RELY-GUARANTEE SEMANTICS FOR SEPARATION-LOGIC-BASED SPECIFICATION EXTRACTION

Paul He

A DISSERTATION

in

Computer and Information Science

Presented to the Faculties of the University of Pennsylvania

in

Partial Fulfillment of the Requirements for the

Degree of Doctor of Philosophy

2024

Supervisor of Dissertation

Steve Zdancewic, Schlein Family President's Distinguished Professor of Computer and Information Science

Graduate Group Chairperson

Mayur Naik, Professor of Computer and Information Science

Dissertation Committee

Rajeev Alur, Zisman Family Professor of Computer and Information Science
Benjamin C. Pierce, Henry Salvatori Professor of Computer and Information Science
Stephanie Weirich, ENIAC President's Distinguished Professor of Computer and Information Science
Eddy Westbrook, Senior Research Scientist, Lawrence Livermore National Laboratory

RELY-GUARANTEE SEMANTICS FOR SEPARATION-LOGIC-BASED SPECIFICATION EXTRACTION

COPYRIGHT

2024

Paul He

ACKNOWLEDGEMENTS

First and foremost I must thank my advisor, Steve Zdancewic, who was always supportive of my ideas and interests. I am still amazed at his ability to pinpoint and solve technical problems, and his flexibility and enthusiasm made my PhD experience an excellent one. Eddy Westbrook was my closest collaborator on the work in this dissertation, and a wonderful one, always optimistic and ready to help with each new problem I had. Thank you to Rajeev Alur, Benjamin C. Pierce, and Stephanie Weirich, for serving on my committee and for your role in making Penn a great place to become a computer scientist.

I am immensely grateful that I could learn to not only be a researcher, but also to be an educator during my time at Penn. My time spent teaching was the most rewarding aspect of my PhD. Thank you to all my students and TAs, as well as everyone who helped improve my teaching, in particular Tony Liu, Travis McGaha, Ian Petrie, Swapneel Sheth, and Harry Smith.

My time in Philadelphia was greatly enriched by the support of my friends and family. Thank you to Irene Yoon, who was like a sister to me. Konstantinos Kallas, Yiyun Liu, Mathieu Ouellet, and Anne-Marie Zaccarin put up with the brunt of my complaints, and for that I am grateful.

Finally, this dissertation would not have been possible without the innumerable people who have influenced me in some way. These include the scientists who came before me, other students throughout the many years of schooling, and those who were briefly in my life whose names I never learned. Thank you—we truly stand on the shoulders of giants.

ABSTRACT

RELY-GUARANTEE SEMANTICS FOR SEPARATION-LOGIC-BASED SPECIFICATION EXTRACTION

Paul He

Steve Zdancewic

While formal verification promises correctness guarantees about software, these guarantees themselves must be verified. This dissertation focuses on the soundness of the Heapster verification tool, which converts imperative programs into functional specifications. Heapster is able to do this by using a type system based on separation logic to guarantee memory safety, ensuring that pointer operations can be erased in the functional program. We prove the soundness of this type system using a novel concept called rely-guarantee permissions as the semantics of types. These rely-guarantee permissions are derived from rely-guarantee reasoning, a technique for reasoning about concurrent code. We show that this approach is expressive enough to represent types to typecheck imperative programs that use complex features like pointers, linked lists, and Rust lifetimes. Additionally, we show that the semantics are flexible enough to represent the extraction of equivalent functional programs—with these features erased—as part of the typechecking process. To increase confidence in the correctness of these results and thus in the correctness of Heapster, all our proofs are formalized in Coq.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS iii

ABSTRACT iv

LIST OF FIGURES vii

CHAPTER 1 : Introduction 1

1.1 Motivation 1

1.2 Heapster 2

1.3 Theory of Heapster 5

1.4 Contribution and Attribution 6

1.5 Dissertation Structure 7

CHAPTER 2 : Background 9

2.1 Rely-Guarantee Reasoning 9

2.2 Separation Logic 13

2.3 Interaction Trees 15

CHAPTER 3 : Related Work 18

3.1 Semantics of Separation Logic 18

3.2 Extracting Rust to Functional Specifications 20

3.3 Combining Rely-Guarantee and Separation Logic 22

3.4 Permission-Based Type Systems 23

CHAPTER 4 : Rely-Guarantee Permissions 26

4.1 The Rely-Guarantee Permission Lattice 27

4.2 Coexistence 32

4.3 Permission Changes 35

4.4 Permission Sets 36

CHAPTER 5 : A Simple Separation-Logic Type System 41
5.1 Defining Typing 42
5.2 General Typing Rules 45
5.3 Defining Memory Operations 48
5.4 Type Soundness 51
5.5 Memory Typing Rules 52

CHAPTER 6 : Specification Extraction 61
6.1 Definitions and Semantic Typing 62
6.2 Permission Types 67
6.3 Pointer Types 78
6.4 Array Types 86
6.5 Recursive Types 91
6.6 A Bigger Example 96
6.7 The Heapster Tool 100

CHAPTER 7 : Lifetimes 104
7.1 Defining Lifetime Operations 107
7.2 Lifetime Permissions 109
7.3 Recovering Split Permissions 115
7.4 Lifetime Types and Typing Rules 119
7.5 A Lifetime Example 122
7.6 Differences with Heapster Lifetimes 126

CHAPTER 8 : Future Work and Conclusion 129
8.1 Concurrency 129
8.2 Adding Invariants to Rely-Guarantee Permissions 132
8.3 Conclusion 133

BIBLIOGRAPHY 135

LIST OF FIGURES

FIGURE 1.1 The input code in C, given as input to Heapster. 3
FIGURE 1.2 The output specification in Coq, extracted by Heapster. 4

FIGURE 4.1 Example permissions for a program that allocates new memory. 27

FIGURE 5.1 An example of a heap. 49
FIGURE 5.2 Two possible valid starting and final memory layouts. 59

FIGURE 6.1 Basic structural typing rules. 68
FIGURE 6.2 Typing rules for equality permission types. 69
FIGURE 6.3 Typing rules for conjunction permission types. 71
FIGURE 6.4 Typing rules for disjunction permission types. 73
FIGURE 6.5 Typing rules for pointer types. 81
FIGURE 6.6 Typing rules for arrays. 87
FIGURE 6.7 Typing rules for recursive types. 94
FIGURE 6.8 Typing rules for reachability types. 95

FIGURE 7.1 A Rust program that uses lifetimes implicitly. 105
FIGURE 7.2 The life cycle of a lifetime ℓ. 107
FIGURE 7.3 How we *want* to use a lifetime ℓ to split a permission π. 109
FIGURE 7.4 How we will use a lifetime ℓ to split a permission π. 114
FIGURE 7.5 How we will use a lifetime ℓ to split a pointer type. 119

CHAPTER 1

Introduction

1.1. Motivation

As software complexity grows, so does the potential for errors. Typical techniques used in software development for ensuring the reliability and correctness of software, like testing and code reviews, can only show the presence of errors and not their absence. In contrast, formal verification can prove formal guarantees about program correctness. This is particularly important for high-stakes applications, like those in the healthcare or automotive industries, where software bugs can have catastrophic consequences. Consequently, to have full confidence in the strong claims made by formal verification tools, these tools must also be reliable and undergo verification themselves.

Imperative languages form one large class of programming languages. Code written in these languages, with their use of memory and pointers, is difficult to verify. One approach for handling imperative programs is separation logic [Rey02], a family of program logics that are effective for modularly verifying pointer-manipulating code. Some separation logics, like Iris [Jun+15] and the Verified Software Toolchain (VST) [App11], have very strong reasoning principles and support many language features. This power has the cost of requiring skilled users to perform the manual verification effort. Other separation logic tools like Infer [Cal+15] focus on automated verification, and must settle for weaker rules and guarantees—a tradeoff for automation and scalability. A second approach for preventing bugs in imperative code that has seen widespread adoption is the type system of the Rust programming language [KN23]. Rust's strong type system is based on linear types to guarantee memory safety. Its notion of ownership in the type system prevents the creation of dangling pointers, thereby preventing a large class of memory errors.

As for the reliability of these verification approaches, there are a variety of methods for formally proving the soundness of these systems. Most separation logics have their own semantics for assertions in the logic, and more complex logics with powerful rules have correspondingly complex semantics and proofs of soundness. Soundness of the Rust type system was proven using

the Iris separation logic [Jun+17], leveraging the ability of separation logic to easily represent ownership of resources, a key concept in the type system. In recent years, Iris in particular has gained popularity as a feature-rich and well-supported separation logic framework, becoming a preferred logic for implementing other verification tools and proving their soundness [HBK19; MJP20; VB23; VP23; Biz+19].

1.2. Heapster

In this dissertation, I focus on the soundness of a verification tool developed at Galois Inc. called Heapster [He+21]. Heapster converts imperative programs to functional ones, removing complex pointer reasoning. Heapster uses ideas from both of the approaches discussed above for verifying imperative code. It uses a type system—one strongly based in separation logic—that has similar restrictions as Rust and that also guarantees memory safety. It is semi-automated, requiring users to supply type annotations during the verification process. Unlike most verification tools, the goal of Heapster is not to prove any specific property of programs, but to *extract* a functional specification from an imperative program. This functional specification—since it no longer contains any memory operations—is then easier to verify than the original imperative program.

The crux of Heapster's approach is the idea that memory-safe imperative programs have equivalent pure functional programs. Several other systems like Electrolysis [Ull16], RustHorn [MTK20] and Aeneas [HP22] have also used this idea by focusing on Rust, since Rust's type system guarantees memory safety. As such, Heapster's approach can be thought of as requiring users to provide Rust-like types for code written in other imperative languages. Chapter 7 also discusses how Heapster can take in Rust code as input directly and simplify the user experience.

As an example of Heapster in action, consider the example by Silver et al. [Sil+23b, Section 3] in Figure 1.1. The input to Heapster is a function written in C for checking whether a value `x` is in a linked list `l`. This function, like all pointer-manipulating code in C, is not memory safe. While the argument `l` has a pointer type, it is not guaranteed to point to either null or a valid linked list node. It could point to something else entirely, which would cause this function to fail. To ensure that this code *is* memory safe and therefore extractible to a functional specification, the user must

```
typedef struct list64_t {
  int64_t data;
  struct list64_t *next;
} list64_t;

int64_t is_elem (int64_t x, list64_t *l) {
  if (l == NULL) {
    return 0;
  } else {
    if (l->data == x) {
      return 1;
    } else {
      list64_t *l2 = l->next;
      return is_elem(x, l2);
    }
  }
}
```

Figure 1.1: The input code in C, given as input to Heapster.

provide type annotations for this function to Heapster:

$$(\texttt{x} : \mathsf{int64}, \texttt{l} : \mathsf{list64}) \to \mathsf{int64}$$

The $\mathsf{int64}$ type is a functional representation of a 64-bit integer as a bitvector (defined like Nat in Section 6.2.2, using an existential type), and $\mathsf{list64}$ is a functional version of a list of 64-bit integers (defined like $\mathsf{List}\ A$ in Section 6.5). The $\mathsf{list64}$ type is defined as a recursive type, telling us that the pointer with this type must be either null or point to an $\mathsf{int64}$ and another $\mathsf{list64}$. This type rules out invalid pointers, and the additional information in the type allows typechecking to produce the functional specification for `is_elem` in Figure 1.2.

This specification program is built using `rec_fix_spec`, a fixed point combinator. Within the body of the generator function for this recursive function, the argument `rec` is used to refer to the recursive call. While it is not as simple as standard pattern matching on a list type, this functional program is essentially pattern matching on the structure of the list `l`. The exact details of this pattern matching using an unfold operation will be presented in Section 6.5. In the base case where the list is empty, the program returns 0. In the recursive case where the list is nonempty, the front

```
Definition is_elem_spec : bitvector 64 * list (bitvector 64) ->
                          itree_spec E (bitvector 64) :=
  rec_fix_spec (fun rec '(x, l) =>
    either                                      (* eliminator for the sum type *)
      unit (bitvector 64 * list (bitvector 64))  (* input types *)
      (itree_spec _ (bitvector 64))             (* output type *)
      (fun _ => Ret (intToVc 64 0))             (* nil case *)
      (fun '(hd, tl) =>                         (* cons case *)
        if bvEq 64 hd x
        then Ret (intToVc 64 1)                 (* return 1 *)
        else rec (x, tl))                       (* recursive call *)
      (unfoldList l)).                          (* unfolded list argument *)
```

Figure 1.2: The output specification in Coq, extracted by Heapster.

of the list is compared with the element `x`, and continues by either returning 1 or continuing to recurse.

With this functional specification in hand, one can then prove arbitrary properties of this program. Heapster provides automation for common cases, like verifying that the specification cannot result in an error, or that it refines another, higher-level specification [Sil+23b]. This task of relating imperative programs to functional specifications is a common one for verifying the functional correctness of programs. As shown by Silver [Sil23, Chapter 5], the process of doing this in tools like VST can be tedious—even if the code is simple—due to the fact that users must reason about pointers using separation logic. It is this tedious but conceptually simple process that Heapster automates. By removing imperative features automatically, relating the extracted specification to a higher level specification then only has to reason about its functional behavior.

Since automation is Heapster's primary goal, it is limited in the features it can support in imperative programs. Like Rust, Heapster's type system must be overly conservative and does not support complex uses of pointers that involve circularity, like doubly linked lists, even when they are used safely. As I will show in Example 5.21 which uses a circular linked list, Heapster *can* in theory typecheck programs that use pointers in more complex ways. However, these more complex uses of pointers are more difficult to typecheck automatically. So while it is theoretically possible to handle such programs, Heapster does not support these uses of pointers for the sake of

automation. As such, Heapster has only been used for programs that have simple uses of pointers, such as singly-linked lists, and I also focus primarily on these use cases in this dissertation.

1.3. Theory of Heapster

This dissertation presents the theory of Heapster, mechanized in the Coq proof assistant, justifying the correctness of Heapster's type system. I will use rely-guarantee reasoning, a technique for reasoning about concurrent code, as a way to prove the soundness of the Heapster type system.

> My thesis is that *rely-guarantee reasoning is a good semantic foundation for separation-logic style reasoning to extract functional specifications from imperative programs*.

For the soundness of the Heapster type system, I use a construct called rely-guarantee permissions, based on rely-guarantee reasoning, to define the semantics of types. Typing judgments in this type system are written as $\Pi \vdash t_i \lessapprox t_s : T$ in the theory of Heapster, and is defined using a bisimulation relation. The arguments t_i and t_s correspond to the input imperative program and the extracted specification program, respectively. The other two components of the typing judgment, Π and T, roughly correspond to the type annotations that users supply to Heapster, where Π are the types of the inputs to a function, and T are the types of outputs from a function. When represented this way, the Heapster type system resembles a separation logic, where Π and T are the precondition and postcondition, respectively, for the pair of programs. Indeed, this type system can be viewed as a separation logic, and my approach of semantic type soundness corresponds to the standard approach for proving soundness of a separation logic by defining a semantic model of assertions.

While we use rely-guarantee permissions as the semantics of types, this is not the only possibility. Using an existing semantics for separation logic to represent types, or implementing the type system in Iris, as many other tools have done, would have likely have also been successful for proving soundness. There are, however, several reasons why rely-guarantee semantics are useful for verifying Heapster. First, rely-guarantee reasoning uses relations to describe how program states can change, rather than most separation logic semantics which use predicates to describe

the current state. This allows us to express very fine-grained constraints, as rely-guarantee permissions can describe the exact state changes permitted by each type, which is especially useful for representing lifetime types. Additionally, since these relations describe the entire change in state, this results in a language-agnostic semantics for the types—useful for a relation involving two different languages. Second, Heapster was designed with these rely-guarantee semantics in mind. Because of this, many of the types have very natural definitions using rely-guarantee permissions, and would have likely required more convoluted encodings using other semantics. Lastly, many of the complex features of other separation logics are not present in Heapster, due to its focus on automated specification extraction. While features like higher-order state and concurrency are very useful for general-purpose separation logics like Iris, the extra complexity they add would not have any benefits for Heapster.

This work on the theory of Heapster was done after the core of Heapster was developed by my collaborators, and is thus a theoretical justification of an existing system. While the theory did not uncover any bugs in the Heapster implementation, it did help in explaining the intuition behind the implementation, and in communicating the usefulness of the tool.

1.4. Contribution and Attribution

The contribution of this dissertation is the theory of Heapster: a semantic interpretation of the Heapster type system, defined using rely-guarantee permissions. Using this semantics, I prove the soundness of the type system using a semantic type soundness approach. The definitions and proofs are formalized in the Coq proof assistant, and the formalization can be found at

> https://github.com/Grain/thesis-formalization/.

Part of the work described in this dissertation has been presented in the paper "A Type System for Extracting Functional Specifications from Memory-Safe Imperative Programs" by He et al. [He+21]. Of the content of this paper, the development of Heapster was done by my collaborators at Galois, and Theorem 4.6 (Theorem 6.13 in this dissertation) was done by Matthew Yacavone and Eddy Westbrook. These are not contributions of this dissertation, though the theorem is included

in the proof development repository for completeness. I was the main contributor to the rest of the paper, and the technical material in Chapters 4 and 6 are based on that work, as is some of the related work in Chapter 3. The technical material described in Chapters 5 and 7 have not been published before, though Chapter 5 includes work that was done in the process of preparing the aforementioned paper.

1.5. Dissertation Structure

Chapter 2 introduces some crucial background topics. First, rely-guarantee reasoning, which is the basis of the definition of rely-guarantee permissions. Second, separation logic, which is a major motivator of the Heapster type system that is proved sound with rely-guarantee permissions. Finally, interaction trees, which are used to represent programs in the theory of Heapster.

Chapter 3 describes related work and compares it to the theory of Heapster presented in this dissertation.

Chapter 4 introduces rely-guarantee permissions, the key definition that I will use throughout this dissertation for representing types in the theory of Heapster. This chapter introduces some intuitions about rely-guarantee permissions and basic definitions without yet defining how programs are related to these permissions.

Chapter 5 defines a language with a heap and uses rely-guarantee permissions to define a basic separation-logic type system. The intuitions from the previous chapter are formalized in these definitions, and I prove that this type system is sound.

Chapter 6 generalizes the type system in the previous section to work with two programs, an imperative heap-manipulating program as in the previous section, and a pure functional program. This models the Heapster type system, which extracts a functional specification from an imperative program. I again prove soundness of this new type system, present the semantic interpretation of types in Heapster, and prove the typing rules of the type system.

Chapter 7 further extends the specification-extraction type system by supporting the lifetimes

present in the Heapster type system. Lifetimes are used in Heapster to both support Rust types which use lifetimes, and to increase the power of the type system. I use rely-guarantee permissions to define the semantic interpretation of several new types related to lifetimes, and prove their associated typing rules.

Finally, Chapter 8 discusses several directions for future work and concludes.

CHAPTER 2

Background

This chapter introduces three topics used heavily in the rest of the dissertation. First, we introduce rely-guarantee reasoning, the core of the rely-guarantee permissions we will use as the semantics of types in the type system. Second, we discuss separation logic, one of the core concepts used in the design of the type system. Third, we describe interaction trees (ITrees), a data structure that we use to represent programs in the Coq formalization of the type system.

2.1. Rely-Guarantee Reasoning

Rely-guarantee reasoning [Jon83] is an extension of Hoare logic [Hoa69] designed to reason about shared-memory concurrent programs. It originates from earlier work for reasoning about concurrency by Owicki and Gries [OG76], which focuses on proving that if two pieces of code do not *interfere*, then they can be safely run concurrently. The so-called Owicki-Gries method provides the following rule to prove a Hoare triple for the parallel composition of two threads:

$$\frac{\{P_1\}C_1\{Q_1\} \qquad \{P_2\}C_2\{Q_2\} \qquad C_1 \text{ and } C_2 \text{ are } \textit{interference-free}}{\{P_1 \cap P_2\}C_1 \parallel C_2\{Q_1 \cap Q_2\}}$$

Here, interference-freedom is defined as all assertions in each parallel statement remaining true when state changes occur in other statements.

For example, this rule can verify that two threads which change different variables works properly when run concurrently:

$$\frac{\{x = 0\}x := 1\{x = 1\} \qquad \{y = 0\}y := 2\{y = 2\} \qquad \text{these two threads are interference-free}}{\{x = 0 \wedge y = 0\}x := 1 \parallel y := 2\{x = 1 \wedge y = 2\}}$$

The interference-freedom portion can be proven by considering all combinations of assertions and statements, and seeing that they do not change the validity of the preconditions and postcondi-

tions in the other thread. Namely, we have to check that the following four Hoare triples hold:

$$\{x = 0\}\ y := 2\ \{x = 0\} \quad \{y = 0\}\ x := 1\ \{y = 0\}$$
$$\{x = 1\}\ y := 2\ \{x = 1\} \quad \{y = 2\}\ x := 1\ \{y = 2\}$$

However, a major problem with this approach is that it is not *compositional*: the interference-freedom property cannot be obtained by composing specifications of its individual threads. If we had a third thread that sets a different variable z, then we would need to look at both the implementation and the assertions of this thread and perform more checks to ensure interference-freedom. In addition to the four checks above, we would also have to check the validity of *eight* additional Hoare triples if we were to compose the new thread $z := 3$ to the two threads from before:

$$\{x = 0\}\ z := 3\ \{x = 0\} \quad \{z = 0\}\ x := 1\ \{z = 0\} \quad \{z = 0\}\ y := 2\ \{z = 0\}$$
$$\{x = 1\}\ z := 3\ \{x = 1\} \quad \{z = 3\}\ x := 1\ \{z = 3\} \quad \{z = 3\}\ y := 2\ \{z = 3\}$$
$$\{y = 0\}\ z := 3\ \{y = 0\}$$
$$\{y = 2\}\ z := 3\ \{y = 2\}$$

This approach does not scale. The issue is that the specification of the thread does not specify its interference, so we need to inspect the implementation of each thread to conclude that they do not interfere. Jones aimed to fix this compositionality issue with rely-guarantee reasoning [Jon83], which adds *rely* and *guarantee* relations to the specifications of programs. These relations, in conjunction with the existing preconditions and postconditions, express the information needed to check for interference-freedom.

The rely relation, a binary relation on states, describes what a thread expects of the environment: the thread can *rely* on the fact that state changes made by other threads satisfy this relation. Similar to the interference-freedom check of Owicki and Gries, a valid rely relation must be checked so that the preconditions and postconditions are *stable* under it: any state change allowed by the rely must not change whether the state satisfies the precondition or postcondition. The guarantee relation, also a binary relation on states, describes how a thread will change the state: the thread

is *guaranteed* to only make state updates that are in the relation. A valid guarantee relation must be checked so that all state changes in a program satisfy the relation. With these new relations, showing interference-freedom is much easier—each thread's guarantee just needs to be included in every other thread's rely.

With this richer specification, rely-guarantee reasoning can verify concurrent programs compositionally with the following rule:[1]

$$\frac{\{P_1, R_1\}C_1\{G_1, Q_1\} \qquad \{P_2, R_2\}C_2\{G_2, Q_2\} \qquad G_1 \subseteq R_2 \qquad G_2 \subseteq R_1}{\{P_1 \cap P_2, R_1 \cap R_2\}C_1 \parallel C_2\{(G_1 \cup G_2)^*, Q_1 \cap Q_2\}} \text{ PAR}$$

where $(-)^*$ denotes reflexive-transitive closure.

This rule is similar to the parallel composition rule of Owicki and Gries, but the non-interference portion is replaced by a simpler check: that each thread's guarantee is included in the other's rely. As with the previous rule, the precondition and postcondition are combined using intersection, as is the rely, since the parallel composition of both threads will rely on any changes relied upon by either thread. The guarantee is combined using the reflexive-transitive closure of the union of both guarantees. The union captures the fact that any updates performed by parallel composition must originate from one of the two threads. The reflexive-transitive closure ensures that combinations of updates from both threads—as well as the case where no update has occurred at all—are included.

This fixes the compositionality issue from the approach of Owicki and Gries, allowing us to compose specifications using parallel composition without needing to inspect the implementation of each. We first need to obtain a specification for the individual threads, which requires the checks

[1] For presentation purposes, this is a simplified version of the actual rule presented by Jones. Combining the various relations is more complex in the original system.

mentioned above:

$$\{\overbrace{x = 0}^{P}, \overbrace{x = x'}^{R}\} x := 1 \{\overbrace{x' = 1 \land y = y' \land z = z'}^{G}, \overbrace{x = 1}^{Q}\}$$

$$\{y = 0, y = y'\} y := 2 \{x = x' \land y' = 2 \land z = z', y = 2\}$$

$$\{z = 0, z = z'\} z := 3 \{x = x' \land y = y' \land z' = 3, z = 3\}$$

In the definitions of the relies and guarantees, we use x and x' to denote the initial and final values of x, respectively. These specifications' relies say that each thread requires that other threads do not change the variable they are changing. The guarantees say that each thread changes their own variable, and do not change the other variables in the program.

Then, to compose them together using the parallel composition rule, we simply need to check their relies and guarantees, and not the implementation of each thread. Applying the parallel composition rule twice gives us the following specification for the entire program:

$$\{x = 0, y = 0, z = 0, x = x' \land y = y' \land z = z'\}$$

$$x := 1 \parallel y := 2 \parallel z := 3$$

$$\{(x = x' \land y = y' \land z = z') \lor \cdots \lor (x' = 1 \land y' = 2 \land z' = 3), x = 1 \land y = 2 \land z = 3\}$$

While this provides compositionality, a major weakness of rely-guarantee reasoning is that thread specifications cannot be created completely independently from other threads. Such a property, called *modularity*, is highly desirable in a real system with many independent pieces and many contributors. With rely-guarantee reasoning, a thread's specification must include information about the entire global state, not just state local to each thread. This is because rely and guarantee relations must include all possible behaviors for the parallel composition rule to check whether two threads interfere. For example, the above specifications for the individual threads need to state in their guarantees that they do not modify the variables used in other threads, in order for their guarantees to be included in other threads' relies. If we wanted to add a fourth thread which

sets a new variable w to 0, then the fact that the original three threads do not modify w would have to be added to their guarantees. The next section on separation logic addresses this lack of modular reasoning.

2.2. Separation Logic

Separation logic [Rey02] is an extension of Hoare logic to handle verification of heap-manipulating programs in a modular way. Unlike the program logics in the previous section, the original goal of separation logic was not to handle concurrent code, though it was eventually extended to concurrency as well. A central concept is the separating conjunction, $*$, which, unlike regular conjunction, states that two assertions are true on two *separate* parts of the heap. These partial heaps are separate, written as $h_1 \perp h_2$, if they have disjoint domains. Then $P * Q$ means that P holds on h_1, Q holds on h_2, and the overall heap can be split into h_1 and h_2 such that $h_1 \perp h_2$. This solves the modularity issue of rely-guarantee and earlier program logics: specifications only need to include state that they depend on. Since the specification is about a disjoint portion of the heap, every other part of the heap can be assumed to stay unchanged. For example, an individual thread from the example in the previous section now has a much simpler specification:

$$\{x \mapsto 0\} x := 1 \{x \mapsto 1\}$$

The points-to assertion $p \mapsto v$ says that a pointer p points to memory containing the value v.

Other parts of the state that are not affected by the program can be added by the following rule:

$$\frac{\{P\}C\{Q\}}{\{P * R\}C\{Q * R\}}$$

This rule is named the frame rule, after the frame problem [Hay71] in artificial intelligence for the problem of having to specify which aspects of the world remain unchanged after an action is performed, exactly the problem we had earlier.

Concurrent separation logic (CSL) [OHe07] was soon developed to extend separation logic to

concurrent heap-manipulating programs. If two threads can be run with disjoint portions of the heap, then they cannot interfere, as shown by this rule:

$$\frac{\{P_1\}C_1\{Q_1\} \quad \{P_2\}C_2\{Q_2\}}{\{P_1 * P_2\}C_1 \parallel C_2\{Q_1 * Q_2\}}$$

A key observation from the development of CSL is that separation logic specifications express a notion of *ownership*. Since assertions hold on disjoint parts of the heap, if we have a points-to assertion $p \mapsto v$ in the specification of a thread, this is equivalent to the thread having exclusive ownership of p, since it could not be composed with another thread that has $p \mapsto v'$ in its specification. Comparing to rely-guarantee reasoning, having exclusive ownership is like having a rely that requires the owned memory to stay the same, and a guarantee that allows the owned memory to be changed arbitrarily. Using this parallel composition rule, we can verify the same parallel program as the previous section with a far more succinct specification:

$$\{x \mapsto 0 * y \mapsto 0 * z \mapsto 0\}x := 1 \parallel y := 2 \parallel z := 3\{x \mapsto 1 * y \mapsto 2 * z \mapsto 3\}$$

There has been much work building on the original concept of separation logic. Many works extended separation logic with additional features, and some used the notion of "fictional" separation, where separating conjunction did not have to mean disjointness of heaps, but could be some more abstract notion of separateness [DGW10; Din+13]. The theory of Heapster falls into this category of "fictional" separation. In our approach, which we will describe in Chapter 4, we use rely-guarantee permissions as the semantic interpretation of types similar to separation logic assertions. By using the rely and guarantee relations, we can express fine-grained properties, resulting in very different definitions and intuitions of separateness and the separating conjunction than the standard ones.

2.3. Interaction Trees

To represent programs in the Coq formalization of the Heapster theory, we will use interaction trees. Interaction trees [Xia+19] are a data structure in Coq [Bar+97] for representing effectful and possibly nonterminating programs. They come with a library of equational reasoning and other lemmas for manipulating ITrees. By using Coq's extraction feature to extract ITrees to code in other functional languages, ITrees are also executable, allowing for testing and linking with other code. For our purposes, they are a good medium between a shallow embedding where programs are represented as functions in Coq, and a deep embedding where programs are represented purely syntactically. While Coq functions are easily executable and easy to work with in proofs, they are also required to be pure and terminating. On the other hand, syntactic representations can be used to express any program, but the semantics of programs must be defined from scratch.

An ITree represents a tree containing interactions with the environment, which we call *events*. Depending on how the environment responds to these events, the computation can vary, resulting in the branching structure of the tree. For example, there could be an event to request for a natural number from the environment. Depending on how this event is used, it could represent getting an input from a theoretical user, or getting a number from some source of randomness, or any number of other interpretations.

Formally, the type of an ITree is $\mathsf{itree}\ E\ R$. The parameter E is the event signature, a type that captures the possible events in an ITree. This parameter is indexed by the type of response each event expects from the environment. For instance, the previous example of an event which gets a natural number from a user would have the type $E\ \mathbb{N}$ to represent the fact that it expects a value of type $\mathbb{N}$ back from the environment. The parameter R is the type of leaf values in the tree, which represent the values returned when the program terminates.

The tree structure of an ITree is defined by three kinds of nodes. First, $\mathsf{ret}\ r$ is a leaf node which represents a program terminating with a value of type R. Second, $\mathsf{vis}\ e\ k$ represents a visible event e of type $E\ X$ for some response type X and a continuation k of type $X \to \mathsf{itree}\ E\ R$. These nodes

represent communication with the environment. Once the environment responds with a response $x : X$, it continues with the ITree $k\ x$. It is this continuation which induces the branching structure of an ITree. Lastly, ITrees can contain a $\tau\ t$ node, a silent step of computation followed by another ITree of type $\mathsf{itree}\ E\ R$. This allows us to represent nonterminating programs, like the diverging program $\mathsf{spin} \stackrel{\text{def}}{=} \tau\ \mathsf{spin}$. To model these infinite behaviors, ITrees are defined as a coinductive data type in Coq.

ITrees have many useful properties for modeling programs. For one, they are a shallow embedding, so ITrees inherit features from Coq, like variable binding, pattern matching, and control flow. The type $\mathsf{itree}\ E$ also forms a monad, where the return operation is simply ret and a bind operation $t \gg= k$. This bind sequences the ITree t followed by the continuation k, replacing each $\mathsf{ret}\ r$ leaf in t with the subtree $k\ r$. We will sometimes write bind using the notation $x \leftarrow t; k\ x$ in place of $t \gg= (\lambda x.\ k\ x)$, omitting the $x \leftarrow$ portion if k does not use x. Furthermore, ITrees can model iteration using an iter combinator of type $(A \rightarrow \mathsf{itree}\ E\ (A + B)) \rightarrow A \rightarrow \mathsf{itree}\ E\ B$. The first argument is the body of the loop, which takes a value of type A and either returns a value of type B to signal that iteration is finished, or returns another value of type A to signal that iteration should continue with this new value. These values of A are the only things that change between iterations, and so we will call them *iteration variables*. Applying the iter operation as $\mathsf{iter}\ \mathit{body}$ then takes an initial A to start this process, and produces the resulting ITree, which—if it does not diverge—will return a final value of type B.

As presented so far, ITrees contain *uninterpreted* events, but events can also be given semantics. The ITree library provides a way to interpret events in an ITree using a handler of type $E\ X \rightarrow M\ X$ for some monad M. For example, state events can be handled into a state monad, and interpreting the state events using this handler removes the original state events from the ITree and gives them semantics via the state monad transformer.

Heapster uses a variant of ITrees called ITree specifications [Sil+23b] to represent the extracted functional specifications, which was used in the example in Figure 1.2. That example used a fixed point combinator to represent recursive functions, which is supported in both ITrees as well as

ITree specifications, though we do not use it in the theory of Heapster presented in this dissertation. Instead of using recursion, we represent such programs using `iter`. For the example in Figure 1.2, rather than making a recursive call using the `x` and `l` variables, we instead iterate, using those variables as iteration variables.

The theory of Heapster uses ITrees and Heapster itself uses ITree specifications, incompatible variants of ITrees. One future direction would be to connect the theory to the extracted specifications from Heapster by changing the formalization to represent specification programs as ITree specifications. By switching to the same data structure, it would be possible to implement something similar to Heapster in Coq using the typing rules we prove in this dissertation, giving us fully-verified end-to-end guarantees, and also giving us access to the infrastructure Heapster provides for verifying those specifications. This would also let us connect Heapster to previous work using ITrees to model programs written in various programming languages [Zak+21; Zha+21; Sil+23a], and to extract specifications from those programs.

CHAPTER 3

Related Work

Part of this chapter is adapted from work previously published as Paul He, Eddy Westbrook, Brent Carmer, Chris Phifer, Valentin Robert, Karl Smeltzer, Andrei Ştefănescu, Aaron Tomb, Adam Wick, Matthew Yacavone, and Steve Zdancewic. "A Type System for Extracting Functional Specifications from Memory-Safe Imperative Programs." In: *Proc. ACM Program. Lang.* 5.OOPSLA (Oct. 2021) [He+21]. I was the primary author and the primary contributor of the theoretical component of the paper, the focus of this dissertation.

This chapter explores previous work related to this dissertation. These topics are categorized into several main areas. First, we discuss other semantics of separation logics, which is the same methodology we use to prove soundness of the Heapster type system. Second, there are other systems that extract specifications from Rust code, like Heapster. We compare the soundness proofs of these systems with our theory of Heapster. Third, rely-guarantee reasoning has been used in combination with separation logic before, and previous approaches resemble our use of rely-guarantee permissions. Lastly, the permission-based approach of Heapster has been used in other type systems, and we compare their soundness results with our approach.

3.1. Semantics of Separation Logic

The work presented in this dissertation can be viewed as the semantic model of a separation logic, rather than a type system. The approach of using semantic type soundness is the same, and the Heapster type system blurs the line between type systems and program logics due to the expressivity of its types. In this section, we compare our semantics using rely-guarantee permissions to the semantics of other separation logics.

Traditionally, the assertions of separation logics are modeled as predicates over heaps, and separating conjunction splits the heap into disjoint pieces [Rey02]. As the field of separation logic grew, more complex separation logics sprang up, each with their own semantics. These logics

typically represent the semantics of their assertions as predicates in increasingly complex logics, or by adding more complex representations of state. While early work used the heap as the state, work on "fictional separation" [JB12; Sie+15] generalized the state into user-defined types, where separation no longer necessarily represented physically splitting the heap. This allowed for more flexible notions of separation that can reason about the splitting of logical resources.

During this time there was an explosion in the number of different separation logics, each with their own implementations and semantics [Par10]. There was work to unify separation logics in a common framework, which today has emerged as Iris [Jun+15], an influential concurrent separation logic framework implemented in Coq. The logic is very expressive, containing higher order state and impredicative assertions among other features, and has been used to implement many other separation logics.[2] Their semantics continues to use predicates, but in a far more complex way, crucially using step-indexing to handle the use of higher-order state in Iris. One significant related project built using Iris is Rustbelt [Jun+17], which formalizes the semantics of Rust and formally proves the safety guarantees of the language. Similar to our approach in Chapter 7, they formalize lifetimes and borrowing by defining the semantics of borrowed types with logical relations in the Iris logic. This more complex approach is required to fully support Rust types, while we focus on the simpler uses of lifetimes in Heapster.

In this dissertation, we use a different semantic domain from other separation logics. The Heapster type system can be viewed as a separation logic, albeit one without the focus on separation logic connectives like the separating implication that most other separation logics feature. As we focus on Heapster, where automation is the goal, such connectives are not needed, though we still are able to define the separating implication operation, $-\!*$, with rely-guarantee permissions. In this view of Heapster as a separation logic, our semantic model is not predicates over states, but rely-guarantee permissions over states, which use binary relations. This has the benefit of being able to encode powerful rules and features using rely-guarantee, but hiding the complexity of rely-guarantee from the user-facing type system or logic. By using rely-guarantee permissions

[2] For a full list of papers using Iris, see https://iris-project.org/.

as an “implementation” of separation logic, we avoid the main downside of using rely-guarantee reasoning as a user, the lack of modular reasoning. This model also uses “fictional” separation by default, since rely-guarantee permissions do not have physical separation of heaps built in to their definitions. As such, rely-guarantee permissions should be able to act as a semantics for simple separation logics close to the original definition of Reynolds [Rey02].

Another clear difference between the theory of Heapster and most separation logics is the presence of a specification program in the typing relation. Relational separation logic [Yan07] is one variant of separation logic that resembles our typing relation in that it is also used to relate two programs. Unlike Heapster, this work uses separation logic to relate two pointer-manipulating programs in the same language, rather than an imperative program and a functional program. This logic can be used, for example, to show that two implementations of the same algorithm are equivalent. Relational separation logic uses a fairly standard heap semantics of assertions, but with two heaps for the two programs in the relation. Later work on similar relational logics in Iris [Gäh+22] or using rely-guarantee reasoning [LFF12] extend this idea to more expressive languages with concurrency. These logics are based around a simulation relation, similar to the bisimulation relation at the heart of the theory of Heapster. The goals of these works differ from that of Heapster, however. Both of these logics are aimed at proving the correctness of program transformations for concurrent programs, which results in more complex logics where verification is not automatic.

3.2. Extracting Rust to Functional Specifications

Several works use the same idea as Heapster: that memory safe Rust programs are equivalent to functional programs. The proofs of soundness for these approaches vary, and all differ from the approach we take with rely-guarantee permissions, since none use separation-logic reasoning.

An early work in this area, Electrolysis [Ull16], translates Rust programs to functional programs in the Lean proof assistant [Mou+15]. They represent functional programs using a monad for non-termination, much like our use of ITrees. To translate Rust references, they use lenses [Fos+07] to represent them as a part of the original mutable variable. This has some limitations, like not being

able to represent mutable references within larger structures or references modified in loops. The translation process directly maps Rust features to Lean features, and does not have a soundness proof.

The Aeneas system [HP22] also uses the knowledge from the guarantees given by Rust types to extract functional programs from Rust. This work extracts Rust to a generic lambda calculus term in F* [Swa+16], but could be represented in any language or proof assistant. Due to this representation, the functional programs cannot represent nontermination, and requires proof about termination or a fuel parameter for recursive functions. Borrowing is at the core of this work. The translation from Rust programs contains a *backward function*, which describes the effect of ending a borrow and updating the borrowed variable. This is similar to the lifetime types in Heapster, which represents a borrow on the specification side as a function that returns all the borrowed values, which can only be applied once we end the lifetime. The treatment of these types in the theory of Heapster is slightly different, as we describe in Section 7.6.

Ho, Fromherz, and Protzenko [HFP24] prove the soundness of part of the Aeneas approach. First, they focus on the internal representation of Rust in Aeneas, which has some unusual features, like not having an explicit heap. This paper relates this representation to a more traditional language closer to Rust, with a standard heap model. The paper then proves that the next step of Aeneas, going from this internal representation to a symbolic semantics is sound: that the concrete semantics refine the symbolic semantics. The final step of the Aeneas tool, the translation from the symbolic semantics to the lambda term, is not yet proven to be sound.

RustHorn [MTK20] reduces Rust programs to constrained Horn clauses (CHCs), which erases memory operations like Heapster. Properties that one wants to verify, like functional correctness, can then be encoded in the CHCs, and existing CHC solvers can be used to solve the resulting clauses. To handle borrows, they use the notion of *prophecy* variables to refer to the future value of the variable when the borrow ends. Soundness of this reduction is proven by a bisimulation between the original programs to the CHC representation. Later work on RustHornBelt [Mat+22] combines RustHorn with the RustBelt project. This work adds the extracted CHC specifications

of RustHorn to RustBelt, resulting in a similar typing judgment to ours, where an imperative program is related to a specification using CHCs. Nakayama et al. [Nak+24] extend RustHorn in a different direction, and adds fractional permissions [Boy03], allowing for both the borrowing and the fractional splitting of references. They show that this addition increases the expressivity of RustHorn, which suggests that fractional permissions may be useful in our type system as well. The soundness of this extension is proved in a similar way as RustHorn, using a simulation relation.

3.3. Combining Rely-Guarantee and Separation Logic

Since the first introduction of concurrent separation logic, there has been a desire to extend it to support more fine-grained concurrency. Since rely-guarantee allows for the expression of fine-grained changes through its relations, there are several works that add rely-guarantee reasoning to concurrent separation logic, such as RGSep [VP07], SAGL [FFS07], LRG [Fen09], and deny-guarantee [Dod+09]. Most of these logics prove soundness by adding rely-guarantee relations to a more standard separation logic semantics. SAGL, however, proves soundness of its logic syntactically, rather than the more standard semantic approach.

Rather than combining rely-guarantee and separation logic directly, rely-guarantee permissions *implement* a separation logic using rely-guarantee reasoning, allowing for a simpler interface. While we do not yet support concurrent code with rely-guarantee permissions (see Section 8.1 for a discussion of future work), once we do, it should be possible for rely-guarantee permissions to act as a semantics for logics like these due to its ability to represent both rely-guarantee relations and separation logic assertions.

Interestingly, Dodds et al. consider a similar definition to ours for the separating conjunction of rely-guarantee relations, but they discard the idea since the definition is not *cancellative*. Cancellativity is defined as the property that if $x * y = x * z$, then $y = z$. This property is required to form a separation algebra [COY07], which they used as the basis of the soundness of their work. Later work generalized to models that do not have the requirement of cancellativity [BV14], so this suggests that there is no inherent barrier in using our approach with rely-guarantee permissions for

other systems like deny-guarantee.

3.4. Permission-Based Type Systems

This section discusses some related work in using types to represent notions of permission or capability to control aliasing and ownership, some of which use rely-guarantee reasoning as well. Most of these type systems are proved sound using a syntactic approach, with progress and preservation theorems [WF94]. Our approach using semantic type soundness is useful if the type system is updated. If additional types are added that do not require changes to the semantic model, then existing proofs for typing rules remain the same. For example, this is the case when we add lifetime types in Chapter 7. Additionally, having a semantic model of types can be useful for design intuitions, rather than having to rely on the typing rules as the only meaning of the types in the type system.

Ownership types [CPN98] are an early instantiation of the idea of using types to represent ownership information in an object-oriented setting. These types are only used to control aliasing, and do not represent *permissions* as our work and later work does, where these permissions can be transferred and change during execution.

As separation logic was developed and became more popular, ideas of ownership transfer and permissions became more widespread. Two projects, Mezzo [PP13] and Asynchronous Liquid Separation Types [KMV15], both use type systems with some element of permission reasoning inspired by separation logic. These works build type systems for realistic functional languages, rather than existing imperative ones.

Nanevski, Morrisett, and Birkedal [NMB08] present a type theory that includes Hoare logic specification in types, similar to the theory of Heapster. They further show how separation logic connectives can be defined in the system. This system is a dependent type theory where much of the emphasis is put on generality, and typechecking is undecidable—whereas Heapster focuses on automation.

Gordon, Ernst, and Grossman [GEG13] uses rely-guarantee relations in a similar way to the the-

ory of Heapster, but exposed to the programmer as part of a type system. Like rely-guarantee permissions, types for references carry a predicate (akin to our precondition), and rely and guarantee relations. This system harnesses much of the power of rely-guarantee reasoning, allowing programmers to control aliasing and to verify fine-grained concurrent code, at the cost of additional complexity for users. Gordon, Ernst, and Grossman prove soundness in two orthogonal ways: first, a syntactic approach using progress and preservation, and second, an embedding into Views [Din+13], a predecessor of the Iris framework. We aim to hide the complexity of using rely-guarantee by using it only to "implement" a simpler interface without the possible complexity of unrestricted rely and guarantee relations.

Militão, Aldrich, and Caires [MAC14] present a similar type system to that of Gordon, Ernst, and Grossman where aliasing is controlled using rely-guarantee relations. Like our work, they support a temporal splitting of resources. Soundness of their type system is proven using a syntactic approach, using progress and preservation.

RefinedC [Sam+21] uses a type system that combines ownership types with refinement types to express both ownership information and invariants on data types for C code. Since it verifies programs directly, these types act like specifications, describing the behavior of these C functions in detail using the refinement types. Once types are provided, RefinedC automatically uses the typechecking process to verify programs against their type specifications, resulting in a fully verified proof in Coq. Crucially, the type system is proved sound via a semantic typing approach, where typing rules are proved as lemmas in Iris. The design of the toolchain ensures that the input to typechecking is in a fragment of Iris where proof search is possible without backtracking. The typechecking process then applies this search process, directly applying the typing rules that were proven in Coq, ensuring the entire process is verified.

One difference with Heapster is that since Heapster extracts specifications instead of verifying the code directly, Heapster types are far less complex than those of RefinedC. Another major difference is that in our work, the verification tool, Heapster, is not formally connected to the proofs presented in this dissertation. That is, if we were to take RefinedC's approach, then the theory

of Heapster presented in this dissertation would *be* the verification tool, rather than the separate Heapster tool. As mentioned in Section 2.3, this is a possible future direction for our work, where we can directly use the typing rules to typecheck programs represented by ITrees, or to build more sophisticated tooling in Coq, as RefineC does, to handle real code.

CHAPTER 4

Rely-Guarantee Permissions

Part of this chapter is adapted from work previously published as Paul He, Eddy Westbrook, Brent Carmer, Chris Phifer, Valentin Robert, Karl Smeltzer, Andrei Ştefănescu, Aaron Tomb, Adam Wick, Matthew Yacavone, and Steve Zdancewic. "A Type System for Extracting Functional Specifications from Memory-Safe Imperative Programs." In: *Proc. ACM Program. Lang.* 5.OOPSLA (Oct. 2021) [He+21]. I was the primary author and the primary contributor of the theoretical component of the paper, the focus of this dissertation.

Rely-guarantee permissions are the central concept of this dissertation that will be used to build up a semantic description of the types of the Heapster type system. This chapter will introduce the definition of rely-guarantee permissions and some important results and related definitions.

At its core, a rely-guarantee permission represents a description of how a program is permitted to behave. In this dissertation, a program is thought of as *holding* a permission, which can evolve throughout execution of the program. In the literature, the term *permission* is sometimes used to refer specifically to Boyland's fractional permissions [Boy03], but we use it as the broader concept of controlling the behavior of programs, sometimes called a *capability*. This concept has been used in a wide variety of settings, and an overview of some of these works can be found in the survey paper by Sadiq, Li, and Ling [SLL20].

Figure 4.1 shows an example of a program that allocates a new variable then checks whether the allocation succeeded. The gray boxes show an example of the permissions that could be held at each point in the program in order to validate the program as a whole. First, the program would need a permission that allows it to allocate memory in order to perform the allocation. After allocation, we would obtain a permission that represents the ability to write to `x`, as long as the allocation was successful. Depending on the result of the check, we then either get a permission allowing write access to `x`, or no new permission at all in the case that the allocation was unsuccessful.

```
Allocation permission
alloc x;
Allocation permission + new variable permission for x
if (x) {
Allocation permission + write permission for x
  x := 1;
Allocation permission + write permission for x
} else {
Allocation permission
}
```

Figure 4.1: Example permissions for a program that allocates new memory.

These are not the only possible permissions for this program. For example, since we do not allocate more memory after the first allocation, it would be possible to drop the allocation permission after the allocation. After performing the write to `x`, we could also drop its write permission.

As we will see in this chapter, rely-guarantee permissions describing different portions of a program can be composed into one larger one for the entire program. In this chapter we will introduce some of the intuitions we use for using permissions with programs. However, the formal connection between permissions and programs will only be defined in Chapter 5.

4.1. The Rely-Guarantee Permission Lattice

Definition 4.1 (Rely-guarantee permission)**.** Given state type S, we define a *rely-guarantee permission over S* as a tuple $\pi = (R, G, P)$ where

1. R and G are preorders on S,
2. P is a predicate on S,
3. P is *stable* under R, i.e. if $(x, y) \in R$ and $x \in P$ then $y \in P$.

These definitions follow closely from the rules of rely-guarantee reasoning, as described in Section 2.1.

The *guarantee* G specifies what state changes this permission allows the holder to perform. The

rely preorder R specifies the state changes permitted by other permissions that this permission can tolerate. This is useful for determining what permissions can be held at the same time, like how rely-guarantee reasoning is used to determine which threads can be run concurrently. The R and G relations are both preorders. The reflexivity of the rely means that a permission always tolerates code that makes no changes. For the guarantee, it means that a program is always allowed to make no changes. They also should both be transitive, meaning that tolerated changes and allowed changes can be combined sequentially. These two names should be interpreted from the perspective of the permission holder. Changes to the state that others make can be *relied upon* to stay within R, and changes that we make are *guaranteed* to stay within G.

The final component P is a *precondition* that describes the assumptions this permission makes about the state. If a program holds a permission, then its precondition should be true as long as the permission continues to be held. Item 3 in the definition means that any such assumptions cannot be violated by changes tolerated by this permission.

In the case of rely-guarantee reasoning, the setting was concurrent programs and "other code" referred to other threads. For rely-guarantee permissions, the intuitions are slightly different. While rely-guarantee reasoning uses rely and guarantee relations to specify the interference of a thread, interference can be generalized to sequential code as well. In fact, one of the earliest uses of the term *interference* was by Reynolds [Rey78], who used it to refer to any code that can have *global* effects, such as aliasing, in the context of general imperative programs. The theory of Heapster uses rely-guarantee permissions in this way to specify the interference of permissions, and permissions describe how programs, or parts of programs, can behave. As we will see in Section 4.2, permissions describing parts of programs can be combined, and this will also relate to how threads are combined in rely-guarantee reasoning.

As an example of a permission, suppose the state only contains a single integer. Then we can define a rely-guarantee permission over $\mathbb{Z}$ that permits this number to increase, while taking an

argument b, a lower bound of the value in the state:

$$\pi_{\uparrow}(b) \stackrel{\mathsf{def}}{=} (\leq, \leq, \{ n \mid b \leq n \})$$

The rely above says that others are allowed to increase the state value, and the guarantee says that we—as the permission holder—are able to as well. The precondition says that we expect to start in a state where b is indeed a lower bound. Additionally, it is stable under the rely—any increases to the state would not change the fact that b is a lower bound. The rely of $\pi_{\uparrow}(b)$ is the weakest possible one for this stability requirement to be true. We could strengthen the rely (for example to $=$) and retain stability, but this would just restrict the changes that this permission can tolerate without any upside.

We use the notations R_π, G_π, and P_π to denote the respective components of permission π. When given a collection of permissions $\pi_1, \pi_2, \ldots$, we will overload this notation and use R_i, G_i, and P_i to refer to components of the i-th permission. We will also write perm_S as the set of all permissions over a state type S.

A crucial operation is to order permissions:

Definition 4.2 (Permission ordering)**.** Given permissions π_1 and π_2 over S, we define the *permission ordering* $\pi_1 \sqsubseteq \pi_2$ to hold if $R_1 \subseteq R_2$, $G_1 \supseteq G_2$, and $P_1 \subseteq P_2$.

Intuitively, a smaller permission is more powerful. Its guarantee is bigger and more permissive, allowing for more state updates than a larger permission. This greater power also allows it to make more "demands"—the smaller permission has a smaller, stricter precondition and rely.

For example, if we had another permission $\pi_{\updownarrow}(b)$ similar to $\pi_{\uparrow}(b)$ but that also allowed decreasing the number in the state, then its guarantee would be $\mathbb{Z} \times \mathbb{Z}$ and its other components would be the same as $\pi_{\uparrow}(b)$:

$$\pi_{\updownarrow}(b) \stackrel{\mathsf{def}}{=} (\leq, \mathbb{Z} \times \mathbb{Z}, \{ n \mid b \leq n \})$$

This permission is smaller than $\pi_{\uparrow}(b)$ since it has a larger guarantee. However, the use of the argument b as a lower bound is slightly different with this permission. Since its guarantee allows the state to either increase or decrease, b may no longer be a lower bound after a state update, unlike the stronger guarantee of $\pi_{\uparrow}(b)$.

As another example, consider the permission $\pi_{EX\updownarrow}(b)$ which has the same guarantee as $\pi_{\updownarrow}(b)$, but has a stronger rely of $=$. That is, nobody else is allowed to change the state, allowing this permission to represent *exclusive* access to it. We can also strengthen the precondition to $\{ n \mid b = n \}$, which allows the argument b to represent the actual value of the state, rather than just a lower bound. Note that this was not possible with the previous $\pi_{\uparrow}$ and $\pi_{\updownarrow}$ permissions, since their relies would not have been stable over this precondition.

$$\pi_{EX\updownarrow}(b) \stackrel{\text{def}}{=} (=, \mathbb{Z} \times \mathbb{Z}, \{ n \mid b = n \})$$

Putting these examples together, we have $\pi_{EX\updownarrow}(b) \sqsubseteq \pi_{\updownarrow}(b) \sqsubseteq \pi_{\uparrow}(b)$. Permissions will often change throughout computation. One way they can change—though not the only way—is through weakening, transforming stronger permissions into weaker ones via $\sqsubseteq$. For example, if we held a $\pi_{\updownarrow}(b)$ permission but did not need the ability to decrease the state, we could give up that ability by weakening it into a $\pi_{\uparrow}(b)$ permission.

The $\pi_{\updownarrow}(b)$ and $\pi_{EX\updownarrow}(b)$ permissions also show the reason why the precondition is not required to be stable under the guarantee—the precondition represents the *current* assumptions on the state, and can change as the state changes. In this case, the guarantees allow the value of the state to change, which may change whether that value still satisfies the precondition.

Naturally, we can show that $\sqsubseteq$ is a preorder:

Lemma 4.1. $\sqsubseteq$ is reflexive and transitive.

Further, with this ordering on permissions, $(\mathsf{perm}_S, \sqsubseteq)$ forms a complete lattice.

The meet is defined as

$$\pi_1 \sqcap \pi_2 \stackrel{\text{def}}{=} (R_1 \cap R_2, (G_1 \cup G_2)^*, P_1 \cap P_2),$$

with the usual properties of a meet:

Lemma 4.2.

- The meet is a lower bound: $\pi_1 \sqcap \pi_2 \sqsubseteq \pi_1$ and $\pi_1 \sqcap \pi_2 \sqsubseteq \pi_2$.
- The meet is the *greatest* lower bound: for any π where $\pi \sqsubseteq \pi_1$ and $\pi \sqsubseteq \pi_2$, $\pi \sqsubseteq \pi_1 \sqcap \pi_2$.

The meet of permissions π_1 and π_2 represents the stronger permission that is the conjunction of both. In its guarantee, it can do anything that either of π_1 and π_2 can do. To ensure that it is still a preorder, we must use the reflexive and transitive closure. Its rely and preorder are more strict than either of its parts—requiring that both the relies and preorders of π_1 and π_2 hold.

The dual operation of meet is the join. It is defined as

$$\pi_1 \sqcup \pi_2 \stackrel{\text{def}}{=} ((R_1 \cup R_2)^*, G_1 \cap G_2, P_1 \cup P_2 \cup \{\, y \mid \exists x, (P_1(x) \vee P_2(x)) \wedge (R_1 \cup R_2)^*(x, y) \,\})$$

and also has the usual properties of a join:

Lemma 4.3.

- The join is an upper bound: $\pi_1 \sqsubseteq \pi_1 \sqcup \pi_2$ and $\pi_2 \sqsubseteq \pi_1 \sqcup \pi_2$.
- The join is the *least* upper bound: if $\pi_1 \sqsubseteq \pi$ and $\pi_2 \sqsubseteq \pi$, then $\pi_1 \sqcup \pi_2 \sqsubseteq \pi$.

The join of permissions π_1 and π_2 represents a weaker and more restricted permission. Its guarantee only allows for updates that are allowed by the guarantees of both π_1 and π_2. Its rely tolerates any changes that either π_1 or π_2 can tolerate, closed under reflexivity and transitivity. Its precondition is similar, holding as long as the preconditions of either π_1 or π_2 hold. To satisfy the requirement that it is stable under the rely, the precondition also holds on states that start from one of the preconditions of π_1 and π_2, and then are updated in accordance to the rely.

We presented these operations as binary operators, but they could also be generalized to operations over any finite set of permissions once we determine the top and bottom elements. These represent the meet and join, respectively, of the empty set. The top element is

$$\pi_\top \stackrel{\text{def}}{=} (S \times S, =, S),$$

and the bottom element is

$$\pi_\perp \stackrel{\text{def}}{=} (=, S \times S, \emptyset).$$

The top element $\pi_\top$ is the weakest permission, permitting no changes to be made to the state using its guarantee. However, its rely can tolerate any change, and its precondition also always holds. This leads to the intuition that $\pi_\top$ is inert and harmless, equivalent to there being no permission at all. Dually, $\pi_\perp$ is the most powerful permission, permitting everything using its guarantee, which contains all possible state updates. However, its rely and precondition are maximally restrictive. The rely permits no changes, and the precondition never holds, so this permission would not be usable in any actual state.

4.2. Coexistence

As hinted at earlier, not all permissions can coexist, and whether they can depends on the rely and guarantee components of the permissions. We define this notion following from the Par rule of rely-guarantee reasoning, which gave a condition for when threads can safely coexist.

Definition 4.3 (Separate permissions)**.** Given permissions π_1 and π_2 over S, we say they are *separate*, written $\pi_1 \perp \pi_2$, when $G_1 \subseteq R_2$ and $G_2 \subseteq R_1$.

Two permissions are separate if the possible updates of each are tolerated by the other. While separateness does not explicitly use the preconditions of these permissions, the stability requirement for permissions ensures that updates allowed by one permission does not interfere with the precondition of the other. For example, it is true that $\pi_\uparrow(b) \perp \pi_\uparrow(b')$, meaning that these permissions can coexist, and any changes they permit would maintain b and b' as lower bounds for

the current state. On the other hand, $\pi_{\uparrow}(b) \not\perp \pi_{EX\updownarrow}(b')$ since the changes allowed by the $\pi_{\uparrow}(b)$ permission would violate the rely of $\pi_{EX\updownarrow}(b)$. If the code holding $\pi_{\uparrow}(b)$ did increase the state, this would also break the precondition of $\pi_{EX\updownarrow}(b)$, that b is the current value of the state. This validates our intuition that $\pi_{EX\updownarrow}$ represents exclusive access, so it should not be able to be combined with another permission which allows write access to the state. It is not separate from any permission that allows the state to change, effectively ensuring that it can be the only permission with such a guarantee.

As another validator of our intuitions, we can prove that the $\pi_{\top}$, the inert, weakest permission, can coexist with any other permission:

Lemma 4.4. $\pi_{\top} \perp \pi$ for any π.

We stated earlier that permissions should be able to change during execution by weakening. One crucial property of separateness that relates to this is that separateness is upward-closed:

Theorem 4.5. If $\pi_1 \perp \pi_2$, $\pi_1 \sqsubseteq \pi_1'$, and $\pi_2 \sqsubseteq \pi_2'$, then $\pi_1' \perp \pi_2'$.

This theorem says that if we have two permissions that can already coexist, this coexistence is maintained if we substitute either permission with a larger one. This supports the idea that larger permissions are "weaker", and that replacing permissions with weaker ones during execution should be permitted. A larger permission, which cannot do as much (via its guarantee) and can tolerate more (via its rely) can always coexist with existing permissions.

Using separateness, we can now define the separating conjunction of separation logic to combine permissions:

Definition 4.4 (Separating conjunction)**.** Given permissions π_1 and π_2 over S, we define the *separating conjunction* of π_1 and π_2 by:

$$\pi_1 * \pi_2 \stackrel{\text{def}}{=} (R_1 \cap R_2, (G_1 \cup G_2)^*, P)$$

where $P = \begin{cases} P_1 \cap P_2 & \text{if } \pi_1 \perp \pi_2 \\ \emptyset & \text{otherwise.} \end{cases}$

If the two permissions are separate, then $\pi_1 * \pi_2$ is exactly the same as $\pi_1 \sqcap \pi_2$, the conjunction of both permissions. If they are not separate, then $\pi_1 * \pi_2$ has an empty precondition, resulting in the permission being unusable for any actual code. We will always use $*$ to combine permissions, rather than the meet, in order to ensure that permissions are compatible.

This operation is analogous to combining two threads using the PAR rule from rely-guarantee reasoning. Whereas rely-guarantee reasoning uses the interference described by the rely and guarantee relations to determine when programs are safe to compose using parallel composition, rely-guarantee permissions use interference to determine when *permissions*, describing the capabilities of fragments of programs, are safe to compose. This notion of composition is not limited to parallel composition, and is more general. As guarantees are combined using the reflexive-transitive closure of the union of both guarantees, this permits any interleaving of state changes allowed by either permission. This supports sequential composition as well as parallel composition. While Heapster only supports sequential code, handling concurrency is a possible future direction, as we will discuss in Section 8.1.

We can prove a few basic properties of the separating conjunction:

Lemma 4.6. The ordering $\sqsubseteq$ induces an equivalence relation, which we will write as $\equiv$ and is defined as $\pi_1 \equiv \pi_2$ if $\pi_1 \sqsubseteq \pi_2$ and $\pi_2 \sqsubseteq \pi_1$. Separating conjunction is associative and commutative with respect to $\equiv$:

- $(\pi_1 * \pi_2) * \pi_3 \equiv \pi_1 * (\pi_2 * \pi_3)$.
- $\pi_1 * \pi_2 \equiv \pi_2 * \pi_1$.

The following monotonicity result is important to allow parts of the overall permission to change independently.

Lemma 4.7. If $\pi_1 \sqsubseteq \pi_1'$ and $\pi_2 \sqsubseteq \pi_2'$ then $\pi_1 * \pi_2 \sqsubseteq \pi_1' * \pi_2'$.

The proof of the monotonicity of $*$ relies crucially on the fact that separateness is upward-closed.

Finally, we can show that $*$ always results in a smaller permission.

Lemma 4.8. $\pi * \pi' \sqsubseteq \pi$.

While this property of $*$ is easy to prove due to its similarity to the meet, it shows that conjoining a new permission onto an existing one always results in a smaller overall permission. Since permissions can always be weakened, this property also implies that rely-guarantee permissions are used to model an *affine* separation logic, where assertions can be dropped. This is in contrast to a linear logic, where discarding permissions is disallowed. The affine logic has the benefit of allowing us to model garbage collected languages, rather than just languages where memory must be explicitly deallocated [Cha24].

4.3. Permission Changes

While weakening permissions throughout execution using $\sqsubseteq$ is useful, it is not always sufficient for updating permissions. A bigger permission has a bigger precondition, meaning that it is a weaker assumption about the state. However, the state can change arbitrarily during computation, which is not captured by this ordering. For example, if we held the permission $\pi_{EX\updownarrow}(0)$ then updated the state to 1, we must change the permission we hold to $\pi_{EX\updownarrow}(1)$ so its precondition matches the new value of the state. This precondition change from $\{0\}$ to $\{1\}$ is not permitted by $\sqsubseteq$, so we will need something different.

When updating permissions, a permission for part of the code should not affect other permissions when it changes, and keep all separateness properties valid. We can try to use this this idea of not affecting other permissions as the *definition* of how permissions can change:

Definition 4.5. Permission π_2 *is at least as separate as* π_1, written $\pi_1 \rightsquigarrow \pi_2$, if $\pi_1 \perp \pi$ implies $\pi_2 \perp \pi$ for all π.

Naturally, since this definition is used to describe how permissions can change, no change is always permitted and changes can be combined:

Lemma 4.9. $\rightsquigarrow$ is reflexive and transitive.

The following theorem about $\rightsquigarrow$ show that it is quite similar to $\sqsubseteq$, except that $\rightsquigarrow$ does not impose any ordering on the preconditions of permissions.

Theorem 4.10. $\pi_1 \rightsquigarrow \pi_2$ if and only if $R_1 \subseteq R_2$ and $G_1 \supseteq G_2$.

As a corollary, $\pi_1 \sqsubseteq \pi_2$ implies $\pi_1 \rightsquigarrow \pi_2$, so $\rightsquigarrow$ generalizes our earlier notion of how permissions can change. Using this theorem, we can show that $\pi_{EX\updownarrow}(0) \rightsquigarrow \pi_{EX\updownarrow}(1)$, since their relies and guarantees are the same.

The following lemma also shows that we can change only part of a larger permission formed by $*$, similarly to how monotonicity for $*$ was used:

Lemma 4.11. If $\pi_1 \rightsquigarrow \pi_2$ and $\pi' \perp \pi_1$, then $\pi_1 * \pi' \rightsquigarrow \pi_2 * \pi'$.

Chapter 5 will formally present how permissions can change through execution and how $\rightsquigarrow$ is used.

4.4. Permission Sets

While $*$ allows us to represent conjunction, a major issue with permissions as we have defined them so far is how to represent disjunction. Disjunction is crucial for modeling permissions with multiple possibilities like possibly-null pointers, which are then refined through control flow in the program. Existential quantification can be considered a form of infinitary disjunction, and is also very desirable for our purposes. For example, suppose we want to use $\pi_\uparrow$ but do not yet have any information about the state. Then no single value of b for $\pi_\uparrow(b)$ will be appropriate for all possible states, since it may not be a lower bound for the current value. Existential quantification to allow a permission to be defined as $\exists_{b\in\mathbb{Z}}\pi_\uparrow(b)$ would solve this issue.

While using a meet-like operation such as $*$ for the conjunction of permissions works, no anal-

ogous operation works for disjunction. Using the dual operation to meet, join—or something similar— does not work for our purposes. We wish to reobtain one of many possible disjunctive possibilities later, which is not possible with an operation like the join, which is "lossy". By intersecting the guarantees as is done by the join, we lose information and will be unable to revert to one of the permissions that make up the disjunction.

For example, if we represent the disjunction of $\pi_{\uparrow}(b)$ and $\pi_{\updownarrow}(b)$ as $\pi_{\uparrow}(b) \sqcup \pi_{\updownarrow}(b)$, the guarantee of this permission would be $\leq$, since the join takes the intersection of the two guarantees. If we later wish to reobtain $\pi_{\updownarrow}(b)$, we are unable to as we have "lost" the guarantee of $\pi_{\updownarrow}(b)$. Trying to change the permission using $\rightsquigarrow$ only allows guarantees to become smaller, so we cannot go from $\leq$ to the larger guarantee of $\pi_{\updownarrow}(b)$, which is $\mathbb{Z} \times \mathbb{Z}$. For this notion of disjunction, we need something different.

Our solution is to represent disjunctions as sets of permissions, which represent the set of possible permissions that might be currently held. In place of using a single rely-guarantee permission, we will instead use sets of permissions.

Definition 4.6 (Permission sets)**.** A *permission set over S* is a downward-closed set of permissions, that is, a subset $\Pi \subseteq \mathsf{perm}_S$ such that $\pi_2 \in \Pi$ and $\pi_1 \sqsubseteq \pi_2$ implies $\pi_1 \in \Pi$. We write Perms_S for the set of all permission sets over S. We overload the notation to define the *permission set ordering* $\Pi_1 \sqsubseteq \Pi_2$ as $\Pi_1 \subseteq \Pi_2$.

Like with permissions, $(\mathsf{Perms}_S, \sqsubseteq)$ forms a complete lattice, where we write $\mathsf{False} = \emptyset$ for the least element and $\mathsf{True} = \mathsf{perm}_S$ for the greatest element. Join and meet are defined as union and intersection, respectively, and are written as $\sqcup$ and $\sqcap$.

For convenience we use the notation $[\pi]$ to represent the smallest permission set containing π. Such a permission set represents a single permission, even though there are other permissions in the set. The largest element (or elements) of the set can be seen as the permission (or permissions) that the set represents.

$$[\pi] \stackrel{\mathsf{def}}{=} \{ \pi' \mid \pi' \sqsubseteq \pi \}$$

With permission sets, we can now use the join to handle the case from earlier, where no specific b was appropriate for $\pi_{\uparrow}(b)$. We can use a permission set $\Pi_{\uparrow}$ to represent this instead. First, let's define a permission set for when we do know a lower bound b:

$$\Pi_{\uparrow}(b) \stackrel{\text{def}}{=} [\pi_{\uparrow}(b)] = \{ \pi \mid \pi_{\uparrow}(b) \sqsubseteq \pi \}.$$

Then the join serves as existential quantification:

$$\Pi_{\uparrow} \stackrel{\text{def}}{=} \bigsqcup_{b \in \mathbb{Z}} \Pi_{\uparrow}(b) = \bigsqcup_{b \in \mathbb{Z}} [\pi_{\uparrow}(b)] = \{ \pi \mid \pi \sqsubseteq \pi_{\uparrow}(b) \text{ for some } b \in \mathbb{Z} \}.$$

This set contains all the possible lower bounds on the value, as well as all the smaller, stronger permissions, and with one of them being "actually" held by the user. Similarly, we can define the analogous permission sets for $\pi_{\updownarrow}$ and $\pi_{EX\updownarrow}$:

$$\Pi_{\updownarrow} = \bigsqcup_{b \in \mathbb{Z}} \Pi_{\updownarrow}(b) = \bigsqcup_{b \in \mathbb{Z}} [\pi_{\updownarrow}(b)]$$

$$\Pi_{EX\updownarrow} = \bigsqcup_{b \in \mathbb{Z}} \Pi_{EX\updownarrow}(b) = \bigsqcup_{b \in \mathbb{Z}} [\pi_{EX\updownarrow}(b)]$$

From the previous result that $\pi_{EX\updownarrow}(b) \sqsubseteq \pi_{\updownarrow}(b) \sqsubseteq \pi_{\uparrow}(b)$ and the downward-closure property of permission sets, this result lifts to the permission set level:

$$\Pi_{EX\updownarrow} \sqsubseteq \Pi_{\updownarrow} \sqsubseteq \Pi_{\uparrow}$$

During typechecking, we often wish to transform permission sets, and the way we do this is by changing them to a weaker permission set using $\sqsubseteq$, just like with permissions. With this intuition, the naming of our top and bottom elements of the lattice become more clear. In our system, some permission is always held, where $\pi_{\top}$ would be used to represent holding a permission that represents "nothing", or the capability to do nothing. The bottom element False is then the inconsistent or contradictory permission set that is the strongest and implies all others, because it

does not represent holding *any* permission at all, which is not allowed. The top element True is the vacuously true permission set that is entailed by all others. Since $\pi_\top \in \mathsf{True}$, the option that this is the "actual" permission being held must be considered, so the set as a whole is effectively equivalent to holding $\pi_\top$, the weakest permission.

The odd-looking definition of the permission set involving downward-closure is similar to the Hoare power domain [AJ94] used for the denotational semantics of nondeterministic programs. The Hoare power domain extends the underlying domain for deterministic programs to subsets of that domain. The set then contains all the possible nondeterministic computations. The ordering of these sets is then defined as:

$$X \sqsubseteq Y \stackrel{\mathsf{def}}{=} \forall x \in X, \exists y \in Y, x \sqsubseteq y$$

Crucially, using this definition, a set is equivalent to its downward-closure, and ordering is equivalent to set inclusion, just like our definitions for permission sets.

In contrast to Smyth and Plotkin power domains [AJ94], the Hoare power domain models *angelic* nondeterminism. Angelic nondeterminism makes nondeterministic choices in favor of termination. With the Hoare power domain, this is represented by the fact that elements of the set higher up in the lattice of programs can be thought of as the representative elements of the set. For instance, the diverging program $\perp$ is part of every nonempty downward-closed set and is essentially ignored, emphasizing the termination properties of the other elements in the set. This intuition matches our intentions for permission sets. Just as how the Hoare power domain ensures that termination dominates, permission sets ensure that weaker permissions dominate, as weaker permissions are higher in the lattice.

In the first paper on presenting rely-guarantee permissions [He+21], the rely-guarantee permission and permission set ordering were reversed from what is presented here, and permission sets were defined as upward-closed. In that setting, this explanation in terms of power domains would have corresponded to the Smyth power domain, which models demonic nondeterminism

rather than the angelic nondeterminism of the Hoare power domain. Ultimately, that explanation is equivalent to the one above, since the permission lattice is flipped. The Smyth power domain can be thought of as biasing towards programs lower in the lattice, those closer to the diverging program, $\bot$. In the setting of that paper, a permission set biases towards permissions lower in the lattice, which are the weaker permissions. This is equivalent to the current setting where we also bias towards weaker permissions, those higher in the lattice.

Finally, we lift the definition of $*$ to permission sets:

Definition 4.7 (Separating conjunction for permission sets)**.** For permission sets Π_1 and Π_2, we define the *separating conjunction*

$$\Pi_1 * \Pi_2 \stackrel{\text{def}}{=} \{\, \pi \mid \exists \pi_1 \in \Pi_1, \exists \pi_2 \in \Pi_2, \pi_1 \perp \pi_2 \wedge \pi \sqsubseteq \pi_1 * \pi_2 \,\}.$$

We can then lift the results we had for separating conjunction at the permission level:

Lemma 4.12. Separating conjunction (at the permission set level) is associative and commutative with respect to $\equiv$, where $\Pi_1 \equiv \Pi_2$ if $\Pi_1 \sqsubseteq \Pi_2$ and $\Pi_2 \sqsubseteq \Pi_1$. The notation is again overloaded with the equivalence for rely-guarantee permissions.

Lemma 4.13. If $\Pi_1 \sqsubseteq \Pi_1'$ and $\Pi_2 \sqsubseteq \Pi_2'$ then $\Pi_1 * \Pi_2 \sqsubseteq \Pi_1' * \Pi_2'$.

Lemma 4.14. $\Pi * \Pi' \sqsubseteq \Pi$.

Finally, we can link $*$ at the rely-guarantee permission level to $*$ at the permission set level:

Lemma 4.15. If $\pi_1 \in \Pi_1$, $\pi_2 \in \Pi_2$, and $\pi_1 \perp \pi_2$, then $\pi_1 * \pi_2 \in \Pi_1 * \Pi_2$.

Unlike the "user-facing" definitions like $*$ and $\sqsubseteq$, which need to be translated to permission sets, $\rightsquigarrow$ will not be. As we will see in the next chapter, changing permissions using $\rightsquigarrow$ is used only within the definition of typing, and will need to operate only on individual rely-guarantee permissions.

CHAPTER 5

A Simple Separation-Logic Type System

Part of this chapter is adapted from work previously published as Paul He, Eddy Westbrook, Brent Carmer, Chris Phifer, Valentin Robert, Karl Smeltzer, Andrei Ştefănescu, Aaron Tomb, Adam Wick, Matthew Yacavone, and Steve Zdancewic. "A Type System for Extracting Functional Specifications from Memory-Safe Imperative Programs." In: *Proc. ACM Program. Lang.* 5.OOPSLA (Oct. 2021) [He+21]. I was the primary author and the primary contributor of the theoretical component of the paper, the focus of this dissertation.

This chapter will present a separation-logic-based type system to formally connect rely-guarantee permissions to programs, in order to formalize the intuitions we introduced in Chapter 4. Furthermore, this type system will act as a bridge to the more complex Heapster type system [He+21]—involving both imperative programs and functional specifications—defined in the next chapter.

There is no technical difference between this type system and a separation logic, but we think of Heapster as implementing a type system, so we also use this framing here. Heapster types are typically quite coarse, not specifying values in memory but rather properties typical of most type systems, like a value being an integer or a pointer.

We will approach the definitions and soundness of the type system *semantically* [Mil78]. Using this approach, rather than defining the syntactic typing rules first, we will instead first define the semantic meaning of the typing judgment and types—using rely-guarantee permissions—then define and prove typing rules one by one as theorems. If we view the type system as a separation logic, a semantic approach for soundness is the default one, as in the original heap semantics for separation logic [Rey02].

Following the examples of the previous chapter, we start with the case where the state is a single integer. After presenting the intuitions and definitions in this simplified case, we then expand to a more standard heap and introduce some more familiar rules for pointers from separation logic.

5.1. Defining Typing

We will aim to develop a typing relation $\Pi \vdash t : \lambda r.\, \Pi'$ which can be thought of as equivalent to a separation logic specification $\{\Pi\}t\{\lambda r.\, \Pi'\}$, where r, the value of t once it finishes, is free in Π'.

The program t is an ITree, and Π is the permission set held before execution of t, which we will call the input permission. The specific state type for rely-guarantee permissions and the event type for ITrees will be defined later on. The output permission, $\lambda r.\, \Pi'$, is a function that produces the permission set held after t is executed. This permission set depends on a value r of type R, the return value of the program t. In our definitions, these functions returning permission sets will be named T, since the output permission can be thought of as the type of the program. To further motivate the notation we use, the input permission can be thought of as the context—the set of permissions held before the program runs.

There are two main goals of this definition of typing. First, during the typechecking of t, permissions must change to reflect what is currently true about the state and what the program is currently permitted to do, which can change as the program executes. As stated in Chapter 4, this will be done using $\rightsquigarrow$. Secondly, typing should ensure that everything the program does is allowed by the guarantee of the held permission. The guarantee is a relation on states, but how do we relate that to programs, represented as ITrees?

The first step to represent state changes is to use events in the ITree to explicitly signal uses of the state. Our event type, E, will be inhabited by these two events:

$$\mathsf{Load} : \mathsf{E}\ \mathbb{Z}$$

$$\mathsf{Store} : \mathbb{Z} \rightarrow \mathsf{E}\ \mathsf{Unit}$$

As discussed in Section 2.3, ITrees can already encode control flow constructs like loops and conditionals, so these events allow us to model a simple imperative language with a single integer variable. In this section, we will continue with this simple integer state so we can continue with the previous rely-guarantee permission examples in Chapter 4. In Section 5.3, we will change

the state to model a heap, so we can work with a more realistic imperative language and define permissions for pointers.

Next, to be able to talk about state changes and when states satisfy permissions' preconditions, we pair each ITree with a state value and define how they evolve together.

Definition 5.1. We define the *steps-to* relation $(s_1, t_1) \to (s_2, t_2)$ on pairs of a state of type $\mathbb{Z}$ and an ITree of type $\mathsf{itree}\ \mathsf{E}\ R$ by the following cases:

$$(s, \tau\ t) \to (s, t) \qquad (s, \mathsf{vis}\ \mathsf{Load}\ k) \to (s, k\ s) \qquad (s, \mathsf{vis}\ (\mathsf{Store}\ s')\ k) \to (s', k\ \mathsf{unit})$$

A program that is a ret does not step, since it is finished.

Now that we have a way of determining how states can change in a program, we can define the typing judgment:

Definition 5.2 (Typing)**.** Let Π be a permission set over $\mathbb{Z}$, t be an $\mathsf{itree}\ \mathsf{E}\ R$, and T be a function $T : R \to \mathsf{Perms}_{\mathbb{Z}}$. Then the semantic typing judgment is defined as the largest relation[3] $\Pi \vdash t : T$ that satisfies the following two cases:

1. $\Pi \vdash \mathsf{ret}\ v : T$ if $\Pi \sqsubseteq T\ v$.

2. $\Pi \vdash t : T$ if:

    - t can step, i.e., there exist s, t', and s' such that $(s, t) \to (s', t')$;

    - for any $\pi \in \Pi$ and $s \in P_\pi$, and any new program state we can step to, i.e., for any s' and t' such that $(s, t) \to (s', t')$,

        – the state update is allowed by the permission π's guarantee: $(s, s') \in G_\pi$.

        – the new program is well-typed for a permission that is at least as separate as the

[3] Defined as a coinductive relation in Coq.

previous permission: there exist Π' and π' where $\pi' \in \Pi'$ and $\Pi' \vdash t' : T$, and $\pi \rightsquigarrow \pi'$ and $s' \in P_{\pi'}$.

The first case says that if the program is finished, then the output permission set must be a weakening of the input permission set. As we discussed in Chapter 4, weakening permissions should always be permitted.

The second case says that if the program is not finished, then it must be able to step. We then consider all possible permissions π we might be holding, formalizing the intuition from Chapter 4 that any of the rely-guarantee permissions in a permission set might be the one actually held. For each of these possibilities, for steps whose states start in the precondition of π, any state updates must be allowed by its guarantee. Additionally, the program we step to must be well-typed as well, though we can change the input permission set to suit the new state. We are permitted to select any new permission set, as long as it contains a permission that is at least as separate as π.

This definition also lets us characterize programs that are *not* well-typed. Programs that are neither finished (a ret) nor able to step are not well-typed. Of those programs that can step, if they step to any ill-typed program, then they are also not well-typed.

Let's walk through this definition on an example of typing a program that performs a store to update the state:

Example 5.1.

$$\frac{}{\Pi_{EX\updownarrow} \vdash \mathsf{vis}\ (\mathsf{Store}\ 1)\ \mathsf{ret} : \lambda_.\, \Pi_{EX\updownarrow}(1)}$$

We use _ to represent an ignored argument to the function.

This typing judgment for a store operation lets us refine a $\Pi_{EX\updownarrow}$ permission to a stronger one, $\Pi_{EX\updownarrow}(1)$, where we know the specific value of the state is 1. The program is not a ret, so we must use case two of Definition 5.2. The program is $\mathsf{vis}\ (\mathsf{Store}\ 1)\ \mathsf{ret}$, which can certainly step, for example by $(0, \mathsf{vis}\ (\mathsf{Store}\ 1)\ \mathsf{ret}) \to (1, \mathsf{ret}\ \mathsf{unit})$, so we can proceed.

Next, let π be any permission in $\Pi_{EX\updownarrow}$ and $s \in P_\pi$. This value s is the specific value of the state that was existentially quantified over by $\Pi_{EX\updownarrow}$. Then the step we take must be $(s, \mathsf{vis}\ (\mathsf{Store}\ 1)\ \mathsf{ret})$ $\rightarrow (1, \mathsf{ret}\ \mathsf{unit})$.

By the definition of $\Pi_{EX\updownarrow}$, it must be that $\pi \sqsubseteq \pi_{EX\updownarrow}(s)$. $(s, 1) \in G_{\pi_{EX\updownarrow}(s)}$ must hold since the guarantee of $\pi_{EX\updownarrow}(s)$ allows any change, and the guarantee of π must be larger than the guarantee of $\pi_{EX\updownarrow}(s)$. Thus $(s, 1) \in G_\pi$ must hold: the state update from s to 1 is permitted by π.

Next, we must make sure the program we step to, $\mathsf{ret}\ \mathsf{unit}$, is well-typed. Since we want to get to the final output permission of $\lambda_. \Pi_{EX\updownarrow}(1)$, this helps us choose the next input permission set to continue with. We choose the next input permission set to be $\Pi_{EX\updownarrow}(1)$, and choose $\pi_{EX\updownarrow}(1) \in \Pi_{EX\updownarrow}(1)$. It must be the case that $\pi \rightsquigarrow \pi_{EX\updownarrow}(1)$ since $\pi \rightsquigarrow \pi_{EX\updownarrow}(s)$ (because the inequality result $\pi \sqsubseteq \pi_{EX\updownarrow}(s)$ holds) and then $\pi_{EX\updownarrow}(s) \rightsquigarrow \pi_{EX\updownarrow}(1)$ by Theorem 4.10 (because they only differ in their preconditions). The final condition to check is that the precondition of $\pi_{EX\updownarrow}(1)$ holds on the new state, 1, which it does.

Finally, we need to show $\Pi_{EX\updownarrow}(1) \vdash \mathsf{ret}\ \mathsf{unit} : \lambda_. \Pi_{EX\updownarrow}(1)$. We use case one of Definition 5.2, and we are done.

5.2. General Typing Rules

Since our programs are represented as ITrees, which are defined coinductively, the typing relation is coinductive as well to handle diverging programs properly. For example, the diverging program that does nothing is well typed with any input and output permissions:

Lemma 5.2. For any Π and T,

$$\frac{}{\Pi \vdash \mathsf{spin} : T}$$

The rule of consequence is a crucial rule in Hoare logic and its descendants. The following rule allows us to strengthen the input permission and weaken the output permission of an existing typing judgment:

Theorem 5.3 (CNSQ)**.**

$$\frac{\Pi \sqsubseteq \Pi' \qquad T'\ r \sqsubseteq T\ r \text{ for any } r \qquad \Pi' \vdash t : T'}{\Pi \vdash t : T}$$

This rule implies that we can weaken any typing judgment to have the input permission False. In fact, False can type any program as long as that program is a ret or can step. Since there is no $\pi \in$ False, the second subcase of case two holds vacuously.

While we use $\rightsquigarrow$ to change permissions between steps of execution, we want to use $\sqsubseteq$ for when the program remains unchanged. This is because we do not want the precondition to change arbitrarily without a step of execution occurring. If we did attempt to prove a similar result with $\rightsquigarrow$ (assuming we had a version for permission sets), it would not hold, as typing requires the precondition to hold on the current state, which may be changed by a new permission that is at least as separate as the previous one.

The CNSQ rule formalizes the intuition that we can always weaken currently-held permissions during typechecking. For example, we can make use of the ordering result between $\Pi_{EX\updownarrow}$ and $\Pi_{EX\updownarrow}(1)$, which holds because $\Pi_{EX\updownarrow}$ is defined as a join of $\Pi_{EX\updownarrow}(b)$ for all b:

$$\Pi_{EX\updownarrow}(1) \sqsubseteq \Pi_{EX\updownarrow}$$

We can then use this result to weaken the output permission of our previous example, without having to redo the typechecking proof:

Example 5.4.

$$\frac{\Pi_{EX\updownarrow} \sqsubseteq \Pi_{EX\updownarrow} \qquad \Pi_{EX\updownarrow}(1) \sqsubseteq \Pi_{EX\updownarrow} \qquad \dfrac{}{\Pi_{EX\updownarrow} \vdash \mathsf{vis}\ (\mathsf{Store}\ 1)\ \mathsf{ret} : \lambda_.\ \Pi_{EX\updownarrow}(1)}\ \text{Ex. 5.1}}{\Pi_{EX\updownarrow} \vdash \mathsf{vis}\ (\mathsf{Store}\ 1)\ \mathsf{ret} : \lambda_.\ \Pi_{EX\updownarrow}}\ \text{CNSQ}$$

This can be useful if we do not need the specific value of the state, or if we want to compose this

with another typing judgment which expects the weaker, more general permission $\Pi_{EX\updownarrow}$.

The following rule allows us to perform such a sequential composition, combining two well-typed programs using the monadic bind:

Theorem 5.5 (BIND).

$$\frac{\Pi \vdash t : T' \qquad T'\ r \vdash k\ r : T \text{ for any } r}{\Pi \vdash x \leftarrow t; k\ x : T}$$

Using this rule, we can now *compositionally* typecheck the program that just repeats the earlier program of storing 1 twice. Example 5.4, where we use CNSQ, is essential to ensure that the intermediate permission set matches up between the two parts of the program in order to use BIND.

Example 5.6.

$$\dfrac{\dfrac{}{\Pi_{EX\updownarrow} \vdash \mathsf{vis}\ (\mathsf{Store}\ 1)\ \mathsf{ret} : \lambda_.\, \Pi_{EX\updownarrow}}\text{Ex. 5.4} \qquad \dfrac{}{\Pi_{EX\updownarrow} \vdash \mathsf{vis}\ (\mathsf{Store}\ 1)\ \mathsf{ret} : \lambda_.\, \Pi_{EX\updownarrow}(1)}\text{Ex. 5.1}}{\Pi_{EX\updownarrow} \vdash \mathsf{vis}\ (\mathsf{Store}\ 1)\ \mathsf{ret}; \mathsf{vis}\ (\mathsf{Store}\ 1)\ \mathsf{ret} : \lambda_.\, \Pi_{EX\updownarrow}(1)}\text{BIND}$$

Finally, we can prove the frame rule of separation logic:

Theorem 5.7 (FRAME).

$$\frac{\Pi \vdash t : T}{\Pi * \Pi' \vdash t : \lambda v.\, T\ v * \Pi'}$$

This rule allows typing rules to be minimal, only including the permissions they need to function. In the present setting with integer state, this rule is not very useful, but with a more interesting state that allows for different permissions on disjoint pieces, like in Section 5.3, this rule is crucial.

One thing to note is that we are permitted to combine permissions that are not separate using $*$. For example, we could use FRAME to add any permission set to our earlier example.

$$\Pi_{EX\updownarrow} * \Pi_{\uparrow} \vdash \mathsf{vis}\ (\mathsf{Store}\ 1)\ \mathsf{ret} : \lambda_.\, \Pi_{EX\updownarrow}(1) * \Pi_{\uparrow}$$

This permission set $\Pi_{EX\updownarrow} * \Pi_{\uparrow}$ ends up being equivalent to False, since its weakest elements, $\pi_{EX\updownarrow}(b)$ and $\pi_{\uparrow}(b')$, are not separate. As mentioned earlier, False types any program that can step, so this program *is* well-typed. When we present the type safety result for this system in the next section, we will only consider programs that are well-typed with "reasonable" input permissions. Permission sets like False, or sets containing permissions with false preconditions will not be considered reasonable.

5.3. Defining Memory Operations

Now that we understand the basic definitions and intuitions of typing, we can move to a more realistic state type and introduce more interesting permissions. We will define a heap for use in the state, and define pointer permissions, similar to points-to assertions in separation logic. The intuitions and the results from the previous section continue to hold, but some definitions will have to change and become more complex. With a definition of the heap, we also get the possibility of memory errors and the notion of soundness for the type system—memory safety.

First, to define the heap, we will use a simplified version of the CompCert memory model [Ler09]. In this memory model, a *memory* is a number of *blocks*, each of which can be allocated or unallocated. Each allocated block stores the size of the block and consists of a sequence of values in the block.

Concretely, a memory of type Mem is implemented as a list of blocks of memory, where a block represents a sequence of allocated memory cells. The choice of using a list is for convenience—it allows new blocks to be easily allocated by appending to the end of the list. A block consists of the size of the block and a partial function from natural number indices to imperative values. We can have addresses (b, o) representing a pointer to the value in the b-th block at an offset of o from the beginning of the block. We will write the type of these addresses as $\mathsf{addr} \stackrel{\mathsf{def}}{=} \mathbb{N} \times \mathbb{N}$. Values in the memory will have type Val, and can be either a VNum n, representing the natural number n,

Block #	Block
0	$\mathit{size} = 10$ $0 \mapsto \mathsf{VPtr}\ (0, 1)$ $1 \mapsto \mathsf{VNum}\ 2$ $\ldots$
1	$\mathit{size} = 1$ $0 \mapsto \mathsf{VNum}\ 2$
2	$\mathit{size} = 0$
$\ldots$	$\ldots$

Figure 5.1: An example of a heap.

or a $\mathsf{VPtr}\ a$, representing a pointer with address a.

$$\mathsf{VNum} : \mathbb{N} \to \mathsf{Val}$$

$$\mathsf{VPtr} : \mathsf{addr} \to \mathsf{Val}$$

Figure 5.1 shows a graphical example of a Mem. The pointer $\mathsf{VPtr}\ (0, 0)$ would point to the value in block 0 at offset 0, which is itself a pointer. The pointer value stored there would in turn refer to the VNum 2 value in the same block, at offset 1. For generality, we permit blocks to have size 0, like the one at block 2.

With this memory model in hand, we can then define functions $\mathsf{read} : \mathsf{Mem} \to \mathsf{addr} \to \mathsf{Option}\ \mathsf{Val}$ and $\mathsf{write} : \mathsf{Mem} \to \mathsf{addr} \to \mathsf{Val} \to \mathsf{Option}\ \mathsf{Mem}$, which read and write to a memory using a pointer value. These functions can fail and return None if the pointer we try to use does not point to valid memory. The definitions are straightforward and are omitted here.

Now that we have memory operations that can fail, we would also like to represent these failing computations formally. We define the state as a pair of a Mem and a boolean that represents whether a memory error has occurred. With this new error flag, we also overload the read and write functions to be able to operate on this state, leaving the error bit unchanged. We define error as the function that sets the error bit to true while leaving the Mem portion of the state unchanged. By adding the error status to the state, we can control whether memory-unsafe operations are

allowed using permissions.

Next, we must also modify the Load and Store ITree events, which previously just read or overwrote the single integer in the state.

$$\mathsf{Load} : \mathsf{Val} \to \mathsf{E}\ \mathsf{Val}$$

$$\mathsf{Store} : \mathsf{Val} \to \mathsf{Val} \to \mathsf{E}\ \mathsf{Unit}$$

Now, these events specify the memory location they access using a Val. This introduces another way our programs can step to error—by trying to access a memory address using a VNum rather than a VPtr. As a convenience, we name the following ITrees for the programs that just perform a load and store, respectively.

$$\mathsf{load}\ p = \mathsf{vis}\ (\mathsf{Load}\ p)\ \mathsf{ret}$$

$$\mathsf{store}\ p\ v = \mathsf{vis}\ (\mathsf{Store}\ p\ v)\ \mathsf{ret}$$

We also introduce the Read and Write functions as a convenient variant of the read and write functions defined earlier that operate on Mems. These functions take a Val argument instead of an addr for the address to access, and return None if the Val is not a VPtr.

Now with these operations, we can define a more complex steps-to relation:

$$\frac{}{(s, \tau\ t) \to (s, t)}$$

$$\frac{\mathsf{Read}\ s\ p = \mathsf{Some}\ v}{(s, v \leftarrow (\mathsf{load}\ p); k\ v) \to (s, k\ v)} \qquad \frac{\mathsf{Write}\ s\ p\ v = \mathsf{Some}\ s'}{(s, (\mathsf{store}\ p\ v); k) \to (s', k\ \mathsf{unit})}$$

$$\frac{\mathsf{Read}\ s\ p = \mathsf{None}}{(s, v \leftarrow (\mathsf{load}\ p); k\ v) \to (\mathsf{error}(s), k\ (\mathsf{VNum}\ 0))} \qquad \frac{\mathsf{Write}\ s\ p\ v = \mathsf{None}}{(s, (\mathsf{store}\ p\ v; k) \to (\mathsf{error}(s), k\ \mathsf{unit})}$$

This relation sets the error bit to true if a memory error occurs, and never sets it back to false, which lets us determine if any memory errors happened during execution.

Using this new steps-to function and the updated Load and Store events, we can then define typing in the same way as in the previous section. All the results in Section 5.2 continue to hold.

5.4. Type Soundness

We will use rely-guarantee permissions to prove that the type system is sound—here, meaning that it ensures memory safety—using semantic type soundness. We have already defined the semantics of types and of typing using rely-guarantee permissions and proved some of our typing rules as theorems. In this section we will prove adequacy: that semantic typing implies soundness.

The adequacy theorem for the type system states that well-typed programs are indeed safe, meaning that they never step to a state where the error bit is true. However, the permissions that we type the program with are crucial. Recall that the input permission set False can type *any* stepping program or a ret. Further, if we had a permission whose guarantee permitted changes from non-error to error, an unsafe program would be well-typed with that input permission as well. To restrict our programs to those typed using *safe* permissions, we first define a non-error permission:

$$\pi_{ne} \stackrel{\mathsf{def}}{=} (\{ (s_1, s_2) \mid \text{if the error bit of } s_1 \text{ is false, then the error bit of } s_2 \text{ is false too} \},$$
$$=,$$
$$\{ s \mid \text{the error bit of } c \text{ is false} \})$$

Due to its rely, this permission is not separate with any permission that allows errors to occur in its guarantee. Then if we add this permission using $*$, we only deal with "safe" permissions that do not permit stepping to an error.

With this non-error permission, we are ready to prove the adequacy theorem:

Theorem 5.8. If $\Pi \vdash t : T$, then for any $\pi \in \Pi * [\pi_{ne}]$, and any $s \in P_\pi$, for any s' and t' where $(s, t) \to^* (s', t')$, the state s' has a false error bit.

As is typical with semantic type soundness, the adequacy theorem follows easily from the definition of typing. The theorem states that well-typed programs obey their permissions. Picking a

rely-guarantee permission $\pi \in \Pi * [\pi_{ne}]$ effectively selects a permission from Π that has a guarantee that does not permit going from non-error to error. This theorem then says that well-typed programs behave safely, as required by the permission π.

While there is little difference in functionality between our type system and a separation logic, our presentation of soundness is influenced by our perspective of it as a type system. A soundness result for a separation logic would typically look like "if the state satisfies the precondition, then the state after execution will satisfy the postcondition". Our soundness result does not mention the output permission, since we do not place much emphasis on output permissions aside from using them for composing programs using BIND. In Heapster, the main emphasis is on providing input permissions that are precise enough to typecheck the entire program, so that a functional specification can be extracted from the imperative program, and the definition of soundness reflects that emphasis.

5.5. Memory Typing Rules

In this section, we introduce some more interesting types and their typing rules. The main novelty of the type system is that it will include types for reading and writing to specific addresses. Multiple aliasing pointers where one of them is allowed to write to memory can lead to memory safety bugs and are disallowed by the type system. However, aliasing is allowed if all the pointers only read from memory.

This type system will be somewhat limited, as this section is meant to be an introduction to the full type system in Chapter 6. More realistic rules for crucial features like malloc and free, as well as more comprehensive coverage of typing rules will be presented in the next chapter.

Now that we have pointers in the language, it is also useful to introduce the notation $x : T$ as the function application $T\ x$, where T is a function from Val to $\mathsf{Perms}_{\mathsf{Mem}}$. Many of our types can be represented as functions from Val to $\mathsf{Perms}_{\mathsf{Mem}}$, so $x : T$ represents a value x having such a type T. For example, if we know that p points to another pointer, we could have a typing judgment $\Pi \vdash \mathsf{load}\ p : T_{ptr}$, where T_{ptr} is a type that describes what it points to and its capabilities. Then,

after this load program returns another pointer p', we can use the new notation to write $p' : T_{ptr}$, to signify that p' has the pointer type T_{ptr}. Concretely, this notation means that the program now holds the $\mathsf{Perms}_{\mathsf{Mem}}$ obtained after applying the function T_{ptr} to the value p' of type Val.

We start with pointer permissions, which will be used for typing Load and Store events. We define the *pointer read* and *pointer write* permissions on values $a \in \mathsf{addr}$ and $v \in \mathsf{Val}$ to represent the ability to read and write, respectively, to the address a which currently points to a value v.

$$\begin{aligned}
\pi_{read}(a, v) &\stackrel{\mathsf{def}}{=} (\{ (s_1, s_2) \mid \mathsf{read}\ s_1\ a = \mathsf{read}\ s_2\ a \}, \\
&\quad\ =, \\
&\quad\ \{ s \mid \mathsf{read}\ s\ a = \mathsf{Some}\ v \}) \\
\pi_{write}(a, v) &\stackrel{\mathsf{def}}{=} (\{ (s_1, s_2) \mid \mathsf{read}\ s_1\ a = \mathsf{read}\ s_2\ a \}, \\
&\quad\ \{ (s_1, s_2) \mid \text{the error bits of } s_1 \text{ and } s_2 \text{ are the same,} \\
&\qquad\quad \text{the number of blocks in } s_1 \text{ and } s_2 \text{ are the same,} \\
&\qquad\quad \text{the size of all blocks in } s_1 \text{ and } s_2 \text{ are the same, and} \\
&\qquad\quad \forall a' \neq a, \mathsf{read}\ s_1\ a' = \mathsf{read}\ s_2\ a' \}, \\
&\quad\ \{ s \mid \mathsf{read}\ s\ a = \mathsf{Some}\ v \})
\end{aligned}$$

The rely of each permission allows anything in the state to be modified other than the value pointed to by a, while the precondition requires the value pointed to by a to be v. The guarantee for π_{read} does not allow any update to be performed by the holder of the permission, while the guarantee for π_{write} allows the value pointed to by a to change, but everything else about the state must stay unchanged.

As the two permissions only differ in their guarantees, we can prove that we can always weaken a write permission into a read permission:

Lemma 5.9. $\pi_{write}(a,v) \sqsubseteq \pi_{read}(a,v)$

Another useful result is that we can duplicate the read permission:

Lemma 5.10. $\pi_{read}(a,v) \sqsubseteq \pi_{read}(a,v) * \pi_{read}(a,v)$.

In fact, we can duplicate any permission π where $G_\pi \subseteq R_\pi$, which implies it is separate from itself.

With these definitions, we can model the points-to assertion $a \mapsto v$ of separation logic, representing ownership of the address, using $\pi_{write}(a,v)$. The pointer-read permission, on the other hand, permits reading the address a using multiple read permissions via duplication, and resembles a fractional permission $a \overset{f}{\mapsto} v$ [Boy03]. The fraction f in a fractional permission can take rational values in $(0,1]$, representing full ownership—write access—when $f = 1$, and fractional ownership—read-only access—when $f < 1$. Fractional permissions can be split by splitting the fraction, and merged by recombining the fraction back together, solving the issue with pointer aliasing and write access mentioned previously. The $\pi_{read}(a,v)$ permission can also be split by duplicating, but unlike fractional permissions, can not be recombined to reobtain a $\pi_{write}(a,v)$. This major shortcoming is addressed by lifetimes in Chapter 7.

As discussed in Chapter 1, Heapster has the flavor of a type system rather than a separation logic. Heapster types typically do not describe the exact values that each pointer points to as in a separation logic, but are coarser descriptions. As such, it is more useful for a pointer type to contain the type of the value pointed to by the pointer, rather than the value itself. To represent this, we define a permission set that includes the *content permission*, the type of the value that the pointer points to:

$$x : ptr(rw \mapsto T) \overset{\mathsf{def}}{=} \begin{cases} \bigsqcup_{v \in \mathsf{Val}} [\pi_{rw}(a,v)] * v : T & \text{if } x = \mathsf{VPtr}\ a \\ \mathsf{False} & \text{if } x = \mathsf{VNum}\ _ \end{cases}$$

where *rw* is either *read* or *write*.

The use of the permission set in this definition is crucial. It allows us to existentially quantify over the value v using the join and combine the pointer permission with the content permission, which relies on the value v.

The content permission of a pointer permission could be another pointer permission. For example, $p : ptr(write \mapsto ptr(read \mapsto \mathsf{True}))$ would represent the knowledge that p is a pointer we have write access to, and the value it points to is another pointer, but one that we cannot update. That second pointer has a True content permission, meaning that we do not know anything about the value it points to.

If needed, we can still represent the exact value at an address in memory using an equality permission, defined as the following:

$$x : \mathsf{eq}(y) = \begin{cases} \mathsf{True} & \text{if } x = y \\ \mathsf{False} & \text{otherwise} \end{cases}$$

With this definition, we are only able to create equality permissions that relate equal values, since unequal values would result in the False permission set:

Theorem 5.11 (EQREFL)**.**

$$\frac{}{\Pi \sqsubseteq \Pi * x : \mathsf{eq}(x)}$$

The properties of equality also give us the following rules to further manipulate equality permissions:

Theorem 5.12 (EQSYM)**.**

$$\frac{}{x : \mathsf{eq}(y) \sqsubseteq y : \mathsf{eq}(x)}$$

Theorem 5.13 (EQTRANS)**.**

$$\frac{}{x : \mathsf{eq}(y) * y : \mathsf{eq}(z) \sqsubseteq x : \mathsf{eq}(z)}$$

Theorem 5.14 (EQCTX).

$$\frac{}{x : \mathsf{eq}(y) \sqsubseteq f\ x : \mathsf{eq}(f\ y)}$$

Like pointer-read permissions, equality permissions are duplicable, since they are just the vacuous True permission set:

Theorem 5.15 (EQDUP).

$$\frac{}{x : \mathsf{eq}(y) \sqsubseteq x : \mathsf{eq}(y) * x : \mathsf{eq}(y)}$$

Once an equality permission is obtained, it can be discharged with the following rule:

Theorem 5.16 (CAST).

$$\frac{}{x : \mathsf{eq}(y) * y : T \sqsubseteq x : T}$$

With the equality permission as the content permission of a pointer-write permission, we can again model the separation logic $p \mapsto v$, this time at the permission set level:

$$p : ptr(write \mapsto \mathsf{eq}(v))$$

Now that we have these types available to us, we can prove the typing rules for loads and stores. First, the rule for a load event:

Theorem 5.17 (LOAD).

$$\frac{}{p : ptr(rw \mapsto T) \vdash \mathsf{load}\ p : \lambda v. (p : ptr(read \mapsto \mathsf{eq}(v)) * v : T)}$$

where rw is *read* or *write*.

The equality permission is necessary here so that the type T is *moved* out of the pointer permission, not copied. By replacing the content permission with an equality permission, a connection to the original type T is retained, but without duplicating the permission set $v : T$. While some

permissions are safe to copy, it is not generally safe to duplicate permissions. For example, copying a pointer-write permission is not sound, since they represent exclusive ownership. That is, it is not true that $\pi_{write}(p, v) \sqsubseteq \pi_{write}(p, v) * \pi_{write}(p, v)$.

Similarly, the typing rule for a store also uses an equality permission:

Theorem 5.18 (STORE)**.**

$$\frac{}{p : ptr(write \mapsto T) \vdash \mathsf{store}\ p\ v : \lambda_.\, p : ptr(write \mapsto \mathsf{eq}(v))}$$

If we knew that $v : T'$ for some type T', it would not be sound to have this typing rule with the output permission $\lambda_.\, p : ptr(write \mapsto T')$. This is because it would be duplicating the permission set $v : T'$, which again, may not be safe. However, if we removed the original instance of the permission set $v : T'$ then the resulting rule would be sound:

Lemma 5.19 (STORE-ALT)**.**

$$\frac{}{p : ptr(write \mapsto T) * v : T' \vdash \mathsf{store}\ p\ v : \lambda_.\, p : ptr(write \mapsto T')}$$

These rules are equivalent, since STORE can be thought of as STORE-ALT where T' is the equality permission $\mathsf{eq}(v)$. We prefer to use STORE since it matches the corresponding rule in Heapster.

Now that we have rules for typing load and store instructions, let's use them in an example. Consider the following program, where we already have a value p:

$$p' \leftarrow \mathsf{load}\ p;$$
$$\mathsf{store}\ p'\ (\mathsf{VNum}\ 1)$$

In order for this to be memory safe, we need to know that p is a pointer to a value p', that p' is also a pointer, and that we can write to it. We can express this with the input permission set

$p : ptr(read \mapsto ptr(write \mapsto \mathsf{True}))$. Since we only do a store with the second pointer, the first pointer permission can be a read permission. For simplicity, we set the output permission to True, which can be acceptable if we do not intend to ever compose this computation with another, for example. Now that we have chosen the input and output permissions, we can typecheck the program:

Example 5.20.

$$\dfrac{\dfrac{}{\begin{array}{l}p : ptr(read \mapsto ptr(write \mapsto \mathsf{True})) \vdash \mathsf{load}\ p :\\ \lambda p'.\, p : ptr(read \mapsto \mathsf{eq}(p')) * p' : ptr(write \mapsto \mathsf{True})\end{array}}\ \textsc{Load} \qquad \dfrac{\dfrac{}{\begin{array}{l}p' : ptr(write \mapsto \mathsf{True}) \vdash \mathsf{store}\ p'\ (\mathsf{VNum}\ 1) :\\ \lambda_.\, p' : ptr(write \mapsto \mathsf{eq}(\mathsf{VNum}\ 1))\end{array}}\ \textsc{Store}}{\begin{array}{l}p : ptr(read \mapsto \mathsf{eq}(p')) * p' : ptr(write \mapsto \mathsf{True}) \vdash\\ \mathsf{store}\ p'\ (\mathsf{VNum}\ 1) : \lambda_.\, \mathsf{True}\end{array}}\ \textsc{Cnsq}}{p : ptr(read \mapsto ptr(write \mapsto \mathsf{True})) \vdash \begin{array}{l}p' \leftarrow \mathsf{load}\ p;\\ \mathsf{store}\ p'\ (\mathsf{VNum}\ 1)\end{array} : \lambda_.\, \mathsf{True}}\ \textsc{Bind}$$

The example is fairly short, using BIND to type each instruction separately using LOAD and STORE. Reading from the bottom up, we use CNSQ to weaken the input permission and strengthen the output permission. For the input permission, we use Lemma 4.14 to drop the unneeded permission about p. For the output permission, we strengthen it from True to the exact permission obtained from STORE.

If we want to strengthen the output permission to give more precise information about the pointers in the program, we have two options. One is to use a permission about p, for example $\lambda_.\, p : ptr(read \mapsto ptr(write \mapsto \mathsf{eq}(\mathsf{VNum}\ 1)))$. This, however, requires us to recombine the pointer permissions about p and p'. To do this we need a typing rule similar to CAST for pointer permissions with equality permissions as their content permissions, which we have not proved yet. Such a rule will be presented when we go through memory operations more comprehensively in Chapter 6. A second option is to slightly change the program by adding ret p' to the end, so we can refer to that pointer value in the output permission. Without this ret, we would not be able to use p' in the output permission, since it is not in scope until after the load instruction. Then, the output permission can be $\lambda p'.\, p : ptr(read \mapsto \mathsf{eq}(p')) * p' : ptr(write \mapsto \mathsf{eq}(\mathsf{VNum}\ 1))$.

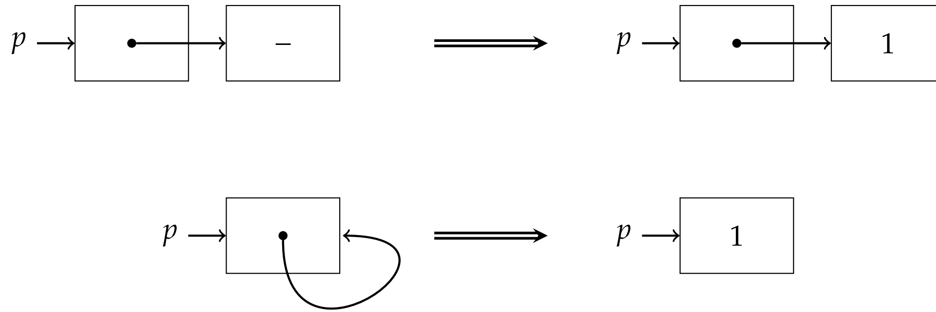


Figure 5.2: Two possible valid starting and final memory layouts.

One common source of bugs in pointer-manipulating programs is aliasing. What if p and p' are actually the same pointer? Edge cases like this commonly result in bugs, but luckily this program runs without error in this case. The memory layout for both cases is shown in Figure 5.2, along with the result of executing the program.

The input permission we used earlier, however, does not allow for this circularity. The content permission of a pointer permission must be *separate* from the pointer permission itself, due to its definition. Since one pointer permission is a read and one is a write, they are not separate and cannot coexist. The fact that the program typechecks in Example 5.20 does not mean that it is safe in all cases, but only that it is safe in the case without aliasing.

However, we can still express this possibility of circularity using an equality permission, proving that this program is also safe in the aliasing case. The input permission $p : ptr(write \mapsto \mathsf{eq}(p))$ represents the case when the pointer p points to a value which is a pointer to itself. In this example we use a more informative output permission of $\lambda_.\, p : ptr(write \mapsto \mathsf{eq}(\mathsf{VNum}\ 1))$, since the process of establishing this output permission illustrates some of the other typing rules we introduced.

Example 5.21.

$$
\dfrac{
\dfrac{}{\begin{array}{l} p : ptr(write \mapsto \mathsf{eq}(p)) \vdash \mathsf{load}\ p : \\ \lambda p'.\, p : ptr(write \mapsto \mathsf{eq}(p')) * p' : \mathsf{eq}(p) \end{array}}\,\textsc{Load}
\qquad
\dfrac{
\dfrac{
\dfrac{}{\begin{array}{l} p' : ptr(write \mapsto \mathsf{eq}(p')) \vdash \mathsf{store}\ p'\ (\mathsf{VNum}\ 1) : \\ \lambda_.\, p' : ptr(write \mapsto \mathsf{eq}(\mathsf{VNum}\ 1)) \end{array}}\,\textsc{Store}
}{\begin{array}{l} p' : ptr(write \mapsto \mathsf{eq}(p')) * p' : \mathsf{eq}(p) \vdash \mathsf{store}\ p'\ (\mathsf{VNum}\ 1) : \\ \lambda_.\, p' : ptr(write \mapsto \mathsf{eq}(\mathsf{VNum}\ 1)) * p' : \mathsf{eq}(p) \end{array}}\,\textsc{Frame}
}{\begin{array}{l} p : ptr(write \mapsto \mathsf{eq}(p')) * p' : \mathsf{eq}(p) \vdash \mathsf{store}\ p'\ (\mathsf{VNum}\ 1) : \\ \lambda_.\, p : ptr(write \mapsto \mathsf{eq}(\mathsf{VNum}\ 1)) \end{array}}\,\textsc{Cnsq}
}{
p : ptr(write \mapsto \mathsf{eq}(p)) \vdash \begin{array}{l} p' \leftarrow \mathsf{load}\ p; \\ \mathsf{store}\ p'\ (\mathsf{VNum}\ 1) \end{array} : \lambda_.\, p : ptr(write \mapsto \mathsf{eq}(\mathsf{VNum}\ 1))
}\,\textsc{Bind}
$$

We again use Bind to type each instruction separately using Load and Store. This time, the use of Cnsq is more complex. The goal is to change both input and output permissions into the form needed to use Store. In the input permission, we apply EqDup to duplicate the equality permission, then consume one of them using Cast to obtain a pointer permission for p'. In the output permission, since we are *strengthening* the permission, we change the permission to $\lambda_.\, p' : ptr(write \mapsto \mathsf{eq}(\mathsf{VNum}\ 1)) * p' : \mathsf{eq}(p)$. This is valid because we can weaken this to the previous output permission using EqSym and Cast. We chose these specific permissions because we then apply Frame to remove the now-unneeded equality permission in both input and output permissions. Finally, the resulting permissions are exactly those needed for Store.

CHAPTER 6

Specification Extraction

Part of this chapter is adapted from work previously published as Paul He, Eddy Westbrook, Brent Carmer, Chris Phifer, Valentin Robert, Karl Smeltzer, Andrei Ştefănescu, Aaron Tomb, Adam Wick, Matthew Yacavone, and Steve Zdancewic. "A Type System for Extracting Functional Specifications from Memory-Safe Imperative Programs." In: *Proc. ACM Program. Lang.* 5.OOPSLA (Oct. 2021) [He+21]. I was the primary author and the primary contributor of the theoretical component of the paper, the focus of this dissertation.

In this chapter, we use rely-guarantee permissions to verify the core of the type system used in Heapster, which extracts specifications from imperative programs. The approach in this chapter extends the ideas presented in Chapter 5, primarily with the addition of a second program—the functional specification—to the typing judgment. The typing judgment in this chapter will relate two programs and the permissions held before and after the execution of these programs. The first program, which we will call the *implementation* or *imperative* program, is like the program in the typing judgment for Chapter 5. Implementation programs represent the user's input code in an imperative language like C. The second program is what we will call the *functional* or *specification* program, which represents the pure functional program that is extracted by Heapster. Specification programs should be simpler versions of the implementation ones, with imperative features like memory accesses removed.

A key detail is that the memory safety of the implementation program is not always easy to determine. For example, whether an array access is safe depends on whether the index is within the bounds of the array, so we cannot easily extract a safe functional specification program. To handle this case, we permit specifications to contain errors, so they can represent the case when the implementation program is *not* safe. For the example of array accesses, the check that the index is within bounds is translated to a dynamic check in the specification program, which can fail.

We first update the definitions of our language and our typing judgment, as well as the soundness result for this type system. With this new definition, we can then introduce the new types of the type system, their typing rules, and some small examples of using these typing rules. Section 6.6 will present a larger example that shows more involved usage of these types and typing rules. Finally, we will discuss the relationship between the type system presented in this chapter and the Heapster tool, and how the tool is typically used.

6.1. Definitions and Semantic Typing

The language considered in this section differs from that of Chapter 5. We wish to support more memory operations, not just loading and storing single values. For example, allocation and deallocation are crucial operations that change entire blocks of memory, not just a single value in a block. Rather than modifying the event type and adding new cases alongside the existing Load and Store events from the previous chapter, we will instead change the event type to be able to handle any state access. We will call the event type E_S, for a state type S. To represent any read or write of the state, we define the event

$$\mathsf{Modify} : (S \to S) \to \mathsf{E}_S\ S.$$

The event Modify f carries a function f that can represent arbitrary changes to the entire state. It also allows us to read and write simultaneously in one event. Whereas the previous model was specific to the concrete Mem memory model and only allowed reading and writing using individual pointers, this one is expressive enough to represent any state update to the state type, which could be specialized to Mem. Now changes to the language will only require updating the state type, and not the event type for our ITrees, allowing us to define the typing judgment for any state and language, rather than a different one for each language.

The steps-to relation then only needs to handle the τ case and the case for this Modify event:[4]

$$(s, \tau\ t) \rightarrow (s, t) \qquad (s, \mathsf{vis}\ (\mathsf{Modify}\ f)\ k) \rightarrow (f\ s, k\ s)$$

For the Modify event, the continuation gets the value of the *previous* state, before the state is modified using f. This is meant to be a convenient way of both getting the previous value and updating the state, rather than having to use two Modify events in a row.

In addition to the Modify event, we also have exception events, which we will use to represent errors as *programs*, rather than as error *states* as in Chapter 5.

$$\mathsf{Throw} : \mathsf{E}_S\ \emptyset$$

This exception event expects a value of the void type, $\emptyset$, from the environment as a response. Of course, no such value exists, so this type effectively makes Throw a special leaf node in ITrees that use it, which can give the resulting ITree *any* return type. We call the ITree that consists of just this event error:

$$\mathsf{error} : \mathsf{itree}\ \mathsf{E}_{\mathsf{Mem}}\ R$$
$$\mathsf{error} \stackrel{\mathsf{def}}{=} \mathsf{vis}\ \mathsf{Throw}\ (\lambda x : \emptyset.\ \bot)$$

We use an error program rather than an error state because we are less interested in completely preventing error from occurring, as in Chapter 5, but rather want to ensure that the implementation and specification programs we are typechecking behave similarly. We no longer need to prevent errors using a permission's guarantee, but will instead require that errors in each program are related using the definition of typing itself. As we will see in more detail later, in some cases, the correctness of the imperative program may depend on memory in some way that cannot be directly translated to a functional specification. In these cases, we can use error to represent the

[4] The Coq formalization also includes nondeterminism events, which we do not use in this dissertation. These events were used to represent concurrent programs, and the approach is described in Section 8.1.

possibility of a memory error in the functional specification.

Generalizing the state type is also necessary since we want the imperative implementation programs and functional specifications to have the same event signature, for ease of defining the typing judgment. To reflect the differences between the two languages, such as the absence of a heap in the functional language, the two will use different state types. In this chapter, imperative implementation programs will have state type Mem, and functional specification programs will have state type Unit, and not use any state at all.

Rather than defining typing all at once as we did in Chapter 5, we define it here in two steps. We first define a notion of *bisimulation* to relate the implementation and specification programs with respect to a specific input rely-guarantee permission and an output permission set. We then use this bisimulation to generalize to a typing judgment with an input permission *set*. By making the currently-held permission explicit rather than having to choose an arbitrary permission from a permission set, this simplifies many typing proofs.

For the bisimulation, we keep the state types for the implementation and specification programs abstract, naming them S_i and S_s respectively. While the programs only deal with their own state types, we use rely-guarantee permissions to relate the two programs, and so they must deal with both state types. The rely-guarantee permissions we use are therefore permissions over $(S_i \times S_s)$.

More specifically, the bisimulation will be a stuttering bisimulation [BCG88], which allows either program to step finitely without the other taking any steps. We use this instead of a standard bisimulation, which requires the two programs to run in lock step, because such synchronization is not always possible. For example, the imperative program may involve some memory operations, but the specification is meant to remove such operations, and will not perform those steps.

In addition to stuttering, the bisimulation will also be asymmetric, allowing errors to occur on the right, in the specification program. This is due to the asymmetry of how we will *use* this definition in typing—since the programs are supposed to be implementations and specifications, there should be some notion of refinement in this definition. The implementation program should

refine the specification, so it is always acceptable to add an extra error behavior to the specification program. As we will see in the adequacy theorem for this type system, this asymmetry tells us that if the specification is safe—that it cannot step to an error—then the implementation program must be safe too.

Definition 6.1 (Stuttering bisimulation up to errors on the right)**.** Let $s_i \in S_i$, $s_s \in S_s$, $t_i \in \mathsf{itree}\ \mathsf{E}_{S_i}\ R_i$, and $t_s \in \mathsf{itree}\ \mathsf{E}_{S_s}\ R_s$. Given a permission π over $S_i \times S_s$ (the "input permission") and a function $T : R_i \to R_s \to \mathsf{Perms}_{S_i \times S_s}$ (the "output permission type"), we define *stuttering bisimulation up to errors on the right* as the biggest relation $(t_i, s_i) \lessapprox_{\pi,T} (t_s, s_s)$ such that one of the following cases hold:

1. $t_i = \mathsf{ret}\ r_i$ and $t_s = \mathsf{ret}\ r_s$ for some $r_i \in R_i$ and $r_s \in R_s$ where $\pi \in T\ r_1\ r_2$ and $(s_i, s_s) \in P_\pi$.

2. $t_s = \mathsf{error}$.

3. $(s_i, s_s) \in P_\pi$ and for any t_i' and s_i' such that $(s_i, t_i) \to (s_i', t_i')$, we have $((s_i, s_s), (s_i', s_s)) \in G_\pi$, and there is some permission π' where $\pi \rightsquigarrow \pi'$ and $(t_i', s_i') \lessapprox_{\pi',T} (t_s, s_s)$.

4. $(s_i, s_s) \in P_\pi$ and for any t_s' and s_s' such that $(s_s, t_s) \to (s_s', t_s')$, we have $((s_i, s_s), (s_i, s_s')) \in G_\pi$, and there is some permission π' where $\pi \rightsquigarrow \pi'$ and $(t_i, s_i) \lessapprox_{\pi',T} (t_s', s_s')$.

5. t_i and t_s start with the same type of event, $(s_i, s_s) \in P_\pi$, and both of the following are true:

   - for any t_i' and s_i' such that $(s_i, t_i) \to (s_i', t_i')$, there exists t_s' and s_s' such that $(s_s, t_s) \to (s_s', t_s')$, where $((s_i, s_s), (s_i', s_s')) \in G_\pi$, and there is some permission π' where $\pi \rightsquigarrow \pi'$ and $(t_i', s_i') \lessapprox_{\pi',T} (t_s', s_s')$.

   - for any t_s' and s_s' such that $(s_s, t_s) \to (s_s', t_s')$, there exists t_i' and s_i' such that $(s_i, t_i) \to (s_i', t_i')$, where $((s_i, s_s), (s_i', s_s')) \in G_\pi$, and there is some permission π' where $\pi \rightsquigarrow \pi'$ and $(t_i', s_i') \lessapprox_{\pi',T} (t_s', s_s')$.

Furthermore, cases 3 and 4 cannot be applied in an infinite chain, whereas case 5 *can*,[5] to ensure

[5]In the Coq development, this is implemented using a mixed inductive-coinductive definition, where cases 3 and 4

diverging computations are not related to every program, and two diverging computations can be related to each other.

There are strong similarities to Definition 5.2, the definition of typing in Chapter 5. Case 1 is similar to the case in the previous definition that relates two ret programs. This case is the only one that uses the output permission type, and ensures that the input permission is in the permission set obtained from the output type. Cases 3, 4, and 5 involve stepping. The process of stepping is also very similar to the previous definition, requiring that the step is allowed by the current permission's guarantee, and that the new program continues to be bisimilar using a permission that is at least as separate as the previous one. For all four of these cases, the starting states must be in the precondition of π, ensuring that they satisfy the assumptions that the permission makes about the state.

Other parts of the definition are not as similar as the previous definition of typing. Case 2 allows any program to be related to the error specification. As discussed earlier, this adds a notion of the implementation refining the specification to this bisimulation. Stuttering steps are included using cases 3 and 4, which allow one program to step while the other program does not. Case 5, on the other hand, captures the "standard" bisimulation definition, requiring that any step either program makes is matched by the other.

Using this definition, if the specification has no errors, then neither does the implementation:

Theorem 6.1 (Error-Freedom)**.** If $(t_i, s_i) \lessapprox_{\pi,T} (t_s, s_s)$ and $(s_s, t_s) \not\to^* \mathsf{error}$ then $(s_i, t_i) \not\to^* \mathsf{error}$.

We define typing on top of this bisimulation:

Definition 6.2 (Typing)**.** Let $\Pi \in \mathsf{Perms}_{S_i \times S_s}$ (the "input permission set"), $T : R_i \to R_s \to \mathsf{Perms}_{S_i \times S_s}$ (the "output permission type"), $t_i \in \mathsf{itree}\ \mathsf{E}_{S_i}\ R_i$ (the implementation program), and $t_s \in \mathsf{itree}\ \mathsf{E}_{S_s}\ R_s$ (the specification program). We define the typing judgment $\Pi \vdash t_i \lessapprox t_s : T$ to hold if for any $\pi \in \Pi$ and any s_i and s_s such that $(s_i, s_s) \in P_\pi$, we have $(t_i, s_i) \lessapprox_{\pi,T} (t_s, s_s)$.

are inductive and case 5 is coinductive. The Coq formalization is also slightly different than presented here in that it inlines the use of $\to$, resulting in 6 stepping cases rather than the 3 shown here.

This definition hides some of the details of bisimulation, like the starting states, and changes the input permission to a permission set, more closely resembling Definition 5.2 for typing in the simplified setting.

The adequacy theorem for this type system follows easily from Theorem 6.1. As in Chapter 5, adequacy is easy to prove from the definition of semantic typing. Since we now represent error as a program rather than a state, we do not need the no-error permission found in Theorem 5.8.

Theorem 6.2 (Adequacy)**.** If $\Pi \vdash t_i \lessapprox t_s : T$, then for any $\pi \in \Pi$ and $(s_i, s_s) \in P_\pi$, $(s_s, t_s) \not\to^* \mathsf{error}$ implies that $(s_i, t_i) \not\to^* \mathsf{error}$.

6.2. Permission Types

In this section we present the semantic interpretation of a type in our type system and introduce some basic types and typing rules. More involved types, like those dealing with memory or arrays, will be covered in other sections. For this section, we will continue to hold the state types for implementation and specification programs abstract, calling them S_i and S_s.

While we used the term "type" loosely in the context of the previous chapter's type system, we define them more formally here:

Definition 6.3 (Permission types)**.** An (A_i, A_s)-*permission type* is a function from an implementation value of type A_i and a specification value of type A_s to a permission set in $\mathsf{Perms}_{S_i \times S_s}$. We write $\mathsf{PType}(A_i, A_s)$ for the set of (A_i, A_s)-permission types.

A permission type should be thought of as relating an implementation value and a specification value. When these values are available, we write $x_i : T \triangleright x_s$ for the application $T\ x_i\ x_s$ of T to $x_i \in A_i$ and $x_s \in A_s$. For example, later in this section we will define a permission type Nat that relates an implementation Val with a specification natural number value. Then $x_i : \mathsf{Nat} \triangleright x_s$ is a permission set which tells us that x_i is a Val that represents a natural number and x_s is a specification value that represents the *same* natural number. The representation of a natural number in specification programs will be described later in this section, when we define Nat.

$$\frac{}{x_i : T \triangleright x_s \vdash \mathsf{ret}\ x_i \lessapprox \mathsf{ret}\ x_s : T}\ \textsc{Ret} \qquad \frac{}{\Pi \vdash t_i \lessapprox \mathsf{error} : T}\ \textsc{Err}$$

$$\frac{\Pi_1 \sqsubseteq \Pi_2 \qquad \forall x_i, x_s,\ x_i : T_2 \triangleright x_s \sqsubseteq x_i : T_1 \triangleright x_s \qquad \Pi_2 \vdash t_i \lessapprox t_s : T_2}{\Pi_1 \vdash t_i \lessapprox t_s : T_1}\ \textsc{Cnsq}$$

$$\frac{\Pi \vdash t_i \lessapprox t_s : T_1 \qquad \forall x_i, x_s,\ x_i : T_1 \triangleright x_s \vdash k_i\ x_i \lessapprox k_s\ x_s : T_2}{\Pi \vdash (x \leftarrow t_i; k_i\ x) \lessapprox (x \leftarrow t_s; k_s\ x) : T_2}\ \textsc{Bind}$$

Figure 6.1: Basic structural typing rules.

The output permission type argument of the typing judgment is an (R_i, R_s)-permission type, where R_i and R_s are the return types of the implementation and specification programs, respectively. Consequently, the output type should be thought of as relating the final value of the implementation program with the final value of the specification program.

As a first example of rules that use these permission types, we present some typing rules that deal with the basic structure of programs in Figure 6.1. The RET and ERR rules follow from their respective base cases in the definition of bisimulation. RET will typecheck two ret programs if they return values that are already known to be related by type T in the input permission set, and maintains this knowledge by using T as the output type. ERR can extract an error specification from *any* implementation program and any input and output permissions, acting as an escape hatch for when the implementation program cannot be typechecked in another way. CNSQ and BIND are the same as the corresponding rules presented in Chapter 5, but with specification programs added.

BIND splits both implementation and specification programs. This may seem like a limitation due to requiring that both programs be split even when they are not one-to-one translations, but the fact that ITree is a monad provides flexibility here. The associativity monad law allows the two programs to be split differently, and the identity monad laws allow us to add extraneous no-op programs like `ret unit`. These no-op programs are especially useful for when a portion of the implementation program corresponds to no computation in the specification program, a pattern we will see often in upcoming rules and examples.

$$\frac{}{\Pi \sqsubseteq \Pi * x_i : \mathsf{eq}(x_i) \triangleright \mathsf{unit}}\ \textsc{EqRefl} \qquad \frac{}{x_i : \mathsf{eq}(y_i) \triangleright \mathsf{unit} \sqsubseteq y_i : \mathsf{eq}(x_i) \triangleright \mathsf{unit}}\ \textsc{EqSym}$$

$$\frac{}{x_i : \mathsf{eq}(y_i) \triangleright \mathsf{unit} * y_i : \mathsf{eq}(z_i) \triangleright \mathsf{unit} \sqsubseteq x_i : \mathsf{eq}(z_i) \triangleright \mathsf{unit}}\ \textsc{EqTrans}$$

$$\frac{}{x_i : \mathsf{eq}(y) \triangleright \mathsf{unit} \sqsubseteq f\ x_i : \mathsf{eq}(f\ y) \triangleright \mathsf{unit}}\ \textsc{EqCtx}$$

$$\frac{}{x_i : \mathsf{eq}(y) \triangleright \mathsf{unit} \sqsubseteq x_i : \mathsf{eq}(y) \triangleright \mathsf{unit} * x_i : \mathsf{eq}(y) \triangleright \mathsf{unit}}\ \textsc{EqDup}$$

$$\frac{}{x_i : \mathsf{eq}(y_i) \triangleright \mathsf{unit} * y_i : T \triangleright y_s \sqsubseteq x_i : T \triangleright y_s}\ \textsc{Cast}$$

Figure 6.2: Typing rules for equality permission types.

6.2.1. Equality Types

We can redefine the equality permissions from chapter 5 to include specification values:

$$\mathsf{eq} : A \to \mathsf{PType}(A, \mathsf{Unit})$$

$$x : \mathsf{eq}(y) \triangleright \mathsf{unit} \stackrel{\mathsf{def}}{=} \begin{cases} \mathsf{True} & \text{if } x = y \\ \mathsf{False} & \text{otherwise} \end{cases}$$

In Heapster, equality types are only relevant in the typechecking process and are not transferred over to specification programs, so they are related to Unit values on the specification side. Figure 6.2 contains all the typing rules for equality types. These are the same as the rules presented in Chapter 5, but with an added unit on the specification side.

As an example of using equality types, we consider a simple program that we will use as a single instruction in imperative programs. This instruction $\mathsf{getNum}\ v$ attempts to return the numerical value from a $\mathsf{Val}\ v$, and fails if the Val is not a number:

$$\mathsf{getNum} : \mathsf{Val} \to \mathsf{itree}\ \mathsf{E}_{S_i}\ \mathbb{N}$$

$$\mathsf{getNum}\ (\mathsf{VNum}\ n) \stackrel{\mathsf{def}}{=} \mathsf{ret}\ n$$

$$\mathsf{getNum}\ (\mathsf{VPtr}\ _) \stackrel{\mathsf{def}}{=} \mathsf{error}$$

Equality types play a crucial role in the typing rule for this instruction:

Lemma 6.3 (GETNUM).

$$\frac{}{x_i : \mathsf{eq}(\mathsf{VNum}\ n) \triangleright \mathsf{unit} \vdash \mathsf{getNum}\ x_i \lessapprox \mathsf{ret}\ \mathsf{unit} : \mathsf{eq}(n)}$$

This rule tells us that if we have an equality type with the information that the Val x_i contains a number n, getNum x_i will succeed and produce that same numerical value n. This property about the output of the program is expressed as the output type. The specification program corresponding to getNum on the implementation side is the trivial program that just returns unit. This is because getNum operates on Vals—implementation values—which we manage using equality types. Equality types have no computational content in specifications, so this typing rule follows suit.

Equality types allow us to relate arbitrary values, even those used in specifications. The following rule requires an implementation value to be equal to a specification value, and extracts conditional expressions from conditional expressions on the implementation side.

Theorem 6.4 (IF).

$$\frac{\Pi \vdash t_{i_1} \lessapprox t_{s_1} : T \qquad \Pi \vdash t_{i_2} \lessapprox t_{s_2} : T}{\Pi * x_i : \mathsf{eq}(x_s) \triangleright \mathsf{unit} \vdash \mathsf{if}\ x_i\ \mathsf{then}\ t_{i_1}\ \mathsf{else}\ t_{i_2} \lessapprox \mathsf{if}\ x_s\ \mathsf{then}\ t_{s_1}\ \mathsf{else}\ t_{s_2} : T}$$

This rule says that if we can show that each branch of the conditional expression is well-typed, then the entire expression is well-typed. An equality type is used to match the condition values on both the implementation and specification sides, ensuring that the branches of the if statements on both sides correspond to each other.

6.2.2. Permission Type Connectives

Now that we have a formal definition of permission types, we will also need connectives for these types. In Chapter 5, we directly used the semantic definition of types as functions. For

$$\frac{}{(x_i : T \triangleright x_s) * \Pi \sqsubseteq x_i : T \oslash \Pi \triangleright x_s}\ \textsc{PermsI} \qquad \frac{}{x_i : T \oslash \Pi \triangleright x_s \sqsubseteq (x_i : T \triangleright x_s) * \Pi}\ \textsc{PermsE}$$

$$\frac{\Pi_1 \vdash t_i \lessapprox t_s : T}{\Pi_1 * \Pi_2 \vdash t_i \lessapprox t_s : T \oslash \Pi_2}\ \textsc{Frame}$$

$$\frac{}{(x_i : T_1 \triangleright x_s) * (y_i : T_2 \triangleright y_s) \sqsubseteq (x_i, y_i) : T_1 \otimes T_2 \triangleright (x_s, y_s)}\ \textsc{ProdI}$$

$$\frac{}{x_i : T_1 \otimes T_2 \triangleright x_s \sqsubseteq (x_i.1 : T_1 \triangleright x_s.1) * (x_i.2 : T_2 \triangleright x_s.2)}\ \textsc{ProdE}$$

$$\frac{}{(x_i : T_1 \triangleright x_s) * (x_i : T_2 \triangleright y_s) \sqsubseteq x_i : T_1 \star T_2 \triangleright (x_s, y_s)}\ \textsc{StarI}$$

$$\frac{}{x_i : T_1 \star T_2 \triangleright x_s \sqsubseteq (x_i : T_1 \triangleright x_s.1) * (x_i : T_2 \triangleright x_s.2)}\ \textsc{StarE}$$

Figure 6.3: Typing rules for conjunction permission types.

example, for the FRAME rule in that chapter, we add a permission set using $*$ in the function that represents the output type. If we want a *syntactic* type system, we cannot make use of this *semantic* interpretation of types in typing rules. Rather, we will define new connectives for combining types and prove typing rules for the way we want them to behave.

The first connectives we start with are connectives for expressing the product, or conjunction, of permission types. Their typing rules are shown in Figure 6.3.

The first connective we present is the permission set conjunction type $T \oslash \Pi$, which conjoins a permission set Π to a permission type T. We need this connective to be able to add permission sets directly to types, which is used in the frame rule. The semantic definition of this type connective is as follows:

$$\oslash : \mathsf{PType}(A_i, A_s) \to \mathsf{Perms}_{S_i \times S_s} \to \mathsf{PType}(A_i, A_s)$$
$$x_i : (T \oslash \Pi) \triangleright x_s \overset{\mathsf{def}}{=} (x_i : T \triangleright x_s) * \Pi$$

The PERMSI and PERMSE typing rules convert between $\oslash$ and its definition, for converting between the different formats used in input and output permissions. In input permissions, $*$ can be used, but in output permissions, $\oslash$ is necessary. FRAME is the frame rule, which is just like the

previous FRAME rule from Chapter 5, but uses this new $\oslash$ definition. As the frame rule does not affect the implementation program, the specification program is also unchanged.

The second conjunction connective we introduce is the product permission type $T_1 \otimes T_2$, which relates pairs on the implementation side to pairs on the specification side. We write values of product types $A \times B$ as (a, b), and represent the first and second projections of a pair with the notation $p.1$ and $p.2$ respectively.

$$\otimes : \mathsf{PType}(A_i, A_s) \to \mathsf{PType}(B_i, B_s) \to \mathsf{PType}(A_i \times B_i, A_s \times B_s)$$
$$x_i : (T_1 \otimes T_2) \triangleright x_s \stackrel{\mathsf{def}}{=} (x_i.1 : T_1 \triangleright x_s.1) * (x_i.2 : T_2 \triangleright x_s.2)$$

This connective relates implementation value x_i to specification value x_s by relating their first projections with T_1 and their second projections with T_2. This connective is useful for implementation programs that use pairs, and will extract pairs to the specification program as well—an example of which we will see in Section 6.6. Like the introduction and elimination rules for $\oslash$, PRODI and PRODE fold and unfold the definition of $\otimes$, for use in either input permissions or output types.

The final conjunction connective we introduce is the separating conjunction permission type $T_1 \star T_2$, which relates a single imperative value x_i to a pair of specification values by relating x_i to the first specification value with T_1 and to the second with T_2.

$$\star : \mathsf{PType}(A_i, A_s) \to \mathsf{PType}(A_i, B_s) \to \mathsf{PType}(A_i, A_s \times B_s)$$
$$x_i : (T_1 \star T_2) \triangleright x_s \stackrel{\mathsf{def}}{=} (x_i : T_1 \triangleright x_s.1) * (x_i : T_2 \triangleright x_s.2)$$

Like the other connectives, the STARI and STARE rules introduce and eliminate the connective by folding and unfolding its definition. This connective is useful for when a single implementation value is associated with multiple types, and through those multiple permission types, multiple specification values. For example, a pointer to a linked list node would have multiple pointer types, one for each value in the node. These types can be combined using the $\star$ connective, as we will see in Section 6.5.

$$\frac{}{x_i : T_1 \rhd x_s \sqsubseteq \mathsf{inl}\ x_i : T_1 \oplus T_2 \rhd \mathsf{inl}\ x_s}\ \textsc{SumI1} \qquad \frac{}{x_i : T_2 \rhd x_s \sqsubseteq \mathsf{inr}\ x_i : T_1 \oplus T_2 \rhd \mathsf{inr}\ x_s}\ \textsc{SumI2}$$

$$\frac{\forall y_i, y_s,\ \Pi * y_i : T_1 \rhd y_s \vdash t_{i_1} \lessapprox t_{s_1} : T_3 \qquad \forall z_i, z_s,\ \Pi * z_i : T_2 \rhd z_s \vdash t_{i_2} \lessapprox t_{s_2} : T_3}{\Pi * x_i : T_1 \oplus T_2 \rhd x_s \vdash \mathsf{case}\ x_i\ \mathsf{of}\ (\lambda y_i.\, t_{i_1})\ (\lambda z_i.\, t_{i_2}) \lessapprox \mathsf{case}\ x_s\ \mathsf{of}\ (\lambda y_s.\, t_{s_1})\ (\lambda z_s.\, t_{s_2}) : T_3}\ \textsc{SumE}$$

$$\frac{}{x_i : T_1 \rhd x_s \sqsubseteq x_i : T_1 \vee T_2 \rhd \mathsf{inl}\ x_s}\ \textsc{OrI1} \qquad \frac{}{x_i : T_2 \rhd x_s \sqsubseteq x_i : T_1 \vee T_2 \rhd \mathsf{inr}\ x_s}\ \textsc{OrI2}$$

$$\frac{\forall y_s,\ \Pi * x_i : T_1 \rhd y_s \vdash t_i \lessapprox t_{s_1} : T_3 \qquad \forall z_s,\ \Pi * x_i : T_2 \rhd z_s \vdash t_i \lessapprox t_{s_2} : T_3}{\Pi * x_i : T_1 \vee T_2 \rhd x_s \vdash t_i \lessapprox \mathsf{case}\ x_s\ \mathsf{of}\ (\lambda y_s.\, t_{s_1})\ (\lambda z_s.\, t_{s_2}) : T_3}\ \textsc{OrE}$$

Figure 6.4: Typing rules for disjunction permission types.

Next we introduce two connectives for disjunctive types. The typing rules for these are shown in Figure 6.4.

The first disjunctive connective is the sum permission type $T_1 \oplus T_2$, which relates sum types on both the implementation and specification sides.

$$\begin{aligned} \oplus : \mathsf{PType}(A_i, A_s) \to \mathsf{PType}(B_i, B_s) \to \mathsf{PType}(A_i + B_i, A_s + B_s) \\ \mathsf{inl}\ x_i : (T_1 \oplus T_2) \rhd \mathsf{inl}\ x_s \stackrel{\mathsf{def}}{=} x_i : T_1 \rhd x_s \\ \mathsf{inr}\ x_i : (T_1 \oplus T_2) \rhd \mathsf{inr}\ x_s \stackrel{\mathsf{def}}{=} x_i : T_2 \rhd x_s \\ \mathsf{inl}\ _ : (T_1 \oplus T_2) \rhd \mathsf{inr}\ _ \stackrel{\mathsf{def}}{=} \mathsf{False} \\ \mathsf{inr}\ _ : (T_1 \oplus T_2) \rhd \mathsf{inl}\ _ \stackrel{\mathsf{def}}{=} \mathsf{False} \end{aligned}$$

We write $\mathsf{inl}\ x$ to represent the injection of a value x from the left type of a sum type, and inr for the right type. In the case where the implementation and specification values are constructed from different sides of their sum types, the resulting permission set is False, representing inconsistency.

The SumI1 and SumI2 rules introduce sum types $T_1 \oplus T_2$ by applying the same injection operator on both sides, while the SumE rule performs sum elimination on both sides using case expressions. The first function in the case expression handles the case where the value comes from the left type, and the second handles the right type. The $\oplus$ connective is useful for when the implementation program makes use of sum types, and extracts to sum types as well in the specification program.

Recall that ITrees support an iter operation that uses sum types to signal whether a loop should continue or finish iterating.

$$\mathsf{iter} : (A \to \mathsf{itree}\ E\ (A + B)) \to A \to \mathsf{itree}\ E\ B$$

The type A signifies that the loop should continue with a new value of type A, and a value of type B ends iteration. This sum connective allows us to write a succinct typing rule for iter:

Theorem 6.5 (ITER).

$$\frac{\forall y_i, y_s,\ y_i : T_1 \triangleright y_s \vdash f_i\ y_i \lessapprox f_s\ y_s : T_1 \oplus T_2}{x_i : T_1 \triangleright x_s \vdash \mathsf{iter}\ f_i\ x_i \lessapprox \mathsf{iter}\ f_s\ x_s : T_2}\ \textsc{Iter}$$

T_1 is used to relate the inputs to the iteration, and T_2 is used to relate the outputs once iteration is complete. This typing rule says that for an iter on the implementation side, we can extract an iter on the specification side. Typechecking these iters reduces to typechecking their loop bodies. The $\oplus$ connective is used to ensure that for both ways the loop body can continue—either by iterating again or finishing—the resulting values are related with the corresponding type, either T_1 or T_2.

The second connective is the disjunctive permission type $T_1 \vee T_2$, which relates an implementation value x_i to a specification value using either T_1 or T_2.

$$\vee : \mathsf{PType}(A_i, A_s) \to \mathsf{PType}(A_i, B_s) \to \mathsf{PType}(A_i, A_s + B_s)$$
$$x_i : (T_1 \vee T_2) \triangleright \mathsf{inl}\ x_s \stackrel{\mathsf{def}}{=} x_i : T_1 \triangleright x_s$$
$$x_i : (T_1 \vee T_2) \triangleright \mathsf{inr}\ x_s \stackrel{\mathsf{def}}{=} x_i : T_2 \triangleright x_s$$

The disjunctive introduction rules ORI1 and ORI2 introduce a disjunctive type $T_1 \vee T_2$ from type T_1 or T_2, respectively, by applying the appropriate constructor to the specification value. The elimination rule ORE eliminates a disjunctive type $T_1 \vee T_2$ by inserting a sum elimination into the specification value. Similar to the SUME rule, ORE involves typing judgments and not just a change in permission sets using $\sqsubseteq$, since the programs must be altered using case expressions.

This connective $\vee$ relates to $\oplus$ much in the same way as $\star$ relates to $\otimes$. $\otimes$ is useful for when products are present in programs, and $\oplus$ is useful for when sums are present. $\star$ is useful for relating one implementation value to multiple specification values, and $\vee$ is useful for relating one implementation value to one of a choice of multiple specification values. As we will see in Section 6.5, this property is useful for recursive objects like linked lists. A node pointer, for example, may either point to the next node or be equal to null to represent the end of the linked list, and $\vee$ can represent this structure.

Another useful connective is the existential permission type $\exists(v : A), F\ v$, which allows us to "hide" the value v. For example, using an existential type, we can define the Nat permission type, telling us that an imperative value is a number rather than a pointer:

$$\mathsf{Nat} \stackrel{\mathsf{def}}{=} \exists n : \mathbb{N}.\mathsf{eq}(\mathsf{VNum}\ n)$$

The existential type relates an implementation value x_i to a dependent pair which contains the hidden value v as well as the specification value that $F\ v$ originally related x_i to.

$$\exists v : A.F\ v : \mathsf{PType}(A_i, \Sigma_{v:A}(F\ v))$$
$$x_i : (\exists v : A.F\ v) \triangleright \{v, x_s\} \stackrel{\mathsf{def}}{=} x_i : F\ v \triangleright x_s$$

Here, $\Sigma_{v:A} T$ denotes a dependent pair type. We will write values of this type as $\{a, b\}$, and overload the projection notation of pairs, $p.1$ and $p.2$ for projections of dependent pairs as well.

With this definition, we can see that Nat has type $\mathsf{PType}(\mathsf{Val}, \Sigma_{n:\mathbb{N}}\mathsf{Unit})$, relating Vals on the implementation side to specification values of type $\Sigma_{n:\mathbb{N}}\mathsf{Unit}$, which is isomorphic to $\mathbb{N}$ itself. The use of the equality type in the definition of Nat tells us that these two values represent the same natural number, just packaged in different types.

The following typing rules introduce and eliminate existential types by folding and unfolding their definitions:

Lemma 6.6 (ExI).

$$\overline{x_i : F\ y_s \triangleright x_s \sqsubseteq x_i : (\exists v : A.F\ v) \triangleright \{y_s, x_s\}}$$

Lemma 6.7 (ExE).

$$\overline{x_i : (\exists v : A.F\ v) \triangleright x_s \sqsubseteq x_i : F\ x_s.1 \triangleright x_s.2}$$

For example, we can use ExI to convert an equality permission to Nat, which would relate implementation and specification values. Using ExE, we can later unfold the Nat type to use the fact that the implementation and specification values they relate contain the same $\mathbb{N}$ value.

Finally, we define a vacuous permission type that always holds. While it is not a connective per se, we introduce it here as well since it is a general permission type, defined for any state type. We overload the notation True to define a type that represents holding the vacuous permission, using our previous definition of True as a permission set.

$$\mathsf{True} : \mathsf{PType}(A_i, \mathsf{Unit})$$
$$x_i : \mathsf{True} \triangleright \mathsf{unit} \stackrel{\text{def}}{=} \mathsf{True}$$

True relates any implementation value to unit on the specification side. This is useful, for example, for store operations, which do not return any useful information, just returning unit. The Store rule from Chapter 5 ignored the return value, but if we want a typing rule that does not use the semantic definition of a type, we need something to describe this return value, which we can do with True.

The following TrueI rule allows us to introduce True at any point.

Lemma 6.8 (TrueI).

$$\overline{\Pi \sqsubseteq \Pi * x_i : \mathsf{True} \triangleright \mathsf{unit}}$$

This rule and the True type in general can also be useful for applying typing rules like SumE,

which have a Π permission set in the input permission, representing any other permissions currently held. If we hold no other permissions, we can use TRUEI to introduce one to fit the right syntactic form to apply the rule. To eliminate these True types, we can always drop them, as with any permission set, using CNSQ and Lemma 4.14.

As an example of using the typing rules presented so far, we consider a simple program which multiplies a Val by 5. This program is represented by the ITree

$$y_i \leftarrow \mathsf{getNum}\ x_i; \mathsf{ret}\ (\mathsf{VNum}\ (5 \times y_i))$$

where x_i is the input variable, a Val, and y_i is an intermediate $\mathbb{N}$ value used to store the numeric value of x_i.

For this program to succeed, x_i must be numeric and not a pointer, which we can assert using input permission $x_i : \mathsf{Nat} \triangleright x_s$. Since Nat is defined as $\exists n : \mathbb{N}.\mathsf{eq}(\mathsf{VNum}\ n)$, the specification value x_s has type $\Sigma n : \mathbb{N}.\mathsf{Unit}$. As for the output type, we will use Nat. This type relates the output values of the implementation and specification programs, ensuring that they represent the same $\mathbb{N}$ value.

Using these inputs, Heapster extracts the specification

$$\mathsf{ret}\ \{5 \times x_s.1, \mathsf{unit}\},$$

which we can verify with the following typing derivation:

Example 6.9.

$$
\dfrac{
\dfrac{}{\begin{array}{l} x_i : \mathsf{eq}(\mathsf{VNum}\ x_s.1) \rhd x_s.2 \vdash \mathsf{getNum}\ x_i \\ \lessapprox \mathsf{ret\ unit} : \mathsf{eq}(x_s.1) \end{array}}\ \textsc{GetNum}
\quad
\dfrac{
\dfrac{
\dfrac{}{\begin{array}{l} \mathsf{VNum}\ (5 \times y_i) : \mathsf{Nat} \rhd \{5 \times x_s.1, \mathsf{unit}\} \\ \vdash \mathsf{ret}\ (\mathsf{VNum}\ (5 \times y_i)) \lessapprox \mathsf{ret}\ \{5 \times x_s.1, \mathsf{unit}\} : \mathsf{Nat} \end{array}}\ \textsc{Ret}
}{\begin{array}{l} \mathsf{VNum}\ (5 \times y_i) : \mathsf{eq}(\mathsf{VNum}\ (5 \times x_s.1)) \rhd \mathsf{unit} \\ \vdash \mathsf{ret}\ (\mathsf{VNum}\ (5 \times y_i)) \lessapprox \mathsf{ret}\ \{5 \times x_s.1, \mathsf{unit}\} : \mathsf{Nat} \end{array}}\ \textsc{ExI}
}{\begin{array}{l} y_i : \mathsf{eq}(x_s.1) \rhd \mathsf{unit} \vdash \mathsf{ret}\ (\mathsf{VNum}\ (5 \times y_i)) \\ \lessapprox \mathsf{ret}\ \{5 \times x_s.1, \mathsf{unit}\} : \mathsf{Nat} \end{array}}\ \textsc{EqCtx}
}{
\dfrac{\begin{array}{l} x_i : \mathsf{eq}(\mathsf{VNum}\ x_s.1) \rhd x_s.2 \vdash y_i \leftarrow \mathsf{getNum}\ x_i ; \mathsf{ret}\ (\mathsf{VNum}\ (5 \times y_i)) \\ \lessapprox \mathsf{ret}\ \{5 \times x_s.1, \mathsf{unit}\} : \mathsf{Nat} \end{array}}{\begin{array}{l} x_i : \mathsf{Nat} \rhd x_s \vdash y_i \leftarrow \mathsf{getNum}\ x_i ; \mathsf{ret}\ (\mathsf{VNum}\ (5 \times y_i)) \\ \lessapprox \mathsf{ret}\ \{5 \times x_s.1, \mathsf{unit}\} : \mathsf{Nat} \end{array}}\ \textsc{Cnsq}
}\ \textsc{Bind}
$$

Reading from the bottom up, we first use Cnsq along with ExE to eliminate the existential in the Nat type on x_i. By unfolding this type, we get the equality type telling us that x_i is a number with value $x_s.1$. Next, we apply the Bind rule to typecheck the two halves of the implementation and specification programs separately. To do this, however, we need to add a ret unit to the front of the specification program, which we can do via the monad laws. The first halves of the two programs are easily discharged with GetNum and has output type $\mathsf{eq}(x_s.1)$. The second halves quantify over values y_i and y_s to be related by $\mathsf{eq}(x_s.1)$, which means that $y_s = \mathsf{unit}$. Continuing upward, EqCtx is used to add the VNum and multiplication by 5 to the equality type. Next, ExI is used to "fold up" the equality type, resulting in the Nat type for implementation value $\mathsf{VNum}\ (5 \times y_i)$ and specification value $\{5 \times x_s.1, \mathsf{unit}\}$. Finally, at the top, rule Ret completes our proof, as the two values being returned match the values related by the type in the input permission set.

6.3. Pointer Types

In this section, we define pointer types, which differ in two ways from those defined in Chapter 5. First, they naturally must include specification values. Second, we add another parameter to the type, a natural number *offset* value. A pointer may be associated with multiple pointer permissions, as mentioned earlier with the example of linked lists. These offset values represent an offset

within a block of memory, so that a pointer can have multiple pointer permissions with different offsets to represent different memory cells. Linked lists and arrays will both use offsets. Linked lists will be presented in Section 6.5, and arrays will be presented in Section 6.4, along with the definitions and types for memory allocation and deallocation.

We now specialize our state type to Mem, and define memory operations for use in imperative programs. Instead of just being the ITree with a single load or store event as in Chapter 5, the definitions of load and store are slightly more complex when defined using Modify. We will informally describe the definitions of these ITrees here, and the full definitions can be found in the Coq formalization.

$$\mathsf{load} : \mathsf{Val} \to \mathsf{itree}\ \mathsf{E}_{\mathsf{Mem}}\ \mathsf{Val}$$

$$\mathsf{store} : \mathsf{Val} \to \mathsf{Val} \to \mathsf{itree}\ \mathsf{E}_{\mathsf{Mem}}\ \mathsf{Unit}$$

The load instruction uses a Modify event with the identity function, which does not update the memory and just returns it for inspection. Using this Mem, the instruction can then use read to look up the value to load from this memory and return it. The store instruction uses a Modify event to execute the store operation using write. However, in case of failure, it must still return a default value as the function in the Modify event has type Mem $\to$ Mem. To detect failure, we use the returned state from the Modify event to check if the write *would have succeeded* in the function. If load or store fail, for example due to an invalid pointer or trying to access memory using a VNum, then they may result in an error program instead.

To compute the final address given a pointer and offset, we use a function offset : Val $\to \mathbb{N} \to$ Val. The function application offset $p\ o$ increases the offset of p by o if it is a VPtr, and leaves p unchanged if it is a VNum. With this definition in hand, we can define pointer permission types, similar to the previous definition in Chapter 5. The only difference is that there, π_{read} and π_{write} were permissions over Mem, whereas here, they are over Mem $\times$ Unit, so we just ignore the

specification state of unit. Like before, rw is either *read* or *write*, and we refer to T—which has the type $\mathsf{PType}(\mathsf{Val}, A_s)$—as the content type.

$$ptr((rw, o) \mapsto T) : \mathsf{PType}(\mathsf{Val}, A_s)$$

$$x_i : ptr((rw, o) \mapsto T) \triangleright x_s \stackrel{\mathsf{def}}{=} \begin{cases} \bigsqcup_{v \in \mathsf{Val}} [\pi_{rw}(a, v)] * v : T \triangleright x_s & \text{if offset } x_i\ o = \mathsf{VPtr}\ a \\ \mathsf{False} & \text{if offset } x_i\ o = \mathsf{VNum}\ _ \end{cases}$$

One key part of how functional specifications differ from imperative implementation programs is that pointers are erased in the specification. To do this, pointers are related to the same specification value as the imperative value being pointed to. Thus, the pointers themselves become *transparent* in terms of how they are represented on the specification side. This is why the pointer type has the same type as the content type, $\mathsf{PType}(\mathsf{Val}, A_S)$. For example, if we have a pointer type with a content type of Nat, then that pointer type relates a Val on the implementation side to a dependent pair representing the natural number on the specification side. The specification value is the same as the value if the program on the implementation side did not involve pointers at all, but just had the Val the pointer points to.

With this definition, we can now present the typing rules for pointer types, shown in Figure 6.5. One new instruction used in these rules is isNull. The ITree isNull v checks whether a Val v is VNum 0, our representation for the null pointer.

$$\mathsf{isNull} : \mathsf{Val} \to \mathsf{itree}\ \mathsf{E_{Mem}}\ \mathbb{B}$$

$$\mathsf{isNull}\ (\mathsf{VNum}\ 0) \stackrel{\mathsf{def}}{=} \mathsf{true}$$

$$\mathsf{isNull}\ (\mathsf{VNum}\ _) \stackrel{\mathsf{def}}{=} \mathsf{error}$$

$$\mathsf{isNull}\ (\mathsf{VPtr}\ _) \stackrel{\mathsf{def}}{=} \mathsf{false}$$

If the value is a VNum but not 0, then it does not represent a valid pointer, and so results in an error computation.

$$\frac{}{\begin{array}{l} x_i : ptr((rw,o) \mapsto T) \triangleright x_s \vdash \mathsf{isNull}\ x_i \\ \lessapprox \mathsf{ret\ unit} : \mathsf{eq}(\mathsf{false}) \oslash (x_i : ptr((rw,o) \mapsto T) \triangleright x_s) \end{array}} \ \textsc{IsNull1}$$

$$\frac{}{x_i : \mathsf{eq}(\mathsf{VNum}\ 0) \triangleright x_s \vdash \mathsf{isNull}\ x_i \lessapprox \mathsf{ret\ unit} : \mathsf{eq}(\mathsf{true})} \ \textsc{IsNull2}$$

$$\frac{}{\begin{array}{l} x_i : ptr((rw,0) \mapsto \mathsf{eq}(y_i)) \triangleright \mathsf{unit} \vdash \mathsf{load}\ x_i \\ \lessapprox \mathsf{ret\ unit} : \mathsf{eq}(y_i) \oslash (x_i : ptr((rw,0) \mapsto \mathsf{eq}(y_i)) \triangleright \mathsf{unit}) \end{array}} \ \textsc{Load}$$

$$\frac{}{\begin{array}{l} x_i : ptr((write,0) \mapsto T)) \triangleright x_s \vdash \mathsf{store}\ x_i\ y_i \\ \lessapprox \mathsf{ret\ unit} : \mathsf{True} \oslash (x_i : ptr((write,0) \mapsto \mathsf{eq}(y_i)) \triangleright \mathsf{unit}) \end{array}} \ \textsc{Store}$$

$$\frac{}{x_i : ptr((write,o) \mapsto T) \triangleright x_s \sqsubseteq x_i : ptr((read,o) \mapsto T) \triangleright x_s} \ \textsc{PtrWeak}$$

$$\frac{}{x_i : ptr((read,o) \mapsto \mathsf{eq}(y_i)) \triangleright \mathsf{unit} \sqsubseteq \begin{array}{l} x_i : ptr((read,o) \mapsto \mathsf{eq}(y_i)) \triangleright \mathsf{unit} * \\ x_i : ptr((read,o) \mapsto \mathsf{eq}(y_i)) \triangleright \mathsf{unit} \end{array}} \ \textsc{ReadDup}$$

$$\frac{o_1 \geq o_2}{x_i : ptr((rw,o_1) \mapsto T) \triangleright x_s \sqsubseteq (\mathsf{offset}\ x_i\ o_2) : ptr((rw,o_1 - o_2) \mapsto T) \triangleright x_s} \ \textsc{PtrOff}$$

$$\frac{}{(x_i : ptr((rw,o) \mapsto \mathsf{eq}(y_i)) \triangleright \mathsf{unit}) * (y_i : T \triangleright y_s) \sqsubseteq x_i : ptr((rw,o) \mapsto T) \triangleright y_s} \ \textsc{PtrI}$$

$$\frac{\forall y_i,\ \Pi * (x_i : ptr((rw,o) \mapsto \mathsf{eq}(y_i)) \triangleright \mathsf{unit}) * (y_i : T_1 \triangleright x_s) \vdash t_i \lessapprox t_s : T_2}{\Pi * (x_i : ptr((rw,o) \mapsto T_1) \triangleright x_s) \vdash t_i \lessapprox t_s : T_2} \ \textsc{PtrE}$$

Figure 6.5: Typing rules for pointer types.

The IsNull1 rule handles the case where the Val x_i is not null, because it has a pointer type. The output type reflects this fact by stating that the result of isNull is false, but also must maintain the pointer type on x_i from the input permission. Since this is a permission set, not a permission type, it is added to the output type using $\oslash$. IsNull2 handles the other case where x_i is null. Unlike IsNull1, the output type does not need to maintain the equality permission in the input permission, since equality permissions are duplicable. Before using this typing rule, the equality permission could have been duplicated and "stored" to use after IsNull2 using Frame.

The Load rule differs slightly from the analogous rule in Chapter 5. Instead of allowing any content type in the input pointer permission, this rule supports only equality types. This is due to our usage of permission types, rather than a function for the output permission. We cannot express that x_i should now point to a value equal to the return value of the load without using the fact that types are semantically defined as functions, as was done in Chapter 5. Rather, we use equality types for the content types. When used in conjunction with the PtrI and PtrE rules described below, they are just as expressive as the version in the previous chapter. The Store rule, on the other hand, is very similar to the one in Chapter 5. The only difference is that it uses True in its output type, since store does not return any useful value. The output types of both these rules use $\oslash$ to maintain the input permission set, just like IsNull1. This is necessary as pointer permissions may not be duplicable and would need to be preserved by these typing rules if they are to be used later on. Since the type system aims to erase pointers in the extracted specifications, the extracted programs for these rules are ret unit.

The PtrWeak rule weakens a pointer-write type into a pointer-read type, lifting the result we had in Lemma 5.9. Once we have a pointer-read type, the ReadDup rule allows us to duplicate it, using the result from Lemma 5.10. ReadDup would also allow us to create an alternative rule for load when the type for x_i is known to be a duplicable pointer-read type, where we do not have to maintain the pointer permission in the output type, like with the IsNull2 rule.

PtrOff allows us to use pointer types where the offset is not 0. The previous Load and Store rules only work with such types, but PtrOff allows a pointer type with non-zero offset to be

converted to one with offset 0. This modifies the value that holds the pointer type, changing its value using offset.

Finally, the pair of rules PTRI and PTRE introduce and eliminate non-equality content types from pointer types. The introduction rule allows for a content type to be created from an equality type, if the value that the pointer points to has some other type. This is similar to CAST, which works on regular equality types, not those inside a permission type. PTRI can also be thought of as eliminating the value that the pointer points to, hiding it inside the pointer type. Conversely, PTRE can be thought of as introducing the value that the pointer points to, y_i. With access to this value, this rule allows us to "move" the content type T_1 out of the pointer type by applying it to y_i. Once T_1 is outside the pointer type, it can then be manipulated using other typing rules.

As alluded to in the discussion of Example 5.20, we can now redo the example with more informative output permission types using PTRI and PTRE to manipulate content types. Recall the implementation program, where we have a value p:

$$p' \leftarrow \mathsf{load}\ p;$$
$$\mathsf{store}\ p'\ (\mathsf{VNum}\ 1)$$

The specification program is just ret unit, since pointer operations are erased in specifications. For the input permission set, we will require that p point to another pointer which can be written to, just as in example 5.20. The input permission set that expresses this is

$$p : ptr((read, 0) \mapsto ptr((write, 0) \mapsto \mathsf{True})) \triangleright \mathsf{unit}.$$

For the output permissions, we use

$$\mathsf{True} \oslash p : ptr((read, 0) \mapsto ptr((write, 0) \mapsto \mathsf{eq}(\mathsf{VNum}\ 1))) \triangleright \mathsf{unit},$$

expressing the fact that p now points to a pointer that in turn points to a value of 1. Unlike in

Chapter 5, we will skip the possibility of circularity in memory. As discussed in Section 1.2, this possibility of circularity is outside the scope of programs handled by Heapster, since it would greatly increase the complexity of automated typechecking. It would still be possible to manually typecheck the program with the presence of circularity in the theory of Heapster, for which the typing derivation would be similar to that of Example 5.21.

Example 6.10. For space reasons, we will split up the typing derivation, going from the bottom up.

$$\dfrac{\dfrac{\text{Part 1} \qquad\qquad \text{Part 2}}{\begin{array}{l} p : ptr((read,0) \mapsto \mathsf{eq}(p')) \triangleright \mathsf{unit} * p' : ptr((write,0) \mapsto \mathsf{True}) \triangleright \mathsf{unit} \vdash p' \leftarrow \mathsf{load}\ p; \mathsf{store}\ p'\ (\mathsf{VNum}\ 1) \\ \lessapprox \mathsf{ret\ unit} : \mathsf{True} \oslash p : ptr((read,0) \mapsto ptr((write,0) \mapsto \mathsf{eq}(\mathsf{VNum}\ 1))) \triangleright \mathsf{unit} \end{array}}\ \textsc{Bind}}{\begin{array}{l} p : ptr((read,0) \mapsto ptr((write,0) \mapsto \mathsf{True})) \triangleright \mathsf{unit} \vdash p' \leftarrow \mathsf{load}\ p; \mathsf{store}\ p'\ (\mathsf{VNum}\ 1) \\ \lessapprox \mathsf{ret\ unit} : \mathsf{True} \oslash p : ptr((read,0) \mapsto ptr((write,0) \mapsto \mathsf{eq}(\mathsf{VNum}\ 1))) \triangleright \mathsf{unit} \end{array}}\ \textsc{PtrE}$$

Part 1:

$$\dfrac{\dfrac{}{\begin{array}{l} p : ptr((read,0) \mapsto \mathsf{eq}(p')) \triangleright \mathsf{unit} \vdash \mathsf{load}\ p \\ \lessapprox \mathsf{ret\ unit} : \mathsf{eq}(p') \oslash p : ptr((read,0) \mapsto \mathsf{eq}(p')) \triangleright \mathsf{unit} \end{array}}\ \textsc{Load}}{\begin{array}{l} p : ptr((read,0) \mapsto \mathsf{eq}(p')) \triangleright \mathsf{unit} * p' : ptr((write,0) \mapsto \mathsf{True}) \triangleright \mathsf{unit} \vdash p' \leftarrow \mathsf{load}\ p \\ \lessapprox \mathsf{ret\ unit} : \mathsf{eq}(p') \oslash p : ptr((read,0) \mapsto \mathsf{eq}(p')) \triangleright \mathsf{unit} \oslash p' : ptr((write,0) \mapsto \mathsf{True}) \triangleright \mathsf{unit} \end{array}}\ \textsc{Frame}$$

Part 2:

$$\dfrac{\dfrac{\dfrac{}{\begin{array}{l} p'' : ptr((write,0) \mapsto \mathsf{True}) \triangleright \mathsf{unit} \vdash \mathsf{store}\ p''\ (\mathsf{VNum}\ 1) \\ \lessapprox \mathsf{ret\ unit} : \mathsf{True} \oslash p'' : ptr((write,0) \mapsto \mathsf{eq}(\mathsf{VNum}\ 1)) \triangleright \mathsf{unit} \end{array}}\ \textsc{Store}}{\begin{array}{l} p : ptr((read,0) \mapsto \mathsf{eq}(p'')) \triangleright \mathsf{unit} * p'' : ptr((write,0) \mapsto \mathsf{True}) \triangleright \mathsf{unit} \vdash \mathsf{store}\ p''\ (\mathsf{VNum}\ 1) \\ \lessapprox \mathsf{ret\ unit} : \mathsf{True} \oslash p : ptr((read,0) \mapsto \mathsf{eq}(p'')) \triangleright \mathsf{unit} * p'' : ptr((write,0) \mapsto \mathsf{eq}(\mathsf{VNum}\ 1)) \triangleright \mathsf{unit} \end{array}}\ \textsc{Frame}}{\begin{array}{l} p'' : \mathsf{eq}(p') \triangleright \mathsf{Unit} \oslash p : ptr((read,0) \mapsto \mathsf{eq}(p')) \triangleright \mathsf{unit} \oslash p' : ptr((write,0) \mapsto \mathsf{True}) \triangleright \mathsf{unit} \vdash \mathsf{store}\ p''\ (\mathsf{VNum}\ 1) \\ \lessapprox \mathsf{ret\ unit} : \mathsf{True} \oslash p : ptr((read,0) \mapsto ptr((write,0) \mapsto \mathsf{eq}(\mathsf{VNum}\ 1))) \triangleright \mathsf{unit} \end{array}}\ \textsc{Cnsq}$$

The first rule we apply is PtrE, to move the content type out of the pointer type. This introduces

the value that p points to, which we name p'. This is necessary now, rather than after applying BIND, because the intermediate type for BIND needs to have p' in scope to refer to it. This intermediate type includes $\mathsf{eq}(p')$ so we can link the output type we get from the first premise of the BIND to the input permissions for the second premise.

Part 1 of the derivation typechecks the first $\mathsf{load}\ p$ portion of the implementation program. The type for p' is not needed to apply LOAD, but unlike in Example 5.20, we cannot just drop it, since it is used in part 2 of the proof. Instead, so we remove it only for the application of LOAD using FRAME, and the resulting types are in exactly the right format to apply LOAD.

Part 2 is more complicated. The BIND rule quantifies over any intermediate value returned by the load in the first portion of the program, so we have another value p'' that we know is equal to p' through an equality type. We use CNSQ to unify the two values, p' and p'', in the input permission. This involves using PERMSE to unfold the definition of $\oslash$, and EQDUP to duplicate the equality type, since we will need one for each remaining use of p' in the input permission. For the portion $p : ptr((read, 0) \mapsto \mathsf{eq}(p')) \triangleright \mathsf{unit}$, we use EQSYM on one of the equality types then PTRI to update this pointer's content type. For the second portion $p' : ptr((write, 0) \mapsto \mathsf{True}) \triangleright \mathsf{unit}$, we use CAST, thus removing all uses of p'. This use of CNSQ also strengthens the output type to include p'', which will be needed for our application of STORE later on. To show that the change in output type is valid, we use PERMSE and PERMSI to unfold and later recreate the $\oslash$, and apply PTRI. Finally, we remove the types for p from both the input and output types using FRAME, and we then have the exact types to conclude by applying STORE.

This typing derivation may not seem to give us anything useful on the specification side. Though the specification program does not do anything and the types do not relate to any useful specification values, this derivation *is* still useful for extracting specifications. The key is that we now have an output type with information about the functionality of the memory operations in the implementation program. As we saw in Example 6.9, an equality type like this can be useful for relating implementation values to specification values. If we follow the pointer that p points to later in the program, the types will contain the information that the value is $\mathsf{VNum}\ 1$, and Heapster could

extract a specification that immediately returns the value 1, without any memory operations.

6.4. Array Types

This section introduces types for describing arrays, rather than single memory cells. These array types are essential for describing the typing rules for malloc and free, which manage blocks of memory using arrays.

Array types relate imperative arrays of length l to vectors of length l, where each element of the arrays and vectors are pairwise related using the array type's content type. We will write $\mathsf{Vect}\ T\ l$ for the type of vectors of Ts with length l. We use $\langle x, y, z \rangle$ to denote a vector with elements x, y, z, the notation $v_1 +\!\!+ v_2$ for v_1 and v_2 appended together, and $v[i]$ for the ith element of a vector.

Semantically, array types are defined using the iterated separating conjunction of l pointer types with a consistent content type $T : \mathsf{PType}(\mathsf{Val}, A_s)$ for each pointer type:

$$\mathsf{arr}((rw, o, l) \mapsto T) : \mathsf{PType}(\mathsf{Val}, \mathsf{Vect}\ A_s\ l)$$

$$x_i : \mathsf{arr}((rw, o, l) \mapsto T) \triangleright x_s \stackrel{\mathsf{def}}{=} \mathop{\ast}_{0 \leq i < l} x_i : ptr((rw, o + i) \mapsto T) \triangleright x_s[i]$$

As with pointer types, rw is either *read* or *write*, and o is a natural number offset in the memory block.

This type could also be defined as the $\star$ of pointer types, but the specification value would then be combined as pairs. Instead of a vector on the specification side, we would instead have nested pairs—an equivalent type, but much less convenient to work with.

The typing rules for array types are shown in Figure 6.6.

The first two rules, PtrArr and ArrPtr, convert between pointer types and array types where the length of the array is one. This is how we use values in arrays, by converting them to pointers and then using our previously-defined pointer rules. We also need rules for obtaining these array types of length one from larger array types. This is done by the next rule, ArrSplit. This rule

$$\frac{}{x_i : ptr((rw, o) \mapsto T) \triangleright x_s \sqsubseteq x_i : \mathsf{arr}((rw, o, 1) \mapsto T) \triangleright \langle x_s \rangle} \text{ PtrArr}$$

$$\frac{}{x_i : \mathsf{arr}((rw, o, 1) \mapsto T) \triangleright x_s \sqsubseteq x_i : ptr((rw, o) \mapsto T) \triangleright x_s[0]} \text{ ArrPtr}$$

$$\frac{\forall x_{s_1}, x_{s_2}, \begin{array}{l} \Pi * x_i : \mathsf{arr}((rw, o, l') \mapsto T_1) \triangleright x_{s_1} * \\ x_i : \mathsf{arr}((rw, o + l', l - l') \mapsto T_1) \triangleright x_{s_2} \end{array} \vdash t_i \lessapprox f_s\ x_{s_1}\ x_{s_2} : T_2}{\Pi * x_i : \mathsf{arr}((rw, o, l) \mapsto T_1) \triangleright x_s \vdash t_i \lessapprox \mathsf{trySplit}\ x_s\ l'\ f_s : T_2} \text{ ArrSplit}$$

$$\frac{}{\begin{array}{l} x_i : \mathsf{arr}((rw, o, l') \mapsto T) \triangleright x_{s_1} * \\ x_i : \mathsf{arr}((rw, o + l', l) \mapsto T) \triangleright x_{s_2} \end{array} \sqsubseteq x_i : \mathsf{arr}((rw, o, l + l') \mapsto T) \triangleright (x_{s_1} +\!\!+ x_{s_2})} \text{ ArrCombine}$$

Figure 6.6: Typing rules for arrays.

can be applied any time to split an array type, by extracting a trySplit program on the specification side. The program trySplit $v\ l'\ f$ *attempts* to split the vector v at index l', and passes the two halves of the split vector to a continuation f, which we can proceed with in the typechecking process. To typecheck an array access, we would have to use this rule at most twice to obtain an array type of length one, at which point we can use ArrPtr.

Crucially, trySplit can fail and result in error if we try to split at an invalid index, one that is greater than the length of the vector. This is the only rule which introduces error on the specification side, aside from Err. This acts as a dynamic check in the specification program rather than requiring this check be statically performed during typechecking, as a precondition on the typing rule. This is because the bounds check we are trying to perform is in general undecidable statically, and we do not want to have to fully give up by using Err when the validity of the check cannot be determined during typechecking. Instead, this rule *shifts* the burden of the bounds check to the specification, so the user can later prove that it cannot occur in the specification, rather than having to leave this task to the typechecker.

The final array rule is ArrCombine, which combines two array types. While this may not seem useful right now, we will soon see the Free rule for deallocating memory, which does require a single array type for the block of memory we are trying to free, rather than several disjoint array types.

Now that we can manipulate array types, we can finally discuss manual memory management.

We first introduce the malloc and free instructions:

$$\mathsf{malloc} : \mathbb{N} \to \mathsf{itree}\ \mathsf{E}_{\mathsf{Mem}}\ \mathsf{Val}$$
$$\mathsf{free} : \mathsf{Val} \to \mathsf{itree}\ \mathsf{E}_{\mathsf{Mem}}\ \mathsf{Unit}$$

The malloc instruction takes the size of the block to allocate as an argument, and appends a block of that size to the Mem's list of blocks. This operation never fails, as our memory model is unbounded. It then returns a pointer value to the front of the newly allocated block.

The free instruction is more involved. Calling free on a VNum or a VPtr that does not refer to the first value of a block will result in error—deallocating only a portion of a block is not allowed. If the argument is a valid pointer to the front of a block, then the contents of the block are erased, by removing them from the partial function that represents the values in the block.

While having a write-array type for the entire block would permit us to free this block (since we would have write access to every element in the block, and free does not remove the block itself, but every element in the block), we do not yet have any permission definitions that would allow for the state change performed by malloc. To remedy this, we first must define two new permissions used in the typing rules for malloc and free: π_{alloc} and π_{block}.

First, we define $\pi_{\mathsf{alloc}}(b)$, which represents exclusive ownership of *all* unallocated blocks, those numbered b or higher. In a program where no memory is allocated yet, $\pi_{\mathsf{alloc}}(0)$ would be held, giving us ownership of the entire Mem.

$$\begin{aligned}\pi_{\mathsf{alloc}}(b) \overset{\mathsf{def}}{=} (\{\, (s_1, s_2) \mid & \text{ the number of blocks in } s_1 \text{ and } s_2 \text{ are the same, and} \\ & \text{blocks} \geq b \text{ do not change between } s_1 \text{ and } s_2 \,\}, \\ & \{\, (s_1, s_2) \mid \text{blocks} < b \text{ do not change between } s_1 \text{ and } s_2 \,\}, \\ & \{\, s \mid \text{there are } b \text{ blocks in } s \,\})\end{aligned}$$

The rely requires that nobody else modify block b and above in memory, which represent the

unallocated blocks. The guarantee permits the holder of this permission to modify those blocks arbitrarily, as long as they do not change existing allocated blocks below b. The precondition formalizes the fact that the argument b should represent the current number of allocated blocks.

We then define the Π_{alloc} permission set to existentially quantify over the number of currently-allocated blocks, allowing us to write generic typing rules without having to know exactly how many allocations have already occurred.

$$\Pi_{\mathsf{alloc}} \stackrel{\mathsf{def}}{=} \bigsqcup_{b \in \mathbb{N}} [\pi_{\mathsf{alloc}}(b)]$$

While this permission is sufficient to typecheck malloc, we will define another permission which we will also use in its typing rule, related to deallocation. The permission $\pi_{\mathsf{block}}(a, n)$ encapsulates the precondition that that the address a points to the front of a block of size n.

$$\begin{aligned} \pi_{\mathsf{block}}(a, n) \stackrel{\mathsf{def}}{=} (&\{ (s_1, s_2) \mid \text{the size of the block that } a \text{ points to is the same in both } s_1 \text{ and } s_2 \}, \\ &=, \\ &\{ s \mid a \text{ points to the front of a block of size } n \text{ in } s \}) \end{aligned}$$

The $\pi_{\mathsf{block}}(a, n)$ permission, with its guarantee of $=$, does not give us the ability to make more changes—pointer-write permissions to every value in the block already suffice to typecheck the deallocation of a block. Rather, this permission tells us that a points to the front of a block, which is what we need to be able to safely use it with free. It also stores the size of the block, n, so we know exactly what array type we need to hold—the length of the array—before we can use free. As a side effect of the precondition having to be stable under the rely, $R_{\pi_{\mathsf{block}}(a,n)}$ also prevents other code from changing the size of the block.

Using π_{block}, we can then define the block permission type to attach this permission to an imper-

ative value:

$$\mathsf{block}(n) : \mathsf{PType}(\mathsf{Val}, \mathsf{Unit})$$

$$x_i : \mathsf{block}(n) \triangleright \mathsf{unit} \stackrel{\mathsf{def}}{=} \begin{cases} [\pi_{\mathsf{block}}(a, n)] & \text{if } x_i = \mathsf{VPtr}\ a \\ \mathsf{False} & \text{if } x_i = \mathsf{VNum}\ _ \end{cases}$$

Now we are ready to present the typing rule for malloc:

Theorem 6.11 (MALLOC)**.**

$$\frac{}{\begin{array}{l} x_i : \mathsf{eq}(\mathsf{VNum}\ n) \triangleright \mathsf{unit} * \Pi_{\mathsf{alloc}} \vdash \mathsf{malloc}\ x_i \\ \lessapprox \mathsf{ret}\ (\langle \mathsf{unit}, \ldots \rangle, \mathsf{unit}) : \mathsf{arr}((W, 0, n) \mapsto \mathsf{True}) \star \mathsf{block}(n) \oslash \Pi_{\mathsf{alloc}} \end{array}}$$

This rule typechecks a malloc as long as we hold the allocation permission Π_{alloc}, which is returned back to us in the output type for future allocations. The output type gives the pointer value returned by the malloc two separate types, combined using the $\star$ connective. The first is the array type, where the content type is the vacuous True. The second is the block type, which will be required to use the FREE typing rule for free. This output type forces the specification program to return a vector of n unit values representing the contents of the array, paired with an additional unit due to the block permission. This rule effectively moves one block's worth of permissions from the Π_{alloc} permission set and gives it to us as an array type. The use of the permission set means that the Π_{alloc} set does not change outwardly, since it keeps the number of currently-allocated blocks hidden in its definition.

Theorem 6.12 (FREE)**.**

$$\frac{}{x_i : \mathsf{arr}((W, 0, n) \mapsto \mathsf{True}) \star \mathsf{block}(n) \triangleright x_s \vdash \mathsf{free}\ x_i \lessapprox \mathsf{ret}\ \mathsf{unit} : \mathsf{True}}$$

The FREE rule requires us to hold the output types from the MALLOC rule, including the True

content type for the array, and not a stronger one. Thankfully, this is possible to achieve. Even if the content changed between allocation and deallocation, we can always introduce a new True type using TrueI and drop the other content type. This is not a straightforward process, though—any change to the content type of the array has to be done one element at a time, using the array typing rules to split the array, convert to pointer types and back, and then recombine them into a single array type.

One difference between this type system that we are defining and the Heapster tool is the Π_{alloc} permission set. This construct is not in the Heapster tool, but it *is* necessary to prove the Malloc rule, since we must have a permission that allows the allocation to occur. In Heapster, this permission to allocate new memory is implicitly present at all points during typechecking, which is sound since it is not duplicated, as the tool only deals with single-threaded code.

6.5. Recursive Types

Recursive permission types allow us to relate recursive data structures on the implementation side to recursive data types on the specification side. To match the implementation in Heapster, the recursive types in this type system will be iso-recursive, and will require recursive types to be explicitly folded and unfolded. A recursive type will be written as $\mu X.\, T$, where X is free in T and refers to the recursive occurrence of the overall type. Alternatively, if we have the generator function $\lambda X.\, T$ defined separately as F, it can be written as μF.

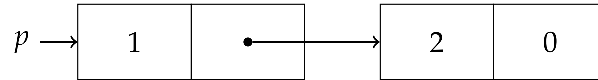
Before we define the behavior and semantics of the recursive permission type, we must introduce some of the building blocks we need. First, we define the lattice operations for permission types, $\mathsf{PType}(A, B)$, using the pointwise lifting of the lattice operations for permission sets, $\mathsf{Perms}_{S_i \times S_s}$. We will again overload the notation for these operations, like $\sqsubseteq$ and $\sqcup$. Then, we can use the Knaster-Tarski theorem [Tar55] to construct the greatest fixed point of monotone functions $F : \mathsf{PType}(A, B) \to \mathsf{PType}(A, B)$.

$$\nu F \stackrel{\mathsf{def}}{=} \bigsqcup \{\, T \mid T \sqsubseteq F\ T \,\}$$

We use the greatest fixed point because greater elements of the lattice represent *weaker* permis-

sions.[6] We wish to use the weakest permission possible that satisfies the fixed point constraints we want, to avoid introducing extra, undesired capabilities to programs.

Now that we can define greatest fixed points, we can work towards defining the recursive permission type in terms of it. To guide us towards a definition, we consider the example of a linked list. The following diagram represents the list $[1, 2]$:



Recall that we use the number 0 as a null pointer, as in the isNull instruction.

We wish to write a type for a pointer pointing to a linked list using our existing permission types and an assumed recursive type $\mu X.\, T$. Each node of the linked list has two values—one for the actual value of the element, and another for the pointer to the next node. This suggests two types conjoined with the $\star$ connective. We use $\star$ rather than $\otimes$ since there is only one implementation value: a pointer which we know points to a value and, when incremented to the next pointer address, a pointer to another linked list. We must also handle the base case, where the pointer to the linked list is null. Putting this all together, we get the following recursive type:

$$\mu X.\, \mathsf{eq}(\mathsf{VNum}\ 0) \vee (ptr((rw, 0) \mapsto \mathsf{Nat}) \star ptr((rw, 1) \mapsto X))$$

An imperative value with this type is either the null pointer or it points to a numeric value and another value of this recursive type. Note that the two values in a single node must be in the same block of memory, due to the use of the offset value, but different nodes can be in different blocks.

As for the specification type, intuitively, recursive permission types should relate pointers on the implementation side to elements of recursive types on the specification side. In this case, the specification type should be $\mathsf{List}\ \Sigma_{n:\mathbb{N}}\mathsf{Unit}$, due to the use of Nat for the values in the linked list. But looking at our definition above, its type is roughly $\mathsf{Unit} + \Sigma_{n:\mathbb{N}}\mathsf{Unit} \times X$, where X is a recursive

[6] In the paper by He et al. [He+21], the lattice is inverted compared to the presentation in this dissertation. Correspondingly, the semantic definition of the recursive type is given by the *least* fixed point, though it is identical to this definition.

occurrence of this type. This is the standard way of *defining* the list type using recursive types, but we will need some way of going between this representation and the closed representation of $\mathsf{List}\ \Sigma_{n:\mathbb{N}}\mathsf{Unit}$.

More generally, specification values will be values of type Z, where Z is a fixed point of the generator function G. We also require functions $\mathsf{fold} : G\ Z \to Z$ and $\mathsf{unfold} : Z \to G\ Z$ that form an isomorphism. As we will see, G is formally connected to the definition of the recursive type. For example, if the specification values should be lists of type A, similar to above, then $G \stackrel{\mathsf{def}}{=} \lambda Z.\, \mathsf{Unit} + (A \times Z)$. The type $\mathsf{List}\ A$ is a fixed point of this function, and we can then define the fold and unfold functions:

$$\begin{aligned}
\mathsf{fold} &: (\mathsf{Unit} + A \times \mathsf{List}\ A) \to \mathsf{List}\ A \\
\mathsf{fold}\ (\mathsf{inl}\ \mathsf{unit}) &\stackrel{\mathsf{def}}{=} [] \\
\mathsf{fold}\ (\mathsf{inr}\ (h, t)) &\stackrel{\mathsf{def}}{=} h :: t
\end{aligned}$$

$$\begin{aligned}
\mathsf{unfold} &: \mathsf{List}\ A \to (\mathsf{Unit} + A \times \mathsf{List}\ A) \\
\mathsf{unfold}\ [] &\stackrel{\mathsf{def}}{=} \mathsf{inl}\ \mathsf{unit} \\
\mathsf{unfold}\ (h :: t) &\stackrel{\mathsf{def}}{=} \mathsf{inr}\ (h, t)
\end{aligned}$$

It can then be proven that these functions form an isomorphism, that $\mathsf{unfold}\ (\mathsf{fold}\ x) = x$ and $\mathsf{fold}\ (\mathsf{unfold}\ x) = x$.

Now, looking at our example above at the function that generates the recursive type,

$$\lambda X.\, \mathsf{eq}(\mathsf{VNum}\ 0) \vee (ptr((rw, 0) \mapsto \mathsf{Nat}) \star ptr((rw, 1) \mapsto X)),$$

its type is

$$\mathsf{PType}(\mathsf{Val}, \mathsf{List}\ \Sigma_{n:\mathbb{N}}\mathsf{Unit}) \to \mathsf{PType}(\mathsf{Val}, \mathsf{Unit} + \Sigma_{n:\mathbb{N}}\mathsf{Unit} \times \mathsf{List}\ \Sigma_{n:\mathbb{N}}\mathsf{Unit}).$$

$$\frac{}{x_i : F\ (\mu F) \triangleright x_s \sqsubseteq x_i : \mu F \triangleright \mathsf{fold}\ x_s}\ \textsc{Fold} \qquad \frac{}{x_i : \mu F \triangleright x_s \sqsubseteq x_i : F\ (\mu F) \triangleright \mathsf{unfold}\ x_s}\ \textsc{Unfold}$$

Figure 6.7: Typing rules for recursive types.

In general, the return type of the generator function uses the "unfolded" type $G\ Z$. The function's type is

$$\mathsf{PType}(A, Z) \to \mathsf{PType}(A, G\ Z).$$

This is how the definition of the recursive type is linked to the type of its specification value. The body of the recursive type determines the definition of G, and we obtain Z by finding a fixed point of G.

Finally, the overall type of the recursive type in our example is

$$\mathsf{PType}(\mathsf{Val}, \mathsf{List}\ \Sigma_{n:\mathbb{N}}\mathsf{Unit}).$$

The type $\mathsf{List}\ \Sigma_{n:\mathbb{N}}\mathsf{Unit}$ is the fixed point Z in the definitions above. In general, the overall type of a recursive type is

$$\mathsf{PType}(A, Z).$$

Now, we can define the recursive type μF, where $F : \mathsf{PType}(A, Z) \to \mathsf{PType}(A, G\ Z)$, using the greatest fixed point combinator. Since ν must be applied to a function where the input and output types are the same, we compose the unfold operation with F to make the types match.

$$\begin{aligned} &\mu F : \mathsf{PType}(A, Z) \\ &\mu F \overset{\mathsf{def}}{=} \nu(\lambda T, x_i, x_s.\, x_i : F\ T \triangleright \mathsf{unfold}\ x_s) \end{aligned}$$

With the recursive type finally defined, we can present the typing rules for recursive types, which are found in figure 6.7. There are only two rules, Fold and Unfold, which fold and unfold the type, along with the specification value. For an example of recursive types in action, Section 6.6

$$\frac{}{\mathsf{True} \sqsubseteq x_i : (\mu X.\, \mathsf{eq}(x_i) \vee (T \star ptr((rw, o) \mapsto X))) \triangleright \langle\rangle}\ \textsc{ReflR}$$

$$\frac{}{\begin{array}{l} x_i : (\mu X.\, \mathsf{eq}(y_i) \vee (T \star ptr((rw, o) \mapsto X))) \triangleright x_s * \\ y_i : (\mu X.\, \mathsf{eq}(z_i) \vee (T \star ptr((rw, o) \mapsto X))) \triangleright y_s \end{array} \sqsubseteq x_i : (\mu X.\, \mathsf{eq}(z_i) \vee (T \star ptr((rw, o) \mapsto X))) \triangleright x_s + y_s}\ \textsc{TransR}$$

Figure 6.8: Typing rules for reachability types.

will include uses of recursive types.

Reachability types are a specialization of recursive types for the case of linked lists where we end the linked list at a value y, rather than a null pointer. This is useful for reasoning about portions of linked lists from the starting pointer down to another pointer y, rather than the entire linked list, and is analogous to the notion of list segments in separation logic [BCO05]. Reachability types are named as such to capture the notion that the end value y is reachable in 0 or more steps from the value holding the reachability type.

Formally, a reachability type is a type of the form

$$\mu X.\, \mathsf{eq}(y) \vee (T \star ptr((rw, o) \mapsto X))$$

The base case of the linked list is represented by the equality type. Otherwise, there is a pointer to the next node, represented by the pointer type with the recursive content type. The value stored at each node of the linked list is represented by the type $T : \mathsf{PType}(\mathsf{Val}, A)$. Then, this reachability type has type $\mathsf{PType}(\mathsf{Val}, \mathsf{List}\ A)$, similar to our example above. For lists, we will overload the notation for vectors, writing a list as $\langle a, b, \ldots \rangle$ and the concatenation of two lists as $l_1 + l_2$. For this special case of reachability types, we can write reflexivity and transitivity rules due to its straight-line structure, shown in Figure 6.8.

The ReflR rule shows that x_i is always reachable from itself. Since it is reachable from itself in 0 steps, this type relates the value to an empty list on the specification side. The TransR rule represents transitivity of reachability types, allowing us to combine reachability types, which appends their specification values.

6.6. A Bigger Example

To illustrate the use of some of the typing rules we have presented in this chapter, we can type-check the following imperative program which computes the length of a linked list pointed to by a value *ptr*.

$$\begin{array}{l}
\mathsf{iter}\ (\lambda\,(v : \mathsf{Val}, p : \mathsf{Val}). \\
\quad n \leftarrow \mathsf{getNum}\ v; \\
\quad b \leftarrow \mathsf{isNull}\ p; \\
\qquad \mathsf{if}\ b \\
\qquad \mathsf{then}\ \mathsf{ret}\ (\mathsf{inr}\ (\mathsf{VNum}\ n)) \\
\qquad \mathsf{else}\ p' \leftarrow \mathsf{load}\ (\mathsf{offset}\ p\ 1); \mathsf{ret}\ (\mathsf{inl}\ (\mathsf{VNum}\ (n+1), p'))) \\
(\mathsf{VNum}\ 0, ptr)
\end{array}$$

This program uses iter to loop, changing a counter variable v and a pointer p between iterations. These two variables start at values VNum 0 and *ptr* respectively. The body of the iter first gets the numerical value n from v, then checks if we are at the end of the linked list. If we are at the end of the list, we return the counter value n and stop iterating. Otherwise, if p is not null, then we get the pointer to the next node of the linked list, and continue iterating with an incremented counter and the next node pointer.

The functional specification for this implementation program is the following program, given a

list *lst*, containing values of some type A:

$$
\begin{array}{l}
\mathsf{iter}\ (\lambda\ (i : \Sigma_{n:\mathbb{N}}\mathsf{Unit}, l : \mathsf{List}\ A). \\
\quad \mathsf{case\ unfold}\ l\ \mathsf{of} \\
\quad\quad \lambda_ : \mathsf{Unit}.\, \mathsf{ret}\ (\mathsf{inr}\ i) \\
\quad\quad \lambda(h : A, t : \mathsf{List}\ A).\, \mathsf{ret}\ (\mathsf{inl}\ (\{i.1 + 1, \mathsf{unit}\}, t))) \\
(\{0, \mathsf{unit}\}, lst)
\end{array}
$$

The specification program is similar in structure to the implementation one, also using iter with a counter variable i and a list variable l. Then to check whether we are done iterating, we perform a case analysis on the unfold of l. In the base case, where this value is a Unit, we return the counter value i and stop iterating. In the recursive case, we continue iterating with an incremented counter and the rest of the list t.

Now we can show that these two programs are related via the typing judgment. One useful type we will use is the recursive type for linked lists defined in Section 6.5, which we will name ListT, which we define in terms of a *rw* which is *read* or *write*, and a permission type $T : \mathsf{PType}(\mathsf{Val}, A)$.

$$
\begin{array}{l}
\mathsf{ListT}_{rw,T} : \mathsf{PType}(\mathsf{Val}, \mathsf{List}\ A) \\
\mathsf{ListT}_{rw,T} \stackrel{\mathsf{def}}{=} \mu X.\, \mathsf{eq}(\mathsf{VNum}\ 0) \vee (ptr((rw, 0) \mapsto T) \star ptr((rw, 1) \mapsto X))
\end{array}
$$

The *rw* argument applies to all the pointer permissions in the recursive type, and the type T describes the value held at each node, relating each implementation value to a specification value.

The input permission set will be $ptr : \mathsf{ListT}_{read,T} \triangleright lst$, using the recursive type to relate the pointer to a linked list, *ptr*, and the functional list, *lst*. Since the program does not need write access to the pointers, we use the *read* tag for our pointer permissions. Our programs do not actually use the value at each node of the linked list, so we keep the type that describes them, T, abstract. We could use, for instance, Nat in place of T to describe linked lists of numerical Vals on the implementation

side and lists of $\Sigma_{n:\mathbb{N}}\mathsf{Unit}$ on the specification side. As for the output type, we use Nat, which ensures that the output values of both programs represent equal numbers.

Since the example will be too large to present diagrammatically as we have done so far, we will describe it informally, adding in snippets of what the proof derivation looks like at important or interesting spots. The full typing derivation can be found in the Coq formalization. While we will mention notable uses of typing rules here, some will be skipped.

Initially, we would like to use ITER to relate the two programs formed by iters by relating their loop bodies. To do this, we first need types relating their initial values. We already have the type $\mathsf{ListT}_{read,T}$ to relate *ptr* and *lst*, but we need a type to relate the two counter variables. We can create such a type using EQREFL and EXI to obtain $\mathsf{VNum}\ 0 : \mathsf{Nat} \triangleright \{0, \mathsf{unit}\}$. Then, we can combine the two types using PRODI for a single type to relate the two pairs of initial values for the two iters:

$$(\mathsf{VNum}\ 0, ptr) : \mathsf{Nat} \otimes \mathsf{ListT}_{read,T} \triangleright (\{0, \mathsf{unit}\}, lst)$$

Now we are ready to use ITER. Once we do, we face the proof obligation

$$\begin{array}{l} (n,p) : \mathsf{Nat} \otimes \mathsf{ListT}_{read,T} \\ \triangleright (i,l) \end{array} \vdash \begin{array}{l} i \leftarrow \mathsf{getNum}\ n; \\ b \leftarrow \mathsf{isNull}\ p; \\ \mathsf{if}\ \ldots \end{array} \lessapprox \begin{array}{l} \mathsf{case\ unfold}\ l\ \mathsf{of} \\ \quad \lambda\ \ldots \\ \quad \lambda\ \ldots \end{array} : (\mathsf{Nat} \otimes \mathsf{ListT}_{read,T}) \oplus \mathsf{Nat}$$

with new iteration variables n, p, i, and l, rather than the initial values for the iter.

Here, $\mathsf{Nat} \otimes \mathsf{ListT}_{read,T}$ can be thought of as the loop invariant. It must relate the iteration variables on both sides before each iteration, and must be reestablished if we are continuing to iterate with new values.

First, we discharge the getNum using BIND and GETNUM, after adding a $\mathsf{ret\ unit}$ no-op to the specification side using the monad laws. Next is case analysis on both sides. To do this, we unfold

the recursive type using UNFOLD:

$$p : \mathsf{ListT}_{read,T} \triangleright l \sqsubseteq p : \mathsf{eq}(\mathsf{VNum}\ 0) \vee (ptr((read, 0) \mapsto \mathsf{Nat}) \star ptr((read, 1) \mapsto \mathsf{ListT}_{read,T})) \triangleright \mathsf{unfold}\ l$$

By unfolding this recursive type, we now have a disjunctive type at the top level, and so we can use ORE to perform a case analysis.

For the first case, we are at the end of the linked list.

$$p : \mathsf{eq}(\mathsf{VNum}\ 0) \triangleright \mathsf{unit} * \ldots \vdash \begin{array}{l} b \leftarrow \mathsf{isNull}\ p; \\ \mathsf{if}\ b\ \ldots \end{array} \lessapprox \mathsf{ret}\ (\mathsf{inr}\ i) : (\mathsf{Nat} \otimes \mathsf{ListT}_{read,T}) \oplus \mathsf{Nat}$$

The types tell us that the pointer p is null, and that the specification value unfold l is unit, representing the empty list. We can then discharge the isNull with BIND and ISNULL2, since we have the equality type on p. Lastly, we need to handle the if statement. We know that the boolean b is true from the output type of ISNULL2: $b : \mathsf{eq}(\mathsf{true}) \triangleright \mathsf{unit}$. However, this does not let us immediately enter the true branch of the if statement. We need to use the IF rule, which requires an if statement on the specification side as well. To remedy this, we replace the specification side with the equivalent program $\mathsf{if}\ \mathsf{true}\ \mathsf{then}\ \mathsf{ret}\ (\mathsf{inr}\ i)\ \mathsf{else}\ \mathsf{error}$. We can then apply IF, handling the else case using ERR. Finally, for the then case, we can handle the two rets on both sides using RET, once we package the permissions back into the appropriate form for the output permission, using SUMI2, EXI, and EQCTX. This successfully shows that when both programs finish iterating, they return the same numeric value for the length of the list.

The other case from applying ORE is the case where the linked list is not empty:

$$\begin{array}{l} p : ptr((read, 0) \mapsto T) \star \\ ptr((read, 1) \mapsto \mathsf{ListT}_{read,T}) \\ \triangleright (h, t) * \ldots \end{array} \vdash \begin{array}{l} b \leftarrow \mathsf{isNull}\ p; \\ \mathsf{if}\ b\ \ldots \end{array} \lessapprox \mathsf{ret}\ (\mathsf{inl}\ (\{i.1 + 1, \mathsf{unit}\}, t)) : \begin{array}{l} (\mathsf{Nat} \otimes \mathsf{ListT}_{read,T}) \\ \oplus \mathsf{Nat} \end{array}$$

Here, the types tell us that p points to a value of type T, and $p + 1$ points to the next node of the

linked list. On the specification side, the value $\mathsf{unfold}\ l$ is a pair with the head of the list, h, and the tail of the list, t.

As before, the types tell us whether p is null, and we can use Bind and IsNull1 to discharge the isNull instruction on the implementation side. As before, this requires us to add a no-op to the specification side. Similarly, we also apply If and can then focus on the else case of the if statement:

$$\begin{array}{l} p : ptr((read, 1) \mapsto \\ \quad \mathsf{ListT}_{read,T}) \triangleright t \\ * \, n : \mathsf{eq}(i.1) \triangleright \mathsf{unit} \end{array} \vdash \begin{array}{l} p' \leftarrow \mathsf{load}\ (\mathsf{offset}\ p\ 1); \\ \mathsf{ret}\ (\mathsf{inl}\ (\mathsf{VNum}\ (n+1), p')) \end{array} \lessapprox \mathsf{ret}\ (\mathsf{inl}(\{i.1+1, \mathsf{unit}\}, t)) : \begin{array}{l} (\mathsf{Nat} \otimes \mathsf{ListT}_{read,T}) \\ \oplus \mathsf{Nat} \end{array}$$

At this point, we have dropped the type on the first element of the list, since we will not use it for the remainder of the typechecking process. We then apply PtrE to shift the content type out of the pointer type, so we can then use the PtrOff and Load rules, discharging the load instruction. Finally, as in the other case, we can repackage our types to match the output type, and conclude with Ret. This shows that when we continue iterating, the values we iterate with are still related via the loop invariant.

6.7. The Heapster Tool

Heapster implements the type system defined in this chapter, using it to extract specification programs from implementation programs and type annotations. Heapster has its origins in the Software Analysis Workbench (SAW) [Doc+16], a verification tool developed at Galois. SAW uses symbolic execution, making it best suited to code with bounded loops, and is primarily used to prove the correctness of implementations of cryptographic algorithms. Heapster was developed as an add-on to SAW to verify code with unbounded loops.

The tool supports input code in LLVM intermediate representation (IR), thus supporting languages that compile to LLVM, like C and Rust. The extracted functional specifications are ITrees augmented with support for logical quantifiers [Sil+23b], which are just data structures in Coq, and can be used in the proof assistant as with any other data structure.

For a user, the first step to use Heapster is to provide their code to the tool, as well as type annotations for each function they wish to extract a specification of. These annotations describe the input and output types of each function. As we have seen in this chapter, these types can be quite expressive, and the type information from code written in imperative languages like C is typically not sufficient to act as Heapster types. For most languages, the user writes these type annotations by hand and provides them to the tool. For input code written in Rust, however, the powerful type system of Rust means that these types *can* sometimes be translated directly to Heapster type annotations. This translation is not exhaustive, and does not support all Rust features. We will describe support in our type system for lifetimes in Chapter 7, and Heapster's support for Rust is similar—"standard" imperative features plus lifetimes and borrowing.

Given the imperative program and appropriate type annotations, the tool then uses heuristics to apply the typing rules of the tool, and generates a specification program in Coq as part of this process. While the type system presented in this chapter is not algorithmic—containing rules like ITER that requires choices of intermediate types, for example—Heapster succeeds in typechecking many programs. While any program can be typechecked using ERR, this only occurs when the tool "gives up", and we consider this a failure. The user can determine whether this occurred by inspecting the extracted specification, as errors in the tool contain error messages, unlike in the theory. In a case like this, the program may require typechecking hints in the form of intermediate type annotations in the middle of a function. For example, programs using loops require such annotations to act as loop invariants.

Once the user has the extracted specification, they are free to verify anything they wish about these programs. One useful thing to verify is that the specification cannot step to any error programs, which tells us that the imperative program is also safe by Theorem 6.2. Another common use case is to verify that the extracted specification refines a higher-level, handwritten specification. For these cases, Heapster provides automation in Coq for these refinement proofs [Sil+23b]. In other cases, manual proof is necessary. For example, ERR could have been used, but the error computation only exists in some unreachable portion of the specification program. The provided

automation may not be able to prove this, and the user is then forced to write a manual proof in Coq if they want a formal guarantee of this property.

Empirically, Heapster works well on nontrivial functions. He et al. [He+21] presents a suite of C functions for a memory buffer data structure, implemented as a linked list. 14 of the 18 functions are typechecked successfully by Heapster, without using ERR, with each function that uses a loop requiring an additional type annotation. Additionally, 11 of the extracted functions were also verified to be both error-free and refine a higher-level specification, with the help of the Coq automation provided by the tool [Sil+23b].

In addition to these empirical results and the basic safety result in Theorem 6.2, He et al. [He+21] further proves a theorem that relates the behavior of the implementation and specification programs. The basis of the semantic typing judgment is our definition of stuttering bisimulation up to errors on the right. In general, stuttering bisimulation preserves temporal properties in computation tree logic (CTL) that do not use the next-time operator [BCG88]. This ensures that the extracted specifications satisfy the same safety, liveness, and fairness properties as the imperative programs.

A variant of this result is proved by He et al. for our notion of bisimulation. Consider the following fragment of CTL where $\mathsf{St}(P)$ denotes a *state predicate* (the atomic formulas in this logic) for a predicate $P \subseteq S$ over values in state type S:

$$\phi ::= \mathsf{St}(P) \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \phi_1 \Rightarrow \phi_2 \mid \mathsf{EF}\ \phi \mid \mathsf{EG}\ \phi \mid \mathsf{AF}\ \phi \mid \mathsf{AG}\ \phi.$$

The semantic entailment relation $t, s \models \phi$ on programs t of type $\mathsf{itree}\ S\ R$, s of type S, and formula ϕ over S from our fragment of CTL can then be defined in the standard way.

Since implementation and specification programs have different state types, we also need a definition to relate formulas over different state types. To do this, these formulas are related with respect to some permission π which encodes the "equivalent" states in the two state types in its precondition.

Definition 6.4 (π-similarity [He+21, Definition 4.5]). Let π be a permission over $S_i \times S_s$, $P_i \subseteq S_i$, and $P_s \subseteq S_s$. Predicates P_i and P_s are *π-similar* if for all $s_i \in S_i$ and $s_s \in S_s$ such that $P_\pi(s_i, s_s)$, we have $P_i(s_i) \iff P_s(s_s)$. CTL formulas ϕ_1 and ϕ_2 are *π-similar* if ϕ_2 can be obtained from ϕ_1 by replacing every subformula $\mathsf{St}(P_i)$ with some $\mathsf{St}(P_s)$ for P_i π-similar to P_s.

This permission π can be thought of as the permission held by an observer trying to prove that programs satisfy some formulas in this fragment of CTL. The precondition of π represents the observer's view of how implementation and specification states correspond to each other, and which states are indistinguishable.

With the definition of π-similarity in hand, we can then prove the result that bisimulation preserves temporal properties.

Theorem 6.13 ([He+21, Theorem 4.6]). If $(t_i, s_i) \lessapprox_{\pi_1, F} (t_s, s_s)$ and $(s_s, t_s) \not\rightarrow^* \mathsf{error}$, then for π_2-similar formulas ϕ_1 and ϕ_2, and $P_{\pi_1 * \pi_2}(s_i, s_s)$, we have $t_i, s_i \models \phi_1 \iff t_s, s_s \models \phi_2$.

Note that $P_{\pi_1 * \pi_2}(s_i, s_s)$ implies $\pi_1 \perp \pi_2$. Since separateness ensures that the rely of π_2 is not invalidated by π_1, the precondition of π_2 remains invariant throughout execution. This is necessary since the precondition of π_2 captures the equivalence of state predicates used in the definition of π-similarity.

CHAPTER 7

Lifetimes

One major feature of Heapster that we did not model in Chapter 6 is *lifetimes*. A lifetime is used to split types *temporally* and describes when each portion of the split type is active. This has two uses in Heapster. First, to handle imperative code written in Rust, because Rust types can be translated to Heapster types and Rust types have a notion of lifetimes. Second, to increase the expressivity of Heapster, even when run on imperative code in languages other than Rust. This is because lifetimes allow us to typecheck safe code patterns that would not be well-typed without them.

A central concept in Rust is that each value has only one owner. This is similar to Heapster, where a pointer-write type has exclusive ownership of memory values. Like Heapster, Rust types also guarantee memory safety by preventing bugs caused by pointer aliasing. Because of this property, code written in the safe subset of Rust should also be well-typed in Heapster with their Rust types, though of course, the Rust types must be translated to Heapster types. This saves users of Heapster from one of the main steps of using the tool: writing type annotations.

To allow for a more natural programming style, where variables can be reused safely, Rust uses a *borrowing* mechanism where the ownership of a value can be temporarily borrowed. The duration of the borrow is also known as a lifetime, and the compiler checks that lifetimes do not overlap inappropriately so that there is still only a single owner at a time. This feature, known as the borrow checker, is both one of the trickiest and most important concepts of Rust, and has been the target of many verification efforts [Jun+17; Jun+19; Pea21; PPS22]. While Rust lifetimes are usually implicit in the code and inferred by the compiler, they can also be written in types manually by the programmer. Thus, to convert Rust types—which may include Rust lifetimes—to Heapster types, there must also be a way to represent lifetimes in the Heapster type system. Due to their similarity, we will refer to both features in Rust and Heapster as simply *lifetimes*.

As an example of lifetimes in Rust, consider the Rust program shown in Figure 7.1. This program

```
fn main() {
    let mut data = 10;

    let ref1 = &data;            //  -+- lifetime a
    println!("ref1: {}", ref1);  //  -+

    let ref2 = &mut data;        //  -+- lifetime b
    *ref2 += 1;                  //   |
    println!("ref2: {}", ref2);  //  -+
}
```

Figure 7.1: A Rust program that uses lifetimes implicitly.

creates an immutable reference `ref1` that borrows the value of `data`. Rust's compiler can infer its lifetime, which is shown in comments. The reference is not used after it is used to print, so its lifetime can end after its final use. The program then creates a *mutable* reference `ref2` that also borrows `data`. Again, the lifetime of this reference can be inferred, and is shown in comments. Rust's borrow checker checks that these lifetimes do not overlap, since it would be unsafe to have mutable and immutable references borrowing the same value. This program compiles, but if `ref2` attempted to borrow `data` after `ref1` borrows the value and before `ref1` is printed, the two lifetimes would overlap and the program would fail to compile.

These borrows and lifetimes in Rust can be translated to similar concepts in Heapster. The first borrow in Rust would correspond to Heapster's notion of splitting a type using a lifetime. The Heapster lifetime, like the lifetime `a` in the Rust program, would start when `ref1` is created, and end after it is used for the last time in the next line. The pointer type for the variable `data` would be split into two pieces using this lifetime, the pieces before and after the lifetime ends. The first piece can be weakened into a read-pointer type, and the second piece can remain a write-pointer type. Since the second borrow is the final use of `data` in the program, it is unnecessary to split the type again in Heapster. Checking whether references are used safely is also different in Heapster. Rather than creating regions and checking whether they overlap, splitting types using Heapster lifetimes cannot result in unsafe code, so no borrow checker is necessary.

Even when Rust is not involved, lifetimes are useful in Heapster. A problem with the type system

in Chapter 6 originates from the fact that permissions can change only monotonically via the $\rightsquigarrow$ relation during typechecking. This means that once a permission is weakened—for example by weakening a write pointer type to a read using the PtrWeak rule—there is no way back. This ability to revert to a previous type is crucial for typechecking some memory-safe programs. A program like Figure 7.1 in a language like C is safe—even when multiple immutable references like `ref1` are used to read in the same region—since pointer aliasing is only unsafe if one of the pointers has write access, which is why we have the ReadDup rule. It is safe to use multiple pointer aliases to read, then later use one alias to write, as long as none of the other aliases are used after the write. However, if we want to typecheck such a program in Heapster, we would have to weaken the original pointer-write type to a pointer-read type, and then later go back to a write type. There is no way to do this in the type system we have presented so far. Fractional permissions [Boy03] are the usual solution in separation logic for this issue. While that is an option for Heapster as well, lifetimes gives us another solution that is often more intuitive. Lifetimes will allow us to split a permission into the portion before the lifetime ends, and a portion after. The portion before the lifetime ends can be weakened, then reverted to the original permission once the lifetime has ended.

In this chapter, we will define the semantic definition of lifetime types and how they interact with existing types. We first introduce the basic operations and the state changes required to represent lifetimes, as well as the permission definitions. The relational nature of rely-guarantee permissions is crucial for these permissions, as they will be used to control specific changes to lifetimes. The definition of the permission types using these permissions is surprisingly intricate. As we will see in Section 7.3, the part of the permission before the lifetime ends cannot be weakened arbitrarily. For example, if we have a pointer type and the pointer is deallocated before a lifetime ℓ ends, then the pointer type should not be recovered once ℓ ends. To solve this, our definitions will require a specific permission to be *returned* before the lifetime is allowed to end, to ensure that split permissions are not rendered invalid after the lifetime ends.

The case of pointer types will also require special treatment due to the content type component

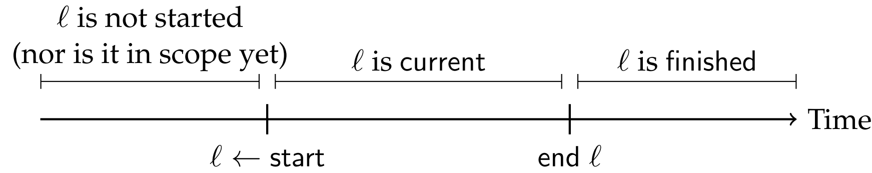


Figure 7.2: The life cycle of a lifetime ℓ.

of pointer types. Though split permissions are not permitted to change arbitrarily, if the value pointed to by a pointer changes, then the pointer type's content type *should* change as well. To handle this, we develop special rules for the combination of lifetime types and pointer types in Section 7.4 where content types are excluded from the splitting process.

As we will see in Section 7.6, this chapter will not model every part of the implementation of lifetimes in Heapster. There are still lifetime features missing from the type system we define, though the theory presented in this chapter shows that the core functionality of lifetimes *can* be modeled by rely-guarantee permissions, and future work can extend the system to model the full Heapster implementation. As such, the treatment of lifetimes in this chapter focuses on expanding the expressivity of the type system, and is more exploratory in nature than the theory presented in Chapter 6.

7.1. Defining Lifetime Operations

We begin by defining our representation of lifetimes, and add them to the state. The life cycle of a lifetime is shown in Figure 7.2. A lifetime is started by the start instruction, which returns the name of the lifetime, ℓ. At that point, the status of ℓ is that it is active, or current. After that, the lifetime can be ended with the end ℓ instruction, which changes ℓ to finished, where it remains—it should not be able to become current again.

Concretely, in the state, we will model the status of all lifetimes as a list. Similarly to a Mem, we will define a lifetime list Ltms as a list of statuses, where each status is either current or finished. Lifetime names like ℓ will be represented by indices into the Ltms list, so $\ell \in \mathbb{N}$.

This model does not prevent invalid updates to lifetimes, like a lifetime going from finished to

current, or a lifetime being deleted from Ltms, which represents returning to a state where it was not started. To represent well-behaved changes to the Ltms list that preclude such invalid updates, we define a relation on two Ltms lists, $\rightharpoondown$, which requires that lifetimes not go "backwards". The relation $s_1 \rightharpoondown s_2$ holds if s_2 is not shorter than s_1, and no lifetimes present in s_1 go from finished to current in s_2. To enforce this, some of our permissions that use lifetimes will include this relation in their rely, thus requiring that the guarantee of other permissions obey this relation.

Now that we have changed the state type to include this new Ltms list, previously-defined permission types must also be modified. The "generic" permission types defined in Chapter 6, like type connectives, are defined over any state type, and so are not affected. The memory permissions defined in Chapter 6 were defined over an implementation state type of Mem, but can be modified to work over any implementation state type that *contains* a Mem. The guarantees of those permissions then say they can only modify the Mem portion of the state, and the relies of those permissions will not have additional constraints on the non-Mem portions of the state. In the Coq formalization, the definitions that depend on a specific state type are all implemented using lenses [Fos+07], so the system can be further extended to larger state types. We will continue to write our types as if the state is exactly equal to the state type of interest, Ltms, but should be taken to mean that our permissions can work on any state that just *contains* a Ltms. When we begin defining types involving the combination of lifetime types and pointer types, it should be assumed that the implementation state type contains both a Ltms and a Mem.

With this state capable of representing our lifetimes, we can then define the start and end instructions for starting and ending lifetimes.

$$\mathsf{start} : \mathsf{itree}\ \mathsf{E}_{\mathsf{Ltms}}\ \mathbb{N}$$
$$\mathsf{end} : \mathbb{N} \to \mathsf{itree}\ \mathsf{E}_{\mathsf{Ltms}}\ \mathsf{unit}$$

These operations for managing lifetimes are similar to malloc and free for managing memory, defined in Chapter 6. The start instruction appends a new lifetime status of current to the lifetime list,

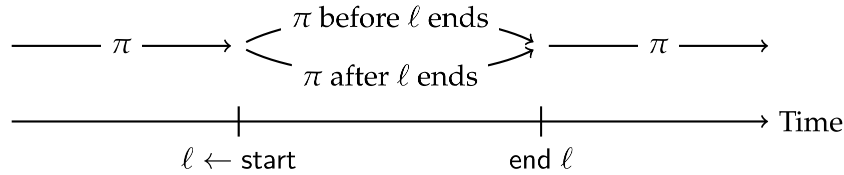


Figure 7.3: How we *want* to use a lifetime ℓ to split a permission π.

and returns the index of this new element to act as the lifetime name. Like malloc, this operation never fails since we have an unbounded number of lifetimes available.

The end instruction takes a lifetime name as an argument, and tries to end this lifetime by setting its status to finished. If the lifetime is either already finished or not started yet (i.e. it is out of scope in the lifetime list), then end results in error. Otherwise, the status of ℓ is successfully updated to finished in the Ltms.

Lifetimes do not exist in most imperative languages, and even in Rust, which does have lifetimes, they are only used by the borrow checker and do not exist in the state. Thus these start and end instructions do not exist in the original implementation programs and must be inserted. As such, our lifetime representations can be thought of as a form of ghost state and the instructions as a form of ghost commands to manipulate the ghost state [Kri+21].

7.2. Lifetime Permissions

The way we make use of these lifetimes that we have just defined is by using them to *modify* existing permission types. A lifetime specifies a duration in time, and we will define lifetime permissions to split a permission using this duration into two portions: the portion before the lifetime ends and the portion after the lifetime ends. This can be visualized in Figure 7.3. Ideally, we should be able to arbitrarily change the portion of π before ℓ ends and still get back the original π after ℓ ends. As we will soon see, the reality of the types and rules we will define and prove will not quite match this ideal.

To start with, at the permission level we will first define permissions to manage the lifetimes themselves. Analogously to the Π_{alloc} permission set defined using a $\pi_{\mathsf{alloc}}(b)$ permission for

memory, we will define a Π_{lifetime} permission set using a $\pi_{\mathsf{lifetime}}(n)$ permission in the same way, but for the lifetime list rather than the list of blocks in Mem.

$$\begin{aligned}\pi_{\mathsf{lifetime}}(n) \stackrel{\text{def}}{=} (&\{\, (s_1, s_2) \mid \text{the number of statuses in } s_1 \text{ and } s_2 \text{ are the same, and} \\ &\qquad \text{statuses} \geq n \text{ do not change between } s_1 \text{ and } s_2 \,\}, \\ &\{\, (s_1, s_2) \mid \text{statuses} < n \text{ do not change between } s_1 \text{ and } s_2\text{, and} \\ &\qquad s_1 \rightharpoondown s_2 \,\}, \\ &\{\, s \mid \text{there are } n \text{ statuses in } s \,\})\end{aligned}$$

The only difference between this definition and that of $\pi_{\mathsf{alloc}}(b)$ is that the guarantee of this permission also requires that the Ltms lists do not go backwards. The permission set version of this is defined identically as Π_{alloc}:

$$\Pi_{\mathsf{lifetime}} \stackrel{\text{def}}{=} \bigsqcup_{n \in \mathbb{N}} [\pi_{\mathsf{lifetime}}(n)]$$

One important property we need for the permissions that we split is that they not affect lifetimes. To do this, we use separateness:

Definition 7.1. A *non-lifetime* permission is a permission π where $\pi \perp \pi_{\mathsf{lifetime}}(n)$ for all n.

The use of separateness ensures that non-lifetime permissions can tolerate updates to the lifetime portion of the state using the rely, and do not change anything in the lifetime portion themselves using their guarantee. This is necessary because the permission being split relies on lifetimes to determine how they behave. If the permission itself can change lifetimes, they could violate some of the implicit invariants of the lifetime permissions that we will define.

In addition to being non-lifetime, we also need the permissions that we split to be *oblivious* to lifetimes. This means that these permissions tolerate updates to the lifetime in their rely, guarantee, and precondition. That is, if a state is in the precondition of the permission, then it should continue to be in the precondition even if the lifetimes in the state change, and similarly for the rely and guarantee—even if the changes to the lifetimes are different in the two states related by the

rely or guarantee. This property seeks to rule out permissions that inspect or change lifetimes, and means that we cannot split the permissions that involve lifetime changes that we define in this chapter. For example, the permission that allows a lifetime ℓ to be ended cannot itself be split using another lifetime, since it would not be oblivious to changes to the lifetime ℓ. Both the non-lifetime and obliviousness properties are satisfied by all the permissions defined in Chapter 6, and so we will omit mentions of these properties from now on.

Now we can define our first permission that works with a specific lifetime. We will call this an owned permission, as it represents exclusive ownership of a lifetime ℓ. This permission $\mathsf{owned}(\ell, \pi)$ also takes an argument π, which represents the permission that will be "given back" once the lifetime ℓ ends.

$$
\begin{aligned}
\mathsf{owned}(\ell, \pi) \stackrel{\text{def}}{=} (&\{ (s_1, s_2) \mid \ell \text{ has the same status in } s_1 \text{ and } s_2\text{, and} \\
&\qquad \text{if } \ell \text{ is } \mathsf{finished} \text{ in } s_1\text{, then } (s_1, s_2) \in R_\pi \}, \\
&\{ (s_1, s_2) \mid \text{either } s_1 = s_2 \text{ or} \\
&\qquad \text{for every lifetime } \ell' \neq \ell\text{, the status of } \ell' \text{ in } s_1 \text{ and } s_2 \text{ are equal,} \\
&\qquad \text{the status of } \ell \text{ in } s_2 \text{ is } \mathsf{finished}\text{, and} \\
&\qquad (s_1[\ell \mapsto \mathsf{finished}], s_2[\ell \mapsto \mathsf{finished}]) \in G_\pi \}, \\
&\{ s \mid \ell \text{ is } \mathsf{current} \text{ in } s \})
\end{aligned}
$$

The rely of this permission requires that nobody else change the status of ℓ, reflecting that it represents exclusive ownership of the lifetime. To represent π once ℓ ends, it also requires that the rely of π holds at that point. The guarantee says that if a change is made, then the change must have ended the lifetime ℓ. Additionally, we permit any changes that the guarantee of π permits, as long as the status of ℓ is updated in those states. No other changes to lifetimes are allowed, which means that $s_1 \rightharpoondown s_2$ must hold. Finally, the precondition for this permission says that ℓ is current in the state.

This permission combines several ideas. First, it is the permission representing exclusive own-

ership of the lifetime ℓ, allowing the holder of the permission to end the lifetime ℓ. Second, this permission also tells us that ℓ is current. Because of this, once ℓ is ended we will need a different permission to represent π. Lastly, it represents the portion of the permission π that holds *after* ℓ ends, which we saw in Figure 7.3. It does this by using the rely and guarantee of π in its own rely and guarantee. Notably, it does not use the precondition of π. If it did attempt to do this in a similar way as the rely, by requiring that the precondition of π hold if ℓ is finished, then that would be vacuously true since the precondition also requires that ℓ be current. The precondition of π will be an issue we will address at the end of this section.

Next, we introduce the other half of the permission split seen in Figure 7.3. The $[\ell]\pi$ permission, which we will call a when permission, represents the portion of the permission π that holds *when* ℓ is current.

$$
\begin{aligned}
[\ell]\pi \stackrel{\text{def}}{=} (\{\, (s_1, s_2) \mid\ & s_1 \rightharpoondown s_2 \text{ and} \\
& \quad \text{if } \ell \text{ is not finished in } s_2\text{, then } (s_1, s_2) \in R_\pi \,\} \\
\{\, (s_1, s_2) \mid\ & \text{either } s_1 = s_2 \text{ or} \\
& \quad \text{the lifetimes of } s_1 \text{ and } s_2 \text{ are equal,} \\
& \quad \ell \text{ is current in } s_1\text{, and} \\
& \quad (s_1, s_2) \in G_\pi \,\}, \\
\{\, s \mid\ & \text{if } \ell \text{ is not finished in } s\text{, then } s \in P_\pi \,\}
\end{aligned}
$$

The rely of this permission requires that lifetime updates are well-behaved, and that the rely of π holds as long as ℓ has not ended yet. We only need to check that it has not ended in s_2, since the $\rightharpoondown$ requirement means that if it is not finished in s_2, then it cannot be finished in s_1. The guarantee says that if any change happens, it must not involve any lifetime changes, and ℓ must be current. Given those conditions, then the guarantee permits any change permitted by the guarantee of π. Finally, we also use the precondition of π as that of $[\ell]\pi$ whenever ℓ is not yet finished.

When the lifetime ℓ is current, we want $[\ell]\pi$ to behave like π, which it does. Additionally, since

the rely and precondition check whether ℓ is not finished, both the rely and precondition of π will hold when ℓ is not started yet. This requirement, added for making some of the proofs simpler, does not affect how this permission functions, as these when permissions cannot exist before the lifetime ℓ is started.

As mentioned earlier, we will need another lifetime permission to represent the split permission after ℓ ends. If we are to allow for the portion of π before ℓ ends to be changed and weakened, then it makes sense that we would not be able to reobtain the original permission π once ℓ ends, since some of the original permission has been lost in this weakening. We will call this permission used for representing π after ℓ has ended a finished permission, written $\{\ell\}\pi$:

$$
\begin{aligned}
\{\ell\}\pi \stackrel{\text{def}}{=} (&\{\, (s_1, s_2) \mid s_1 \rightharpoondown s_2 \text{ and} \\
&\qquad \text{if } \ell \text{ is finished in } s_1 \text{, then } (s_1, s_2) \in R_\pi \,\} \\
&\{\, (s_1, s_2) \mid \text{either } s_1 = s_2 \text{ or} \\
&\qquad \text{the lifetimes of } s_1 \text{ and } s_2 \text{ are equal,} \\
&\qquad \ell \text{ is finished in } s_1 \text{, and} \\
&\qquad (s_1, s_2) \in G_\pi \,\}, \\
&\{\, s \mid \ell \text{ is finished in } s \text{ and } s \in P_\pi \,\}
\end{aligned}
$$

This permission is very similar to the when permission. The rely requires that lifetime updates obey $\rightharpoondown$, and that the rely of π holds as long as ℓ is finished in the earlier state. Like the when permission, we only enforce the rely of π when ℓ has the status of interest (finished in this case) in both the starting and ending states. The guarantee, like that of the when permission, only permits changes permitted by π, without any lifetime changes, and only when ℓ is finished. The precondition, however, differs from that of when. Rather than an implication, the precondition of the finished permission *states* that ℓ is finished and that the precondition is the same as that of π. This is because the finished permission is the final step in the life cycle of the permission, and ℓ cannot change to another status. The implication was necessary for the when permission because the lifetime could

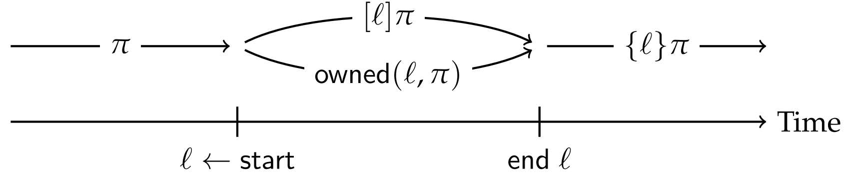


Figure 7.4: How we will use a lifetime ℓ to split a permission π.

become finished, and any when permissions still remaining must then stop behaving like π.

By carefully choosing the definitions of these permissions, we can prove the following results, showing that the properties we want for splitting permissions hold. First, when permissions and owned permissions for the same lifetime ℓ can coexist, even if π_1 and π_2 are not separate, because their relies and guarantees split the underlying relies and guarantees by the lifetime ℓ.

Lemma 7.1.

$$[\ell]\pi_1 \perp \mathsf{owned}(\ell, \pi_2)$$

We can then prove the following theorem, which tells us that we can split permissions using owned and when permissions.

Theorem 7.2 (Permission splitting)**.**

$$\pi * \mathsf{owned}(\ell, \pi') \sqsubseteq [\ell]\pi * \mathsf{owned}(\ell, \pi * \pi')$$

Now we have a more concrete diagram of how permissions are split, shown in Figure 7.4.

While most of the pieces are in place, we run into a problem with preconditions when we try to prove the typing rule for ending a lifetime. As mentioned earlier, the owned(ℓ, π) permission does not enforce the precondition of π, so how can we reobtain π once ℓ ends? Even if owned did use the precondition of π when ℓ is finished, we would still need some way of requiring the state to be in that precondition when the lifetime ends. During the period where ℓ is current, we should have been able to change $[\ell]\pi$ arbitrarily, so the precondition of π may not hold at all at that point. It

turns out that we cannot permit *unrestricted* changes to the permission while ℓ is current in order to regain π after the lifetime ends. The next section will define permission *sets* for lifetimes, and in that definition, introduce a solution for this problem.

7.3. Recovering Split Permissions

Reobtaining the original permission after ending a lifetime cannot be done unconditionally. That is, if we defined permission set versions of the lifetime permissions as follows:

$$\begin{aligned}
\mathsf{owned}(\ell, \Pi) &\stackrel{\mathsf{def}}{=} \bigsqcup_{\pi \in \Pi} [\mathsf{owned}(\ell, \pi)] \\
[\ell]\Pi &\stackrel{\mathsf{def}}{=} \bigsqcup_{\pi \in \Pi} [[\ell]\pi] \\
\{\ell\}\Pi &\stackrel{\mathsf{def}}{=} \bigsqcup_{\pi \in \Pi} [\{\ell\}\pi],
\end{aligned}$$

we cannot prove a rule like

$$\frac{}{\mathsf{owned}(\ell, \Pi) \vdash \mathsf{end}\ \ell \lessapprox \mathsf{ret\ unit} : \mathsf{True} \oslash \{\ell\}\Pi,}$$

since we cannot ensure that the precondition of the returned permission holds.

Instead, what we will do—following from Heapster—is to add an extra argument to the definition of the owned permission *set*, representing the permissions that must be *returned* to recover the original split permissions. We will write this as $\mathsf{owned}(\ell, \Pi_1 \multimap \Pi_2)$, where Π_1 must be given back in order to recover Π_2 when ending ℓ. We want the following rule for ending a lifetime to be provable:

$$\frac{}{\Pi_1 * \mathsf{owned}(\ell, \Pi_1 \multimap \Pi_2) \vdash \mathsf{end}\ \ell \lessapprox \mathsf{ret\ unit} : \mathsf{True} \oslash \{\ell\}\Pi_2}$$

This new argument to the owned permission set is used purely to ensure the precondition holds, so the values of Π_1 could vary depending on the specific Π_2. Specific rules for different Π_1 and Π_2 values for different uses could be written, depending on what precondition is needed. It should always be sound to have $\Pi_1 = [\ell]\Pi_2$, but a different permission could work in special cases, like

when we are dealing with pointer permissions. In that case, the weaker pointer-read permission should be sufficient to recover a pointer-write permission, since these two permissions have the same precondition. This also means that our example from earlier of temporarily weakening a pointer-write type into a read type should still be possible with this approach, since the read type will be sufficient to recover the write type.

Another feature we would like to support with this owned permission set is the ability to return *part* of the permissions that are needed to recover the split permission. These partial returns are used in Heapster to restrict access to part of the split permission. For example, a function for an object might split the type for the entire object, but only give access to one specific field of the object. The equivalent in Rust would be to borrow a structure and discard access to part of the structure, for example with a getter function for one field of the structure. Partially returning the portion of the type for every field except the desired one would express this pattern. More concretely, we want the following rule to be provable:

$$\frac{}{\Pi * \mathsf{owned}(\ell, \Pi * \Pi_1 \multimap \Pi_2) \sqsubseteq \mathsf{owned}(\ell, \Pi_1 \multimap \Pi_2)}$$

To support both of these typing rules, we define the permission set version of owned as follows:

$$\begin{aligned}
\mathsf{owned}(\ell, \Pi_1 \multimap \Pi_2) \stackrel{\text{def}}{=} \{\, x \mid\ & x \sqsubseteq \pi * \mathsf{owned}(\ell, \pi_2), \text{ for some } \pi_2 \in \Pi_2 \text{ such that} \\
& \text{for any } \pi_1 \in \Pi_1 \text{ where } \pi_1 \perp \pi * \mathsf{owned}(\ell, \pi_2), \\
& \quad \text{there exists a } \pi_2' \in \Pi_2 \text{ where } \pi_2 \rightsquigarrow \pi_2' \text{ and} \\
& \quad\quad \text{for any state } s, \\
& \quad\quad\quad \text{if } s[\ell \mapsto \mathsf{current}] \in P_{\pi_1} \text{ and } s[\ell \mapsto \mathsf{current}] \in P_{\pi}, \\
& \quad\quad\quad \text{then } s[\ell \mapsto \mathsf{finished}] \in P_{\pi_2'}. \,\}
\end{aligned}$$

An element of this permission set is one that can be weakened to $\mathsf{owned}(\ell, \pi_2)$ combined with a permission π, the partially returned permission to this owned permission set. The rest of this

definition says that if we have a $\pi_1 \in \Pi_1$, then that should be able to give us an element of Π_2. This element π_2' may not be the exact permission returned to us by the owned permission, but it will be in Π_2. Together, the preconditions of π and π_1 should be sufficient to imply the precondition of the permission π_2'. The use of $\rightsquigarrow$ here for relating π_2 and π_2' is crucial, as the precondition of π_2 may not hold. The final part of the definition tells us that the precondition of this π_2' must hold, as long the preconditions of π_1 and π hold, at the corresponding statuses of ℓ. We should hold π_1 and π before the lifetime has ended, so they are checked when ℓ is $\mathsf{current}$. We want to be holding π_2' after the lifetime has ended, so the conclusion tells us that the precondition holds on the state where ℓ is $\mathsf{finished}$.

With this definition, we can now prove the desired typing rules, though we still have not defined these concepts as permission types, so some types will be represented as functions. When we do define the appropriate permission types in the next section, we will also prove new typing rules that use these ones in their proofs.

First, we can prove the rule for starting a lifetime.

Theorem 7.3.

$$\frac{}{\Pi_{\mathsf{lifetime}} \vdash \mathsf{start} \precapprox \mathsf{ret\ unit} : \lambda(\ell, _).\, \mathsf{owned}(\ell, \Pi_\top \multimap \Pi_\top) * \Pi_{\mathsf{lifetime}}}$$

The new owned permission set does not give any permission set back, and does not require any permission set to be returned to it, represented by the vacuous $\Pi_\top$ permission sets in both positions.

We can then prove the rule for ending a lifetime, where if we have the exact permission set needed to return to the owned construct, then it can be consumed and we can reobtain the Π_2 set.

Theorem 7.4.

$$\frac{}{\Pi_1 * \mathsf{owned}(\ell, \Pi_1 \multimap \Pi_2) \vdash \mathsf{end}\ \ell \precapprox \mathsf{ret\ unit} : \mathsf{True} \oslash \{\ell\}\Pi_2}$$

Next, we can prove a splitting rule specialized to pointer types, using the idea that a read permission is sufficient to recover a write permission after the lifetime ends.

Theorem 7.5.

$$\frac{}{\begin{array}{l}x_i : ptr((rw, o) \mapsto \mathsf{eq}(x)) \triangleright \mathsf{unit} \, * \\ \mathsf{owned}(\ell, \Pi_1 \multimap \Pi_2)\end{array} \sqsubseteq \begin{array}{l} x_i : [\ell]ptr((rw, o) \mapsto \mathsf{eq}(x)) \triangleright \mathsf{unit} \, * \\ \mathsf{owned}(\ell, \\ \quad x_i : [\ell]ptr((read, o) \mapsto \mathsf{eq}(x)) \triangleright \mathsf{unit} * \Pi_1 \multimap \\ \quad x_i : ptr((rw, o) \mapsto \mathsf{eq}(x)) \triangleright \mathsf{unit} * \Pi_2)\end{array}}$$

This rule says if we have an owned permission set and a pointer type with an equality content type, then we can apply this rule to split the pointer type, ending up with the portion inside a when and the other portion inside the owned set. To regain the full pointer type after the lifetime ends, we must return the portion inside the when, but it does not have to be the original pointer type—it can be the weaker pointer-read type.

Note that here, when we apply a when to a pointer type, we only apply it to the pointer permission set portion of the type, and not the content type. Heapster does not recursively split the content type of pointer types. If we did want to split the content type, we can always move the content type out using PtrE and split it manually. Similarly, we will write $\{\ell\}ptr((rw, o) \mapsto T)$ to apply a finished to a pointer type in the same way, where we apply the finished permission set definition to the pointer permission set portion of the type, and not to the permission type T.

Finally, we can prove the rule for partially returning a permission to the owned permission set, specialized to pointer types:

Theorem 7.6.

$$\frac{}{\begin{array}{l}x_i : [\ell]ptr((read, o) \mapsto \mathsf{eq}(x)) \triangleright \mathsf{unit} \, * \\ \mathsf{owned}(\ell, (x_i : [\ell]ptr((read, o) \mapsto \mathsf{eq}(x)) \triangleright \mathsf{unit}) * \Pi_1 \multimap \Pi_2)\end{array} \sqsubseteq \mathsf{owned}(\ell, \Pi_1 \multimap \Pi_2)}$$

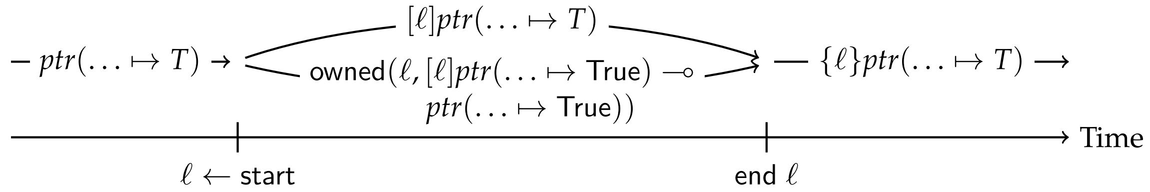


Figure 7.5: How we will use a lifetime ℓ to split a pointer type.

These rules allow us to temporarily weaken a pointer-write type to a pointer-read and regain the original write type afterwards, but do not allow much else. The definitions require that we return the *exact* permission type that we started with, including the equality content type, which disallows any change to what the pointer points to, effectively banning uses of STORE while the type is split. To fix this issue, we will once again define a solution at the next level of definitions, this time at the permission *type* level.

7.4. Lifetime Types and Typing Rules

In this section we will introduce a permission type called ownedptrs and its typing rules for the special case of splitting pointer types. While the splitting rule we presented in the previous section is too restrictive, we can build on those permission set definitions to allow for changes to the content types during the duration of a lifetime. Instead of using Theorem 7.6 to handle partially returned permission sets, we will instead define ownedptrs as bundling both the partially returned permission sets and the owned permission set together. Using this approach, once all the permissions necessary to end the lifetime are obtained, we use all of them at once along with the owned permission set to end the lifetime with Theorem 7.4.

This approach effectively never uses the π portion of the definition of owned$(\ell, \Pi_1 \multimap \Pi_2)$ that enables Theorem 7.6. Internally, that permission will always be $\pi_\top$, as we never partially return anything to the permission set. This does not mean that the full definition of the owned permission set or Theorem 7.6 are useless, but rather that they are not well-suited to handling our pointer types, due to the content type. Other types and typing rules that involve state without combining two permissions—as pointer types do with content types—could make use of them.

The approach we will take, shown in Figure 7.5, is to keep content types with the pointer type wrapped in a when, and not with the owned permission set at all. Crucially, the pointer type attached to the owned portion will have its content type weakened to True. When a pointer type is split, its content type T will be moved to the when pointer type that is usable while the lifetime is current. For the duration of the lifetime, this content type T can change arbitrarily, since we do not need it to end the lifetime. When we do end the lifetime, T must be temporarily moved so the when pointer types have a True content type to match the one in the owned permission set. After the lifetime is ended, we reobtain the pointer type with a True content type, and can then move T back in to replace True as the content type.

The ownedptrs type, written as $\mathsf{ownedptrs}(l_1/l_2)$, will take two arguments. The first, l_1, is a list of pointer types that have already been returned, represented as a list of the pointer Vals, offsets, and content types. The second, l_2, is a list of pointer types that are still outstanding, represented as a list of the pointer Vals and offsets. Since these pointer types have True content types, we do not need to include those in the list. For the already-returned pointer types, the j-th one has content type $T_j : \mathsf{PType}(\mathsf{Val}, A_j)$ of each type, and the ownedptrs type relates the pointer in l_1 to the corresponding specification value of type A_j on the specification side.

$$\mathsf{ownedptrs}((p_1, o_1, T_1), \ldots, (p_i, o_i, T_i) / (p_{i+1}, o_{i+1}), \ldots, (p_n, o_n)) : \mathsf{PType}(\mathbb{N}, A_1 \times \cdots \times A_i)$$

$$\begin{aligned}
&\ell : \mathsf{ownedptrs}((p_1, o_1, T_1), \ldots, (p_i, o_i, T_i) / (p_{i+1}, o_{i+1}), \ldots, (p_n, o_n)) \triangleright (x_1, \ldots, x_i) \stackrel{\mathsf{def}}{=} \\
&\quad (p_1, \ldots, p_i) : ([\ell]ptr((read, o_1) \mapsto T_1) \otimes \cdots \otimes [\ell]ptr((read, o_i) \mapsto T_i)) \triangleright (x_1, \ldots, x_i) * \\
&\quad \mathsf{lowned}(\ell, \\
&\qquad (p_1 : [\ell]ptr((read, o_1) \mapsto \mathsf{True}) \triangleright \mathsf{unit}) * \ldots * (p_n : [\ell]ptr((read, o_n) \mapsto \mathsf{True}) \triangleright \mathsf{unit}) \multimap \\
&\qquad (p_1 : ptr((rw, o_1) \mapsto \mathsf{True}) \triangleright \mathsf{unit}) * \ldots * (p_n : ptr((rw, o_n) \mapsto \mathsf{True}) \triangleright \mathsf{unit}))
\end{aligned}$$

We can then prove the following rule for returning one of the outstanding pointer types, where the pointer information is moved from the unreturned list to the returned list, and its content type is also moved to the returned list.

Theorem 7.7 (RETURNL)**.**

$$\frac{}{\begin{array}{l} p : [\ell]ptr((read, o) \mapsto T) \triangleright x \ast \\ \ell : \mathsf{ownedptrs}(l_1 / l_2 \mathbin{+\!\!+} \langle (p, o) \rangle) \triangleright xs \end{array} \sqsubseteq \ell : \mathsf{ownedptrs}(l_1 \mathbin{+\!\!+} \langle (p, o, T) \rangle / l_2) \triangleright (xs, x)}$$

While this rule has the pointer information for the pointer type as the *last* element of the unreturned list, this list can be reordered without issue, since they are just representing permission sets combined with $\ast$ internally. The rule then returns this pointer type to the ownedptrs type by consuming it, moving the pointer information to the returned list and adding its content type to the returned list as well. The specification value x it was associated with is also added to the tuple of specification values that the ownedptrs type is associated with.

Next, we prove the rule for splitting a pointer type:

Theorem 7.8 (SPLITL)**.**

$$\frac{}{\begin{array}{l} p : ptr((write, o) \mapsto T) \triangleright x \ast \\ \ell : \mathsf{ownedptrs}(l_1 / l_2) \triangleright xs \end{array} \sqsubseteq \begin{array}{l} p : [\ell]ptr((write, o) \mapsto T) \triangleright x \ast \\ \ell : \mathsf{ownedptrs}(l_1 / l_2 \mathbin{+\!\!+} \langle (p, o) \rangle) \triangleright xs \end{array}}$$

The proof of this rule relies on Theorem 7.2 to perform the split at the permission level. The pointer type is then wrapped inside a when for the rest of the lifetime, and this also adds the pointer information for the pointer type to the list of types yet to be returned to the ownedptrs type.

Next we can prove the rule for starting a new lifetime, which gives us an ownedptrs type with nothing in its lists:

Theorem 7.9 (STARTL)**.**

$$\frac{}{\Pi_{\mathsf{lifetime}} \vdash \mathsf{start} \lessapprox \mathsf{ret\ unit} : \mathsf{ownedptrs}(\langle\rangle / \langle\rangle) \oslash \Pi_{\mathsf{lifetime}}}$$

Note that the product type on the specification side was defined recursively, and has a base case where if we are taking the product of zero elements, then its type is Unit. This is reflected here in the specification program, but this extra Unit type (and corresponding unit value) will be omitted in other definitions for simplicity.

For the final rule for ending a lifetime and regaining the stored pointer types, we first need a permission type to apply a finished modifier to all the pointer types combined together using $\otimes$:

$$\mathsf{finishedptrs}(\langle(p_1, o_1, T_1), \ldots, (p_n, o_n, T_n)\rangle, (x_1, \ldots, x_n)) : \mathsf{PType}(\mathbb{N}, \mathsf{Unit})$$

$$\ell : \mathsf{finishedptrs}(\langle(p_1, o_1, T_1), \ldots, (p_n, o_n, T_n)\rangle, (x_1, \ldots, x_n)) \triangleright \mathsf{unit} \stackrel{\mathsf{def}}{=}$$
$$(p_1, \ldots, p_n) :$$
$$\{\ell\}ptr((write, o_1) \mapsto T_1) \otimes \cdots \otimes \{\ell\}ptr((write, o_n) \mapsto T_n) \triangleright$$
$$(x_1, \ldots, x_n)$$

Now we can prove the rule for ending a lifetime:

Theorem 7.10 (ENDL)**.**

$$\frac{}{\begin{array}{l}\ell : \mathsf{ownedptrs}(\langle(p_1, o_1, T_1), \ldots, (p_n, o_n, T_n)\rangle / \langle\rangle) \triangleright (x_1, \ldots, x_n) \vdash \mathsf{end}\ \ell \lessapprox \mathsf{ret\ unit} \\ : \mathsf{True} \oslash \ell : \mathsf{finishedptrs}(\langle(p_1, o_1, T_1), \ldots, (p_n, o_n, T_n)\rangle, (x_1, \ldots, x_n)) \triangleright \mathsf{unit}\end{array}}$$

This rule says that if all the outstanding pointer types have been returned using RETURNL, then we can end the lifetime and recover the original pointer types, wrapped inside a finishedptrs.

7.5. A Lifetime Example

Now that we have types and typing rules for handling pointer types with lifetimes, we can go through an example of using these rules to temporarily duplicate a pointer type, something we could not do with only the types in Chapter 6. In this example, we will assume there is a program t which requires two values p and p' with pointer-read types. Specifically, we assume it is well-

typed with the following input and output types:

$$\begin{aligned}
&\ell : \mathsf{ownedptrs}(l_1/l_2) \rhd xs \ast \\
&\quad p : [\ell]ptr((read, 0) \mapsto \mathsf{eq}(v)) \rhd \mathsf{unit} \ast \\
&\quad p' : [\ell]ptr((read, 0) \mapsto \mathsf{eq}(v)) \rhd \mathsf{unit} \vdash \\
&t\ p\ p' \rightsquigarrow \\
&\mathsf{ret\ unit} : \\
&\mathsf{True} \oslash \ell : \mathsf{ownedptrs}(l_1/l_2) \rhd xs \ast \\
&\quad p : [\ell]ptr((read, 0) \mapsto \mathsf{eq}(v)) \rhd \mathsf{unit} \ast \\
&\quad p' : [\ell]ptr((read, 0) \mapsto \mathsf{eq}(v)) \rhd \mathsf{unit}
\end{aligned}$$

This program t requires pointer-read types that have already been split for both of its input values. With our new types and typing rules, we can apply t with the same pointer value for both of its arguments, and still be able to use STORE with that pointer later on.

Throughout this typing derivation, we will use pointer rules defined in Chapter 6, but with different types. We hold the pointer types wrapped inside a when, and also hold a ownedptrs type with the same lifetime, telling us that the lifetime for the pointer type is current, so the pointer type inside the when can be used just as a normal pointer type. These alternate versions of the typing rules are proved in the Coq formalization, and we will use them implicitly here.

The overall implementation program we wish to typecheck is

$$\ell \leftarrow \mathsf{start}; t\ p\ p; \mathsf{end}\ \ell; \mathsf{ret}\ \ell.$$

Note that we return the lifetime ℓ manually at the end of this program. This is because we cannot use ℓ directly in the output type, since it was not in scope originally, but was created by the start instruction. By returning ℓ, we can apply the output type to this lifetime.

To typecheck this program, we will prove the following typing judgment, with the lifetime per-

mission set Π_{lifetime} included to allow for lifetime creation:

$$\Pi_{\mathsf{lifetime}} * p : ptr((write, 0) \mapsto \mathsf{Nat}) \triangleright x$$
$$\vdash \ell \leftarrow \mathsf{start}; t\ p\ p; \mathsf{end}\ \ell; \mathsf{ret}\ \ell \lesssim \mathsf{ret\ unit} : \mathsf{finishedptrs}(\langle(p, 0, \mathsf{Nat})\rangle, \langle x\rangle) \oslash \Pi_{\mathsf{lifetime}}$$

Due to the size of the typing derivation, we will describe the major steps and intermediate states of the typing derivation, rather than presenting the all the details or the entire proof tree. We use BIND to typecheck each of the instructions in the implementation separately. The most interesting part is manipulating the types to typecheck the t portion of the program, where we will need to split, then weaken and duplicate the pointer type. Afterwards, we return the pointer type to the ownedptrs type and end the lifetime.

The first instruction we handle is the start instruction. We use BIND to handle this operation separately and FRAME to hide the unnecessary types. This gives us an ownedptrs type for the new lifetime ℓ we get from the instruction.

$$\ell : \mathsf{ownedptrs}(\langle\rangle / \langle\rangle) \oslash (\Pi_{\mathsf{lifetime}} * p : ptr((write, 0) \mapsto \mathsf{Nat}) \triangleright x) \triangleright \mathsf{unit}$$
$$\vdash t\ p\ p; \mathsf{end}\ \ell; \mathsf{ret}\ \ell \lesssim \mathsf{ret\ unit} : \mathsf{finishedptrs}(\langle(p, 0, \mathsf{Nat})\rangle, \langle x\rangle) \oslash \Pi_{\mathsf{lifetime}}$$

Next we need to manipulate the input type in order to get the types needed to typecheck $t\ p\ p$. First, we use CNSQ along with PERMSE to discharge the $\oslash$, and FRAME to hide the now-unneeded Π_{lifetime}. The input type is now

$$\ell : \mathsf{ownedptrs}(\langle\rangle / \langle\rangle) \triangleright \mathsf{unit} * p : ptr((write, 0) \mapsto \mathsf{Nat}) \triangleright x.$$

Next we use SPLITL to split the pointer-write type, then weaken it using PTRWEAK (the version we proved for pointer types inside a when) to a pointer-read type.

$$\ell : \mathsf{ownedptrs}(\langle\rangle / \langle(p, 0)\rangle) \triangleright \mathsf{unit} * p : [\ell]ptr((read, 0) \mapsto \mathsf{Nat}) \triangleright x$$

Then, we use PTRE to move out the content type of Nat, and then we can duplicate the pointer-read type using CNSQ and READDUP.

$$\ell : \mathsf{ownedptrs}(\langle\rangle / \langle(p, 0)\rangle) \triangleright \mathsf{unit} *$$
$$p : [\ell]ptr((read, 0) \mapsto \mathsf{eq}(v)) \triangleright \mathsf{unit} * p : [\ell]ptr((read, 0) \mapsto \mathsf{eq}(v)) \triangleright \mathsf{unit} * v : \mathsf{Nat} \triangleright x$$

Now we have the exact types needed to typecheck t, which we can do after applying BIND and FRAME. The types remain the same after handling t, though we first remove the extra True type and discharge the $\oslash$ connectives. Next, we need to return the pointer-read type to the ownedptrs type, which will let us end the lifetime. We first need to move the Nat type back into the pointer type, which we do using CNSQ and PTRI. We no longer have a use for the second pointer type on p, so we drop it. Even if we kept it, it would not be usable after the lifetime is ended, since the typing rules for pointer types inside when types require an ownedptrs type to be present to ensure that the lifetime is current. Now the input type is back to the same type as in a previous step:

$$\ell : \mathsf{ownedptrs}(\langle\rangle / \langle(p, 0)\rangle) \triangleright \mathsf{unit} * p : [\ell]ptr((read, 0) \mapsto \mathsf{Nat}) \triangleright x$$

At this point, we are ready to use RETURNL.

$$\ell : \mathsf{ownedptrs}(\langle(p, 0, \mathsf{Nat})\rangle / \langle\rangle) \triangleright x$$

Finally, now that the ownedptrs type says every outstanding pointer type has been returned to it, we can handle the end ℓ operation using BIND and STOPL. The final piece of the program we need to typecheck after this is the return:

$$\ell : \mathsf{finishedptrs}(\langle(p, 0, \mathsf{Nat})\rangle, x) \triangleright \mathsf{unit} \vdash \mathsf{ret}\ \ell \lessapprox \mathsf{ret}\ \mathsf{unit} : \mathsf{finishedptrs}(\langle(p, 0, \mathsf{Nat})\rangle, x)$$

We can conclude with an application of the RET rule.

For most programs that use multiple aliasing read-pointers, they can be typechecked by using

a single pointer-write type and simply moving the type around different aliases using equality types. However, even if we know that a program is using pointer aliases, it is not always sufficient to have just one pointer type. For example, applying the ITER rule *consumes* the type relating the starting values for iteration. If the body of an iter uses the starting values again and we need their type again when typechecking the body, then we would need to duplicate this type before applying ITER. One could also imagine other programs that require pointer types for each alias. If we had support for concurrency, for example, then a rule like the parallel composition rule from concurrent separation logic would require each thread to have its own pointer type.

7.6. Differences with Heapster Lifetimes

While we have shown that rely-guarantee permissions can model the core concepts of lifetimes and splitting permissions using these lifetimes, the type system presented in this chapter does not fully capture all the features of lifetimes in Heapster.

Some of these differences are minor and could be easily resolved. For example, we only proved some basic pointer rules when the pointer type is in a when, but did not prove rules for array types, which would be necessary to support crucial operations like malloc and free. If we needed those rules, we could prove them as well for use with lifetimes. After using a lifetime type and ending the lifetime, any types split using the lifetime are then held inside a finished modifier. Similar to the rules for using pointer types when they are inside a when, we could prove rules for using pointer types inside a finished. In fact, we have proven the following theorem, which could be used for this task.

Theorem 7.11. If $\Pi \vdash t_i \lessapprox t_s : T$, then $\{\ell\}\Pi \vdash t_i \lessapprox t_s : \lambda(r_i, r_s).\, \{\ell\}(T\ r_i\ r_s)$

Another small difference is the representation of lifetimes on the specification side. While lifetime operations like start and end are not present in specification programs, just as with our system—the specification values that lifetime values are related to differ. With our definitions, ownedptrs relates a lifetime on the implementation side to a tuple of specification values. The tuple contains all the specification values for the types that have been returned to the ownedptrs type. Heapster,

on the other hand, represents these as a function from the specification values that have *not* been returned yet to the combination of already-returned values and not-yet-returned values. These representations are equivalent, however, and one could be converted to the other.

The ability to handle *nested* lifetimes is a bigger difference that is more difficult to resolve with our current semantics of lifetime types, though this does not seem to be an inherent limitation of the rely-guarantee permission approach. Heapster supports nested lifetimes, where one lifetime is completely subsumed by another—the inner lifetime starts after the outer one, and ends before the outer lifetime. This originates from Rust, where lifetimes are often tied to lexical scope. So one scope that is nested in another would result in a lifetime that is nested in another. This is used heavily in Rust, as any lifetime can be implicitly converted into a lifetime nested within it. In contrast, for the system we presented, we must pick the "right" lifetime upfront when splitting a type, as there is no way to convert to a different lifetime. To represent this nesting, Heapster's owned types have another argument: a list of the lifetimes that are nested in the one that it owns. This determines how lifetimes can be converted, and a lifetime cannot be ended until all of its nested lifetimes are already finished.

We made several attempts to model this feature in the theory we presented in this chapter. One approach to adding this that we tried was to modify the state, by encoding the lifetime hierarchy in the way we store lifetimes. Rather than the list definition of Ltms, we used a rose tree [Mee88] and used the structure of the tree to represent the nesting relationship. This approach, however, required that we know the exact structure of lifetime nesting when we start a lifetime. The "parent" of each lifetime had to be known statically, but Heapster allows the nesting behavior be determined dynamically, after a lifetime is started.

Another approach we used is to encode these nesting relationship as *invariants*. Lifetimes have several properties that would be useful to encode using invariants within rely-guarantee permissions. For instance, when a lifetime ends, the fact that it is finished should be invariant for the rest of the program. With that invariant formalized in the system, we could imagine a rule where $\{\ell\}\Pi$ is proved to be equivalent to Π itself. Nested lifetimes could also be encoded as invariants.

Once one lifetime is determined to be nested within another, this relationship could be added to the invariant and remain true for the remainder of the program. We tried to add invariants to the type system using dependent types to constrain the state type to states where the invariant holds, and strengthen the invariant throughout typechecking. The benefit of this was that the type system would not have to change, as we can reuse all the existing definitions and rules. However, the type system was designed to operate over a consistent state type, and rules that involved changing the state type were not provable.

While the theory of Heapster presented in this chapter does not implement nested lifetimes, we believe that extending rely-guarantee permissions with a fourth component representing the invariant can be used to add nested lifetimes to the lifetime types. We will discuss this future work further in Chapter 8.

A last difference between the treatment of lifetimes in the theory in this chapter and lifetimes in Heapster is the status of lifetimes in the language. In this chapter, lifetime statuses are stored in the state and there are `start` and `end` instructions in the language to manage lifetimes. In Heapster, lifetimes are not part of the language at all, but are only part of the type system. So while in our type system, we have to decide where to insert instructions to start and end lifetimes, in Heapster one has to infer where to apply the typing rules to start and end lifetimes. These approaches are similar, but one possible piece of future work would be to connect them formally. This would first require formalizing the Heapster approach in the type system, then—as with other works using ghost state—we could prove an *erasure* theorem, proving the equivalence of the code with and without the ghost state [FGP14].

CHAPTER 8

Future Work and Conclusion

We have already mentioned some possible areas for future work in this dissertation, like connecting our Coq formalization to other work using ITrees and using rely-guarantee permissions as the semantics for other verification techniques. In this chapter, we consider two more major directions for future work and conclude.

8.1. Concurrency

Since rely-guarantee is a logic for reasoning about concurrency and separation logic has significant applications for concurrency, a natural next step is to extend the theory of Heapster to deal with concurrent code. As Heapster does not yet support concurrent code, this is also an opportunity to use the insights from extending the theory using rely-guarantee permissions to guide the design of the tool.

In fact, there is already some work in this area to suggest that rely-guarantee permissions can be used as the semantics for a type system that supports concurrency. We had previously extended the type system without specification extraction—presented in Chapter 5—with an operation representing parallel composition. To do this, we used an interleaving semantics, as in representations of concurrency like CCS [Mil80], by adding nondeterministic choice to our programs. Then, at each step, a concurrent program would be able to nondeterministically choose which thread to run. We did this by adding an event to represent nondeterministic choice to the ITrees' event type:

$$\mathsf{Or} : \mathsf{E}\ \mathbb{B}.$$

This event gets a boolean value from the environment, representing the nondeterministic choice. With this addition to the event type and after adding it to the steps-to relation, we defined the parallel composition of two programs, $t_1 \parallel t_2$, as a corecursive function. This function uses nondeterminism events in the output ITree to choose which of the two input ITrees to run. Then,

it corecursively inserts nondeterminism events after each node from the input ITrees to choose whether to run t_1 or t_2 next. Effectively, we give the overall program the option to switch threads at every possible point during execution.

With this definition in hand, we proved a parallel composition rule inspired by the analogous one in concurrent separation logic:

$$\frac{\Pi_1 \vdash t_1 : Q_1 \qquad \Pi_2 \vdash t_2 : Q_2}{\Pi_1 * \Pi_2 \vdash t_1 \parallel t_2 : \lambda(r_1, r_2).Q_1\, r_1 * Q_2\, r_2}$$

However, this approach quickly became untenable for the full type system defined in Chapter 6. Defining parallel composition by considering all interleavings is quite low level and proofs involving it are difficult because of this. When we generalized to the full definition of bisimulation with both the implementation and the specification program, relating two parallel programs in the parallel composition rule was too difficult due to the number of possibly-related implementation and specification programs that bisimulation must consider. Up-to techniques for reducing the number of related programs in the bisimulation could help with this problem, but we eventually set aside concurrency for future work.

8.1.1. Designing the Type System

The core question underlying this direction of work is one of design. The work described in this dissertation has only dealt with formalizing and proving semantics for existing types and typing rules in Heapster. For supporting concurrent code, one must instead consider both the semantics and the design of types together.

Since Heapster currently takes input as programs in LLVM IR, one possibility is to provide support for the low-level concurrency primitives in this language, like with pointer types for memory. Extracting useful specifications from such code is likely to be difficult, however. Without being able to represent higher-level concurrency constructs like synchronization primitives and communication between threads, specifications will not be able to express such abstractions either. A

better option may be to focus on specific higher level language features, like how Heapster has special support for Rust lifetimes.

The current direction that Heapster takes is to automate the simple-but-tedious parts of verifying that imperative programs implement functional specifications. For memory-manipulating programs, these are the parts of programs that use pointers safely, which Heapster captures in its type system. We should then find the simple parts of *concurrent* programs that are easy to translate to functional programs. Safe Rust programs again seem like a good candidate, due to the similarity between Rust types and Heapster types.

Well-typed Rust programs guarantee not only memory safety, but also thread safety. This, like for memory safety, is due to its ownership mechanism, which controls aliasing and by doing so, prevents data races. We could then introduce types in Heapster for Rust types used for concurrency, similar to how lifetimes in Heapster are used to support Rust lifetimes. One difference is that Rust lifetimes are solely in the type system, so we did not need to extract anything new to specifications. Concurrent code requires mechanisms for creating new threads and for synchronization, so specifications will need additional components compared to the pure functional programs in this dissertation. Those programs were all of type $\mathsf{itree}\ \mathsf{E}_{\mathsf{Unit}}\ R$—that is, none of those programs included any use of state. However, types to support Rust types for concurrency would require state. For example, the `Arc` type represents an atomic reference counted pointer, used to share ownership between threads. A value of this type would have to be extracted to some similar object on the specification side, which would require state to dynamically update the reference count and ownership information.

Unlike lifetimes, which can be used even for non-Rust code, types for concurrency will be for specific concurrency features, which can differ drastically between different languages. Thus, these design decisions are especially important for the future direction of Heapster.

8.1.2. Representing Concurrent Programs

To support concurrency, we first need to be able to represent concurrent programs in the theory of Heapster.

As described above, using ITrees to represent concurrent programs was not successful, though an option if we want to continue using an ITree-like representation for programs is to use the choice trees (CTrees) extension of ITrees [Cha+23]. CTrees build in a notion of nondeterminism, or *choice*, into the definition of the data structure by replacing the τ node with a node for nondeterministic branching. The library also provides equational reasoning for CTrees, much like the ITree library. As a part of their paper, Chappe et al. developed a case study for cooperative multithreading using CTrees that could be used to represent concurrent programs. With this system, context switching can only happen at specific points, which are marked by events. To represent preemptive multithreading, these events could be added after each node in a thread.

Another benefit of using CTrees is that they subsume ITrees. Chappe et al. prove that ITrees can be represented using CTrees, and as this operation is defined and proved in Coq, we can formally relate the work in this dissertation directly to the new CTrees representation, rather than rebuilding the proofs for the type system using the new library.

A different approach not involving CTrees is to build concurrency directly into the language. For example, a program could be represented as a collection of threads, and the small-step semantics could be updated to handle this. Such a big change would require significant changes to the rest of the semantics, and to definitions like bisimulation and semantic typing.

8.2. Adding Invariants to Rely-Guarantee Permissions

Aside from concurrency, a second major direction of future work is adding a notion of an invariant to the definition of a rely-guarantee permission. As mentioned in Section 7.6, one weakness of our treatment of lifetimes was the lack of support for nested lifetimes, which could be solved by invariants. In addition to being able to define more features for lifetime types, the idea of invariants itself is a natural one to express evolution of the state. For example, invariants could

be used to record the blocks of memory that have been freed, since they cannot be reused in our type system. For future additions to the state, like information about threads, there could also be some natural uses for invariants. While we tried several ways of adding invariants, what seems the most promising is to make the invariant a part of the definition of rely-guarantee permissions, so that a rely-guarantee permission has four instead of three components.

An invariant, once established, should remain true for the rest of the program. This can be formalized by changing how permissions are allowed to change during typechecking. Rather than only allowing permissions to change via $\rightsquigarrow$, we can add the requirement to also strengthen their invariants as they change. While invariants should remain true for the entire program, they are attached to individual permissions, which can always be dropped. This should not be an issue, however, because these invariants can "spread" when we combine permissions. Like how preconditions are combined using intersection in $*$, the invariants can also be combined using intersection, so the overall permission always has the conjunction of all the individual invariants.

By adding invariants, many of our existing definitions would have to change. This simplifies some definitions, since instead of reasoning about all states in the state type, these definitions only have to reason about states that satisfy the invariant. This was needed in the definition of when in Section 7.2. We needed to reason about the case where the lifetime is not started yet by adding this case to the rely and precondition. By adding invariants, we would have an invariant that says the lifetime is either started or finished, so these extra cases would not be necessary. Invariants will also complicate other definitions. For example, separateness would no longer be upward-closed, since weaker permissions would have weaker invariants, and require us to reason about separateness in a bigger state space. Instead, we would have to strengthen the invariant of the weakened permission to retain this useful property.

8.3. Conclusion

In this dissertation we have defined rely-guarantee permissions and used them to prove the soundness of the Heapster type system. We focused on two main concepts in the type system. First, rely-guarantee permissions were used to represent types for controlling memory. Types in

Heapster limit the capabilities of pointer-manipulating programs, resulting in memory safe programs. We defined a semantic representation of these types using rely-guarantee permissions, which relied on using rely-guarantee permissions to to model concepts in separation logic—like separation and separating conjunction—that are used in the type system. As a result of the memory safety guarantees of the type system, typechecking also extracts equivalent functional programs from these pointer-manipulating imperative programs. We showed that rely-guarantee permissions are able to represent the semantics of this relational typing judgment using a bisimulation relation, where rely-guarantee permissions control the behavior of *both* the imperative program and the functional program.

Next, we expanded the type system and focused on types for lifetimes, which allow us to temporarily weaken other types. The relational nature of our approach was crucial for representing lifetime types using rely-guarantee permissions. We defined rely-guarantee permissions for controlling lifetimes which involved fine-grain splitting of other permissions using these lifetime permissions. These permissions formed the semantics of new types and typing rules. We showed that these new types add expressivity to the type system, and we were able to typecheck programs that were not well-typed previously.

BIBLIOGRAPHY

[AJ94] S. Abramsky and A. Jung. "Domain Theory." In: *Handbook of Logic in Computer Science*. Ed. by S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum. Vol. 3. Clarendon Press, 1994, pp. 1–168.

[App11] Andrew W. Appel. "Verified Software Toolchain." In: *Programming Languages and Systems*. Ed. by Gilles Barthe. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 1–17. ISBN: 978-3-642-19718-5.

[Bar+97] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Jean-Christophe Filliatre, Eduardo Gimenez, Hugo Herbelin, Gerard Huet, Cesar Munoz, Chetan Murthy, et al. "The Coq proof assistant reference manual: Version 6.1." PhD thesis. Inria, 1997.

[BCG88] M.C. Browne, E.M. Clarke, and O. Grümberg. "Characterizing finite Kripke structures in propositional temporal logic." In: *Theoretical Computer Science* 59.1 (1988), pp. 115–131. ISSN: 0304-3975. DOI: https://doi.org/10.1016/0304-3975(88)90098-9. URL: https://www.sciencedirect.com/science/article/pii/0304397588900989.

[BCO05] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. "A Decidable Fragment of Separation Logic." In: *FSTTCS 2004: Foundations of Software Technology and Theoretical Computer Science*. Ed. by Kamal Lodaya and Meena Mahajan. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 97–109. ISBN: 978-3-540-30538-5.

[Biz+19] Aleš Bizjak, Daniel Gratzer, Robbert Krebbers, and Lars Birkedal. "Iron: managing obligations in higher-order concurrent separation logic." In: *Proc. ACM Program. Lang.* 3.POPL (Jan. 2019). DOI: 10.1145/3290378. URL: https://doi.org/10.1145/3290378.

[Boy03] John Boyland. "Checking Interference with Fractional Permissions." In: *Static Analysis*. Ed. by Radhia Cousot. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 55–72. ISBN: 978-3-540-44898-3.

[BV14] James Brotherston and Jules Villard. "Parametric completeness for separation theories." In: *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '14. San Diego, California, USA: Association for Computing Machinery, 2014, pp. 453–464. ISBN: 9781450325448. DOI: 10.1145/2535838.2535844. URL: https://doi.org/10.1145/2535838.2535844.

[Cal+15] Cristiano Calcagno, Dino Distefano, Jeremy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. "Moving Fast with Software Verification." In: *NASA Formal Methods*. Ed. by Klaus Havelund, Gerard Holzmann, and Rajeev Joshi. Cham: Springer International Publishing, 2015, pp. 3–11. ISBN: 978-3-319-17524-9.

[Cha+23] Nicolas Chappe, Paul He, Ludovic Henrio, Yannick Zakowski, and Steve Zdancewic. "Choice Trees: Representing Nondeterministic, Recursive, and Impure Programs in Coq." In: *Proc. ACM Program. Lang.* 7.POPL (Jan. 2023). DOI: 10.1145/3571254. URL: https://doi.org/10.1145/3571254.

[Cha24] Arthur Charguéraud. *Separation Logic Foundations*. Ed. by Benjamin C. Pierce. Vol. 6. Software Foundations. Electronic textbook, 2024.

[COY07] Cristiano Calcagno, Peter W. O'Hearn, and Hongseok Yang. "Local Action and Abstract Separation Logic." In: *Proceedings of the 22nd Annual IEEE Symposium on Logic in Computer Science*. LICS '07. USA: IEEE Computer Society, 2007, pp. 366–378. ISBN: 0769529089. DOI: 10.1109/LICS.2007.30. URL: https://doi.org/10.1109/LICS.2007.30.

[CPN98] David G. Clarke, John M. Potter, and James Noble. "Ownership types for flexible alias protection." In: *SIGPLAN Not.* 33.10 (Oct. 1998), pp. 48–64. ISSN: 0362-1340. DOI: 10.1145/286942.286947. URL: https://doi.org/10.1145/286942.286947.

[DGW10] Thomas Dinsdale-Young, Philippa Gardner, and Mark Wheelhouse. "Abstraction and Refinement for Local Reasoning." In: *Verified Software: Theories, Tools, Experiments*. Ed. by Gary T. Leavens, Peter O'Hearn, and Sriram K. Rajamani. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 199–215. ISBN: 978-3-642-15057-9.

[Din+13] Thomas Dinsdale-Young, Lars Birkedal, Philippa Gardner, Matthew Parkinson, and Hongseok Yang. "Views: Compositional Reasoning for Concurrent Programs." In: *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '13. Rome, Italy: Association for Computing Machinery, 2013, pp. 287–300. ISBN: 9781450318327. DOI: 10.1145/2429069.2429104. URL: https://doi.org/10.1145/2429069.2429104.

[Doc+16] Robert Dockins, Adam Foltzer, Joe Hendrix, Brian Huffman, Dylan McNamee, and Aaron Tomb. "Constructing Semantic Models of Programs with the Software Analysis Workbench." In: *Verified Software. Theories, Tools, and Experiments*. Ed. by Sandrine Blazy and Marsha Chechik. Cham: Springer International Publishing, 2016, pp. 56–72. ISBN: 978-3-319-48869-1.

[Dod+09] Mike Dodds, Xinyu Feng, Matthew Parkinson, and Viktor Vafeiadis. "Deny-Guarantee Reasoning." In: *Proceedings of the 18th European Symposium on Programming Languages and Systems: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*. ESOP '09. York, UK: Springer-Verlag, 2009, pp. 363–377. ISBN: 9783642005893. DOI: 10.1007/978-3-642-00590-9_26. URL: https://doi.org/10.1007/978-3-642-00590-9_26.

[Fen09] Xinyu Feng. "Local Rely-Guarantee Reasoning." In: *SIGPLAN Not.* 44.1 (Jan. 2009), pp. 315–327. ISSN: 0362-1340. DOI: 10.1145/1594834.1480922. URL: https://doi.org/10.1145/1594834.1480922.

[FFS07] Xinyu Feng, Rodrigo Ferreira, and Zhong Shao. "On the Relationship Between Concurrent Separation Logic and Assume-Guarantee Reasoning." In: *Programming Languages and Systems*. Ed. by Rocco De Nicola. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 173–188. ISBN: 978-3-540-71316-6.

[FGP14] Jean-Christophe Filliâtre, Léon Gondelman, and Andrei Paskevich. "The Spirit of Ghost Code." In: *Computer Aided Verification*. Ed. by Armin Biere and Roderick Bloem. Cham: Springer International Publishing, 2014, pp. 1–16. ISBN: 978-3-319-08867-9.

[Fos+07] J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. "Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem." In: *ACM Trans. Program. Lang. Syst.* 29.3 (May 2007), 17–es. ISSN: 0164-0925. DOI: 10.1145/1232420.1232424. URL: https://doi.org/10.1145/1232420.1232424.

[Gäh+22] Lennard Gäher, Michael Sammler, Simon Spies, Ralf Jung, Hoang-Hai Dang, Robbert Krebbers, Jeehoon Kang, and Derek Dreyer. "Simuliris: a separation logic framework for verifying concurrent program optimizations." In: *Proc. ACM Program. Lang.* 6.POPL (Jan. 2022). DOI: 10.1145/3498689. URL: https://doi.org/10.1145/3498689.

[GEG13] Colin S. Gordon, Michael D. Ernst, and Dan Grossman. "Rely-Guarantee References for Refinement Types over Aliased Mutable Data." In: *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '13. Seattle, Washington, USA: Association for Computing Machinery, 2013, pp. 73–84. ISBN: 9781450320146. DOI: 10.1145/2491956.2462160. URL: https://doi.org/10.1145/2491956.2462160.

[Hay71] Patrick J. Hayes. *The frame problem and related problems in artificial intelligence*. Tech. rep. Stanford, CA, USA, 1971.

[HBK19] Jonas Kastberg Hinrichsen, Jesper Bengtson, and Robbert Krebbers. "Actris: session-type based reasoning in separation logic." In: *Proc. ACM Program. Lang.* 4.POPL (Dec. 2019). DOI: 10.1145/3371074. URL: https://doi.org/10.1145/3371074.

[He+21] Paul He, Eddy Westbrook, Brent Carmer, Chris Phifer, Valentin Robert, Karl Smeltzer, Andrei Ştefănescu, Aaron Tomb, Adam Wick, Matthew Yacavone, and Steve Zdancewic. "A Type System for Extracting Functional Specifications from Memory-Safe Imperative Programs." In: *Proc. ACM Program. Lang.* 5.OOPSLA (Oct. 2021). DOI: 10.1145/3485512. URL: https://doi.org/10.1145/3485512.

[HFP24] Son Ho, Aymeric Fromherz, and Jonathan Protzenko. *Sound Borrow-Checking for Rust via Symbolic Semantics*. 2024. arXiv: 2404.02680 [cs.PL].

[Hoa69] C. A. R. Hoare. "An Axiomatic Basis for Computer Programming." In: *Commun. ACM* 12.10 (Oct. 1969), pp. 576–580. ISSN: 0001-0782. DOI: 10.1145/363235.363259. URL: https://doi.org/10.1145/363235.363259.

[HP22] Son Ho and Jonathan Protzenko. "Aeneas: Rust verification by functional translation." In: *Proc. ACM Program. Lang.* 6.ICFP (Aug. 2022). DOI: 10.1145/3547647. URL: https://doi.org/10.1145/3547647.

[JB12] Jonas Braband Jensen and Lars Birkedal. "Fictional Separation Logic." In: *Programming Languages and Systems*. Ed. by Helmut Seidl. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 377–396. ISBN: 978-3-642-28869-2.

[Jon83] C. B. Jones. "Tentative Steps toward a Development Method for Interfering Programs." In: *ACM Trans. Program. Lang. Syst.* 5.4 (Oct. 1983), pp. 596–619. ISSN: 0164-0925. DOI: 10.1145/69575.69577. URL: https://doi.org/10.1145/69575.69577.

[Jun+15] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. "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 '15. Mumbai, India: Association for Computing Machinery, 2015, pp. 637–650. ISBN: 9781450333009. DOI: 10.1145/2676726.2676980. URL: https://doi.org/10.1145/2676726.2676980.

[Jun+17] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. "RustBelt: Securing the Foundations of the Rust Programming Language." In: *Proc. ACM Program. Lang.* 2.POPL (Dec. 2017). DOI: 10.1145/3158154. URL: https://doi.org/10.1145/3158154.

[Jun+19] Ralf Jung, Hoang-Hai Dang, Jeehoon Kang, and Derek Dreyer. "Stacked borrows: an aliasing model for Rust." In: *Proc. ACM Program. Lang.* 4.POPL (Dec. 2019). DOI: 10.1145/3371109. URL: https://doi.org/10.1145/3371109.

[KMV15] Johannes Kloos, Rupak Majumdar, and Viktor Vafeiadis. "Asynchronous liquid separation types." In: *29th European Conference on Object-Oriented Programming (ECOOP 2015)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2015.

[KN23] Steve Klabnik and Carol Nichols. *The Rust programming language*. No Starch Press, 2023.

[Kri+21] Siddharth Krishna, Nisarg Patel, Dennis Shasha, and Thomas Wies. "Ghost State." In: *Automated Verification of Concurrent Search Structures*. Cham: Springer International Publishing, 2021, pp. 37–49. ISBN: 978-3-031-01806-0. DOI: 10.1007/978-3-031-01806-0_4. URL: https://doi.org/10.1007/978-3-031-01806-0_4.

[Ler09] Xavier Leroy. "Formal Verification of a Realistic Compiler." In: *Commun. ACM* 52.7 (July 2009), pp. 107–115. ISSN: 0001-0782. DOI: 10.1145/1538788.1538814. URL: https://doi.org/10.1145/1538788.1538814.

[LFF12] Hongjin Liang, Xinyu Feng, and Ming Fu. "A rely-guarantee-based simulation for verifying concurrent program transformations." In: *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '12. Philadelphia, PA, USA: Association for Computing Machinery, 2012, pp. 455–468. ISBN: 9781450310833. DOI: 10.1145/2103656.2103711. URL: https://doi.org/10.1145/2103656.2103711.

[MAC14] Filipe Militão, Jonathan Aldrich, and Luís Caires. "Rely-Guarantee Protocols." In: *ECOOP 2014 – Object-Oriented Programming*. Ed. by Richard Jones. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 334–359. ISBN: 978-3-662-44202-9.

[Mat+22] Yusuke Matsushita, Xavier Denis, Jacques-Henri Jourdan, and Derek Dreyer. "RustHornBelt: a semantic foundation for functional verification of Rust programs with unsafe code." In: *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. PLDI 2022. San Diego, CA, USA: Association for Computing Machinery, 2022, pp. 841–856. ISBN: 9781450392655. DOI: 10.1145/3519939.3523704. URL: https://doi.org/10.1145/3519939.3523704.

[Mee88] Lambert Meertens. "First steps towards the theory of rose trees." In: *CWI, Amsterdam* (1988).

[Mil78] Robin Milner. "A theory of type polymorphism in programming." In: *Journal of Computer and System Sciences* 17.3 (1978), pp. 348–375. ISSN: 0022-0000. DOI: https://doi.org/10.1016/0022-0000(78)90014-4. URL: https://www.sciencedirect.com/science/article/pii/0022000078900144.

[Mil80] Robin Milner. *A calculus of communicating systems*. Springer, 1980.

[MJP20] Glen Mével, Jacques-Henri Jourdan, and François Pottier. "Cosmo: a concurrent separation logic for multicore OCaml." In: *Proc. ACM Program. Lang.* 4.ICFP (Aug. 2020). DOI: 10.1145/3408978. URL: https://doi.org/10.1145/3408978.

[Mou+15] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. "The Lean Theorem Prover (System Description)." In: *Automated Deduction - CADE-25*. Ed. by Amy P. Felty and Aart Middeldorp. Cham: Springer International Publishing, 2015, pp. 378–388. ISBN: 978-3-319-21401-6.

[MTK20] Yusuke Matsushita, Takeshi Tsukada, and Naoki Kobayashi. "RustHorn: CHC-Based Verification for Rust Programs." In: *Programming Languages and Systems*. Ed. by Peter Müller. Cham: Springer International Publishing, 2020, pp. 484–514. ISBN: 978-3-030-44914-8.

[Nak+24] Takashi Nakayama, Yusuke Matsushita, Ken Sakayori, Ryosuke Sato, and Naoki Kobayashi. "Borrowable Fractional Ownership Types for Verification." In: *Verification, Model Checking, and Abstract Interpretation*. Ed. by Rayna Dimitrova, Ori Lahav, and Sebastian Wolff. Cham: Springer Nature Switzerland, 2024, pp. 224–246. ISBN: 978-3-031-50521-8.

[NMB08] Aleksandar Nanevski, Greg Morrisett, and Lars Birkedal. "Hoare Type Theory, Polymorphism and Separation." In: *J. Funct. Program.* 18.5–6 (Sept. 2008), pp. 865–911. ISSN: 0956-7968. DOI: 10.1017/S0956796808006953. URL: https://doi.org/10.1017/S0956796808006953.

[OG76] Susan Owicki and David Gries. "An Axiomatic Proof Technique for Parallel Programs I." In: *Acta Inf.* 6.4 (Dec. 1976), pp. 319–340. ISSN: 0001-5903. DOI: 10.1007/BF00268134. URL: https://doi.org/10.1007/BF00268134.

[OHe07] Peter W. O'Hearn. "Resources, Concurrency, and Local Reasoning." In: *Theor. Comput. Sci.* 375.1–3 (Apr. 2007), pp. 271–307. ISSN: 0304-3975. DOI: 10.1016/j.tcs.2006.12.035. URL: https://doi.org/10.1016/j.tcs.2006.12.035.

[Par10] Matthew Parkinson. "The Next 700 Separation Logics." In: *Verified Software: Theories, Tools, Experiments*. Ed. by Gary T. Leavens, Peter O'Hearn, and Sriram K. Rajamani. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 169–182. ISBN: 978-3-642-15057-9.

[Pea21] David J. Pearce. "A Lightweight Formalism for Reference Lifetimes and Borrowing in Rust." In: *ACM Trans. Program. Lang. Syst.* 43.1 (Apr. 2021). ISSN: 0164-0925. DOI: 10.1145/3443420. URL: https://doi.org/10.1145/3443420.

[PP13] François Pottier and Jonathan Protzenko. "Programming with Permissions in Mezzo." In: *SIGPLAN Not.* 48.9 (Sept. 2013), pp. 173–184. ISSN: 0362-1340. DOI: 10.1145/2544174.2500598. URL: https://doi.org/10.1145/2544174.2500598.

[PPS22] Étienne Payet, David J. Pearce, and Fausto Spoto. "On the Termination of Borrow Checking in Featherweight Rust." In: *NASA Formal Methods*. Ed. by Jyotirmoy V. Deshmukh, Klaus Havelund, and Ivan Perez. Cham: Springer International Publishing, 2022, pp. 411–430. ISBN: 978-3-031-06773-0.

[Rey02] J.C. Reynolds. "Separation logic: a logic for shared mutable data structures." In: *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*. 2002, pp. 55–74. DOI: 10.1109/LICS.2002.1029817.

[Rey78] John C. Reynolds. "Syntactic control of interference." In: *Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL '78. Tucson, Arizona: Association for Computing Machinery, 1978, pp. 39–46. ISBN:

9781450373487. DOI: 10.1145/512760.512766. URL: https://doi.org/10.1145/512760.512766.

[Sam+21] Michael Sammler, Rodolphe Lepigre, Robbert Krebbers, Kayvan Memarian, Derek Dreyer, and Deepak Garg. "RefinedC: automating the foundational verification of C code with refined ownership types." In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. PLDI 2021. Virtual, Canada: Association for Computing Machinery, 2021, pp. 158–174. ISBN: 9781450383912. DOI: 10.1145/3453483.3454036. URL: https://doi.org/10.1145/3453483.3454036.

[Sie+15] Filip Sieczkowski, Kasper Svendsen, Lars Birkedal, and Jean Pichon-Pharabod. "A separation logic for fictional sequential consistency." In: *European Symposium on Programming Languages and Systems*. Springer. 2015, pp. 736–761.

[Sil+23a] Lucas Silver, Paul He, Ethan Cecchetti, Andrew K. Hirsch, and Steve Zdancewic. "Semantics for Noninterference with Interaction Trees." In: *37th European Conference on Object-Oriented Programming (ECOOP 2023)*. Ed. by Karim Ali and Guido Salvaneschi. Vol. 263. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023, 29:1–29:29. ISBN: 978-3-95977-281-5. DOI: 10.4230/LIPIcs.ECOOP.2023.29. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ECOOP.2023.29.

[Sil+23b] Lucas Silver, Eddy Westbrook, Matthew Yacavone, and Ryan Scott. "Interaction Tree Specifications: A Framework for Specifying Recursive, Effectful Computations That Supports Auto-Active Verification." In: *37th European Conference on Object-Oriented Programming (ECOOP 2023)*. Ed. by Karim Ali and Guido Salvaneschi. Vol. 263. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023, 30:1–30:26. ISBN: 978-3-95977-281-5. DOI: 10.4230/LIPIcs.ECOOP.2023.30. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ECOOP.2023.30.

[Sil23] Lucas Silver. "Interaction Trees and Formal Specifications." PhD thesis. University of Pennsylvania, 2023.

[SLL20] Ayesha Sadiq, Yuan-Fang Li, and Sea Ling. "A survey on the use of access permission-based specifications for program verification." In: *Journal of Systems and Software* 159 (2020), p. 110450. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2019.110450. URL: https://www.sciencedirect.com/science/article/pii/S0164121219302249.

[Swa+16] Nikhil Swamy, Cătălin Hriţcu, 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. "Dependent types and multi-monadic effects in F*." In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*.

POPL '16. St. Petersburg, FL, USA: Association for Computing Machinery, 2016, pp. 256–270. ISBN: 9781450335492. DOI: 10.1145/2837614.2837655. URL: https://doi.org/10.1145/2837614.2837655.

[Tar55] Alfred Tarski. "A lattice-theoretical fixpoint theorem and its applications." In: *Pacific journal of Mathematics* 5.2 (1955), pp. 285–309.

[Ull16] Sebastian Ullrich. "Simple Verification of Rust Programs via Functional Purification." MA thesis. Karlsruhe Institute of Technology, 2016.

[VB23] Simon Friis Vindum and Lars Birkedal. "Spirea: A Mechanized Concurrent Separation Logic for Weak Persistent Memory." In: *Proc. ACM Program. Lang.* 7.OOPSLA2 (Oct. 2023). DOI: 10.1145/3622820. URL: https://doi.org/10.1145/3622820.

[VP07] Viktor Vafeiadis and Matthew Parkinson. "A Marriage of Rely/Guarantee and Separation Logic." In: *Proceedings of the 18th International Conference on Concurrency Theory*. CONCUR'07. Lisbon, Portugal: Springer-Verlag, 2007, pp. 256–271. ISBN: 3540744061.

[VP23] Paulo Emílio de Vilhena and François Pottier. "A Type System for Effect Handlers and Dynamic Labels." In: *Programming Languages and Systems*. Ed. by Thomas Wies. Cham: Springer Nature Switzerland, 2023, pp. 225–252. ISBN: 978-3-031-30044-8.

[WF94] A.K. Wright and M. Felleisen. "A Syntactic Approach to Type Soundness." In: *Inf. Comput.* 115.1 (Nov. 1994), pp. 38–94. ISSN: 0890-5401. DOI: 10.1006/inco.1994.1093. URL: https://doi.org/10.1006/inco.1994.1093.

[Xia+19] Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. "Interaction Trees: Representing Recursive and Impure Programs in Coq." In: *Proc. ACM Program. Lang.* 4.POPL (Dec. 2019). DOI: 10.1145/3371119. URL: https://doi.org/10.1145/3371119.

[Yan07] Hongseok Yang. "Relational separation logic." In: *Theor. Comput. Sci.* 375.1–3 (Apr. 2007), pp. 308–334. ISSN: 0304-3975. DOI: 10.1016/j.tcs.2006.12.036. URL: https://doi.org/10.1016/j.tcs.2006.12.036.

[Zak+21] Yannick Zakowski, Calvin Beck, Irene Yoon, Ilia Zaichuk, Vadim Zaliva, and Steve Zdancewic. "Modular, compositional, and executable formal semantics for LLVM IR." In: *Proc. ACM Program. Lang.* 5.ICFP (Aug. 2021). DOI: 10.1145/3473572. URL: https://doi.org/10.1145/3473572.

[Zha+21] Hengchu Zhang, Wolf Honoré, Nicolas Koh, Yao Li, Yishuai Li, Li-Yao Xia, Lennart Beringer, William Mansky, Benjamin Pierce, and Steve Zdancewic. "Verifying an HTTP Key-Value Server with Interaction Trees and VST." In: *12th International Conference on Interactive Theorem Proving (ITP 2021)*. Ed. by Liron Cohen and Cezary Kaliszyk. Vol. 193. Leibniz International Proceedings in Informatics

(LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021, 32:1–32:19. ISBN: 978-3-95977-188-7. DOI: 10.4230/LIPIcs.ITP.2021.32. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ITP.2021.32.