example 6.6 (Update of a List via a Unique Reference). The verification problem Example 1.10 verified by RustHorn can also be verified in our refined type system.

First, the function `sum` can be formalized as the following function $\mathsf{sum}_{\mathsf{list}}$.

$$\begin{array}{rcl} \mathsf{sum}_{\mathsf{list}} & := & \mathsf{fn}\, self\,(a)\, \{\, \mathsf{case}\, *a\, \mathsf{of}\, \{ \\ & & \quad 0 \to 0, \quad 1 \to \mathsf{let}\, a' = a.1\, \mathsf{in}\, *a' + self\,(*(a'.1)) \\ & & \} \} \end{array}$$

It satisfies the following specification using the function $\mathsf{sum} : \mathsf{List}\,\mathbb{Z} \to \mathbb{Z}$.

$$\mathsf{sum}_{\mathsf{list}} : \forall\, \alpha.\ \mathsf{fn}(\&^{\alpha}_{\mathsf{shr}}\, \mathsf{list\,int}) \to \mathsf{int}\ \mid\ \lambda\, post, (v).\ post\,(\mathsf{sum}\, v)$$

We elaborate its proof here.

Proof. We use Rfn-Fn-Rec. We can assume that *self* is modeled as the predicate transformer $\lambda\, post, (v).\ post\,(\mathsf{sum}\, v)$. The variable a has the type $\&^{\alpha}_{\mathsf{shr}}\, \mathsf{list\,int}$. We know that the lifetime α is alive during the function call. Let v: $\mathsf{List}\,\mathbb{Z}$ the pure value of a. By Subty-Ty-InjTy, we transform a into the type $\&^{\alpha}_{\mathsf{shr}}(\lightning_0 + \mathsf{int} \times \mathsf{box\,list\,int})$ and the value $\mathsf{out}_{\mathsf{List}}\, v : \mathsf{Unit} + \mathbb{Z} \times \mathsf{List}\,\mathbb{Z}$. By Acc-ShrRef-PlnRef and Rfn-Case-Vrnt-Access, we temporarily access the tag of a.

Let us discuss the case 0. By the effect of Rfn-Case-Vrnt-Access, we know $v = \mathsf{nil}$. Since $\mathsf{sum\,nil} = 0$ holds and we return 0 here, the postcondition is satisfied.

Let us discuss the case 1. We know that v is of the form $n :: w$. By Rfn-Subdiv-ShrRef-Vrnt, the variable a' defined as $a.1$ has the type $\&^{\alpha}_{\mathsf{shr}}(\mathsf{int} \times \mathsf{box\,list\,int})$ and the value (n, w). By Subty-Subdiv-ShrRef-Pair-L, a' can be used as a shared reference of the type $\&^{\alpha}_{\mathsf{shr}}\, \mathsf{int}$ and the value n. By Acc-ShrRef-PlnRef and Rfn-Deref-PlnRef-Copy, we get an integer of the value n from $*a'$. By Rfn-Subdiv-ShrRef-Pair-R and Rfn-Deref-ShrRef-BoxPtr, $*(a'.1)$ is a shared reference the type $\&^{\alpha}_{\mathsf{shr}}\, \mathsf{list\,int}$ and the value w. By passing it to *self*, we get an integer of the value $\mathsf{sum}\, w$. By Hoare-IntOp, we finally get the value $n + \mathsf{sum}\, w$, which is equal to $\mathsf{sum}(n :: w)$, i.e., $\mathsf{sum}\, v$. So the postcondition is satisfied. □

The function `take_some` can be formalized as the following function take_some.

$$\begin{array}{rcl} \mathsf{take\_some} & := & \mathsf{fn}\, self\,(a)\, \{\, \mathsf{case}\, *a\, \mathsf{of}\, \{ \\ & & \quad 0 \to self\,(a), \\ & & \quad 1 \to \mathsf{let}\, a' = a.1\, \mathsf{in}\, \mathsf{let}\, a'' = *(a'.1)\, \mathsf{in}\, \mathsf{if}\, \mathsf{ndint} \geq 0\, \{\lightning; a'\}\, \mathsf{else}\, \{\lightning; self\,(a'')\} \\ & & \} \} \end{array}$$

We can prove that it satisfies the following specification, which is enough for verification of the test function `test`.

$$\mathsf{take\_some}\colon \forall\, \alpha.\ \mathsf{fn}(\&^{\alpha}_{\mathsf{unq}}\, \mathsf{list\,int}) \to \&^{\alpha}_{\mathsf{unq}}\, \mathsf{int}\ \mid \\ \lambda\, post, ((v, v_\circ)).\ \forall\, n, n_\circ.\ \big(n_\circ = n + 1 \Rightarrow \mathsf{sum}\, v_\circ = \mathsf{sum}\, v + 1\big) \Rightarrow post\,((n, n_\circ))$$

The predicate transformer means that we get the postcondition $n_\circ = n + 1 \Rightarrow \mathsf{sum}\, v_\circ = \mathsf{sum}\, v + 1$ on the result $(n, n_\circ)$.

Proof. We use Rfn-Fn-Rec. We can assume that *self* satisfies the expected specification. The variable a has the type $\&^{\alpha}_{\mathsf{unq}}\,\mathsf{list\,int}$. Let $(v, v_\circ)$ be the pure value of a. We update the type of a into $\&^{\alpha}_{\mathsf{unq}}(\lightning_0 + \mathsf{int} \times \mathsf{box\,list\,int})$ by Subty-Ty-InjTy and temporarily access the tag by Acc-UnqRef-PlnRef and Rfn-Case-Vrnt-Access. In the case 0, we revert the type of a, and the postcondition is trivially satisfied by the recursive call on *self*.

Let us discuss the case 1. We know that v has the form $n :: v'$. By Rfn-Subdiv-UnqRef-Vrnt, the variable $a' = a.1$ has the type $\&^{\alpha}_{\mathsf{unq}}(\mathsf{int} \times \mathsf{box\,list\,int})$ and the value $((n, w), (n_\circ, w_\circ))$ for some $n_\circ, w_\circ$. By Rfn-Split-UnqRef-Pair-R, we split a' into two unique references, (i) $a'.1$, which has the type $\&^{\alpha}_{\mathsf{unq}}\,\mathsf{box\,list\,int}$ and the value $(w, w_\circ)$ and (ii) a', which has the type $\&^{\alpha}_{\mathsf{unq}}\,\mathsf{int}$ and the value $(n, n_\circ)$. Then by Rfn-Deref-UnqRef-BoxPtr, we subdivide the former into the variable $a'' = *(a'.1)$, which has the type $\&^{\alpha}_{\mathsf{unq}}\,\mathsf{list\,int}$ and the value $(w, w_\circ)$. Now by Rfn-Val-Bool, it suffices to verify the then case and the else case.

In the then case, we release a'' by SWkn-UnqRef-Release and get the postcondition $w_\circ = w$. We return a', which has the value $(n, n_\circ)$. When $n_\circ = n + 1$ holds, we have $\mathsf{sum}\, v_\circ = \mathsf{sum}(n_\circ :: w_\circ) = n_\circ + \mathsf{sum}\, w_\circ = n + 1 + \mathsf{sum}\, w = \mathsf{sum}(n :: w) + 1 = \mathsf{sum}\, v + 1$. Therefore, the postcondition is satisfied.

In the else case, we release a' by SWkn-UnqRef-Release and get the postcondition $n_\circ = n$. We recursively call *self* with a''. Let (m, m_0) be the finally returned value. Assume that $m_\circ = m+1$ holds. By the assumption on *self*, we know $\mathsf{sum}\, w_\circ = \mathsf{sum}\, w+1$. Therefore, we have $\mathsf{sum}\, v_\circ = \mathsf{sum}(n_\circ :: w_\circ) = n_\circ + \mathsf{sum}\, w_\circ = n + \mathsf{sum}\, w + 1 = \mathsf{sum}(n :: w) + 1 = \mathsf{sum}\, v + 1$. Therefore, the postcondition is satisfied. □

The test function `test` can be formalized as the following function test_take_some (we omit the structural deallocation of the list).

$$\begin{aligned}\mathsf{test\_take\_some} \;:=\; & \mathsf{fn}(a)\,\{\; \mathsf{let}\; n = \mathsf{sum}_{\mathsf{list}}(a)\; \mathsf{in} \\ & \quad \mathsf{let}\; b = \mathsf{take\_some}(a)\; \mathsf{in}\; \; b \leftarrow *b + 1;\; \lightning;\; \mathsf{sum}_{\mathsf{list}}(a) = n + 1\;\}\end{aligned}$$

It satisfies the following specification.

$$\mathsf{test\_take\_some} : \mathsf{box\,list\,int} \to \mathsf{bool} \;\vdash\; \lambda \mathit{post}, (v).\; \mathit{post}\,(\mathsf{tt})$$

It can be proved using the specifications on $\mathsf{sum}_{\mathsf{list}}$ and take_some, as well as Wkn-UnqBor-BoxPtr and SWkn-UnqRef-Release.

Example 6.7 (Splitting a Unique Reference to a List). The following function splits the unique reference to a list a into a box pointer to a list of unique references to each element of the list.

$$\begin{aligned}\mathsf{split}^{\tau}_{\mathsf{list}} \;:=\; & \mathsf{fn}\; \mathit{self}\,(a)\,\{\; \mathsf{let}\; b = \mathsf{alloc}\,3\; \mathsf{in}\; \; \mathsf{case}\; *a\; \mathsf{of}\; \{ \\ & \quad 0 \to \; b \leftarrow 0;\; b, \\ & \quad 1 \to \; \mathsf{let}\; a' = a.1\; \mathsf{in}\; \big(b \leftarrow 1;\; b.2 \leftarrow \mathit{self}\,(*(a'.|\tau|));\; b.1 \leftarrow a'\big);\; b \\ & \}\,\}\end{aligned}$$

We can derive the following specification on this function.

$$\begin{aligned}\mathsf{split}^{\tau}_{\mathsf{list}} : \forall\, \alpha.\; & \mathsf{fn}(\&^{\alpha}_{\mathsf{unq}}\,\mathsf{list}\,\tau) \to \mathsf{box\,list}\,\&^{\alpha}_{\mathsf{unq}}\,\tau \;\vdash \\ & \lambda \mathit{post}, ((v, v_\circ)).\; \mathsf{len}\, v_\circ = \mathsf{len}\, v \;\Rightarrow\; \mathit{post}\,(\mathsf{zip}\, v\, v_\circ)\end{aligned}$$

Proof. We use Rfn-Fn-Rec. We can assume that *self* satisfies the expected specification.

After the allocation alloc 3, b is typed box $\lightning_3$ by Rfn-Alloc. We convert the type of the unique reference a into $\&^{\alpha}_{\mathsf{unq}}(\lightning_0 + \tau \times \mathsf{box\,list}\,\tau)$ by Wkn-Subdiv-UnqRef-InjTy. We perform the case operation using Rfn-Case-Vrnt-Access.

Let us discuss the case where the tag is 0. We convert the type of the unique reference a back to $\&^{\alpha}_{\mathsf{unq}}\,\mathsf{list}\,\tau$ by Wkn-Subdiv-UnqRef-InjTy. We know that the current target value of a is nil by the effect of Rfn-Case-Vrnt-Access. By Acc-BoxPtr-PlnRef and Subty-Split-Invalid, we can temporarily take out a full plain reference $\&_1((\lightning_1 \times \lightning_0) \times \lightning_2)$ out of b. By Rfn-Store-PlnRef-PlnRef-Copy and Wkn-PlnRef-Triple-Vrnt, we can store $\mathsf{inj}_0()$ to the box pointer b, which is turned into nil by Subty-Ty-InjTy. Then we release the unique reference a by SWkn-UnqRef-Release, which resolves the prophecy value of a into nil. Since $\mathsf{len}\,\mathsf{nil} = \mathsf{nil}$ and $\mathsf{zip}\,\mathsf{nil}\,\mathsf{nil} = \mathsf{nil}$ hold, the specification is satisfied in this case.

Let us discuss the case where the tag is 1. We convert the type of the unique reference a back to $\&^{\alpha}_{\mathsf{unq}}\,\mathsf{list}\,\tau$. We know that the pure value of a is of the form $(v :: w, w'_\circ)$. By Raw-Subdiv-UnqRef-Vrnt, the unique reference $a.1$ named a' has the type $\&^{\alpha}_{\mathsf{unq}}(\tau \times \mathsf{box}\,\mathsf{list}\,\tau)$ and the pure value $((v, w), (v_\circ, w_\circ))$, where $w'_\circ$ has been resolved into $v_\circ :: w_\circ$. Inside the parenthesis, we access b by Acc-BoxPtr-PlnRef and Subty-Split-Invalid. By Rfn-Split-UnqRef-Pair-R and Rfn-Deref-UnqRef-BoxPtr, we can take out the unique reference $*(a'.|\tau|)$, which has the pure value $(w, w_\circ)$. By passing it to *self*, which satisfies the expected specification by the assumption, we get the postcondition $\mathsf{len}\,w_\circ = \mathsf{len}\,w$ and acquire as the returned object a box pointer of the type box list $\&^{\alpha}_{\mathsf{unq}}\,\tau$ and the pure value $\mathsf{zip}\,w\,w_\circ$, which is stored to $b.2$. Now we have $\mathsf{len}(v_\circ :: w_\circ) = 1 + \mathsf{len}\,w_\circ = 1 + \mathsf{len}\,w = \mathsf{len}(v :: w)$. We still have a unique reference a' of the type $\&^{\alpha}_{\mathsf{unq}}\,\tau$ and the pure value $(v, v_\circ)$, which is stored to $b.1$. By Wkn-PlnRef-Triple-Vrnt, b finally has the type box list $\&^{\alpha}_{\mathsf{unq}}\,\tau$ and the pure value $(v, v_\circ) :: \mathsf{zip}\,w\,w_\circ$, which is equal to $\mathsf{zip}\,(v :: w)\,(v_\circ :: w_\circ)$. So in this case the specification is satisfied, which concludes the proof. □

Example 6.8 (Iterative Update of Vector by Increment). The following function incr increments the target integer of the unique reference a and then release a.

$$\mathsf{incr} \;:=\; \mathsf{fn}(a)\,\{\, a \leftarrow *a + 1;\ \lightning \,\}$$

It satisfies the following specification, which follows from the specification of Example 6.3.

$$\mathsf{incr} : \forall\,\alpha.\ \mathsf{fn}(\&^{\alpha}_{\mathsf{unq}}\,\mathsf{int}) \;\mathrel{\mathsf{I}}\; \lambda\, post, ((n, n_\circ)).\ n_\circ = n + 1 \;\Rightarrow\; post\,()$$

Combining this and Rfn-Iter-Vec-UnqRef, we can derive the following specification.

$$\alpha \;\mathrel{\mathsf{I}}\; \mathfrak{a}\colon \&^{\alpha}_{\mathsf{unq}}\,\mathsf{int} \;\vdash\; \mathsf{iter}^{\mathsf{int}}_{\mathsf{vec}}(\mathfrak{a}, \mathsf{incr}) \;\mathrel{\mathsf{I}}\!\mathrel{\mathsf{I}}\; \lambda\, post, ((v, v_\circ)).\ v_\circ = \mathsf{map}\,(\lambda n.\, n{+}1)\,v \;\Rightarrow\; post\,()$$

6.5 Related Work

RustHorn As explained in §1.3, the idea of using *prophecy* for modeling unique references comes from RustHorn (Matsushita et al., 2020a,b). They also presented the proof of *soundness and completeness* of the prophecy-based reduction from Rust programs to CHCs.

As a basis for the soundness and completeness proof, they made a core calculus of Rust dubbed Calculus of Ownership and Reference (COR), which models some basic features of Rust. In COR, a function consists of a set of very simple statements, each of which is associated with a program point and finally either jumps to another program point or returns from the function. Because of this design of COR, their translation of COR programs into CHCs is rather simple; a statement of each program point in a program is translated into one CHC (or two CHCs, for conditional branching).

As explained in § 3.1, their soundness and completeness proof is based on construction of bisimulation between the execution on the COR side and some deduction algorithm (called SLDC resolution) on the CHC side. On this construction, they associate each object in COR with a multiset of ownership tokens on memory cells of the heap memory and manage the safety invariant on the multiset of ownership tokens over all the objects at each program point, which has a flavor of separation logic but is performed rather in an ad-hoc way. A prophecy variable of each unique reference is represented as syntactic logic variables in CHCs, which can later be specialized in the deduction algorithm. Their proof depends heavily on syntactic structures, which requires careful design of the calculus COR, the reduction from COR to CHCs, and the deduction on CHCs. Because of the syntactic nature, it is unclear how we can extend their proof to other features and libraries of Rust, although they conjectured that their prophecy-based approach is applicable to a wide class of features and libraries.

Unlike RustHorn, the functional-correctness proof of our thesis is highly extensible, thanks to the power of the higher-order separation logic Iris and the lifetime logic. We support function types, concurrency and vector types, which are not supported by the formalization of RustHorn. Also, we can flexibly add new typing rules and libraries, whose correctness can be checked separately.

Still, for the logic model we can make using the refined type system we give the proof of *soundness* (with the adequacy theorems Theorem 6.1 and Theorem 4.1) but not the proof of *completeness*, whereas RustHorn proves both soundness and completeness of the reduction. We believe that we can in theory prove completeness of our logic model by appropriately restricting the range of programs, in a way that enforces resolution of the prophecy variable of every unique reference. Still, in order to attain an extensible proof of completeness of the logic model, we probably need to design a new program logic.

RustBelt Our *semantic* approach to program verification comes from RustBelt (Jung et al., 2018a). RustBelt verified *safety* of the core type system and some basic libraries of Rust, but it did not verify functional correctness. In particular, RustBelt models unique reference as a kind of invariant saying that *some* object of the type is stored at the target address and does not track precise information about the object.

We extended their approach for verification of *functional correctness* of basic operations and libraries in Rust using the *prophecy*-based reduction of RustHorn. We refined semantic types with pure values parametrized over π, the assignment on prophecy variables, in order to handle prophecy information.

Still, the verification platform of this thesis does not support libraries with *interior mutability*. RustBelt verified *safety* of various libraries with interior mutability, including `Cell`, `RefCell`, `Mutex` and `RwLock`, using the power of invariants provided by Iris. We can rather easily incorporate their safety proofs into RustHornBelt. However, in order to verify more precise information about *functional correctness*, we need more machinery. This point is discussed more in depth in Chapter 7.

Semantic Soundness Proof There are some existing studies on *semantic* soundness proof (i.e., soundness proof based on logical relations) for a substructural type system, including those by Morrisett et al. (2005); Ahmed et al. (2005); Krishnaswami et al. (2012); Jung et al. (2018a); Dang et al. (2020), although semantic approaches remain rather rare compared to *syntactic* approaches. However, we are not aware of any existing study that presents semantic soundness proof for a substructural type system that takes into account *functional correctness*, not only memory and thread safety.

Other Studies on Verification of Rust Programs There are a number of studies on automated low-level verification of Rust programs. Toman et al. (2015) developed a bounded model checker CRUST for automatically detecting violation of Rust's ownership invariants by a Rust library with *unsafe code* within some bounds. The tool exhaustively generates tests on client use of a Rust library, imposing some upper bound on the number of library method calls, and passes the tests to SMBC, an existing SMT-based bounded model checker for the C programming language. They successfully rediscovered some real-world bugs of the standard Rust library that had been fixed at that time but been missed for a long time. Lindner et al. (2018) performed automated verification on Rust using symbolic execution in KLEE on the LLVM bitcode extracted from a Rust program. They particularly focused on the property that the program does not cause abortion, for example by out-of-bounds access or division by zero. Baranowski et al. (2018) performed automated verification on Rust programs using the SMACK verifier. In order to reuse SMACK's existing verification tools for C, they modeled common Rust libraries as special C code tailored to SMACK.

There are also a number of existing studies on verification of Rust programs that exploit the guarantees by Rust's ownership principle. Ullrich (2016) developed a tool for automatically translating Rust programs of some subset into the purely functional language of the Lean Theorem Prover and verified functional correctness of some Rust programs in Lean using that translation. The basic idea of his translation is to model a Rust function that inputs and consumes a unique reference (e.g., `Vec::push`) as a function that inputs the target value of the unique reference and outputs the new target value of the unique reference. The translation also supports one-to-one subdivision of unique references like `Vec::index_mut` through functional lenses. However, the translation still cannot handle various common usages of unique references, including *split* of a unique reference (e.g., `&mut (T, U)` into `&mut T` and `&mut U`), unlike the prophecy-based approach of RustHorn and RustHornBelt. Hahn (2016) and Astrauskas et al. (2019) performed semi-automated verification of Rust programs with annotations for verification on top of Viper (Müller et al., 2016), a verification platform that employs a separation logic with fractional permission. In particular, Astrauskas et al. (2019) supported one-to-one subdivision of unique references like `Vec::index_mut` using an assertion called a pledge, which is modeled in Viper's separation logic using a magic wand. Also, Viper's separation logic is much simpler than Iris and can possibly be more suited for automation. Still, their encoding does not support general use cases of unique references, including split of a unique reference, unlike RustHorn and RustHornBelt.

Chapter 7

Conclusion and Future Work

We proposed a novel extensible logical foundation to specify and verify functional correctness of Rust programs, using a prophecy-based clean logic model in the style of RustHorn and taking a semantic approach in Iris in the style of RustBelt, under the project name RustHornBelt. We designed a new type system with a logic model for Rust, which is highly flexible and supports various features that were not supported by RustHorn. As a basis for that, we also presented a new flexible platform of prophecy in Iris, which manages information about prophecy only in the ghost state and allows an operation that we call dependent resolution. We also presented a new technique for spending many logical steps at a time in Iris.

We believe that this work serves as a significant step for extensible and scalable verification of Rust programs, but we still have a lot of work to do in this direction.

Mechanization in Coq We plan to *mechanize* the results of this thesis in the Coq Proof Assistant in the near future. We already have a large size of Coq code that we can utilize. The separation logic Iris is mechanized in Coq with great support of tactics and proof modes. The lifetime logic is mechanized on top of the Coq library of Iris. Also, we can reuse some parts of the implementation of RustBelt. Still, since we handle information about *functional correctness* involving *prophecy* in RustHornBelt, we need to develop useful lemmas and tactics for verifying typing rules and libraries efficiently. In particular, it is great if we can verify involved libraries (e.g., `Vec` and `HashMap`) in a *scalable* way. We can possibly adopt some methods from a recently emerging study by Sammler et al. (2020b), which works on automated and verification of functional correctness of C programs with an extensible and reliable foundation built in Coq and Iris.

Semi-Automated Verification of Rust Programs Although this thesis presents a logical foundation to give clean logical models to basic operations and libraries of Rust by the technique of prophecy, we have yet to design and implement a *semi-automated* verification platform for Rust programs that can perform scalable verification of Rust programs using the clean models given by our foundation. We expect that we can achieve such a scalable verification platform for Rust on top of existing semi-automated verification platforms for functional programming languages such as F* (Swamy et al., 2016) and Why3 (Filliâtre and Paskevich, 2013).

Interior Mutability In the verification platform of this thesis, we omitted support of libraries with *interior mutability* like `Cell` and `RefCell`. We can prove memory and thread *safety* of such libraries using atomic and non-atomic *invariants* asserting we have *some* object of the expect type inside the cell-like object, which is the proof

technique used by RustBelt.[1] However, if we further want to prove *functional correctness*, we need to track more precise information about the *value* of the inner object of the cell-like object.

With interior mutability, we need to handle almost general imperative programming, although we have solid memory and thread safety. The situation is similar for example to verification of OCaml programs with reference cells, where memory safety is ensured by the type system and the garbage collection.

In principle, we can precisely encode a Rust program using interior mutability by carrying around a *global* array which maps the id of each cell-like object to its inner value in the store-passing style, but this is not very *scalable* for involved programs. One idea is to use *region types* (Fluet et al., 2006) to separate out some groups of cell-like objects that we manipulate, but we have yet to tailor this technique to Rust.

Concurrency In the verification platform of this thesis, we supported only the structured concurrent execution $e \parallel e'$ for concurrency.

To support more features of concurrency, we can use the proof techniques of RustBelt, which verified *safety* of various features for concurrency in Rust including threads, channels, mutexes `Mutex<T>` and read-write locks `RwLock<T>`. Still, in order to verify *functional correctness* of concurrent programs, we need other techniques.

One idea is to use *prophecy* also for concurrency. For example, when we create a channel to get a sender and a receiver, we can prophesy the list of values of the objects that will pass through the channel.[2] When the sender sends an object, it can resolve the head of the prophecy list and subdivide the prophecy list into its tail. When the receiver receives an object, we know the value of the object from the prophesied list. When we clone the sender, we split the prophecy list in a non-deterministic way. We believe that we can formalize this prophecy-based encoding of a channel using our formulation of prophecy (Chapter 3). Still, this encoding is not very precise. For example, in this encoding, we cannot detect dead locks or discuss interaction between multiple senders.

Another idea is to incorporate the *rely-guarantee* reasoning (Vafeiadis and Parkinson, 2007; Feng et al., 2007) for more fine-grained verification on concurrency, but we have yet to tailor this to RustHornBelt.

Termination-Sensitive Verification The verification platform of this thesis does not support *termination-sensitive* verification. Using time credits (Atkey, 2011; Mével et al., 2019), we can verify a specific upper bound on the execution time of a program, which can probably be incorporated into RustHornBelt. However, when a function takes an unbounded but finite number of physical steps (e.g., repeat a loop n times for a random natural number n), it is hard to verify its termination in Iris, particularly because Iris uses *finite* step indexing. We may be able to overcome this situation by using recently emerging Transfinite Iris (Spies et al., 2020) instead of Iris for the underlying separation logic. However, as discussed in §4.3, whether we can use Transfinite Iris for RustHornBelt remains unclear because a number of good properties of Iris are lost in Transfinite Iris.

[1] To support types like `Cell` in the presence of concurrency, we need to introduce Rust's `Send` and `Sync` traits, which manage what kind of object can be passed between thread. RustBelt models these traits respectively as the condition that the ownership or sharing predicate of the type does not depend on the thread id.

[2] This idea is similar to *sequence prophecies* used by Jung et al. (2020b).

References

Martín Abadi and Leslie Lamport. 1988. The Existence of Refinement Mappings. In *Proceedings of the Third Annual Symposium on Logic in Computer Science (LICS '88), Edinburgh, Scotland, UK, July 5-8, 1988*. IEEE Computer Society, 165–175. https://doi.org/10.1109/LICS.1988.5115

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

Amal J. Ahmed, Matthew Fluet, and Greg Morrisett. 2005. A Step-Indexed Model of Substructural State. In *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming, ICFP 2005, Tallinn, Estonia, September 26-28, 2005*, Olivier Danvy and Benjamin C. Pierce (Eds.). ACM, 78–91. https://doi.org/10.1145/1086365.1086376

Hussain M. J. Almohri and David Evans. 2018. Fidelius Charm: Isolating Unsafe Rust Code. In *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy, CODASPY 2018, Tempe, AZ, USA, March 19-21, 2018*, Ziming Zhao, Gail-Joon Ahn, Ram Krishnan, and Gabriel Ghinita (Eds.). ACM, 248–255. https://doi.org/10.1145/3176258.3176330

Brian Anderson, Lars Bergstrom, Manish Goregaokar, Josh Matthews, Keegan McAllister, Jack Moffitt, and Simon Sapin. 2016. Engineering the Servo Web Browser Engine Using Rust. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016 - Companion Volume*, Laura K. Dillon, Willem Visser, and Laurie Williams (Eds.). ACM, 81–89. https://doi.org/10.1145/2889160.2889229

Andrew W. Appel and David A. McAllester. 2001. An Indexed Model of Recursive Types for Foundational Proof-Carrying Code. *ACM Trans. Program. Lang. Syst.* 23, 5 (2001), 657–683. https://doi.org/10.1145/504709.504712

Vytautas Astrauskas, Christoph Matheja, Federico Poli, Peter Müller, and Alexander J. Summers. 2020. How Do Programmers Use Unsafe Rust? *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 136:1–136:27. https://doi.org/10.1145/3428204

Vytautas Astrauskas, Peter Müller, Federico Poli, and Alexander J. Summers. 2019. Leveraging Rust Types for Modular Specification and Verification. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 147:1–147:30. https://doi.org/10.1145/3360573

Robert Atkey. 2011. Amortised Resource Analysis with Separation Logic. *Log. Methods Comput. Sci.* 7, 2 (2011). https://doi.org/10.2168/LMCS-7(2:17)2011

Abhiram Balasubramanian, Marek S. Baranowski, Anton Burtsev, Aurojit Panda, Zvonimir Rakamaric, and Leonid Ryzhyk. 2017. System Programming in Rust: Beyond

Safety. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems, HotOS 2017, Whistler, BC, Canada, May 8-10, 2017*, Alexandra Fedorova, Andrew Warfield, Ivan Beschastnikh, and Rachit Agarwal (Eds.). ACM, 156–161. https://doi.org/10.1145/3102980.3103006

Marek S. Baranowski, Shaobo He, and Zvonimir Rakamaric. 2018. Verifying Rust Programs with SMACK. In *Automated Technology for Verification and Analysis - 16th International Symposium, ATVA 2018, Los Angeles, CA, USA, October 7-10, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 11138)*, Shuvendu K. Lahiri and Chao Wang (Eds.). Springer, 528–535. https://doi.org/10.1007/978-3-030-01090-4_32

Ariel Ben-Yehuda. 2015. *`std::thread::JoinGuard` (and `scoped`) are Unsound Because of Reference Cycles.* https://github.com/rust-lang/rust/issues/24292

Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R. Newton, Simon Peyton Jones, and Arnaud Spiwack. 2018. Linear Haskell: Practical Linearity in a Higher-Order Polymorphic Language. *PACMPL* 2, POPL (2018), 5:1–5:29. https://doi.org/10.1145/3158093

Lars Birkedal, Rasmus Ejlers Møgelberg, Jan Schwinghammer, and Kristian Støvring. 2011. First Steps in Synthetic Guarded Domain Theory: Step-Indexing in the Topos of Trees. In *Proceedings of the 26th Annual IEEE Symposium on Logic in Computer Science, LICS 2011, June 21-24, 2011, Toronto, Ontario, Canada*. IEEE Computer Society, 55–64. https://doi.org/10.1109/LICS.2011.16

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

Konrad Borowski. 2019. *`PartialEq` Implementation for `RangeInclusive` is Unsound.* https://github.com/rust-lang/rust/issues/67194

Adrien Champion, Naoki Kobayashi, and Ryosuke Sato. 2018. HoIce: An ICE-Based Non-linear Horn Clause Solver. In *Programming Languages and Systems - 16th Asian Symposium, APLAS 2018, Wellington, New Zealand, December 2-6, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 11275)*, Sukyoung Ryu (Ed.). Springer, 146–156. https://doi.org/10.1007/978-3-030-02768-1_8

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

Byron Cook and Eric Koskinen. 2011. Making Prophecies with Decision Predicates. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, Thomas Ball and Mooly Sagiv (Eds.). ACM, 399–410. https://doi.org/10.1145/1926385.1926431

Coq Community. 2021. *The Coq Proof Assistant.* https://coq.inria.fr/

Hoang-Hai Dang, Jacques-Henri Jourdan, Jan-Oliver Kaiser, and Derek Dreyer. 2020. RustBelt Meets Relaxed Memory. *Proc. ACM Program. Lang.* 4, POPL (2020), 34:1–34:29. https://doi.org/10.1145/3371102

Paulo Emílio de Vilhena and François Pottier. 2021. A Separation Logic for Effect Handlers. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–28. https://doi.org/10.1145/3434314

Paulo Emílio de Vilhena, François Pottier, and Jacques-Henri Jourdan. 2020. Spy Game: Verifying a Local Generic Solver in Iris. *Proc. ACM Program. Lang.* 4, POPL (2020), 33:1–33:28. https://doi.org/10.1145/3371101

Robert DeLine and Manuel Fähndrich. 2001. Enforcing High-Level Protocols in Low-Level Software. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Snowbird, Utah, USA, June 20-22, 2001*, Michael Burke and Mary Lou Soffa (Eds.). ACM, 59–69. https://doi.org/10.1145/378795.378811

Yu Ding, Ran Duan, Long Li, Yueqiang Cheng, Yulong Zhang, Tanghui Chen, Tao Wei, and Huibo Wang. 2017. Rust SGX SDK: Towards Memory Safety in Intel SGX Enclave. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu (Eds.). ACM, 2491–2493. https://doi.org/10.1145/3133956.3138824

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

Paul Emmerich, Sebastian Voit, Georg Carle, Simon Ellmann, Fabian Bonk, Alex Egger, Esaú García Sánchez-Torija, Thomas Günzel, Sebastian Di Luzio, Alexandru Obada, and Maximilian Stadlmeier. 2019. The Case for Writing Network Drivers in High-Level Programming Languages. In *2019 ACM/IEEE Symposium on Architectures for Networking and Communications Systems, ANCS 2019, Cambridge, United Kingdom, September 24-25, 2019*. IEEE, 1–13. https://doi.org/10.1109/ANCS.2019.8901892

Manuel Fähndrich and Robert DeLine. 2002. Adoption and Focus: Practical Linear Types for Imperative Programming. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Berlin, Germany, June 17-19, 2002*, Jens Knoop and Laurie J. Hendren (Eds.). ACM, 13–24. https://doi.org/10.1145/512529.512532

Chris Fallin. 2020. Safe, Flexible Aliasing with Deferred Borrows. In *34th European Conference on Object-Oriented Programming, ECOOP 2020, November 15-17, 2020, Berlin, Germany (Virtual Conference) (LIPIcs, Vol. 166)*, Robert Hirschfeld and Tobias Pape (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 30:1–30:26. https://doi.org/10.4230/LIPIcs.ECOOP.2020.30

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

Xinyu Feng, Rodrigo Ferreira, and Zhong Shao. 2007. On the Relationship Between Concurrent Separation Logic and Assume-Guarantee Reasoning. In *Programming Languages and Systems, 16th European Symposium on Programming, ESOP 2007, Held as Part of the Joint European Conferences on Theory and Practics of Software, ETAPS 2007, Braga, Portugal, March 24 - April 1, 2007, Proceedings (Lecture Notes in Computer Science, Vol. 4421)*, Rocco De Nicola (Ed.). Springer, 173–188. https://doi.org/10.1007/978-3-540-71316-6_13

Jean-Christophe Filliâtre and Andrei Paskevich. 2013. Why3 - Where Programs Meet Provers. In *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 7792)*, Matthias Felleisen and Philippa Gardner (Eds.). Springer, 125–128. https://doi.org/10.1007/978-3-642-37036-6_8

Matthew Fluet, Greg Morrisett, and Amal J. Ahmed. 2006. Linear Regions Are All You Need. In *Programming Languages and Systems, 15th European Symposium on Programming, ESOP 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 27-28, 2006, Proceedings (Lecture Notes in Computer Science, Vol. 3924)*, Peter Sestoft (Ed.). Springer, 7–21. https://doi.org/10.1007/11693024_2

J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. 2007. Combinators for Bidirectional Tree Transformations: A Linguistic Approach to the View-Update Problem. *ACM Trans. Program. Lang. Syst.* 29, 3 (2007), 17. https://doi.org/10.1145/1232420.1232424

Dan Frumin, Robbert Krebbers, and Lars Birkedal. 2018. ReLoC: A Mechanised Relational Logic for Fine-Grained Concurrency. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, Anuj Dawar and Erich Grädel (Eds.). ACM, 442–451. https://doi.org/10.1145/3209108.3209174

Dan R. Ghica and Alex I. Smith. 2014. Bounded Linear Types in a Resource Semiring. In *Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings (Lecture Notes in Computer Science, Vol. 8410)*, Zhong Shao (Ed.). Springer, 331–350. https://doi.org/10.1007/978-3-642-54833-8_18

Léon Gondelman, Simon Oddershede Gregersen, Abel Nieto, Amin Timany, and Lars Birkedal. 2021. Distributed Causal Memory: Modular Specification and Verification in Higher-Order Distributed Separation logic. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–29. https://doi.org/10.1145/3434323

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

Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A. Navas. 2015. The SeaHorn Verification Framework. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*

(Lecture Notes in Computer Science, Vol. 9206), Daniel Kroening and Corina S. Pasareanu (Eds.). Springer, 343–361. https://doi.org/10.1007/978-3-319-21690-4_20

Florian Hahn. 2016. Rust2Viper: Building a Static Verifier for Rust. Master Thesis. ETH Zürich. https://www.research-collection.ethz.ch/handle/20.500.11850/155723

Philipp Haller and Martin Odersky. 2010. Capabilities for Uniqueness and Borrowing. In *ECOOP 2010 - Object-Oriented Programming, 24th European Conference, Maribor, Slovenia, June 21-25, 2010. Proceedings (Lecture Notes in Computer Science, Vol. 6183)*, Theo D'Hondt (Ed.). Springer, 354–378. https://doi.org/10.1007/978-3-642-14107-2_17

Timothy L. Harris, Keir Fraser, and Ian A. Pratt. 2002. A Practical Multi-word Compare-and-Swap Operation. In *Distributed Computing, 16th International Conference, DISC 2002, Toulouse, France, October 28-30, 2002 Proceedings (Lecture Notes in Computer Science, Vol. 2508)*, Dahlia Malkhi (Ed.). Springer, 265–279. https://doi.org/10.1007/3-540-36108-1_18

Jonas Kastberg Hinrichsen, Jesper Bengtson, and Robbert Krebbers. 2020. Actris: Session-Type Based Reasoning in Separation Logic. *Proc. ACM Program. Lang.* 4, POPL (2020), 6:1–6:30. https://doi.org/10.1145/3371074

Hossein Hojjat and Philipp Rümmer. 2018. The Eldarica Horn Solver. In *2018 Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA, October 30 - November 2, 2018*, Nikolaj Bjørner and Arie Gurfinkel (Eds.). IEEE, 1–7. https://doi.org/10.23919/FMCAD.2018.8603013

Koen Jacobs, Amin Timany, and Dominique Devriese. 2021. Fully Abstract from Static to Gradual. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–30. https://doi.org/10.1145/3434288

Thomas Bracht Laumann Jespersen, Philip Munksgaard, and Ken Friis Larsen. 2015. Session Types for Rust. In *Proceedings of the 11th ACM SIGPLAN Workshop on Generic Programming, WGP@ICFP 2015, Vancouver, BC, Canada, August 30, 2015*, Patrick Bahr and Sebastian Erdweg (Eds.). ACM, 13–22. https://doi.org/10.1145/2808098.2808100

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

Ralf Jung. 2017. *`MutexGuard<Cell<i32>>` Must not be `Sync`*. https://github.com/rust-lang/rust/issues/41622

Ralf Jung. 2020. *Understanding and Evolving the Rust Programming Language*. Ph. D. Dissertation. Saarland University, Saarbrücken, Germany. https://publikationen.sulb.uni-saarland.de/handle/20.500.11880/29647

Ralf Jung, Hoang-Hai Dang, Jeehoon Kang, and Derek Dreyer. 2020a. Stacked Borrows: an Aliasing Model for Rust. *Proc. ACM Program. Lang.* 4, POPL (2020), 41:1–41:32. https://doi.org/10.1145/3371109

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

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2018b. RustBelt: Securing the Foundations of the Rust Programming Language — Technical Appendix. https://plv.mpi-sws.org/rustbelt/popl18/appendix.pdf

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018c. Iris from the Ground Up: A Modular Foundation for Higher-Order Concurrent Separation Logic. *J. Funct. Program.* 28 (2018), e20. https://doi.org/10.1017/S0956796818000151

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

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

Jan-Oliver Kaiser, Hoang-Hai Dang, Derek Dreyer, Ori Lahav, and Viktor Vafeiadis. 2017. Strong Logic for Weak Memory: Reasoning About Release-Acquire Consistency in Iris. In *31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19-23, 2017, Barcelona, Spain (LIPIcs, Vol. 74)*, Peter Müller (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 17:1–17:29. https://doi.org/10.4230/LIPIcs.ECOOP.2017.17

Steve Klabnik, Carol Nichols, and Rust Community. 2018. *The Rust Programming Language*. https://doc.rust-lang.org/book/

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

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

Robbert Krebbers, Jacques-Henri Jourdan, Ralf Jung, Joseph Tassarotti, Jan-Oliver Kaiser, Amin Timany, Arthur Charguéraud, and Derek Dreyer. 2018. MoSeL: A General, Extensible Modal Framework for Interactive Proofs in Separation Logic. *Proc. ACM Program. Lang.* 2, ICFP (2018), 77:1–77:30. https://doi.org/10.1145/3236772

Robbert Krebbers, Amin Timany, and Lars Birkedal. 2017. Interactive Proofs in Higher-Order Concurrent Separation Logic. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, Jan-*

uary 18-20, 2017, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 205–217. https://doi.org/10.1145/3009837

Neelakantan R. Krishnaswami, Aaron Turon, Derek Dreyer, and Deepak Garg. 2012. Superficially Substructural Types. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'12, Copenhagen, Denmark, September 9-15, 2012*, Peter Thiemann and Robby Bruce Findler (Eds.). ACM, 41–54. https://doi.org/10.1145/2364527.2364536

Benjamin Lamowski, Carsten Weinhold, Adam Lackorzynski, and Hermann Härtig. 2017. Sandcrust: Automatic Sandboxing of Unsafe Components in Rust. In *Proceedings of the 9th Workshop on Programming Languages and Operating Systems, Shanghai, China, October 28, 2017*, Julia Lawall (Ed.). ACM, 51–57. https://doi.org/10.1145/3144555.3144562

Leslie Lamport and Stephan Merz. 2017. Auxiliary Variables in TLA+. *CoRR* abs/1703.05121 (2017). arXiv:1703.05121 http://arxiv.org/abs/1703.05121

Amit Levy, Bradford Campbell, Branden Ghena, Pat Pannuto, Prabal Dutta, and Philip Levis. 2017. The Case for Writing a Kernel in Rust. In *Proceedings of the 8th Asia-Pacific Workshop on Systems, Mumbai, India, September 2, 2017*. ACM, 1:1–1:7. https://doi.org/10.1145/3124680.3124717

Amit A. Levy, Michael P. Andersen, Bradford Campbell, David E. Culler, Prabal Dutta, Branden Ghena, Philip Levis, and Pat Pannuto. 2015. Ownership is theft: experiences building an embedded OS in Rust. In *Proceedings of the 8th Workshop on Programming Languages and Operating Systems, PLOS 2015, Monterey, California, USA, October 4, 2015*, Shan Lu (Ed.). ACM, 21–26. https://doi.org/10.1145/2818302.2818306

Hongjin Liang and Xinyu Feng. 2013. Modular Verification of Linearizability with Non-Fixed Linearization Points. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, Hans-Juergen Boehm and Cormac Flanagan (Eds.). ACM, 459–470. https://doi.org/10.1145/2491956.2462189

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

Nicholas D. Matsakis. 2018. *An Alias-based Formulation of the Borrow Checker*. http://smallcultfollowing.com/babysteps/blog/2018/04/27/an-alias-based-formulation-of-the-borrow-checker/

Nicholas D. Matsakis. 2020. *Polonius: Either Borrower or Lender Be, but Responsibly*. https://www.youtube.com/watch?v=_agDeiWek8w

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

Yusuke Matsushita. 2019. 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). Senior Thesis. University of Tokyo. http://www.kb.is.s.u-tokyo.ac.jp/~yskm24t/papers/senior-thesis.pdf

Yusuke Matsushita, Takeshi Tsukada, and Naoki Kobayashi. 2020a. RustHorn: CHC-based Verification for Rust Programs. In *Programming Languages and Systems - 29th European Symposium on Programming, ESOP 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings (Lecture Notes in Computer Science, Vol. 12075)*, Peter Müller (Ed.). Springer, 484–514. https://doi.org/10.1007/978-3-030-44914-8_18

Yusuke Matsushita, Takeshi Tsukada, and Naoki Kobayashi. 2020b. RustHorn: CHC-based Verification for Rust Programs (full version). *CoRR* abs/2002.09002 (2020). arXiv:2002.09002 https://arxiv.org/abs/2002.09002

Karl Mazurak, Jianzhou Zhao, and Steve Zdancewic. 2010. Lightweight Linear Types in System F°. In *Proceedings of TLDI 2010: 2010 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, Madrid, Spain, January 23, 2010*, Andrew Kennedy and Nick Benton (Eds.). ACM, 77–88. https://doi.org/10.1145/1708016.1708027

Glen Mével, Jacques-Henri Jourdan, and François Pottier. 2019. Time Credits and Time Receipts in Iris. In *Programming Languages and Systems - 28th European Symposium on Programming, ESOP 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11423)*, Luís Caires (Ed.). Springer, 3–29. https://doi.org/10.1007/978-3-030-17184-1_1

Matt Miller. 2019. *Trends, Challenge, and Shifts in Software Vulnerability Mitigation.* https://www.youtube.com/watch?v=PjbGojjnBZQ

Kai Mindermann, Philipp Keck, and Stefan Wagner. 2018. How Usable Are Rust Cryptography APIs?. In *2018 IEEE International Conference on Software Quality, Reliability and Security, QRS 2018, Lisbon, Portugal, July 16-20, 2018*. IEEE, 143–154. https://doi.org/10.1109/QRS.2018.00028

J. Garrett Morris. 2016. The Best of Both Worlds: Linear Functional Programming Without Compromise. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, Jacques Garrigue, Gabriele Keller, and Eijiro Sumii (Eds.). ACM, 448–461. https://doi.org/10.1145/2951913.2951925

Greg Morrisett, Amal J. Ahmed, and Matthew Fluet. 2005. L^3: A Linear Language with Locations. In *Typed Lambda Calculi and Applications, 7th International Conference, TLCA 2005, Nara, Japan, April 21-23, 2005, Proceedings (Lecture Notes in Computer Science, Vol. 3461)*, Pawel Urzyczyn (Ed.). Springer, 293–307. https://doi.org/10.1007/11417170_22

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

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

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

Peter W. O'Hearn. 2007. Resources, Concurrency, and Local Reasoning. *Theor. Comput. Sci.* 375, 1-3 (2007), 271–307. https://doi.org/10.1016/j.tcs.2006.12.035

Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. 2001. Local Reasoning about Programs that Alter Data Structures. In *Computer Science Logic, 15th International Workshop, CSL 2001. 10th Annual Conference of the EACSL, Paris, France, September 10-13, 2001, Proceedings (Lecture Notes in Computer Science, Vol. 2142)*, Laurent Fribourg (Ed.). Springer, 1–19. https://doi.org/10.1007/3-540-44802-0_1

Boqin Qin, Yilun Chen, Zeming Yu, Linhai Song, and Yiying Zhang. 2020. Understanding Memory and Thread Safety Practices and Issues in real-World Rust Programs. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, Alastair F. Donaldson and Emina Torlak (Eds.). ACM, 763–779. https://doi.org/10.1145/3385412.3386036

Azalea Raad, John Wickerson, Gil Neiger, and Viktor Vafeiadis. 2020. Persistency Semantics of the Intel-x86 Architecture. *Proc. ACM Program. Lang.* 4, POPL (2020), 11:1–11:31. https://doi.org/10.1145/3371079

Eric Reed. 2015. Patina: A Formalization of the Rust Programming Language. Master Thesis. University of Washington. https://dada.cs.washington.edu/research/tr/2015/03/UW-CSE-15-03-02.pdf

John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*. IEEE Computer Society, 55–74. https://doi.org/10.1109/LICS.2002.1029817

Rust Community. 2020. *2094-nll — The Rust RFC Book*. https://rust-lang.github.io/rfcs/2094-nll.html

Rust Community. 2021a. *Rust Programming Language*. https://www.rust-lang.org/

Rust Community. 2021b. *Sponsors — Rust Programming Language*. https://www.rust-lang.org/sponsors

Rust Community. 2021c. *rust-lang/polonius: Defines the Rust Borrow Checker*. https://github.com/rust-lang/polonius

Michael Sammler, Deepak Garg, Derek Dreyer, and Tadeusz Litak. 2020a. The High-Level Benefits of Low-Level Sandboxing. *Proc. ACM Program. Lang.* 4, POPL (2020), 32:1–32:32. https://doi.org/10.1145/3371100

Michael Sammler, Rodolphe Lepigre, Robbert Krebbers, Kayvan Memarian, Derek Dreyer, and Deepak Garg. 2020b. RefinedC: A Foundational Refinement Type System for C Based on Separation Logic Programming. (2020). https://plv.mpi-sws.org/refinedc/paper.pdf

Ali Sezgin, Serdar Tasiran, and Shaz Qadeer. 2010. Tressa: Claiming the Future. In *Verified Software: Theories, Tools, Experiments, Third International Conference, VSTTE 2010, Edinburgh, UK, August 16-19, 2010. Proceedings (Lecture Notes in Computer Science, Vol. 6217)*, Gary T. Leavens, Peter W. O'Hearn, and Sriram K. Rajamani (Eds.). Springer, 25–39. https://doi.org/10.1007/978-3-642-15057-9_2

Simon Spies, Lennard Gäher, Daniel Gratzer, Joseph Tassarotti, Robbert Krebbers, Derek Dreyer, and Lars Birkedal. 2020. Transfinite Iris: Resolving an Existential Dilemma of Step-Indexed Separation Logic. (2020). https://iris-project.org/pdfs/2020-transfinite-iris-submission.pdf

Nikhil Swamy, Catalin Hritcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean Karim Zinzindohoue, and Santiago Zanella Béguelin. 2016. Dependent Types and Multi-Monadic Effects in F*. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, Rastislav Bodík and Rupak Majumdar (Eds.). ACM, 256–270. https://doi.org/10.1145/2837614.2837655

Amin Timany, Léo Stefanesco, Morten Krogh-Jespersen, and Lars Birkedal. 2018. A Logical Relation for Monadic Encapsulation of State: Proving Contextual Equivalences in the Presence of runST. *Proc. ACM Program. Lang.* 2, POPL (2018), 64:1–64:28. https://doi.org/10.1145/3158152

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

Jesse A. Tov and Riccardo Pucella. 2011. Practical Affine Types. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, Thomas Ball and Mooly Sagiv (Eds.). ACM, 447–458. https://doi.org/10.1145/1926385.1926436

Aaron Joseph Turon, Jacob Thamsborg, Amal Ahmed, Lars Birkedal, and Derek Dreyer. 2013. Logical Relations for Fine-Grained Concurrency. In *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, Roberto Giacobazzi and Radhia Cousot (Eds.). ACM, 343–356. https://doi.org/10.1145/2429069.2429111

Sebastian Ullrich. 2016. Simple Verification of Rust Programs via Functional Purification. Master Thesis. Karlsruhe Institute of Technology. https://pp.ipd.kit.edu/uploads/publikationen/ullrich16masterarbeit.pdf

Viktor Vafeiadis. 2008. *Modular Fine-Grained Concurrency Verification.* Ph. D. Dissertation. University of Cambridge, UK. http://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.612221

Viktor Vafeiadis and Matthew J. Parkinson. 2007. A Marriage of Rely/Guarantee and Separation Logic. In *CONCUR 2007 - Concurrency Theory, 18th International Conference, CONCUR 2007, Lisbon, Portugal, September 3-8, 2007, Proceedings (Lecture Notes in Computer Science, Vol. 4703)*, Luís Caires and Vasco Thudichum Vasconcelos (Eds.). Springer, 256–271. https://doi.org/10.1007/978-3-540-74407-8_18

Philip Wadler. 1990. Linear Types Can Change the World!. In *Programming concepts and methods: Proceedings of the IFIP Working Group 2.2, 2.3 Working Conference on Programming Concepts and Methods, Sea of Galilee, Israel, 2-5 April, 1990*, Manfred Broy (Ed.). North-Holland, 561.

Aaron Weiss, Daniel Patterson, Nicholas D. Matsakis, and Amal Ahmed. 2019. Oxide: The Essence of Rust. *CoRR* abs/1903.00982 (2019). arXiv:1903.00982 http://arxiv.org/abs/1903.00982

Andrew K. Wright and Matthias Felleisen. 1994. A Syntactic Approach to Type Soundness. *Inf. Comput.* 115, 1 (1994), 38–94. https://doi.org/10.1006/inco.1994.1093

Zipeng Zhang, Xinyu Feng, Ming Fu, Zhong Shao, and Yong Li. 2012. A Structural Approach to Prophecy Variables. In *Theory and Applications of Models of Computation - 9th Annual Conference, TAMC 2012, Beijing, China, May 16-21, 2012. Proceedings (Lecture Notes in Computer Science, Vol. 7287)*, Manindra Agrawal, S. Barry Cooper, and Angsheng Li (Eds.). Springer, 61–71. https://doi.org/10.1007/978-3-642-29952-0_12

Yoav Zibin, Alex Potanin, Paley Li, Mahmood Ali, and Michael D. Ernst. 2010. Ownership and Immutability in Generic Java. In *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA*, William R. Cook, Siobhán Clarke, and Martin C. Rinard (Eds.). ACM, 598–617. https://doi.org/10.1145/1869459.1869509