MONADIC AND SUBSTRUCTURAL TYPE SYSTEMS FOR REGION-BASED MEMORY MANAGEMENT

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Matthew Thomas Fluet

January 2007

MONADIC AND SUBSTRUCTURAL TYPE SYSTEMS FOR REGION-BASED MEMORY MANAGEMENT

Matthew Thomas Fluet, Ph.D.

Cornell University 2007

Region-based memory management is a scheme for managing dynamically allocated data. A defining characteristic of region-based memory management is the bulk deallocation of data, which avoids both the tedium of `malloc`/`free` and the overheads of a garbage collector. Type systems for region-based memory management enhance the utility of this scheme by statically determining when a program is guaranteed to not perform any erroneous region operations.

We describe three type systems for region-based memory management:

- a type-and-effect system (à la the Tofte-Talpin region calculus);
- a novel monadic type system;
- a novel substructural type system.

We demonstrate how to successively encode the type-and-effect system into the monadic type system and the monadic type system into the substructural type system. These type systems and encodings support the argument that the type-and-effect systems that have traditionally been used to ensure the safety of region-based memory management are neither the simplest nor the most expressive type systems for this purpose.

The monadic type system generalizes the state monad of Launchbury and Peyton Jones and demonstrates that the well-understood parametric polymorphism of System F provides sufficient encapsulation to ensure the safety of region-based memory management. The essence of the first encoding is to translate effects to an indexed monad, trading the subtleties of a type-and-effect system for the simplicity of a monadic type system.

However, both the type-and-effect system and the monadic type system require that regions have nested lifetimes, following the lexical scope of the program, restricting when data may be effectively reclaimed. Hence, we introduce a substructural type system that eliminates the nested-lifetimes requirement. The key idea is to introduce first-class capabilities that mediate access to a region and to provide separate primitives for creating and destroying regions. The essence of the second encoding is to "break open" the monad to reveal its store-passing implementation.

Finally, we show that the substructural type system is expressive enough to faithfully encode other advanced memory-management features.

BIOGRAPHICAL SKETCH

Matthew Fluet graduated from Harvey Mudd College (Claremont, CA) in 1999 with a Bachelor's of Science degree. At Harvey Mudd, he majored in Mathematics and his humanities concentration was in Medieval Studies. Dr. Arthur Benjamin oversaw his senior thesis work. In the fall of 1999, Matthew began graduate studies at Cornell University (Ithaca, NY), where his major field is Computer Science and his minor field is English and Medieval Studies. Dr. Greg Morrisett advised his doctoral research in Computer Science, with a research focus on programming languages. Matthew is completing the requirements for the degree of Doctor of Philosophy in the summer and fall of 2006. In the fall of 2006, he will begin a position as Research Assistant Professor at the Toyota Technological Institute (Chicago, IL).

ACKNOWLEDGEMENTS

Portions of this material is based upon work supported by National Science Foundation under Grant No. 0204193 and Grant No. 9875536, by the Air Force Office of Scientific Research under Award No. F49620-03-1-0156 and Award No. F49620-01-1-0298, and by the Office of Naval Research under Award No. N00014-01-1-0968. Any opinions, findings, and conclusions or recommendations expressed in this dissertation are those of the author and do not necessarily reflect the views of these organizations or the U.S. Government.

First and foremost, thanks and acknowledgments are extended to my advisor, Dr. Greg Morrisett, for direction, guidance, and collaboration during my graduate studies. Thanks and acknowledgments are also extended to Amal Ahmed and Dan Wang, with whom I collaborated on portions of the work appearing in this dissertation.

I would also like to extend my thanks and acknowledgments to Stephen Weeks, Suresh Jagannathan, Henry Cejtin, Riccardo Pucella, Christoph Kreitz, Robert Constable, Stuart Allen, and Kevin Donnelly, with whom I have had the pleasure of collaborating on other projects during my graduate studies.

Many thanks go to the Division of Engineering and Applied Sciences at Harvard University; I am grateful for the opportunity to have spent time there with the members of the Computer Science program and to have been warmly welcomed as a visiting student/researcher. Dr. Norman Ramsey, especially, has been a great source of inspiration and advice.

My office mates and fellow students at Cornell and Harvard have been, without exception, the finest of companions on this journey: Amal Ahmed, Michael Clarkson, João Dias, Paul Govereau, Kim Hazelwood, Kelly Heffner, David Malan,

Aleksander Nanevski, Nate Nystrom, Riccardo Pucella, Kevin Redwine, Yanling Wang, Vicky Weissman.

The Cornell Catholic Community and the Cornell Catholic Grads group provided much appreciated spiritual guidance and support throughout my time at Cornell. They will always hold a special place in my memories of Cornell.

Finally, I am forever grateful for all the support and encouragement from my fiancée, Kimberly, and my parents, Marcia and Thomas. Thank you so much for being there for me when there were difficult times.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1

Introduction

Memory is an essential resource used by computer programs to carry out computations. Almost all data manipulated by a computer program must be represented (in some fashion) in memory. While today's desktop computers come equipped with more memory than ever before, memory remains a finite resource. In order to achieve good overall system performance, a program should acquire and release memory as needed, thereby leaving excess memory available to other programs. In today's embedded computer systems (cell-phones, media players, etc.), the finiteness of memory is felt more acutely, as these systems come equipped with significantly less memory than a typical desktop computer system; correspondingly, in these systems, it is even more desirable that a program not retain memory for data it no longer needs.

We refer to the process of acquiring memory for data as *allocation* and the process of releasing memory for data as *deallocation*. Typically, a program will *dynamically* allocate and deallocate data, ideally retaining only the memory it needs for future computation. Conceptually, both allocation and deallocation are straightforward: acquire memory when new a data object is needed, release memory when an old data object is no longer needed. In practice, though, it can be difficult to know precisely when data are not needed for future computation. Furthermore, accessing memory after it has been released (equivalently, accessing a data object after it has been deallocated) is a program error. This kind of program error may manifest itself in an obvious manner as the abnormal termination of the program or in a more subtle manner as the corruption of data; the latter occurs

when the program accesses memory that has been released but then reacquired for the allocation of a new data object.

In order to structure the allocation and deallocation of data in a program, a variety of memory-management schemes have been developed. For example, in the C programming language, a programmer explicitly manages memory, using the functions `malloc` to allocate new data and `free` to deallocate old data. Releasing memory in this scheme can be both tedious and error prone; failure to deallocate a data object that is no longer needed may lead to a *memory leak*, whereby a program retains more memory than necessary; accidentally deallocating a data object that is needed may lead to a program error.

Another common memory-management scheme is to use a garbage collector to automatically deallocate data during the execution of a program. In this scheme, a programmer allocates new data, but never explicitly deallocates old data. Rather, the garbage collector periodically makes a conservative estimate of the data needed by the program, and deallocates any data determined to be no longer needed. Releasing memory in this scheme is convenient and safe (because the garbage collector's conservative estimate of data needed by the program ensures that the program never accesses data after it has been deallocated); however, using a garbage collector incurs some overheads. Additional execution time is required to estimate the data needed by a program; additional memory is required to represent data managed by the garbage collector; finally, the conservative nature of the garbage collector's estimate of data needed by the program means that the garbage collector may retain more memory than necessary (i.e., a memory leak).

Since both explicit memory management and automatic memory management have different advantages and disadvantages, a better situation would be one where

a programmer may freely choose among memory management schemes. The best situation would be one that additionally avoids memory leaks, by ensuring that memory for data is eventually deallocated, and avoids program errors, by ensuring that deallocated memory for data is not accessed. A compile-time static analysis is a convenient way to inform a programmer about potential memory leaks and program errors. While there are a variety of possible static analyses, the use of a static type system has a number of advantages. A type system is naturally compositional, leading to checking of programs in a modular fashion. A type system ensures that a well-typed program is necessarily error free, rather than detecting only some of the potential errors. Finally, a type system integrates program properties into the programming language, rather than leaving program analysis to a separate, extra-linguistic mechanism.

The work in this dissertation has been motivated by the desire to realize this combination of flexible memory management along with strong static guarantees enforced by type systems. We take as our starting point a third memory-management scheme: region-based memory management. It stands in contrast to explicit memory management using operations like `malloc` and `free` and to fully automatic memory management using a garbage collector. In a program using region-based memory management, a *region* is a collection of allocated data and the corresponding acquired memory. During the program's execution, it creates and destroys regions in order to acquire and release memory. A region is created empty; once a region is created, data may be allocated in and read from the region. When a region is destroyed, all data in the region are deallocated and the corresponding memory is released. Hence, the program's acquired memory corresponds to a collection of regions.

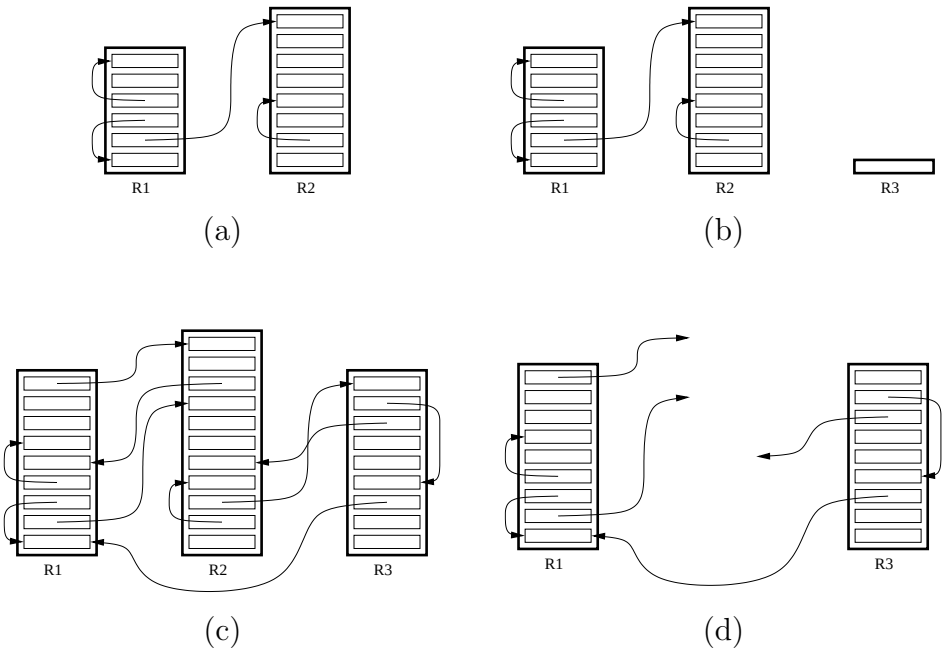

Figure 1.1: Region-based memory management example

Figure 1.1 shows the progression of a typical program's memory when using region-based memory management. Each large box represents a region; each smaller box represents a data object allocated in a region; each arrow represents a pointer (or reference) from one data object to another. Note that there may be both pointers from data in one region to data in another region and pointers from data in one region to data in the same region.

Figure 1.1(a) shows an initial allocation of data in regions. In Figure 1.1(b), a new, empty region R3 is created. In Figure 1.1(c), the program has allocated more data in the three regions. In Figure 1.1(d), the program has destroyed the region R2, deallocating all of the data in the region. Note that destroying R2 has led to *dangling pointers*: pointers to data that has been deallocated. While the existence of dangling pointers during a program's execution is not an error, dereferencing such pointers (that is, attempting to access the deallocated data) is an error.

Region-based memory management comes with both advantages and disadvantages. The performance of region-based memory management has been shown to be competitive with (or better than) other memory-management schemes for certain classes of programs [24, 6, 44]. This follows from the fact that the operations for memory management (create a region, destroy a region, and allocate a data object in a region) can be implemented efficiently. Regions provide a compelling alternative to garbage collection, by avoiding some of the overheads incurred by garbage collection. It has the advantage of supporting bulk deallocation of data, which avoids the tedium of using `free` to deallocate individual units of data; bulk deallocation may also be more efficient than individual deallocation. Region-based memory management also has the advantage of allowing dangling pointers, which can lead to better memory usage than that achieved by using a garbage collector, which does not allow dangling pointers. However, dangling pointers also present a disadvantage: a programmer must be careful to never dereference a dangling pointer, since doing so would be an error; it corresponds to an attempt to access deallocated data. A programmer must also avoid other, less obvious, errors, such as allocating in a destroyed region and destroying a region more than once.

We say that a region which has been created and not yet destroyed is *live*. Correspondingly, we say that a region which has been destroyed is *dead*. A region's *lifetime* refers to the time starting when the region is created and ending when the region is destroyed. Note that in order to dereference a pointer without errors, the region in which the pointed-to unit of data is allocated must be live. Similarly, in order to destroy a region without errors, the region must be live.

Type systems for programming languages have proven to be extremely effective at statically determining when a program is guaranteed not to perform erroneous

operations. Therefore, it comes at no surprise that researchers have proposed type systems for region-based memory management. Such type systems are designed to ensure *region safety*, which guarantees that there is no access to a region (for allocating in or reading from the region) before it is created or after it is destroyed.

The Tofte-Talpin region calculus [79, 80] introduced one of the first type systems for region-based memory management. In their calculus, regions are created and destroyed with a lexically-scoped construct:

$$\texttt{letregion}\ \rho\ \texttt{in}\ e$$

In this construct, a region corresponding to ρ is created when the expression starts evaluating; while the expression evaluates (in particular, during the evaluation of the sub-expression e), data can be allocated in and read from the region ρ; when e has been evaluated to a value, the region ρ is destroyed and the value is returned. Note that the Tofte-Talpin region calculus restricts region-based memory management in the following manner: when a region is destroyed, it must be the most recently created (and not yet destroyed) region. Hence, the collection of live regions may be organized as a stack, with the most recently created region at the top; furthermore, regions must have nested lifetimes: if two regions have overlapping lifetimes, then the lifetime of one must encompass the lifetime of the other. Nonetheless, the Tofte-Talpin region calculus does allow dangling pointers.

The key contribution of the Tofte-Talpin region calculus was a *type-and-effect* system that ensures the region safety of the language; in particular, it statically detects and rejects programs that would dereference dangling pointers. The type-and-effect system introduces a typing judgment $\Gamma \vdash e : \tau, \phi$, which reads "in the environment Γ, the expression e has the type τ and the effect ϕ." The effect ϕ describes the regions that may be allocated in and read from when the expression

is evaluated; hence, it describes those regions that must be live in order to evaluate the expression without errors.

Variations on the Tofte-Talpin region calculus and type-and-effect system have been used in a number of projects. The ML-Kit compiler [78] uses automatic region inference to translate Standard ML programs into executables that use region-based memory management instead of a garbage collector.

The Cyclone language [29], a type-safe dialect of C, uses regions as an organizing principle for memory management. The initial design of Cyclone was based up on the region calculus and type-and-effect system of Tofte-Talpin. This initial design included various kinds of regions, including: lexical regions, corresponding to the Tofte-Talpin `letregion` ρ `in` e construct, and a heap region, which is created when the program starts and is never destroyed, but data allocated in this region is garbage collected. Furthermore, the type-and-effects system of Cyclone extends that of the Tofte-Talpin region calculus with a form of region pointer subtyping — pointers into a region whose lifetime encompasses the lifetime of a second region can be safely treated as pointers into the second region.

Unfortunately, the nested lifetimes of lexically-scoped regions place severe restrictions on when data and memory can be effectively reclaimed. Many programs using the lexical regions available in Cyclone result in (unbounded) memory leaks when compared to the same programs using a garbage collector. For example, a loop that allocates data each iteration and uses that data only in the next iteration cannot be executed with a fixed amount of acquired memory under the nested-lifetimes regime.

To address these concerns, later versions of Cyclone have added a number of new memory management features [77], including *dynamic regions* and *unique point-*

ers that provide more control over memory. Dynamic regions are not restricted to nested lifetimes and can be treated as first-class objects; essentially, dynamic regions may be created and destroyed by a program in an arbitrary order. They are particularly well suited for iterative computations, continuation-passing style computations, and event-based servers where lexical regions do not suffice. Unique pointers are essentially lightweight, dynamic regions that hold exactly one object. The efficacy of these new memory management features has been justified [44], by analyzing a range of applications, including a streaming media server and a space-conscious web server.

Unfortunately, the type-and-effect system of the Tofte-Talpin region calculus is relatively complicated. At the type level, it introduces new syntactic classes for regions and effects. Effects are meant to be treated as sets of regions, so standard term equality no longer suffices for type checking. Finally, the typing rule for `letregion` is extremely subtle because of the interplay of dangling pointers and effects. Indeed, over the past few years, a number of papers have been published attempting to simplify or at least clarify the soundness of the construct [16, 5, 39, 10, 11, 41].

All of these problems are amplified in Cyclone because the additional features make the meta-theory considerably more complicated. Indeed, while the soundness of Cyclone's initial design (with lexical regions and region pointer subtyping) has been established [30], an argument that justifies the soundness of the new memory management features has proved elusive, due to sheer complexity. Much of the complexity arises from the presence of related, but subtly different, features.

Thus, we may identify two major disadvantages in the present state of type systems for region-based memory management. First, the traditional type-and-effect

systems are complicated, both from the perspective of a programmer (who must understand the meaning of the type-and-effect system) and from the perspective of a language designer (who must prove the soundness of the type-and-effect system). Second, the traditional limitation to lexically-scoped regions with nested lifetimes restricts the applications that may effectively use region-based memory management; furthermore, generalizing a type-and-effect system to handle non-lexically-scoped regions results in an even more complicated type-and-effect system.

Therefore, the goal of this work is to find simpler and more expressive accounts of type systems for region-based memory management. In particular, we wish to explain the type soundness of languages like the Tofte-Talpin region calculus and Cyclone via translation to target languages with simpler type systems that nonetheless provide all of the power and safety of region-based memory management with type-and-effect systems.

1.1 Summary

The central thesis of this dissertation, then, is that the type-and-effect systems that have traditionally been used to ensure the safety of region-based memory management are neither the only nor the simplest systems for this purpose. We propose that monadic and substructural type systems give rise to simpler, more expressive, and more uniform languages that continue to provide the power and safety of region-based memory management.

In order to substantiate this claim, we define two languages with novel type systems that ensure the safety of region-based memory management:

- the $\mathsf{F}^{\mathsf{RGN}}$ language, with a monadic type system, in which monadic encapsulation ensures the safety of region-based memory management operations;

- the rgnURAL language, with a substructural type system, in which linear capabilities ensure the safety of region-based memory management operations.

The first major technical contribution of this work is the design of these languages and their respective monadic and substructural type systems; we believe that the type systems for these languages are simpler than the type-and-effect systems previously proposed. We have proven the soundness of these type systems, thereby establishing that these languages ensure the safety of their respective region-based memory management operations.

The monadic language ($\mathsf{F}^{\mathsf{RGN}}$) is inspired by the design of the ST monad of Launchbury and Peyton Jones [56, 55], which is used to encapsulate a "stateful" computation within a pure functional language. We introduce a monadic type, $\mathsf{RGN}\ \theta\ \tau$, as the type of a computation which transforms a stack of regions indexed by θ and delivers a value of type τ; the index θ denotes the stack of regions which are live during the computation. The key element in the design of $\mathsf{F}^{\mathsf{RGN}}$ is the introduction of terms that witness the relationship between the lifetimes of lexically-scoped regions. These terms provide the evidence needed to safely "shift" computations between regions with nested lifetimes. The safety of the language relies upon the familiar parametric polymorphism of System F.

The ideas of the monadic type system for $\mathsf{F}^{\mathsf{RGN}}$ have been adapted by other to manage file handles and database resources in the Haskell programming language [51].

The substructural language (rgnURAL) is inspired by the design of linear type systems [85, 65], the Calculus of Capabilities [90], and Alias Types [75, 91], each of which is concerned with the ways in which resources are used in programs. We introduce primitives for separately creating and destroying regions; these primitives

allow regions to have non-nested lifetimes and, hence, they are more powerful than lexically-scoped regions alone. The key element in the design of rgnURAL is the introduction of a type, $\overline{\mathsf{Cap}}\ \rho$, as the type of a capability that mediates access to a region (for allocating data in and reading data from the region and for destroying the region). A capability provides evidence that its corresponding region is live. The safety of the language relies upon a substructural type system that ensures that all capabilities for a region are consumed when the region is destroyed.

Many of the ideas of the substructural type system for rgnURAL have been adapted in the advanced memory management features of the Cyclone language (see Chapter 5). Similar ideas have been exploited in other systems that are concerned with the ways in which resources are used in programs, particular in the Vault language [18, 20], the Singularity project [46, 19], and for certified inline reference monitoring [33, 34].

We believe that the descriptions of $\mathsf{F}^{\mathsf{RGN}}$ and rgnURAL to be given in subsequent chapters will develop sufficient intuition to reasonably establish our goal of finding simpler accounts of type systems for region-based memory management. However, while $\mathsf{F}^{\mathsf{RGN}}$ and rgnURAL will share many operational similarities with other languages providing region-based memory management (e.g., evaluation with a collection of regions), their type systems will appear to be quite different from type-and-effect systems. Hence, we may wonder if the simplicity of the $\mathsf{F}^{\mathsf{RGN}}$ and rgnURAL type systems point to some deficiency, failing to capture all of the idioms available in type-and-effect systems for region-based memory management.

The second major technical contribution of this work is to demonstrate that we have lost no expressive power by adopting the type systems of $\mathsf{F}^{\mathsf{RGN}}$ and rgnURAL. To justify this claim, we define a language with a traditional type-and-effect system

and we show how this language may be translated to the $\mathsf{F}^{\mathsf{RGN}}$ language and we show how the $\mathsf{F}^{\mathsf{RGN}}$ language may be translated to the rgnURAL language. The first translation shows how monadic encapsulation may be used to eliminate the complexity of type-and-effect systems, while the second translation shows how linear capabilities may be used to eliminate the nested lifetimes of monadic encapsulation. We also sketch the definition of a hybrid monadic and substructural language that captures key features of the Cyclone language and discuss a translation from this hybrid language to rgnURAL. This translation shows that Cyclone's advanced memory management features may be explained in terms of the rgnURAL language.

Throughout this dissertation, we only focus on core languages, suitable for service as a compiler intermediate language or as a vehicle for formal reasoning. These languages are not suitable for service as high-level programming languages, as they lack many features that one would expect from such a programming language. Nonetheless, the $\mathsf{F}^{\mathsf{RGN}}$ and rgnURAL languages serve to isolate the essential aspects of region-based memory management that must be handled by a type system that ensures region safety. Hence, this work furthers the general understanding of type systems in the context of region-based memory management and serves as a useful starting point in the design of future, high-level programming languages that wish to offer region-based memory management as a powerful and safe memory-management scheme.

Furthermore, we may note that the issues that arise with the management of memory using regions also arise in the management of any scarce resource that is used during a computation. There are many sorts of resources that may be acquired and released during the execution of a program: file handles, database connections, concurrency locks, graphics processor texture and shader units, etc.

There are also less tangible, but equally important, "resources" that are used by a program, such as the current state within a network or cryptographic protocol. The techniques developed in this dissertation will be applicable to many resource management problems.

1.2 Outline

Because one of our main goals in this dissertation is to demonstrate that the monadic type system of $\mathsf{F}^{\mathsf{RGN}}$ and the substructural type system of rgnURAL are suitable for encoding traditional type-and-effect systems for region-based memory management, we structure the main body of this dissertation (Chapters 2, 3, and 4) as a sequence of languages and companion type systems; for each language and type system in this sequence, we present a type- and meaning-preserving translation from the previous language in the sequence. These translations demonstrate that each language is at least as expressive as the previous language; hence, they validate the claim that our monadic and substructural type systems may express all of the idioms available in type-and-effect systems for region-based memory management. While the soundness of the type systems for each language is an important consideration, we believe that the central thesis of this dissertation is best addressed by focusing on the definitions of the languages and type systems and the translations. Hence, the main body of this dissertation is supplemented by a series of appendices (Appendices A, B, and C), which include technical details (including arguments for the soundness of the type systems) that would otherwise detract from the main focus.

In Chapters 2, 3, and 4, we present the three "flavors" of type systems for region-based memory management introduced above: a type-and-effect system, a

monadic type system, and a substructural type system. Each chapter begins with an overview, followed by a section of background material. This background material reviews the relevant history and nature of the "flavor" of type system under consideration. This motivates the definition of a language and type system, which is presented formally by giving syntax, dynamic semantics (a formal description of the execution of the language), and static semantics (a formal description of the type system). Chapters 3 and 4 also give formal type- and meaning-preserving translations: in Chapter 3, from a type-and-effect system to the monadic language $\mathsf{F}^{\mathsf{RGN}}$; in Chapter 4, from $\mathsf{F}^{\mathsf{RGN}}$ to the substructural language rgnURAL. These chapters also review the relevant related work, before concluding with a chapter summary.

Thus, the remainder of this dissertation is structured as follows.

In Chapter 2, we consider type-and-effect systems for region-based management. We review the history and nature of type-and-effect systems, and informally introduce the key aspects of the Tofte-Talpin region calculus. This motivates the definition of the Single Effect Calculus, a variation of the Tofte-Talpin region calculus with a novel type-and-effect system. The design of the Single Effect Calculus builds on the following insight: in languages with lexically-scoped regions, only the most-recently allocated region can be deallocated. This constraint can be leveraged to reduce the effect of a computation from a set of regions to a single region. We also demonstrate that the Single Effect Calculus is sufficient to encode a Traditional Region Calculus, which corresponds directly to type-and-effect systems given in the literature.

In Chapter 3, we introduce a monadic type system for region-based memory management. We review a closely related monadic type system that served as

inspiration: the ST monad of Launchbury and Peyton Jones, which is used to encapsulate "stateful" computations within a pure functional language. We show why the ST monad and its variants are insufficient for encoding a the Tofte-Talpin region calculus. This motivates the definition of the $\mathsf{F}^{\mathsf{RGN}}$ language, a monadic extension of the the familiar System F. The design of $\mathsf{F}^{\mathsf{RGN}}$ builds on two insights: 1) explicit terms may be used to witness the relationships between the lifetimes of lexically-scoped regions and to provide the evidence needed to safely "shift" computations between regions with nested lifetimes; 2) the familiar parametric polymorphism of System F may be used to ensure that dangling pointers are never dereferenced. We demonstrate that $\mathsf{F}^{\mathsf{RGN}}$ is sufficient for encoding type-and-effect systems for region-based memory management by giving a type- and meaning-preserving translation from the Single Effect Calculus to $\mathsf{F}^{\mathsf{RGN}}$.

In Chapter 4, we introduce a substructural type system for region-based memory management. We review the nature of substructural type systems, which may be used to limit the number of uses of data and operations in a program. For example, data may be annotated in ways that ensure that the data is used either *exactly once*, *at most once*, *at least once*, or an *arbitrary number of times*. This motivates the definition of the rgnURAL language, an extension of a substructural λ-calculus. We introduce primitives for separately creating and destroying regions; these primitives allow regions to have non-nested lifetimes, which gives rise to a more expressive language than those previously considered, which only support lexically-scoped regions. The design of rgnURAL builds on two insights: 1) separating the *name* of a region from the *property* that the region is live allows more flexible region lifetimes; 2) representing the property that a region is live by a capability that mediates access to the region and limiting the number of uses of a

capability allows a substructural type system to ensure region safety. We demonstrate that $\mathsf{F}^{\mathsf{RGN}}$ is sufficient for encoding monadic type systems for region-based memory management by giving a type- and meaning-preserving translation from $\mathsf{F}^{\mathsf{RGN}}$ to rgnURAL.

In Chapter 5, we consider the expressiveness of the various type-and-effect, monadic, and substructural languages presented in the previous chapters, consider extensions that provide support for additional programming features, and consider an advanced application of region-based memory management. This short investigation, along with the translations from the Single Effect Calculus to $\mathsf{F}^{\mathsf{RGN}}$ (Section 3.3) and from $\mathsf{F}^{\mathsf{RGN}}$ to rgnURAL (Section 4.3), helps to justify $\mathsf{F}^{\mathsf{RGN}}$ and rgnURAL as realistic formal languages that capture the essential aspects of region-based memory management. We present a high-level overview of the Cyclone language, introduce a hybrid monadic and substructural language that captures the key features of Cyclone, and sketch a translation from this hybrid language to rgnURAL. Finally, we consider an advanced application of region-based memory management: expressing a type-safe copying garbage collector.

Chapter 6 concludes by reviewing the technical developments in this dissertation and considering avenues for future work. As noted above, Appendices A, B, and C supplement Chapters 2, 3, and 4, respectively, with technical details that would otherwise detract from the main focus on the definitions of the languages and type systems and the translations from one language to the next.

Chapter 2

Type-and-Effect Systems for Region-Based Memory Management

In this chapter, we consider a variation of the Tofte-Talpin region calculus and three type-and-effect systems for this region calculus. The reason for presenting multiple type-and-effect systems arises from our goal of demonstrating that the monadic and substructural type systems presented in Chapters 3 and 4 are suitable for encoding region calculi; recall that our method for accomplishing this goal will be to give type- and meaning-preserving translations from a source language with a type-and-effect system to a target language with a monadic type system (Chapter 3) and to a target language with a substructural type system (Chapter 4). As should become clear, there is a large "semantic gap" between the type-and-effect system for a typical region calculus and the monadic and substructural type systems. Our conclusion is that the gap is too large to be bridged by a single translation. Instead, we give three type-and-effect systems, which successively close the gap.

The key insight that drives this progression of type-and-effect systems is that a LIFO stack of regions, such as that found in the Tofte-Talpin region calculus, imposes a partial order on live (that is, created and not yet destroyed) regions. Older regions (lower on the stack) outlive younger regions (higher on the stack). Hence, the liveness of a region implies the liveness of all regions below it on the stack.

The remainder of this chapter is structured as follows. In the following section, we examine more closely the history and nature of type-and-effect systems and informally introduce the key aspects of the Tofte-Talpin region calculus. This

motivates the definition of the Untyped Region Calculus, which is presented more formally in Sections 2.2.1 and 2.2.2. The Untyped Region Calculus provides a core syntax and dynamic semantics for a typical region calculus in the style of the Tofte-Talpin region calculus. Sections 2.2.3, 2.2.4, and 2.2.5 present three type-and-effect systems for the Untyped Region Calculus.

The first is a Traditional Region Calculus (Section 2.2.3), which corresponds directly to type-and-effect systems for region-based memory management given in the literature [39, 10, 11]. The second is the Bounded Region Calculus (Section 2.2.4), which augments the Traditional Region Calculus with a form of bounded region polymorphism. The Bounded Region Calculus can be seen as a core model of early designs for the Cyclone language [30, 29]. The third is the Single Effect Calculus (Section 2.2.5), which restricts the Bounded Region Calculus by admitting only a single region as the latent effect of an expression.

Type- and meaning-preserving translations from the Traditional Region Calculus to the Bounded Region Calculus and from the Bounded Region Calculus to the Single Effect Calculus are relatively straightforward (meaning-preservation following directly from the shared dynamic semantics) and will presented as succinctly as possible. Because the Single Effect Calculus will be the source language for our translation to the monadic type system of Chapter 3, we present the type-and-effect system for the Single Effect Calculus in somewhat more detail than the other two type-and-effect systems.

Appendix A compliments this chapter by including technical details that would otherwise detract from the focus on the definition of the Single Effect Calculus.

2.1 Background: Type-and-Effect Systems

Computational effects abound in realistic programs; they correspond to communication though IO, manipulation of mutable state, and execution of irregular control flow. Identifying the various computational effects in a program can yield insight into the ways in which the various components of a program interact. For example, by identifying that two expressions in a program manipulate disjoint portions of the program's mutable state, we may conclude that the two expressions could be evaluated in parallel, without changing the observable behavior of the program.

A *type-and-effect* system provides the core mechanisms necessary to describe the computational effects of a program. A conventional type system, such as that employed by the simply-typed λ-calculus, with a typing judgment like $\Gamma \vdash e : \tau$, describes only a property of the final value (if any) that is produced by the evaluation of e; for example, $\cdot \vdash e : \mathsf{Int}$ asserts that the evaluation of e produces a final value that is an integer. Note that it does not describe anything about the computational effects that might occur during the evaluation of e.

In contrast, a type-and-effect system is designed so that the typing judgment both describes the *type* of the final value and describes the important computational *effects* that occur during the evaluation of e. Type-and-effect systems use a unified judgment to simultaneously derive both the type and the effect of an expression. The basic type-and-effect judgment is $\Gamma \vdash e : \tau, \phi$, where ϕ is an *effect expression* and τ, ϕ together form the *type and effect*. Informally, the judgment is read "in the environment Γ, the evaluation of the expression e may have the observable effect ϕ and eventually yields a value of type τ, if any." Variation amongst type-and-effect systems largely arises from the choice of effect expressions and the choice of auxiliary judgments that prove when one effect expression is equivalent

or subsumed by another effect expression. Every type-and-effect system includes a number of atomic effects and a number of operations for combining effects.

Another defining characteristic of type-and-effect systems is the form of the function type: $\tau_1 \xrightarrow{\phi} \tau_2$. Note that the function type describes not only the types of the argument and result, but also the *latent* (or *delayed*) *effect* of the function. This latent effect describes the computational effect that occurs when the function is applied to an argument.

The *FX* language [25, 26] was the first programming language to incorporate a type-and-effect system. In the *FX* language, the effects are used to discover scheduling constraints for a parallel implementation of *FX* programs [35]: for example, two expressions that write to a mutable reference cannot be executed in parallel, whereas two expressions that read from a mutable reference can be executed in parallel. Hence, the type-and-effect system of *FX* tracks the allocating, reading, and writing of shared, mutable references. This gives rise to the following structure for effects:

Atomic effects

$a \quad ::= \quad \mathsf{new} \mid \mathsf{read} \mid \mathsf{write}$

Effects

$\phi \quad ::= \quad \{a_1, \ldots, a_n\}$

The main operation for combining effects is the union of two sets of atomic effects.

In order to more formally introduce type-and-effect systems, we consider a very simple λ-calculus with shared, mutable references, dubbed λ^{FX} (since it is a simplification of the *FX* language), whose syntax is given in Figure 2.1. The type $\mathsf{Ref}\ \tau$ denotes a shared, mutable reference, containing a value of type τ; there is an expression form to allocate (`new`) references, as well as expression forms to read (`read`) and write (`write`) their contents.

Atomic effects

$$a \quad ::= \quad \mathsf{new} \mid \mathsf{read} \mid \mathsf{write}$$

Effects

$$\phi \quad ::= \quad \{a_1, \ldots, a_n\}$$

Types

$$\tau \quad ::= \quad \mathsf{Bool} \mid \tau_1 \xrightarrow{\phi} \tau_2 \mid \tau_1 \times \cdots \times \tau_n \mid \mathsf{Ref}\ \tau$$

Boolean constants

$$\mathfrak{b} \quad \in \quad \{\mathtt{true}, \mathtt{false}\}$$

Value variables

$$f, x \quad \in \quad \mathit{VVars}$$

Terms

$$\begin{aligned} e \quad ::= \quad & \mathfrak{b} \mid \mathtt{if}\ e_b\ \mathtt{then}\ e_t\ \mathtt{else}\ e_f \mid \\ & x \mid \lambda x.\, e \mid e_1\ e_2 \mid \\ & \langle e_1, \ldots, e_n \rangle \mid \mathtt{sel}_i\ e \mid \\ & \mathtt{let}\ x = e_a\ \mathtt{in}\ e_b \mid \\ & \mathtt{new}\ e \mid \mathtt{read}\ e \mid \mathtt{write}\ e_1\ e_2 \end{aligned}$$

Figure 2.1: Syntax of λ^{FX}

$\boxed{\Gamma \vdash_{\text{exp}} e : \tau}$

$$\frac{}{\Gamma \vdash_{\text{exp}} \mathfrak{b} : \mathsf{Bool}, \{\}} \qquad \frac{\Gamma \vdash_{\text{exp}} e_b : \mathsf{Bool}, \phi_b \qquad \Gamma \vdash_{\text{exp}} e_t : \tau, \phi_t \qquad \Gamma \vdash_{\text{exp}} e_f : \tau, \phi_f}{\Gamma \vdash_{\text{exp}} \texttt{if}\ e_b\ \texttt{then}\ e_t\ \texttt{else}\ e_f : \tau, e_b \cup \phi_t \cup \phi_f}$$

$$\frac{x \in dom(\Gamma) \qquad \Gamma(x) = \tau}{\Gamma \vdash_{\text{exp}} x : \tau, \{\}} \qquad \frac{\Gamma, x{:}\tau_x \vdash_{\text{exp}} e : \tau, \phi'}{\Gamma \vdash_{\text{exp}} \lambda x.\, e : \tau_x \xrightarrow{\phi'} \tau, \{\}}$$

$$\frac{\Gamma_f \vdash_{\text{exp}} e_f : \tau_x \xrightarrow{\phi'} \tau, \phi_f \qquad \Gamma_a \vdash_{\text{exp}} e_a : \tau_x, \phi_a}{\Gamma \vdash_{\text{exp}} e_f\ e_a : \tau, \phi_f \cup \phi_a \cup \phi'}$$

$$\frac{\Gamma \vdash_{\text{exp}} e_i : \tau_i, \phi_i \quad {}^{i \in 1 \ldots n}}{\Gamma \vdash_{\text{exp}} \langle e_1, \ldots, e_n \rangle : \tau_1 \times \cdots \times \tau_n, \phi_1 \cup \cdots \cup \phi_n}$$

$$\frac{\Gamma \vdash_{\text{exp}} e : \tau_1 \times \cdots \times \tau_n, \phi \qquad 0 \leq i \leq n}{\Delta; \Gamma \vdash_{\text{exp}} \texttt{sel}_i\ e : \tau_i, \phi}$$

$$\frac{\Gamma \vdash_{\text{exp}} e_a : \tau_x, \phi_a \qquad \Gamma, x{:}\tau_x \vdash e_b : \tau, \phi_b}{\Gamma \vdash_{\text{exp}} \texttt{let}\ x = e_a\ \texttt{in}\ e_b : \tau, \phi_a \cup \phi_b} \qquad \frac{\Gamma \vdash_{\text{exp}} e : \tau, \phi}{\Gamma \vdash_{\text{exp}} \texttt{new}\ e : \mathsf{Ref}\ \tau, \phi \cup \{\mathsf{new}\}}$$

$$\frac{\Gamma \vdash_{\text{exp}} e : \mathsf{Ref}\ \tau, \phi}{\Gamma \vdash_{\text{exp}} \texttt{read}\ e : \tau, \phi \cup \{\mathsf{read}\}} \qquad \frac{\Gamma \vdash_{\text{exp}} e : \mathsf{Ref}\ \tau, \phi \qquad \Gamma \vdash_{\text{exp}} e_\star : \tau, \phi_\star}{\Gamma \vdash_{\text{exp}} \texttt{write}\ e\ e_\star : \mathsf{1}, \phi \cup \phi_\star \cup \{\mathsf{write}\}}$$

Figure 2.2: Static semantics of λ^{FX} (expressions)

Figure 2.2 presents the type-and-effect for λ^{FX}; as expected, type-and-effect judgments have the form $\Gamma \vdash_{\text{exp}} e : \tau, \phi$. The rule for a boolean constant has the empty effect ($\{\}$), as its evaluation has no computational effect. The rule for `if` e_b `then` e_t `else` e_f combines the effects for e_b, e_t, and e_f. We may see that the judgment derives a conservative approximation of the effect of an expression, since, at run time, either e_t will be evaluated (and the effects denoted by ϕ_t will occur) or e_f will be evaluated (and the effects denoted by ϕ_f will occur), but not both.

In the rule for function abstraction, we may see that the effect of the function body becomes the latent effect, while the function abstraction itself has the empty effect. Since the body of the function is evaluated when the function is applied, the rule for function application adds the latent effect to the effect of the entire expression. In general, the rules combine the effects evaluated sub-expressions into the effect of the entire expression (e.g., the rule for tuple introduction).

Finally, the rules for `new`, `read`, and `write` each introduce their respective atomic effect.

Note that the type-and-effect system for λ^{FX} is very conservative and coarse. For example, two expressions with the effects $\{\mathsf{read}\}$ and $\{\}$ *must not* manipulate the same mutable state, while two expressions with the effects $\{\mathsf{read}\}$ and $\{\mathsf{new}, \mathsf{read}, \mathsf{write}\}$ *may* manipulate the same mutable state. Furthermore, the effects derived by the type-and-effect system never decrease; they always accumulate more effects, saturating with the effect $\{\mathsf{new}, \mathsf{read}, \mathsf{write}\}$.

Consider the following λ^{FX} expression:

```
let r = new true in
let u = write r false in
read r
```

This expression has the type Bool and the effect $\{\mathsf{new}, \mathsf{read}, \mathsf{write}\}$. Yet, it is clear that this expression must not manipulate the same mutable state as any other expression, since the only mutable state it manipulates is freshly allocated in the expression itself (and, hence, must be disjoint from any other allocated mutable state). Since the mutable state manipulated by the expression is private to the expression, we might like to assign it the empty effect $\{\}$. Intuitively, this corresponds to the fact that the side-effects of this expression cannot be observed outside of the expression, and, hence, need not be reported in the effect of the expression.

To handle these ideas, the base *FX* language was extended with regions [58], which describe the portion of the mutable state in which side-effects may occur.[1] We may incorporate this extension into λ^{FX} in the following manner:

Region names

$\mathfrak{r} \quad \in \quad RNames$

Atomic effects

$a \quad ::= \quad \mathsf{new}\ \mathfrak{r} \mid \mathsf{read}\ \mathfrak{r} \mid \mathsf{write}\ \mathfrak{r}$

Effects

$\phi \quad ::= \quad \{a_1, \ldots, a_n\}$

Types

$\tau \quad ::= \quad \cdots \mid \mathsf{Ref}\ \mathfrak{r}\ \tau$

Terms

$e \quad ::= \quad \cdots \mid \mathtt{new}\ \mathfrak{r}\ e$

Note that the atomic effects, the reference type, and the `new` expression are parameterized by a region. We may reconsider the λ^{FX} expression given above,

[1] In the *FX* language, regions were not used for memory management. Rather, they were used to improve the precision of the effect system.

incorporating a region name:

$$\begin{array}{l}\texttt{let } r = \texttt{new } \mathfrak{r}_1 \texttt{ true in} \\ \texttt{let } u = \texttt{write } r \texttt{ false in} \\ \texttt{read } r\end{array}$$

Now, this expression has the effect $\{\mathsf{new}\ \mathfrak{r}_1, \mathsf{read}\ \mathfrak{r}_1, \mathsf{write}\ \mathfrak{r}_1\}$. Furthermore, this expression must not manipulate the same mutable state as any other expression that has an effect which does not mention $\mathfrak{r}_1$.

This refinement with regions allows, under certain circumstances, side-effects that cannot be observed outside a given expression to be *masked* by the effect system. However, this effect masking comes with a restriction to lexical scopes. In particular, effect masking in FX is accomplished through the `private` expression form, which declares a private region for local use. Side-effects on this region cannot be observed outside of the expression and need not be reported in the effect of the expression. We may incorporate this extension into λ^{FX} in the following manner:

$$\begin{array}{rcl}\multicolumn{3}{l}{\text{Terms}} \\ e & ::= & \cdots \mid \texttt{private } \mathfrak{r} \texttt{ in } e\end{array}$$

The type-and-effect rule for the $\texttt{private } \mathfrak{r} \texttt{ in } e$ expression is as follows:

$$\frac{\Gamma \vdash_{\text{exp}} e : \tau, \phi \qquad \mathfrak{r} \notin RN(\Gamma) \qquad \mathfrak{r} \notin RN(\tau)}{\Gamma \vdash_{\text{exp}} \texttt{private } \mathfrak{r} \texttt{ in } e : \tau, \phi \setminus \{\mathsf{new}\ \mathfrak{r}, \mathsf{read}\ \mathfrak{r}, \mathsf{write}\ \mathfrak{r}\}}$$

where $RN(\Gamma)$ and $RN(\tau)$ denote the set of region names in the context Γ and in the type τ, respectively. Note that the type and effect of the entire expression is the same as the type and effect of e, except that the effects on the private region $\mathfrak{r}$ are masked (i.e., removed from the effect). The rule also ensures that the private region does not appear in either the (types of the) free variables of or the type of

e; this ensures that the region is private to the evaluation of e. We may reconsider the λ^{FX} expression given above, incorporating a private region name:

$$
\begin{array}{l}
\texttt{private}\ \mathfrak{r}_1\ \texttt{in} \\
\quad \texttt{let}\ r = \texttt{new}\ \mathfrak{r}_1\ \texttt{true}\ \texttt{in} \\
\quad \texttt{let}\ u = \texttt{write}\ r\ \texttt{false}\ \texttt{in} \\
\quad \texttt{read}\ r
\end{array}
$$

Now, this expression has the type Bool and the effect $\{\}$.

Tofte and Talpin recognized that this combination of regions denoting portions of the program's state, private regions, and effect masking in a type-and-effect system could be used to account for the allocation and deallocation of values in a program [79, 80]. One particular insight is that, if an expression has the type Bool (as in the example λ^{FX} expression above), then all memory allocated during the computation of the boolean could be deallocated at the end of the computation.

In order to realize this memory behavior, they introduced the concept of region-based memory management. In their calculus, regions are areas of memory holding heap allocated data. Expression forms that correspond to heap allocated values (e.g., constant expressions, λ-abstractions, and tuple introductions) are annotated with a region:

$$
\begin{array}{l}
\mathfrak{b}\ \texttt{at}\ \rho \\
\lambda x.\, e\ \texttt{at}\ \rho \\
\langle e_1, \ldots, e_n \rangle\ \texttt{at}\ \rho
\end{array}
$$

The annotation $\texttt{at}\,\rho$ indicates that the value should be allocated in the region bound to the region variable ρ.[2]

[2]The $\texttt{at}\,\rho$ annotation is analogous to the $\texttt{new}\ \mathfrak{r}\ e$ expression form in λ^{FX}.

Regions are introduced and eliminated with a lexically-scoped construct:

$$\texttt{letregion}\ \rho\ \texttt{in}\ e$$

and thus have last-in-first-out (LIFO) lifetimes following the block structure of the program. The collection of regions in a program may be organized as a stack. In the construct above, a region corresponding to ρ is created when the expression starts evaluating; while the expression evaluates (in particular, during the evaluation of the sub-expression e), data can be allocated in and read from the region; when e has been evaluated to a value, the region is destroyed and the value is returned.[3]

Tofte and Talpin designed a type-and-effects system that ensures the safety of this allocation and deallocation scheme. The types of allocated data values are augmented with the region in which they are allocated. For example the type:

$$((\mathsf{Int}, \rho_1) \times (\mathsf{Int}, \rho_2), \rho_1)$$

describes pairs of integers where the pair and integer in the first component are allocated in region ρ_1 and the integer in the second component is allocated in region ρ_2.[4]

In the Tofte-Talpin region calculus, the atomic effects are regions (ρ) and effects are finite sets of regions (ϕ). Hence, in the type-and-effect system, the effect denotes the set of regions that may be accessed during the evaluation of the expression; alternatively, it denotes the set of regions that must still be allocated (live) in order to safely evaluate the expression. In general, any expression that needs to read a value allocated in a region will require that region to be in the effect of the expression; alternatively, it will require that the region be live.

[3]The `letregion` expression form is analogous to the `private` expression form in λ^{FX}.

[4]The (τ, ρ) type is analogous to the $\mathsf{Ref}\ \mathtt{r}\ \tau$ type in λ^{FX}.

Region polymorphism makes it possible to abstract over the regions a computation manipulates. For example, a function `fst` that takes in a pair of integers and returns the first component without examining it could have a type of the form:

$$\texttt{fst} :: \forall \rho_1, \rho_2, \rho_3.((\mathsf{Int}, \rho_1) \times (\mathsf{Int}, \rho_2), \rho_3) \xrightarrow{\{\rho_3\}} (\mathsf{Int}, \rho_1)$$

Such a function is polymorphic over regions ρ_1, ρ_2, and ρ_3; the caller can effectively re-use the function regardless of where the data are allocated. However, the latent effect $\{\rho_3\}$ on the function type indicates that whatever region instantiates ρ_3 needs to still be allocated when `fst` is called. In principle, neither of the other regions needs to be live across the call since the function does not examine the integer values. In practice, ρ_1 will be live assuming the caller wishes to use the result.

As we noted earlier, region-based memory management allows evaluation to lead to values with *dangling pointers*: pointers to data in some region that has been deallocated. For some programs, this allows a region-based memory manager to reclaim strictly more objects than a trace-based garbage collector. Consider, for example, the following program:

$$\begin{array}{l}
\texttt{letregion}\ \rho_a\ \texttt{in} \\
\quad \texttt{let}\ g = \texttt{letregion}\ \rho_b\ \texttt{in} \\
\qquad\qquad \texttt{let}\ p = (3\,\texttt{at}\,\rho_a, 4\,\texttt{at}\,\rho_b)\,\texttt{at}\,\rho_a \\
\qquad\qquad \texttt{in}\ \lambda z{:}\mathsf{1}.\,\texttt{fst}\ [\rho_a, \rho_b, \rho_a]\ p \\
\quad \texttt{in}\ g\ \langle\rangle
\end{array}$$

The pair p and its first component are allocated in the outer (older) region ρ_a whereas p's second component is allocated in an inner (younger) region ρ_b. The closure bound to g is a thunk that calls `fst` on p. Note that the region ρ_b is deallocated before the thunk is run, and thus g's closure contains a dangling pointer

to an object that is never dereferenced. The Tofte-Talpin type-and-effect system is strong enough to show that the code is safe.

2.2 Region and Effect Calculi

The Untyped Region Calculus is a variation of the Tofte-Talpin region calculus, given as a language with syntax and dynamic semantics, but no type-and-effect system. The Traditional Region Calculus, the Bounded Region Calculus, and the Single Effect Calculus are a succession of type-and-effect systems for the Untyped Region Calculus that ensure the region safety of the language; that is, they ensure that there is no access to a region (for allocating in or reading from) before it is created or after it is destroyed.

In this section, we present the Untyped Region Calculus and the three type-and-effect systems in sufficient detail to establish the Single Effect Calculus as a reasonable source language for the translation into a target language with a monadic type system (Chapter 3) and to a language with a substructural type system (Chapter 4). To this end, we include the syntax for both the surface language of and abstract machine configurations for the Untyped Region Calculus, dynamic semantics for the abstract machine configurations for the Untyped Region Calculus, and static semantics for the surface languages of the Traditional Region Calculus, the Bounded Region Calculus,a nd the Single Effect Calculus.

The dynamic semantics defines a large-step (or natural) semantics, which defines an *evaluation relation* from *stacks of regions* and *expressions* to *values*. Our main reason for adopting a large-step operational semantics is to simplify the theorems and proofs of Section 3.3 and Appendix B.3; establishing the correctness of the translation from the Single Effect Calculus to the monadic language would

Region variables

$\varrho, \varpi \quad \in \quad RVars \qquad \text{where } \mathcal{H} \in RVars$

Surface regions

$\rho, \pi \quad ::= \quad \varrho$

Effects

$\phi \quad ::= \quad \{\rho_1, \ldots, \rho_n\}$

Figure 2.3: Surface syntax of URC (I)

be more difficult using small-step operational semantics, due to differing numbers of intermediate small-steps. Nonetheless, there is a straightforward mechanism for distinguishing divergent computations from stuck configurations; see Appendix B.1.1.

We purposefully omit the static semantics for the abstract machine configurations (normally included for a syntactic proof of type soundness), since it requires a number of technical details that detract from the focus on the translation. Appendix A.1 includes additional technical details for the Single Effect Calculus language and (sketches) a syntactic proof of type soundness.

2.2.1 Syntax of the Untyped Region Calculus

Our first region calculus is the Untyped Region Calculus (URC), which is a variation of the region calculus of Tofte and Talpin [79, 80], in the spirit of more recent direct presentations of region calculi [39, 10, 11, 41]. This calculus will provide core syntax and dynamic semantics for the subsequent type systems.

Integer constants

$$\mathfrak{i} \quad \in \quad \mathbb{Z}$$

Boolean constants

$$\mathfrak{b} \quad \in \quad \{\texttt{true}, \texttt{false}\}$$

Value variables

$$f, x \quad \in \quad \mathit{VVars}$$

Surface terms

$$\begin{aligned} e \quad ::= \quad & \mathfrak{i}\,\texttt{at}\,\rho \mid e_1 \oplus e_2\,\texttt{at}\,\rho \mid e_1 \oslash e_2 \mid \mathfrak{b} \mid \texttt{if}\ e_b\ \texttt{then}\ e_t\ \texttt{else}\ e_f \mid \\ & x \mid \lambda x.\, e\,\texttt{at}\,\rho \mid e_1\ e_2 \mid (e_1, \ldots, e_n)\,\texttt{at}\,\rho \mid \texttt{sel}_i\ e \mid \\ & \texttt{letregion}\ \varrho\ \texttt{in}\ e \mid \Lambda\varrho.\, u\,\texttt{at}\,\rho \mid e\ [\rho] \mid \end{aligned}$$

Abstractions

$$u \quad ::= \quad \lambda x.\, e\,\texttt{at}\,\rho \mid \Lambda\varrho.\, u\,\texttt{at}\,\rho$$

Figure 2.4: Surface syntax of URC (II)

Surface Syntax of URC

Figures 2.3 and 2.4 presents the syntax of "surface programs" (that is, excluding syntax and semantic objects that will appear in the dynamic semantics) of URC. In the following sections, we explain and motivate the main constructs of URC.

Terms Terms are similar to those found in the λ-calculus. One major difference is that introduction forms corresponding to heap allocated values carry a region annotation $\texttt{at}\,\rho$, which indicates in which region the value is to be allocated. We assume that integers, tuples, and function closures, and region abstractions require heap allocated storage, while booleans do not. New regions are introduced (and implicitly created and destroyed) by the $\texttt{letregion}\ \varrho\ \texttt{in}\ e$ term. The region variable ϱ is bound within e, demarcating the scope of the region. Within e, values may be read from or allocated in the region ϱ. Executing $\texttt{letregion}\ \varrho\ \texttt{in}\ e$ allocates a new region of memory, then executes e, and finally deallocates the region.

The term $\lambda\varrho.\, u\,\texttt{at}\,\rho$ introduces a region abstraction (allocated in the region ρ), where the term u is polymorphic in the region ϱ.[5] Such region polymorphism is particularly useful in the definition of functions, in which we parameterize over the regions necessary for the evaluation of the function. The term $e\ [\rho]$ eliminates a region abstraction; operationally, it substitutes the region ρ for the region variable ϱ in a region abstraction body and evaluates the resulting term.

Regions and effects We will discuss the meaning of regions and effects in more detail in the subsequent sections that introduce type-and-effect systems for URC. At this point, we simply note that we introduce syntactic classes for regions and

[5]Limiting the body of a region abstraction to abstractions ensures that an erasure function that removes region annotations and produces a λ-calculus term is meaning preserving.

Region names

$\mathfrak{r} \quad \in \quad RNames \qquad \text{where } \mathfrak{H} \in RNames$

Constant regions

$r \quad ::= \quad \mathfrak{r} \mid \bullet$

Pointer names

$\mathfrak{p} \quad \in \quad PNames$

Abstract machine regions

$\rho, \pi \quad ::= \quad \ldots \mid r$

Abstract machine terms

$e \quad ::= \quad \ldots \mid \texttt{ref}\ r\ \mathfrak{p}$

Values

$v \quad ::= \quad \mathfrak{b} \mid \texttt{ref}\ r\ \mathfrak{p}$

Figure 2.5: Abstract machine syntax of URC (I)

effects. Effects are simply finite sets of regions. In the surface syntax, it suffices to allow regions to range over region variables (*RVars*), which include a distinguished member $\mathcal{H}$, corresponding to a global region that remains allocated throughout the execution of the program.

Abstract Machine Configurations for URC

Figures 2.5 and 2.6 presents the syntax of abstract machine configurations for URC, which extends the syntax of the previous section with semantic objects that appear in the operational semantics.

Storable values

$$w \quad ::= \quad \mathfrak{i} \mid \lambda x.\, e \mid (v_1, \ldots, v_n) \mid \Lambda \varrho.\, u$$

Regions

$$R \quad ::= \quad \{\mathfrak{p}_1 \mapsto w_1, \ldots, \mathfrak{p}_n \mapsto w_n\}$$

Stacks

$$S \quad ::= \quad \cdot \mid S, \mathfrak{r} \mapsto R \quad \text{(ordered domain)}$$

Abstract machine configurations

$$(S; e)$$

Figure 2.6: Abstract machine syntax of URC (II)

Region names and pointers are used to represent references to region allocated values. Region constants distinguish between live and dead regions; a dead region ($\bullet$) corresponds to a deallocated region. There is a distinguished region name $\mathfrak{H}$, corresponding to a global region that remains allocated throughout the execution of the program.

The abstract machine syntax adds one new region form and one new expression form. The region r is the instantiated form of a region variables (hence, $\bullet$ corresponds to a dead region). The expression `ref` $\mathfrak{r}$ $\mathfrak{p}$ is the (live) pointer associated with a region allocated value. Likewise, the expression `ref` $\bullet$ $\mathfrak{p}$ is the is the (dangling) pointer associated with a region deallocated value.

Thus far, we have talked about region allocated data without discussing where such data is stored. Because the introduction forms for region allocated values are not themselves values, we formalize the syntactic class of storable values. Storable

values are associated with pointers in regions R and regions are ordered into stacks S. Intuitively, evaluating a `letregion` expression adds a new region to the top of the stack (the new region is deallocated upon completing the `letregion` body). These intuitions are formalized in the dynamic semantics of the next section.

2.2.2 Dynamic Semantics of the Untyped Region Calculus

An inductive judgment (Figures 2.7 and 2.8) defines the dynamic semantics of URC. We state without proof that the dynamic semantics is deterministic; it is syntax directed, taking $(S; e)$ configurations modulo α-conversion, including conversion of region names and pointers, which are (uniquely) bound in the stack S.

We use the notation $S(\mathfrak{r})$ for the lookup of regions in stacks and the notation $S(\mathfrak{r}, \mathfrak{p})$ for the iterated lookup of storable values in stacks. These are partial functions, defined as follows:

$$\begin{array}{rcll} S(\mathfrak{r}) & = & \mathsf{undefined} & \text{if } \mathfrak{r} \notin dom(S) \\ S(\mathfrak{r}) & = & R & \text{if } \mathfrak{r} \in dom(S) \text{ and } S \equiv \ldots, \mathfrak{r} \mapsto R, \ldots \end{array}$$

$$\begin{array}{rcll} S(\mathfrak{r}, \mathfrak{p}) & = & \mathsf{undefined} & \text{if } S(\mathfrak{r}) = \mathsf{undefined} \\ S(\mathfrak{r}, \mathfrak{p}) & = & \mathsf{undefined} & \text{if } S(\mathfrak{r}) = R \text{ and } \mathfrak{p} \notin dom(R) \\ S(\mathfrak{r}, \mathfrak{p}) & = & R & \text{if } S(\mathfrak{r}) = R \text{ and } \mathfrak{p} \in dom(R) \text{ and } R \equiv \{\ldots, \mathfrak{p} \mapsto w, \ldots\} \end{array}$$

We also use the notation $S\{(\mathfrak{r}, \mathfrak{p}) \mapsto w\}$ to denote the stack S' which extends the stack S with a mapping from $\mathfrak{p}$ to w in the region $S(\mathfrak{r})$. This function is defined when $\mathfrak{r} \in dom(S)$ and $S(\mathfrak{r}) = R$ and $\mathfrak{p} \notin dom(R)$.

The judgment $(S; e) \Downarrow (S'; v')$ asserts that evaluating the closed expression e in stack S results in a new stack S' and a value v'. Note that the rules for $(S; e) \Downarrow (S'; v')$ thread the modified stack through each expression evaluation, im-

$\boxed{(S; e) \Downarrow (S'; v')}$

$$\frac{\mathfrak{r} \in dom(S) \qquad \mathfrak{p} \notin dom(S(\mathfrak{r}))}{(S; \mathfrak{i}\,\texttt{at}\,\mathfrak{r}) \Downarrow (S\{(\mathfrak{r}, \mathfrak{p}) \mapsto \mathfrak{i}\}; \texttt{ref}\ \mathfrak{r}\ \mathfrak{p})}$$

$$\frac{\begin{array}{c}(S; e_1) \Downarrow (S_1; \texttt{ref}\ \mathfrak{r}_1\ \mathfrak{p}_1) \qquad S_1(\mathfrak{r}_1, \mathfrak{p}_2) = \mathfrak{i}_1 \\ (S_1; e_2) \Downarrow (S_2; \texttt{ref}\ \mathfrak{r}_2\ \mathfrak{p}_2) \qquad S_2(\mathfrak{r}_2, \mathfrak{p}_2) = \mathfrak{i}_2 \\ \mathfrak{r} \in dom(S) \qquad \mathfrak{p} \notin dom(S(\mathfrak{r})) \qquad \mathfrak{i}_1 \oplus \mathfrak{i}_2 = \mathfrak{i}\end{array}}{(S; e_1 \oplus e_2\,\texttt{at}\,\mathfrak{r}) \Downarrow (S_2\{(\mathfrak{r}, \mathfrak{p}) \mapsto \mathfrak{i}\}; \texttt{ref}\ \mathfrak{r}\ \mathfrak{p})}$$

$$\frac{\begin{array}{c}(S; e_1) \Downarrow (S_1; \texttt{ref}\ \mathfrak{r}_1\ \mathfrak{p}_1) \qquad S_1(\mathfrak{r}_1, \mathfrak{p}_2) = \mathfrak{i}_1 \\ (S_1; e_2) \Downarrow (S_2; \texttt{ref}\ \mathfrak{r}_2\ \mathfrak{p}_2) \qquad S_2(\mathfrak{r}_2, \mathfrak{p}_2) = \mathfrak{i}_2 \\ \mathfrak{i}_1 \otimes \mathfrak{i}_2 = \mathfrak{b}\end{array}}{(S; e_1 \otimes e_2) \Downarrow (S_2; \mathfrak{b})} \qquad \frac{}{(S; \mathfrak{b}) \Downarrow (S; \mathfrak{b})}$$

$$\frac{(S; e_b) \Downarrow (S'; \texttt{true}) \qquad (S'; e_t) \Downarrow (S''; v'')}{(S; \texttt{if}\ e_b\ \texttt{then}\ e_t\ \texttt{else}\ e_f) \Downarrow (S''; v'')}$$

$$\frac{(S; e_b) \Downarrow (S'; \texttt{false}) \qquad (S'; e_f) \Downarrow (S''; v'')}{(S; \texttt{if}\ e_b\ \texttt{then}\ e_t\ \texttt{else}\ e_f) \Downarrow (S''; v'')}$$

Figure 2.7: Dynamic semantics of URC (I)

$\boxed{(S; e) \Downarrow (S'; v')}$

$$\frac{\mathfrak{r} \in dom(S) \qquad \mathfrak{p} \notin dom(S(\mathfrak{r}))}{(S; \lambda x.\, e \,\texttt{at}\, \mathfrak{r}) \Downarrow (S\{(\mathfrak{r}, \mathfrak{p}) \mapsto \lambda x.\, e\}; \texttt{ref}\ \mathfrak{r}\ \mathfrak{p})}$$

$$\frac{(S; e_f) \Downarrow (S_f; \texttt{ref}\ \mathfrak{r}_f\ \mathfrak{p}_f) \qquad S_f(\mathfrak{r}_f, \mathfrak{p}_f) = \lambda x.\, e_b \qquad (S_f; e_a) \Downarrow (S_a; v_a) \qquad (S_a; e_b[v_a/x]) \Downarrow (S'; v')}{(S; e_f\ e_a) \Downarrow (S'; v')}$$

$$\frac{(S; e_1) \Downarrow (S_1; v_1) \qquad \ldots \qquad (S_{n-1}; e_n) \Downarrow (S_n; v_n) \qquad \mathfrak{r} \in dom(S_n) \qquad \mathfrak{p} \notin dom(S_n(\mathfrak{r}))}{(S; (e_1, \ldots, e_n) \,\texttt{at}\, \mathfrak{r}) \Downarrow (S_2\{(\mathfrak{r}, \mathfrak{p}) \mapsto (v_1, \ldots, v_n)\}; \texttt{ref}\ \mathfrak{r}\ \mathfrak{p})}$$

$$\frac{(S; e) \Downarrow (S'; \texttt{ref}\ \mathfrak{r}\ \mathfrak{p}) \qquad S'(\mathfrak{r}, \mathfrak{p}) = (v_1, \ldots, v_2) \qquad 1 \leq i \leq n}{(S; \texttt{sel}_i\ e) \Downarrow (S'; v_i)}$$

$$\frac{\mathfrak{r} \in dom(S) \qquad \mathfrak{p} \notin dom(S(\mathfrak{r}))}{(S; \Lambda\varrho.\, u \,\texttt{at}\, \mathfrak{r}) \Downarrow (S\{(\mathfrak{r}, \mathfrak{p}) \mapsto \Lambda\varrho.\, u\}; \texttt{ref}\ \mathfrak{r}\ \mathfrak{p})}$$

$$\frac{(S; e_f) \Downarrow (S_f; \texttt{ref}\ \mathfrak{r}_f\ \mathfrak{p}_f) \qquad S_f(\mathfrak{r}_f, \mathfrak{p}_f) = \Lambda\varrho.\, u_b \qquad (S_f; u[\rho_a/\varrho]) \Downarrow (S'; v')}{(S; e_f\ [\rho_a]) \Downarrow (S'; v')}$$

$$\frac{\mathfrak{r} \notin dom(S) \qquad (S, \mathfrak{r} \mapsto \{\}; e[\mathfrak{r}/\varrho]) \Downarrow (S', \mathfrak{r} \mapsto R'; v')}{(S; \texttt{letregion}\ \varrho\ \texttt{in}\ e) \Downarrow (S'[\bullet/\mathfrak{r}]; v'[\bullet/\mathfrak{r}])}$$

Figure 2.8: Dynamic semantics of URC (II)

posing a left-to-right evaluation order. Consider, for example, the following rule:

$$\frac{\begin{array}{c}(S; e_1) \Downarrow (S_1; \texttt{ref}\ \mathfrak{r}_1\ \mathfrak{p}_1) \qquad S_1(\mathfrak{r}_1, \mathfrak{p}_2) = \mathfrak{i}_1 \\ (S_1; e_2) \Downarrow (S_2; \texttt{ref}\ \mathfrak{r}_2\ \mathfrak{p}_2) \qquad S_2(\mathfrak{r}_2, \mathfrak{p}_2) = \mathfrak{i}_2 \\ \mathfrak{r} \in dom(S) \qquad \mathfrak{p} \notin dom(S(\mathfrak{r})) \qquad \mathfrak{i}_1 \oplus \mathfrak{i}_2 = \mathfrak{i}\end{array}}{(S; e_1 \oplus e_2\ \texttt{at}\ \mathfrak{r}) \Downarrow (S_2\{(\mathfrak{r}, \mathfrak{p}) \mapsto \mathfrak{i}\}; \texttt{ref}\ \mathfrak{r}\ \mathfrak{p})}$$

The first line evaluates e_1 to a live reference ($\texttt{ref}\ \mathfrak{r}_1\ \mathfrak{p}_1$) and reads out the integer stored at $\mathfrak{p}_1$ in the region $\mathfrak{r}_1$. Likewise, the second line evaluates e_2 to a live reference ($\texttt{ref}\ \mathfrak{r}_2\ \mathfrak{p}_2$) and reads out the integer stored at $\mathfrak{p}_2$ in the region $\mathfrak{r}_2$. Finally, a fresh pointer in the region $\mathfrak{r}$ is chosen, and the final stack with the computed integer stored at the freshly chosen location and the location are returned. The other rules work in much the same manner.

The rule for `letregion` introduces (and subsequently eliminates) a new region. The rule executes in the following manner. First, a fresh region name $\mathfrak{r}$ is chosen. Next, the region $\mathfrak{r}$ is substituted for the region variable ϱ in the body of the `letregion` expression. The expression is then evaluated under the extended stack $S, \mathfrak{r} \mapsto \{\}$ (that is, the stack S extended with an empty region (bound to $\mathfrak{r}$)), yielding a modified stack (of the form $S', \mathfrak{r} \mapsto R'$) and a value v'. The modified top region is discarded, while occurrences of $\mathfrak{r}$ are replaced by $\bullet$ in the modified stack S' and value v'. This replacement ensures that any occurrences of `ref` terms in S' or v' are marked as dead, since the region has been deallocated and is no longer accessible.

It is important to note that the execution of any expression that allocates or reads a region allocated value is predicated upon having a live region in the stack. While it will be possible to have expressions that reference deallocated regions, it will not be possible to evaluate them. The type-and-effect systems of the next

$\boxed{e \Downarrow_{\text{prog}} v}$

$$\frac{(\cdot, \mathfrak{H} \mapsto \{\}; e[\mathfrak{H}/\mathcal{H}]) \Downarrow (\cdot; \mathfrak{H} \mapsto R'; v')}{e \Downarrow_{\text{prog}} v'[\bullet/\mathfrak{H}]}$$

Figure 2.9: Dynamic semantics of URC (programs)

sections ensure that these invariants are preserved during the execution of well-typed programs.

Finally, there is a special rule for the evaluation of surface programs (Figure 2.9). Programs in the Untyped Region Calculus are simply terms. We distinguish programs because the type-and-effect systems presented in the next sections have special judgments for top-level programs. Essentially, this judgment establishes reasonable "boundary conditions" for a program's execution, an aspect that is often overlooked in other descriptions of region calculi. Programs are evaluated under a stack with a distinguished region $\mathfrak{H}$, which is substituted for the region variable $\mathcal{H}$ during the evaluation of the program. Essentially, one can consider the evaluation of a program e as being equivalent to the evaluation of the expression `letregion` $\mathcal{H}$ `in` e, where the final stack is discarded.

2.2.3 Static Semantics of the Traditional Region Calculus

Our first type-and-effect system for URC is the Traditional Region Calculus (TRC), which corresponds to type-and-effect systems given in the literature [39, 10, 11].

The static semantics of TRC modifies the surface syntax of URC by adding the syntactic classes of boxed types and types and adding effect annotations to functions and region abstractions (Figure 2.10).

Boxed types

$$\omega \quad ::= \quad \mathsf{Int} \mid \tau_1 \xrightarrow{\phi'} \tau_2 \mid \tau_1 \times \cdots \times \tau_n \mid \quad \mid \forall \varrho.^{\phi'}\, \tau$$

Types

$$\tau \quad ::= \quad \mathsf{Bool} \mid (\omega, \rho)$$

Surface expressions

$$e \quad ::= \quad \cdots \mid \lambda x{:}\tau.^{\phi'}\, e\, \mathtt{at}\, \rho \mid \Lambda \varrho.^{\phi'}\, u\, \mathtt{at}\, \rho$$

Figure 2.10: Surface syntax of TRC

As noted before, a region is associated with every value that requires heap allocated storage. This is reflected in the syntax of types. The type (ω, ρ) pairs together a boxed type (a type requiring heap allocated storage) and a region placeholder; we interpret (ω, ρ) as the type of values of boxed type ω allocated in region ρ. The forms of the function boxed type $(\tau_1 \xrightarrow{\phi'} \tau_2)$ and the region-abstraction boxed type $(\forall \varrho.^{\phi'}\, \tau)$ are defining characteristics of "traditional" type-and-effect systems. Recall that an effect ϕ is a finite set of regions. In the function and region-abstraction types, the effect ϕ' is a *latent effect*: a (super)set of those regions allocated in or read from when the function or region abstraction is applied and evaluated.

Definitions Figure 2.11 presents additional definitions for syntactic objects that appear in the static semantics. Contexts Δ are ordered lists of region variables and contexts Γ are ordered lists of variables with types. We tacitly assume that all contexts are well-formed: Δ contains distinct region variables and Γ contains distinct value variables.

$$\begin{array}{lrcl} \text{Region contexts} & \Delta & ::= & \cdot \mid \Delta, \varrho \\ \text{Value contexts} & \Gamma & ::= & \cdot \mid \Gamma, x{:}\tau \end{array}$$

Figure 2.11: Static semantics of TRC (definitions)

Terms Figure 2.12 gives an abbreviated static semantics for TRC; we omit some of the auxiliary judgments and some typing rules for expressions, as they are similar to the ones presented in full for the Single Effect Calculus in Section 2.2.5.

The judgment $\Delta; \Gamma \vdash_{\text{exp}} e : \tau, \phi$ asserts that under the region context Δ and the value context Γ, the expression e has the type τ and the effect ϕ. The effect ϕ describes the regions that may be accessed when the expression is evaluated.

The rules for constants, arithmetic and boolean operations, function abstraction and application, tuple introduction and selection, and region abstraction and instantiation all have similar forms. Rules for those expression introduction forms with a region annotation `at` ρ add $\{\rho\}$ to the effect of the entire expression, indicating that ρ is accessed in order to allocate a value during evaluation. Rules for expression elimination forms for region allocated values also add $\{\rho\}$ to the effect of the entire expression, indicating that $\{\rho\}$ is accessed in order to read the value during evaluation. The rules for function and region abstraction check that the bodies have the correct latent effect, while the rules for application and region instantiation add the latent effect to the effect of the entire expression. Finally, the rules generally accumulate the effect of sub-expressions (e.g., the rule for tuple introduction).

The key rule in region calculi is the typing rule for `letregion`:

$$\frac{\Delta \vdash_{\text{type}} \tau \qquad \vdash_{\text{ctxt}} \Delta; \Gamma; (\phi \setminus \varrho) \qquad \Delta, \varrho; \Gamma \vdash_{\text{exp}} e : \tau, \phi}{\Delta; \Gamma \vdash_{\text{exp}} \texttt{letregion}\ \varrho\ \texttt{in}\ e : \tau, \phi \setminus \varrho}$$

$\boxed{\Delta; \Gamma \vdash_{\text{exp}} e : \tau, \phi}$

$$\frac{\Delta; \Gamma \vdash_{\text{exp}} e_1 : \tau_1, \phi_1 \qquad \cdots \qquad \Delta; \Gamma \vdash_{\text{exp}} e_n : \tau_n, \phi_n \qquad \Delta \vdash_{\text{region}} \rho}{\Delta; \Gamma \vdash_{\text{exp}} \langle e_1, \ldots, e_n \rangle \, \texttt{at} \, \rho : (\tau_1 \times \cdots \times \tau_n, \rho), \phi_1 \cup \cdots \cup \phi_n \cup \{\rho\}}$$

$$\frac{\Delta; \Gamma \vdash_{\text{exp}} e : (\tau_1 \times \cdots \times \tau_n, \rho), \phi \qquad 0 \leq i \leq n}{\Delta; \Gamma \vdash_{\text{exp}} \texttt{sel}_i \; e : \tau_i, \phi \cup \{\rho\}}$$

$$\frac{\Delta; \Gamma, x : \tau_x \vdash_{\text{exp}} e : \tau, \phi' \qquad \Delta \vdash_{\text{region}} \rho}{\Delta; \Gamma \vdash_{\text{exp}} \lambda x : \tau_x .^{\phi'} \, e \, \texttt{at} \, \rho : (\tau_x \xrightarrow{\phi'} \tau, \rho), \{\rho\}}$$

$$\frac{\Delta; \Gamma \vdash_{\text{exp}} e_f : (\tau_x \xrightarrow{\phi'_f} \tau, \rho_f), \phi_f \qquad \Delta; \Gamma \vdash_{\text{exp}} e_a : \tau_x, \phi_a}{\Delta; \Gamma \vdash_{\text{exp}} e_f \; e_a : \tau, \phi_f \cup \phi_a \cup \{\rho_f\} \cup \phi'_f}$$

$$\frac{\Delta \vdash_{\text{type}} \tau \qquad \vdash_{\text{ctxt}} \Delta; \Gamma; (\phi \setminus \varrho) \qquad \Delta, \varrho; \Gamma \vdash_{\text{exp}} e : \tau, \phi}{\Delta; \Gamma \vdash_{\text{exp}} \texttt{letregion} \; \varrho \; \texttt{in} \; e : \tau, \phi \setminus \varrho}$$

$$\frac{\Delta, \varrho; \Gamma \vdash_{\text{exp}} u : \tau, \phi' \qquad \Delta \vdash_{\text{region}} \rho}{\Delta; \Gamma \vdash_{\text{exp}} \Lambda \varrho .^{\phi'} \, u \, \texttt{at} \, \rho : (\forall \varrho .^{\phi'} \, \tau, \rho), \{\rho\}}$$

$$\frac{\Delta; \Gamma \vdash_{\text{exp}} e_f : (\forall \varrho .^{\phi'_f} \, \tau, \rho_f), \phi_f \qquad \Delta \vdash_{\text{region}} \rho_a}{\Delta; \Gamma \vdash_{\text{exp}} e_f \; [\rho_a] : \tau[\rho_a/\varrho], \phi_f \cup \{\rho_f\} \cup \phi'_f[\rho_a/\varrho]}$$

$\boxed{\vdash_{\text{prog}} e}$

$$\frac{\cdot, \mathcal{H}; \cdot \vdash_{\text{exp}} e : \mathsf{Bool}, \phi \qquad \phi \subseteq \{\mathcal{H}\}}{\vdash_{\text{prog}} e}$$

Figure 2.12: Static semantics of TRC (abbreviated)

The antecedent $\Delta \vdash_{\text{type}} \tau$ asserts that the new region variable ϱ does not appear in the result type; in particular, it does not appear in any effects occurring in function or region abstraction types that appear in the result. Note further that the implicit antecedent $\varrho \notin dom(\Delta)$ and the explicit antecedent $\vdash_{\text{ctxt}} \Delta; \Gamma; (\phi \setminus \varrho)$ ensure that ϱ does not appear in the types of the value environment nor in the effect of the entire expression. Together, these facts guarantee that the region ϱ is not needed before the evaluation of e, nor is it needed after, corresponding to the allocation and deallocation of a new region. Nonetheless, the region ϱ may appear in the effect of the body $(\Delta, \varrho; \Gamma \vdash_{\text{exp}} e : \tau, \phi)$.

Note that the typing rules rely upon set theoretic operations ($\in$, $\cup$, and $\setminus$) to check and synthesize effects. As the translation in Chapter 3 will require witnessing effect subsumption by explicit coercions, the Bounded Region Calculus and the Single Effect Calculus of the next sections will formalize these relations as separate judgments.

2.2.4 Static Semantics of the Bounded Region Calculus

Our second type-and-effect system for URC is the Bounded Region Calculus (BRC), which augments TRC with a form of bounded region polymorphism. The Bounded Region Calculus can be seen as a core model of early versions of Cyclone [30, 29]. One key difference (among many) between Cyclone and the Tofte-Talpin region calculus is that the type-and-effects system of Cyclone extends that of Tofte-Talpin's with a form of bounded region polymorphism. The abstraction of a region variable ϱ may be bounded by a set of regions ϕ. At the instantiation of a region variable ϱ by a region $\mathfrak{r}$, we must show that the liveness of the region $\mathfrak{r}$ implies the liveness of all the regions in ϕ. Within the body of the abstraction, we may assume

Boxed types

$$\omega \quad ::= \quad \mathsf{Int} \mid \tau_1 \xrightarrow{\phi'} \tau_2 \mid \tau_1 \times \cdots \times \tau_n \mid \quad \mid \forall \varrho \succeq \phi.^{\phi'} \tau$$

Types

$$\tau \quad ::= \quad \mathsf{Bool} \mid (\omega, \rho)$$

Surface expressions

$$e \quad ::= \quad \cdots \mid \lambda x{:}\tau.^{\phi'} e \,\mathtt{at}\, \rho \mid \Lambda \varrho \succeq \phi.^{\phi'} u \,\mathtt{at}\, \rho$$

Figure 2.13: Surface syntax of BRC

that ϱ is an upper bound on the set of regions ϕ. However, like the Tofte-Talpin region calculus, Cyclone treats effects as sets of regions affected by the evaluation of an expression.

The static semantics of TRC modifies the surface syntax of URC by adding the syntactic classes of boxed types and types, adding effect annotations to functions and region abstractions, and adding an effect bound to region abstractions (Figure 2.13).

In a region-abstraction type $\forall \varrho \succeq \phi.^{\phi'} \tau$, the effect ϕ serves as a lower bound on the lifetime of the region variable ϱ. (Note that the region variable ϱ is bound within ϕ' and τ, but not ϕ.) The abstraction can only be instantiated by a region ρ that has been pushed on the stack more recently than those regions in ϕ. Within the body of the abstraction, we may safely assume that ϱ is outlived by all of the regions in ϕ. Put another way, if ϱ is live, then all of the regions in ϕ must be live.

Definitions Figure 2.14 presents additional definitions for syntactic objects that appear in the static semantics. Contexts Δ are ordered lists of region variables bounded by effects (finite sets of regions) and contexts Γ are ordered lists of vari-

$$\begin{array}{lrcl} \text{Region contexts} & \Delta & ::= & \cdot \mid \Delta, \varrho \succeq \phi \\ \text{Value contexts} & \Gamma & ::= & \cdot \mid \Gamma, x{:}\tau \end{array}$$

Figure 2.14: Static semantics of BRC (definitions)

ables with types. We tacitly assume that all contexts are well-formed: Δ contains distinct region variables and Γ contains distinct value variables.

Outlives judgments Figure 2.15 gives the judgments that formalize the liveness relationships between regions and effects. We summarize these judgments in the following table:

Judgment	Meaning
$\Delta \vdash_{\text{rr}} \rho_2 \succeq \rho_1$	If region ρ_2 is live, then region ρ_1 is live. (Alt.: region ρ_1 outlives region ρ_2.)
$\Delta \vdash_{\text{re}} \rho \succeq \phi$	If region ρ is live, then all regions in ϕ are live. (Alt.: all regions in ϕ outlive region ρ.)
$\Delta \vdash_{\text{er}} \phi \ni \rho$	Region ρ is a region in ϕ.
$\Delta \vdash_{\text{ee}} \phi \supseteq \phi'$	All region in ϕ' are regions in ϕ.

We note that the typing rules for the judgments $\vdash_{\text{rr}}$ and $\vdash_{\text{re}}$ simply formalize the reflexive, transitive closure of the syntactic constraints in Δ, each of which asserts a particular "outlived by" relation between a region variable and an effect set. Likewise, the judgments $\vdash_{\text{er}}$ and $\vdash_{\text{rr}}$ formalize the set theoretic operations used by the Traditional Region Calculus. The $\Delta \vdash_{\text{region}} \rho$ and $\Delta \vdash_{\text{eff}} \phi$ judgments check that ρ and ϕ, respectively, are well-formed in the region context Δ (see Figure 2.25).

Terms Figures 2.16 and 2.17 give an abbreviated static semantics for BRC; we omit some of the auxiliary judgments and some typing rules for expressions, as

$\boxed{\Delta \vdash_{\text{rr}} \rho_2 \succeq \rho_1}$

$$\frac{(\varrho \succeq \{\rho_1, \ldots, \rho_i, \ldots, \rho_n\}) \in \Delta}{\Delta \vdash_{\text{rr}} \varrho \succeq \rho_i} \qquad \frac{\Delta \vdash_{\text{region}} \rho}{\Delta \vdash_{\text{rr}} \rho \succeq \rho} \qquad \frac{\Delta \vdash_{\text{rr}} \rho_2 \succeq \rho' \qquad \Delta \vdash_{\text{rr}} \rho' \succeq \rho_1}{\Delta \vdash_{\text{rr}} \rho_2 \succeq \rho_1}$$

$\boxed{\Delta \vdash_{\text{re}} \rho \succeq \phi}$

$$\frac{\Delta \vdash_{\text{rr}} \rho \succeq \rho_i \quad {}^{i \in 1 \ldots n}}{\Delta \vdash_{\text{re}} \rho \succeq \{\rho_1, \ldots, \rho_n\}}$$

$\boxed{\Delta \vdash_{\text{er}} \phi \ni \rho}$

$$\frac{\Delta \vdash_{\text{eff}} \{\rho_1, \ldots, \rho_n\}}{\Delta \vdash_{\text{er}} \{\rho_1, \ldots, \rho_n\} \ni \rho_i}$$

$\boxed{\Delta \vdash_{\text{ee}} \phi \supseteq \phi'}$

$$\frac{\Delta \vdash_{\text{eff}} \phi \qquad \Delta \vdash_{\text{er}} \phi \ni \rho_i \quad {}^{i \in 1 \ldots n}}{\Delta \vdash_{\text{ee}} \phi \supseteq \{\rho_1, \ldots, \rho_n\}}$$

Figure 2.15: Static semantics of BRC (outlives judgments)

$\boxed{\Delta; \Gamma \vdash_{\text{exp}} e : \tau, \phi}$

$$\frac{\Delta; \Gamma \vdash_{\text{exp}} e_1 : \tau_1, \phi \qquad \cdots \qquad \Delta; \Gamma \vdash_{\text{exp}} e_n : \tau_n, \phi \qquad \Delta \vdash_{\text{region}} \rho \qquad \Delta \vdash_{\text{er}} \phi \ni \rho}{\Delta; \Gamma \vdash_{\text{exp}} \langle e_1, \ldots, e_n \rangle \,\texttt{at}\, \rho : (\tau_1 \times \cdots \times \tau_n, \rho), \phi}$$

$$\frac{\Delta; \Gamma \vdash_{\text{exp}} e : (\tau_1 \times \cdots \times \tau_n, \rho), \phi \qquad \Delta \vdash_{\text{er}} \rho \ni \phi \qquad 0 \leq i \leq n}{\Delta; \Gamma \vdash_{\text{exp}} \texttt{sel}_i \; e : \tau_i, \phi}$$

$$\frac{\Delta; \Gamma, x : \tau_x \vdash_{\text{exp}} e : \tau, \phi' \qquad \Delta \vdash_{\text{region}} \rho \qquad \Delta \vdash_{\text{er}} \phi \ni \rho}{\Delta; \Gamma \vdash_{\text{exp}} \lambda x{:}\tau_x.^{\phi'} \, e \,\texttt{at}\, \rho : (\tau_x \xrightarrow{\phi'} \tau, \rho), \phi}$$

$$\frac{\begin{array}{c}\Delta; \Gamma \vdash_{\text{exp}} e_f : (\tau_x \xrightarrow{\phi'_f} \tau, \rho_f), \phi \qquad \Delta \vdash_{\text{er}} \phi \ni \rho_f \\ \Delta; \Gamma \vdash_{\text{exp}} e_a : \tau_x, \phi \qquad \Delta \vdash_{\text{ee}} \phi \supseteq \phi'_f\end{array}}{\Delta; \Gamma \vdash_{\text{exp}} e_f \; e_a : \tau, \phi}$$

$$\frac{\begin{array}{c}\Delta \vdash_{\text{type}} \tau \qquad \vdash_{\text{ctxt}} \Delta; \Gamma; \{\rho_1, \ldots, \rho_n\} \\ \Delta, \varrho \succeq \{\rho_1, \ldots, \rho_n\}; \Gamma \vdash_{\text{exp}} e : \tau, \{\rho_1, \ldots, \rho_n, \varrho\}\end{array}}{\Delta; \Gamma \vdash_{\text{exp}} \texttt{letregion}\; \varrho \;\texttt{in}\; e : \tau, \{\rho_1, \ldots, \rho_n\}}$$

$$\frac{\Delta, \varrho \succeq \phi_b; \Gamma \vdash_{\text{exp}} u : \tau, \phi' \qquad \Delta \vdash_{\text{region}} \rho \qquad \Delta \vdash_{\text{er}} \phi \ni \rho}{\Delta; \Gamma \vdash_{\text{exp}} \Lambda \varrho \succeq \phi_b.^{\phi'} \, u \,\texttt{at}\, \rho : (\forall \varrho \succeq \phi_b.^{\phi'} \, \tau, \rho), \phi}$$

$$\frac{\begin{array}{c}\Delta; \Gamma \vdash_{\text{exp}} e_f : (\forall \varrho \succeq \phi_b.^{\phi'_f} \, \tau, \rho_f), \phi \qquad \Delta \vdash_{\text{er}} \phi \ni \rho_f \\ \Delta \vdash_{\text{region}} \rho_a \qquad \Delta \vdash_{\text{re}} \rho_a \succeq \phi_b \qquad \Delta \vdash_{\text{ee}} \phi \supseteq \phi'_f[\rho_a/\varrho]\end{array}}{\Delta; \Gamma \vdash_{\text{exp}} e_f \; [\rho_a] : \tau[\rho_a/\varrho], \phi}$$

Figure 2.16: Static semantics of BRC (abbreviated (I))

$\boxed{\vdash_{\text{prog}} e}$

$$\frac{\cdot, \mathcal{H} \succeq \{\}; \cdot \vdash_{\text{exp}} e : \mathsf{Bool}, \{\mathcal{H}\}}{\vdash_{\text{prog}} e}$$

Figure 2.17: Static semantics of BRC (abbreviated (II))

they are similar to the ones presented in full for the Single Effect Calculus in Section 2.2.5.

As in TRC, the judgment $\Delta; \Gamma \vdash_{\text{exp}} e : \tau, \phi$ asserts that under the region context Δ and the value context Γ, the expression e has the type τ and the effect ϕ, which describes the regions that may be accessed when the expression is evaluated.

The rules for constants, arithmetic and boolean operations, function abstraction and application, tuple introduction and selection, and region abstraction and instantiation all have similar forms. Rules for those expression introduction forms with a region annotation `at` ρ check that ρ is in the effect of the entire expression ($\Delta \vdash_{er} \phi \ni \rho$), since ρ is accessed in order to allocate a value during evaluation. Rules for expression elimination forms for region allocated values also check that ρ is in the effect of the entire expression, since ρ is accessed in order to read the value during evaluation. The rule for function abstraction checks that the body has the correct latent effect, while the rule for function application checks that the latent effect is in the effect of the entire expression ($\Delta \vdash_{ee} \phi \supseteq \phi'_f$).

The rules for region abstraction and instantiation are similar, except that the rule for region instantiation requires that we be able to show that the region argument ρ_a is outlived by all of the regions in the region abstraction bound ϕ_b; it further checks that the latent effect (with the region argument ρ_a substituted for the region variable ϱ) is in the effect of the entire expression ($\Delta \vdash_{\text{ee}} \phi \supseteq \phi'[\rho_a/\varrho]$).

Finally, the rules generally require sub-expressions to have the same effect as the entire expression (e.g., the rule for tuple introduction).

As always, the key rule is the typing rule for `letregion`:

$$\frac{\Delta \vdash_{\text{type}} \tau \qquad \vdash_{\text{ctxt}} \Delta; \Gamma; \{\rho_1, \ldots, \rho_n\} \qquad \Delta, \varrho \succeq \{\rho_1, \ldots, \rho_n\}; \Gamma \vdash_{\text{exp}} e : \tau, \{\rho_1, \ldots, \rho_n, \varrho\}}{\Delta; \Gamma \vdash_{\text{exp}} \texttt{letregion}\ \varrho\ \texttt{in}\ e : \tau, \{\rho_1, \ldots, \rho_n\}}$$

As before, the rule ensures that the new region ϱ is not needed before the evaluation of e, nor is it needed after, corresponding to the allocation and deallocation of a new region. Furthermore, the rule relates the new region to the currently live regions by introducing ϱ into the region context with an appropriate bound: while ϱ is live, all regions in $\{\rho_1, \ldots, \rho_n\}$ are live.

Note that the typing rules have replaced the set theoretic operations used in TRC with separate judgments.

Translation of TRC to BRC

There is a trivial translation from the Traditional Region Calculus to the Bounded Region Calculus, whereby every region abstraction becomes a region abstraction with an empty bound; Figure 2.18 gives an abbreviated translation (the translation is homomorphic in the other syntactic forms).

Type preservation corresponds to the validity of effect enlargement.

Lemma 2.1 (Translation Preserves Types)

(1) If $\Delta; \Gamma \vdash^{\mathsf{TRC}}_{\text{exp}} e : \tau, \phi$, *then forall* ϕ', *if* $\vdash^{\mathsf{BRC}}_{\text{ctxt}} \tilde{\mathbb{D}}[\![\Delta]\!]\,; \tilde{\mathbb{G}}[\![\Gamma]\!]\,; \phi'$ *and* $\tilde{\mathbb{D}}[\![\Delta]\!] \vdash^{\mathsf{BRC}}_{\text{ee}} \phi' \supseteq \phi$, *then* $\tilde{\mathbb{D}}[\![\Delta]\!]\,; \tilde{\mathbb{G}}[\![\Gamma]\!] \vdash^{\mathsf{BRC}}_{\text{exp}} \tilde{\mathbb{E}}[\![e]\!] : \tilde{\mathbb{T}}[\![\tau]\!]\,, \phi'$.

(2) If $\vdash^{\mathsf{TRC}}_{\text{prog}} e$, *then* $\vdash^{\mathsf{BRC}}_{\text{prog}} \tilde{\mathbb{E}}[\![e]\!]$.

Boxed types

$$\begin{aligned}\tilde{\mathbb{T}}\left[\!\left[\tau_1 \xrightarrow{\phi'} \tau_2\right]\!\right] &= \tilde{\mathbb{T}}[\![\tau_1]\!] \xrightarrow{\phi'} \tilde{\mathbb{T}}[\![\tau_2]\!] \\ \tilde{\mathbb{T}}[\![\forall \varrho.^{\phi'}\, \tau]\!] &= \forall \varrho \succeq \{\}.^{\phi'}\, \tilde{\mathbb{T}}[\![\tau]\!]\end{aligned}$$

Types

$$\tilde{\mathbb{T}}[\![(\omega, \rho)]\!] = (\tilde{\mathbb{T}}[\![\omega]\!], \rho)$$

Expressions

$$\begin{aligned}\tilde{\mathbb{E}}\left[\!\left[\lambda x{:}\tau.^{\phi'}\, e\, \mathtt{at}\, \rho\right]\!\right]_\pi &= \lambda x{:}\tilde{\mathbb{T}}[\![\tau]\!].^{\phi'}\, \tilde{\mathbb{E}}[\![e]\!]\ \mathtt{at}\ \rho) \\ \tilde{\mathbb{E}}[\![e_1\ e_2]\!] &= \tilde{\mathbb{E}}[\![e_1]\!]\ \tilde{\mathbb{E}}[\![e_2]\!] \\ \tilde{\mathbb{E}}[\![\mathtt{letregion}\ \varrho\ \mathtt{in}\ e]\!] &= \mathtt{letregion}\ \varrho\ \mathtt{in}\ \tilde{\mathbb{E}}[\![e]\!] \\ \tilde{\mathbb{E}}\left[\!\left[\lambda \varrho.^{\phi'}\, u\, \mathtt{at}\, \rho\right]\!\right] &= \lambda \varrho \succeq \{\}.^{\phi'}\, \tilde{\mathbb{E}}[\![u]\!]\ \mathtt{at}\ \rho \\ \tilde{\mathbb{E}}[\![e\ [\rho]]\!] &= \tilde{\mathbb{E}}[\![e]\!]\ [\rho]\end{aligned}$$

Programs

$$\tilde{\mathbb{E}}[\![e]\!] = \tilde{\mathbb{E}}[\![e]\!]$$

Figure 2.18: Translation from TRC to BRC (abbreviated)

Boxed types

$$\omega \quad ::= \quad \mathsf{Int} \mid \tau_1 \xrightarrow{\pi'} \tau_2 \mid \tau_1 \times \cdots \times \tau_n \mid \quad \mid \forall \varrho \succeq \phi.^{\pi'} \tau$$

Types

$$\tau \quad ::= \quad \mathsf{Bool} \mid (\omega, \rho)$$

Surface expressions

$$e \quad ::= \quad \cdots \mid \lambda x{:}\tau.^{\pi'} e \,\mathtt{at}\, \rho \mid \Lambda \varrho \succeq \phi.^{\pi'} u \,\mathtt{at}\, \rho$$

Figure 2.19: Surface syntax of SEC

Meaning preservation is trivial, as the languages share the same dynamic semantics.

Lemma 2.2 (Translation Correctness (Programs))

If $\vdash^{\mathsf{TRC}}_{\mathrm{prog}} e$ *and* $e \Downarrow_{\mathrm{prog}} \mathfrak{b}$ *and* $\tilde{\mathbb{E}}[\![e]\!] = e^{\dagger}$,
then $e^{\dagger} \Downarrow_{\mathrm{prog}} \mathfrak{b}$.

2.2.5 Static Semantics of the Single Effect Calculus

Our third type-and-effect system for URC is the Single Effect Calculus (SEC), which is a restricted form of BRC, where latent effects consist of a *single region* instead of a finite set of regions. As convention, we will use π to represent regions that correspond to such effects and we will use ϖ to represent region variables that correspond to such effects.

Because SEC will be the source of our translation into the monadic type system of Chapter 3, we present the static semantics in somewhat more detail than previous systems.

$$\begin{array}{lrcl} \text{Region contexts} & \Delta & ::= & \cdot \mid \Delta, \varrho \succeq \phi \\ \text{Expression contexts} & \Gamma & ::= & \cdot \mid \Gamma, x{:}\tau \end{array}$$

Figure 2.20: Static semantics of SEC (definitions)

The static semantics of SEC modifies the syntax of URC by adding the syntactic classes of boxed types and types, adding effect annotations to functions and region abstractions, and adding an effect bound to region abstractions (Figure 2.13).

In SEC, the latent effect π' of functions and region abstractions denotes an upper bound (in the ordering of live regions) on the set of regions affected when the function or region abstraction is applied and evaluated.

Definitions Figure 2.20 presents additional definitions for syntactic objects that appear in the static semantics. Contexts Δ are ordered lists of region variables bounded by effects (finite sets of regions) and contexts Γ are ordered lists of variables with types. We tacitly assume that all contexts are well-formed: Δ contains distinct region variables and Γ contains distinct value variables.

Outlives judgments Figure 2.21 reproduces the judgments $\Delta \vdash_{\text{rr}} \rho_2 \succeq \rho_1$ and $\Delta \vdash_{\text{re}} \rho \succeq \phi$ from BRC. Note that for SEC, we do not require the other judgments ($\Delta \vdash_{\text{er}} \phi \ni \rho$ and $\Delta \vdash_{\text{ee}} \phi \supseteq \phi'$).

Terms Figures 2.22, 2.23, and 2.24 present the typing rules for the judgment $\Delta; \Gamma; \vdash_{\text{exp}} e : \tau, \pi$, which asserts that under region context Δ and value context Γ, the expression e has type τ and effects bounded by the region π. In practice, and as suggested by the typing rules, π usually corresponds to the most recently allocated region (also referred to as the top or current region).

$\boxed{\Delta \vdash_{\text{rr}} \rho_2 \succeq \rho_1}$

$$\frac{\vdash_{\text{rctxt}} \Delta \qquad (\varrho \succeq \{\rho_1, \ldots, \rho_i, \ldots, \rho_n\}) \in \Delta}{\Delta \vdash_{\text{rr}} \varrho \succeq \rho_i} \qquad \frac{\Delta \vdash_{\text{region}} \rho}{\Delta \vdash_{\text{rr}} \rho \succeq \rho}$$

$$\frac{\Delta \vdash_{\text{rr}} \rho_2 \succeq \rho' \qquad \Delta \vdash_{\text{rr}} \rho' \succeq \rho_1}{\Delta \vdash_{\text{rr}} \rho_2 \succeq \rho_1}$$

$\boxed{\Delta \vdash_{\text{re}} \rho \succeq \phi}$

$$\frac{\vdash_{\text{rctxt}} \Delta \qquad \Delta \vdash_{\text{rr}} \rho \succeq \rho_i \quad {}^{i \in 1 \ldots n}}{\Delta \vdash_{\text{re}} \rho \succeq \{\rho_1, \ldots, \rho_n\}}$$

Figure 2.21: Static semantics of SEC (outlives judgments)

The rules for constants, arithmetic and boolean operations, function abstraction and application, tuple introduction and selection, and region abstraction and instantiation all have similar forms. Rules for those expression introduction forms with a region annotation `at` ρ check that the liveness of ρ is implied by the liveness of the single region bounding the effect for the entire expression ($\Delta \vdash_{rr} \pi \succeq \rho$), since ρ is accessed in order to allocate a value during evaluation. Rules for expression elimination forms for region allocated values also check that the liveness of ρ is implied by the single region bounding the effect of the entire expression, since ρ is accessed in order to read the value during evaluation. The rule for function abstraction checks that the body has the correct single region bounding the latent effect, while the rule for function application checks that the liveness of single region is implied by the liveness of the single region bounding the effect of the entire expression ($\Delta \vdash_{rr} \pi \succeq \pi_f$).

$\boxed{\Delta; \Gamma \vdash_{\text{exp}} e : \tau, \pi}$

$$\frac{\vdash_{\text{ctxt}} \Delta; \Gamma; \pi \qquad \Delta \vdash_{\text{region}} \rho \qquad \Delta \vdash_{\text{rr}} \pi \succeq \rho}{\Delta; \Gamma \vdash_{\text{exp}} \mathtt{i}\,\mathtt{at}\,\rho : (\mathsf{Int}, \rho), \pi}$$

$$\frac{\begin{array}{c}\Delta; \Gamma \vdash_{\text{exp}} e_1 : (\mathsf{Int}, \rho_1), \pi \qquad \Delta \vdash_{\text{rr}} \pi \succeq \rho_1 \\ \Delta; \Gamma \vdash_{\text{exp}} e_2 : (\mathsf{Int}, \rho_2), \pi \qquad \Delta \vdash_{\text{rr}} \pi \succeq \rho_2 \\ \Delta \vdash_{\text{region}} \rho \qquad \Delta \vdash_{\text{rr}} \pi \succeq \rho\end{array}}{\Delta; \Gamma \vdash_{\text{exp}} e_1 \oplus e_2\,\mathtt{at}\,\rho : (\mathsf{Int}, \rho), \pi}$$

$$\frac{\begin{array}{c}\Delta; \Gamma \vdash_{\text{exp}} e_1 : (\mathsf{Int}, \rho_1), \pi \qquad \Delta \vdash_{\text{rr}} \pi \succeq \rho_1 \\ \Delta; \Gamma \vdash_{\text{exp}} e_2 : (\mathsf{Int}, \rho_2), \pi \qquad \Delta \vdash_{\text{rr}} \pi \succeq \rho_2\end{array}}{\Delta; \Gamma \vdash_{\text{exp}} e_1 \lessgtr e_2 : \mathsf{Bool}, \pi} \qquad \frac{\vdash_{\text{ctxt}} \Delta; \Gamma; \pi}{\Delta; \Gamma \vdash_{\text{exp}} \mathfrak{b} : \mathsf{Bool}, \pi}$$

$$\frac{\begin{array}{c}\Delta; \Gamma \vdash_{\text{exp}} e_b : \mathsf{Bool}, \pi \\ \Delta; \Gamma \vdash_{\text{exp}} e_t : \tau, \pi \qquad \Delta; \Gamma \vdash_{\text{exp}} e_f : \tau, \pi\end{array}}{\Delta; \Gamma \vdash_{\text{exp}} \mathtt{if}\ e_b\ \mathtt{then}\ e_t\ \mathtt{else}\ e_f : \tau, \pi}$$

Figure 2.22: Static semantics of SEC (expressions (I))

$\boxed{\Delta; \Gamma \vdash_{\text{exp}} e : \tau, \pi}$

$$\frac{\vdash_{\text{ctxt}} \Delta; \Gamma; \pi \qquad x \in dom(\Gamma) \qquad \Gamma(x) = \tau}{\Delta; \Gamma \vdash_{\text{exp}} x : \tau, \pi}$$

$$\frac{\Delta; \Gamma, x{:}\tau_x \vdash_{\text{exp}} e : \tau, \pi' \qquad \Delta \vdash_{\text{region}} \rho \qquad \Delta \vdash_{\text{rr}} \pi \succeq \rho}{\Delta; \Gamma \vdash_{\text{exp}} \lambda x : \tau_x.^{\pi'}\, e\, \texttt{at}\, \rho : (\tau_x \xrightarrow{\pi'} \tau, \rho), \pi}$$

$$\frac{\begin{array}{c}\Delta; \Gamma \vdash_{\text{exp}} e_f : (\tau_x \xrightarrow{\pi'_f} \tau, \rho_f), \pi \qquad \Delta \vdash_{\text{rr}} \pi \succeq \rho_f \\ \Delta; \Gamma \vdash_{\text{exp}} e_a : \tau_x, \pi \qquad \Delta \vdash_{\text{rr}} \pi \succeq \pi'_f\end{array}}{\Delta; \Gamma \vdash_{\text{exp}} e_f\ e_a : \tau, \pi}$$

$$\frac{\begin{array}{c}\Delta; \Gamma \vdash_{\text{exp}} e_1 : \tau_1, \pi \qquad \cdots \qquad \Delta; \Gamma \vdash_{\text{exp}} e_n : \tau_n, \pi \\ \Delta \vdash_{\text{region}} \rho \qquad \Delta \vdash_{\text{rr}} \pi \succeq \rho\end{array}}{\Delta; \Gamma \vdash_{\text{exp}} \langle e_1, \ldots, e_n \rangle\, \texttt{at}\, \rho : (\tau_1 \times \cdots \times \tau_n, \rho), \pi}$$

$$\frac{\begin{array}{c}\Delta; \Gamma \vdash_{\text{exp}} e : (\tau_1 \times \cdots \times \tau_n, \rho), \pi \\ \Delta \vdash_{\text{rr}} \pi \succeq \rho \qquad 0 \leq i \leq n\end{array}}{\Delta; \Gamma \vdash_{\text{exp}} \texttt{sel}_i\ e : \tau_i, \pi}$$

Figure 2.23: Static semantics of SEC (expressions (II))

$\boxed{\Delta; \Gamma \vdash_{\text{exp}} e : \tau, \pi}$

$$\frac{\Delta \vdash_{\text{type}} \tau \qquad \vdash_{\text{ctxt}} \Delta; \Gamma; \pi \qquad \Delta, \varrho \succeq \{\pi\}; \Gamma \vdash_{\text{exp}} e_b : \tau, \varrho}{\Delta; \Gamma \vdash_{\text{exp}} \texttt{letregion}\ \varrho\ \texttt{in}\ e_b : \tau, \pi}$$

$$\frac{\Delta, \varrho \succeq \phi; \Gamma \vdash_{\text{exp}} u : \tau, \pi' \qquad \Delta \vdash_{\text{region}} \rho \qquad \Delta \vdash_{\text{rr}} \pi \succeq \rho}{\Delta; \Gamma \vdash_{\text{exp}} \Lambda\varrho \succeq \phi.^{\pi'}\, u\, \texttt{at}\, \rho : (\forall \varrho \succeq \phi.^{\pi'}\, \tau, \rho), \pi}$$

$$\frac{\Delta; \Gamma \vdash_{\text{exp}} e_f : (\forall \varrho \succeq \phi.^{\pi'_f}\, \tau, \rho_f), \pi \qquad \Delta \vdash_{\text{rr}} \pi \succeq \rho_f \qquad \Delta \vdash_{\text{region}} \rho_a \qquad \Delta \vdash_{\text{re}} \rho_a \succeq \phi \qquad \Delta \vdash_{\text{rr}} \pi \succeq \pi'_f[\rho_a/\varrho]}{\Delta; \Gamma \vdash_{\text{exp}} e_f\ [\rho_a] : \tau[\rho_a/\varrho], \pi}$$

Figure 2.24: Static semantics of SEC (expressions (III))

The rules for region abstraction and instantiation are similar, except that the rule for region instantiation requires that we be able to show that the region argument ρ_a is outlived by all of the regions in the region abstraction bound ϕ_b; it further checks that single region bounding the latent effect (with the region argument ρ_a substituted for the region variable ϱ) is in the effect of the entire expression ($\Delta \vdash_{\text{rr}} \pi \succeq \pi_f[\rho_a/\varrho]$).

Finally, the rules generally require sub-expressions to have the same single region effect as the entire expression (e.g., the rule for tuple introduction).

As always, the key rule is the typing rule for `letregion`:

$$\frac{\Delta \vdash_{\text{type}} \tau \qquad \vdash_{\text{ctxt}} \Delta; \Gamma; \pi \qquad \Delta, \varrho \succeq \{\pi\}; \Gamma \vdash_{\text{exp}} e_b : \tau, \varrho}{\Delta; \Gamma \vdash_{\text{exp}} \texttt{letregion}\ \varrho\ \texttt{in}\ e_b : \tau, \pi}$$

The antecedent $\Delta \vdash_{\text{type}} \tau$ asserts that the new region variable ϱ does not appear in the result type; in particular, it does not appear in any latent single region effects or in any effect bounds occurring in function or region abstraction types that appear in the result. Note further that the implicit antecedent $\varrho \notin dom(\Delta)$ and the explicit antecedent $\vdash_{\text{ctxt}} \Delta; \Gamma; \pi$ ensure that ϱ does not appear in the types of the value environment. Together, these facts guarantee that the region ϱ is not needed before the evaluation of e, nor is it needed after, corresponding to the allocation and deallocation of a new region. This new region is clearly related to the current region π — it is outlived by the "old" current region and becomes the "new" current region for the evaluation of e. These facts are captured by the final antecedent $\Delta, \varrho \succeq \{\pi\}; \Gamma \vdash_{\text{exp}} e : \tau, \varrho$.

It is worth comparing the treatment of latent effects in the Single Effect Calculus with their treatment in the other two type systems:

$$\frac{\Delta; \Gamma \vdash_{\text{exp}} e_f : (\tau_x \xrightarrow{\phi'_f} \tau, \rho_f), \phi_f \qquad \Delta; \Gamma \vdash_{\text{exp}} e_a : \tau_x, \phi_a}{\Delta; \Gamma \vdash_{\text{exp}} e_f\ e_a : \tau, \phi_f \cup \phi_a \cup \{\rho_f\} \cup \phi'_f} \ \textsf{TRC}$$

$$\frac{\begin{array}{c}\Delta; \Gamma \vdash_{\text{exp}} e_f : (\tau_x \xrightarrow{\phi'_f} \tau, \rho_f), \phi \qquad \Delta \vdash_{\text{er}} \phi \ni \rho_f \\ \Delta; \Gamma \vdash_{\text{exp}} e_a : \tau_x, \phi \qquad \Delta \vdash_{\text{ee}} \phi \supseteq \phi'_f\end{array}}{\Delta; \Gamma \vdash_{\text{exp}} e_f\ e_a : \tau, \phi} \ \textsf{BRC}$$

$$\frac{\begin{array}{c}\Delta; \Gamma \vdash_{\text{exp}} e_f : (\tau_x \xrightarrow{\pi'_f} \tau, \rho_f), \pi \qquad \Delta \vdash_{\text{rr}} \pi \succeq \rho_f \\ \Delta; \Gamma \vdash_{\text{exp}} e_a : \tau_x, \pi \qquad \Delta \vdash_{\text{rr}} \pi \succeq \pi'_f\end{array}}{\Delta; \Gamma \vdash_{\text{exp}} e_f\ e_a : \tau, \pi} \ \textsf{SEC}$$

In the Single Effect Calculus, the composite effect $\phi_f \cup \phi_a \cup \{\rho_f\} \cup \phi'_f$ is witnessed by a single region π that subsumes the effect of the entire expression. We interpret π as an upper bound on the composite effect; hence, π is an upper bound

$\boxed{\Delta \vdash_{\text{region}} \rho}$

$$\frac{\vdash_{\text{rctxt}} \Delta \qquad \varrho \in dom(\Delta)}{\Delta \vdash_{\text{region}} \varrho}$$

$\boxed{\Delta \vdash_{\text{eff}} \varphi}$

$$\frac{\vdash_{\text{rctxt}} \Delta \qquad \Delta \vdash_{\text{region}} \rho_i \quad {}^{i \in 1 \ldots n}}{\Delta \vdash_{\text{eff}} \{\rho_1, \ldots, \rho_n\}}$$

Figure 2.25: Static semantics of SEC (regions and effects)

on each of the effect sets ϕ_f and ϕ_a, which explains why π is used in the antecedents that type-check the sub-expressions e_f and e_a. We require ρ_f to outlive the current region π by the antecedent $\Delta \vdash_{\text{rr}} \pi \succeq \rho_f$. Finally, we require the latent single effect π', which is an upper bound on the set of regions affected by executing the function (ϕ'_f), to outlive the current region, which ensures that π is also an upper bound on the set of regions affected by executing the function.

In the Bounded Region Calculus, the effect for which π is an upper bound is manifest as ϕ. The antecedents $\Delta \vdash_{\text{er}} \phi \ni \rho_f$ and $\Delta \vdash_{\text{ee}} \phi \supseteq \phi'_f$ serve the same purpose as $\Delta \vdash_{\text{rr}} \pi \succeq \rho'_1$ and $\Delta \vdash_{\text{rr}} \pi \succeq \pi'$, namely to ensure that the region of the function closure and the latent effect are subsumed by the effect of the application.

Regions, effects, boxed types, types, and contexts Figures 2.25 and 2.26 contain additional (completely standard) judgments for ensuring that regions ρ, effects ϕ, boxed types ω, types τ, region contexts Δ, and value contexts Γ are well-formed. These judgments simply enforce the invariant that no type or expression may depend upon unbound region variables.

$\boxed{\Delta \vdash_{\text{btype}} \omega}$

$$\frac{\vdash_{\text{rctxt}} \Delta}{\Delta \vdash_{\text{btype}} \mathsf{Int}} \qquad \frac{\Delta \vdash_{\text{type}} \tau_1 \qquad \Delta \vdash_{\text{region}} \pi' \qquad \Delta \vdash_{\text{type}} \tau_2}{\Delta \vdash_{\text{btype}} \tau_1 \xrightarrow{\pi'} \tau_2}$$

$$\frac{\Delta \vdash_{\text{type}} \tau_1 \qquad \cdots \qquad \Delta \vdash_{\text{type}} \tau_n}{\Delta \vdash_{\text{btype}} \tau_1 \times \cdots \times \tau_n}$$

$$\frac{\Delta \vdash_{\text{eff}} \phi \qquad \Delta, \varrho \succeq \phi \vdash_{\text{region}} \pi' \qquad \Delta, \varrho \succeq \phi \vdash_{\text{type}} \tau}{\Delta \vdash_{\text{btype}} \forall \varrho \succeq \phi .^{\pi'} \tau}$$

$\boxed{\Delta \vdash_{\text{type}} \tau}$

$$\frac{\vdash_{\text{rctxt}} \Delta}{\Delta \vdash_{\text{type}} \mathsf{Bool}} \qquad \frac{\Delta \vdash_{\text{btype}} \omega \qquad \Delta \vdash_{\text{region}} \rho}{\Delta \vdash_{\text{type}} (\omega, \rho)}$$

Figure 2.26: Static semantics of SEC (boxed types and types)

$\boxed{\vdash_{\text{prog}} e}$

$$\frac{\cdot, \mathcal{H} \succeq \{\}; \cdot \vdash_{\text{exp}} e : \mathsf{Bool}, \mathcal{H}}{\vdash_{\text{prog}} e}$$

Figure 2.27: Static semantics of SEC (programs)

Surface programs Since surface programs have a distinguished evaluation rule, we adopt the judgment $\vdash_{\text{prog}} e$ given in Figure 2.27. The rule for top-level surface programs requires that an expression evaluate to a boolean value in the context of distinguished region $\mathcal{H}$ that remains live throughout the execution of the program. It also serves as the single effect that bounds the effects of the entire program. Alternative formulations of these "boundary conditions" exist; we have adopted these to simplify the translation in Chapter 3.

Translation of BRC to SEC

We can give a straightforward translation from the Bounded Region Calculus into the Single Effect Calculus.

At the type level, this translation expands every function type into a region abstraction and function type:

$$\hat{\mathbb{T}}\left[\!\left[(\tau_1 \xrightarrow{\phi} \tau_2, \rho)\right]\!\right] = (\forall \varpi \succeq \phi.^{\rho} (\hat{\mathbb{T}}[\![\tau_1]\!] \xrightarrow{\varpi} \hat{\mathbb{T}}[\![\tau_2]\!], \rho), \rho)$$

At the term level, source functions become region abstractions and functions, and applications become region instantiations and applications. A similar approach deals with region abstractions in the source language. Essentially, this translation works by looking for the places where region sets are used in BRC and simply replacing them by an abstraction bounded by that set. Clearly, this is not the

Boxed types

$$\begin{aligned}
\hat{\mathbb{T}}\left[\!\left[\tau_1 \xrightarrow{\phi'} \tau_2\right]\!\right]_\rho &= \forall\varpi \succeq \phi'.^\rho \left(\hat{\mathbb{T}}[\![\tau_1]\!] \xrightarrow{\varpi} \hat{\mathbb{T}}[\![\tau_2]\!], \rho\right) \\
\hat{\mathbb{T}}\left[\!\left[\forall\varrho \succeq \phi.^{\phi'} \tau\right]\!\right]_\rho &= \forall\varrho \succeq \phi.^\rho \left(\forall\varpi \succeq \phi'.^\varpi \hat{\mathbb{T}}[\![\tau]\!], \rho\right)
\end{aligned}$$

Types

$$\hat{\mathbb{T}}[\![(\omega, \rho)]\!] = (\hat{\mathbb{T}}[\![\omega]\!]_\rho, \rho)$$

Expressions

$$\begin{aligned}
\hat{\mathbb{E}}\left[\!\left[\lambda x{:}\tau.^{\phi'} e\ \mathtt{at}\ \rho\right]\!\right]_\pi &= \lambda\varpi \succeq \phi'.^\rho \left(\lambda x{:}\hat{\mathbb{T}}[\![\tau]\!].^\varpi \hat{\mathbb{E}}[\![e]\!]_\varpi\ \mathtt{at}\ \rho\right) \mathtt{at}\ \rho \\
\hat{\mathbb{E}}[\![e_1\ e_2]\!]_\pi &= \hat{\mathbb{E}}[\![e_1]\!]_\pi\ [\pi]\ \hat{\mathbb{E}}[\![e_2]\!]_\pi \\
\hat{\mathbb{E}}[\![\mathtt{letregion}\ \varrho\ \mathtt{in}\ e]\!]_\pi &= \mathtt{letregion}\ \varrho\ \mathtt{in}\ \hat{\mathbb{E}}[\![e]\!]_\varrho \\
\hat{\mathbb{E}}\left[\!\left[\lambda\varrho \succeq \phi.^{\phi'} u\ \mathtt{at}\ \rho\right]\!\right]_\pi &= \lambda\varrho \succeq \phi.^\rho \left(\lambda\varpi \succeq \phi'.^\varpi \hat{\mathbb{E}}[\![u]\!]_\varpi\ \mathtt{at}\ \rho\right) \mathtt{at}\ \rho \\
\hat{\mathbb{E}}[\![e\ [\rho]]\!]_\pi &= \hat{\mathbb{E}}[\![e]\!]_\pi\ [\rho]\ [\pi]
\end{aligned}$$

Programs

$$\hat{\mathbb{E}}[\![e]\!] = \hat{\mathbb{E}}[\![e]\!]_{\mathcal{H}}$$

Figure 2.28: Translation from BRC to SEC (abbreviated)

most efficient translation. For example, in places where we could statically identify an upper bound on the region set (e.g., a singleton region set), we could elide the abstraction and simply use the upper bound.

Figure 2.28 gives an abbreviated translation from the Bounded Effect Calculus to the Single Effect Calculus (the translation is homomorphic on the other syntactic forms). The translation witnesses each introduced bounded abstraction with the current region, which is threaded through the translation by the π component of $\hat{\mathbb{E}}[\![e]\!]_\pi$. We can prove that the translation is type- and meaning-preserving.

Lemma 2.3 (Translation Preserves Types)

(1) If $\Delta; \Gamma \vdash^{\mathsf{BRC}}_{\text{exp}} e : \tau, \phi$, *then forall* Δ' *and* π, *if* $\vdash^{\mathsf{SEC}}_{\text{ctxt}} \hat{\mathbb{D}}[\![\Delta]\!], \Delta'; \hat{\mathbb{G}}[\![\Gamma]\!]; \pi$ *and* $\hat{\mathbb{D}}[\![\Delta]\!], \Delta' \vdash^{\mathsf{SEC}}_{\text{re}} \pi \succeq \phi$, *then* $\hat{\mathbb{D}}[\![\Delta]\!], \Delta'; \hat{\mathbb{G}}[\![\Gamma]\!] \vdash^{\mathsf{SEC}}_{\text{exp}} \hat{\mathbb{E}}[\![e]\!]_\pi : \hat{\mathbb{T}}[\![\tau]\!], \pi$.

(2) If $\vdash^{\mathsf{BRC}}_{\text{prog}} e$, *then* $\vdash^{\mathsf{SEC}}_{\text{prog}} \hat{\mathbb{E}}[\![e]\!]$.

Lemma 2.4 (Translation Correctness (Programs))

If $\vdash^{\mathsf{BRC}}_{\text{prog}} e$ *and* $e \Downarrow_{\text{prog}} \mathfrak{b}$ *and* $\hat{\mathbb{E}}[\![e]\!] = e^\dagger$,
then $e^\dagger \Downarrow_{\text{prog}} \mathfrak{b}$.

2.3 Summary

We have given three type-and-effect systems for a language in the spirit of the Tofte-Talpin region calculus. The Traditional Region Calculus (TRC) corresponds directly to type-and-effect systems for region calculi given in the literature [39, 10, 11]. Its defining characteristics are the form of the function type, the region abstraction type, and the type-and-effect judgment for expressions:

$$\tau_1 \xrightarrow{\phi} \tau_2 \qquad \forall \varrho.^\phi \tau \qquad \Delta; \Gamma \vdash_{\text{exp}} e : \tau, \phi$$

where the effect ϕ is a finite set of regions, which denote a (super)set of those regions allocated in or read from when the function or region abstraction is applied or when the expression is evaluated.

The Bounded Region Calculus (BRC) takes inspiration from Cyclone [30, 29] and the Calculus of Capabilities [16, 90], where the "outlives" relationship between regions is recognized as an important component of type systems for region calculi. The Bounded Region Calculus extends TRC with a form of bounded region polymorphism. Hence, the form of the function type, the region abstraction type, and the type-and-effect judgment for expressions are given as follows:

$$\tau_1 \xrightarrow{\phi} \tau_2 \qquad \forall \varrho \succeq \phi'.^{\phi}\, \tau \qquad \Delta; \Gamma \vdash_{\text{exp}} e : \tau, \phi$$

where the effect ϕ' serves as a lower bound on the lifetime of any region that instantiates ϱ. The partial order on regions is given by the LIFO stack of regions. Older regions (lower on the stack) outlive younger regions (higher on the stack). The liveness of a region implies the liveness of all regions below it on the stack.

Finally, the Single Effect Calculus (SEC) makes further use of the partial order on regions. We note that, in any finite set of regions (which are all live), there must be a youngest region, whose liveness implies the liveness of all. This youngest region can serve as a witness for the set of regions; the region appears as a single effect in place of the set. The form of the function type, the region abstraction type, and the type-and-effect judgment for expressions are given as follows:

$$\tau_1 \xrightarrow{\pi} \tau_2 \qquad \forall \varrho \succeq \phi'.^{\pi}\, \tau \qquad \Delta; \Gamma \vdash_{\text{exp}} e : \tau, \pi$$

where π denotes a single region. We will shortly see that SEC may be translated into the monadic type system of Chapter 3 and the substructural type system of Chapter 4.

Chapter 3

A Monadic Type System for Region-Based Memory Management

The region calculi of the previous chapter captured the essence of type-and-effect systems for region-based memory management. However, the type-and-effect systems that they introduce are relatively complicated, both from the perspective of a programmer (who must understand the meaning of the type-and-effect system) and from the perspective of a language designer (who must prove the soundness of the type-and-effect system). In particular, effects and their propagation appear in every typing rule, even those that do not manipulate regions. Furthermore, the typing rule for `letregion` is extremely subtle, sue to the interplay of dangling pointers and effects.

However, we can encode the (complicated) type-and-effect system of the Single Effect Calculus using nothing more than the parametric polymorphism granted by the (simple) type system of System F. This chapter demonstrates that parametric polymorphism and the technique of monadic encapsulation give rise to a simpler and more uniform language that continues to provide the power and safety of region-based memory management.

The work in this chapter was inspired by the ST monad of Launchbury and Peyton Jones [56, 55], which is used to encapsulate a "stateful" computation within a pure functional language such as Haskell. Indeed, the `runST` primitive of the ST monad turns out to be a good approximation of `letregion`: it creates a new store, allows one to allocate values in the store, and upon completion, deallocates the store and returns a value that may have dangling pointers. The `runST` primitive

can be assigned a conventional polymorphic type, which, via the parametricity of the type, ensures that dangling pointers are never dereferenced. Unfortunately, `runST` is not sufficient to encode region-based languages since there is no support for nested stores. In particular, a nested application of `runST` cannot allocate or touch data in an outer store. An extension to ST that admits a limited form of nested stores was proposed by Launchbury and Sabry [57] but, as we discuss in Section 3.1, it does not provide enough flexibility to encode the region polymorphism of the Tofte-Talpin region calculus or Cyclone.

In this chapter, we consider a monad family, called RGN, which does provide the necessary power to encode region calculi and back this claim by giving a translation from the Single Effect Calculus of the previous chapter to a monadic extension of System F, which we dub $\mathsf{F}^{\mathsf{RGN}}$. The central element of the translation is the presence of terms that witness the outlives relation and region subtyping of SEC. These terms provide the evidence needed to safely "shift" computations from one store to another. We believe that this translation sheds new light on both region calculi as well as Haskell's ST monad. In particular, it shows that the notion of region subtyping is in some sense central for supporting nested stores.

The remainder of this chapter is structured as follows. In the following section, we examine more closely why the ST monad and its variants are insufficient for encoding region-based languages. This motivates the design for $\mathsf{F}^{\mathsf{RGN}}$, which is presented more formally in Section 3.2. A key aspect of $\mathsf{F}^{\mathsf{RGN}}$ is that it adopts the type system of System F with no (significant) extensions. Encapsulation of region computations in $\mathsf{F}^{\mathsf{RGN}}$ is ensured by the type system, using parametric polymorphism. We feel that Sections 3.1 and 3.2 develop sufficient intuition to reasonably establish our goal of finding a simpler account of region-based type systems.

However, the skeptical reader may well wonder if the simplicity of the $\mathsf{F}^{\mathsf{RGN}}$ type system points to some deficiency, failing to capture all of the idioms available in type-and-effect systems for region calculi. Hence, in Section 3.3, we show how the Single Effect Calculus can be translated to $\mathsf{F}^{\mathsf{RGN}}$ in a type- and meaning-preserving fashion, thereby establishing our claim that a monadic type system is sufficient for encoding the type-and-effects systems of region calculi.

In Sections 3.4 and 3.5, we consider related work and summarize and note directions for future work. Appendix B compliments this chapter by including technical details that would otherwise detract from the focus on the translation.

3.1 Background: From ST to RGN

Launchbury and Peyton Jones [56, 55] introduced the ST monad to encapsulate stateful computations within the pure functional language Haskell. Three key insights give rise to a safe and efficient implementation of stateful computations. First, a stateful computation is represented as a *store transformer*, a description of commands to be applied to an initial store to yield a final store. Second, the store can not be duplicated, because the state type is opaque and all primitive store transformers use the store in a single-threaded manner; hence, a stateful computation can update the store in place. Third, parametric polymorphism can be used to safely encapsulate and run a stateful computation.

The types and operations associated with the ST monad are the following:

$$\tau \quad ::= \quad \ldots \mid \mathsf{ST}\ \tau_s\ \tau \mid \mathsf{STRef}\ \tau_s\ \tau$$

$$\begin{aligned} \texttt{returnST} &:: \forall s.\, \forall \alpha.\, \alpha \to \mathsf{ST}\ s\ \alpha \\ \texttt{thenST} &:: \forall s.\, \forall \alpha, \beta.\, \mathsf{ST}\ s\ \alpha \to (\alpha \to \mathsf{ST}\ s\ \beta) \to \mathsf{ST}\ s\ \beta \end{aligned}$$

$$\begin{aligned}
\texttt{newSTRef} &:: \forall s.\, \forall \alpha.\, \alpha \to \mathsf{ST}\ s\ (\mathsf{STRef}\ s\ \alpha) \\
\texttt{readSTRef} &:: \forall s.\, \forall \alpha.\, \mathsf{STRef}\ s\ \alpha \to \mathsf{ST}\ s\ \alpha \\
\texttt{writeSTRef} &:: \forall s.\, \forall \alpha.\, \mathsf{STRef}\ s\ \alpha \to \alpha \to \mathsf{ST}\ s\ () \\
\texttt{runST} &:: \forall \alpha.\, (\forall s.\, \mathsf{ST}\ s\ \alpha) \to \alpha
\end{aligned}$$

The type $\mathsf{ST}\ s\ \tau$ is the type of computations which operate on a store and deliver a value of type τ. The type s behaves like an index (or name) for the store and serves to distinguish computations operating on one store from computations operating on another store. The type $\mathsf{STRef}\ s\ \tau$ is the type of references allocated from a store indexed by s and containing a value of type τ.

The operations `returnST` and `thenST` are the *unit* and *bind* operations of the ST monad. The former yields the trivial store transformer that delivers its argument without affecting the store. The latter composes store transformers in sequence, passing the result and final store of the first computation to the second; notice that the two computations must manipulate stores indexed by the same type.

The next three operations are primitive store transformers that operate on the store. `newSTRef` takes an initial value and yields a store transformer, which, when applied to a store, allocates a fresh reference, and delivers the reference and the store augmented with the reference mapping to the initial value. Similarly, `readSTRef` and `writeSTRef` yield computations that respectively query and update the mappings of references to values in the current store. Note that all of these operations require the store index types of ST and STRef to be equal.

In this section, we will write short code examples in pseudo-Haskell syntax using the `do` notation, which provides a more conventional syntax for monadic

programming.[1] Here is a function yielding a computation that adds the contents of two references into a new reference:

$$
\begin{aligned}
&\mathit{add} :: \forall s.\, \mathsf{STRef}\ s\ \mathsf{Int} \to \mathsf{STRef}\ s\ \mathsf{Int} \to \mathsf{ST}\ s\ \mathsf{Int} \\
&\mathit{add}\ v\ w = \mathtt{do}\ a \leftarrow \mathtt{readSTRef}\ v; \\
&\qquad\qquad\qquad\quad b \leftarrow \mathtt{readSTRef}\ w; \\
&\qquad\qquad\qquad\quad \mathtt{newSTRef}\ (a + b)
\end{aligned}
$$

Finally, the operation `runST` encapsulates a stateful computation. To do so, it takes a store transformer as its argument, applies it to an initial empty store, and returns the result while discarding the final store. Note that to apply `runST`, we instantiate α with the type of the result to be returned, and then supply a store transformer, *which is polymorphic in the store index type*. The effect of this universal quantification is that the store transformer makes no assumptions about the initial store (e.g., the existence of pre-allocated references). Furthermore, the instantiation of the type variable α occurs outside the scope of the type variable s; this prevents the store transformer from delivering a value whose type mentions s. Thus, references or computations depending on the final store cannot escape beyond the encapsulation of `runST`.

All of these observations can be carried over to the region case, where we interpret stores as regions. We introduce the type $\mathsf{RGN}\ r\ \tau$ as the type of a computation which transforms a region indexed by r and delivers a value of type τ. Likewise, the type $\mathsf{RGNRef}\ r\ \tau$ is the type of (mutable) references allocated in a region indexed by r and containing a value of type τ. Each of the operations in the ST

[1] This notation allows "`do` $x \leftarrow e$; *stmts*" as shorthand for "`thenST` e $(\lambda x.\, \mathtt{do}\ \mathit{stmts})$" and "`do` e" as shorthand for "e". We use Haskell as a convenient and familiar notation, but the correspondence is somewhat weak. In particular, all of the calculi presented in this dissertation will evaluate under a call-by-value semantics.

monad has an analogue in the RGN monad:

$$
\begin{aligned}
\texttt{returnRGN} &:: \forall r.\, \forall \alpha.\, \alpha \to \mathsf{RGN}\ r\ \alpha \\
\texttt{thenRGN} &:: \forall r.\, \forall \alpha, \beta.\, \mathsf{RGN}\ r\ \alpha \to (\alpha \to \mathsf{RGN}\ r\ \beta) \to \mathsf{RGN}\ r\ \beta \\
\texttt{newRGNRef} &:: \forall r.\, \forall \alpha.\, \alpha \to \mathsf{RGN}\ r\ (\mathsf{RGNRef}\ r\ \alpha) \\
\texttt{readRGNRef} &:: \forall r.\, \forall \alpha.\, \mathsf{RGNRef}\ r\ \alpha \to \mathsf{RGN}\ r\ \alpha \\
\texttt{writeRGNRef} &:: \forall r.\, \forall \alpha.\, \mathsf{RGNRef}\ r\ \alpha \to \alpha \to \mathsf{RGN}\ r\ \alpha \\
\texttt{runRGN} &:: \forall \alpha.\, (\forall r.\, \mathsf{RGN}\ r\ \alpha) \to \alpha
\end{aligned}
$$

Does this suffice to encode region-based languages, where `runRGN` corresponds to `letregion`? In short, it does not. In a region-based language, it is critical to allocate locations in and read locations from an outer region while in the scope of an inner region. For example, an essential idiom in region-based languages is to enter a `letregion` in which temporary data is allocated, while reading input from and allocating output in an outer region; upon leaving the `letregion`, the temporary data is reclaimed, but the input and output data are still available.

Unfortunately, this idiom cannot be accommodated in the framework presented thus far. For example, consider this canonical example of region-based memory management usage:

$$
\begin{aligned}
&\texttt{letregion}\ \rho_1\ \texttt{in} \\
&\quad \texttt{let}\ a = 1\,\texttt{at}\,\rho_1\ \texttt{in} \\
&\quad \texttt{let}\ c = \texttt{letregion}\ \rho_2\ \texttt{in} \\
&\qquad\qquad \texttt{let}\ b = 7\,\texttt{at}\,\rho_2\ \texttt{in} \\
&\qquad\qquad \texttt{let}\ z = a + b\,\texttt{at}\,\rho_1 \\
&\qquad\qquad \texttt{in}\ z \\
&\quad \texttt{in}\ \ldots\ c\ \ldots
\end{aligned}
$$

where we think of a as an input, b as a temporary, and c as an output. A naïve

translation fails to type-check:

$$
\begin{array}{l}
\texttt{runRGN}\ (\Lambda r_1. \\
\quad \texttt{do}\ a \leftarrow \texttt{newRGNRef}\ [r_1]\ 1; \\
\qquad c \leftarrow \texttt{runRGN}\ (\Lambda r_2. \\
\qquad\quad \texttt{do}\ b \leftarrow \texttt{newRGNRef}\ [r_2]\ 7; \\
\qquad\qquad z \leftarrow a + b; \\
\qquad\qquad \texttt{newRGNRef}\ [r_1]\ z); \\
\qquad \ldots\ c\ \ldots)
\end{array}
$$

The error arises from the fact that allocating a temporary in the younger region (`newRGNRef` $[r_2]$ 7) yields a computation of type $\mathsf{RGN}\ r_2\ (\mathsf{RGNRef}\ r_2\ \mathsf{Int})$, while allocating the result in the older region (`newRGNRef` $[r_1]$ z) yields a computation of type $\mathsf{RGN}\ r_1\ (\mathsf{RGNRef}\ r_1\ \mathsf{Int})$. These computations cannot be sequenced, since their region indices differ.

Launchbury and Sabry [57] argue that the principle behind `runST` can be generalized to provide nested scope. They introduce two additional operations

$$\texttt{blockST} \ :: \ \forall s.\, \forall \alpha.\, (\forall r.\, \mathsf{ST}\ (s \times r)\ \alpha) \rightarrow \mathsf{ST}\ s\ \alpha$$

$$\texttt{importSTRef} \ :: \ \forall s, r.\, \forall \alpha.\, \mathsf{STRef}\ s\ \alpha \rightarrow \mathsf{STRef}\ (s \times r)\ \alpha$$

where `blockST` encapsulates a nested scope and `importSTRef` explicitly allows references from an enclosing scope to be manipulated by the inner scope. Similarly, Peyton Jones[2] suggests introducing the constant

$$\texttt{liftST} \ :: \ \forall s, r.\, \forall \alpha.\, \mathsf{ST}\ s\ \alpha \rightarrow \mathsf{ST}\ (s \times r)\ \alpha$$

in lieu of `importSTRef`, with the same intention of importing computations from an outer scope into the inner scope. In essence, `liftST` encodes the stack of stores using a tuple type for the index of the ST monad.

[2]private communication

Should we adopt `blockST` and `liftST` in the RGN monad as `letRGN` and `liftRGN`? At first glance, doing so would appear to provide sufficient expressiveness to encode region-based languages. We can "fix" our previous translation as follows:

```
runRGN (Λr1.
  do a ← newRGNRef [r1] 1;
     c ← runRGN (Λr2.
           do b ← newRGNRef [r2] 7;
              z ← a + b;
              liftRGN [r1] [r2] (newRGNRef [r1] z));
     ... c ...)
```

However, another critical aspect of region-based languages is region polymorphism. For example, consider a generalization of the *add* function above, where each of the two input references is allocated in a different region, the output reference is to be allocated in a third region, and the result computation is to be indexed by a fourth region; such a function would have the type:

```
gadd :: ∀r1, r2, r3, r4.
          RGNRef r1 Int →
          RGNRef r2 Int →
          RGN r4 (RGNRef r3 Int)
```

However, there is no way to write *gadd* with liftRGN terms alone that will result in sufficient polymorphism over regions. For example, if we write

```
gadd v w = liftRGN (do a ← readRGNRef v;
                       b ← liftRGN (readRGNRef w);
                       liftRGN (liftRGN (newRGNRef (a + b))))
```

then we produce a function with the type:

$$
\begin{aligned}
\mathit{gadd} :: \forall r_1, r_2, r_3, r_4.\ & \\
& \mathsf{RGNRef}\ ((r_1 \times r_2) \times r_3)\ \mathsf{Int} \rightarrow \\
& \mathsf{RGNRef}\ (r_1 \times r_2)\ \mathsf{Int} \rightarrow \\
& \mathsf{RGN}\ (((r_1 \times r_2) \times r_3) \times r_4)\ (\mathsf{RGNRef}\ r_1\ \mathsf{Int})
\end{aligned}
$$

The problem is that the explicit connection between the outer and inner regions in the product type enforces a total order on regions, which leaks into the types of region allocated values. The function above *only* works when the four regions are consecutive and the output reference is allocated in the outermost region, the input references are allocated in the next two regions, and the computation is indexed by the innermost region.

However, the order of the regions should not matter. The only requirement is that if the final computation (indexed by r_4) is ever run, then each of the regions r_1, r_2, and r_3 must be live. To put it another way, the three regions are older than (i.e., subtypes of) region r_4. Hence, we adopt a simple solution, one that enables the translation given in Section 3.3, whereby we abstract the `liftRGN` applications and pass evidence that witnesses the region subtyping.

$$
\begin{aligned}
\mathit{gadd} :: \forall r_1, r_2, r_3, r_4.\ & \\
& (\forall \beta.\, \mathsf{RGN}\ r_1\ \beta \rightarrow \mathsf{RGN}\ r_4\ \beta) \rightarrow \\
& (\forall \beta.\, \mathsf{RGN}\ r_2\ \beta \rightarrow \mathsf{RGN}\ r_4\ \beta) \rightarrow \\
& (\forall \beta.\, \mathsf{RGN}\ r_3\ \beta \rightarrow \mathsf{RGN}\ r_4\ \beta) \rightarrow \\
& \mathsf{RGNRef}\ r_1\ \mathsf{Int} \rightarrow \mathsf{RGNRef}\ r_2\ \mathsf{Int} \rightarrow \mathsf{RGN}\ r_4\ (\mathsf{RGNRef}\ r_3\ \mathsf{Int}) \\
\mathit{gadd}\ \mathit{ev}_1\ \mathit{ev}_2\ \mathit{ev}_3\ v\ w = \mathtt{do}\ & a \leftarrow \mathit{ev}_1\ (\mathtt{readRGNRef}\ v); \\
& b \leftarrow \mathit{ev}_2\ (\mathtt{readRGNRef}\ w); \\
& \mathit{ev}_3\ (\mathtt{newRGNRef}\ (a + b))
\end{aligned}
$$

While this evidence can be assembled from `liftRGN` terms, we find that the key notion is subtyping on regions and evidence that witnesses the subtyping. The product type used in `blockST` is one way of connecting the outer and inner stores, but all the "magic" happens with `liftST`. Therefore, we adopt an approach that fuses the two operations together in the `letRGN` operation:

$$\begin{aligned} \mathsf{RGNPf}(r_1 \preceq r_2) &\equiv \forall\beta.\, \mathsf{RGN}\ r_1\ \beta \to \mathsf{RGN}\ r_2\ \beta \\ \texttt{letRGN} &:: \forall r_1.\, \forall\alpha.\, (\forall r_2.\, \mathsf{RGNPf}(r_1 \preceq r_2) \to \mathsf{RGN}\ r_2\ \alpha) \to \mathsf{RGN}\ r_1\ \alpha \end{aligned}$$

The function argument to `letRGN` is given evidence, of type $\mathsf{RGNPf}(r_1 \preceq r_2)$, that the outer (older) region (denoted by r_1) is a subtype of the inner (younger) region (denoted by r_2), which it can use in the region computation. The same parametricity argument that applied to `runST` applies here: references and computations from the inner region cannot escape beyond the encapsulation of `letRGN`. We no longer need a product type connecting the outer and inner regions, as this relationship is given by the witness function.

3.2 The $\mathsf{F}^{\mathsf{RGN}}$ Language

The $\mathsf{F}^{\mathsf{RGN}}$ language is an extension of System F [68, 28] (also referred to as the polymorphic lambda calculus), adding the types and operations from the RGN monad. As described in the previous section, the design of $\mathsf{F}^{\mathsf{RGN}}$ takes inspiration from the work on monadic state [56, 55, 57, 4, 72, 63]. Essentially, $\mathsf{F}^{\mathsf{RGN}}$ uses an explicit region monad to enforce the locality of region allocated values.

In this section, we present the $\mathsf{F}^{\mathsf{RGN}}$ language in sufficient detail to describe the translation from the Single Effect Calculus to $\mathsf{F}^{\mathsf{RGN}}$ in Section 3.3. To this end, we include the syntax for both the surface language and abstract machine

configurations, dynamic semantics for the abstract machine configurations, and static semantics for the surface language.

The dynamic semantics defines a large-step (or natural) semantics, which defines an *evaluation relation* from *towers of stacks of regions* and *expressions* to *values*. Our main reason for adopting a large-step operational semantics is to simplify the theorems and proofs of Section 3.3 and Appendix B.3; establishing the correctness of the translation would be more difficult using small-step operational semantics, due to differing numbers of intermediate small-steps.

We purposefully omit the static semantics for the abstract machine configurations (normally included for a syntactic proof of type soundness), since it requires a number of technical details that detract from the focus on the translation. Appendix B.1 includes additional technical details for the $\mathsf{F}^{\mathsf{RGN}}$ language and (sketches) a syntactic proof of type soundness.

3.2.1 Syntax of $\mathsf{F}^{\mathsf{RGN}}$

Surface Syntax of $\mathsf{F}^{\mathsf{RGN}}$

Figures 3.1 and 3.2 present the syntax of "surface programs" (that is, excluding syntax and semantic objects that will appear in the dynamic semantics) of $\mathsf{F}^{\mathsf{RGN}}$. In the following sections, we explain and motivate the main constructs of $\mathsf{F}^{\mathsf{RGN}}$.

Types and indices Types in $\mathsf{F}^{\mathsf{RGN}}$ include those found in System F (function and product types and type abstractions) along with the primitive types Int and Bool. The RGN θ τ, RGNRef θ τ, and RGNPf$(\theta_1 \preceq \theta_2)$ types were introduced in the previous section. We introduce RGN indices θ as a distinguished syntactic object, distinct from types τ, since doing so simplifies the presentation and does

Type variables

$\alpha \quad \in \quad TVars$

Surface types

$\tau \quad ::= \quad \mathsf{Int} \mid \mathsf{Bool} \mid \tau_1 \to \tau_2 \mid \tau_1 \times \cdots \times \tau_n \mid \alpha \mid \forall\alpha.\,\tau \mid$

$\qquad\qquad \mathsf{RGN}\ \theta\ \tau \mid \mathsf{RGNRef}\ \theta\ \tau \mid \mathsf{RGNHnd}\ \theta \mid \mathsf{RGNPf}(\theta_1 \preceq \theta_2) \mid \forall\vartheta.\,\tau$

RGN index variables

$\vartheta \quad \in \quad IVars$

Surface RGN indices

$\theta \quad ::= \quad \vartheta$

Figure 3.1: Surface syntax of $\mathsf{F}^{\mathsf{RGN}}$ (I)

not unnecessarily complicate the type system. [3] Note that surface programs never require a region index to be represented by anything other than an index variable. RGN index polymorphism is available through the index abstraction type $\forall\vartheta.\,\tau$.[4] Finally, we add the type RGNHnd θ as the type of handles for the region indexed by θ. A value of this type is a *region handle* – a run-time value holding the data necessary to allocate values within a region. RGN indices (static objects) and region handles (dynamic objects) are distinguished in order to maintain a phase distinction between compile-time and run-time expressions and to more accurately reflect implementations of regions. Recall that in the region calculi of Chapter 2,

[3] The choice of "θ" as the meta-variable for a region index, as opposed to "ρ", is motivated by the fact that the index identifies both the stack and region in which a monadic region computation is executing, rather than just the region.

[4] There are a variety of other ways to handle both types and indices. We could introduce two kinds, say Type and Index, and collapse the syntactic classes of types and indices. Noting that surface programs do not admit indices that are not region indices, one can simply represent a Index variable as a Type variable, and eliminate the Index kind. This has the advantage that the type system can be encoded in standard System F (e.g., in Haskell).

Integer constants

$$\mathfrak{i} \quad \in \quad \mathbb{Z}$$

Boolean constants

$$\mathfrak{b} \quad \in \quad \{\texttt{true}, \texttt{false}\}$$

Value variables

$$f, x \quad \in \quad \mathit{VVars}$$

Surface terms

$$\begin{aligned} e \quad ::= \quad & \mathfrak{i} \mid e_1 \oplus e_2 \mid e_1 \oslash e_3 \mid \mathfrak{b} \mid \texttt{if}\ e_b\ \texttt{then}\ e_t\ \texttt{else}\ e_f \mid \\ & x \mid \lambda x{:}\tau.\, e \mid e_1\ e_2 \mid \langle e_1, \ldots, e_n \rangle \mid \texttt{sel}_i\ e \mid \Lambda\alpha.\, e \mid e\ [\tau] \mid \\ & \texttt{let}\ x = e_a\ \texttt{in}\ e_b \mid \texttt{runRGN}\ [\tau]\ v \mid \kappa \mid \Lambda\vartheta.\, e \mid e\ [\theta] \end{aligned}$$

Surface RGN commands

$$\begin{aligned} \kappa \quad ::= \quad & \texttt{returnRGN}\ [\theta]\ [\tau]\ v \mid \texttt{thenRGN}\ [\theta]\ [\tau_a]\ [\tau_b]\ v_a\ v_f \mid \\ & \texttt{letRGN}\ [\theta]\ [\tau]\ v \mid \texttt{newRGNRef}\ [\theta]\ [\tau_a]\ v_h\ v_\star \mid \\ & \texttt{readRGNRef}\ [\theta]\ [\tau_a]\ v_r \mid \texttt{writeRGNRef}\ [\theta]\ [\tau_a]\ v_r\ v_\star \end{aligned}$$

Surface values

$$v \quad ::= \quad \mathfrak{i} \mid \mathfrak{b} \mid x \mid \lambda x{:}\tau.\, e \mid \langle v_1, \ldots, v_n \rangle \mid \Lambda\alpha.\, e \mid \kappa \mid \Lambda\vartheta.\, e$$

Figure 3.2: Surface syntax of $\mathsf{F}^{\mathsf{RGN}}$ (II)

the introduction forms for region allocated values carried a region annotation $\texttt{at}\,\rho$, which indicated the region in which the value is to be allocated, while the expression form $\texttt{ref}\ \mathfrak{r}\ \mathfrak{p}$ was the (live) pointer associated with a region allocated value. A region handle is required to "name" the region into which a value is to be allocated; on the other hand, reading through a pointer to a region allocated value does not require a handle, since the pointer itself "names" the region. This ensures that indices, like types, have no run-time significance and may be erased from compiled code. On the other hand, region handles are necessary at run-time to allocate values within a region.

Terms As with types, $\mathsf{F}^{\mathsf{RGN}}$ adopts terms found in System F; constants, arithmetic and boolean operations, function abstraction and application, tuple introduction and elimination, and type abstraction and instantiation are all standard.

We let κ range over the syntactic class of RGN monad commands. (Equivalently, and as suggested by the explicit type annotations and the restriction of subexpressions to values, we can consider the monadic commands as constants with polymorphic types in a call-by-value interpretation of $\mathsf{F}^{\mathsf{RGN}}$. Presenting monadic commands in this fashion avoids intermediate terms in the operational semantics corresponding to partial application.) Each of the commands has been described previously. Finally, we include RGN index abstraction and instantiation, analogous to type abstraction and instantiation.

Abstract Machine Configurations for $\mathsf{F}^{\mathsf{RGN}}$

Figures 3.3 and 3.4 present abstract machine configurations for $\mathsf{F}^{\mathsf{RGN}}$, which extend the syntax of the previous section with semantic objects that appear in the operational semantics.

Stack names

$\mathfrak{s} \quad \in \quad \mathit{SNames}$

Constant stacks

$s \quad ::= \quad \mathfrak{s} \mid \circ$

Region names

$\mathfrak{r} \quad \in \quad \mathit{RNames}$

Constant regions

$r \quad ::= \quad \mathfrak{r} \mid \bullet$

Pointer names

$\mathfrak{p} \quad \in \quad \mathit{PNames}$

Abstract machine RGN indices

$\theta \quad ::= \quad \ldots \mid s\sharp r$

Abstract machine terms

$e \quad ::= \quad \ldots \mid \mathtt{ref}\ s\sharp r\ \mathfrak{p} \mid \mathtt{hnd}\ s\sharp r$

Abstract machine RGN commands

$\kappa \quad ::= \quad \ldots \mid \mathtt{witnessRGN}\ s\sharp r_1\ s\sharp r_2\ [\tau]\ v_\kappa$

Abstract machine values

$v \quad ::= \quad \ldots \mid \mathtt{ref}\ s\sharp r\ \mathfrak{p} \mid \mathtt{hnd}\ s\sharp r$

Figure 3.3: Abstract machine syntax of $\mathsf{F}^{\mathsf{RGN}}$ (I)

Regions

$$R \quad ::= \quad \{\mathfrak{p}_1 \mapsto v_1, \ldots, \mathfrak{p}_n \mapsto v_n\}$$

Stacks

$$S \quad ::= \quad \cdot \mid S, \mathfrak{r} \mapsto R \quad \text{(ordered domain)}$$

Towers

$$T \quad ::= \quad \cdot \mid T, \mathfrak{s} \mapsto S \quad \text{(ordered domain)}$$

Abstract machine configurations

$$(T; e)$$

Figure 3.4: Abstract machine syntax of $\mathsf{F}^{\mathsf{RGN}}$ (II)

Stack names, region names, and pointers are used to represent references to region allocated data. Because `runRGN` computations can be nested, we need a means to distinguish data allocated in regions from different `runRGN` computations; stack names serve this purpose. Each `runRGN` computation is associated with a unique stack, which collects and identifies all regions belonging to that computation. Stack and region constants distinguish between live and dead stacks and regions; a dead stack ($\circ$) or region ($\bullet$) corresponds to a deallocated stack or region.

The abstract machine syntax adds one new region index form and two new expression forms. The index $s\sharp r$ is the instantiated form of a region index variable (hence, $\circ\sharp\bullet$ corresponds to a dead region in a dead stack). Such an index identifies the stack and region in which a monadic region computation is executing. The expression `ref` $s\sharp r$ $\mathfrak{p}$ is the run-time representation of a $\mathsf{RGNRef}\ s\sharp r\ \tau$; that is, it is the pointer reference associated with a region allocated value. Likewise, the expression `hnd` $s\sharp r$ is the run-time representation of a $\mathsf{RGNHnd}\ s\sharp r$.

The abstract machine syntax also adds a new command form. The command $\texttt{witnessRGN}\ s\sharp r_1\ s\sharp r_2\ [\tau]\ v_\kappa$ casts a computation from the type $\mathsf{RGN}\ s\sharp r_1\ \tau$ to the type $\mathsf{RGN}\ s\sharp r_2\ \tau$. (This command is used to construct terms of the type $\mathsf{RGNPf}(\theta_1 \preceq \theta_2) \equiv \forall\beta.\,\mathsf{RGN}\ \theta_1\ \beta \to \mathsf{RGN}\ \theta_2\ \beta$ introduced in Section 3.1.) Operationally, such a command is the identity function, so long as the cast is valid. The static semantics of the next section and Appendix B.1.2 ensure that all such casts in a well-typed program are valid.

Thus far, we have talked about region allocated data without discussing where such data is stored. Storable (i.e., closed) values are associated with pointers in regions R; regions are ordered into stacks S; finally, stacks are ordered into towers T. We use the notation $S(\mathfrak{r}, \mathfrak{p})$ and $T(\mathfrak{s}, \mathfrak{r}, \mathfrak{p})$ for iterated lookups of values in stacks and towers, respectively. Again, towers are a technical device that serve to distinguish nested `runRGN` computations from one another. Intuitively, executing a `runRGN` computation adds a new stack to the top of the tower (the new stack is deallocated upon completing the computation), while executing a `letRGN` command adds a new region to the top of the topmost stack (the new region is deallocated upon completing the nested computation). These intuitions are formalized in the dynamic semantics of the next section.

3.2.2 Dynamic Semantics of $\mathsf{F}^{\mathsf{RGN}}$

Two mutually inductive judgments (for pure expressions (Figures 3.5, 3.6, and 3.7) and for monadic commands (Figure 3.8)) define the dynamic semantics. We state without proof that the dynamic semantics is deterministic; it is syntax-directed, taking $(T; e)$ configurations modulo α-conversion, including conversion of stack names, region names, and pointers, which are (uniquely) bound in the tower T.

$\boxed{(T; e) \Downarrow v}$

$$\frac{}{(T; \mathfrak{i}) \Downarrow \mathfrak{i}} \qquad \frac{(T; e_1) \Downarrow v_1 \qquad v_1 \equiv \mathfrak{i}_1 \qquad (T; e_2) \Downarrow v_2 \qquad v_2 \equiv \mathfrak{i}_2 \qquad \mathfrak{i}_1 \oplus \mathfrak{i}_2 = \mathfrak{i}}{(T; e_1 \oplus e_2) \Downarrow i}$$

$$\frac{(T; e_1) \Downarrow v_1 \qquad v_1 \equiv \mathfrak{i}_1 \qquad (T; e_2) \Downarrow v_2 \qquad v_2 \equiv \mathfrak{i}_2 \qquad \mathfrak{i}_1 \otimes \mathfrak{i}_2 = \mathfrak{b}}{(T; e_1 \otimes e_2) \Downarrow \mathfrak{b}} \qquad \frac{}{(T; \mathfrak{b}) \Downarrow \mathfrak{b}}$$

$$\frac{(T; e_b) \Downarrow v_b \qquad v_b \equiv \texttt{true} \qquad (T; e_t) \Downarrow v}{(T; \texttt{if } e_b \texttt{ then } e_t \texttt{ else } e_f) \Downarrow v}$$

$$\frac{(T; e_b) \Downarrow v_b \qquad v_b \equiv \texttt{false} \qquad (T; e_f) \Downarrow v}{(T; \texttt{if } e_b \texttt{ then } e_t \texttt{else } e_f) \Downarrow v}$$

Figure 3.5: Dynamic semantics of $\mathsf{F}^{\mathsf{RGN}}$ (expressions (I))

$\boxed{(T; e) \Downarrow v}$

$$\frac{}{(T; \lambda x{:}\tau.\, e) \Downarrow \lambda x{:}\tau.\, e}$$

$$\frac{(T; e_f) \Downarrow v_f \qquad v_f \equiv \lambda x{:}\tau_x.\, e_b \qquad (T; e_a) \Downarrow v_a \qquad (T; e_b[v_a/x]) \Downarrow v}{(T; e_f\ e_a) \Downarrow v}$$

$$\frac{(T; e_1) \Downarrow v_1 \qquad \ldots \qquad (T; e_n) \Downarrow v_n}{(T; \langle e_1, \ldots, e_n \rangle) \Downarrow \langle v_1, \ldots, v_n \rangle}$$

$$\frac{(T; e) \Downarrow v \qquad v \equiv \langle v_1, \ldots, v_n \rangle \qquad 1 \leq i \leq n}{(T; \mathtt{sel}_i\ e) \Downarrow v_i} \qquad \frac{}{(T; \Lambda\alpha.\, e) \Downarrow \Lambda\alpha.\, e}$$

$$\frac{(T; e_f) \Downarrow v_f \qquad v_f \equiv \Lambda\alpha.\, e_b \qquad (T; e_b[\tau_a/\alpha]) \Downarrow v}{(T; e_f\ [\tau_a]) \Downarrow v}$$

$$\frac{(T; e_a) \Downarrow v_a \qquad (T; e_b[v_a/x]) \Downarrow v}{(T; \mathtt{let}\ x = e_a\ \mathtt{in}\ e_b) \Downarrow v}$$

Figure 3.6: Dynamic semantics of $\mathsf{F}^{\mathsf{RGN}}$ (expressions (II))

$\boxed{(T; e) \Downarrow v}$

$$\frac{\mathfrak{s} \notin dom(T) \qquad \mathfrak{r} \notin dom(\cdot) \qquad (T, \mathfrak{s} \mapsto (\cdot, \mathfrak{r} \mapsto \{\}); v\ [\mathfrak{s}\sharp\mathfrak{r}]\ (\texttt{hnd}\ \mathfrak{s}\sharp\mathfrak{r})) \Downarrow v' \qquad v' \equiv \kappa' \qquad (T, \mathfrak{s} \mapsto (\cdot, \mathfrak{r} \mapsto \{\}); \kappa') \Downarrow_\kappa S''; v'' \qquad S'' \equiv \cdot, \mathfrak{r} \mapsto R''}{(T; \texttt{runRGN}\ [\tau]\ v) \Downarrow v''[\circ\sharp\bullet/\mathfrak{s}\sharp\mathfrak{r}]} \qquad \frac{}{(T; \kappa) \Downarrow \kappa}$$

$$\frac{}{(T; \Lambda\vartheta.\, e) \Downarrow \Lambda\vartheta.\, e} \qquad \frac{(T; e_f) \Downarrow v_f \qquad v_f \equiv \Lambda\vartheta.\, e_b \qquad (T; e_b[\theta_a/\vartheta]) \Downarrow v}{(T; e_f\ [\theta_a]) \Downarrow v}$$

$$\frac{}{(T; \texttt{ref}\ s\sharp r\ \mathfrak{p}) \Downarrow \texttt{ref}\ s\sharp r\ \mathfrak{p}} \qquad \frac{}{(T; \texttt{hnd}\ s\sharp r) \Downarrow \texttt{hnd}\ s\sharp r}$$

Figure 3.7: Dynamic semantics of $\mathsf{F}^{\mathsf{RGN}}$ (expressions (III))

We use the notation $S(\mathfrak{r})$ for the lookup of regions in stacks and the notation $S(\mathfrak{r}, \mathfrak{p})$ for the iterated lookup of storable values in stacks. These are partial functions, defined as follows:

$$\begin{array}{rcll} S(\mathfrak{r}) & = & \mathsf{undefined} & \text{if } \mathfrak{r} \notin dom(S) \\ S(\mathfrak{r}) & = & R & \text{if } \mathfrak{r} \in dom(S) \text{ and } S \equiv \ldots, \mathfrak{r} \mapsto R, \ldots \end{array}$$

$$\begin{array}{rcll} S(\mathfrak{r}, \mathfrak{p}) & = & \mathsf{undefined} & \text{if } S(\mathfrak{r}) = \mathsf{undefined} \\ S(\mathfrak{r}, \mathfrak{p}) & = & \mathsf{undefined} & \text{if } S(\mathfrak{r}) = R \text{ and } \mathfrak{p} \notin dom(R) \\ S(\mathfrak{r}, \mathfrak{p}) & = & R & \text{if } S(\mathfrak{r}) = R \text{ and } \mathfrak{p} \in dom(R) \text{ and } R \equiv \{\ldots, \mathfrak{p} \mapsto w, \ldots\} \end{array}$$

We also use the notation $S\{(\mathfrak{r}, \mathfrak{p}) \mapsto v\}$ to denote the stack S' which extends the stack S with a mapping from $\mathfrak{p}$ to v in the region $S(\mathfrak{r})$. This function is defined when $\mathfrak{r} \in dom(S)$ and $S(\mathfrak{r}) = R$ and $\mathfrak{p} \notin dom(R)$.

$\boxed{(T, \mathfrak{s} \mapsto S; \kappa) \Downarrow_\kappa (S'; v)}$

$$\frac{\theta \equiv \mathfrak{s}\sharp\mathfrak{r}}{(T, \mathfrak{s} \mapsto S; \texttt{returnRGN}\ [\theta]\ [\tau]\ v) \Downarrow_\kappa (S; v)}$$

$$\frac{\theta \equiv \mathfrak{s}\sharp\mathfrak{r} \qquad v_a \equiv \kappa_a \qquad (T, \mathfrak{s} \mapsto S; \kappa_a) \Downarrow_\kappa (S'; v'_a) \qquad (T, \mathfrak{s} \mapsto S'; v_f\ v'_a) \Downarrow v'' \qquad v'' \equiv \kappa'' \qquad (T, \mathfrak{s} \mapsto S'; \kappa'') \Downarrow_\kappa (S'''; v''')}{(T, \mathfrak{s} \mapsto S; \texttt{thenRGN}\ [\theta]\ [\tau_a]\ [\tau_b]\ v_a\ v_f) \Downarrow_\kappa (S'''; v''')}$$

$$\frac{\begin{array}{c}\theta \equiv \mathfrak{s}\sharp\mathfrak{r}_1 \\ \mathfrak{r}_1 \in dom(S) \qquad \mathfrak{r}_2 \notin dom(S) \qquad (T, \mathfrak{s} \mapsto (S, \mathfrak{r}_2 \mapsto \{\}); v\ [\mathfrak{s}\sharp\mathfrak{r}_2]\ w\ (\texttt{hnd}\ \mathfrak{s}\sharp\mathfrak{r}_2)) \Downarrow v' \\ v' \equiv \kappa' \qquad (T, \mathfrak{s} \mapsto (S, \mathfrak{r}_2 \mapsto \{\}); \kappa') \Downarrow_\kappa (S'''; v'') \qquad S''' \equiv S'', \mathfrak{r}_2 \mapsto R''_2\end{array}}{(T, \mathfrak{s} \mapsto S; \texttt{letRGN}\ [\theta]\ [\tau]\ v) \Downarrow_\kappa (S''[\mathfrak{s}\sharp\bullet/\mathfrak{s}\sharp\mathfrak{r}_2]; v''[\mathfrak{s}\sharp\bullet/\mathfrak{s}\sharp\mathfrak{r}_2])}$$

where $w = (\Lambda\beta.\, \lambda k{:}\mathsf{RGN}\ \mathfrak{s}\sharp\mathfrak{r}_1\ \beta.\, \texttt{witnessRGN}\ \mathfrak{s}\sharp\mathfrak{r}_1\ \mathfrak{s}\sharp\mathfrak{r}_2\ [\beta]\ k)$

$$\frac{\begin{array}{c}s\sharp r_1 \equiv \mathfrak{s}\sharp\mathfrak{r}_1 \\ s\sharp r_2 \equiv \mathfrak{s}\sharp\mathfrak{r}_2 \qquad v \equiv \kappa \qquad S \equiv S_1, \mathfrak{r}_1 \mapsto R_1, S_2, \mathfrak{r}_2 \mapsto R_2, S_3 \qquad (T, \mathfrak{s} \mapsto S; \kappa) \Downarrow_\kappa (S'; v')\end{array}}{(T, \mathfrak{s} \mapsto S; \texttt{witnessRGN}\ s\sharp r_1\ s\sharp r_2\ [\tau]\ v) \Downarrow_\kappa (S'; v')}$$

$$\frac{\theta \equiv \mathfrak{s}\sharp\mathfrak{r} \qquad v_h \equiv \texttt{hnd}\ \mathfrak{s}\sharp\mathfrak{r} \qquad \mathfrak{r} \in dom(S) \qquad \mathfrak{p} \notin dom(S(\mathfrak{r}))}{(T, \mathfrak{s} \mapsto S; \texttt{newRGNRef}\ [\theta]\ [\tau]\ v_h\ v_\star) \Downarrow_\kappa (S\{(\mathfrak{r}, \mathfrak{p}) \mapsto v_\star\}; \texttt{ref}\ \mathfrak{s}\sharp\mathfrak{r}\ \mathfrak{p})}$$

$$\frac{\theta \equiv \mathfrak{s}\sharp\mathfrak{r} \qquad v_r \equiv \texttt{ref}\ \mathfrak{s}\sharp\mathfrak{r}\ \mathfrak{p} \qquad \mathfrak{r} \in dom(S) \qquad \mathfrak{p} \in dom(S(\mathfrak{r})) \qquad S(\mathfrak{r}, \mathfrak{p}) = v}{(T, \mathfrak{s} \mapsto S; \texttt{readRGNRef}\ [\theta]\ [\tau]\ v_r) \Downarrow_\kappa (S; v')}$$

$$\frac{\theta \equiv \mathfrak{s}\sharp\mathfrak{r} \qquad v_r \equiv \texttt{ref}\ \mathfrak{s}\sharp\mathfrak{r}\ \mathfrak{p} \qquad \mathfrak{r} \in dom(S) \qquad \mathfrak{p} \in dom(S(\mathfrak{r}))}{(T, \mathfrak{s} \mapsto S; \texttt{writeRGNRef}\ [\theta]\ [\tau]\ v_r\ v_\star) \Downarrow_\kappa (S\{(\mathfrak{r}, \mathfrak{p}) \mapsto v_\star\}; \langle\rangle)}$$

Figure 3.8: Dynamic semantics of $\mathsf{F}^{\mathsf{RGN}}$ (commands)

The judgment $(T; e) \Downarrow v$ asserts that evaluating the closed expression e in tower T results in a value v. Likewise, the judgment $(T, \mathfrak{s} \mapsto S; \kappa) \Downarrow_\kappa (S'; v)$ asserts that evaluating the closed monadic command κ in a non-empty tower whose top stack is S results in a new top stack S' and a value v. Note that the existence of towers of stacks of regions (to accommodate nested `runRGN` commands) and the division of evaluation into pure expressions and monadic commands precludes re-using the abstract machine and operational semantics from Chapter 2.

The rules for $(T; e) \Downarrow v$ for expression forms other than `runRGN` are completely standard. The tower T is passed unchanged to sub-evaluations. The rule for `runRGN` $[\tau]$ v runs a monadic computation and executes in the following manner. First, fresh stack and region names $\mathfrak{s}$ and $\mathfrak{r}$ are chosen. Next, the argument v is applied to the region index $\mathfrak{s}\sharp\mathfrak{r}$ and the region handle `hnd` $\mathfrak{s}\sharp\mathfrak{r}$ and evaluated in the extended tower $T, \mathfrak{s} \mapsto (\cdot, \mathfrak{r} \mapsto \{\})$ (that is, the tower T extended with a stack (bound to $\mathfrak{s}$) consisting of a single empty region (bound to $\mathfrak{r}$)) to a monadic command κ'. This command is evaluated under the extended tower to a modified stack (of the form $\cdot, \mathfrak{r} \mapsto R''$) and a value v''. The modified stack is discarded, while occurrences of $\mathfrak{s}\sharp\mathfrak{r}$ are replaced by $\circ\sharp\bullet$ in v''; this replacement ensures that any occurrences of `ref` or `hnd` terms in v'' are marked as dead, since the stack and region have been deallocated and are no longer accessible.

The rules for $(T, \mathfrak{s} \mapsto S; \kappa) \Downarrow_\kappa (S'; v)$ perform monadic operations that side-effect the top stack in the tower. The monadic unit and bind operations are standard; in particular, note the manner in which the rule for `thenRGN` threads the modified top stack through the computation.

The rule for `letRGN` $[\mathfrak{s}\sharp\mathfrak{r}_1]$ τ v executes in much the same way as the rule for `runRGN`. First, a fresh region name $\mathfrak{r}_2$ is chosen. Next, the argument v is applied

to the region index $\mathfrak{s}\sharp\mathfrak{r}_2$, a witness function, and the region handle $\texttt{hnd}\ \mathfrak{s}\sharp\mathfrak{r}_2$ and evaluated under an extended tower that adds an empty region (bound to $\mathfrak{r}_2$) to the top of the stack. This evaluation yields a monadic command κ', which is also evaluated under the extended tower to a modified top stack and value v''. The modified top region (still bound to $\mathfrak{r}$) is discarded, while occurrences of $\mathfrak{s}\sharp\mathfrak{r}_2$ are replaced by $\mathfrak{s}\sharp\bullet$ in the modified top stack S'' and value v''; again, this replacement ensures that any occurrences of `ref` or `hnd` terms in S'' or v'' are marked as dead.

The rule for `witnessRGN` permits a monadic computation to occur when the region names $\mathfrak{r}_1$ and $\mathfrak{r}_2$ appear in order in the top stack.

The rules for `newRGNRef`, `readRGNRef`, and `writeRGNRef` respectively allocate, read, and write region allocated data. The rule for `newRGNRef` requires a region handle for a region in the top stack, chooses a fresh pointer in the region, and returns a modified top stack (with the value stored at the freshly chosen pointer) and the reference. The rule for `readRGNRef` requires a reference into a region in the top stack, and returns the value stored in the reference. Finally, the rule for `writeRGNRef` requires a reference into a region in the top stack and a new value, and returns a modified to stack (with the new values stored a the pointer).

It is important to note that the execution of a monadic command is predicated upon the command's region index corresponding to a live region in the top stack. While it will be possible to have commands that reference deallocated stacks and regions, it will not be possible to execute them. Furthermore, the restriction to the top stack corresponds to the fact that while `runRGN` computations can be nested, the inner computation must complete before executing a command in the outer computation. The type system of the next section and Appendix B.1 ensures that these invariants are preserved during the execution of well-typed programs.

$$\begin{array}{lrcl} \text{Type and index contexts} & \Delta & ::= & \cdot \mid \Delta, \alpha \mid \Delta, \vartheta \\ \text{Value contexts} & \Gamma & ::= & \cdot \mid \Gamma, x{:}\tau \end{array}$$

$$\mathsf{RGNPf}(\theta_1 \preceq \theta_2) \equiv \forall \beta.\, \mathsf{RGN}\ \theta_1\ \beta \to \mathsf{RGN}\ \theta_2\ \beta$$

Figure 3.9: Static semantics of $\mathsf{F}^{\mathsf{RGN}}$ (definitions)

3.2.3 Static Semantics of $\mathsf{F}^{\mathsf{RGN}}$

As noted above, well-typed programs obey several invariants, which are enforced with typing judgments. In particular, the typing judgments for an $\mathsf{F}^{\mathsf{RGN}}$ expression must ensure that the evaluation of the expression does not attempt to access a dead stack or region. For the surface syntax of $\mathsf{F}^{\mathsf{RGN}}$, it suffices to include typing judgments for expressions and various well-formedness judgments for types, indices, and contexts. As was stated previously, we purposefully omit judgments for the additional semantic objects introduced by the abstract machine configurations for $\mathsf{F}^{\mathsf{RGN}}$ (for example, typing judgments for towers), but these additional technical details and a (sketch of a) syntactic proof of type soundness for $\mathsf{F}^{\mathsf{RGN}}$ may be found in Appendix B.1.

Definitions Figure 3.9 presents additional definitions for syntactic objects that appear in the static semantics. Contexts Δ are ordered lists of type and index variables and contexts Γ are ordered lists of variables with types. We tacitly assume that all contexts are well-formed: Δ contains distinct type and index variables and Γ contains distinct value variables.

We recall the abbreviation $\mathsf{RGNPf}(\theta_1 \preceq \theta_2)$ for the type of a function that coerces any computation taking place in the region indexed by θ_1 into a computation taking place in the region indexed by θ_2.

$\boxed{\Delta; \Gamma \vdash_{\text{exp}} e : \tau}$

$$\frac{\vdash_{\text{ctxt}} \Delta; \Gamma}{\Delta; \Gamma \vdash_{\text{exp}} \mathtt{i} : \mathsf{Int}} \qquad \frac{\Delta; \Gamma \vdash_{\text{exp}} e_1 : \mathsf{Int} \quad \Delta; \Gamma \vdash_{\text{exp}} e_2 : \mathsf{Int}}{\Delta; \Gamma \vdash_{\text{exp}} e_1 \oplus e_2 : \mathsf{Int}} \qquad \frac{\Delta; \Gamma \vdash_{\text{exp}} e_1 : \mathsf{Int} \quad \Delta; \Gamma \vdash_{\text{exp}} e_2 : \mathsf{Int}}{\Delta; \Gamma \vdash_{\text{exp}} e_1 \ominus e_2 : \mathsf{Bool}}$$

$$\frac{\vdash_{\text{ctxt}} \Delta; \Gamma}{\Delta; \Gamma \vdash_{\text{exp}} \mathfrak{b} : \mathsf{Bool}} \qquad \frac{\Delta; \Gamma \vdash_{\text{exp}} e_b : \mathsf{Bool} \quad \Delta; \Gamma \vdash_{\text{exp}} e_t : \tau \quad \Delta; \Gamma \vdash_{\text{exp}} e_f : \tau}{\Delta; \Gamma \vdash_{\text{exp}} \mathtt{if}\ e_b\ \mathtt{then}\ e_t\ \mathtt{else}\ e_f : \tau}$$

Figure 3.10: Static semantics of $\mathsf{F}^{\mathsf{RGN}}$ (expressions (I))

Terms Figures 3.10–3.14 present the typing rules for the judgment $\Delta; \Gamma \vdash_{\text{exp}} e : \tau$, which asserts that under the type and index context Δ and the value context Γ, the expression e has the type τ.

The rules for constants, arithmetic and boolean operations, function abstraction and application, tuple introduction and selection, and type abstraction and instantiation are all completely standard. As expected in a monadic language, each command expression is given the monadic type $\mathsf{RGN}\ \theta\ \tau$ the appropriate region index and result type. The typing rules for `returnRGN` and `thenRGN` correspond to the standard typing rules for monadic unit and bind operations. The typing rules for `newRGNRef`, `readRGNRef`, and `writeRGNRef` are straight-forward.

The key rules are those relating to the creation of new regions. Recall that we would like to consider a value of type $\mathsf{RGN}\ \theta\ \tau$ as a region-transformer – that is, it accepts a region (indexed by θ), performs some operations (such as allocating into the region), and returns a value and the modified region. However, this is slightly inaccurate, owing to the fact that the $\Downarrow_\kappa$ judgment takes a stack S and returns a

$\boxed{\Delta; \Gamma \vdash_{\text{exp}} e : \tau}$

$$\frac{\vdash_{\text{ctxt}} \Delta; \Gamma \qquad x \in dom(\Gamma) \qquad \Gamma(x) = \tau}{\Delta; \Gamma \vdash_{\text{exp}} x : \tau} \qquad \frac{\Delta; \Gamma, x{:}\tau_x \vdash_{\text{exp}} e : \tau}{\Delta; \Gamma \vdash_{\text{exp}} \lambda x{:}\tau_x.\, e : \tau_x \to \tau}$$

$$\frac{\Delta; \Gamma \vdash_{\text{exp}} e_f : \tau_x \to \tau \qquad \Delta; \Gamma \vdash_{\text{exp}} e_a : \tau_x}{\Delta; \Gamma \vdash_{\text{exp}} e_f\ e_a : \tau} \qquad \frac{\vdash_{\text{ctxt}} \Delta; \Gamma \qquad \Delta; \Gamma \vdash_{\text{exp}} e_i : \tau_i \quad {}^{i \in 1 \ldots n}}{\Delta; \Gamma \vdash_{\text{exp}} \langle e_1, \ldots, e_n \rangle : \tau_1 \times \cdots \times \tau_n}$$

$$\frac{\Delta; \Gamma \vdash_{\text{exp}} e : \tau_1 \times \cdots \times \tau_n \qquad 1 \leq i \leq n}{\Delta; \Gamma \vdash_{\text{exp}} \mathtt{sel}_i\ e : \tau_i} \qquad \frac{\vdash_{\text{ctxt}} \Delta; \Gamma \qquad \Delta, \alpha; \Gamma \vdash_{\text{exp}} e : \tau}{\Delta; \Gamma \vdash_{\text{exp}} \Lambda\alpha.\, e : \forall\alpha.\, \tau}$$

$$\frac{\Delta; \Gamma \vdash_{\text{exp}} e_f : \forall\alpha.\, \tau \qquad \Delta \vdash_{\text{type}} \tau_a}{\Delta; \Gamma \vdash_{\text{exp}} e_f\ [\tau_a] : \tau[\tau_a/\alpha]} \qquad \frac{\Delta; \Gamma \vdash_{\text{exp}} e_a : \tau_x \qquad \Delta; \Gamma, x{:}\tau_x \vdash_{\text{exp}} e_b : \tau}{\Delta; \Gamma \vdash_{\text{exp}} \mathtt{let}\ x = e_a\ \mathtt{in}\ e_b : \tau}$$

Figure 3.11: Static semantics of $\mathsf{F}^{\mathsf{RGN}}$ (expressions (II))

$\boxed{\Delta; \Gamma \vdash_{\text{exp}} e : \tau}$

$$\frac{\Delta \vdash_{\text{type}} \tau \qquad \Delta; \Gamma \vdash_{\text{exp}} v : \forall \vartheta.\, \mathsf{RGNHnd}\ \vartheta \to \mathsf{RGN}\ \vartheta\ \tau}{\Delta; \Gamma \vdash_{\text{exp}} \texttt{runRGN}\ [\tau]\ v : \tau}$$

$$\frac{\vdash_{\text{ctxt}} \Delta; \Gamma \qquad \Delta, \vartheta; \Gamma \vdash_{\text{exp}} e : \tau}{\Delta; \Gamma \vdash_{\text{exp}} \Lambda\vartheta.\, e : \forall \vartheta.\, \tau} \qquad \frac{\Delta; \Gamma \vdash_{\text{exp}} e_f : \forall \vartheta.\, \tau \qquad \Delta \vdash_{\text{index}} \theta_a}{\Delta; \Gamma \vdash_{\text{exp}} e_f\ [\theta_a] : \tau[\theta_a/\vartheta]}$$

Figure 3.12: Static semantics of $\mathsf{F}^{\mathsf{RGN}}$ (expressions (III))

new stack S'; furthermore, a stack of regions admits a region *outlives* relationship. Hence, we should consider a value of type $\mathsf{RGN}\ \theta\ \tau$ as a region-stack-transformer – that is, it accepts a stack of regions (indexed by θ, corresponding to a particular member of the region stack), performs some operations (such as allocating into the regions), and returns a value and the modified stack of regions. Note that the actual stack of regions passed at runtime may include regions younger than the region to which θ corresponds; θ simply ensures the liveness of a particular region (and all regions older than it), without excluding the liveness of younger regions.

We first examine the typing rule for the `runRGN` expression:

$$\frac{\Delta \vdash_{\text{type}} \tau \qquad \Delta; \Gamma \vdash_{\text{exp}} v : \forall \vartheta.\, \mathsf{RGNHnd}\ \vartheta \to \mathsf{RGN}\ \vartheta\ \tau}{\Delta; \Gamma \vdash_{\text{exp}} \texttt{runRGN}\ [\tau]\ v : \tau}$$

As stated above, the argument to `runRGN` should describe a region computation. In fact, we require v to be an index polymorphic function that yields a region computation after being applied to a (fresh) region handle. The effect of universally quantifying over the index in the type of v is to require v to make no assumptions about the input stack of regions (e.g., the existence of pre-allocated

$\boxed{\Delta; \Gamma \vdash_{\text{exp}} e : \tau}$

$$\frac{\Delta \vdash_{\text{index}} \theta \qquad \Delta \vdash_{\text{type}} \tau \qquad \Delta; \Gamma \vdash_{\text{exp}} v : \tau}{\Delta; \Gamma \vdash_{\text{exp}} \texttt{returnRGN}\ [\theta]\ [\tau]\ v : \mathsf{RGN}\ \theta\ \tau}$$

$$\frac{\Delta \vdash_{\text{index}} \theta \qquad \Delta \vdash_{\text{type}} \tau_a \qquad \Delta \vdash_{\text{type}} \tau_b \qquad \Delta; \Gamma \vdash_{\text{exp}} v_a : \mathsf{RGN}\ \theta\ \tau_a \qquad \Delta; \Gamma \vdash_{\text{exp}} v_f : \tau_a \to \mathsf{RGN}\ \theta\ \tau_b}{\Delta; \Gamma \vdash_{\text{exp}} \texttt{thenRGN}\ [\theta]\ [\tau_a]\ [\tau_b]\ v_a\ v_f : \mathsf{RGN}\ \theta\ \tau_b}$$

$$\frac{\Delta \vdash_{\text{index}} \theta_1 \qquad \Delta \vdash_{\text{type}} \tau \qquad \Delta; \Gamma \vdash_{\text{exp}} v : \forall \vartheta.\, \mathsf{RGNPf}(\theta_1 \preceq \vartheta_2) \to \mathsf{RGNHnd}\ \vartheta_2 \to \mathsf{RGN}\ \vartheta_2\ \tau}{\Delta; \Gamma \vdash_{\text{exp}} \texttt{letRGN}\ [\theta_1]\ [\tau]\ v : \mathsf{RGN}\ \theta_1\ \tau}$$

Figure 3.13: Static semantics of $\mathsf{F}^{\mathsf{RGN}}$ (commands (I))

$\boxed{\Delta; \Gamma \vdash_{\text{exp}} e : \tau}$

$$\frac{\Delta \vdash_{\text{index}} \theta \qquad \Delta \vdash_{\text{type}} \tau \qquad \Delta; \Gamma \vdash_{\text{exp}} v_h : \mathsf{RGNHnd}\ \theta \qquad \Delta; \Gamma \vdash_{\text{exp}} v_\star : \tau}{\Delta; \Gamma \vdash_{\text{exp}} \texttt{newRGNRef}\ [\theta]\ [\tau]\ v_h\ v_\star : \mathsf{RGN}\ \theta\ (\mathsf{RGNRef}\ \theta\ \tau)}$$

$$\frac{\Delta \vdash_{\text{index}} \theta \qquad \Delta \vdash_{\text{type}} \tau \qquad \Delta; \Gamma \vdash_{\text{exp}} v_r : \mathsf{RGNRef}\ \theta\ \tau}{\Delta; \Gamma \vdash_{\text{exp}} \texttt{readRGNRef}\ [\theta]\ [\tau]\ v_r : \mathsf{RGN}\ \theta\ \tau}$$

$$\frac{\Delta \vdash_{\text{index}} \theta \qquad \Delta \vdash_{\text{type}} \tau \qquad \Delta; \Gamma \vdash_{\text{exp}} v_r : \mathsf{RGNRef}\ \theta\ \tau \qquad \Delta; \Gamma \vdash_{\text{exp}} v_\star : \tau}{\Delta; \Gamma \vdash_{\text{exp}} \texttt{writeRGNRef}\ [\theta]\ [\tau]\ v_r\ v_\star : \mathsf{1}_\times}$$

Figure 3.14: Static semantics of $\mathsf{F}^{\mathsf{RGN}}$ (commands (II))

references). Furthermore, all region-transformer operations are "infected" with the index: when combining operations, the rule for `thenRGN` requires the indices in the RGN computations to be the same; references allocated, read, and written using `newRGNRef`, `readRGNRef`, and `writeRGNRef` require the index of the RGNRef to be the same as the computation in which the operation occurs. While witness functions (discussed in more detail below) may coerce a region computation indexed by θ to a region computation indexed by θ' for a younger index θ', this coercion simply "infects" the computation with a younger index whose liveness implies the liveness of the older index. Thus, if a region computation $\mathsf{RGN}\ \theta\ \tau$ were to return a value that depended upon the region indexed by θ, then θ (or some younger, as of yet unintroduced, index θ') would appear in the type τ. Since the type τ appears outside the scope of the type variable θ in the typing rule for `runRGN`, it follows that θ cannot appear in the type τ. Therefore, it must be the case that the value returned by the computation described by v does not depend upon the index which will instantiate θ. Taken together, these facts ensure that an arbitrary new stack and region can be supplied to the computation and that the value returned will not leak any means of accessing the region or values allocated within it; hence, the region can be deallocated at the end of the computation. Finally, because we require region handles for allocating within regions, we provide the region handle for the newly created region as the argument to a function that yields the computation to be executed.

The typing rule for `letRGN` is very similar:

$$\frac{\Delta \vdash_{\text{index}} \theta_1 \qquad \Delta \vdash_{\text{type}} \tau \qquad \Delta;\Gamma \vdash_{\text{exp}} v : \forall\vartheta.\, \mathsf{RGNPf}(\theta_1 \preceq \vartheta_2) \rightarrow \mathsf{RGNHnd}\ \vartheta_2 \rightarrow \mathsf{RGN}\ \vartheta_2\ \tau}{\Delta;\Gamma \vdash_{\text{exp}} \texttt{letRGN}\ [\theta_1]\ [\tau]\ v : \mathsf{RGN}\ \theta_1\ \tau}$$

$\boxed{\Delta \vdash_{\text{type}} \tau}$

$$\frac{\alpha \in dom(\Delta)}{\Delta \vdash_{\text{type}} \alpha} \qquad \frac{}{\Delta \vdash_{\text{type}} \mathsf{Int}} \qquad \frac{}{\Delta \vdash_{\text{type}} \mathsf{Bool}} \qquad \frac{\Delta \vdash_{\text{type}} \tau_1 \qquad \Delta \vdash_{\text{type}} \tau_2}{\Delta \vdash_{\text{type}} \tau_1 \rightarrow \tau_2}$$

$$\frac{\Delta \vdash_{\text{type}} \tau_i \quad {}^{i \in 1 \ldots n}}{\Delta \vdash_{\text{type}} \tau_1 \times \cdots \times \tau_n} \qquad \frac{\Delta, \alpha \vdash_{\text{type}} \tau}{\Delta \vdash_{\text{type}} \forall \alpha. \tau} \qquad \frac{\Delta \vdash_{\text{index}} \theta \qquad \Delta \vdash_{\text{type}} \tau}{\Delta \vdash_{\text{type}} \mathsf{RGN}\ \theta\ \tau}$$

$$\frac{\Delta \vdash_{\text{index}} \theta \qquad \Delta \vdash_{\text{type}} \tau}{\Delta \vdash_{\text{type}} \mathsf{RGNRef}\ \theta\ \tau} \qquad \frac{\Delta \vdash_{\text{index}} \theta}{\Delta \vdash_{\text{type}} \mathsf{RGNHnd}\ \theta} \qquad \frac{\Delta, \vartheta \vdash_{\text{type}} \tau}{\Delta \vdash_{\text{type}} \forall \vartheta. \tau}$$

$\boxed{\Delta \vdash_{\text{index}} \theta}$

$$\frac{\vartheta \in dom(\Delta)}{\Delta \vdash_{\text{index}} \vartheta}$$

Figure 3.15: Static semantics of $\mathsf{F}^{\mathsf{RGN}}$ (types and indices)

Exactly the same argument as above applies, except that we additionally have a witness argument of type $\mathsf{RGNPf}(\theta_1 \preceq \vartheta_2)$. The operational behavior of `letRGN` ensures that the newly allocated region is related to previously allocated regions according to the stack discipline. The witness argument is provided to the computation taking place in the stack with the inner/younger region allocated in order to coerce computations (such as allocating a new value in some outer/older region) from a computation indexed by the outer region to a computation indexed by the the inner region. This coercion is safe because every region in the stack denoted by θ_1 outlives every region in the stack denoted by ϑ_2. Operationally, such a witness function acts as the identity function.

$\boxed{\Delta \vdash_{\text{vctxt}} \Gamma}$

$$\frac{}{\Delta \vdash_{\text{vctxt}} \cdot} \qquad \frac{\Delta \vdash_{\text{vctxt}} \Gamma \qquad x \notin dom(\Gamma) \qquad \Delta \vdash_{\text{type}} \tau}{\Delta \vdash_{\text{vctxt}} \Gamma, x{:}\tau}$$

$\boxed{\vdash_{\text{ctxt}} \Delta; \Gamma}$

$$\frac{\Delta \vdash_{\text{vctxt}} \Gamma}{\vdash_{\text{ctxt}} \Delta; \Gamma}$$

Figure 3.16: Static semantics of $\mathsf{F}^{\mathsf{RGN}}$ (contexts)

Types, indices, and contexts Figures 3.15 and 3.16 contain additional (completely standard) judgments for ensuring that types τ, indices θ, and value contexts Γ are well-formed. These judgments simply enforce the invariant that no type or expression may depend upon unbound type or index variables.

Remarks

We may simplify the static semantics by noting that the typing judgments for each of the monadic commands are equivalent to the following type assignment:

$\texttt{runRGN} \;::\; \forall\alpha.\,(\forall\vartheta.\,\mathsf{RGNHnd}\ \vartheta \to \mathsf{RGN}\ \vartheta\ \alpha) \to \alpha$

$\texttt{returnRGN} \;::\; \forall\vartheta.\,\forall\alpha.\alpha \to \mathsf{RGN}\ \vartheta\ \alpha$

$\texttt{thenRGN} \;::\; \forall\vartheta.\,\forall\alpha,\beta.\,\mathsf{RGN}\ \vartheta\ \alpha \to (\alpha \to \mathsf{RGN}\ \vartheta\ \beta) \to \mathsf{RGN}\ \vartheta\ \beta$

$\texttt{letRGN} \;::$

$\quad \forall\vartheta_1.\,\forall\alpha.\,(\forall\vartheta_2.\,\mathsf{RGNPf}(\vartheta_1 \preceq \vartheta_2) \to \mathsf{RGNHnd}\ \vartheta_2 \to \mathsf{RGN}\ \vartheta_2\ \alpha) \to \mathsf{RGN}\ \vartheta_1\ \alpha$

$\texttt{newRGNRef} \;::\; \forall\vartheta.\,\forall\alpha.\,\mathsf{RGNHnd}\ \vartheta \to \alpha \to \mathsf{RGN}\ \vartheta\ (\mathsf{RGNRef}\ \vartheta\ \alpha)$

$\texttt{readRGNRef} \;::\; \forall\vartheta.\,\forall\alpha.\,\mathsf{RGNRef}\ \vartheta\ \alpha \to \mathsf{RGN}\ \vartheta\ \alpha$

$\texttt{writeRGNRef} \;::\; \forall\vartheta.\,\forall\alpha.\,\mathsf{RGNRef}\ \vartheta\ \alpha \to \alpha \to \mathsf{RGN}\ \vartheta\ 1_\times$

Treating the monadic commands as syntactic forms simplifies the dynamic semantics and proofs, as there is no need to consider partially applied forms.

Finally, it is easy to see that the typing rules prevent running a computation that would dereference a dangling pointer. Consider the following code fragment:

$$
\begin{array}{l}
\texttt{letRGN}\ [\theta_1]\ [?] \\
\quad (\Lambda\vartheta_2. \\
\quad \lambda w{:}\mathsf{RGNPf}(\theta_1 \preceq \vartheta_2). \\
\quad \lambda h{:}\mathsf{RGNHnd}\ \vartheta_1. \\
\quad \texttt{thenRGN}\ [\vartheta_2]\ [\mathsf{RGNRef}\ \vartheta_2\ \mathsf{Int}]\ [\mathsf{Bool} \to \mathsf{RGN}\ \vartheta_2\ \mathsf{Int}] \\
\qquad\quad (\texttt{newRGNRef}\ [\vartheta_2]\ [\mathsf{Int}]\ h\ 42) \\
\qquad\quad (\lambda r{:}\mathsf{RGNRef}\ \vartheta_2\ \mathsf{Int}. \\
\qquad\quad \texttt{returnRGN}\ [\vartheta_2]\ [\mathsf{Bool} \to \mathsf{RGN}\ \vartheta_2\ \mathsf{Int}] \\
\qquad\qquad\quad (\lambda b{:}\mathsf{Bool}.\ \texttt{readRGNRef}\ [\vartheta_2]\ [\mathsf{Int}]\ r)))
\end{array}
$$

This fragment creates a new region, allocates an integer reference in the newly created region, and finally returns a function and destroys the region. The returned function, when applied, yields a computation which attempts to reads from the reference (in the now destroyed region). Note that there is no type for ? that allows the code fragment to be accepted by the typing rules. The only possible type for ? is $\mathsf{Bool} \to \mathsf{RGN}\ \vartheta_2\ \mathsf{Int}$, but this type cannot be used outside the scope of ϑ_2. It is this use of parametric polymorphism that prevents dangling pointers from being dereferenced.

3.3 Translation: From SEC to $\mathsf{F}^{\mathsf{RGN}}$

In this section we present a type- and semantics-preserving translation from the Single Effect Calculus to $\mathsf{F}^{\mathsf{RGN}}$. Many of the key components of the translation

should be obvious from the suggestive naming of the previous sections. We clearly intend `letregion` to be translated (in some fashion) to `letRGN`. Likewise, we can expect types of the form (ω, ρ) to be translated to types of the form $\mathsf{RGNRef}\ \theta\ \tau$. It further seems likely that the outlives relation $\rho_2 \succeq \rho_1$ should be related to the witness functions $\mathsf{RGNPf}(\theta_1 \preceq \theta_2)$. We present the translation in stages, as there are some subtleties that require explanation.

We start with a few preliminaries. We assume an injection from the set $VVars^{\mathsf{SEC}}$ to the set $VVars^{\mathsf{FRGN}}$ respectively. In the translation, applications of such injections will be clear from context and we freely use source value variables as target value variables. We also assume an injection from the set $RVars^{\mathsf{SEC}}$ to the set $IVars^{\mathsf{FRGN}}$; this injection, written ϑ_ϱ, will denote the RGN index for the region ϱ. We further assume two additional injections from the set $RVars^{\mathsf{SEC}}$ to the set $VVars^{\mathsf{FRGN}}$; the first, written h_ϱ, will denote the handle for the region ϱ, while the second, written w_ϱ, will denote the witnesses which coerce the region ϱ to its bounding regions.

The translation is a typed call-by-value monad translation, similar to the standard translation given by Sabry and Wadler [70]. We have not attempted to optimize the translation to avoid the introduction of "administrative" redexes. We feel that this simplifies the translation, and it does not significantly complicate the proof that the translation preserves the semantics, owing to the fact that only three expression forms in the source calculus are value forms. The translation is given by a number of functions: $\mathbb{I}[\![\cdot]\!]$ translates into indices, $\mathbb{T}[\![\cdot]\!]$ translates into types, $\mathbb{D}[\![\cdot]\!]$ translates into type and index contexts, $\mathbb{G}[\![\cdot]\!]$ translates into value contexts, and $\mathbb{E}[\![\cdot]\!]$ translates into expressions. Technically, there are separate functions for each syntactic class in the source calculus, but we elide this detail as it is always clear

Translations yielding indices

Regions

$$\mathbb{I}\left[\!\!\left[\frac{\vdash_{\text{rctxt}} \Delta \qquad \varrho \in dom(\Delta)}{\Delta \vdash_{\text{region}} \varrho}\right]\!\!\right] \;=\; \vartheta_{\varrho}$$

Figure 3.17: Translation from SEC to $\mathsf{F}^{\mathsf{RGN}}$ (regions (I))

from context. Additionally, to reduce notational clutter, translations from judgments are often written in an abbreviated form giving only the main component; the corresponding judgment should be clear from context.

Regions, boxed types, types, and outlives relations Figures 3.17, 3.18, and 3.19 shows the translation of regions, types and boxed types, and the outlives relations and Figure 3.20 gives the extension of the translation to contexts. As expected, the type (ω, ρ) is translated to $\mathsf{RGNRef}\ \mathbb{I}[\![\rho]\!]\ \mathbb{T}[\![\omega]\!]$, whereby region allocated values in the source are also region allocated in the target. The translations of primitive types and product types are trivial. More interesting are the translations of function types and region abstraction types. Functions with effects bounded by the region π are translated into pure functions that yield computations in the RGN monad indexed by π, whereas region abstractions are translated into RGN index abstractions. Because the target calculus requires explicit region handles for allocation, each time a region is in scope in the source calculus, the region handle must be in scope in the target calculus. This explains the appearance of the $\mathsf{RGNHnd}\ \varrho$ type in the translation. Likewise, the target calculus makes witness functions explicit, whereas in the source calculus such coercions are implied by $\succeq$ related regions. Hence, we interpret $\varrho \succeq \{\rho_1, \ldots, \rho_n\}$ as an n-tuple of functions,

Translations yielding types

Types

$$\mathbb{T}\left[\!\!\left[\frac{\vdash_{\text{rctxt}} \Delta}{\Delta \vdash_{\text{type}} \mathsf{Bool}}\right]\!\!\right] \;=\; \mathsf{Bool}$$

$$\mathbb{T}\left[\!\!\left[\frac{\Delta \vdash_{\text{btype}} \omega \qquad \Delta \vdash_{\text{region}} \rho}{\Delta \vdash_{\text{type}} (\omega, \rho)}\right]\!\!\right] \;=\; \mathsf{RGNRef}\ \mathbb{I}[\![\rho]\!]\ \ \mathbb{T}[\![\omega]\!]$$

Boxed types

$$\mathbb{T}\left[\!\!\left[\frac{\vdash_{\text{rctxt}} \Delta}{\Delta \vdash_{\text{btype}} \mathsf{Int}}\right]\!\!\right] \;=\; \mathsf{Int}$$

$$\mathbb{T}\left[\!\!\left[\frac{\Delta \vdash_{\text{type}} \tau_1 \qquad \Delta \vdash_{\text{region}} \pi' \qquad \Delta \vdash_{\text{type}} \tau_2}{\Delta \vdash_{\text{btype}} \tau_1 \xrightarrow{\pi'} \tau_2}\right]\!\!\right] \;=$$

$$\mathbb{T}[\![\tau_1]\!] \to \mathsf{RGN}\ \mathbb{I}[\![\pi']\!]\ \ \mathbb{T}[\![\tau_2]\!]$$

$$\mathbb{T}\left[\!\!\left[\frac{\Delta \vdash_{\text{type}} \tau_1 \qquad \cdots \qquad \Delta \vdash_{\text{type}} \tau_n}{\Delta \vdash_{\text{btype}} \tau_1 \times \cdots \times \tau_n}\right]\!\!\right] \;=\; \mathbb{T}[\![\tau_1]\!] \times \cdots \times \mathbb{T}[\![\tau_n]\!]$$

$$\mathbb{T}\left[\!\!\left[\frac{\Delta \vdash_{\text{eff}} \phi \qquad \Delta, \varrho \succeq \phi \vdash_{\text{region}} \pi' \qquad \Delta, \varrho \succeq \phi \vdash_{\text{type}} \tau}{\Delta \vdash_{\text{btype}} \forall \varrho \succeq \phi.^{\pi'}\, \tau}\right]\!\!\right] \;=$$

$$\forall \vartheta_\varrho.\, \mathbb{T}[\![\varrho \succeq \phi]\!] \to \mathsf{RGNHnd}\ \vartheta_\varrho \to \mathsf{RGN}\ \mathbb{I}[\![\pi']\!]\ \ \mathbb{T}[\![\tau]\!]$$

Figure 3.18: Translation from SEC to $\mathsf{F}^{\mathsf{RGN}}$ (types and boxed types)

Translations yielding types

Outlives relations

$$\mathbb{T}[\![\Delta \vdash_{\mathrm{rr}} \rho_2 \succeq \rho_1]\!] \;=\; \mathsf{RGNPf}(\mathbb{I}[\![\rho_1]\!] \preceq \mathbb{I}[\![\rho_2]\!]) \;=\; \forall\beta.\, \mathsf{RGN}\ \mathbb{I}[\![\rho_1]\!]\ \beta \to \mathsf{RGN}\ \mathbb{I}[\![\rho_2]\!]\ \beta$$

$$\mathbb{T}\left[\!\!\left[\frac{\vdash_{\mathrm{rctxt}} \Delta \qquad \Delta \vdash_{\mathrm{rr}} \rho \succeq \rho_i \quad {}^{i\in 1\ldots n}}{\Delta \vdash_{\mathrm{re}} \rho \succeq \{\rho_1, \ldots, \rho_n\}}\right]\!\!\right] \;=\; (\mathbb{T}[\![\rho \succeq \rho_1]\!] \times \cdots \times \mathbb{T}[\![\rho \succeq \rho_n]\!])$$

Figure 3.19: Translation from SEC to $\mathsf{F}^{\mathsf{RGN}}$ (outlives relations (I))

each witnessing a coercion from region ρ_i to ϱ. This interpretation is formalized by the $\mathbb{T}[\![\varrho \succeq \{\rho_1, \ldots \rho_n\}]\!]$ translation.[5]

Contexts We extend the region and type translations to contexts in the obvious manner. In addition to translating region variables to type variables and translating the types of variables in value contexts, we have additional translations from region contexts to value contexts. As explained above, region handles and witness functions are explicit values in the target calculus. Hence, our translation maintains the invariant that whenever a region variable $\varrho \succeq \phi$ is in scope in the source calculus, the variables h_ϱ and w_ϱ are in scope in the target calculus. The variable h_ϱ (of type $\mathsf{RGNHnd}\ \mathbb{I}[\![\varrho]\!]$) is the handle for the region ϱ and the variable w_ϱ (of type $\mathbb{T}[\![\varrho \succeq \phi]\!]$) is the tuple holding the witness functions that coerce to region ϱ.

[5]Note that in the Single Effect Calculus, we only substitute regions for region variables. This means that the sets of regions that appear in the program never change size (although they may change elements as a result of substitution). The $\mathbb{T}[\![\Delta \vdash_{\mathrm{re}} \rho \succeq \{\rho_1, \ldots \rho_n\}]\!]$ translations require keeping the ordering of regions in a set $\{\rho_1, \ldots, \rho_n\}$ constant. It does not require a global ordering on region variables; such an ordering would not suffice for our purposes, because the ordering of elements in a set might change after substitution. Instead, we take $\{\rho_1, \ldots, \rho_n\}$ as a list with fixed order, where substitution preserves the order. Hence, we can realize the witness with an ordered tuple.

Translations yielding type contexts

Region contexts

$$\mathbb{D}\left[\!\!\left[\frac{}{\vdash_{\text{rctxt}} \cdot}\right]\!\!\right] \;=\; \cdot$$

$$\mathbb{D}\left[\!\!\left[\frac{\vdash_{\text{rctxt}} \Delta \qquad \varrho \notin dom(\Delta) \qquad \vdash_{\text{eff}} \phi}{\vdash_{\text{rctxt}} \Delta, \varrho \succeq \phi}\right]\!\!\right] \;=\; \mathbb{D}[\![\Delta]\!]\,, \vartheta_\varrho$$

Translations yielding value contexts

Region contexts

$$\mathbb{G}\left[\!\!\left[\frac{}{\vdash_{\text{rctxt}} \cdot}\right]\!\!\right] \;=\; \cdot$$

$$\mathbb{G}\left[\!\!\left[\frac{\vdash_{\text{rctxt}} \Delta \qquad \varrho \notin dom(\Delta) \qquad \vdash_{\text{eff}} \phi}{\vdash_{\text{rctxt}} \Delta, \varrho \succeq \phi}\right]\!\!\right] \;=$$

$$\mathbb{G}[\![\Delta]\!]\,, h_\varrho{:}\mathsf{RGNHnd}\ \vartheta_\varrho, w_\varrho{:}\mathbb{T}[\![\varrho \succeq \phi]\!]$$

Value contexts

$$\mathbb{G}\left[\!\!\left[\frac{\vdash_{\text{rctxt}} \Delta}{\Delta \vdash_{\text{vctxt}} \cdot}\right]\!\!\right] \;=\; \cdot$$

$$\mathbb{G}\left[\!\!\left[\frac{\Delta \vdash_{\text{vctxt}} \Gamma \qquad x \notin dom(\Gamma) \qquad \Delta \vdash_{\text{type}} \tau}{\Delta \vdash_{\text{vctxt}} \Gamma, x{:}\tau}\right]\!\!\right] \;=\; \mathbb{G}[\![\Gamma]\!]\,, x{:}\mathbb{T}[\![\tau]\!]$$

Figure 3.20: Translation from SEC to $\mathsf{F}^{\mathsf{RGN}}$ (contexts)

Translations yielding expressions

Witnesses

$$\mathbb{E}\left[\!\!\left[\frac{\vdash_{\text{rctxt}} \Delta \qquad (\varrho \succeq \{\rho_1, \ldots, \rho_i, \ldots, \rho_n\}) \in \Delta}{\Delta \vdash_{\text{rr}} \varrho \succeq \rho_i}\right]\!\!\right] =$$

$$\Lambda\beta.\, \lambda k{:}\mathsf{RGN}\ \mathbb{I}[\![\rho_i]\!]\ \ \beta.\, \texttt{let}\ w = \texttt{sel}_i\ w_\varrho\ \texttt{in}\ w\ [\beta]\ k$$

$$\mathbb{E}\left[\!\!\left[\frac{\Delta \vdash_{\text{region}} \rho}{\Delta \vdash_{\text{rr}} \rho \succeq \rho}\right]\!\!\right] = \Lambda\beta.\, \lambda k{:}\mathsf{RGN}\ \mathbb{I}[\![\rho]\!]\ \ \beta.\, k$$

$$\mathbb{E}\left[\!\!\left[\frac{\Delta \vdash_{\text{rr}} \rho_2 \succeq \rho' \qquad \Delta \vdash_{\text{rr}} \rho' \succeq \rho_1}{\Delta \vdash_{\text{rr}} \rho_2 \succeq \rho_1}\right]\!\!\right] =$$

$$\Lambda\beta.\, \lambda k{:}\mathsf{RGN}\ \mathbb{I}[\![\rho_1]\!]\ \ \beta.\, \texttt{let}\ k' = \mathbb{E}[\![\rho' \succeq \rho_1]\!]\ \ [\beta]\ k\ \texttt{in}\ \mathbb{E}[\![\rho_2 \succeq \rho']\!]\ \ [\beta]\ k'$$

$$\mathbb{E}\left[\!\!\left[\frac{\vdash_{\text{rctxt}} \Delta \qquad \Delta \vdash_{\text{rr}} \rho \succeq \rho_i \quad {\scriptstyle i \in 1 \ldots n}}{\Delta \vdash_{\text{re}} \rho \succeq \{\rho_1, \ldots, \rho_n\}}\right]\!\!\right] = (\mathbb{E}[\![\rho \succeq \rho_1]\!], \ldots, \mathbb{E}[\![\rho \succeq \rho_n]\!])$$

Figure 3.21: Translation from SEC to $\mathsf{F}^{\mathsf{RGN}}$ (outlives relations (II))

Outlives relations Figure 3.21 shows the translation from SEC outlives relations to $\mathsf{F}^{\mathsf{RGN}}$ witness terms. The first three translations map the reflexive, transitive closure of the syntactic constraints in the source Δ into an appropriate coercion function. The final translation collects a set of coercion functions into a tuple; such a term is suitable as an argument to the translation of a region abstraction. Figure 3.22 translates a single region variable to its corresponding region handle (as a value variable).

Terms Figures 3.23–3.27 give the translation of terms. In order to make the translation easier to read, we introduce the following notation, reminiscent of

Translations yielding expressions

Regions

$$\mathbb{E}\left[\!\!\left[\frac{\vdash_{\text{rctxt}} \Delta \qquad \varrho \in dom(\Delta)}{\Delta \vdash_{\text{region}} \varrho}\right]\!\!\right] \;=\; h_\varrho$$

Figure 3.22: Translation from SEC to $\mathsf{F}^{\mathsf{RGN}}$ (regions (II))

Haskell's `do` notation:

$$\texttt{bindRGN}\ x{:}\tau_a \Leftarrow e_a\ ;\ e_b \equiv \texttt{let}\ k = e_a\ \texttt{in}\ \texttt{thenRGN}\ [\theta]\ [\tau_a]\ [\tau_b]\ k\ (\lambda x{:}\tau_a.\, e_b)$$

where k fresh

where θ and τ_b are inferred from context. Note that this induces the following derived rules:

$$\frac{(T; e_a) \Downarrow v}{(T; \texttt{bindRGN}\ x{:}\tau_a \Leftarrow e_a\ ;\ e_b) \Downarrow (\texttt{thenRGN}\ [\theta]\ [\tau_a]\ [\tau_b]\ v\ (\lambda x{:}\tau_a.\, e_b))}$$

$$\frac{\Delta \vdash_{\text{index}} \theta \qquad \Delta \vdash_{\text{type}} \tau_a \qquad \Delta \vdash_{\text{type}} \tau_b \qquad \Delta; \Gamma \vdash_{\text{exp}} e_a : \mathsf{RGN}\ \theta\ \tau_a \qquad \Delta; \Gamma, x{:}\tau_a \vdash_{\text{exp}} e_b : \mathsf{RGN}\ \theta\ \tau_b}{\Delta; \Gamma \vdash_{\text{exp}} \texttt{bindRGN}\ x{:}\tau_a \Leftarrow e_1\ ;\ e_2 : \mathsf{RGN}\ \theta\ \tau_b}$$

The translation of an integer constant is a canonical example of allocation in the target calculus. The allocation is accomplished by the `newRGNRef` command, applied to the appropriate region handle and value. However, the resulting command has type $\mathsf{RGN}\ \mathbb{I}[\![\rho]\!]\ (\mathsf{RGNRef}\ \mathbb{I}[\![\rho]\!]\ \mathsf{Int})$, whereas the source typing judgment requires the computation to be expressed relative to the region π. We coerce the computation using a witness function, whose existence is implied by the judgment $\Delta \vdash_{\text{rr}} \pi \succeq \rho$. Allocation of a function proceeds in exactly the same manner.

Translations yielding expressions

Expressions

$$\mathbb{E}\left[\!\!\left[\frac{\vdash_{\text{ctxt}} \Delta;\Gamma;\pi \qquad \Delta \vdash_{\text{region}} \rho \qquad \Delta \vdash_{\text{rr}} \pi \succeq \rho}{\Delta;\Gamma \vdash_{\text{exp}} \mathtt{i}\,\mathtt{at}\,\rho : (\mathsf{Int},\rho),\pi}\right]\!\!\right] =$$

$$\mathbb{E}[\![\pi \succeq \rho]\!]\ [\mathbb{T}[\![(\mathsf{Int},\rho)]\!]\ (\mathtt{newRGNRef}\ [\mathbb{I}[\![\rho]\!]]\ [\mathbb{T}[\![\mathsf{Int}]\!]\ \mathbb{E}[\![\rho]\!]\ \mathtt{i})$$

$$\mathbb{E}\left[\!\!\left[\frac{\begin{array}{c}\Delta;\Gamma \vdash_{\text{exp}} e_1 : (\mathsf{Int},\rho_1),\pi \qquad \Delta \vdash_{\text{rr}} \pi \succeq \rho_1 \\ \Delta;\Gamma \vdash_{\text{exp}} e_2 : (\mathsf{Int},\rho_2),\pi \qquad \Delta \vdash_{\text{rr}} \pi \succeq \rho_2 \\ \Delta \vdash_{\text{region}} \rho \qquad \Delta \vdash_{\text{rr}} \pi \succeq \rho\end{array}}{\Delta;\Gamma \vdash_{\text{exp}} e_1 \oplus e_2\,\mathtt{at}\,\rho : (\mathsf{Int},\rho),\pi}\right]\!\!\right] =$$

$\mathtt{bindRGN}\ a{:}\mathbb{T}[\![(\mathsf{Int},\rho_1)]\!] \Leftarrow \mathbb{E}[\![e_1]\!]$;

$\mathtt{bindRGN}\ a'{:}\mathbb{T}[\![\mathsf{Int}]\!] \Leftarrow \mathbb{E}[\![\pi \succeq \rho_1]\!]\ [\mathbb{T}[\![\mathsf{Int}]\!]]\ (\mathtt{readRGNRef}\ [\mathbb{I}[\![\rho_1]\!]]\ [\mathbb{T}[\![\mathsf{Int}]\!]]\ a)$;

$\mathtt{bindRGN}\ b{:}\mathbb{T}[\![(\mathsf{Int},\rho_2)]\!] \Leftarrow \mathbb{E}[\![e_2]\!]$;

$\mathtt{bindRGN}\ b'{:}\mathbb{T}[\![\mathsf{Int}]\!] \Leftarrow \mathbb{E}[\![\pi \succeq \rho_2]\!]\ [\mathbb{T}[\![\mathsf{Int}]\!]]\ (\mathtt{readRGNRef}\ [\mathbb{I}[\![\rho_2]\!]]\ [\mathbb{T}[\![\mathsf{Int}]\!]]\ b)$;

$\mathtt{let}\ z = a' \oplus b'\ \mathtt{in}$

$\mathbb{E}[\![\pi \succeq \rho]\!]\ [\mathbb{T}[\![(\mathsf{Int},\rho)]\!]\ (\mathtt{newRGNRef}\ [\mathbb{I}[\![\rho]\!]]\ [\mathbb{T}[\![\mathsf{Int}]\!]]\ \mathbb{E}[\![\rho]\!]\ z)$

where a, a', b, b', z fresh

Figure 3.23: Translation from SEC to $\mathsf{F}^{\mathsf{RGN}}$ (terms (I))

Translations yielding expressions

Expressions

$$\mathbb{E}\left[\!\!\left[\frac{\Delta;\Gamma \vdash_{\text{exp}} e_1 : (\mathsf{Int}, \rho_1), \pi \qquad \Delta \vdash_{\text{rr}} \pi \succeq \rho_1 \quad \Delta;\Gamma \vdash_{\text{exp}} e_2 : (\mathsf{Int}, \rho_2), \pi \qquad \Delta \vdash_{\text{rr}} \pi \succeq \rho_2}{\Delta;\Gamma \vdash_{\text{exp}} e_1 \circledcirc e_2 : \mathsf{Bool}, \pi}\right]\!\!\right] =$$

$$\begin{array}{l}
\texttt{bindRGN}\ a{:}\mathbb{T}[\![(\mathsf{Int}, \rho_1)]\!] \Leftarrow \mathbb{E}[\![e_1]\!]\ ; \\
\texttt{bindRGN}\ a'{:}\mathbb{T}[\![\mathsf{Int}]\!] \Leftarrow \mathbb{E}[\![\pi \succeq \rho_1]\!]\ [\mathbb{T}[\![\mathsf{Int}]\!]\ (\texttt{readRGNRef}\ [\mathbb{I}[\![\rho_1]\!]]\ [\mathbb{T}[\![\mathsf{Int}]\!]\ a)\ ; \\
\texttt{bindRGN}\ b{:}\mathbb{T}[\![(\mathsf{Int}, \rho_2)]\!] \Leftarrow \mathbb{E}[\![e_2]\!]\ ; \\
\texttt{bindRGN}\ b'{:}\mathbb{T}[\![\mathsf{Int}]\!] \Leftarrow \mathbb{E}[\![\pi \succeq \rho_2]\!]\ [\mathbb{T}[\![\mathsf{Int}]\!]\ (\texttt{readRGNRef}\ [\mathbb{I}[\![\rho_2]\!]]\ [\mathbb{T}[\![\mathsf{Int}]\!]\ b)\ ; \\
\texttt{let}\ z = a' \circledcirc b'\ \texttt{in} \\
\texttt{returnRGN}\ [\mathbb{I}[\![\pi]\!]]\ [\mathbb{T}[\![\mathsf{Bool}]\!]\ z
\end{array}$$

where a, a', b, b', z fresh

$$\mathbb{E}\left[\!\!\left[\frac{\vdash_{\text{ctxt}} \Delta;\Gamma;\pi}{\Delta;\Gamma \vdash_{\text{exp}} \mathfrak{b} : \mathsf{Bool}, \pi}\right]\!\!\right] = \texttt{returnRGN}\ [\mathbb{I}[\![\pi]\!]]\ [\mathbb{T}[\![\mathsf{Bool}]\!]\ \mathfrak{b}$$

$$\mathbb{E}\left[\!\!\left[\frac{\Delta;\Gamma \vdash_{\text{exp}} e_b : \mathsf{Bool}, \pi \quad \Delta;\Gamma \vdash_{\text{exp}} e_t : \tau, \pi \qquad \Delta;\Gamma \vdash_{\text{exp}} e_f : \tau, \pi}{\Delta;\Gamma \vdash_{\text{exp}} \texttt{if}\ e_b\ \texttt{then}\ e_t\ \texttt{else}\ e_f : \tau, \pi}\right]\!\!\right] =$$

$$\texttt{bindRGN}\ z{:}\mathbb{T}[\![\mathsf{Bool}]\!] \Leftarrow \mathbb{E}[\![e_b]\!]\ ;\ \texttt{if}\ z\ \texttt{then}\ \mathbb{E}[\![e_t]\!]\ \texttt{else}\ \mathbb{E}[\![e_f]\!]$$

where z fresh

Figure 3.24: Translation from SEC to $\mathsf{F}^{\mathsf{RGN}}$ (terms (II))

Translations yielding expressions

Expressions

$$\mathbb{E}\left[\!\!\left[\frac{\vdash_{\text{ctxt}} \Delta;\Gamma;\pi \qquad x \in dom(\Gamma) \qquad \Gamma(x) = \tau}{\Delta;\Gamma \vdash_{\text{exp}} x : \tau, \pi}\right]\!\!\right] =$$

$$\texttt{returnRGN}\ [\mathbb{I}[\![\pi]\!]]\ [\mathbb{T}[\![\tau]\!]]\ x$$

$$\mathbb{E}\left[\!\!\left[\frac{\Delta;\Gamma, x{:}\tau_x \vdash_{\text{exp}} e : \tau, \pi' \qquad \Delta \vdash_{\text{region}} \rho \qquad \Delta \vdash_{\text{rr}} \pi \succeq \rho}{\Delta;\Gamma \vdash_{\text{exp}} \lambda x : \tau_x .^{\pi'}\, e\ \texttt{at}\ \rho : (\tau_x \xrightarrow{\pi'} \tau, \rho), \pi}\right]\!\!\right] =$$

$$\mathbb{E}[\![\pi \succeq \rho]\!]\ \ [\mathbb{T}\left[\!\!\left[(\tau_x \xrightarrow{\pi'} \tau, \rho)\right]\!\!\right]]$$

$$(\texttt{newRGNRef}\ [\mathbb{I}[\![\rho]\!]]\ [\mathbb{T}\left[\!\!\left[\tau_x \xrightarrow{\pi'} \tau\right]\!\!\right]]\ \mathbb{E}[\![\rho]\!]\ \ (\lambda x{:}\mathbb{T}[\![\tau_x]\!]\,.\,\mathbb{E}[\![e]\!]))$$

$$\mathbb{E}\left[\!\!\left[\frac{\begin{array}{c}\Delta;\Gamma \vdash_{\text{exp}} e_f : (\tau_x \xrightarrow{\pi'_f} \tau, \rho_f), \pi \qquad \Delta \vdash_{\text{rr}} \pi \succeq \rho_f \\ \Delta;\Gamma \vdash_{\text{exp}} e_a : \tau_x, \pi \qquad \Delta \vdash_{\text{rr}} \pi \succeq \pi'_f\end{array}}{\Delta;\Gamma \vdash_{\text{exp}} e_f\ e_a : \tau, \pi}\right]\!\!\right] =$$

$$\texttt{bindRGN}\ f{:}\mathbb{T}\left[\!\!\left[(\tau_x \xrightarrow{\pi_f} \tau, \rho_f)\right]\!\!\right] \Leftarrow \mathbb{E}[\![e_f]\!]\ ;$$

$$\texttt{bindRGN}\ g{:}\mathbb{T}\left[\!\!\left[\tau_x \xrightarrow{\pi'_f} \tau\right]\!\!\right]$$

$$\Leftarrow \mathbb{E}[\![\pi \succeq \rho_f]\!]\ \ [\mathbb{T}\left[\!\!\left[\tau_x \xrightarrow{\pi'_f} \tau\right]\!\!\right]]$$

$$(\texttt{readRGNRef}\ [\mathbb{I}[\![\rho_f]\!]]\ [\mathbb{T}\left[\!\!\left[\tau_x \xrightarrow{\pi'_f} \tau\right]\!\!\right]]\ f)\ ;$$

$$\texttt{bindRGN}\ a{:}\mathbb{T}[\![\tau_x]\!] \Leftarrow \mathbb{E}[\![e_a]\!]\ ;$$

$$\mathbb{E}\left[\!\!\left[\pi \succeq \pi'_f\right]\!\!\right]\ \ [\mathbb{T}[\![\tau]\!]]\ (g\ a)$$

where f, g, a fresh

Figure 3.25: Translation from SEC to $\mathsf{F}^{\mathsf{RGN}}$ (terms (III))

Translations yielding expressions

Expressions

$$\mathbb{E}\left[\!\!\left[\frac{\Delta;\Gamma \vdash_{\text{exp}} e_1 : \tau_1, \pi \qquad \cdots \qquad \Delta;\Gamma \vdash_{\text{exp}} e_n : \tau_n, \pi \qquad \Delta \vdash_{\text{region}} \rho \qquad \Delta \vdash_{\text{rr}} \pi \succeq \rho}{\Delta;\Gamma \vdash_{\text{exp}} \langle e_1, \ldots, e_n \rangle \,\texttt{at}\, \rho : (\tau_1 \times \cdots \times \tau_n, \rho), \pi}\right]\!\!\right] =$$

$\texttt{bindRGN}\ x_1{:}\mathbb{T}[\![\tau_1]\!] \Leftarrow \mathbb{E}[\![e_1]\!]\ ;$

$\cdots$

$\texttt{bindRGN}\ x_n{:}\mathbb{T}[\![\tau_2]\!] \Leftarrow \mathbb{E}[\![e_2]\!]\ ;$

$\mathbb{E}[\![\pi \succeq \rho]\!]\ [\mathbb{T}[\![(\tau_1 \times \cdots \times \tau_n, \rho)]\!]]$

$\qquad (\texttt{newRGNRef}\ [\mathbb{I}[\![\rho]\!]]\ [\mathbb{T}[\![\tau_1 \times \cdots \times \tau_n]\!]]\ \mathbb{E}[\![\rho]\!]\ \langle x_1, \ldots, x_n \rangle)$

where x_i fresh

$$\mathbb{E}\left[\!\!\left[\frac{\Delta;\Gamma \vdash_{\text{exp}} e : (\tau_1 \times \cdots \times \tau_n, \rho), \pi \qquad \Delta \vdash_{\text{rr}} \pi \succeq \rho \qquad 0 \leq i \leq n}{\Delta;\Gamma \vdash_{\text{exp}} \texttt{sel}_i\ e : \tau_i, \pi}\right]\!\!\right] =$$

$\texttt{bindRGN}\ x{:}\mathbb{T}[\![(\tau_1 \times \cdots \times \tau_2, \rho)]\!] \Leftarrow \mathbb{E}[\![e]\!]\ ;$

$\texttt{bindRGN}\ y{:}\mathbb{T}[\![\tau_1 \times \cdots \times \tau_2]\!]$

$\qquad \Leftarrow \mathbb{E}[\![\pi \succeq \rho]\!]\ [\mathbb{T}[\![\tau_1 \times \cdots \times \tau_2]\!]]$

$\qquad\qquad (\texttt{readRGNRef}\ [\mathbb{I}[\![\rho]\!]]\ [\mathbb{T}[\![\tau_1 \times \cdots \times \tau_n]\!]]\ x)\ ;$

$\texttt{returnRGN}\ [\mathbb{I}[\![\pi]\!]]\ [\mathbb{T}[\![\tau_i]\!]]\ (\texttt{sel}_i\ y)$

where x, y, z fresh

$$\mathbb{E}\left[\!\!\left[\frac{\Delta;\Gamma \vdash_{\text{exp}} e_a : \tau_x, \pi \qquad \Delta;\Gamma, x{:}\tau_x \vdash_{\text{exp}} e_b : \tau, \pi}{\Delta;\Gamma \vdash_{\text{exp}} \texttt{let}\ x = e_a\ \texttt{in}\ e_b : \tau, \pi}\right]\!\!\right] =$$

$\texttt{bindRGN}\ x{:}\mathbb{T}[\![\tau_x]\!] \Leftarrow \mathbb{E}[\![e_a]\!]\ ;\ \mathbb{E}[\![e_b]\!]$

Figure 3.26: Translation from SEC to $\mathsf{F}^{\mathsf{RGN}}$ (terms (IV))

Translations yielding expressions

Expressions

$$\mathbb{E}\left[\!\!\left[\frac{\begin{array}{c}\Delta \vdash_{\text{type}} \tau \qquad \vdash_{\text{ctxt}} \Delta;\Gamma;\pi \\ \Delta, \varrho \succeq \{\pi\};\Gamma \vdash_{\text{exp}} e_b : \tau, \varrho\end{array}}{\Delta;\Gamma \vdash_{\text{exp}} \texttt{letregion}\ \varrho\ \texttt{in}\ e_b : \tau, \pi}\right]\!\!\right] =$$

$$\texttt{letRGN}\ [\![\pi]\!]\ [\![\tau]\!]\ (\Lambda\vartheta_\varrho.\,\lambda w_\varrho{:}\mathbb{T}[\![\varrho \succeq \{\pi\}]\!]\,.\,\lambda h_\varrho{:}\mathsf{RGNHnd}\ \vartheta_\varrho.\,\mathbb{E}[\![e_b]\!])$$

$$\mathbb{E}\left[\!\!\left[\frac{\Delta, \varrho \succeq \phi;\Gamma \vdash_{\text{exp}} u : \tau, \pi' \qquad \Delta \vdash_{\text{region}} \rho \qquad \Delta \vdash_{\text{rr}} \pi \succeq \rho}{\Delta;\Gamma \vdash_{\text{exp}} \Lambda\varrho \succeq \phi.^{\pi'}\, u\ \texttt{at}\ \rho : (\forall \varrho \succeq \phi.^{\pi'}\, \tau, \rho), \pi}\right]\!\!\right] =$$

$$\mathbb{E}[\![\pi \succeq \rho]\!]\ \ [\mathbb{T}[\![(\forall \varrho \succeq \phi.^{\pi'}\, \tau, \rho)]\!]]$$

$$(\texttt{newRGNRef}\ [\mathbb{I}[\![\rho]\!]]\ [\mathbb{T}[\![\forall \varrho \succeq \phi.^{\pi'}\, \tau]\!]]$$

$$\mathbb{E}[\![\rho]\!]\ \ (\Lambda\vartheta_\varrho.\,\lambda w_\varrho{:}\mathbb{T}[\![\varrho \succeq \phi]\!]\,.\,\lambda h_\varrho{:}\mathsf{RGNHnd}\ \vartheta_\varrho.\,\mathbb{E}[\![u]\!]))$$

$$\mathbb{E}\left[\!\!\left[\frac{\begin{array}{c}\Delta;\Gamma \vdash_{\text{exp}} e_f : (\forall \varrho \succeq \phi.^{\pi'_f}\, \tau, \rho_f), \pi \qquad \Delta \vdash_{\text{rr}} \pi \succeq \rho_f \\ \Delta \vdash_{\text{region}} \rho_a \qquad \Delta \vdash_{\text{re}} \rho_a \succeq \phi \qquad \Delta \vdash_{\text{rr}} \pi \succeq \pi'_f[\rho_a/\varrho]\end{array}}{\Delta;\Gamma \vdash_{\text{exp}} e_f\ [\rho_a] : \tau[\rho_a/\varrho], \pi}\right]\!\!\right] =$$

$$\texttt{bindRGN}\ f{:}\mathbb{T}\Big[\!\Big[(\forall \varrho \succeq \phi.^{\pi'_f}\, \tau, \rho_f)\Big]\!\Big] \Leftarrow \mathbb{E}[\![e_f]\!]\ ;$$

$$\texttt{bindRGN}\ g{:}\mathbb{T}\Big[\!\Big[\forall \varrho \succeq \phi.^{\pi'_f}\, \tau\Big]\!\Big]$$

$$\Leftarrow \mathbb{E}[\![\pi \succeq \rho_f]\!]\ \ [\mathbb{T}\Big[\!\Big[\forall \varrho \succeq \phi.^{\pi'_f}\, \tau\Big]\!\Big]]$$

$$(\texttt{readRGNRef}\ [\mathbb{I}[\![\rho_f]\!]]\ [\mathbb{T}\Big[\!\Big[\forall \varrho \succeq \phi.^{\pi'_f}\, \tau\Big]\!\Big]]\ f)\ ;$$

$$\mathbb{E}\Big[\!\Big[\pi \succeq \pi'_f[\rho_a/\varrho]\Big]\!\Big]\ \ [\mathbb{T}[\![\tau[\rho_a/\varrho]]\!]]\ (g\ [\mathbb{I}[\![\rho_a]\!]]\ \mathbb{E}[\![\rho_a \succeq \phi]\!]\ \ \mathbb{E}[\![\rho_a]\!])$$

where f, g fresh

Figure 3.27: Translation from SEC to $\mathsf{F}^{\mathsf{RGN}}$ (terms (V))

Translations yielding terms

Programs

$$\mathbb{E}\left[\!\!\left[\frac{\cdot, \mathcal{H} \succeq \{\}; \cdot \vdash_{\text{exp}} e : \mathsf{Bool}, \mathcal{H}}{\vdash_{\text{prog}} e}\right]\!\!\right] =$$

$$\begin{aligned}
&\texttt{runRGN}\ [\mathbb{T}[\![\cdot, \mathcal{H} \succeq \{\}; \cdot \vdash_{\text{type}} \mathsf{Bool}]\!]\\
&\quad (\Lambda\vartheta_{\mathcal{H}}.\, \lambda h_{\mathcal{H}}{:}\mathsf{RGNHnd}\ \vartheta_{\mathcal{H}}.\\
&\quad \texttt{let}\ w_{\mathcal{H}} = \langle\rangle\ \texttt{in}\\
&\quad \mathbb{E}[\![\cdot, \mathcal{H} \succeq \{\}; \cdot \vdash_{\text{exp}} p : \mathsf{Bool}, \mathcal{H}]\!])
\end{aligned}$$

Figure 3.28: Translation from SEC to $\mathsf{F}^{\mathsf{RGN}}$ (programs)

Function application, while notationally heavy, is simple. The `thenRGN` commands (implicit in the `bindRGN` expressions) sequence the evaluation of the function to a reference, the reading of the reference, the evaluation of the argument, and the application of the function to the argument.

The translation of `letregion` ϱ `in` e is pleasantly direct. We introduce ϑ_ϱ, h_ϱ, and w_ϱ through Λ- and λ-abstractions. The region handle and coercion function are supplied by the `letRGN` command when the computation is executed.

The translation of region abstraction is similar to the translation of functions. Once again, region handles and witness functions are λ-bound in accordance to the invariants described above. During the translation of region applications, the appropriate tuple of witness functions (constructed by $\mathbb{E}[\![\Delta \vdash_{\text{re}} \rho_2 \succeq \phi]\!]$) and region handle are supplied as arguments.

Programs Figure 3.28 shows the translation of SEC programs to $\mathsf{F}^{\mathsf{RGN}}$ expressions. An entire region computation is encapsulated and run by the `runRGN` ex-

pression. We bind $w_{\mathcal{H}}$ to an empty tuple, which corresponds to the absence of any coercion functions to the region $\mathcal{H}$.

3.3.1 Translation Properties

The translation is type preserving, as formalized by the following lemma. The proof is by (mutual) induction on the structure of the typing judgments, making frequent appeals to various well-formedness lemmas.

Lemma 3.1 (Translation Preserves Types)

(1) If $\vdash^{\mathsf{SEC}}_{\mathrm{rctxt}} \Delta$, *then* $\mathbb{D}[\![\Delta]\!]$ *is well-formed.*

(2) If $\Delta \vdash^{\mathsf{SEC}}_{\mathrm{region}} \rho$, *then* $\mathbb{D}[\![\vdash^{\mathsf{SEC}}_{\mathrm{rctxt}} \Delta]\!] \vdash^{\mathsf{F}^{\mathsf{RGN}}}_{\mathrm{index}} \mathbb{I}[\![\Delta \vdash^{\mathsf{SEC}}_{\mathrm{region}} \rho]\!]$.

(3) If $\vdash^{\mathsf{SEC}}_{\mathrm{rctxt}} \Delta$, *then* $\mathbb{D}[\![\vdash^{\mathsf{SEC}}_{\mathrm{rctxt}} \Delta]\!] \vdash^{\mathsf{F}^{\mathsf{RGN}}}_{\mathrm{vctxt}} \mathbb{G}[\![\vdash^{\mathsf{SEC}}_{\mathrm{rctxt}} \Delta]\!]$.

(4) If $\Delta \vdash^{\mathsf{SEC}}_{\mathrm{btype}} \omega$, *then* $\mathbb{D}[\![\vdash^{\mathsf{SEC}}_{\mathrm{rctxt}} \Delta]\!] \vdash^{\mathsf{F}^{\mathsf{RGN}}}_{\mathrm{type}} \mathbb{T}[\![\Delta \vdash^{\mathsf{SEC}}_{\mathrm{btype}} \omega]\!]$.

(5) If $\Delta \vdash^{\mathsf{SEC}}_{\mathrm{type}} \tau$, *then* $\mathbb{T}[\![\vdash^{\mathsf{SEC}}_{\mathrm{rctxt}} \Delta]\!] \vdash^{\mathsf{F}^{\mathsf{RGN}}}_{\mathrm{type}} \mathbb{T}[\![\Delta \vdash^{\mathsf{SEC}}_{\mathrm{type}} \tau]\!]$.

(6) If $\Delta \vdash^{\mathsf{SEC}}_{\mathrm{vctxt}} \Gamma$, *then* $\mathbb{D}[\![\vdash^{\mathsf{SEC}}_{\mathrm{rctxt}} \Delta]\!] \vdash^{\mathsf{F}^{\mathsf{RGN}}}_{\mathrm{vctxt}} \mathbb{G}[\![\Delta \vdash^{\mathsf{SEC}}_{\mathrm{vctxt}} \Gamma]\!]$.

(7) If $\Delta \vdash^{\mathsf{SEC}}_{\mathrm{vctxt}} \Gamma$, *then* $\mathbb{D}[\![\vdash^{\mathsf{SEC}}_{\mathrm{rctxt}} \Delta]\!] \vdash^{\mathsf{F}^{\mathsf{RGN}}}_{\mathrm{vctxt}} \mathbb{G}[\![\vdash^{\mathsf{SEC}}_{\mathrm{rctxt}} \Delta]\!], \mathbb{G}[\![\Delta \vdash^{\mathsf{SEC}}_{\mathrm{vctxt}} \Gamma]\!]$.

(8) If $\Delta \vdash^{\mathsf{SEC}}_{\mathrm{rr}} \rho_2 \succeq \rho_1$, *then* $\mathbb{D}[\![\vdash^{\mathsf{SEC}}_{\mathrm{rctxt}} \Delta]\!] \vdash^{\mathsf{F}^{\mathsf{RGN}}}_{\mathrm{type}} \mathbb{T}[\![\Delta \vdash^{\mathsf{SEC}}_{\mathrm{rr}} \rho_2 \succeq \rho_1]\!]$.

(9) If $\Delta \vdash^{\mathsf{SEC}}_{\mathrm{re}} \rho \succeq \phi$, *then* $\mathbb{D}[\![\vdash^{\mathsf{SEC}}_{\mathrm{rctxt}} \Delta]\!] \vdash^{\mathsf{F}^{\mathsf{RGN}}}_{\mathrm{type}} \mathbb{T}[\![\Delta \vdash^{\mathsf{SEC}}_{\mathrm{re}} \rho \succeq \phi]\!]$.

(10) If $\Delta \vdash^{\mathsf{SEC}}_{\mathrm{rr}} \rho_2 \succeq \rho_1$, *then*

$$\mathbb{D}[\![\vdash^{\mathsf{SEC}}_{\mathrm{rctxt}} \Delta]\!] ; \mathbb{G}[\![\vdash^{\mathsf{SEC}}_{\mathrm{rctxt}} \Delta]\!]$$
$$\vdash^{\mathsf{F}^{\mathsf{RGN}}}_{\mathrm{exp}} \mathbb{E}[\![\Delta \vdash^{\mathsf{SEC}}_{\mathrm{rr}} \rho_2 \succeq \rho_1]\!] : \mathbb{T}[\![\Delta \vdash^{\mathsf{SEC}}_{\mathrm{rr}} \rho_2 \succeq \rho_1]\!] .$$

(11) If $\Delta \vdash^{\mathsf{SEC}}_{\mathrm{re}} \rho \succeq \phi$, *then*

$$\mathbb{D}[\![\vdash^{\mathsf{SEC}}_{\mathrm{rctxt}} \Delta]\!] ; \mathbb{G}[\![\vdash^{\mathsf{SEC}}_{\mathrm{rctxt}} \Delta]\!]$$
$$\vdash^{\mathsf{F}^{\mathsf{RGN}}}_{\mathrm{type}} \mathbb{E}[\![\Delta \vdash^{\mathsf{SEC}}_{\mathrm{re}} \rho \succeq \phi]\!] : \mathbb{T}[\![\Delta \vdash^{\mathsf{SEC}}_{\mathrm{re}} \rho \succeq \phi]\!] .$$

(12) If $\Delta \vdash^{\mathsf{SEC}}_{\text{region}} \rho$, *then*

$$\mathbb{D}\llbracket \vdash^{\mathsf{SEC}}_{\text{rctxt}} \Delta \rrbracket \,;\, \mathbb{G}\llbracket \vdash^{\mathsf{SEC}}_{\text{rctxt}} \Delta \rrbracket$$
$$\vdash^{\mathsf{F}^{\mathsf{RGN}}}_{\text{exp}} \mathbb{E}\llbracket \Delta \vdash^{\mathsf{SEC}}_{\text{region}} \rho \rrbracket : \mathsf{RGNHnd}\ \mathbb{I}\llbracket \Delta \vdash^{\mathsf{SEC}}_{\text{region}} \rho \rrbracket.$$

(13) If $\Delta; \Gamma \vdash^{\mathsf{SEC}}_{\text{exp}} e : \tau, \pi$, *then*

$$\mathbb{D}\llbracket \vdash^{\mathsf{SEC}}_{\text{rctxt}} \Delta \rrbracket \,;\, \mathbb{G}\llbracket \vdash^{\mathsf{SEC}}_{\text{rctxt}} \Delta \rrbracket \,, \mathbb{G}\llbracket \Delta \vdash^{\mathsf{SEC}}_{\text{vctxt}} \Gamma \rrbracket$$
$$\vdash^{\mathsf{F}^{\mathsf{RGN}}}_{\text{exp}} \mathbb{E}\llbracket \Delta; \Gamma \vdash^{\mathsf{SEC}}_{\text{exp}} e : \tau, \pi \rrbracket : \mathsf{RGN}\ \mathbb{I}\llbracket \Delta \vdash^{\mathsf{SEC}}_{\text{region}} \pi \rrbracket\ \ \mathbb{T}\llbracket \Delta \vdash^{\mathsf{SEC}}_{\text{type}} \tau \rrbracket\,.$$

(14) If $\vdash^{\mathsf{SEC}}_{\text{prog}} p$, *then* $\cdot; \cdot \vdash^{\mathsf{F}^{\mathsf{RGN}}}_{\text{exp}} \mathbb{E}\llbracket \vdash^{\mathsf{SEC}}_{\text{prog}} p \rrbracket : \mathsf{Bool}$.

Furthermore, the translation is meaning preserving, with respect to the dynamic semantics of SEC and $\mathsf{F}^{\mathsf{RGN}}$, as formalized by the following theorem:

Theorem 3.2 (Translation Correctness (Programs))

If $\vdash^{\mathsf{SEC}}_{\text{prog}} e$ *and* $e \Downarrow^{\mathsf{SEC}}_{\text{prog}} \mathfrak{b}$ *and* $\mathbb{E}\llbracket \vdash^{\mathsf{SEC}}_{\text{prog}} e \rrbracket = e^{\dagger}$,
then $(\cdot; e^{\dagger}) \Downarrow^{\mathsf{F}^{\mathsf{RGN}}} \mathfrak{b}$.

The essence of this proof relies on a *coherence* lemma stating that the translation of SEC outlives relations to $\mathsf{F}^{\mathsf{RGN}}$ witness terms yields functions that are operationally equivalent to the identity function. Coherence is used throughout the proof of correctness to show that every evaluation derivation for the source can be simulated by a derivation involving the translation of the source.

We note that the proof is greatly simplified by using large-step operational semantics for both the source and target languages, since for many expression forms, a single operational step in the source language is expanded to many operational steps in the target language. Additional details concerning the translation from SEC to $\mathsf{F}^{\mathsf{RGN}}$ and the proof of correctness may be found in Appendix B.1.

3.4 Related Work

The work in this chapter draws heavily from two distinct lines of research. The first is the work done in type-and-effect systems for region-based memory management, introduced by Tofte and Talpin [79, 80] and explored by others [39, 10, 11], discussed in detail in the previous chapter.

The work of Banerjee, Heintze, and Riecke [5] deserves special mention. They show how to translate the Tofte-Talpin region calculus into an extension of the polymorphic λ-calculus called $\mathsf{F}_{\#}$. A new type operator # is used as a mechanism to hide and reveal the structure of types. Capabilities to allocate and read values from a region are explicitly passed as polymorphic functions of types $\forall\alpha.\alpha \to (\alpha\#\rho)$ and $\forall\alpha.(\alpha\#\rho) \to \alpha$; however, regions have no run-time significance in $\mathsf{F}_{\#}$ and there is no notion of deallocation upon exiting a region. The equality theory of types in $\mathsf{F}_{\#}$ is nontrivial, due to the treatment of #; in contrast, type equality on $\mathsf{F}^{\mathsf{RGN}}$ types is purely syntactic. Furthermore, their proof of soundness is based on denotational techniques, whereas ours are based on syntactic techniques which tend to scale more easily to other linguistic features. Finally, it is worth noting that there is almost certainly a connection between the $\mathsf{F}_{\#}$ `lift` and `seq` expressions and the monadic return and bind operations, although it is not mentioned or explored in their paper.

The second line of research on which we draw is the work done in monadic encapsulation of effects [61, 62, 69, 56, 87, 55, 57, 70, 4, 50, 72, 63, 88]. The majority of this work has focused on effects arising from reading and writing mutable state, which we reviewed in Section 3.1. While recent work [87, 63, 88] has considered more general combinations of effects and monads, only a small amount of work has examined the combination of regions and monads [48, 49, 23].

We note that Wadler and Thiemann [88] advocate marrying effects and monads by translating a type $\tau_1 \xrightarrow{\sigma} \tau_2$ to the type $\mathbb{T}[\![\tau_1]\!] \to \mathsf{T}^{\sigma}\ \mathbb{T}[\![\tau_2]\!]$, where $\mathsf{T}^{\sigma}\ \tau$ represents a computation that yields a value of type τ and has effects delimited by (the set) σ. As with the work of Banerjee *et al.*described above, this introduces a nontrivial theory of equality (and subtyping) on types; the types $\mathsf{T}^{\sigma}\ \tau$ and $\mathsf{T}^{\sigma'}\ \tau$ are equal so long as σ and σ' are equivalent sets. However, few programming languages allow one to express such nontrivial equalities between types.

Kagawa [48, 49] anticipates a number of themes from this work, although a formal treatment is left to future work. As a means of bridging the work of Wadler [86] and Launchbury and Peyton Jones [56], Kagawa [48] suggests extending the ST monad with the following type and operations:

$$\tau ::= \ldots \mid \mathsf{Mutable}\ \tau_s\ \tau_t$$

$$\begin{aligned}
\texttt{appR} &:: \forall s,t.\forall\alpha.\mathsf{Mutable}\ s\ t \to \mathsf{ST}\ t\ \alpha \to \mathsf{ST}\ s\ \alpha \\
\texttt{cmpR} &:: \forall s,t,u.\mathsf{Mutable}\ s\ t \to \mathsf{Mutable}\ t\ u \to \mathsf{Mutable}\ s\ u \\
\texttt{extendST} &:: \forall t.\forall\alpha.(\forall s.\mathsf{Mutable}\ s\ t \to \mathsf{ST}\ t\ \alpha) \to \mathsf{ST}\ t\ \alpha
\end{aligned}$$

The intention is that the type $\mathsf{Mutable}\ s\ t$ is equivalent to the type $(s \to t) \times (s \to t \to s)$; hence, it serves as a witness to the embedding of the state t into a larger state s. `extendST` generalizes `blockST` of Section 3.1 in the same manner as our `letRGN`. `appR` coerces a state transformer, given the appropriate witness, while `cmpR` composes witnesses; hence, the latter is a "proof" of the transitivity of the state embedding. In our setting, the transparency of the $\mathsf{RGNPf}(\theta_1 \preceq \theta_2)$ type obviates the need for these explicit operations. The lack of formal dynamic and static semantics makes a thorough evaluation difficult; in particular, the relationship between the global state "conjured up" by `runST` and an individual mutable object is rather *ad hoc*. In the abstract machine configurations for $\mathsf{F}^{\mathsf{RGN}}$,

a `witnessRGN` term concretely captures the relationship between an older and a younger region.

In later work, Kagawa [49] argues that these techniques can be extended to accommodate region-based memory management. In spite of the title and notation, the paper does not present an explicitly monadic language. Rather, the language is presented with a type-and-effect system, and the connection to a monadic setting is left (vaguely) implicit in the choice of notation and reference to the previous work. A dynamic semantics and type system, along with a proof that the new letextend operator can safely deallocate the extended region, is left to future work. Our present work addresses all these deficiencies by giving clear descriptions of both the Single Effect Calculus and $\mathsf{F}^{\mathsf{RGN}}$, proving the soundness of the $\mathsf{F}^{\mathsf{RGN}}$ type system, and giving a type- and meaning-preserving translation between the two languages. On the other hand, Kagawa presents a type inference algorithm for the language, which may suggest a means of reducing the notational overhead of passing witnesses and handles.

Ganz [23] relates the type-and-effect system of Tofte and Talpin to monad transformers. Ganz distinguishes among encapsulation with a single monad, encapsulation with a monad per region, and encapsulation with a monad transformer per region. He concludes that only a monad transformer per region is expressive enough to encode nested regions. This corresponds to our presentation where `runRGN` introduces a monad per stack of regions and `letRGN` introduces a monad transformer per region. Ganz imposes a peculiar restriction: upward references (i.e., allocating a reference to an inner region at an outer region) are not allowed. This is a severe restriction for a region-based language; it appears to arise from a failure to distinguish encapsulation of a stack of regions from encapsulation of

a single region. Recall that while `runRGN` computations may be nested, it is not possible for the outer computation to have references to the inner computation; on the other hand, there may be arbitrary references among regions of a single stack. Finally, Ganz claims to support early deallocation of regions, a facet of region-based memory management that is not available in $\mathsf{F}^{\mathsf{RGN}}$. However, in the following chapters, we will demonstrate how a substructural type system may be used to eliminate the restriction to lexically-scoped regions, thereby supporting the early deallocation of regions.

Finally, other researchers have utilized the power of System F as a target language. For example, Washburn and Weirich [95] demonstrate how to encode higher-order abstract syntax using parametric polymorphism, while Tse and Zdancewic [81] show how to encode the dependency core calculus.

3.5 Summary and Future Work

We have given a type- and meaning-preserving translation from the Single Effect Calculus to $\mathsf{F}^{\mathsf{RGN}}$. Both the source and the target languages use static type systems to delimit the effects of allocating in and reading from regions. The Single Effect Calculus uses the partial order implied by the "outlives" relation on regions to use single regions as bounds for sets of effects. We feel that this is an important insight that leads to a relatively straight-forward translation into the monadic setting. $\mathsf{F}^{\mathsf{RGN}}$ draws inspiration from the work on monadic encapsulation of state to give parametric types to `runRGN` and `letRGN` that prevent access of regions beyond their lifetimes. Explicit functions witness the outlives relationship between regions, enabling computations from outer regions to be cast to computations in inner regions. These witnesses cannot be forged and are only introduced via `letRGN`.

Recall that in Section 3.1, we briefly considered supporting region-based memory management using the operators `letRGN` and `liftRGN` with types analogous to `blockST` and `liftST` (i.e., using a product type). Pursuing this approach would require appropriately assembling evidence from `liftRGN` terms. While much of the development in this chapter could be accomplished using this alternative approach, we have presented an approach that fuses the two operations together in the `letRGN` operation, whereby witness functions are only introduced via `letRGN`. There are a number of reasons for this choice. First, the types are smaller than under the alternative scheme. Looking at Section 3.3, we trade the number of terms in scope for the size of the types in scope. Second, one is encouraged to write region polymorphic functions with the fused `letRGN`, whereas one can write region constrained functions with `liftRGN`. Third, `letRGN` makes it clear that the only witness functions are those that arise from entering a new region. Finally, although we have made the type $\mathsf{RGNPf}(r_1 \preceq r_2)$ a synonym for a witness function, we can imagine a scheme in which this primitive evidence is abstract and we provide additional operations for combining evidence and operations for taking evidence to functions for importing RGN computations or RGNRef references. The latter corresponds to pointer subtyping in Cyclone, where a pointer to region r_1 may be coerced to a pointer to region r_2 when r_1 outlives r_2. We explore this scheme further in Chapter 5.

There are numerous directions for future work. One idea is to provide the RGN monad to Haskell programmers and to try to leverage type classes so that witnesses and handles can be passed implicitly, thereby reducing the notational overhead of programming with nested stores. While a direct encoding of subtyping leads to undecidable and overlapping instances, the use of type-indexed products [52] may

provide a partial solution, at the expense of reintroducing a product type (see comments at the end of Section 3.1). Obviously, a language that incorporates subtyping directly, such as System $\mathsf{F}_{\leq}$, would simplify the encoding.

Finally, as is well known, Tofte and Talpin's original region calculus can lead to inefficient memory usage for some programs. In particular, the nested lifetimes of lexically-scoped regions make it impossible to destroy a region before the end of its lexical scope, even if the values in the region could be reclaimed without introducing program errors. We will shortly see that $\mathsf{F}^{\mathsf{RGN}}$ may be translated into the substructural type system of Chapter 4, which will eliminate the restriction of lexically-scoped regions.

Chapter 4

A Substructural Type System for Region-Based Memory Management

The $\mathsf{F}^{\mathsf{RGN}}$ language of the previous chapter demonstrated that parametric polymorphism and the technique of monadic encapsulation give rise to a simpler and more uniform language, as compared to the region calculi with type-and-effect systems, that nonetheless continues to provide the power and safety of region-based memory management. However, the nested lifetimes of lexically-scoped regions make it impossible to destroy a region before the end of its lexical scope. This can lead to inefficient memory usage for some programs.

Consider, for example, the following pseudo-code:

```
let fun loop (rold, dold) =
            let rnew = newrgn () in
            let dnew = copyData (rnew, dold) in
            freergn (rold);
            loop (rnew, dnew)     in
let r0 = newrgn () in
let d0 = initData (r0) in
loop (r0, d0)
```

Our intention is to define a function *loop*, which accepts a region *rold* and some data *dold* allocated in the region and copies the data into a new region *rnew*. After copying the data, neither the old data nor the old region are needed, so we would like to destroy the old region. Notice that this pseudo-code has introduced separate

operations for creating and destroying a region. It is not possible to translate this pseudo-code into a language with lexically-scoped regions, since the regions *rold* and *rnew* do not have nested lifetimes.

A language that supports the explicit creation and destruction of regions has some distinct advantages. The ability to dispense with nested lifetimes means that we can write programs that are more space efficient: such programs can destroy a region as soon as the region is no longer needed. However, as noted in Chapter 1, a programmer must be careful to never dereference a pointer into a region after the region has been destroyed; similarly, a programmer must be careful to avoid allocating in a destroyed region and destroying a region more than once. A type system that supports the explicit creation and destruction of regions, but rules out the various errors described above, would give rise to a more expressive language with (safe) region-based memory management. This chapter demonstrates that substructural type systems may be used to define just such a language.

The key insight is to revise the pseudo-code above into the following:

```
let fun loop (rold, dold) =
        let rnew = newrgn () in
        let (rold, rnew, dnew) = copyData (rold, rnew, dold) in
        freergn (rold');
        loop (rnew', dnew)    in
let r0 = newrgn () in
let (r0', d0) = initData (r0) in
loop (r0', d0)
```

We now require that the regions (*rold*, *rnew*, *r0*, etc.) serve as region capabilities. These region capabilities must be presented at each use of the region: for

allocating in the region, for reading from the region, and for destroying the region. Furthermore, we require that a region capability be used exactly once in the program. For operations that access, but don't destroy a region, like allocating in a region or reading from a region, a new capability is returned. Since our intention is that `initData` and `copyData` allocate in and read from (but do not destroy) regions, they also take region capabilities and return new region capabilities. On the other hand, `freergn` takes a region capability but returns no new capability. This ensures that the region may not be accessed after it is destroyed; in essence, the capability serves as a proof that the region is live. The simple "must use exactly once" constraint on region capabilities ensures the safety of the region operations. The substructural type system presented in this chapter naturally enforces this sort of constraint.

The work in this chapter was inspired by the work on linear type systems [85, 65], the Calculus of Capabilities [90], and Alias Types [75, 91]. At a high-level, each of these lines of research is concerned with the ways in which resources may be precisely accounted for in programs. We turn to these systems because they are designed to handle a broad range of resource usage scenarios; in particular, they accommodate acquiring, using, and releasing resources in non-LIFO order. By viewing a region as a managed resource, we are able to overcome the limitations of lexically-scoped regions. Two key insights contribute to the ability to view a region as a managed resource. First, from the work on the Calculus of Capabilities and Alias Types, we adopt the flexibility of separating the name of a region from the property that the region is allocated.[1] Second, from the work on linear types,

[1]Recall that in SEC and $\mathsf{F}^{\mathsf{RGN}}$, the lexically-scoped regions were allocated for precisely the scope of the corresponding region variable (ϱ introduced by `letregion` in SEC) or the corresponding index variable (ϑ introduced by `letRGN` in $\mathsf{F}^{\mathsf{RGN}}$).

we adopt a straightforward type system for controlling how the property that a region is live is used in the program.

In this chapter, we consider a substructural type system, which provides the necessary power to encode region calculi (and more) and back this claim by giving a translation from the $\mathsf{F}^{\mathsf{RGN}}$ language of the previous chapter to a region extension of a substructural lambda calculus, which we dub rgnURAL. The central element of the translation is to "break open" the RGN monad, exposing its interpretation as a region-stack transformer. In addition, a substructural type system turns out to be an effective means of managing individual references as independent resources; hence, rgnURAL provides a richer collection of reference primitives than that provided by SEC or $\mathsf{F}^{\mathsf{RGN}}$. In the subsequent chapter, we demonstrate that rgnURAL is a versatile target language by showing how to encode Cyclone's dynamic regions and unique pointers, as well as their interactions with lexically-scoped regions.

The remainder of this chapter is structured as follows. In the following section, we examine more closely the nature of substructural type systems. This motivates the design for rgnURAL, which is presented more formally in Section 4.2. Instead of a lexically-scoped region primitive, the primitives of rgnURAL include `newrgn` and `freergn` for separately creating and destroying a region. All access to a region (for allocating, reading, and writing references) is mediated by a capability that is produced by `newrgn` and consumed by `freergn`. In addition, the primitives of rgnURAL include `free` for deallocating an individual reference and `write` and `swap` for strong (type-varying) updates. Furthermore, rgnURAL adopts a relatively simple type system.[2]

[2]While the type system of rgnURAL is more complicated than that of $\mathsf{F}^{\mathsf{RGN}}$, we believe that it is simpler (that is, more intuitive) than that of SEC and other type-and-effect systems.

In Section 4.3, we show how $\mathsf{F}^{\mathsf{RGN}}$ can be translated to rgnURAL in a type- and meaning-preserving fashion, thereby establishing our claim that a substructural type system is sufficient for encoding the type-and-effects systems of region calculi.

In Sections 4.4 and 4.5, we consider related work and summarize and note directions for future work. Appendix C compliments this chapter by including technical details that would otherwise detract from the focus on the translation.

4.1 Background: A Substructural λ-Calculus

Advanced type systems for resources limit the order and number of uses of data and operations to ensure that resources are handled in a safe manner. For example, (safely) deallocating a data structure requires that the data structure is never used in the future.[3] In order to conservatively establish this property, a type system may ensure that the data structure is used *at most once*; after one use, the data structure may be safely deallocated, since there can be no further uses.[4]

A *substructural* type system provides the core mechanisms necessary to restrict the number and order of uses of data and operations. A conventional type system, such as that used by the simply-typed λ-calculus (and those used by SEC and $\mathsf{F}^{\mathsf{RGN}}$), with a typing judgment like $\Gamma \vdash e : \tau$, satisfies three structural properties:

Exchange If $\Gamma_1, x{:}\tau_x, y{:}\tau_y, \Gamma_2 \vdash e : \tau$, then $\Gamma_1, y{:}\tau_y, x{:}\tau_x, \Gamma_2 \vdash e : \tau$.

Contraction If $\Gamma_1, x{:}\tau_z, y{:}\tau_z, \Gamma_2 \vdash e : \tau$, then $\Gamma_1, z{:}\tau_z, \Gamma_2 \vdash e[z/x][z/y] : \tau$.

Weakening If $\Gamma \vdash e : \tau$, then $\Gamma, x{:}\tau_x \vdash e : \tau$.

[3] Similarly, and of particular importance in our setting, (safely) deallocating a region requires that the region is never accessed in the future.

[4] While safe, ensuring that a region is accessed *at most once* would be too limiting. In the next section, we show how a variation on this idea yields an expressive language.

The **Exchange** property asserts that the order of variables in the context does not affect the type checking of a term. The **Contraction** property asserts that if we can type check a term with multiple assumptions about variables of the same type, then we may also type check the same term with a single assumption about a variable of the type.[5] Finally, the **Weakening** property asserts that extra assumptions do not affect the type checking of a term.

In contrast, a substructural type system is designed so that one or more of these structural properties does not hold in general. Among the most widely studied substructural type systems are the *linear* type systems [85, 65], derived from Girard's linear logic [27], in which all variables satisfy **Exchange**, but linearly typed variables satisfy neither **Contraction** nor **Weakening**.

In this section, we informally present a *substructural* λ-calculus, similar to Walker's linear λ-calculus [89]. In our calculus, types are qualified as unrestricted (U), relevant (R), affine (A), or linear (L). All variables will satisfy **Exchange**, while only unrestricted variables will satisfy both **Contraction** and **Weakening**, allowing such variables to be used an arbitrary number of times. We will require

- linear variables to satisfy neither **Contraction** nor **Weakening**, ensuring that such variables are used exactly once,
- affine variables to satisfy **Weakening** (but not **Contraction**), ensuring that such variables are used at most once, and
- relevant variables to satisfy **Contraction** (but not **Weakening**), ensuring that such variables are used at least once.[6]

[5] Note, however, that while the two terms (e and $e[z/x][y/x]$) have the same type, they do not necessarily have the same behavior.

[6] In the logic community, it is perhaps more accurate to use the qualifier "strict"

The diagram below demonstrates the relationship between these qualifiers, inducing a partial order $\sqsubseteq$:

We integrate qualifiers into the simply-typed λ-calculus to yield a substructural λ-calculus, dubbed the λ^{URAL}-calculus, whose syntax is given in Figure 4.1.

Note that we structure our types τ as a qualifier $\mathfrak{q}$ applied to a pre-type $\overline{\tau}$, yielding the four sorts of types noted above. The qualifier of a type dictates the (implicit) structural operations that may be applied to values of the type, while the pre-type dictates the (explicit) introduction and elimination forms. The pre-types $\tau_1 \otimes \cdots \otimes \tau_n$ and $\tau_1 \multimap \tau_2$ correspond to the tuple and function types of the simply-typed λ-calculus. The types $^{\mathsf{U}}(^{\mathsf{U}}\overline{\tau_1} \otimes {}^{\mathsf{U}}\overline{\tau_2})$ and $^{\mathsf{U}}(^{\mathsf{U}}\overline{\tau_1} \multimap {}^{\mathsf{U}}\overline{\tau_2})$ directly correspond to the more familiar $\tau_1 \times \tau_2$ and $\tau_1 \to \tau_2$; see the discussion in Section 4.3.

This structuring of types as a qualifier applied to a pre-type follows that of Walker [89], but differs from other presentations of linear lambda calculi that use exactly one modality ($!\tau$) to distinguish unrestricted from linear types. While it seems possible to introduce alternative modalities (e.g, $-\tau$ for affine and $+\tau$ for relevant), we would have to consider their interaction (e.g., what does $-!+\tau$ denote?). Also, with four distinct qualifiers, it is natural to introduce qualifier polymorphism (as we do so in the next section), which is best formulated by separating qualifiers from pre-types.

Each pre-type has an associated introduction form; note, that a qualifier annotates the introduction form for all data structures. The qualifier annotation

for such variables. However, "strict" is already an overloaded term in the functional programming community; so, like Walker [89], we use "relevant."

Constant Qualifiers

$$\mathfrak{q} \quad \in \quad \mathit{CQuals} = \{\mathsf{U}, \mathsf{R}, \mathsf{A}, \mathsf{L}\}$$

Pre-types

$$\overline{\tau} \quad ::= \quad \overline{\mathsf{Bool}} \mid \tau_1 \multimap \tau_2 \mid \tau_1 \otimes \cdots \otimes \tau_n$$

Types

$$\tau \quad ::= \quad {}^{\mathfrak{q}}\overline{\tau}$$

Boolean constants

$$\mathfrak{b} \quad \in \quad \{\texttt{true}, \texttt{false}\}$$

Value variables

$$f, x \quad \in \quad \mathit{VVars}$$

Terms

$$\begin{aligned} e \quad ::= \quad & {}^{q}\mathfrak{b} \mid \texttt{if } e_b \texttt{ then } e_t \texttt{ else } e_f \mid \\ & x \mid {}^{q}\lambda x{:}\tau.\, e \mid e_1\ e_2 \mid \\ & {}^{q}\langle e_1, \ldots, e_n \rangle \mid \texttt{let } \langle x_1, \ldots, x_n \rangle = e_a \texttt{ in } e_b \mid \\ & \texttt{let } x = e_a \texttt{ in } e_b \end{aligned}$$

Figure 4.1: Syntax of λ^{URAL}

indicates the number of times that the data structure will be used (i.e., appear in an appropriate elimination form) during the evaluation of the program; a linear (L) qualified data structure will be used *exactly once*, an affine (A) *at most once*, a relevant (R) *at least once*, and an unrestricted (U) an arbitrary number of times.

A boolean is eliminated by the $\texttt{if}\ e_b\ \texttt{then}\ e_t\ \texttt{else}\ e_f$ expression. The pattern matching expression form $\texttt{let}\ \langle x_1, \ldots, x_n \rangle = e\ \texttt{in}\ e_b$ is used to eliminate tuples ($\otimes$).[7] Finally, a function with pre-type $\tau_1 \multimap \tau_2$ is eliminated via application $e_1\ e_2$.

While we present an operational semantics for (an extension of) λ^{URAL} in the next section and in Appendix C.1.1, it is important to note how the evaluation of a program may implicitly copy and/or drop values. Consider, for example, the following two terms:

$$^{\mathsf{U}}(\lambda x{:}^{\mathsf{L}}\overline{\tau}.\ ^{\mathsf{L}}\langle x, x\rangle)\ ^{\mathsf{L}}\overline{v}$$

$$^{\mathsf{U}}(\lambda x{:}^{\mathsf{L}}\overline{\tau}.\ ^{\mathsf{U}}\langle\rangle)\ ^{\mathsf{L}}\overline{v}$$

The substitution of $^{\mathsf{L}}\overline{v}$ for x in the evaluation of the first term will duplicated the $^{\mathsf{L}}\overline{v}$ value; if this term were embedded in a larger program, further evaluation might deconstruct the result pair and subsequently use the $^{\mathsf{L}}\overline{v}$ value more than once – violating the *exactly one use* of a linear value. Similarly, the substitution of $^{\mathsf{L}}\overline{v}$ for x in the second term will discard the $^{\mathsf{L}}\overline{v}$ value; no further evaluation will ever use the $^{\mathsf{L}}\overline{v}$ value – again, violating the *exactly one use* of a linear value.

Hence, the type system for λ^{URAL} should ensure that only unrestricted and relevant values are duplicated and only unrestricted and affine values are discarded. To prevent values from being implicitly copied or dropped when their containing value is duplicated or discarded, the type system must also ensure that a value with

[7] Note that this form of tuple elimination extracts all components of the tuple, while only counting as a single use of the tuple.

$\boxed{\vdash \tau \sqsubseteq \mathsf{q}'}$

$$\frac{\mathsf{q} \preceq \mathsf{q}'}{\vdash {}^{\mathsf{q}}\overline{\tau} \sqsubseteq \mathsf{q}'}$$

$\boxed{\vdash \Gamma \sqsubseteq \mathsf{q}'}$

$$\frac{}{\vdash \cdot \sqsubseteq \mathsf{q}'} \qquad \frac{\vdash \Gamma \sqsubseteq \mathsf{q}' \qquad \vdash \tau \sqsubseteq \mathsf{q}'}{\vdash \Gamma, x{:}\tau \sqsubseteq \mathsf{q}'}$$

Figure 4.2: Static semantics of λ^{URAL} ($\sqsubseteq$)

a qualifier lower in the partial order may not contain values with qualifiers higher in the partial order. For example, an affine (A) pair may not contain linear (L) components, since we could end up dropping the linear components by dropping the pair; hence, the type system must rule out expressions of type ${}^{\mathsf{A}}({}^{\mathsf{L}}\overline{\tau}_1 \otimes {}^{\mathsf{L}}\overline{\tau}_2)$.

Despite these requirements, the type system for λ^{URAL} is relatively simple. λ^{URAL} typing judgments have the form $\Gamma \vdash e : \tau$ and Figure 4.4 presents the λ^{URAL} typing rules. In order to ensure the correct relationship between a data structure and its components, we extend the partial order on constant qualifiers to types and contexts (see Figure 4.2).

As is usual in a substructural setting, the type system relies upon a judgment $\vdash \Gamma \rightsquigarrow \Gamma_1 \boxdot \Gamma_2$ that splits the assumptions in Γ between the contexts Γ_1 and Γ_2 (see Figure 4.3). Splitting the context is necessary to ensure that variables are used appropriately by sub-expressions. Note that $\boxdot$ ensures that an A or L assumption appears in exactly one sub-context. On the other hand, U and R assumptions may appear in both sub-contexts (via the Contr), corresponding to implicit duplication of the variables.

$\boxed{\vdash \Gamma \rightsquigarrow \Gamma_1 \boxdot \Gamma_2}$

$$\frac{}{\vdash \cdot \rightsquigarrow \cdot \boxdot \cdot} \qquad \frac{\vdash \Gamma \rightsquigarrow \Gamma_1 \boxdot \Gamma_2}{\vdash \Gamma, x{:}\tau \rightsquigarrow \Gamma_1, x{:}\tau \boxdot \Gamma_2} \qquad \frac{\vdash \Gamma \rightsquigarrow \Gamma_1 \boxdot \Gamma_2}{\vdash \Gamma, x{:}\tau \rightsquigarrow \Gamma_1 \boxdot \Gamma_2, x{:}\tau}$$

$$\textsc{Contr}\ \frac{\vdash \Gamma \rightsquigarrow \Gamma_1 \boxdot \Gamma_2 \qquad \vdash \tau \sqsubseteq \mathsf{R}}{\vdash \Gamma, x{:}\tau \rightsquigarrow \Gamma_1, x{:}\tau \boxdot \Gamma_2, x{:}\tau}$$

Figure 4.3: Static semantics of λ^{URAL} ($\boxdot$)

The rule for tuples is representative: the context is split by $\boxdot$ to type each of the tuple components, and the types of each component are bounded by the qualifier assigned to the tuple. Intuitively, each of the the L and A assumptions in the context is exclusively "owned" by exactly one of the tuple components. Likewise, in the rule for abstractions, the types of the free variables of Γ, which constitute the closure of the function, must be bounded by the qualifier assigned to the function. Note that the qualifier assigned to a function type is unrelated to the types of the argument and result; rather, it is related to the abstracted components that are used when the function is executed.

The Weak rule splits the context into a sub-context used to type the expression e and a discardable sub-context, consisting of U and A variables, that are not required to type the expression. Note that the rule Weak acts as a strengthened **Weakening** property, allowing an arbitrary number of U and A variables to be dropped at once. The corresponding strengthened **Contraction** property is incorporated into the judgment $\vdash \Gamma \rightsquigarrow \Gamma_1 \boxdot \Gamma_2$ (via the Contr rule), which allows an arbitrary number of U and R variables to be copied at once.

$\boxed{\Gamma \vdash_{\text{exp}} e : \tau}$

$$\frac{}{\cdot \vdash_{\text{exp}} {}^{\mathfrak{q}}\mathfrak{b} : {}^{\mathfrak{q}}\overline{\mathsf{Bool}}}$$

$$\frac{\vdash \Gamma \rightsquigarrow \Gamma_1 \boxdot \Gamma_2 \qquad \Gamma_1 \vdash_{\text{exp}} e_b : {}^{\mathfrak{q}}\overline{\mathsf{Bool}} \qquad \Gamma_2 \vdash_{\text{exp}} e_t : \tau \qquad \Gamma_2 \vdash_{\text{exp}} e_f : \tau}{\Gamma \vdash_{\text{exp}} \texttt{if}\ e_b\ \texttt{then}\ e_t\ \texttt{else}\ e_f : \tau}$$

$$\frac{}{\cdot, x{:}\tau \vdash_{\text{exp}} x : \tau} \qquad \frac{\vdash \Gamma \sqsubseteq \mathfrak{q} \qquad \Gamma, x{:}\tau_x \vdash_{\text{exp}} e : \tau}{\Gamma \vdash_{\text{exp}} {}^{\mathfrak{q}}(\lambda x{:}\tau_x.\, e) : {}^{\mathfrak{q}}(\tau_x \multimap \tau)}$$

$$\frac{\vdash \Gamma \rightsquigarrow \Gamma_f \boxdot \Gamma_a \qquad \Gamma_f \vdash_{\text{exp}} e_f : {}^{\mathfrak{q}}(\tau_x \multimap \tau) \qquad \Gamma_a \vdash_{\text{exp}} e_a : \tau_x}{\Gamma \vdash_{\text{exp}} e_f\ e_a : \tau}$$

$$\frac{\vdash \Gamma \rightsquigarrow \Gamma_1 \boxdot \cdots \boxdot \Gamma_n \qquad \Gamma_i \vdash_{\text{exp}} e_i : \tau_i \quad {}^{i \in 1 \ldots n} \qquad \vdash \tau_i \sqsubseteq \mathfrak{q} \quad {}^{i \in 1 \ldots n}}{\Gamma \vdash_{\text{exp}} {}^{\mathfrak{q}}\langle e_1, \ldots, e_n \rangle : {}^{\mathfrak{q}}(\tau_1 \otimes \cdots \otimes \tau_n)}$$

$$\frac{\vdash \Gamma \rightsquigarrow \Gamma_a \boxdot \Gamma_b \qquad \Gamma_a \vdash e_a : {}^{\mathfrak{q}}(\tau_1 \otimes \cdots \otimes \tau_n) \qquad \Gamma_b, x_1{:}\tau_1, \ldots, x_n{:}\tau_n \vdash_{\text{exp}} e_b : \tau}{\Gamma \vdash_{\text{exp}} \texttt{let}\ \langle x_1, \ldots, x_n \rangle = e_a\ \texttt{in}\ e_b : \tau}$$

$$\frac{\vdash \Gamma \rightsquigarrow \Gamma_a \boxdot \Gamma_b \qquad \Gamma_a \vdash_{\text{exp}} e_a : \tau_x \qquad \Gamma_b, x{:}\tau_x \vdash e_b : \tau}{\Gamma \vdash_{\text{exp}} \texttt{let}\ x = e_a\ \texttt{in}\ e_b : \tau}$$

Weak

$$\frac{\vdash \Gamma \rightsquigarrow \Gamma_1 \boxdot \Gamma_2 \qquad \vdash \Gamma_1 \sqsubseteq \mathsf{A} \qquad \Gamma_2 \vdash e : \tau}{\Gamma \vdash_{\text{exp}} e : \tau}$$

Figure 4.4: Static semantics of λ^{URAL} (expressions)

Finally, note that the rules for constants require the value context to be empty; this ensures that every variable is either explicitly used by an expression or implicitly dropped by the WEAK rule.

One may easily verify that the substructural type system for λ^{URAL} satisfies these variations of the structural properties:

λ^{URAL} **Exchange** If $\Gamma_1, x{:}\tau_x, y{:}\tau_y, \Gamma_2 \vdash e : \tau$, then $\Gamma_1, y{:}\tau_y, x{:}\tau_x, \Gamma_2 \vdash e : \tau$.

λ^{URAL} **Contraction** If $\Gamma_1, x{:}^{\mathfrak{q}_z}\overline{\tau}_z, y{:}^{\mathfrak{q}_z}\overline{\tau}_z, \Gamma_2 \vdash e : \tau$ and $\mathfrak{q}_z \sqsubseteq \mathsf{R}$, then $\Gamma_1, z{:}^{\mathfrak{q}_z}\overline{\tau}_z, \Gamma_2 \vdash e[z/x][z/y] : \tau$.

λ^{URAL} **Weakening** If $\Gamma \vdash e : \tau$ and $\mathfrak{q}_x \sqsubseteq \mathsf{A}$, then $\Gamma, x{:}^{\mathfrak{q}_x}\overline{\tau}_x \vdash e : \tau$.

4.2 The rgnURAL Language

The rgnURAL language is an extension of the λ^{URAL}-calculus of the previous section, adding universal and existential quantification and adding pre-types and operations for regions, handles, and references. As described in the previous section, the design of rgnURAL takes inspiration from the work on linear and substructural type systems [85, 65, 89], the Calculus of Capabilities [90], and Alias Types [75, 91]. Essentially, rgnURAL uses an explicit (linear) *capability* to mediate all access to a region (for allocating, reading, writing, swapping, and deallocating references).

In this section, we present the rgnURAL language in sufficient detail to describe the translation from $\mathsf{F}^{\mathsf{RGN}}$ to rgnURAL in Section 4.3. To this end, we include the syntax for both the surface language and abstract machine configurations, dynamic semantics for the abstract machine configurations, and static semantics for the surface language.

The dynamic semantics defines a small-step semantics, which defines an *evaluation relation* from *heaps of regions* and *expressions* to *heaps of regions* and *expressions*. In Section 4.2.2, we will discuss why it is convenient to adopt a small-step operational semantics for rgnURAL

We purposefully omit the static semantics for the abstract machine configurations (normally included for a syntactic proof of type soundness), since it requires a number of technical details that detract from the focus on the translation. Appendix C.1 includes additional technical details for the rgnURAL language and discusses a mechanically verified proof of type soundness.

4.2.1 Syntax of rgnURAL

Surface Syntax of rgnURAL

Figures 4.5 and 4.6 present the syntax of "surface programs" (that is, excluding syntax and semantic objects that will appear in the dynamic semantics) of rgnURAL. In the following sections, we explain and motivate the main constructs of rgnURAL.

Qualifiers, pre-types, types, and regions Types in rgnURAL are structured as a qualifier applied to a pre-type, just as in the λ^{URAL}-calculus. However, rgnURAL enriches the type structure by introducing qualifier variables, pre-type variables, and type variables, as well as pre-types for universal and existential quantification over each of these kinds of variables.[8]

[8] As an alternative to introducing separate pre-types for quantification over each of these kinds of variables, we could introduce multiple kinds, say Qual, PreType, and Type, and collapse the syntactic classes of qualifiers, types, and indices. However, while this simplifies the syntax of the language, it does not significantly simplify the meta-theory.

Constant Qualifiers

$$\mathfrak{q} \quad \in \quad \mathit{CQuals} \;=\; \{\mathsf{U}, \mathsf{R}, \mathsf{A}, \mathsf{L}\}$$

Qualifier variables

$$\xi \quad \in \quad \mathit{QVars}$$

Qualifiers

$$q \quad ::= \quad \xi \mid \mathfrak{q}$$

Pre-type variables

$$\overline{\alpha} \quad \in \quad \mathit{PTVars}$$

Pre-types

$$\begin{aligned}\overline{\tau} \quad ::= \quad & \overline{\alpha} \mid \overline{\mathsf{Int}} \mid \overline{\mathsf{Bool}} \mid \tau_1 \multimap \tau_2 \mid \tau_1 \otimes \cdots \otimes \tau_n \mid \\ & \overline{\forall}\xi.\,\tau \mid \overline{\exists}\xi.\,\tau \mid \overline{\forall}\overline{\alpha}.\,\tau \mid \overline{\exists}\overline{\alpha}.\,\tau \mid \overline{\forall}\alpha.\,\tau \mid \overline{\exists}\alpha.\,\tau \mid \\ & \overline{\mathsf{Ref}}\ \rho\ \tau \mid \overline{\mathsf{Hnd}}\ \rho \mid \overline{\mathsf{Cap}}\ \rho \mid \overline{\forall}\varrho.\,\tau \mid \overline{\exists}\varrho.\,\tau\end{aligned}$$

Type variables

$$\alpha \quad \in \quad \mathit{TVars}$$

Types

$$\tau \quad ::= \quad \alpha \mid {}^{q}\overline{\tau}$$

Region variables

$$\varrho \quad \in \quad \mathit{RVars}$$

Surface regions

$$\rho \quad ::= \quad \varrho$$

Figure 4.5: Surface syntax of rgnURAL (I)

Integer constants

$$\mathfrak{i} \quad \in \quad \mathbb{Z}$$

Boolean constants

$$\mathfrak{b} \quad \in \quad \{\texttt{true}, \texttt{false}\}$$

Value variables

$$f, x \quad \in \quad \mathit{VVars}$$

Surface terms

$$\begin{aligned} e \quad ::= \quad & {}^{q}\mathfrak{i} \mid {}^{q}(e_1 \oplus e_2) \mid {}^{q}(e_1 \oslash e_2) \mid {}^{q}\mathfrak{b} \mid \texttt{if } e_b \texttt{ then } e_t \texttt{ else } e_f \mid \\ & x \mid {}^{q}\lambda x{:}\tau.\, e \mid e_1\ e_2 \mid \\ & {}^{q}\langle e_1, \ldots, e_n \rangle \mid \texttt{let } \langle x_1, \ldots, x_n \rangle = e_a \texttt{ in } e_b \mid \\ & {}^{q}\Lambda\xi.\, e \mid e\ [q] \mid {}^{q}\texttt{pack}(q, e) \mid \texttt{let pack}(\xi, x) = e_a \texttt{ in } e_b \mid \\ & {}^{q}\Lambda\overline{\alpha}.\, e \mid e\ [\overline{\tau}] \mid {}^{q}\texttt{pack}(\overline{\tau}, e) \mid \texttt{let pack}(\overline{\alpha}, x) = e_a \texttt{ in } e_b \mid \\ & {}^{q}\Lambda\alpha.\, e \mid e\ [\tau] \mid {}^{q}\texttt{pack}(\tau, e) \mid \texttt{let pack}(\alpha, x) = e_a \texttt{ in } e_b \mid \\ & \texttt{let } x = e_a \texttt{ in } e_b \mid \\ & {}^{q_c,q_h}\texttt{newrgn} \mid \texttt{freergn } e_c\ e_h \mid \\ & {}^{q_r}\texttt{new } e_c\ e_h\ e_\star \mid \texttt{free } e_c\ e_r \mid \\ & \texttt{read } e_c\ e_r \mid \texttt{write } e_c\ e_r\ e_\star \mid \texttt{swap } e_c\ e_h\ e_\star \mid \\ & {}^{q}\Lambda\varrho.\, e \mid e\ [\rho] \mid {}^{q}\texttt{pack}(\rho, e) \mid \texttt{let pack}(\varrho, x) = e_a \texttt{ in } e_b \end{aligned}$$

Figure 4.6: Surface syntax of rgnURAL (II)

As was done in our previous languages, we introduce regions ρ as a distinguished syntactic object.[9] Universal and existential quantification over regions is provided by the pre-types $\overline{\forall}\varrho.\,\tau$ and $\overline{\exists}\varrho.\,\tau$.

The pre-types $\overline{\mathsf{Ref}}\ \rho\ \tau$ and $\overline{\mathsf{Hnd}}\ \rho$ are similar to the corresponding types in $\mathsf{F}^{\mathsf{RGN}}$; the former is the type of mutable references allocated in the region ρ and the latter is the type of handles for the region ρ. Finally, the pre-type $\overline{\mathsf{Cap}}\ \rho$ is the type of capabilities for accessing the region ρ. We shall shortly see how (linear) capabilities effectively mediate access to a region.

Terms As with types, terms in rgnURAL include those found in the λ^{URAL}-calculus; constants, arithmetic and boolean operations, function abstraction and application, tuple introduction and elimination, and the various forms of quantifier introduction and elimination are completely standard.

There are seven primitives that deal with regions and references. Instead of a lexically-scoped region primitive, the `newrgn` and `freergn` primitives separate the creation and destruction of a region. The `newrgn` is the introduction form for both $\overline{\mathsf{Hnd}}$ and $\overline{\mathsf{Cap}}$ values; hence, it is annotated with two qualifiers.

We also introduce primitives to allocate (`new`) and deallocate (`free`) references, as well as to read (`read`), write (`write`), and swap (`swap`) their contents. Not all of these operations can be safely performed with all sorts of references, as we discuss in Section 4.2.3. Finally, note that `new` is the introduction form for $\overline{\mathsf{Ref}}$ values; hence, it is annotated with a qualifier.

Pointer names

$\mathfrak{p} \quad \in \quad \mathit{PNames}$

Region names

$\mathfrak{r} \quad \in \quad \mathit{RNames}$

Regions

$\rho \quad ::= \quad \ldots \mid \mathfrak{r}$

Abstract machine terms

$e \quad ::= \quad \ldots \mid {}^{\mathfrak{q}_r}(\mathtt{ref}\ \mathfrak{r}\ \mathfrak{p}) \mid {}^{\mathfrak{q}_h}(\mathtt{hnd}\ \mathfrak{r}) \mid {}^{\mathfrak{q}_c}(\mathtt{cap})$

Abstract machine values

$v \quad ::= \quad {}^{q}\overline{v}$

Abstract machine pre-values

$\overline{v} \quad ::= \quad \mathfrak{i} \mid \mathfrak{b} \mid \lambda x{:}\tau.\, e \mid \langle v_1, \ldots, v_n \rangle \mid$

$\Lambda\xi.\, e \mid \mathtt{pack}(q, v) \mid \Lambda\overline{\alpha}.\, e \mid \mathtt{pack}(\overline{\tau}, v) \mid \Lambda\alpha.\, e \mid \mathtt{pack}(\tau, v) \mid$

$\mathtt{ref}\ \mathfrak{r}\ \mathfrak{p} \mid \mathtt{hnd}\ \mathfrak{r} \mid \mathtt{cap} \mid \Lambda\varrho.\, e \mid \mathtt{pack}(\rho, v)$

Figure 4.7: Abstract machine syntax of rgnURAL (I)

Abstract Machine Configurations for rgnURAL

Figures 4.7 and 4.8 present abstract machines configurations for rgnURAL, which extend the syntax of the previous section with semantic objects that appear in the operational semantics.

Region names and pointers are used to represent references to region allocated data. Unlike the SEC and $\mathsf{F}^{\mathsf{RGN}}$ languages, terms in rgnURAL do not distinguish references and handles to live regions from references and handles to dead regions; hence, there is no dead region ($\bullet$) syntactic form. We discuss the reasons for this formulation in more detail when we consider the dynamic semantics of rgnURAL in Section 4.2.2.

The abstract machine syntax adds three new expression forms. The expression $^{\mathsf{q}_r}(\texttt{ref}\ \mathfrak{r}\ \mathfrak{p})$ is the run-time representation of a $^{\mathsf{q}_r}(\overline{\mathsf{Ref}}\ \mathfrak{r}\ \tau)$; that is, it is the pointer reference associated with a region allocated value. Likewise, the expression $^{\mathsf{q}_h}(\texttt{hnd}\ \mathfrak{r})$ is the run-time representation of a $^{\mathsf{q}_h}(\overline{\mathsf{Hnd}}\ \mathfrak{r})$. Finally, the expression $^{\mathsf{q}_c}(\texttt{cap})$ is the run-time representation of a $^{\mathsf{q}_c}(\overline{\mathsf{Cap}}\ \mathfrak{r})$. Note that the expression form of a capability does not name the region; as will be seen in the next section, a capability has no run-time significance.[10]

Value forms in rgnURAL are structured as a qualifier applied to a pre-value, which mirrors the structuring of types.

Thus far, we have talked about region allocated data without discussing where such data is stored. Storable (i.e., closed) values are associated with regions R; regions are collected into heaps H. In order to support a syntactic proof of type soundness, the structure of regions and heaps includes some additional instrumen-

[9]Note that surface programs never require a region to be represented by anything other than a region variable.

[10]That is, they could be erased without affecting the evaluation of a program.

Regions

$$R \;::=\; \{\mathfrak{p}_1 \mapsto (\mathfrak{q}_1, v_1), \ldots, \mathfrak{p}_n \mapsto (\mathfrak{q}_n, v_n)\}$$

Region mark

$$v \;::=\; {}^{\mathfrak{q}}\texttt{live} \mid \texttt{dead}$$

Heaps

$$H \;::=\; \{\mathfrak{r}_1 \mapsto (v_1, R_1), \ldots, \mathfrak{r}_n \mapsto (v_n, R_n)\}$$

Abstract machine configurations

$$(H; e)$$

Figure 4.8: Abstract machine syntax of rgnURAL (II)

tation. A region R maps pointers $\mathfrak{p}$ to a pair of a qualifier $\mathfrak{q}$ and a value; the qualifier records the qualifier that annotated the `new` primitive that allocated the corresponding reference. A heap H maps region names $\mathfrak{r}$ to a pair of a region mark ν and a region; the region mark records whether the named region is allocated (${}^{\mathfrak{q}_c}\texttt{live}$) or deallocated (`dead`). The operational semantics of the next section will not allow the evaluation of a rgnURAL program to access a deallocated region. When a region is allocated, the region mark ${}^{\mathfrak{q}_c}\texttt{live}$ records the qualifier of the capability associated with the region.

The notation $H_1 \uplus H_2$ (respectively, $R_1 \uplus R_2$) denotes the disjoint union of the heaps H_1 and H_2 (respectively, the regions R_1 and R_2); the operation is undefined if the domains of H_1 and H_2 (respectively, R_1 and R_2) are not disjoint.

$\boxed{(H; e) \longmapsto (H'; e')}$

$$\frac{\mathfrak{i}_1 \oplus \mathfrak{i}_2 = \mathfrak{i}}{(H; {}^{\mathfrak{q}}({}^{-}\mathfrak{i}_1 \oplus {}^{-}\mathfrak{i}_2)) \longmapsto (H; {}^{\mathfrak{q}}\mathfrak{i})} \qquad \frac{\mathfrak{i}_1 \otimes \mathfrak{i}_2 = \mathfrak{b}}{(H; {}^{\mathfrak{q}}({}^{-}\mathfrak{i}_1 \otimes {}^{-}\mathfrak{i}_2)) \longmapsto (H; {}^{\mathfrak{q}}\mathfrak{b})}$$

$$\frac{}{(H; \texttt{if } {}^{-}\texttt{true then } e_t \texttt{ else } e_f) \longmapsto (H; e_t)}$$

$$\frac{}{(H; \texttt{if } {}^{-}\texttt{false then } e_t \texttt{ else } e_f) \longmapsto (H; e_f)}$$

$$\frac{}{(H; {}^{-}(\lambda x{:}\tau_x.\, e_b)\ v_a) \longmapsto (H; e_b[v_a/x])}$$

$$\frac{}{(H; \texttt{let } \langle x_1, \ldots, x_n \rangle = {}^{-}\langle v_1, \ldots, v_n \rangle \texttt{ in } e_b) \longmapsto (H; e_b[v_1/x_1] \cdots [v_n/x_n])}$$

Figure 4.9: Dynamic semantics of rgnURAL (expressions (I))

4.2.2 Dynamic Semantics of rgnURAL

An inductive judgment (Figures 4.9–4.13) defines the dynamic semantics. We state without proof that the dynamic semantics is deterministic; it is syntax-directed, taking $(H; e)$ configurations modulo α-conversion, including conversion of region names and pointers, which are (uniquely) bound in the heap H.

The judgment $(H; e) \longmapsto (H'; e')$ asserts that one step of evaluation of the closed expression e in heap H results in a new heap H' and new expression e'.

The rules for $(H; e) \longmapsto (H'; e')$ for expression forms other than the region and reference primitives are completely standard. Note that in each of these rules, the heap H is returned unchanged. We use evaluation contexts E (Figure 4.11) to lift the base rewriting rules to a standard, left-to-right, innermost-to-outermost, call-by-value interpretation of the language.

$\boxed{(H; e) \longmapsto (H'; e')}$

$$\frac{}{(H; \text{-}(\lambda\xi.\, e_b)\ [q_a]) \longmapsto (H; e_b[q_a/\xi])}$$

$$\frac{}{(H; \texttt{let pack}(\xi, x) = \text{-}\texttt{pack}(q, v_x)\ \texttt{in}\ e_b) \longmapsto (H; e_b[q/\xi][v_x/x])}$$

$$\frac{}{(H; \text{ꟾ}(\lambda\overline{\alpha}.\, e_b)\ [\overline{\tau}_a]) \longmapsto (H; e_b[\overline{\tau}_a/\overline{\alpha}])}$$

$$\frac{}{(H; \texttt{let pack}(\overline{\alpha}, x) = \text{-}\texttt{pack}(\overline{\tau}, v_x)\ \texttt{in}\ e_b) \longmapsto (H; e_b[\overline{\tau}/\overline{\alpha}][\text{ɬ}_x/x])}$$

$$\frac{}{(H; \text{-}(\lambda\alpha.\, e_b)\ [\tau_a]) \longmapsto (H; e_b[\tau_a/\alpha])}$$

$$\frac{}{(H; \texttt{let pack}(\alpha, x) = \text{-}\texttt{pack}(\tau, v_x)\ \texttt{in}\ e_b) \longmapsto (H; e_b[\tau/\alpha][v_x/x])}$$

$$\frac{}{(H; \texttt{let}\ x = v_x\ \texttt{in}\ e_b) \longmapsto (H; e_b[v_x/x])}$$

$$\frac{}{(H; \text{-}(\lambda\varrho.\, e_b)\ [\rho_a]) \longmapsto (H; e_b[\rho_a/\varrho])}$$

$$\frac{}{(H; \texttt{let pack}(\varrho, x) = \text{-}\texttt{pack}(\rho, v_x)\ \texttt{in}\ e_b) \longmapsto (H; e[\rho/\varrho][\text{ɬ}_x/x])}$$

Figure 4.10: Dynamic semantics of rgnURAL (expressions (II))

$\boxed{(H; e) \longmapsto (H'; e')}$

Evaluation contexts

$$
\begin{array}{rcl}
E & ::= & [\cdot] \mid {}^{q}(E_1 \oplus e_2) \mid {}^{q}(v_1 \oplus E_2) \mid {}^{q}(E_1 \otimes e_2) \mid {}^{q}(v_1 \otimes E_2) \mid \\
& & \texttt{if } E_b \texttt{ then } e_t \texttt{ else } e_f \mid \\
& & E_f\ e_a \mid v_f\ E_a \mid \\
& & {}^{q}\langle v_1, \ldots, E_i, \ldots, e_n \rangle \mid \texttt{let } \langle x_1, \ldots, x_n \rangle = E_1 \texttt{ in } e_2 \mid \\
& & E_f\ [q_a] \mid {}^{q}\texttt{pack}(q, E) \mid \texttt{let pack}(\xi, x) = E_a \texttt{ in } e_b \mid \\
& & E_f\ [\overline{\tau}_a] \mid {}^{q}\texttt{pack}(\overline{\tau}, E) \mid \texttt{let pack}(\overline{\alpha}, x) = E_a \texttt{ in } e_b \mid \\
& & E_f\ [\tau_a] \mid {}^{q}\texttt{pack}(\tau, E) \mid \texttt{let pack}(\alpha, x) = E_f \texttt{ in } e_a \mid \\
& & \texttt{let } x = E_a \texttt{ in } e_b \mid \\
& & \texttt{freergn } E_c\ e_h \mid \texttt{freergn } v_c\ E_h \mid \\
& & {}^{q}\texttt{new } E_c\ e_h\ e_a \mid {}^{q}\texttt{new } v_c\ E_h\ e_a \mid {}^{q}\texttt{new } v_c\ v_h\ E_a \mid \\
& & \texttt{free } E_c\ e_r \mid \texttt{free } v_c\ E_r \mid \\
& & \texttt{read } E_c\ e_r \mid \texttt{read } v_c\ E_r \mid \\
& & \texttt{write } E_c\ e_r\ e_a \mid \texttt{write } v_c\ E_r\ e_a \mid \texttt{write } v_c\ v_r\ E_a \mid \\
& & \texttt{swap } E_c\ e_r\ e_a \mid \texttt{swap } v_c\ E_r\ e_a \mid \texttt{swap } v_c\ v_r\ E_a \mid \\
& & E_f\ [\rho_a] \mid {}^{q}\texttt{pack}(\rho, E) \mid \texttt{let pack}(\varrho, x) = E_a \texttt{ in } e_b
\end{array}
$$

$$\frac{(H; e) \longmapsto (H'; e')}{(H; E[e]) \longmapsto (H'; E[e'])}$$

Figure 4.11: Dynamic semantics of rgnURAL (contexts)

$\boxed{(H; e) \longmapsto (H'; e')}$

$$\frac{\mathfrak{r} \notin dom(H)}{(H; {}^{q_c,q_h}\texttt{newrgn}) \longmapsto (H \uplus \{\mathfrak{r} \mapsto ({}^{q_c}\texttt{live}, \{\})\}; {}^{\mathsf{L}}\texttt{pack}(\mathfrak{r}, {}^{\mathsf{L}}\langle {}^{q_c}(\texttt{cap}), {}^{q_h}(\texttt{hnd}\ \mathfrak{r})\rangle))}$$

$$\frac{}{(H \uplus \{\mathfrak{r} \mapsto ({}^{q_c}\texttt{live}, R)\}; \texttt{freergn}\ {}^{q_c}(\texttt{cap})\ {}^{q_h}(\texttt{hnd}\ \mathfrak{r})) \longmapsto (H \uplus \{\mathfrak{r} \mapsto (\texttt{dead}, R)\}; {}^{\mathsf{L}}\langle\rangle)}$$

Figure 4.12: Dynamic semantics of rgnURAL (expressions (III))

The rules for $(H; e) \longmapsto (H'; e')$ for the region and reference primitives perform operations that side-effect the heap of regions. The rule for ${}^{q_c,q_h}\texttt{newrgn}$ allocates a new region in the heap by choosing a fresh region name $\mathfrak{r}$, extending the heap with an empty region (bound to $\mathfrak{r}$), and returning an existential package (that hides the name of the fresh region) containing the capability and a handle for the region. The rule for `freergn` "deallocates" a region in the heap by changing the region mark from ${}^{q_c}\texttt{live}$ to `dead`. Note that `freergn` requires both a capability and a handle, though only the handle is required to name the region to be deallocated. (Hence, the capability could be erased without affecting the evaluation of the `freergn` primitive.)

The rules for `new`, `free`, `read`, `write`, and `swap` all access a region to manipulate references. For the most part, `new`, `read`, and `write` behave as their counterparts in $\mathsf{F}^{\mathsf{RGN}}$. The major differences are (1) that each of the operations threads a ${}^{q_c}(\texttt{cap})$ value through the evaluation and (2) that the `read`, `write`, and `swap` operations return the ${}^{q_r}(\texttt{ref}\ \mathfrak{r}\ \mathfrak{p})$ argument. The capability is simply presented at each access of a region and returned to allow future access. (Note that, as with

$\boxed{(H; e) \longmapsto (H'; e')}$

$$\frac{\mathfrak{p} \notin dom(R)}{\begin{array}{l}(H \uplus \{\mathfrak{r} \mapsto ({}^{\mathfrak{q}_c}\texttt{live}, R)\}; {}^{\mathfrak{q}_r}\texttt{new}\ {}^{\mathfrak{q}_c}(\texttt{cap})\ {}^{\mathfrak{q}_h}(\texttt{hnd}\ \mathfrak{r})\ v_a) \longmapsto \\ (H \uplus \{\mathfrak{r} \mapsto ({}^{\mathfrak{q}_c}\texttt{live}, R \uplus \{\mathfrak{p} \mapsto (\mathfrak{q}_r, v)\})\}; {}^{\mathsf{L}}\langle {}^{\mathfrak{q}_c}(\texttt{cap}), {}^{\mathfrak{q}_r}(\texttt{ref}\ \mathfrak{r}\ \mathfrak{p})\rangle)\end{array}}$$

$$\frac{}{\begin{array}{l}(H \uplus \{\mathfrak{r} \mapsto ({}^{\mathfrak{q}_c}\texttt{live}, R \uplus \{\mathfrak{p} \mapsto (\mathfrak{q}_r, v)\})\}; \texttt{free}\ {}^{\mathfrak{q}_c}(\texttt{cap})\ {}^{\mathfrak{q}_r}(\texttt{ref}\ \mathfrak{r}\ \mathfrak{p})) \longmapsto \\ (H \uplus \{\mathfrak{r} \mapsto ({}^{\mathfrak{q}_c}\texttt{live}, R)\}; {}^{\mathsf{L}}\langle {}^{\mathfrak{q}_c}(\texttt{cap}), v\rangle)\end{array}}$$

$$\frac{}{\begin{array}{l}(H \uplus \{\mathfrak{r} \mapsto ({}^{\mathfrak{q}_c}\texttt{live}, R \uplus \{\mathfrak{p} \mapsto (\mathfrak{q}_r, v)\})\}; \texttt{read}\ {}^{\mathfrak{q}_c}(\texttt{cap})\ {}^{\mathfrak{q}_r}(\texttt{ref}\ \mathfrak{r}\ \mathfrak{p})) \longmapsto \\ (H \uplus \{\mathfrak{r} \mapsto ({}^{\mathfrak{q}_c}\texttt{live}, R \uplus \{\mathfrak{p} \mapsto (\mathfrak{q}_r, v)\})\}; {}^{\mathsf{L}}\langle {}^{\mathfrak{q}_c}(\texttt{cap}), v\rangle)\end{array}}$$

$$\frac{}{\begin{array}{l}(H \uplus \{\mathfrak{r} \mapsto ({}^{\mathfrak{q}_c}\texttt{live}, R \uplus \{\mathfrak{p} \mapsto (\mathfrak{q}_r, v)\})\}; \texttt{write}\ {}^{\mathfrak{q}_c}(\texttt{cap})\ {}^{\mathfrak{q}_r}(\texttt{ref}\ \mathfrak{r}\ \mathfrak{p})\ v_\star) \longmapsto \\ (H \uplus \{\mathfrak{r} \mapsto ({}^{\mathfrak{q}_c}\texttt{live}, R \uplus \{\mathfrak{p} \mapsto (\mathfrak{q}_r, v_\star)\})\}; {}^{\mathsf{L}}\langle {}^{\mathfrak{q}_c}(\texttt{cap}), {}^{\mathfrak{q}_r}(\texttt{ref}\ \mathfrak{r}\ \mathfrak{p})\rangle)\end{array}}$$

$$\frac{}{\begin{array}{l}(H \uplus \{\mathfrak{r} \mapsto ({}^{\mathfrak{q}_c}\texttt{live}, R \uplus \{\mathfrak{p} \mapsto (\mathfrak{q}_r, v)\})\}; \texttt{swap}\ {}^{\mathfrak{q}_c}(\texttt{cap})\ {}^{\mathfrak{q}_r}(\texttt{ref}\ \mathfrak{r}\ \mathfrak{p})\ v_\star) \longmapsto \\ (H \uplus \{\mathfrak{r} \mapsto ({}^{\mathfrak{q}_c}\texttt{live}, R \uplus \{\mathfrak{p} \mapsto (\mathfrak{q}_r, v_\star)\})\}; {}^{\mathsf{L}}\langle {}^{\mathfrak{q}_c}(\texttt{cap}), {}^{\mathfrak{q}_r}(\texttt{ref}\ \mathfrak{r}\ \mathfrak{p}), v\rangle)\end{array}}$$

Figure 4.13: Dynamic semantics of rgnURAL (expressions (IV))

`freergn`, the capability could be erased without affecting the evaluation of the reference primitives.) Similarly, we do not wish to consider reading or writing a linear (respectively, affine) reference as the *exactly one use* (respectively, *at least one use*) of the reference. Therefore, the primitives return the reference that was accessed, so that it remains available for future use.

The rules for ${}^{\mathsf{q}_r}(\texttt{new}\ e_c\ e_h\ e_\star)$ and $\texttt{free}\ e_c\ e_r$ perform the complementary actions of allocating and deallocating a mutable reference in a region in the heap. The rule for `new` requires a handle as a run-time value holding the data necessary to allocate values within a region. The rule for `free` deallocates a reference by removing the pointer $\mathfrak{p}$ mapping from the region; note that `free` returns the value previously stored at $\mathfrak{p}$.

The primitives for reading and writing a mutable reference *implicitly* duplicate and discard (respectively) the contents of the reference. Note that the rule for `read` duplicates the value stored at $\mathfrak{p}$, by returning v, but also leaving the value stored at $\mathfrak{p}$ unchanged. Meanwhile, the rule for `write` discards the value stored at $\mathfrak{p}$, by replacing the value stored at $\mathfrak{p}$ with a new value.

The rule for `swap` combines the operations of dereferencing and updating a mutable reference, but has the attractive property that it neither duplicates nor discards a value. Note that performing a write or swap operation on a reference may change the type of the reference's contents. The static semantics of the next section will permit weak (type-invariant) updates on all references (with some additional caveats), but will restrict strong (type-varying) updates to unique references.

It is important to note that the execution of `freergn` and the reference primitives is predicated upon the primitive's capability and arguments corresponding to a live region in the heap. While it will be possible to have (suspended) primitives

that reference deallocated regions, it will not be possible to execute them. The type system of the next section and Appendix C.1.2 ensures that these invariants are preserved during the execution of well-typed programs.

Remarks

Before turning to the substructural type system for rgnURAL, it is worth considering in more detail one of the major differences in the abstract machine configurations and dynamic semantics of rgnURAL as compared to those of SEC and $\mathsf{F}^{\mathsf{RGN}}$. In particular, rgnURAL "deallocates" a region in the heap by changing the region mark from $^{\mathsf{q_c}}$`live` to `dead`, while SEC and $\mathsf{F}^{\mathsf{RGN}}$ deallocate a region by removing it from the stack of regions. SEC and $\mathsf{F}^{\mathsf{RGN}}$ also replace occurrences of the deallocated region name with $\bullet$ in the stack and result value; this replacement ensured that any occurrences of `ref` or `hnd` terms in the stack or result are marked as dead.

The reason that we must adopt a different strategy for dealing with terms that mention the deallocated region is simple: in rgnURAL, occurrences of the region name need not be local to the primitive that deallocates the region. This is in contrast to a language that provides lexically-scoped regions, such as $\mathsf{F}^{\mathsf{RGN}}$, in which occurrences of the region name are limited to the stack of regions and the result value of the `letRGN` command.

In moving to a language that provides separate primitives for allocating and deallocating a region, we gain flexibility, but it is significantly more difficult to track down occurrences of the region name in the rgnURAL program under evaluation. Consider, for example, the following program, which is well-typed according to the

static semantics of the next section:

$$\begin{array}{l}
\texttt{let pack}(\varrho, ch) = {}^{\mathsf{L},\mathsf{U}}\texttt{newrgn in} \\
\texttt{let } \langle cap, hnd \rangle = ch \texttt{ in} \\
\texttt{let } \langle cap, r_1 \rangle = {}^{\mathsf{U}}(\texttt{ref } cap\ hnd\ {}^{\mathsf{U}}1) \texttt{ in} \\
\texttt{let } \langle cap, r_2 \rangle = {}^{\mathsf{U}}(\texttt{ref } cap\ hnd\ {}^{\mathsf{U}}2) \texttt{ in} \\
\texttt{let } \langle\rangle = \texttt{freergn } cap\ hnd \texttt{ in} \\
\texttt{let } p = {}^{\mathsf{U}}\langle r_1, r_2 \rangle \texttt{ in } {}^{\mathsf{U}}7
\end{array}$$

After seven steps of evaluation, the program will be in the following configuration:

$$\begin{array}{l}
(\ \{\mathfrak{r} \mapsto ({}^{\mathsf{L}}\texttt{live}, \{\mathfrak{p}_1 \mapsto (\mathsf{U}, {}^{\mathsf{U}}1), \mathfrak{p}_2 \mapsto (\mathsf{U}, {}^{\mathsf{U}}2)\})\}; \\
\quad \texttt{let } \langle\rangle = \texttt{freergn } {}^{\mathsf{L}}(\texttt{cap})\ {}^{\mathsf{U}}(\texttt{hnd } \mathfrak{r}) \texttt{ in} \\
\quad \texttt{let } p = {}^{\mathsf{U}}\langle {}^{\mathsf{U}}(\texttt{ref } \mathfrak{r}\ \mathfrak{p}_1), {}^{\mathsf{U}}(\texttt{ref } \mathfrak{r}\ \mathfrak{p}_2) \rangle \texttt{ in } {}^{\mathsf{U}}7\ )
\end{array}$$

In order to maintain a compositional evaluation strategy, the rule that evaluates the `freergn` primitive must focus on exactly the `freergn` term and leave the remainder of the program unchanged. Hence, the occurrences of $\mathfrak{r}$ in the `let` $p = \ldots$ expression cannot be modified by the `freergn` rule. (The same holds true even if we were to adopt a large-step operational semantics.)

After two more steps of evaluation, the program will be in the following configuration:

$$\begin{array}{l}
(\ \{\mathfrak{r} \mapsto (\texttt{dead}, \{\mathfrak{p}_1 \mapsto (\mathsf{U}, {}^{\mathsf{U}}1), \mathfrak{p}_2 \mapsto (\mathsf{U}, {}^{\mathsf{U}}2)\})\}; \\
\quad \texttt{let } p = {}^{\mathsf{U}}\langle {}^{\mathsf{U}}(\texttt{ref } \mathfrak{r}\ \mathfrak{p}_1), {}^{\mathsf{U}}(\texttt{ref } \mathfrak{r}\ \mathfrak{p}_2) \rangle \texttt{ in } {}^{\mathsf{U}}7\ )
\end{array}$$

We may consider the references in the `let` $p = \ldots$ expression to be dangling pointers, since their corresponding region has been "deallocated." We leave the region $\mathfrak{r}$ mapping in the heap (but marked as dead) to prevent such dangling pointers from becoming associated with a later allocated region, one that happens to choose the same region name.

Qualifier, pre-type, type, region contexts

$$\Delta \quad ::= \quad \cdot \mid \Delta, \xi \mid \Delta, \overline{\alpha} \mid \Delta, \alpha \mid \Delta, \varrho$$

Value contexts $\quad \Gamma \quad ::= \quad \cdot \mid \Gamma, x{:}\tau$

Figure 4.14: Static semantics of rgnURAL (definitions)

4.2.3 Static Semantics of rgnURAL

As noted above, well-typed programs obey several invariants, which are enforced with typing judgments. In particular, the typing judgments for an rgnURAL expression must ensure that the evaluation of the expression does not attempt to access a deallocated region. For the surface syntax of rgnURAL, it suffices to include typing judgments for expressions and various well-formedness judgments for qualifiers, pre-types, types, regions, and contexts. As was stated previously, we purposefully omit judgments for the additional semantic objects introduced by the abstract machine configurations for rgnURAL (for example, typing judgments for heaps), but these additional technical details and a syntactic proof of type soundness for rgnURAL may be found in Appendix C.1.

Definitions Figure 4.14 presents additional definitions for syntactic objects that appear in the static semantics. Contexts Δ are ordered lists of qualifier, pre-type, type, and region variables and contexts Γ are ordered lists of variables with types. We tacitly assume that all contexts are well-formed: Δ contains distinct qualifier, pre-type, type, and region variables and Γ contains distinct value variables.

Qualifier order As in the λ^{URAL}-calculus, in order to ensure the correct relationship between a data structure and its components, we extend the partial order on constant qualifiers to arbitrary qualifiers, types, and value contexts (see Fig-

$\boxed{\Delta \vdash q \preceq q'}$

$$\frac{\Delta \vdash_{\text{qual}} q}{\Delta \vdash \mathsf{U} \preceq q} \qquad \frac{\mathfrak{q}_1 \sqsubseteq \mathfrak{q}_2}{\Delta \vdash \mathfrak{q}_1 \preceq \mathfrak{q}_2} \qquad \frac{\Delta \vdash_{\text{qual}} q}{\Delta \vdash q \preceq \mathsf{L}}$$

$$\frac{\Delta \vdash_{\text{qual}} q}{\Delta \vdash q \preceq q} \qquad \frac{\Delta \vdash q_1 \preceq q_2 \qquad \Delta \vdash q_2 \preceq q_3}{\Delta \vdash q_1 \preceq q_2}$$

$\boxed{\Delta \vdash \tau \preceq q'}$

$$\frac{\Delta \vdash_{\text{type}} \tau}{\Delta \vdash \tau \preceq \mathsf{L}} \qquad \frac{\Delta \vdash q \preceq q' \qquad \Delta \vdash_{\text{ptype}} \overline{\tau}}{\Delta \vdash {}^{q}\overline{\tau} \preceq q'}$$

$\boxed{\Delta \vdash \Gamma \preceq q'}$

$$\frac{\Delta \vdash_{\text{qual}} q'}{\Delta \vdash \cdot \preceq q'} \qquad \frac{\Delta \vdash \tau \preceq q' \qquad \Delta \vdash \Gamma \preceq q'}{\Delta \vdash \Gamma, x{:}\tau \preceq q'}$$

Figure 4.15: Static semantics of rgnURAL ($\preceq$)

$\boxed{\Delta \vdash \Gamma \rightsquigarrow \Gamma_1 \boxdot \Gamma_2}$

$$\frac{}{\Delta \vdash \cdot \rightsquigarrow \cdot \boxdot \cdot} \qquad \frac{\Delta \vdash \Gamma \rightsquigarrow \Gamma_1 \boxdot \Gamma_2}{\Delta \vdash \Gamma, x{:}\tau \rightsquigarrow \Gamma_1, x{:}\tau \boxdot \Gamma_2} \qquad \frac{\Delta \vdash \Gamma \rightsquigarrow \Gamma_1 \boxdot \Gamma_2}{\Delta \vdash \Gamma, x{:}\tau \rightsquigarrow \Gamma_1 \boxdot \Gamma_2, x{:}\tau}$$

$$\textsc{Contr}\ \frac{\Delta \vdash \Gamma \rightsquigarrow \Gamma_1 \boxdot \Gamma_2 \qquad \Delta \vdash \tau \preceq \mathsf{R}}{\Delta \vdash \Gamma, x{:}\tau \rightsquigarrow \Gamma_1, x{:}\tau \boxdot \Gamma_2, x{:}\tau}$$

Figure 4.16: Static semantics of rgnURAL ($\boxdot$)

ure 4.15). In the presence of qualifier and type quantification, we include the rules $\Delta \vdash \mathsf{U} \preceq q$, $\Delta \vdash q \preceq \mathsf{L}$, and $\Delta \vdash \tau \preceq \mathsf{L}$, a conservative extension, since U and L are the bottom and top of the lattice. These extensions are useful, since they admit the derivation of $\Delta \vdash \alpha \preceq \mathsf{L}$, which, for example, allows one to always make a linear pair of polymorphic values. A more general approach would incorporate bounded qualifier constraints, which we believe is straightforward, but doing so does not add to the discussion at hand.

Context splitting Figure 4.16 recalls the context splitting judgment of the λ^{URAL}-calculus, extending it in the presence of qualifier and type quantification.

Terms Figures 4.17–4.24 present the typing rules for the judgment $\Delta; \Gamma \vdash_{\text{exp}} e : \tau$, which asserts that under the qualifier, pre-type, type, and region context Δ and the value context Γ, the expression e has the type τ.

The rules for constants, arithmetic and boolean operations, function abstraction and application, tuple introduction and projection, and the various forms of quantifier introduction and elimination are all straightforward, following directly

$\boxed{\Delta; \Gamma \vdash_{\text{exp}} e : \tau}$

$$\frac{\Delta \vdash_{\text{qual}} q}{\Delta; \cdot \vdash_{\text{exp}} {}^{q}\mathtt{i} : {}^{q}\overline{\mathsf{Int}}}$$

$$\frac{\Delta \vdash_{\text{qual}} q \qquad \Delta \vdash \Gamma \rightsquigarrow \Gamma_1 \boxdot \Gamma_2 \qquad \Delta; \Gamma_1 \vdash_{\text{exp}} e_1 : {}^{q_1}\overline{\mathsf{Int}} \qquad \Delta; \Gamma_2 \vdash_{\text{exp}} e_2 : {}^{q_2}\overline{\mathsf{Int}}}{\Delta; \Gamma \vdash_{\text{exp}} {}^{q}(e_1 \oplus e_2) : {}^{q}\overline{\mathsf{Int}}}$$

$$\frac{\Delta \vdash_{\text{qual}} q \qquad \Delta \vdash \Gamma \rightsquigarrow \Gamma_1 \boxdot \Gamma_2 \qquad \Delta; \Gamma_1 \vdash_{\text{exp}} e_1 : {}^{q_1}\overline{\mathsf{Int}} \qquad \Delta; \Gamma_2 \vdash_{\text{exp}} e_2 : {}^{q_2}\overline{\mathsf{Int}}}{\Delta; \Gamma \vdash_{\text{exp}} {}^{q}(e_1 \oslash e_2) : {}^{q}\overline{\mathsf{Bool}}}$$

$$\frac{\Delta \vdash_{\text{qual}} q}{\Delta; \cdot \vdash_{\text{exp}} {}^{q}\mathfrak{b} : {}^{q}\overline{\mathsf{Bool}}}$$

$$\frac{\Delta \vdash \Gamma \rightsquigarrow \Gamma_1 \boxdot \Gamma_2 \qquad \Delta; \Gamma_1 \vdash_{\text{exp}} e_b : {}^{q}\overline{\mathsf{Bool}} \qquad \Delta; \Gamma_2 \vdash_{\text{exp}} e_t : \tau \qquad \Delta; \Gamma_2 \vdash_{\text{exp}} e_f : \tau}{\Delta; \Gamma \vdash_{\text{exp}} \texttt{if } e_b \texttt{ then } e_t \texttt{ else } e_f : \tau}$$

Figure 4.17: Static semantics of rgnURAL (expressions (I))

$\boxed{\Delta; \Gamma \vdash_{\text{exp}} e : \tau}$

$$\frac{\Delta \vdash_{\text{type}} \tau}{\Delta; \cdot, x{:}\tau \vdash_{\text{exp}} x : \tau}$$

Weak

$$\frac{\Delta \vdash \Gamma \rightsquigarrow \Gamma_1 \boxdot \Gamma_2 \qquad \Delta \vdash \Gamma_1 \preceq \mathsf{A} \qquad \Delta; \Gamma_2 \vdash e : \tau}{\Delta; \Gamma \vdash_{\text{exp}} e : \tau}$$

$$\frac{\Delta \vdash_{\text{qual}} q \qquad \Delta \vdash \Gamma \preceq q \qquad \Delta; \Gamma, x{:}\tau_x \vdash_{\text{exp}} e : \tau}{\Delta; \Gamma \vdash_{\text{exp}} {}^{q}\lambda x{:}\tau_x.\, e : {}^{q}(\tau_x \multimap \tau)}$$

$$\frac{\begin{array}{c}\Delta \vdash \Gamma \rightsquigarrow \Gamma_f \boxdot \Gamma_a \\ \Delta; \Gamma_f \vdash_{\text{exp}} e_f : {}^{q}(\tau_x \multimap \tau) \qquad \Delta; \Gamma_a \vdash_{\text{exp}} e_a : \tau_x\end{array}}{\Delta; \Gamma \vdash_{\text{exp}} e_f\ e_a : \tau}$$

$$\frac{\begin{array}{c}\Delta \vdash_{\text{qual}} q \qquad \Delta \vdash \Gamma \rightsquigarrow \Gamma_1 \boxdot \cdots \boxdot \Gamma_n \\ \Delta; \Gamma_i \vdash_{\text{exp}} e_i : \tau_i \quad {}^{i \in 1 \ldots n} \qquad \Delta \vdash \tau_i \preceq q \quad {}^{i \in 1 \ldots n}\end{array}}{\Delta; \Gamma \vdash_{\text{exp}} {}^{q}\langle e_1, \ldots, e_n \rangle : {}^{q}(\tau_1 \otimes \cdots \otimes \tau_n)}$$

$$\frac{\begin{array}{c}\Delta \vdash \Gamma \rightsquigarrow \Gamma_a \boxdot \Gamma_b \\ \Delta; \Gamma_a \vdash e_a : {}^{q}(\tau_1 \otimes \cdots \otimes \tau_n) \qquad \Delta; \Gamma_b, x_1{:}\tau_1, \ldots, x_n{:}\tau_n \vdash_{\text{exp}} e_b : \tau\end{array}}{\Delta; \Gamma \vdash_{\text{exp}} \texttt{let}\ \langle x_1, \ldots, x_n \rangle = e_a\ \texttt{in}\ e_b : \tau}$$

Figure 4.18: Static semantics of rgnURAL (expressions (II))

$\boxed{\Delta;\Gamma \vdash_{\text{exp}} e : \tau}$

$$\frac{\Delta \vdash_{\text{qual}} q \qquad \Delta \vdash \Gamma \preceq q \qquad \Delta,\xi;\Gamma \vdash_{\text{exp}} e : \tau}{\Delta;\Gamma \vdash_{\text{exp}} {}^{q}\Lambda\xi.\, e : {}^{q}(\overline{\forall}\xi.\, \tau)} \qquad \frac{\Delta;\Gamma \vdash_{\text{exp}} e_f : {}^{q}(\overline{\forall}\xi.\, \tau) \qquad \Delta \vdash_{\text{qual}} q_a}{\Delta;\Gamma \vdash_{\text{exp}} e_f\ [q_a] : \tau[q_a/\xi]}$$

$$\frac{\Delta \vdash_{\text{qual}} q \qquad \Delta;\Gamma \vdash_{\text{exp}} e_2 : \tau[q_1/\xi] \qquad \Delta \vdash \tau[q_1/\xi] \preceq q}{\Delta;\Gamma \vdash_{\text{exp}} {}^{q}\texttt{pack}(q_1, e_2) : {}^{q}(\overline{\exists}\xi.\, \tau)}$$

$$\frac{\Delta \vdash \Gamma \rightsquigarrow \Gamma_a \boxdot \Gamma_b \qquad \Delta \vdash_{\text{type}} \tau \qquad \Delta;\Gamma_a \vdash e_a : {}^{q}(\overline{\exists}\xi.\, \tau_x) \qquad \Delta,\xi;\Gamma_b, x{:}\tau_x \vdash e_b : \tau}{\Delta;\Gamma \vdash_{\text{exp}} \texttt{let pack}(\xi, x) = e_a\ \texttt{in}\ e_b : \tau}$$

$$\frac{\Delta \vdash_{\text{qual}} q \qquad \Delta \vdash \Gamma \preceq q \qquad \Delta,\overline{\alpha};\Gamma \vdash_{\text{exp}} e : \tau}{\Delta;\Gamma \vdash_{\text{exp}} {}^{q}\Lambda\overline{\alpha}.\, e : {}^{q}(\overline{\forall}\overline{\alpha}.\, \tau)} \qquad \frac{\Delta;\Gamma \vdash_{\text{exp}} e_f : {}^{q}(\overline{\forall}\overline{\alpha}.\, \tau) \qquad \Delta \vdash_{\text{ptype}} \overline{\tau}_a}{\Delta;\Gamma \vdash_{\text{exp}} e_f\ [\overline{\tau}_a] : \tau[\overline{\tau}_a/\overline{\alpha}]}$$

$$\frac{\Delta \vdash_{\text{qual}} q \qquad \Delta;\Gamma \vdash_{\text{exp}} e_2 : \tau[\overline{\tau}_1/\overline{\alpha}] \qquad \Delta \vdash \tau[\overline{\tau}_1/\overline{\alpha}] \preceq q}{\Delta;\Gamma \vdash_{\text{exp}} {}^{q}\texttt{pack}(\overline{\tau}_1, e_2) : {}^{q}(\overline{\exists}\overline{\alpha}.\, \tau)}$$

$$\frac{\Delta \vdash \Gamma \rightsquigarrow \Gamma_a \boxdot \Gamma_b \qquad \Delta;\Gamma_a \vdash e_a : {}^{q}(\overline{\exists}\overline{\alpha}.\, \tau_x) \qquad \Delta \vdash_{\text{type}} \tau \qquad \Delta,\overline{\alpha};\Gamma_b, x{:}\tau_x \vdash e_b : \tau}{\Delta;\Gamma \vdash_{\text{exp}} \texttt{let pack}(\overline{\alpha}, x) = e_a\ \texttt{in}\ e_b : \tau}$$

Figure 4.19: Static semantics of rgnURAL (expressions (III))

$\boxed{\Delta; \Gamma \vdash_{\text{exp}} e : \tau}$

$$\frac{\Delta \vdash_{\text{qual}} q \qquad \Delta \vdash \Gamma \preceq q \qquad \Delta, \alpha; \Gamma \vdash_{\text{exp}} e : \tau}{\Delta; \Gamma \vdash_{\text{exp}} {}^{q}\Lambda\alpha.\, e : {}^{q}(\overline{\forall}\alpha.\, \tau)} \qquad \frac{\Delta; \Gamma \vdash_{\text{exp}} e_f : {}^{q}(\overline{\forall}\alpha.\, \tau) \qquad \Delta \vdash_{\text{type}} \tau_a}{\Delta; \Gamma \vdash_{\text{exp}} e_f\ [\tau_a] : \tau[\tau_a/\alpha]}$$

$$\frac{\Delta \vdash_{\text{qual}} q \qquad \Delta; \Gamma \vdash_{\text{exp}} e_2 : \tau[\tau_1/\alpha] \qquad \Delta \vdash \tau[\tau_1/\alpha] \preceq q}{\Delta; \Gamma \vdash_{\text{exp}} {}^{q}\texttt{pack}(\tau_1, e_2) : {}^{q}(\overline{\exists}\alpha.\, \tau)}$$

$$\frac{\Delta \vdash \Gamma \rightsquigarrow \Gamma_a \boxdot \Gamma_b \qquad \Delta; \Gamma_a \vdash_{\text{exp}} e_a : {}^{q}(\overline{\exists}\alpha.\, \tau_x) \qquad \Delta \vdash_{\text{type}} \tau \qquad \Delta, \alpha; \Gamma_b, x{:}\tau_x \vdash e_b : \tau}{\Delta; \Gamma \vdash_{\text{exp}} \texttt{let pack}(\overline{\alpha}, x) = e_a\ \texttt{in}\ e_b : \tau}$$

$$\frac{\Delta \vdash \Gamma \rightsquigarrow \Gamma_a \boxdot \Gamma_b \qquad \Delta; \Gamma_a \vdash_{\text{exp}} e_a : \tau_x \qquad \Delta; \Gamma_b, x{:}\tau_x \vdash e_b : \tau}{\Delta; \Gamma \vdash_{\text{exp}} \texttt{let}\ x = e_a\ \texttt{in}\ e_b : \tau}$$

Figure 4.20: Static semantics of rgnURAL (expressions (IV))

$\boxed{\Delta; \Gamma \vdash_{\text{exp}} e : \tau}$

$$\frac{\Delta \vdash_{\text{qual}} q_c \qquad \Delta \vdash_{\text{qual}} q_h}{\Delta; \cdot \vdash_{\text{exp}} {}^{q_c,q_h}\texttt{newrgn} : {}^{\mathsf{L}}(\overline{\exists}\varrho.\, {}^{\mathsf{L}}({}^{q_c}(\overline{\mathsf{Cap}}\ \varrho) \otimes {}^{q_h}(\overline{\mathsf{Hnd}}\ \varrho)))}$$

$$\frac{\Delta \vdash \Gamma \rightsquigarrow \Gamma_c \boxdot \Gamma_h \qquad \Delta; \Gamma_c \vdash_{\text{exp}} e_c : {}^{q_c}(\overline{\mathsf{Cap}}\ \rho) \qquad \Delta \vdash \mathsf{A} \preceq q_c \qquad \Delta; \Gamma_h \vdash_{\text{exp}} e_h : {}^{q_h}(\overline{\mathsf{Hnd}}\ \rho)}{\Delta; \Gamma \vdash_{\text{exp}} \texttt{freergn}\ e_c\ e_h : {}^{\mathsf{L}}\overline{1}_{\otimes}}$$

Figure 4.21: Static semantics of rgnURAL (expressions (V))

from the corresponding rules in the λ^{URAL}-calculus. Note that the rules for qualifier, pre-type and type abstraction are similar to the rule for function abstraction: the free variables of Γ, which constitute the closure of the abstraction, must be bounded by the qualifier assigned to the abstraction. Meanwhile, the rules for qualifier, pre-type, and type existential packages are similar to the rule for tuple introduction: the type of the packaged component is bounded by the qualifier assigned to the package.

The key rules are those relating to regions and references. Figure 4.21 presents the typing rules for the `newrgn` and `freergn` primitives. Given the operational behavior of the primitives, the typing rules follow naturally. The rule for `newrgn` specifies that it yields an existential package (that hides the name of a fresh region) containing the capability and a handle for the region. The qualifiers for the capability and handle are taken from the `newrgn` annotations. The rule for `freergn` specifies that it takes a capability and a handle for the same region; crucially, the rule for `freergn` requires that the capability's qualifier is either A or L

($\Delta \vdash \mathsf{A} \preceq q_c$). This ensures that there is exactly one capability for ρ in the program state. (If there were more than one copy of the capability for ρ in the program state, this would violate the *at least one use* and *exactly one use* requirements for A and L qualified values.) Since a capability is required for every access of a region, consuming the capability for ρ when the region is deallocated guarantees that the program may not access the region in the future. Conversely, a region for which the capability's qualifier is either U or R may never be deallocated. Since there may be many copies of the capability in the program state, all of which grant access to the region, there is no guarantee that the program will not access the region in the future.

As was noted in Section 4.2.2, each of the reference primitives take a capability as an argument ("proving" that the region is allocated) and return the capability for future use. Likewise, the `read`, `write`, and `swap` primitives take a reference as an argument and return the reference for future use. Finally, it is clear that `new` returns a reference, while `free` takes a reference (without returning it). Hence, the typing rules for the reference primitives must approximate the following types (where we replace multiple arguments by a single linear tuple argument):

$$^{q_r}\texttt{new} \;::\; {}^{\mathsf{U}}(^{\mathsf{L}}(^{q_c}(\overline{\mathsf{Cap}}\ \rho) \otimes {}^{q_h}(\overline{\mathsf{Hnd}}\ \rho) \otimes \tau) \multimap \mathsf{L}(^{q_c}(\overline{\mathsf{Cap}}\ \rho) \otimes {}^{q_r}(\overline{\mathsf{Ref}}\ \rho\ \tau)))$$

$$\texttt{free} \;::\; {}^{\mathsf{U}}(^{\mathsf{L}}(^{q_c}(\overline{\mathsf{Cap}}\ \rho) \otimes {}^{q_r}(\overline{\mathsf{Ref}}\ \rho\ \tau)) \multimap \mathsf{L}(^{q_c}(\overline{\mathsf{Cap}}\ \rho) \otimes \tau))$$

$$\texttt{read} \;::\; {}^{\mathsf{U}}(^{\mathsf{L}}(^{q_c}(\overline{\mathsf{Cap}}\ \rho) \otimes {}^{q_r}(\overline{\mathsf{Ref}}\ \rho\ \tau)) \multimap \mathsf{L}(^{q_c}(\overline{\mathsf{Cap}}\ \rho) \otimes {}^{q_r}(\overline{\mathsf{Ref}}\ \rho\ \tau) \otimes \tau))$$

$$\texttt{write} \;::\; {}^{\mathsf{U}}(^{\mathsf{L}}(^{q_c}(\overline{\mathsf{Cap}}\ \rho) \otimes {}^{q_r}(\overline{\mathsf{Ref}}\ \rho\ \tau) \otimes \tau_\star) \multimap \mathsf{L}(^{q_c}(\overline{\mathsf{Cap}}\ \rho) \otimes {}^{q_r}(\overline{\mathsf{Ref}}\ \rho\ \tau_\star)))$$

$$\texttt{swap} \;::\; {}^{\mathsf{U}}(^{\mathsf{L}}(^{q_c}(\overline{\mathsf{Cap}}\ \rho) \otimes {}^{q_r}(\overline{\mathsf{Ref}}\ \rho\ \tau) \otimes \tau_\star) \multimap \mathsf{L}(^{q_c}(\overline{\mathsf{Cap}}\ \rho) \otimes {}^{q_r}(\overline{\mathsf{Ref}}\ \rho\ \tau_\star) \otimes \tau))$$

It remains to answer (and to understand the answers to) the following questions: What primitives may safely operate on the different sorts of qualified references? What combinations of qualifiers for a reference and qualifiers for its contents are

$^{q_r}(\overline{\mathsf{Ref}}\ \rho\ ^{q}\overline{\tau})$ q_r \ q		Prims	Contents and Prims			
			U	R	A	L
shared	U	$^{\mathsf{U}}$`new` *(weak updates)*	`read` `write` `swap`	X	`write` `swap`	X
shared	R	$^{\mathsf{R}}$`new` *(weak updates)*	`read` `write` `swap`	`read` `swap`	`write` `swap`	`swap`
unique	A	$^{\mathsf{A}}$`new` `free` *(strong updates)*	`read` `write` `swap`	X	`write` `swap`	X
unique	L	$^{\mathsf{L}}$`new` `free` *(strong updates)*	`read` `write` `swap`	`read` `swap`	`write` `swap`	`swap`

Figure 4.22: Reference primitives for rgnURAL

safe (that is, how are q_r and τ related in the type $^{q_r}(\overline{\mathsf{Ref}}\ \rho\ \tau)$)? The answers to these questions are summarized in Figure 4.22.

First, consider what it means to duplicate a reference. Operationally, a reference is a pointer in the global heap of regions. Therefore, duplicating an unrestricted or relevant reference $\overline{\mathsf{Ref}}\ \mathfrak{r}\ \mathfrak{p}$, simply yields two copies of $\overline{\mathsf{Ref}}\ \mathfrak{r}\ \mathfrak{p}$ — while the value stored at pointer $\mathfrak{p}$ in region $\mathfrak{r}$ is *not* duplicated. Since duplicating a shared reference does not alter the uniqueness of its contents, it is both reasonable and extremely useful to allow shared references to store unique values. This permits the sharing of (large) unique data structures without expensive copying.

On the other hand, dropping an unrestricted or affine reference $\overline{\mathsf{Ref}}\ \mathtt{r}\ \mathtt{p}$ effectively drops its contents, since this reference may (must, in the case of affine) have been the only copy of $\overline{\mathsf{Ref}}\ \mathtt{r}\ \mathtt{p}$ in the program state. If the contents were a linear or relevant value, then the *exactly one use* and *at least one use* invariants (respectively) would be violated. Hence, we cannot allow linear and relevant values (which cannot be discarded) to be stored in unrestricted or affine references (which can be discarded).

Consider yet another axis. Recall that linear and affine references must be unique (in the program state), while relevant and unrestricted references may be shared (in the program state). Hence, uniquene references can be deallocated (`free`) and can support strong (type-varing) updates. On the other hand, shared references can never be deallocated and can only support weak updates.

As we noted above, the `read` operator induces an implicit copy while the `write` operator induces an implicit drop. Therefore, whether we can read from or write to a reference depends entirely on the qualifier of its contents: `read` is permitted if the contents are unrestricted or relevant (i.e., duplicable), `write` is permitted if the contents are unrestricted or affine (i.e., discardable). The operation `swap` is permitted on any sort of reference, regardless of the qualifier of its contents. As noted above, strong writes and strong swaps, which change the type of the contents of the location, are only permitted on unique references.

Figures 4.23 and 4.24 presents the typing rules for the reference primitives. Note that the invariant that a U or A qualified reference may not contain an R or L qualified value is established by the New(Any) rule and is maintained (implicitly) by the Write(Weak) and Swap(weak) rules and (explicitly) by the Write(Strong) and Swap(Strong) rules.

$\boxed{\Delta; \Gamma \vdash_{\text{exp}} e : \tau}$

NEW(ANY)

$$\frac{\Delta \vdash_{\text{qual}} q_r \qquad \Delta \vdash \Gamma \rightsquigarrow \Gamma_c \boxdot \Gamma_h \boxdot \Gamma_a \qquad \Delta; \Gamma_c \vdash_{\text{exp}} e_c : {}^{q_c}(\overline{\mathsf{Cap}}\ \rho) \qquad \Delta; \Gamma_h \vdash_{\text{exp}} e_h : {}^{q_h}(\overline{\mathsf{Hnd}}\ \rho) \qquad \Delta; \Gamma_a \vdash_{\text{exp}} e_\star : \tau \qquad \Delta \vdash \tau \preceq \mathsf{A}}{\Delta; \Gamma \vdash_{\text{exp}} {}^{q_r}\texttt{new}\ e_c\ e_h\ e_\star : {}^{\mathsf{L}}({}^{q_c}(\overline{\mathsf{Cap}}\ \rho) \otimes {}^{q_r}(\overline{\mathsf{Ref}}\ \rho\ \tau))}$$

NEW(R,L)

$$\frac{\Delta \vdash_{\text{qual}} q_r \qquad \Delta \vdash \Gamma \rightsquigarrow \Gamma_c \boxdot \Gamma_h \boxdot \Gamma_a \qquad \Delta; \Gamma_c \vdash_{\text{exp}} e_c : {}^{q_c}(\overline{\mathsf{Cap}}\ \rho) \qquad \Delta; \Gamma_h \vdash_{\text{exp}} e_h : {}^{q_h}(\overline{\mathsf{Hnd}}\ \rho) \qquad \Delta; \Gamma_a \vdash_{\text{exp}} e_\star : \tau \qquad \Delta \vdash \mathsf{R} \preceq q_r}{\Delta; \Gamma \vdash_{\text{exp}} {}^{q_r}\texttt{new}\ e_c\ e_h\ e_\star : {}^{\mathsf{L}}({}^{q_c}(\overline{\mathsf{Cap}}\ \rho) \otimes {}^{q_r}(\overline{\mathsf{Ref}}\ \rho\ \tau))}$$

$$\frac{\Delta \vdash \Gamma \rightsquigarrow \Gamma_c \boxdot \Gamma_r \qquad \Delta; \Gamma_c \vdash_{\text{exp}} e_c : {}^{q_c}(\overline{\mathsf{Cap}}\ \rho) \qquad \Delta; \Gamma_r \vdash_{\text{exp}} e_r : {}^{q_r}(\overline{\mathsf{Ref}}\ \rho\ \tau) \qquad \Delta \vdash \mathsf{A} \preceq q_r}{\Delta; \Gamma \vdash_{\text{exp}} \texttt{free}\ e_c\ e_r : {}^{\mathsf{L}}({}^{q_c}(\overline{\mathsf{Cap}}\ \rho) \otimes \tau)}$$

Figure 4.23: Static semantics of rgnURAL (expressions (VI))

$\boxed{\Delta; \Gamma \vdash_{\text{exp}} e : \tau}$

$$\frac{\Delta \vdash \Gamma \rightsquigarrow \Gamma_c \boxdot \Gamma_r \qquad \Delta; \Gamma_c \vdash_{\text{exp}} e_c : {}^{q_c}(\overline{\mathsf{Cap}}\ \rho) \qquad \Delta; \Gamma_r \vdash_{\text{exp}} e_r : {}^{q_r}(\overline{\mathsf{Ref}}\ \rho\ \tau) \qquad \Delta \vdash \tau \preceq \mathsf{R}}{\Delta; \Gamma \vdash_{\text{exp}} \texttt{read}\ e_c\ e_r : {}^{\mathsf{L}}({}^{q_c}(\overline{\mathsf{Cap}}\ \rho) \otimes {}^{q_r}(\overline{\mathsf{Ref}}\ \rho\ \tau) \otimes \tau)}$$

WRITE(WEAK)

$$\frac{\Delta \vdash \Gamma \rightsquigarrow \Gamma_c \boxdot \Gamma_r \boxdot \Gamma_\star \qquad \Delta; \Gamma_c \vdash_{\text{exp}} e_c : {}^{q_c}(\overline{\mathsf{Cap}}\ \rho) \qquad \Delta; \Gamma_r \vdash_{\text{exp}} e_r : {}^{q_r}(\overline{\mathsf{Ref}}\ \rho\ \tau) \qquad \Delta \vdash \tau \preceq \mathsf{A} \qquad \Delta; \Gamma_\star \vdash_{\text{exp}} e_\star : \tau}{\Delta; \Gamma \vdash_{\text{exp}} \texttt{write}\ e_c\ e_r\ e_\star : {}^{\mathsf{L}}({}^{q_c}(\overline{\mathsf{Cap}}\ \rho) \otimes {}^{q_r}(\overline{\mathsf{Ref}}\ \rho\ \tau))}$$

WRITE(STRONG)

$$\frac{\Delta \vdash \Gamma \rightsquigarrow \Gamma_c \boxdot \Gamma_r \boxdot \Gamma_\star \qquad \Delta; \Gamma_c \vdash_{\text{exp}} e_c : {}^{q_c}(\overline{\mathsf{Cap}}\ \rho) \qquad \Delta; \Gamma_r \vdash_{\text{exp}} e_r : {}^{q_r}(\overline{\mathsf{Ref}}\ \rho\ \tau) \qquad \Delta \vdash \tau \preceq \mathsf{A} \qquad \Delta; \Gamma_\star \vdash_{\text{exp}} e_\star : \tau_\star \qquad \Delta \vdash \mathsf{A} \preceq q_r \qquad \Delta \vdash \tau_\star \preceq q_r}{\Delta; \Gamma \vdash_{\text{exp}} \texttt{write}\ e_c\ e_r\ e_\star : {}^{\mathsf{L}}({}^{q_c}(\overline{\mathsf{Cap}}\ \rho) \otimes {}^{q_r}(\overline{\mathsf{Ref}}\ \rho\ \tau_\star))}$$

SWAP(WEAK)

$$\frac{\Delta \vdash \Gamma \rightsquigarrow \Gamma_c \boxdot \Gamma_r \boxdot \Gamma_\star \qquad \Delta; \Gamma_c \vdash_{\text{exp}} e_c : {}^{q_c}(\overline{\mathsf{Cap}}\ \rho) \qquad \Delta; \Gamma_r \vdash_{\text{exp}} e_r : {}^{q_r}(\overline{\mathsf{Ref}}\ \rho\ \tau) \qquad \Delta; \Gamma_\star \vdash_{\text{exp}} e_\star : \tau}{\Delta; \Gamma \vdash_{\text{exp}} \texttt{swap}\ e_c\ e_r\ e_\star : {}^{\mathsf{L}}({}^{q_c}(\overline{\mathsf{Cap}}\ \rho) \otimes {}^{q_r}(\overline{\mathsf{Ref}}\ \rho\ \tau) \otimes \tau)}$$

SWAP(STRONG)

$$\frac{\Delta \vdash \Gamma \rightsquigarrow \Gamma_c \boxdot \Gamma_r \boxdot \Gamma_\star \qquad \Delta; \Gamma_c \vdash_{\text{exp}} e_c : {}^{q_c}(\overline{\mathsf{Cap}}\ \rho) \qquad \Delta; \Gamma_r \vdash_{\text{exp}} e_r : {}^{q_r}(\overline{\mathsf{Ref}}\ \rho\ \tau) \qquad \Delta; \Gamma_\star \vdash_{\text{exp}} e_\star : \tau_\star \qquad \Delta \vdash \mathsf{A} \preceq q_r \qquad \Delta \vdash \tau_\star \preceq q_r}{\Delta; \Gamma \vdash_{\text{exp}} \texttt{swap}\ e_c\ e_r\ e_\star : {}^{\mathsf{L}}({}^{q_c}(\overline{\mathsf{Cap}}\ \rho) \otimes {}^{q_r}(\overline{\mathsf{Ref}}\ \rho\ \tau_\star) \otimes \tau)}$$

Figure 4.24: Static semantics of rgnURAL (expressions (VII))

Finally, we note that `write` may be encoded using an explicit `swap` and an implicit drop:[11]

$$\frac{\Delta \vdash \Gamma \rightsquigarrow \Gamma_c \boxdot \Gamma_r \boxdot \Gamma_\star \qquad \Delta; \Gamma_c \vdash_{\text{exp}} e_c : {}^{q_c}(\overline{\mathsf{Cap}}\ \rho) \qquad \Delta; \Gamma_r \vdash_{\text{exp}} e_r : {}^{q_r}(\overline{\mathsf{Ref}}\ \rho\ \tau) \qquad \Delta \vdash \tau \preceq \mathsf{A} \qquad \Delta; \Gamma_\star \vdash_{\text{exp}} e_\star : \tau}{\Delta; \Gamma \vdash_{\text{exp}} \texttt{write}\ e_c\ e_r\ e_\star : {}^{\mathsf{L}}({}^{q_c}(\overline{\mathsf{Cap}}\ \rho) \otimes {}^{q_r}(\overline{\mathsf{Ref}}\ \rho\ \tau))}\ \textsc{Write}(\textsc{Weak})$$

$$\equiv \text{// using } \textsc{Swap}(\textsc{Weak}) \text{ and } \Delta; \Gamma \vdash_{\text{exp}} e_\star : \tau$$

$$\texttt{let}\ \langle c, r, x \rangle = \texttt{swap}\ e_c\ e_r\ e_\star\ \texttt{in}$$

$$\text{// using } \textsc{Weak} \text{ and } \Delta \vdash \tau \preceq \mathsf{A}\text{, drop } x{:}\tau$$

$${}^{\mathsf{L}}\langle c, r \rangle$$

However, `read` may not be encoded using an explicit `swap` and an implicit copy, as a suitable (discardable) dummy value cannot in general be synthesized:

$$\frac{\Delta \vdash \Gamma \rightsquigarrow \Gamma_c \boxdot \Gamma_r \qquad \Delta; \Gamma_c \vdash_{\text{exp}} e_c : {}^{q_c}(\overline{\mathsf{Cap}}\ \rho) \qquad \Delta; \Gamma_r \vdash_{\text{exp}} e_r : {}^{q_r}(\overline{\mathsf{Ref}}\ \rho\ \tau) \qquad \Delta \vdash \tau \preceq \mathsf{R}}{\Delta; \Gamma \vdash_{\text{exp}} \texttt{read}\ e_c\ e_r : {}^{\mathsf{L}}({}^{q_c}(\overline{\mathsf{Cap}}\ \rho) \otimes {}^{q_r}(\overline{\mathsf{Ref}}\ \rho\ \tau) \otimes \tau)}$$

$$\equiv \text{// using } \textsc{Swap}(\textsc{Weak}) \text{ and } \Delta; \Gamma \vdash_{\text{exp}} ? : \tau$$

$$\texttt{let}\ \langle c, r, x \rangle = \texttt{swap}\ e_c\ e_r\ ?\ \texttt{in}$$

$$\text{// using } \textsc{Contr} \text{ and } \Delta \vdash \tau \preceq \mathsf{R}\text{, copy } x{:}\tau$$

$$\texttt{let}\ \langle c, r, y \rangle = \texttt{swap}\ c\ r\ x\ \texttt{in}\ \text{// using } \textsc{Swap}(\textsc{Weak})$$

$$\text{// using } \textsc{Weak} \text{ and } \Delta \vdash \tau \preceq \mathsf{A}\text{, drop } y{:}\tau$$

$$\langle c, r, x \rangle$$

The shaded expression and antecedents, which would be necessary for the implied typing derivation, cannot be satisfied by the antecedents of the rule for `read`.

[11] The encoding of a `write` typed by the $\textsc{Write}(\textsc{Strong})$ rule makes use of the same term, but an alternate typing derivation.

$\boxed{\Delta \vdash_{\text{qual}} q}$

$$\frac{\xi \in dom(\Delta)}{\Delta \vdash_{\text{qual}} \xi} \qquad \frac{}{\Delta \vdash_{\text{qual}} \mathsf{q}}$$

$\boxed{\Delta \vdash_{\text{ptype}} \overline{\tau}}$

$$\frac{\overline{\alpha} \in dom(\Delta)}{\Delta \vdash_{\text{ptype}} \overline{\alpha}} \qquad \frac{}{\Delta \vdash_{\text{ptype}} \overline{\mathsf{Int}}} \qquad \frac{}{\Delta \vdash_{\text{ptype}} \overline{\mathsf{Bool}}} \qquad \frac{\Delta \vdash_{\text{type}} \tau_1 \qquad \Delta \vdash_{\text{type}} \tau_2}{\Delta \vdash_{\text{ptype}} \tau_1 \multimap \tau_2}$$

$$\frac{\Delta \vdash_{\text{type}} \tau_i \quad {}^{i \in 1 \ldots n}}{\Delta \vdash_{\text{ptype}} \tau_1 \otimes \cdots \otimes \tau_n} \qquad \frac{\Delta, \xi \vdash_{\text{type}} \tau}{\Delta \vdash_{\text{ptype}} \overline{\forall} \xi.\, \tau} \qquad \frac{\Delta, \xi \vdash_{\text{type}} \tau}{\Delta \vdash_{\text{ptype}} \overline{\exists} \xi.\, \tau} \qquad \frac{\Delta, \overline{\alpha} \vdash_{\text{type}} \tau}{\Delta \vdash_{\text{ptype}} \overline{\forall} \overline{\alpha}.\, \tau}$$

$$\frac{\Delta, \overline{\alpha} \vdash_{\text{type}} \tau}{\Delta \vdash_{\text{ptype}} \overline{\exists} \overline{\alpha}.\, \tau} \qquad \frac{\Delta, \alpha \vdash_{\text{type}} \tau}{\Delta \vdash_{\text{ptype}} \overline{\forall} \alpha.\, \tau} \qquad \frac{\Delta, \alpha \vdash_{\text{type}} \tau}{\Delta \vdash_{\text{ptype}} \overline{\exists} \alpha.\, \tau} \qquad \frac{\Delta \vdash_{\text{region}} \rho \qquad \Delta \vdash_{\text{type}} \tau}{\Delta \vdash_{\text{ptype}} \overline{\mathsf{Ref}}\ \rho\ \tau}$$

$$\frac{\Delta \vdash_{\text{region}} \rho}{\Delta \vdash_{\text{ptype}} \overline{\mathsf{Hnd}}\ \rho} \qquad \frac{\Delta \vdash_{\text{region}} \rho}{\Delta \vdash_{\text{ptype}} \overline{\mathsf{Cap}}\ \rho} \qquad \frac{\Delta, \varrho \vdash_{\text{type}} \tau}{\Delta \vdash_{\text{ptype}} \overline{\forall} \varrho.\, \tau} \qquad \frac{\Delta, \varrho \vdash_{\text{type}} \tau}{\Delta \vdash_{\text{ptype}} \overline{\exists} \varrho.\, \tau}$$

$\boxed{\Delta \vdash_{\text{type}} \tau}$

$$\frac{\alpha \in dom(\Delta)}{\Delta \vdash_{\text{type}} \alpha} \qquad \frac{\Delta \vdash_{\text{qual}} q \qquad \Delta \vdash_{\text{ptype}} \overline{\tau}}{\Delta \vdash_{\text{type}} {}^{q}\overline{\tau}}$$

$\boxed{\Delta \vdash_{\text{region}} \rho}$

$$\frac{\varrho \in dom(\Delta)}{\Delta \vdash_{\text{region}} \varrho}$$

Figure 4.25: Static semantics of rgnURAL (qualifiers, pre-types, types, and regions)

Qualifiers, pre-types, types, and regions Figure 4.25 contains additional (completely standard) judgments for ensuring that qualifiers q, pre-types $\overline{\tau}$, types τ, and regions ρ are well-formed. These judgments simply enforce the invariant that no type or expression may depend upon unbound qualifier, pre-type, type, or region variables.

Type Soundness

As expected, the type system for rgnURAL is sound with respect to its operational semantics:

Theorem 4.1 (rgnURAL **Soundness)**

If $\cdot;\cdot \vdash e_1 : \tau$ *and* $(\{\}; e_1) \longmapsto^* (H_2; e_2)$*, then either there exists* v *such that* $e_2 \equiv v$ *or there exists* H_3 *and* e_3 *such that* $(H_2; e_2) \longmapsto (H_3; e_3)$.

We have formally verified this result (for a rich superset of rgnURAL) in the Twelf system [66] using its metatheorem checker [71]. See Appendix C.2 for more details.

Remarks

An interesting corollary of the type soundness of rgnURAL is that if a closed, well-typed program of base type (e.g., $^{\mathsf{U}}\overline{\mathsf{Bool}}$) is evaluated to a value, then the resulting heap will have no $^{\mathsf{L}}\texttt{live}$ regions and no $^{\mathsf{L}}(\texttt{ref}\ \mathfrak{r}\ \mathfrak{p})$ values. That is, any region introduced by $^{\mathsf{L},q_h}\texttt{newrgn}$ and any reference introduced by $^{\mathsf{L}}\texttt{new}$ will be freed during the evaluation of the program, by either `freergn` or `free`.

We argue that every linear region is deallocated in the following manner. First, we note that the evaluation of $^{\mathsf{L},q_h}\texttt{newrgn}$ introduces a $^{\mathsf{L}}(\mathsf{cap})$ value of type $^{\mathsf{L}}(\overline{\mathsf{Cap}}\ \mathfrak{r})$ into the program; it simultaneously allocates the corresponding region, initializing

its mark to $^{\mathsf{L}}$`live`. Next, we note that the reference primitives of rgnURAL all take a capability as an argument, but also return a capability as a result. Hence, they neither duplicate nor discard the capability. Furthermore, such a linear value may not be implicitly duplicated or discarded. Only the `freergn` primitive uses a capability in a manner that removes it from the program state; it simultaneously "deallocates" the corresponding region, by transitioning its mark from $^{\mathsf{L}}$`live` to `dead`. In a fully evaluated program of type $^{\mathsf{U}}\overline{\mathsf{Bool}}$, every linear value must be used exactly once – hence, every $^{\mathsf{L}}$(`cap`) must be used to deallocate the corresponding region.

Similar reasoning shows that every linear reference is deallocated. At first, one might be worried about allocating linear references in a region, because deallocating the region would implicitly discard the linear reference (hence, implicitly discarding the contents of the reference). Initially, one might be concerned how one could ensure that at a `freergn`, every linear reference in the region had already been reclaimed by `free`.

However, note that the evaluation of $^{\mathsf{L}}$`new` introduces a $^{\mathsf{L}}$(`ref` $\mathfrak{r}$ $\mathfrak{p}$) value of type $^{\mathsf{L}}$($\overline{\mathsf{Ref}}$ $\mathfrak{r}$ τ) into the program. The `read`, `write`, and `swap` primitives all take a reference as an argument and return a reference as a result. Hence, they neither duplicate nor discard the reference. Only the `free` primitive uses a reference in a manner that removes it from the program state.

Now, suppose that the region $\mathfrak{r}$ is deallocated before the reference $^{\mathsf{L}}$(`ref` $\mathfrak{r}$ $\mathfrak{p}$) is freed. Deallocating the region removes the capability for the region from the program state. Since every reference primitive requires the region capability, the reference $^{\mathsf{L}}$(`ref` $\mathfrak{r}$ $\mathfrak{p}$) remains in the program, but may not be accessed. Furthermore, the type of the reference must ultimately manifest itself in some manner in

the type of the program; while the reference may be hidden in a closure or existential package, the containing data structure will necessarily have a linear type. Since the final type of the program is $^{\mathsf{U}}\overline{\mathsf{Bool}}$, there can't be any linear values (of any type) in the program state. Furthermore, since the region capability is needed to both free a reference and to deallocate a region, it must be the case that every linear reference is freed before deallocating its region.

4.3 Translation: From $\mathsf{F}^{\mathsf{RGN}}$ to rgnURAL

In this section, we present a type- and meaning-preserving translation from $\mathsf{F}^{\mathsf{RGN}}$ to rgnURAL. Before giving the details, we discuss a few of the high-level issues.

First, we note that $\mathsf{F}^{\mathsf{RGN}}$ has no notion of linearity in the syntax or type system. Rather, all types (and, therefore, all expressions) are implicitly considered unrestricted. Hence, we can expect that the translation of all $\mathsf{F}^{\mathsf{RGN}}$ expressions will yield rgnURAL expressions with a U qualified pre-type.

On the other hand, we claimed that a stateful region computation could be interpreted as a region-stack transformer. Recall that the type $\mathsf{RGN}\ \theta\ \tau$ is the type of a computation which transforms a region stack indexed by θ and delivers a value of type τ. A key characteristic of $\mathsf{F}^{\mathsf{RGN}}$ is that all primitive region-stack transformers are meant to use the region stack in a single-threaded manner; hence, a stateful computation can update the region stack in place. This single-threaded behavior is precisely the sort of resource management that may be captured by a substructural type system. Hence, we can expect that the representation of a stack of regions in rgnURAL will be a value with L qualified pre-type. In particular, we will represent a stack of regions as a sequence of linear capabilities, formed out of nested linear tuples.

Third, we must be mindful of a slight mismatch between the RGNHnd and RGNRef types in $\mathsf{F}^{\mathsf{RGN}}$ and the $\overline{\mathsf{Hnd}}$ and $\overline{\mathsf{Ref}}$ pre-types in $\mathsf{rgnURAL}$. Recall that, in $\mathsf{F}^{\mathsf{RGN}}$, $\mathsf{RGNHnd}\ \theta$ and $\mathsf{RGNRef}\ \theta\ \tau$ are handles for and references allocated in the region at the top of the stack indexed by θ. Whereas, in $\mathsf{rgnURAL}$, $\overline{\mathsf{Hnd}}\ \rho$ and $\overline{\mathsf{Ref}}\ \rho\ \tau$ explicitly name the region of the handle or reference. This subtle distinction (whether the region is implicit or explicit) will need to be handled by the translation.

We start with a few preliminaries. We assume an injection from the set $VVars^{\mathsf{F}^{\mathsf{RGN}}}$ to the set $VVars^{\mathsf{rgnURAL}}$. In the translation, applications of this injection will be clear from context and we freely use source value variables as target value variables. We also assume an injection from the set $TVars^{\mathsf{F}^{\mathsf{RGN}}}$ to the set $PTVars^{\mathsf{rgnURAL}}$; this injection, written $\overline{\alpha}_{\alpha}$, will denote the $\mathsf{rgnURAL}$ pre-type variable for the $\mathsf{F}^{\mathsf{RGN}}$ type variable α. Similarly,we assume an injection from the set $IVars^{\mathsf{F}^{\mathsf{RGN}}}$ to the set $TVars^{\mathsf{F}^{\mathsf{RGN}}}$; this injection, written α_{ϑ}, will denote the $\mathsf{rgnURAL}$ type variable for the $\mathsf{F}^{\mathsf{RGN}}$ index variable ϑ.

The translation is a typed call-by-value translation, given by a number of functions: $\mathbb{T}[\![\cdot]\!]$ translates into types, $\mathbb{P}[\![\cdot]\!]$ translates into pre-types, $\mathbb{D}[\![\cdot]\!]$ translates into qualifier, pre-type, type, and region contexts, $\mathbb{G}[\![\cdot]\!]$ translates into value contexts, and $\mathbb{E}[\![\cdot]\!]$ translates into expressions. Technically, there are separate functions for each syntactic class in the source calculus, but we elide this detail as it is always clear from context. Additionally, to reduce notational clutter, translations from judgments are often written in an abbreviated form giving only the main component; the corresponding judgment should be clear from context.

Translations yielding types

Indices

$$\mathbb{T}\left[\!\!\left[\frac{\vartheta \in dom(\Delta)}{\Delta \vdash_{\text{index}} \vartheta} \right]\!\!\right] \;=\; \alpha_\vartheta$$

Translations yielding types

Types

$$\mathbb{T}[\![\Delta \vdash_{\text{type}} \tau]\!] \;=\; {}^{\mathsf{U}}\mathbb{P}[\![\tau]\!]$$

Figure 4.26: Translation from $\mathsf{F}^{\mathsf{RGN}}$ to rgnURAL (indices and types (I))

Indices and types Figure 4.26 shows the translations of $\mathsf{F}^{\mathsf{RGN}}$ indices and types to rgnURAL types, while Figures 4.27 and 4.28 show the translation of $\mathsf{F}^{\mathsf{RGN}}$ types to rgnURAL pre-types. Since the surface syntax of $\mathsf{F}^{\mathsf{RGN}}$ admits only index variables as index terms, the index variable ϑ is translated to the rgnURAL type variable α_ϑ.

$\mathbb{T}[\![\Delta \vdash_{\text{type}} \tau]\!]$ and $\mathbb{P}[\![\Delta \vdash_{\text{type}} \tau]\!]$ translate a $\mathsf{F}^{\mathsf{RGN}}$ type to a rgnURAL type and pre-type, respectively. As was observed above, when we translate a $\mathsf{F}^{\mathsf{RGN}}$ type to a rgnURAL type, we ensure that the result is a U qualified pre-type. The $\mathbb{P}[\![\Delta \vdash_{\text{type}} \tau]\!]$ translation is straightforward on the functional types in Figure 4.27. However, note that a $\mathsf{F}^{\mathsf{RGN}}$ type variable α is translated to a rgnURAL pre-type variable $\overline{\alpha}_\alpha$; this ensures that *every* type corresponding to a $\mathsf{F}^{\mathsf{RGN}}$ type is manifestly qualified with U. We discuss this issue more when we consider the translation of $\mathsf{F}^{\mathsf{RGN}}$ expressions.

More interesting are the translations of the types associated with the RGN monad (Figure 4.28). In the translation of the RGN θ τ type, we see the familiar store (or, in this case, stack) passing interpretation of computations: RGN θ τ is translated to a function taking (the representation of) a stack of regions and

Translations yielding pre-types

Types

$$\mathbb{P}\left[\!\!\left[\frac{\alpha \in dom(\Delta)}{\Delta \vdash_{\text{type}} \alpha}\right]\!\!\right] \;=\; \overline{\alpha}_{\alpha}$$

$$\mathbb{P}\left[\!\!\left[\frac{}{\Delta \vdash_{\text{type}} \mathsf{Int}}\right]\!\!\right] \;=\; \overline{\mathsf{Int}}$$

$$\mathbb{P}\left[\!\!\left[\frac{}{\Delta \vdash_{\text{type}} \mathsf{Bool}}\right]\!\!\right] \;=\; \overline{\mathsf{Bool}}$$

$$\mathbb{P}\left[\!\!\left[\frac{\Delta \vdash_{\text{type}} \tau_1 \qquad \Delta \vdash_{\text{type}} \tau_2}{\Delta \vdash_{\text{type}} \tau_1 \to \tau_2}\right]\!\!\right] \;=\; \mathbb{T}[\![\tau_1]\!] \multimap \mathbb{T}[\![\tau_2]\!]$$

$$\mathbb{P}\left[\!\!\left[\frac{\Delta \vdash_{\text{type}} \tau_i \quad {}^{i\in 1\ldots n}}{\Delta \vdash_{\text{type}} \tau_1 \times \cdots \times \tau_n}\right]\!\!\right] \;=\; \mathbb{T}[\![\tau_1]\!] \otimes \cdots \otimes \mathbb{T}[\![\tau_n]\!]$$

$$\mathbb{P}\left[\!\!\left[\frac{\Delta, \alpha \vdash_{\text{type}} \tau}{\Delta \vdash_{\text{type}} \forall \alpha.\, \tau}\right]\!\!\right] \;=\; \overline{\forall}\overline{\alpha}_{\alpha}.\, \mathbb{T}[\![\tau]\!]$$

Figure 4.27: Translation from $\mathsf{F}^{\mathsf{RGN}}$ to rgnURAL (types (II))

Translations yielding pre-types

Types

$$\mathbb{P}\left[\!\!\left[\frac{\Delta \vdash_{\text{index}} \theta \qquad \Delta \vdash_{\text{type}} \tau}{\Delta \vdash_{\text{type}} \mathsf{RGN}\ \theta\ \tau}\right]\!\!\right] \;=\; \mathbb{T}[\![\theta]\!] \multimap {}^{\mathsf{L}}(\mathbb{T}[\![\theta]\!] \otimes \mathbb{T}[\![\tau]\!])$$

$$\mathbb{P}\left[\!\!\left[\frac{\Delta \vdash_{\text{index}} \theta \qquad \Delta \vdash_{\text{type}} \tau}{\Delta \vdash_{\text{type}} \mathsf{RGNRef}\ \theta\ \tau}\right]\!\!\right] =$$

$$\overline{\exists}\varrho_r.\, {}^{\mathsf{U}}({}^{\mathsf{U}}(\overline{\exists}\beta.\, \mathbf{Iso}(\mathbb{T}[\![\theta]\!], {}^{\mathsf{L}}(\beta \otimes {}^{\mathsf{L}}(\overline{\mathsf{Cap}}\ \varrho_r)))) \otimes {}^{\mathsf{U}}(\overline{\mathsf{Ref}}\ \varrho_r\ \mathbb{T}[\![\tau]\!]))$$

$$\mathbb{P}\left[\!\!\left[\frac{\Delta \vdash_{\text{index}} \theta}{\Delta \vdash_{\text{type}} \mathsf{RGNHnd}\ \theta}\right]\!\!\right] =$$

$$\overline{\exists}\varrho_r.\, {}^{\mathsf{U}}({}^{\mathsf{U}}(\overline{\exists}\beta.\, \mathbf{Iso}(\mathbb{T}[\![\theta]\!], {}^{\mathsf{L}}(\beta \otimes {}^{\mathsf{L}}(\overline{\mathsf{Cap}}\ \varrho_r)))) \otimes {}^{\mathsf{U}}(\overline{\mathsf{Hnd}}\ \varrho_r))$$

$$\mathbb{P}\left[\!\!\left[\frac{\Delta, \vartheta \vdash_{\text{type}} \tau}{\Delta \vdash_{\text{type}} \forall\vartheta.\, \tau}\right]\!\!\right] \;=\; \overline{\forall}\alpha_\vartheta.\, \mathbb{T}[\![\tau]\!]$$

Isomorphism Macro

$$\mathbf{Iso}(\tau_1, \tau_2) \;=\; {}^{\mathsf{U}}({}^{\mathsf{U}}(\tau_1 \multimap \tau_2) \otimes {}^{\mathsf{U}}(\tau_2 \multimap \tau_1))$$

Figure 4.28: Translation from $\mathsf{F}^{\mathsf{RGN}}$ to rgnURAL (types (III))

returning the pair of a stack of regions and a result value. Since the representation of a stack of regions is linear, the resulting stack/value pair is qualified with L.

Next, consider the translation of the RGNRef θ τ type. Recall that this is the type of references allocated in the (top-most region of the) region-stack indexed by θ. Since the type does not directly name the region in which the reference is allocated, the translation must both explicitly name the region and relate the region to the (translation of the) index. In the translation, an existentially

bound region name ϱ_r fixes the region for the $\mathsf{rgnURAL}$ reference, while an isomorphism witnesses the fact that (the capability for) ϱ_r may be found within the stack $\mathbb{T}[\![\theta]\!]$. The isomorphism expresses the fact that $\mathbb{T}[\![\theta]\!]$ may be coerced to and from $\mathsf{L}(\beta \otimes \mathsf{L}(\overline{\mathsf{Cap}}\ \varrho_r))$, for some "slack" β. Note that while the types $\mathbb{T}[\![\theta]\!]$, $\mathsf{L}(\overline{\mathsf{Cap}}\ \varrho_r)$, and β may be linear, the pair of functions witnessing the isomorphism is unrestricted. This corresponds to the fact that the *proof* that the capability $\mathsf{L}(\overline{\mathsf{Cap}}\ \varrho_r)$ is a member of the stack $\mathbb{T}[\![\theta]\!]$ is *persistent*, while the *existence* of the the capability $\mathsf{L}(\overline{\mathsf{Cap}}\ \varrho_r)$ and the stack $\mathbb{T}[\![\theta]\!]$ are *ephemeral*.

The translation of the $\mathsf{RGNHnd}\ \theta$ type is similar.

At this time, we recall the $\mathsf{F}^{\mathsf{RGN}}$ abbreviation $\mathsf{RGNPf}(\theta_1 \preceq \theta_2)$ for the type of a function that coerces any computation transforming the region stack indexed by θ_1 into a computation transforming the region stack indexed by θ_2, and consider its translation:

$$
\begin{aligned}
&\mathbb{T}[\![\mathsf{RGNPf}(\theta_1 \preceq \theta_2)]\!] \\
&\quad = \mathbb{T}[\![\forall\beta.\,\mathsf{RGN}\ \theta_1\ \beta \to \mathsf{RGN}\ \theta_2\ \beta]\!] \\
&\quad = \mathsf{U}(\overline{\forall}\overline{\alpha}_\beta.\,\mathbb{T}[\![\mathsf{RGN}\ \theta_1\ \beta \to \mathsf{RGN}\ \theta_2\ \beta]\!]) \\
&\quad = \mathsf{U}(\overline{\forall}\overline{\alpha}_\beta.\,\mathsf{U}(\mathbb{T}[\![\mathsf{RGN}\ \theta_1\ \beta]\!] \multimap \mathbb{T}[\![\mathsf{RGN}\ \theta_2\ \beta]\!])) \\
&\quad = \mathsf{U}(\overline{\forall}\overline{\alpha}_\beta.\,\mathsf{U}(\mathsf{U}(\mathbb{T}[\![\theta_1]\!] \multimap \mathsf{L}(\mathbb{T}[\![\theta_1]\!] \otimes \mathbb{T}[\![\beta]\!])) \multimap \mathsf{U}(\mathsf{U}(\mathbb{T}[\![\theta_2]\!] \multimap \mathsf{L}(\mathbb{T}[\![\theta_2]\!] \otimes \mathbb{T}[\![\beta]\!])))))) \\
&\quad = \mathsf{U}(\overline{\forall}\overline{\alpha}_\beta.\,\mathsf{U}(\mathsf{U}(\mathbb{T}[\![\theta_1]\!] \multimap \mathsf{L}(\mathbb{T}[\![\theta_1]\!] \otimes \mathsf{U}\overline{\alpha}_\beta)) \multimap \mathsf{U}(\mathsf{U}(\mathbb{T}[\![\theta_2]\!] \multimap \mathsf{L}(\mathbb{T}[\![\theta_2]\!] \otimes \mathsf{U}\overline{\alpha}_\beta))))) \\
&\quad \equiv_\alpha \mathsf{U}(\overline{\forall\beta}.\,\mathsf{U}(\mathsf{U}(\mathbb{T}[\![\theta_1]\!] \multimap \mathsf{L}(\mathbb{T}[\![\theta_1]\!] \otimes \mathsf{U}\overline{\beta})) \multimap \mathsf{U}(\mathsf{U}(\mathbb{T}[\![\theta_2]\!] \multimap \mathsf{L}(\mathbb{T}[\![\theta_2]\!] \otimes \mathsf{U}\overline{\beta})))))
\end{aligned}
$$

While the type is rather verbose, we will see that it is quite easy to construct a value of this type given a value of the type $\overline{\exists}\beta.\,\mathbf{Iso}(\mathbb{T}[\![\theta_2]\!], \mathsf{L}(\mathbb{T}[\![\theta_1]\!] \otimes \beta))$. This latter type may be seen to be an isomorphism that witnesses the fact that the stack indexed by θ_1 is a substack of the stack indexed by θ_2. This correspondence will be made more precise in the translation of the `letRGN` operation.

Translations yielding qualifier, pre-type, type, and region contexts

Type and index contexts

$$\begin{aligned}
\mathbb{D}[\![\cdot]\!] &= \cdot \\
\mathbb{D}[\![\Delta, \alpha]\!] &= \mathbb{D}[\![\Delta]\!], \overline{\alpha}_{\alpha} \\
\mathbb{D}[\![\Delta, \vartheta]\!] &= \mathbb{D}[\![\Delta]\!], \alpha_{\vartheta}
\end{aligned}$$

Translations yielding value contexts

Value contexts

$$\mathbb{G}\left[\!\!\left[\frac{}{\Delta \vdash_{\text{vctxt}} \cdot}\right]\!\!\right] = \cdot$$

$$\mathbb{G}\left[\!\!\left[\frac{\Delta \vdash_{\text{vctxt}} \Gamma \qquad x \notin dom(\Gamma) \qquad \Delta \vdash_{\text{type}} \tau}{\Delta \vdash_{\text{vctxt}} \Gamma, x{:}\tau}\right]\!\!\right] = \mathbb{G}[\![\Gamma]\!], x{:}\mathbb{T}[\![\tau]\!]$$

Figure 4.29: Translation from $\mathsf{F}^{\mathsf{RGN}}$ to rgnURAL (contexts)

Contexts Figure 4.29 extends the index and type translations to contexts in the obvious manner.

Functional terms With the translation of $\mathsf{F}^{\mathsf{RGN}}$ indices, types, and contexts in place, the translation of $\mathsf{F}^{\mathsf{RGN}}$ expressions follows almost directly. Figures 4.30, 4.31, and 4.32 give the straightforward translation of the introduction and elimination forms for the functional types in $\mathsf{F}^{\mathsf{RGN}}$. Note that every introduction form qualifies the introduced expression with the U qualifier.

Before turning to the translation of the RGN operations, it is useful to further explain how the type polymorphism in $\mathsf{F}^{\mathsf{RGN}}$ is handled under the translation to rgnURAL. One might ask: "Why is it necessary to translate $\mathsf{F}^{\mathsf{RGN}}$ type variables to rgnURAL pre-type variables?" To answer this question, consider the following

Translations yielding expressions

Expressions

$$\mathbb{E}\left[\!\!\left[\frac{\vdash_{\text{ctxt}} \Delta;\Gamma}{\Delta;\Gamma \vdash_{\text{exp}} \mathfrak{i} : \mathsf{Int}}\right]\!\!\right] \;=\; {}^{\cup}\mathfrak{i}$$

$$\mathbb{E}\left[\!\!\left[\frac{\Delta;\Gamma \vdash_{\text{exp}} e_1 : \mathsf{Int} \qquad \Delta;\Gamma \vdash_{\text{exp}} e_2 : \mathsf{Int}}{\Delta;\Gamma \vdash_{\text{exp}} e_1 \oplus e_2 : \mathsf{Int}}\right]\!\!\right] \;=\; {}^{\cup}(\mathbb{E}[\![e_1]\!] \oplus \mathbb{E}[\![e_2]\!])$$

$$\mathbb{E}\left[\!\!\left[\frac{\Delta;\Gamma \vdash_{\text{exp}} e_1 : \mathsf{Int} \qquad \Delta;\Gamma \vdash_{\text{exp}} e_2 : \mathsf{Int}}{\Delta;\Gamma \vdash_{\text{exp}} e_1 \oplus e_2 : \mathsf{Int}}\right]\!\!\right] \;=\; {}^{\cup}(\mathbb{E}[\![e_1]\!] \oslash \mathbb{E}[\![e_2]\!])$$

$$\mathbb{E}\left[\!\!\left[\frac{\vdash_{\text{ctxt}} \Delta;\Gamma}{\Delta;\Gamma \vdash_{\text{exp}} \mathfrak{b} : \mathsf{Bool}}\right]\!\!\right] \;=\; {}^{\cup}\mathfrak{b}$$

$$\mathbb{E}\left[\!\!\left[\frac{\Delta;\Gamma \vdash_{\text{exp}} e_b : \mathsf{Bool} \qquad \Delta;\Gamma \vdash_{\text{exp}} e_t : \tau \qquad \Delta;\Gamma \vdash_{\text{exp}} e_f : \tau}{\Delta;\Gamma \vdash_{\text{exp}} \texttt{if } e_b \texttt{ then } e_t \texttt{ else } e_f : \tau}\right]\!\!\right] \;=\; \texttt{if } \mathbb{E}[\![e_b]\!] \texttt{ then } \mathbb{E}[\![e_t]\!] \texttt{ else } \mathbb{E}[\![e_f]\!]$$

Figure 4.30: Translation from $\mathsf{F}^{\mathsf{RGN}}$ to rgnURAL (terms (I))

Translations yielding expressions

Expressions

$$\mathbb{E}\left[\!\!\left[\frac{\vdash_{\text{ctxt}} \Delta;\Gamma \qquad x \in dom(\Gamma) \qquad \Gamma(x) = \tau}{\Delta;\Gamma \vdash_{\text{exp}} x : \tau}\right]\!\!\right] \;=\; x$$

$$\mathbb{E}\left[\!\!\left[\frac{\Delta;\Gamma, x{:}\tau_x \vdash_{\text{exp}} e : \tau}{\Delta;\Gamma \vdash_{\text{exp}} \lambda x{:}\tau_x.\, e : \tau_x \to \tau}\right]\!\!\right] \;=\; \mathsf{U}(\lambda x{:}\mathbb{T}[\![\tau]\!]\,.\,\mathbb{E}[\![e]\!])$$

$$\mathbb{E}\left[\!\!\left[\frac{\Delta;\Gamma \vdash_{\text{exp}} e_f : \tau_x \to \tau \quad \Delta;\Gamma \vdash_{\text{exp}} e_a : \tau_x}{\Delta;\Gamma \vdash_{\text{exp}} e_f\ e_a : \tau}\right]\!\!\right] \;=\; \mathbb{E}[\![e_f]\!]\ \mathbb{E}[\![e_a]\!]$$

$$\mathbb{E}\left[\!\!\left[\frac{\vdash_{\text{ctxt}} \Delta;\Gamma \quad \Delta;\Gamma \vdash_{\text{exp}} e_i : \tau_i \;\; {}^{i\in 1\ldots n}}{\Delta;\Gamma \vdash_{\text{exp}} \langle e_1, \ldots, e_n\rangle : \tau_1 \times \cdots \times \tau_n}\right]\!\!\right] \;=\; \mathsf{U}\langle \mathbb{E}[\![e_1]\!], \ldots, \mathbb{E}[\![e_n]\!]\rangle$$

$$\mathbb{E}\left[\!\!\left[\frac{\Delta;\Gamma \vdash_{\text{exp}} e : \tau_1 \times \cdots \times \tau_n \quad 1 \le i \le n}{\Delta;\Gamma \vdash_{\text{exp}} \mathtt{sel}_i\ e : \tau_i}\right]\!\!\right] \;=\; \mathtt{let}\ \langle x_1, \ldots, x_n\rangle = \mathbb{E}[\![e]\!]\ \mathtt{in}\ x_i$$

where $x_1, \ldots, x_n$ fresh

Figure 4.31: Translation from $\mathsf{F}^{\mathsf{RGN}}$ to rgnURAL (terms (II))

Translations yielding expressions

Expressions

$$\mathbb{E}\left[\!\!\left[\frac{\vdash_{\text{ctxt}} \Delta;\Gamma \qquad \Delta,\alpha;\Gamma \vdash_{\text{exp}} e : \tau}{\Delta;\Gamma \vdash_{\text{exp}} \Lambda\alpha.\, e : \forall\alpha.\, \tau}\right]\!\!\right] = {}^{\mathsf{U}}(\Lambda\overline{\alpha}_{\alpha}.\, \mathbb{E}[\![e]\!])$$

$$\mathbb{E}\left[\!\!\left[\frac{\Delta;\Gamma \vdash_{\text{exp}} e_f : \forall\alpha.\, \tau \qquad \Delta \vdash_{\text{type}} \tau_a}{\Delta;\Gamma \vdash_{\text{exp}} e_f\ [\tau_a] : \tau[\tau_a/\alpha]}\right]\!\!\right] = \mathbb{E}[\![e_f]\!]\ [\mathbb{P}[\![\tau_a]\!]]$$

$$\mathbb{E}\left[\!\!\left[\frac{\Delta;\Gamma \vdash_{\text{exp}} e_a : \tau_x \qquad \Delta;\Gamma, x{:}\tau_x \vdash_{\text{exp}} e_b : \tau}{\Delta;\Gamma \vdash_{\text{exp}} \texttt{let}\ x = e_a\ \texttt{in}\ e_b : \tau}\right]\!\!\right] = \texttt{let}\ x = \mathbb{E}[\![e_a]\!]\ \texttt{in}\ \mathbb{E}[\![e_b]\!]$$

Figure 4.32: Translation from $\mathsf{F}^{\mathsf{RGN}}$ to rgnURAL (terms (III))

polymorphic function in $\mathsf{F}^{\mathsf{RGN}}$:

$$\Lambda\alpha.\, \lambda x{:}\alpha.\, \langle x, x\rangle$$

with the type

$$\forall\alpha.\, \alpha \to (\alpha \times \alpha).$$

Note that this function duplicates the argument x of type α. If we were to translate $\mathsf{F}^{\mathsf{RGN}}$ type variables to $\mathsf{F}^{\mathsf{RGN}}$ type variables, then we would expect the translation of the function above to be:

$$^{\mathsf{U}}(\Lambda\alpha.\, ^{\mathsf{U}}(\lambda x{:}\alpha.\, ^{\mathsf{U}}\langle x, x\rangle)).$$

However, it is not possible to construct a typing derivation for this rgnURAL expression; in particular, it is not possible to show that this expression has the type

$$^{\mathsf{U}}(\overline{\forall}\alpha.\, ^{\mathsf{U}}(\alpha \multimap\, ^{\mathsf{U}}(\alpha \otimes \alpha)).$$

The reason that there is no typing derivation for this rgnURAL expression is that the variable x, with type α, is used in two sub-expressions in the body of the function. Hence, the value context $\cdot, x{:}\alpha$ must be split into two contexts, each of the form $\cdot, x{:}\alpha$; more precisely, any typing derivation would require a sub-derivation of the judgment

$$\cdot, \alpha \vdash \cdot, x{:}\alpha \rightsquigarrow \cdot, x{:}\alpha \boxdot \cdot, x{:}\alpha.$$

Such a derivation would, in turn, require a sub-derivation of the judgment

$$\cdot, \alpha \vdash \alpha \preceq \mathsf{R}.$$

From the static semantics of rgnURAL given in the previous section, we recall the rules in Figure 4.15 for the judgment $\Delta \vdash \tau \preceq q'$, which judges that a type is

bounded by a qualifier. Note that we include the rule

$$\frac{\Delta \vdash_{\text{type}} \tau}{\Delta \vdash \tau \preceq \mathsf{L}}$$

which may be instantiated as

$$\frac{\Delta \vdash_{\text{type}} \alpha}{\Delta \vdash \alpha \preceq \mathsf{L}.}$$

This rule conservatively judges that a type variable is bounded by the L qualifier; this is sound, as L is the top of the qualifier lattice. Furthermore, note that this is the *only* way to judge that a type variable is bounded by a qualifier. It would be unsound to include the rule:

$$\frac{\Delta \vdash_{\text{type}} \alpha}{\Delta \vdash \alpha \preceq \mathsf{R},}$$

since α might later be instantiated with an L-qualified pre-type, which can't be bounded by R.

Returning to the translation of the polymorphic function, we must ensure that the type of the variable x is translated to a type which may be judged to be bounded by R. To do so, we translate from $\mathsf{F}^{\mathsf{RGN}}$ type abstraction and instantiation to $\mathsf{rgnURAL}$ pre-type abstraction and instantiation, while the $\mathbb{T}[\![\Delta \vdash_{\text{type}} \tau]\!]$ translation ensures that the translation of every $\mathsf{F}^{\mathsf{RGN}}$ type is a U qualified pre-type. Hence, the translation of the polymorphic function above is:

$$^{\mathsf{U}}(\Lambda\overline{\alpha}.\,^{\mathsf{U}}(\lambda x{:}^{\mathsf{U}}\overline{\alpha}.\,^{\mathsf{U}}\langle x, x\rangle)).$$

We may easily construct a typing derivation for this $\mathsf{rgnURAL}$ expression, showing that it has the type

$$^{\mathsf{U}}(\overline{\forall}\overline{\alpha}.\,^{\mathsf{U}}(^{\mathsf{U}}\overline{\alpha} \multimap\, ^{\mathsf{U}}(^{\mathsf{U}}\overline{\alpha} \otimes\, ^{\mathsf{U}}\overline{\alpha})),$$

Translations yielding expressions

Expressions

$$\mathbb{E}\left[\!\!\left[\frac{\Delta \vdash_{\text{index}} \theta \qquad \Delta \vdash_{\text{type}} \tau \qquad \Delta;\Gamma \vdash_{\text{exp}} v : \tau}{\Delta;\Gamma \vdash_{\text{exp}} \texttt{returnRGN}\ [\theta]\ [\tau]\ v : \mathsf{RGN}\ \theta\ \tau}\right]\!\!\right] =$$

$$\texttt{let}\ res_v{:}\mathbb{T}[\![\tau]\!] = \mathbb{E}[\![v]\!]\ \texttt{in}$$

$$^{\mathsf{U}}(\lambda stk{:}\mathbb{T}[\![\theta]\!]\,.\,{}^{\mathsf{L}}\langle stk, res_v\rangle)$$

$$\mathbb{E}\left[\!\!\left[\frac{\Delta \vdash_{\text{index}} \theta \qquad \Delta \vdash_{\text{type}} \tau_a \qquad \Delta \vdash_{\text{type}} \tau_b \qquad \Delta;\Gamma \vdash_{\text{exp}} v_a : \mathsf{RGN}\ \theta\ \tau_a \qquad \Delta;\Gamma \vdash_{\text{exp}} v_f : \tau_a \to \mathsf{RGN}\ \theta\ \tau_b}{\Delta;\Gamma \vdash_{\text{exp}} \texttt{thenRGN}\ [\theta]\ [\tau_a]\ [\tau_b]\ v_a\ v_f : \mathsf{RGN}\ \theta\ \tau_b}\right]\!\!\right] =$$

$$\texttt{let}\ f_a{:}\mathbb{T}[\![\mathsf{RGN}\ \theta\ \tau]\!] = \mathbb{E}[\![v_a]\!]\ \texttt{in}$$

$$\texttt{let}\ f_f{:}\mathbb{T}[\![\tau \to \mathsf{RGN}\ \theta\ \tau]\!] = \mathbb{E}[\![v_f]\!]\ \texttt{in}$$

$$^{\mathsf{U}}(\lambda stk{:}\mathbb{T}[\![\theta]\!]\,.\,\texttt{let}\ \langle stk, res_a\rangle = f_a\ stk\ \texttt{in}\ f_f\ res_a\ stk)$$

Figure 4.33: Translation from $\mathsf{F}^{\mathsf{RGN}}$ to rgnURAL (commands (I))

since

$$\cdot, \overline{\alpha} \vdash {}^{\mathsf{U}}\overline{\alpha} \preceq \mathsf{R}$$

may be shown directly.

RGN monad commands Next, we turn to the translation of the RGN monad commands, given in Figures 4.33–4.37. The only interesting decision left to be made is to choose the representation of a stack of regions in rgnURAL. As we noted above, there is an obvious representation that suffices: a sequence of region capabilities formed out of nested tuples.

Translations yielding expressions

Expressions

$$\mathbb{E}\left[\!\!\left[\frac{\Delta \vdash_{\text{index}} \theta \qquad \Delta \vdash_{\text{type}} \tau \qquad \Delta;\Gamma \vdash_{\text{exp}} v_h : \mathsf{RGNHnd}\ \theta \qquad \Delta;\Gamma \vdash_{\text{exp}} v_\star : \tau}{\Delta;\Gamma \vdash_{\text{exp}} \texttt{newRGNRef}\ [\theta]\ [\tau]\ v_h\ v_\star : \mathsf{RGN}\ \theta\ (\mathsf{RGNRef}\ \theta\ \tau)}\right]\!\!\right] =$$

$$\begin{aligned}
&\texttt{let}\ \mathit{phnd}{:}\mathbb{T}[\![\mathsf{RGNHnd}\ \theta]\!] = \mathbb{E}[\![v_h]\!]\ \texttt{in}\\
&\texttt{let}\ x_\star{:}\mathbb{T}[\![\tau]\!] = \mathbb{E}[\![v_\star]\!]\ \texttt{in}\\
&{}^{\mathsf{U}}(\lambda \mathit{stk}{:}\mathbb{T}[\![\theta]\!]\,.\,\texttt{let}\ \texttt{pack}(\varrho, \langle \mathit{iso}, \mathit{hnd}\rangle) = \mathit{phnd}\ \texttt{in}\\
&\qquad\qquad \texttt{let}\ \texttt{pack}(\alpha, \langle \mathit{prj}, \mathit{inj}\rangle) = \mathit{iso}\ \texttt{in}\\
&\qquad\qquad \texttt{let}\ \langle \mathit{stk}_\alpha, \mathit{cap}\rangle = \mathit{prj}\ \mathit{stk}\ \texttt{in}\\
&\qquad\qquad \texttt{let}\ \langle \mathit{cap}, \mathit{ref}\rangle = {}^{\mathsf{U}}\texttt{new}\ \mathit{cap}\ \mathit{hnd}\ x_\star\ \texttt{in}\\
&\qquad\qquad \texttt{let}\ \mathit{pref} = {}^{\mathsf{U}}\texttt{pack}(\varrho, {}^{\mathsf{U}}\langle \mathit{iso}, \mathit{ref}\rangle)\ \texttt{in}\\
&\qquad\qquad \texttt{let}\ \mathit{stk}{:}\mathbb{T}[\![\theta]\!] = \mathit{inj}\ {}^{\mathsf{L}}\langle \mathit{stk}_\alpha, \mathit{cap}\rangle\ \texttt{in}\\
&\qquad\qquad {}^{\mathsf{L}}\langle \mathit{stk}, \mathit{pref}\rangle)
\end{aligned}$$

Figure 4.34: Translation from $\mathsf{F}^{\mathsf{RGN}}$ to rgnURAL (commands (II))

The translation of `returnRGN` and `thenRGN` follow directly from our stack-passing interpretation of $\mathsf{RGN}\ \theta\ \tau$ types (see Figure 4.33).

The translation of `newRGNRef` in Figure 4.34 shows how the isomorphisms in the translation of the RGNHnd and RGNRef are used. Since `newRGNRef` is a RGN monad command, it is translated to a stack-passing function. Within the body of the function, the (translation of the) handle is unpacked, revealing the region ϱ, the isomorphism, and the rgnURAL handle. Note that the variables *hnd*, *inj*, and *prj* have the types ${}^{\mathsf{U}}(\overline{\mathsf{Hnd}}\ \varrho)$, ${}^{\mathsf{U}}({}^{\mathsf{L}}(\alpha \otimes {}^{\mathsf{L}}(\overline{\mathsf{Cap}}\ \varrho)) \multimap \mathbb{T}[\![\theta]\!])$, and ${}^{\mathsf{U}}(\mathbb{T}[\![\theta]\!] \multimap {}^{\mathsf{L}}(\alpha \otimes {}^{\mathsf{L}}(\overline{\mathsf{Cap}}\ \varrho)))$, respectively.

The *prj* function is used to split the stack (*stk* of type $\mathbb{T}[\![\theta]\!]$) into a capability (*cap* of type $\mathsf{L}(\overline{\mathsf{Cap}}\ \varrho)$) and some "rest of the stack" (stk_α of type α). Having both the capability and the handle for the region ϱ, we are able to allocate a new reference (*ref* of type $\mathsf{U}(\overline{\mathsf{Ref}}\ \varrho\ \mathbb{T}[\![\tau]\!])$). However, before returning the new reference, we must package it with an isomorphism that witnesses the fact that the capability for ϱ may be found within the stack $\mathbb{T}[\![\theta]\!]$. Conveniently, the isomorphism that came packaged with the handle witnesses precisely the same fact. Finally, the *inj* function is used to combine the capability and the "rest of the stack" back into a stack of type $\mathbb{T}[\![\theta]\!]$.

Figure 4.35 gives the translations of `readRGNRef` and `writeRGNRef`. These translations work in much the same manner as the translation of `newRGNRef`. The (translation of the) reference is unpacked, revealing the region ϱ, the isomorphism and the rgnURAL reference. Next, the isomorphism is used to split out the capability for the region ϱ. The capability is used to access the reference. Finally, the isomorphism is used to combine the capability with the remainder of the stack.

The translation of `letRGN` is the most complicated, but breaks down into conceptually simple components (see Figure 4.36). We bracket the execution of the inner computation with a `newrgn`/`freergn` pair, creating and destroying a new region. We construct the representation of the new stack for the inner computation (stk_2 of type $\mathsf{L}(\mathbb{T}[\![\theta_1]\!] \otimes \mathsf{L}(\overline{\mathsf{Cap}}\ \varrho))$) by pairing the old stack (stk_1) with the new region capability (*cap*). In order to package the region handle (*phnd*) in the manner expected by the translation, we must construct an isomorphism witnessing the relationship between the new region capability and the new stack. Note that we carefully chose the isomorphism types so that the identity function suffices as a witness, where the old stack type $\mathbb{T}[\![\theta_1]\!]$ serves as the "slack".

Translations yielding expressions

Expressions

$$\mathbb{E}\left[\!\!\left[\frac{\Delta \vdash_{\text{index}} \theta \qquad \Delta \vdash_{\text{type}} \tau \qquad \Delta;\Gamma \vdash_{\text{exp}} v_r : \mathsf{RGNRef}\ \theta\ \tau}{\Delta;\Gamma \vdash_{\text{exp}} \texttt{readRGNRef}\ [\theta]\ [\tau]\ v_r : \mathsf{RGN}\ \theta\ \tau}\right]\!\!\right] =$$

$$\begin{array}{l}
\texttt{let}\ \mathit{pref}{:}\mathbb{T}[\![\mathsf{RGNRef}\ \theta\ \tau]\!] = \mathbb{E}[\![v_r]\!]\ \texttt{in} \\
{}^{\mathsf{U}}(\lambda \mathit{stk}{:}\mathbb{T}[\![\theta]\!]\,.\,\texttt{let}\ \texttt{pack}(\varrho, \langle \texttt{pack}(\alpha, \langle \mathit{prj}, \mathit{inj}\rangle), \mathit{ref}\rangle) = \mathit{ref}\ \texttt{in} \\
\qquad\qquad \texttt{let}\ \langle \mathit{stk}_\alpha, \mathit{cap}\rangle = \mathit{prj}\ \mathit{stk}\ \texttt{in} \\
\qquad\qquad \texttt{let}\ \langle \mathit{cap}, \mathit{ref}', \mathit{res}\rangle = \texttt{read}\ \mathit{cap}\ \mathit{ref}\ \texttt{in} \\
\qquad\qquad \texttt{let}\ \mathit{stk}{:}\mathbb{T}[\![\theta]\!] = \mathit{inj}\ {}^{\mathsf{L}}\langle \mathit{stk}_\alpha, \mathit{cap}\rangle\ \texttt{in} \\
\qquad\qquad {}^{\mathsf{L}}\langle \mathit{stk}, \mathit{res}\rangle)
\end{array}$$

$$\mathbb{E}\left[\!\!\left[\frac{\Delta \vdash_{\text{index}} \theta \qquad \Delta \vdash_{\text{type}} \tau \qquad \Delta;\Gamma \vdash_{\text{exp}} v_r : \mathsf{RGNRef}\ \theta\ \tau \qquad \Delta;\Gamma \vdash_{\text{exp}} v_\star : \tau}{\Delta;\Gamma \vdash_{\text{exp}} \texttt{writeRGNRef}\ [\theta]\ [\tau]\ v_r\ v_\star : \mathsf{1}_\times}\right]\!\!\right] =$$

$$\begin{array}{l}
\texttt{let}\ \mathit{pref}{:}\mathbb{T}[\![\mathsf{RGNRef}\ \theta\ \tau]\!] = \mathbb{E}[\![v_r]\!]\ \texttt{in} \\
\texttt{let}\ x_\star{:}\mathbb{T}[\![\tau]\!] = \mathbb{E}[\![v_\star]\!]\ \texttt{in} \\
{}^{\mathsf{U}}(\lambda \mathit{stk}{:}\mathbb{T}[\![\theta]\!]\,.\,\texttt{let}\ \texttt{pack}(\varrho, \langle \texttt{pack}(\alpha, \langle \mathit{prj}, \mathit{inj}\rangle), \mathit{ref}\rangle) = \mathit{ref}\ \texttt{in} \\
\qquad\qquad \texttt{let}\ \langle \mathit{stk}_\alpha, \mathit{cap}\rangle = \mathit{prj}\ \mathit{stk}\ \texttt{in} \\
\qquad\qquad \texttt{let}\ \langle \mathit{cap}, \mathit{ref}'\rangle = \texttt{write}\ \mathit{cap}\ \mathit{ref}\ x_\star\ \texttt{in} \\
\qquad\qquad \texttt{let}\ \mathit{stk}{:}\mathbb{T}[\![\theta]\!] = \mathit{inj}\ {}^{\mathsf{L}}\langle \mathit{stk}_\alpha, \mathit{cap}\rangle\ \texttt{in} \\
\qquad\qquad {}^{\mathsf{L}}\langle \mathit{stk}, {}^{\mathsf{U}}\langle\rangle\rangle)
\end{array}$$

Figure 4.35: Translation from $\mathsf{F}^{\mathsf{RGN}}$ to rgnURAL (commands (III))

Translations yielding expressions

Expressions

$$\mathbb{E}\left[\!\!\left[\frac{\Delta \vdash_{\text{index}} \theta_1 \qquad \Delta \vdash_{\text{type}} \tau}{\Delta;\Gamma \vdash_{\text{exp}} v : \forall\vartheta.\,\mathsf{RGNPf}(\theta_1 \preceq \vartheta_2) \to \mathsf{RGNHnd}\ \vartheta_2 \to \mathsf{RGN}\ \vartheta_2\ \tau}{\Delta;\Gamma \vdash_{\text{exp}} \texttt{letRGN}\ [\theta_1]\ [\tau]\ v : \mathsf{RGN}\ \theta_1\ \tau}\right]\!\!\right] =$$

$$\begin{array}{l}
\texttt{let}\ f_v{:}\mathbb{T}[\![\forall\vartheta.\,\mathsf{RGNPf}(\theta_1 \preceq \vartheta_2) \to \mathsf{RGNHnd}\ \vartheta_2 \to \mathsf{RGN}\ \vartheta_2\ \tau]\!] = \mathbb{E}[\![v]\!]\ \texttt{in} \\
{}^{\mathsf{U}}(\lambda stk_1{:}\mathbb{T}[\![\theta_1]\!]\,.\,\texttt{let}\ \texttt{pack}(\varrho, \langle cap, hnd\rangle) = {}^{\mathsf{L},\mathsf{U}}\texttt{newrgn}\ \texttt{in} \\
\qquad \texttt{let}\ id = {}^{\mathsf{U}}(\lambda stk_2{:}{}^{\mathsf{L}}(\mathbb{T}[\![\theta_1]\!] \otimes {}^{\mathsf{L}}(\overline{\mathsf{Cap}}\ \varrho)).\ stk_2)\ \texttt{in} \\
\qquad \texttt{let}\ phnd = {}^{\mathsf{U}}\texttt{pack}(\varrho, {}^{\mathsf{U}}\langle {}^{\mathsf{U}}\texttt{pack}(\mathbb{T}[\![\theta_1]\!], \langle id, id\rangle), hnd\rangle)\ \texttt{in} \\
\qquad \texttt{let}\ ppf = {}^{\mathsf{U}}\texttt{pack}({}^{\mathsf{L}}(\overline{\mathsf{Cap}}\ \varrho), \langle id, id\rangle)\ \texttt{in} \\
\qquad \texttt{let}\ wit = {}^{\mathsf{U}}(\Lambda\overline{\beta}. \\
\qquad\qquad {}^{\mathsf{U}}(\lambda g{:}{}^{\mathsf{U}}(\mathbb{T}[\![\theta_1]\!] \multimap {}^{\mathsf{L}}(\mathbb{T}[\![\theta_1]\!] \otimes {}^{\mathsf{U}}\overline{\beta})). \\
\qquad\qquad\quad {}^{\mathsf{U}}(\lambda stk_2{:}{}^{\mathsf{L}}(\mathbb{T}[\![\theta_1]\!] \otimes {}^{\mathsf{L}}(\overline{\mathsf{Cap}}\ \varrho)). \\
\qquad\qquad\qquad \texttt{let}\ \texttt{pack}(\alpha, \langle spl, cmb\rangle) = ppf\ \texttt{in} \\
\qquad\qquad\qquad \texttt{let}\ \langle stk_1, stk_\alpha\rangle = spl\ stk_2\ \texttt{in} \\
\qquad\qquad\qquad \texttt{let}\ \langle stk_1, res\rangle = g\ stk_1\ \texttt{in} \\
\qquad\qquad\qquad \texttt{let}\ stk_2 = cmb\ {}^{\mathsf{L}}\langle stk_1, stk_\alpha\rangle\ \texttt{in} \\
\qquad\qquad\qquad {}^{\mathsf{L}}\langle stk_2, res\rangle))))\ \texttt{in} \\
\qquad \texttt{let}\ stk_2{:}{}^{\mathsf{L}}(\mathbb{T}[\![\theta_1]\!] \otimes {}^{\mathsf{L}}(\overline{\mathsf{Cap}}\ \varrho)) = {}^{\mathsf{L}}\langle stk_1, cap\rangle\ \texttt{in} \\
\qquad \texttt{let}\ \langle stk_2, res\rangle = f_v\ [{}^{\mathsf{L}}(\mathbb{T}[\![\theta_1]\!] \otimes {}^{\mathsf{L}}(\overline{\mathsf{Cap}}\ \varrho))]\ wit\ phnd\ stk_2\ \texttt{in} \\
\qquad \texttt{let}\ \langle stk_1, cap\rangle = stk_2\ \texttt{in} \\
\qquad \texttt{let}\ \langle\rangle = \texttt{freergn}\ {}^{\mathsf{L}}\langle cap, hnd\rangle\ \texttt{in} \\
\qquad {}^{\mathsf{L}}\langle stk_1, res\rangle)
\end{array}$$

Figure 4.36: Translation from $\mathsf{F}^{\mathsf{RGN}}$ to rgnURAL (commands (IV))

The final component of the translation is the definition of a term that acts like $\mathsf{RGNPf}(\theta_1 \preceq \vartheta_2)$. Recall that $\mathsf{RGNPf}(\theta_1 \preceq \vartheta_2)$ is used to coerce, for any $\mathsf{F}^{\mathsf{RGN}}$ type β, a computation from the type $\mathsf{RGN}\ \theta_1\ \beta$ to the type $\mathsf{RGN}\ \vartheta_2\ \beta$. Under the translation, this means coercing a function of the type $^{\mathsf{U}}(\mathbb{T}[\![\theta_1]\!] \multimap {}^{\mathsf{L}}(\mathbb{T}[\![\theta_1]\!] \otimes {}^{\mathsf{U}}\overline{\beta}))$ to a function of the type $^{\mathsf{U}}({}^{\mathsf{L}}(\mathbb{T}[\![\theta_1]\!] \otimes {}^{\mathsf{L}}(\overline{\mathsf{Cap}}\ \varrho)) \multimap {}^{\mathsf{L}}({}^{\mathsf{L}}(\mathbb{T}[\![\theta_1]\!] \otimes {}^{\mathsf{L}}(\overline{\mathsf{Cap}}\ \varrho)) \otimes {}^{\mathsf{U}}\overline{\beta}))$, since $^{\mathsf{L}}(\mathbb{T}[\![\theta_1]\!] \otimes {}^{\mathsf{L}}(\overline{\mathsf{Cap}}\ \varrho))$ is the type of representation of the stack indexed by ϑ_2.

This coercion is easily accomplished, as seen by the definition of the *wit* term. In order to more easily relate the translation in this section to extensions in the following chapter, we factor the construction of the coercion into two pieces.

We first define a general isomorphism (*ppf*) witnessing the relationship between the old stack and the new stack; as in the definition of *phnd*, the isomorphism types are carefully chosen so that the identity function suffices as a witness, where the region capability type ($^{\mathsf{L}}(\overline{\mathsf{Cap}}\ \varrho)$)serves as the "slack."

We can see the isomorphism in action in the definition of the *wit* term. The result of the coercion works by first splitting the representation of the new stack (stk_2 of type $^{\mathsf{L}}(\mathbb{T}[\![\theta_1]\!] \otimes {}^{\mathsf{L}}(\overline{\mathsf{Cap}}\ \varrho))$) into a representation of the old stack (stk_1 of type $\mathbb{T}[\![\theta_1]\!]$) and some "slack" ($stk_\alpha$ of type α). Next, the function g, corresponding to a $\mathsf{RGN}\ \theta_1\ \beta$ computation, is applied to the representation of the old stack and returns a fresh copy of the representation of the old stack and a result value. Finally, a fresh copy of the representation of the new stack is constructed, by combining the old stack with the "slack", in order to return a stack/value pair of the type $^{\mathsf{L}}({}^{\mathsf{L}}(\mathbb{T}[\![\theta_1]\!] \otimes {}^{\mathsf{L}}(\overline{\mathsf{Cap}}\ \varrho)) \otimes {}^{\mathsf{U}}\overline{\beta})$.

Putting all of these pieces together, we have the translation in Figure 4.36.

Translations yielding expressions

Expressions

$$\mathbb{E}\left[\!\!\left[\frac{\Delta \vdash_{\text{type}} \tau \qquad \Delta;\Gamma \vdash_{\text{exp}} v : \forall\vartheta.\,\mathsf{RGNHnd}\ \vartheta \to \mathsf{RGN}\ \vartheta\ \tau}{\Delta;\Gamma \vdash_{\text{exp}} \mathtt{runRGN}\ [\tau]\ v : \tau}\right]\!\!\right] =$$

$$\begin{aligned}
&\mathtt{let}\ f_v{:}\mathbb{T}[\![\forall\vartheta.\,\mathsf{RGNHnd}\ \vartheta \to \mathsf{RGN}\ \vartheta\ \tau]\!] = \mathbb{E}[\![v]\!]\ \mathtt{in}\\
&\mathtt{let}\ \mathtt{pack}(\varrho, \langle cap, hnd\rangle) = {}^{\mathsf{L},\mathsf{U}}\mathtt{newrgn}\ \mathtt{in}\\
&\mathtt{let}\ id = {}^{\mathsf{U}}(\lambda stk{:}{}^{\mathsf{L}}({}^{\mathsf{U}}\mathbf{1}_{\otimes} \otimes {}^{\mathsf{L}}(\overline{\mathsf{Cap}}\ \varrho)).\, stk)\ \mathtt{in}\\
&\mathtt{let}\ phnd = {}^{\mathsf{U}}\mathtt{pack}(\varrho, {}^{\mathsf{U}}\langle {}^{\mathsf{U}}\mathtt{pack}({}^{\mathsf{U}}\mathbf{1}_{\otimes}, \langle id, id\rangle), hnd\rangle)\ \mathtt{in}\\
&\mathtt{let}\ stk{:}{}^{\mathsf{L}}({}^{\mathsf{U}}\mathbf{1}_{\otimes} \otimes {}^{\mathsf{L}}(\overline{\mathsf{Cap}}\ \varrho)) = {}^{\mathsf{L}}\langle {}^{\mathsf{U}}\langle\rangle, cap\rangle\ \mathtt{in}\\
&\mathtt{let}\ \langle stk, res\rangle = f_v\ [{}^{\mathsf{L}}({}^{\mathsf{U}}\mathbf{1}_{\otimes} \otimes {}^{\mathsf{L}}(\overline{\mathsf{Cap}}\ \varrho))]\ phnd\ stk\ \mathtt{in}\\
&\mathtt{let}\ \langle\langle\rangle, cap\rangle = stk\ \mathtt{in}\\
&\mathtt{let}\ \langle\rangle = \mathtt{freergn}\ {}^{\mathsf{L}}\langle cap, hnd\rangle\ \mathtt{in}\\
&res
\end{aligned}$$

Figure 4.37: Translation from $\mathsf{F}^{\mathsf{RGN}}$ to rgnURAL (terms (IV))

Translations yielding expressions

Expressions

$$\mathbb{E}\left[\!\!\left[\frac{\vdash_{\text{ctxt}} \Delta;\Gamma \qquad \Delta,\vartheta;\Gamma \vdash_{\text{exp}} e : \tau}{\Delta;\Gamma \vdash_{\text{exp}} \Lambda\vartheta.\, e : \forall\vartheta.\, \tau}\right]\!\!\right] \;=\; {}^{\mathsf{U}}(\Lambda\alpha_{\vartheta}.\, \mathbb{E}[\![e]\!])$$

$$\mathbb{E}\left[\!\!\left[\frac{\Delta;\Gamma \vdash_{\text{exp}} e_f : \forall\vartheta.\, \tau \qquad \Delta \vdash_{\text{index}} \theta_a}{\Delta;\Gamma \vdash_{\text{exp}} e_f\ [\theta_a] : \tau[\theta_a/\vartheta]}\right]\!\!\right] \;=\; \mathbb{E}[\![e_f]\!]\ \ [\mathbb{T}[\![\theta_a]\!]]$$

Figure 4.38: Translation from $\mathsf{F}^{\mathsf{RGN}}$ to rgnURAL (terms (V))

RGN monad terms Figure 4.37 gives the translation of `runRGN`, which works in much the same manner as the translation of `letRGN`. The most significant difference is that `runRGN` is not translated into a stack-passing computation; rather, it fully evaluates the monadic computation, returning the final value of type $\mathbb{T}[\![\tau]\!]$. The other subtle difference is that the translation of `runRGN` constructs the representation of the stack for the monadic computation by pairing an "empty" stack (represented by a ${}^{\mathsf{U}}\mathbf{1}_{\otimes}$ value) with the new region capability.

Finally, Figure 4.38 gives the translation of $\mathsf{F}^{\mathsf{RGN}}$ index abstraction and instantiation. Since the representation of a stack of regions in rgnURAL is a value with L qualified pre-type, we translate $\mathsf{F}^{\mathsf{RGN}}$ index abstraction and instantiation to rgnURAL type abstraction and instantiation.

4.3.1 Translation Properties

The translation is type preserving, as formalized by the following lemma. The proof is by (mutual) induction on the structure of the typing judgments, making frequent appeals to various well-formedness lemmas.

Lemma 4.2 (Translation Preserves Types)

(1) If Δ *is well-formed, then* $\mathbb{D}[\![\Delta]\!]$ *is well-formed.*

(2) If $\Delta \vdash^{\mathsf{F}^{\mathsf{RGN}}}_{\text{index}} \theta$*, then* $\mathbb{D}[\![\Delta]\!] \vdash^{\mathsf{rgnURAL}}_{\text{type}} \mathbb{T}\left[\!\!\left[\Delta \vdash^{\mathsf{F}^{\mathsf{RGN}}}_{\text{index}} \theta\right]\!\!\right]$.

(3) If $\Delta \vdash^{\mathsf{F}^{\mathsf{RGN}}}_{\text{type}} \tau$*, then* $\mathbb{D}[\![\Delta]\!] \vdash^{\mathsf{rgnURAL}}_{\text{ptype}} \mathbb{P}\left[\!\!\left[\Delta \vdash^{\mathsf{F}^{\mathsf{RGN}}}_{\text{type}} \tau\right]\!\!\right]$.

(4) If $\Delta \vdash^{\mathsf{F}^{\mathsf{RGN}}}_{\text{type}} \tau$*, then* $\mathbb{D}[\![\Delta]\!] \vdash^{\mathsf{rgnURAL}}_{\text{type}} \mathbb{T}\left[\!\!\left[\Delta \vdash^{\mathsf{F}^{\mathsf{RGN}}}_{\text{type}} \tau\right]\!\!\right]$.

(5) If $\Delta \vdash^{\mathsf{F}^{\mathsf{RGN}}}_{\text{vctxt}} \Gamma$*, then* $\mathbb{D}[\![\Delta]\!] \vdash^{\mathsf{rgnURAL}}_{\text{vctxt}} \mathbb{G}\left[\!\!\left[\Delta \vdash^{\mathsf{F}^{\mathsf{RGN}}}_{\text{vctxt}} \Gamma\right]\!\!\right]$.

(6) If $\vdash^{\mathsf{F}^{\mathsf{RGN}}}_{\text{ctxt}} \Delta;\Gamma$*, then* $\vdash^{\mathsf{rgnURAL}}_{\text{ctxt}} \mathbb{D}[\![\Delta]\!];\mathbb{G}\left[\!\!\left[\Delta \vdash^{\mathsf{F}^{\mathsf{RGN}}}_{\text{vctxt}} \Gamma\right]\!\!\right]$.

(7) If $\Delta;\Gamma \vdash^{\mathsf{F}^{\mathsf{RGN}}}_{\text{exp}} e : \tau$*, then*

$$\mathbb{D}[\![\Delta]\!];\mathbb{G}\left[\!\!\left[\Delta \vdash^{\mathsf{F}^{\mathsf{RGN}}}_{\text{vctxt}} \Gamma\right]\!\!\right] \vdash^{\mathsf{rgnURAL}}_{\text{exp}} \mathbb{E}\left[\!\!\left[\Delta;\Gamma \vdash^{\mathsf{F}^{\mathsf{RGN}}}_{\text{exp}} e : \tau\right]\!\!\right] : \mathbb{T}\left[\!\!\left[\Delta \vdash^{\mathsf{F}^{\mathsf{RGN}}}_{\text{type}} \tau\right]\!\!\right].$$

Furthermore, we firmly believe that the translation is meaning preserving, with respect to the dynamic semantics of $\mathsf{F}^{\mathsf{RGN}}$ and $\mathsf{rgnURAL}$, as formalized by the following conjecture:

Conjecture 4.3 (Translation Correctness (Programs))

If $\cdot;\cdot \vdash^{\mathsf{F}^{\mathsf{RGN}}}_{\text{exp}} e : \mathsf{Bool}$ *and* $(\cdot;e) \Downarrow^{\mathsf{F}^{\mathsf{RGN}}} \mathfrak{b}$ *and* $\mathbb{E}\left[\!\!\left[\cdot;\cdot \vdash^{\mathsf{F}^{\mathsf{RGN}}}_{\text{exp}} e\right]\!\!\right] = e^{\dagger}$*,*
then $(\{\};e^{\dagger}) \longmapsto^{\mathsf{rgnURAL}*} \mathfrak{b}$.

Intuitively, the conjecture follows from the fact that the translations of the RGN monad commands directly implement the large-step operational semantics of

$\mathsf{F}^{\mathsf{RGN}}$, where the sequence of linear capabilities (formed out of nested linear tuples) represents the stack which is threaded through the evaluation of $\mathsf{F}^{\mathsf{RGN}}$ commands.

The major difficulty with a rigorous proof of this conjecture is the fact that the operational semantics for $\mathsf{F}^{\mathsf{RGN}}$ and rgnURAL have significant differences. The most obvious difference is that $\mathsf{F}^{\mathsf{RGN}}$ uses a large-step operational semantics, whereas rgnURAL uses a small-step operational semantics. Another major difference is that the operational semantics of $\mathsf{F}^{\mathsf{RGN}}$ replaces constant-region names ($\mathfrak{r}$) with a dead region ($\bullet$) in expression forms when a region is deallocated, whereas the operational semantics of rgnURAL leaves constant-region names ($\mathfrak{r}$) in expression forms, but marks the region as dead (`dead`) in the heap, when a region is deallocated.

In order to undertake a rigorous proof that the translation is meaning preserving, it would be advisable to first consider a second operational semantics for $\mathsf{F}^{\mathsf{RGN}}$. In particular, we would consider an operational semantics in which constant-region names are left in expression forms, and the region stack is instrumented with marks for live and dead regions. Since the first and second operational semantics for $\mathsf{F}^{\mathsf{RGN}}$ would be very similar, it should be straightforward to show that for any $\mathsf{F}^{\mathsf{RGN}}$ program, evaluating it under the first operational semantics has the same observable behavior as evaluating it under the second operational semantics. Similarly, since there would be a closer correspondence between the second operational semantics for $\mathsf{F}^{\mathsf{RGN}}$ and the operational semantics for rgnURAL, it should be somewhat easier to show that the translation is meaning preserving.[12]

[12]However, it should be noted that even in this case, there are significant technical details that need to be formalized. For example, as is done in Appendix B.3 for the translation from SEC to $\mathsf{F}^{\mathsf{RGN}}$, we would need to extend the translation in this section with cases for the additional semantic objects in the abstract machine configurations of $\mathsf{F}^{\mathsf{RGN}}$.

4.4 Related Work

Our rgnURAL is most directly influenced by the presentation of substructural type systems by Walker [89], which in turn draws upon the work of Wansbrough and Peyton-Jones [94] and Walker and Watkins [92]. Relative to that work, we have added both relevant and affine qualifiers, which is necessary to account for the varied forms of linearity found in higher-level programming-language proposals; see below.

A related body of work is that on type systems used to track resource usage [82, 60, 94, 53, 32, 47]. We note that the usage subsumption found in these systems (e.g., a "possibly used many times" variable may be subsumed to appear in a context requiring a "used exactly once" value) is not applicable in our setting (e.g., it is clearly unsound to subsume $^{\mathsf{U}}(\overline{\mathsf{Ref}}\ \rho\ \tau)$ to $^{\mathsf{L}}(\overline{\mathsf{Ref}}\ \rho\ \tau)$), due to differences in the interpretation of type qualifiers.

There has been much prior work aimed at relaxing the stack discipline imposed on region lifetimes by the Tofte-Talpin region calculus. The ML Kit [78] uses a storage-mode analysis to determine when it is safe to deallocate data in a region (known as region resetting) prior to the deallocation of the region itself. The safety of the storage-mode analysis has not been established formally.

Aiken *et al.* [1] eliminate the requirement that region allocation and deallocation should coincide with the lexical scope of region variables introduced by the `letregion` construct. Their approach uses *late allocation/early deallocation*, which delays the allocation of a region until just before its first access, and deallocates the region just after its last access. A constraint-based analysis determines where to insert region allocation and deallocation commands. We believe that the results of their analysis can be encoded explicitly in rgnURAL. However, we note that, as

formulated, rgnURAL does not support late allocation, since the `newrgn` primitive both generates a fresh region name and allocates the region. We believe that it would be straightforward to revise rgnURAL, giving `newrgn` the typing rule:

$$\frac{\Delta \vdash_{\text{qual}} q_c \qquad \Delta \vdash_{\text{qual}} q_h}{\Delta; \cdot \vdash_{\text{exp}} {}^{q_c,q_h}\texttt{newrgn} : {}^{\mathsf{L}}(\overline{\exists}\varrho.\, {}^{\mathsf{L}}({}^{\mathsf{L}}\mathbf{1}_{\otimes} \multimap {}^{\mathsf{L}}({}^{q_c}(\overline{\mathsf{Cap}}\ \varrho) \otimes {}^{q_h}(\overline{\mathsf{Hnd}}\ \varrho))))}$$

Note that with the rule, the result of `newrgn` generates a fresh region name ($\overline{\exists}\varrho$), along with a function (${}^{\mathsf{L}}({}^{\mathsf{L}}\mathbf{1}_{\otimes} \multimap {}^{\mathsf{L}}({}^{q_c}(\overline{\mathsf{Cap}}\ \varrho) \otimes {}^{q_h}(\overline{\mathsf{Hnd}}\ \varrho))))$, which may be evaluated later to allocate the region and yield the region capability and handle; Section 5.6 considers a similar idea.

Unlike the previous two approaches which build on the Tofte-Talpin region calculus, Henglein *et al.* [40] present a region system that (like ours) replaces the `letregion` primitive with explicit commands to allocate and deallocate a region, eliminating the need for the LIFO discipline. To ensure safety, they use a region type system with Hoare-logic and consequently have no support for higher-order functions. While they provide an inference algorithm to annotate programs with region manipulation commands, we intend for rgnURAL to serve as a target language for programs annotated using a region inference system or for programs written in languages like Cyclone. The Calculus of Capabilities [90] is also intended as a target for annotated programs, but unlike rgnURAL, it is defined in terms of a continuation-passing style language and does not support first-class regions.

The region system presented by Walker and Watkins [92] is perhaps the most closely related work. Like our target, they require a linear capability to be presented upon each access to a region. However, they provide a primitive, similar to `letregion`, that allows a capability to be temporarily treated as unrestricted for convenience's sake. We have shown that no such primitive is needed. Rather, we

use a combination of monadic encapsulation (to thread capabilities) coupled with *unrestricted witnesses* to achieve the same flexibility. In Section 5.5, we show how to translate Cyclone's dynamic regions into rgnURAL in a manner that achieves the same effect as the Walker-Watkin's primitive.

Another related body of work has used regions as a low-level primitive on which to build type-safe garbage collectors [93, 64, 38]. Each of these approaches requires non-lexical regions, since, in a copying collector, the from- and to-spaces have non-nested lifetimes. Hawblitzel *et al.* [38] introduce a very low-level language in which they begin with a single linear array of words, construct lists and arrays out of the basic linear memory primitives, introduce *type sequences* for building regions of nonlinear data. Such a foundational approach is admirable, but there is a large semantic gap between a high-level language and such a target. Hence, rgnURAL serves as a useful intermediate point, and we may envision further translation from rgnURAL to such a low-level language. A first step in this direction is taken in Chapter 5.

Our treatment of mutable references in rgnURAL is related to a number of projects (e.g., Clean, CQual, Cyclone, Vault), which have introduced some form of "uniqueness" to "tame" state. In each these systems, unique objects make it possible to perform operations that would otherwise be prohibited (e.g., deallocating an object) or to ensure that some obligation will be met (e.g., an opened file will be closed).

For instance, the Clean programming language [74] relies upon a form of uniqueness to ensure equational reasoning in the presence of mutable data structures. Cyclone's unique pointers are also used to allow fine-grained memory management. For example, a unique pointer may be updated from uninitialized to initialized, and

its contents may also be deallocated. In both of these languages, a unique object may be implicitly discarded, yielding affine objects (a weak form of uniqueness).

On the other hand, the Vault programming language [18] uses tracked keys to enforce resource management protocols. For example, an interface may specify that opening a file returns a new tracked key, which must be present when reading the file, and which is consumed when closing the file. Because tracked keys may be neither duplicated nor discarded, Vault supports linear objects (a strong form of uniqueness), which ensures that an opened file must be closed exactly once, much in the way that our linear capabilities ensures that an allocated region is deallocated exactly once.

Since programming in a language with only unique (i.e., linear or affine) objects is much too painful, it is not surprising that both Cyclone and Vault allow a programmer to put unique objects in shared objects, with a variety of restrictions to ensure that these mixed objects behave in a safe manner. In fact, understanding the various mechanisms by which unique objects (with strong updates) may safely coexist and mix with shared objects is currently an active area of research [2], though much of it has focused on high-level programming features, often without a complete formal account. Our treatment of mutable references with four sorts of qualifiers gives an integrated design that demonstrates exactly when unique objects may be stored in shared references.

There has also been a great deal of work on adapting some notion of linearity to real programming languages such as Java. Examples include ownership types [15, 8], uniqueness types [9, 13], confinement types [14, 31, 83], balloon types [3], islands [45], and roles [54]. Each of these mechanisms is aimed at supporting local reasoning in the presence of aliasing and updates.

4.5 Summary and Future Work

We have given a type- and meaning-preserving translation from $\mathsf{F}^{\mathsf{RGN}}$ to rgnURAL. The central element of the translation is to "break open" the RGN monad, exposing its interpretation as a region-stack transformer. Whereas $\mathsf{F}^{\mathsf{RGN}}$ uses monadic encapsulation and the parametric types for `runRGN` and `letRGN` to prevent access to regions beyond their lifetimes, rgnURAL separates the creation and destruction of a region and uses a substructural type system to manage capabilities that grant access to regions. By ensuring that the (one) capability for a region is consumed when the region is destroyed, we ensure that although the region may be named after it is destroyed, it cannot be accessed. The $\mathsf{F}^{\mathsf{RGN}}$ witness terms, used to safely "shift" computations from one region stack to another, are realized in the translation as functions that "shuffle" capabilities in the representation of a region stack as a sequence of linear capabilities.

There are numerous directions for future work. Perhaps the most important direction is to better account for the fact that capabilities have no run-time significance in a program. (Recall that the run-time representation of a $^{\mathsf{q_c}}(\overline{\mathsf{Cap}}\ \mathtt{r})$ is $^{\mathsf{q_c}}(\mathsf{cap})$, which does not name the region.) In principle, we could erase capabilities (and the translations of $\mathsf{F}^{\mathsf{RGN}}$ witness terms, which do nothing but "shuffle" capabilities) before running a program, without changing the behavior of the program.

To realize this goal, we should introduce a phase distinction, where capabilities are treated as static objects, and other terms are treated as dynamic objects. Just as an unrestricted data structure cannot contain linear components, static values could not depend upon dynamic values. In order to justify the erasure of static computations, we would need to ensure that such computations correspond to total, effect-free functions. This sort of phase-splitting has been used in other

settings that mix programming languages and logics, such as Xi *et al.*'s Applied Type System [12] and Sheard's Omega [73]. Another promising approach is suggested by Mandelbaum, Walker, and Harper's refinement language [59], where they developed a two-level language for reasoning about effectful programs.

Another important direction is to further explore the ways in which unique and shared data may be mixed. For example, Cyclone's `alias` construct [44] takes a unique pointer and returns a shared pointer to the same object, which is available for a limited lexical scope. Vault's `focus` and CQuals's `restrict` constructs [20, 2] provide the opposite behavior: temporarily giving a linear view of an object of shared type. Both behaviors are of great practical significance. More work is required to understand how best to model these advanced features in terms of a substructural language like rgnURAL.

Nonetheless, rgnURAL does capture the essential aspects of region-based memory management. In the next chapter, we consider the expressiveness of $\mathsf{F}^{\mathsf{RGN}}$ and rgnURAL and consider extensions that provide support for additional programming features (though, not the advanced features mentioned above). We also consider an advanced application of region-based memory management.

Chapter 5

Expressiveness and Applications

In this chapter, we consider the expressiveness of the various type-and-effect, monadic, and substructural languages presented in the previous chapters, consider extensions that provide support for additional programming features, and consider an advanced application of region-based memory management. This short investigation, along with the translations from the Single Effect Calculus to $\mathsf{F}^{\mathsf{RGN}}$ (Section 3.3) and from $\mathsf{F}^{\mathsf{RGN}}$ to rgnURAL (Section 4.3), helps to justify $\mathsf{F}^{\mathsf{RGN}}$ and rgnURAL as realistic formal languages that capture the essential aspects of region-based memory management.

An important issue to consider is the expressiveness of the Single Effect Calculus (and, subsequently, $\mathsf{F}^{\mathsf{RGN}}$ and rgnURAL) relative to the original Tofte-Talpin region calculus. Tofte and Talpin's formulation of the region calculus as the implicit target of an inference system makes a direct comparison difficult. Fortunately, there has been sufficient interest in region-based memory management to warrant direct presentations of region calculi [39, 10, 11, 41], which are better suited for comparison. Three aspects of the region calculus are highlighted as essential features: region polymorphism, region polymorphic recursion, and effect polymorphism.

Similarly, we would like to consider the expressiveness of $\mathsf{F}^{\mathsf{RGN}}$ and rgnURAL relative to Cyclone [29]. As noted previously, Cyclone includes a number of different kinds of regions. Cyclone also extends the type-and-effect system of the Tofte-Talpin region calculus with a form of region subtyping: pointers into older regions can be safely coerced into pointers into younger regions. Finally, in order

to address the limitations of lexically-scoped regions, later versions of Cyclone have added a number of features [77], including dynamic regions and unique pointers. Recall that while the soundness of Cyclone's initial design (with region subtyping and lexically-scoped regions) has been established [30], an argument that justifies the soundness of the new features has proved elusive, due to sheer complexity. Much of the complexity arises from the presence of related, but subtly different, features in the language.

The remainder of this chapter is structured as follows. In the next four sections, we show how the languages of the previous chapters either handle or may be extended to handle some of the essential aspects of region calculi: region polymorphism (Section 5.1), general recursion and region polymorphic recursion (Section 5.2), region reference subtyping (Section 5.3), and effect polymorphism (Section 5.4). In Section 5.5, we present a high-level overview of Cyclone, introduce a hybrid monadic and substructural language that captures the key features of Cyclone, and sketch a translation from this hybrid language to rgnURAL. Finally, in Section 5.6, we consider an advanced application of region-based memory management: expressing a type-safe copying garbage collector. We show that rgnURAL is nearly expressive enough to handle this application, and show how to extend rgnURAL (using many of the insights which drove the original development of rgnURAL) in order to handle this application.

5.1 Region Polymorphism

It should be clear that each of the languages considered (the Single Effect Calculus, $\mathsf{F}^{\mathsf{RGN}}$, and rgnURAL) directly support region polymorphism. We note that $\mathsf{F}^{\mathsf{RGN}}$ technically supports index polymorphism (through the index abstraction type

$\forall\vartheta.\,\tau$). Nonetheless, the translation from the Single Effect Calculus to $\mathsf{F}^{\mathsf{RGN}}$ shows that index polymorphism effectively provides region polymorphism.

5.2 General Recursion and Region Polymorphic Recursion

5.2.1 The Single Effect Calculus

General recursion can be supported in the Single Effect Calculus by adding `fix` and `fix`ing a function. Similarly, region polymorphic recursion can be supported by `fix`ing a region abstraction (as is shown by Henglein, Makholm, and Niss for the Tofte-Talpin region calculus [41]); Figure 5.1 presents the extensions to SEC necessary to support `fix`. Note that the dynamic semantics implements the recursion by region allocating an abstraction, and substituting the region reference for the recursive variable in the body of the abstraction.

As an example, consider the following term to compute a factorial (in which we elide the type annotation on *fact*):

$$
\begin{array}{l}
\texttt{fix}\ \mathit{fact}.(\Lambda\varrho_i \succeq \{\}.^{\rho_f}\ (\Lambda\varrho_o \succeq \{\}.^{\rho_f}\ (\Lambda\varrho_b \succeq \{\rho_f, \varrho_i, \varrho_o\}.^{\rho_f} \\
\qquad (\lambda n{:}(\mathsf{Int}, \varrho_i).^{\varrho_b} \\
\qquad\ \texttt{if}\ (\texttt{letregion}\ \varrho\ \texttt{in}\ n \leq (1\ \texttt{at}\ \varrho)) \\
\qquad\quad \texttt{then}\ 1\ \texttt{at}\ \varrho_o \\
\qquad\quad \texttt{else letregion}\ \varrho_{i'}\ \texttt{in}\ (\texttt{letregion}\ \varrho_{o'}\ \texttt{in} \\
\qquad\qquad\qquad (\mathit{fact}\ [\varrho_{i'}]\ [\varrho_{o'}]\ [\varrho_{o'}] \\
\qquad\qquad\qquad\qquad (\texttt{letregion}\ \varrho\ \texttt{in}\ (n - (1\ \texttt{at}\ \varrho)\ \texttt{at}\ \varrho_{i'}))) * n\ \texttt{at}\ \varrho_o \\
\qquad)\ \texttt{at}\ \rho_f)\ \texttt{at}\ \rho_f)\ \texttt{at}\ \rho_f)\ \texttt{at}\ \varrho_f)
\end{array}
$$

The function *fact* is parameterized by three regions: ϱ_i is the region in which the input integer is allocated, ϱ_o is the region in which the output integer is to be

Terms

$$e \;::=\; \ldots \mid \texttt{fix}\ f{:}\tau.u$$

Abstractions

$$u \;::=\; \lambda x{:}\tau.^{\pi'}\, e\ \texttt{at}\ \rho \mid \Lambda\varrho \succeq \phi.^{\pi'}\, u\ \texttt{at}\ \rho$$

$\boxed{(S; e) \Downarrow (S'; v')}$

$$\frac{\mathfrak{r} \in dom(S) \qquad \mathfrak{p} \notin dom(S(\mathfrak{r}))}{(S; \texttt{fix}\ f{:}\tau.\lambda x{:}\tau_x.^{\pi'}\, e'\ \texttt{at}\ \mathfrak{r}) \Downarrow (S\{(\mathfrak{r},\mathfrak{p}) \mapsto (\lambda x{:}\tau_x.^{\pi'}\, e')[\texttt{ref}\ \mathfrak{r}\ \mathfrak{p}/f]\}; \texttt{ref}\ \mathfrak{r}\ \mathfrak{p})}$$

$$\frac{\mathfrak{r} \in dom(S) \qquad \mathfrak{p} \notin dom(S(\mathfrak{r}))}{(S; \texttt{fix}\ f{:}\tau.\Lambda\varrho \succeq \phi.^{\pi'}\, u'\ \texttt{at}\ \mathfrak{r}) \Downarrow (S\{(\mathfrak{r},\mathfrak{p}) \mapsto (\Lambda\varrho \succeq \phi.^{\pi'}\, u')[\texttt{ref}\ \mathfrak{r}\ \mathfrak{p}/f]\}; \texttt{ref}\ \mathfrak{r}\ \mathfrak{p})}$$

$\boxed{\Delta; \Gamma \vdash_{\text{exp}} e : \tau, \pi}$

$$\frac{\Delta; \Gamma, f{:}\tau \vdash_{\text{exp}} u : \tau, \pi}{\Delta; \Gamma \vdash_{\text{exp}} \texttt{fix}\ f{:}\tau.u : \tau, \pi}$$

Figure 5.1: Extensions to SEC for `fix`

allocated, and ϱ_b is a region that bounds the latent effect of the function. (Region ρ_f is assumed to be bound in an outer context and holds the closures.) We see that the bounds on ϱ_i and ϱ_o indicate that they are not constrained to be outlived by any other regions. On the other hand, the bound on ϱ_b indicates that ρ_f, ϱ_i, and ϱ_o must outlive ϱ_b. Hence, ϱ_b suffices to bound the effects within the body of the function, in which we expect regions ρ_f (at the recursive call) and ϱ_i to be read from and region ϱ_o to be allocated in. Note that the regions passed to the recursive call of *fact* satisfy the bounds, as $\varrho_{o'}$ outlives ρ_f (through $\varrho_{i'}$ and ϱ_b), $\varrho_{i'}$ is allocated before (and deallocated after) $\varrho_{o'}$, and $\varrho_{o'}$ clearly outlives itself.

5.2.2 The $\mathsf{F}^{\mathsf{RGN}}$ Language

General recursion can be supported in $\mathsf{F}^{\mathsf{RGN}}$ by adding `fix` in the standard manner [67, Section 11.11]. This extension admits `fix`ing functions, type abstractions, and index abstractions.

Translation: From SEC to $\mathsf{F}^{\mathsf{RGN}}$

The extension of SEC with `fix` can be translated into the extension of $\mathsf{F}^{\mathsf{RGN}}$ with `fix`. The translation of Section 3.3 is extended with the translations in Figure 5.3. Recall that all abstractions (including recursive abstractions) in SEC are region allocated; hence, they must be translated to region references in $\mathsf{F}^{\mathsf{RGN}}$, in order that the recursive invocations are correctly handled by the translations of application and instantiations. We handle this using the standard trick of "back-patching" a mutable reference. In both translations, we allocate a region reference, bound to a divergent abstraction of the appropriate type (constructed using the $\mathsf{F}^{\mathsf{RGN}}$ `fix`). We then immediately update the region reference with the proper translation, which

Terms

$e \;::=\; \ldots \mid \texttt{fix}\ f{:}\tau.u$

Abstractions

$u \;::=\; \lambda x{:}\tau.\, e \mid \Lambda\alpha.\, \tau \mid \Lambda\vartheta.\, e$

$\boxed{(T; e) \Downarrow v}$

$$\frac{}{(T; \texttt{fix}\ f{:}\tau.u) \Downarrow u[\texttt{fix}\ f{:}\tau.u/f]}$$

$\boxed{\Delta; \Gamma \vdash_{\text{exp}} e : \tau}$

$$\frac{\Delta; \Gamma, f{:}\tau \vdash_{\text{exp}} u : \tau}{\Delta; \Gamma \vdash_{\text{exp}} \texttt{fix}\ f{:}\tau.u : \tau}$$

Figure 5.2: Extensions to $\mathsf{F}^{\mathsf{RGN}}$ for `fix`

$$\mathbb{E}\left[\!\!\left[\frac{\dfrac{\Delta;\Gamma,f{:}(\tau_x \xrightarrow{\pi'} \tau,\rho),x{:}\tau_x \vdash_{\text{exp}} e : \tau,\pi' \qquad \Delta \vdash_{\text{region}} \rho \qquad \Delta \vdash_{\text{rr}} \pi \succeq \rho}{\Delta;\Gamma,f{:}(\tau_x \xrightarrow{\pi'} \tau,\rho) \vdash_{\text{exp}} \lambda x{:}\tau_x.^{\pi'}\, e\ \texttt{at}\ \rho : (\tau_x \xrightarrow{\pi'} \tau,\rho),\pi}}{\Delta;\Gamma \vdash_{\text{exp}} \texttt{fix}\ f{:}(\tau_x \xrightarrow{\pi'} \tau,\rho).\lambda x{:}\tau_x.^{\pi'}\, e\ \texttt{at}\ \rho : (\tau_x \xrightarrow{\pi'} \tau,\rho),\pi}\right]\!\!\right] =$$

$$\begin{array}{l}
\texttt{bindRGN}\ f{:}\mathbb{T}[\![(\tau_x \xrightarrow{\pi'} \tau,\rho)]\!] \Leftarrow \mathbb{E}[\![\pi \succeq \rho]\!]\ [\mathbb{T}[\![(\tau_x \xrightarrow{\pi'} \tau,\rho)]\!]] \\
\quad (\texttt{newRGNRef}\ [\mathbb{I}[\![\rho]\!]]\ [\mathbb{T}[\![(\tau_x \xrightarrow{\pi'} \tau,\rho)]\!]] \\
\qquad \mathbb{E}[\![\rho]\!]\ (\texttt{fix}\ g{:}\mathbb{T}[\![(\tau_x \xrightarrow{\pi'} \tau,\rho)]\!]\ .\lambda y{:}\mathbb{T}[\![\tau_x]\!]\,.\,g\ y))\ ; \\
\texttt{bindRGN}\ u{:}1 \Leftarrow \mathbb{E}[\![\pi \succeq \rho]\!]\ [1] \\
\quad (\texttt{writeRGNRef}\ [\mathbb{I}[\![\rho]\!]]\ [\mathbb{T}[\![(\tau_x \xrightarrow{\pi'} \tau,\rho)]\!]] \\
\qquad f\ (\lambda x{:}\mathbb{T}[\![\tau_x]\!]\,.\,\mathbb{E}[\![e]\!]))\ ; \\
\texttt{returnRGN}\ [\mathbb{I}[\![\pi]\!]]\ [\mathbb{T}[\![(\tau_x \xrightarrow{\pi'} \tau,\rho)]\!]]\ f
\end{array}$$

$$\mathbb{E}\left[\!\!\left[\frac{\dfrac{\Delta,\varrho \succeq \phi;\Gamma,f{:}(\forall \varrho \succeq \phi.^{\pi'}\, \tau,\rho) \vdash_{\text{exp}} u : \tau,\pi' \qquad \Delta \vdash_{\text{region}} \rho \qquad \Delta \vdash_{\text{rr}} \pi \succeq \rho}{\Delta;\Gamma,f{:}(\forall \varrho \succeq \phi.^{\pi'}\, \tau,\rho) \vdash_{\text{exp}} \Lambda \varrho \succeq \phi.^{\pi'}\, u\ \texttt{at}\ \rho : (\forall \varrho \succeq \phi.^{\pi'}\, \tau,\rho),\pi}}{\Delta;\Gamma \vdash_{\text{exp}} \texttt{fix}\ f{:}(\forall \varrho \succeq \phi.^{\pi'}\, \tau,\rho).\Lambda \varrho \succeq \phi.^{\pi'}\, u\ \texttt{at}\ \rho : (\forall \varrho \succeq \phi.^{\pi'}\, \tau,\rho),\pi}\right]\!\!\right] =$$

$$\begin{array}{l}
\texttt{bindRGN}\ f{:}\mathbb{T}[\![(\forall \varrho \succeq \phi.^{\pi'}\, \tau,\rho)]\!] \Leftarrow \mathbb{E}[\![\pi \succeq \rho]\!]\ [\mathbb{T}[\![(\forall \varrho \succeq \phi.^{\pi'}\, \tau,\rho)]\!]] \\
\quad (\texttt{newRGNRef}\ [\mathbb{I}[\![\rho]\!]]\ [\mathbb{T}[\![(\forall \varrho \succeq \phi.^{\pi'}\, \tau,\rho)]\!]] \\
\qquad \mathbb{E}[\![\rho]\!]\ (\texttt{fix}\ g{:}\mathbb{T}[\![(\forall \varrho \succeq \phi.^{\pi'}\, \tau,\rho)]\!]\ .\Lambda \vartheta_\varrho.\, g\ [\vartheta_\varrho]))\ ; \\
\texttt{bindRGN}\ u{:}1 \Leftarrow \mathbb{E}[\![\pi \succeq \rho]\!]\ [1] \\
\quad (\texttt{writeRGNRef}\ [\mathbb{I}[\![\rho]\!]]\ [\mathbb{T}[\![(\forall \varrho \succeq \phi.^{\pi'}\, \tau,\rho)]\!]] \\
\qquad f\ (\Lambda \vartheta_\varrho.\, \lambda w_\varrho{:}\mathbb{T}[\![\varrho \succeq \phi]\!]\,.\,\lambda h_\varrho{:}\mathsf{RGNHnd}\ \vartheta_\varrho.\, \mathbb{E}[\![u]\!]))\ ; \\
\texttt{returnRGN}\ [\mathbb{I}[\![\pi]\!]]\ [\mathbb{T}[\![(\forall \varrho \succeq \phi.^{\pi'}\, \tau,\rho)]\!]]\ f
\end{array}$$

Figure 5.3: Translation from SEC to $\mathsf{F}^{\mathsf{RGN}}$ (`fix`)

Terms

$e \;::=\; \ldots \mid \texttt{fix}\ f{:}\tau.u$

Abstractions

$u \;::=\; {}^{q}\lambda x{:}\tau.\, e \mid {}^{q}\Lambda\xi.\, e \mid {}^{q}\Lambda\overline{\alpha}.\, e \mid {}^{q}\Lambda\alpha.\, e \mid {}^{q}\Lambda\varrho.\, e$

$\boxed{(H; e) \longmapsto (H'; e')}$

$$\frac{}{(H; \texttt{fix}\ f{:}\tau.u) \longmapsto (H; u[\texttt{fix}\ f{:}\tau.u/f])}$$

$\boxed{\Delta; \Gamma \vdash_{\text{exp}} e : \tau}$

$$\frac{\Delta \vdash \tau \preceq \mathsf{U} \qquad \Delta; \Gamma, f{:}\tau \vdash_{\text{exp}} u : \tau}{\Delta; \Gamma \vdash_{\text{exp}} \texttt{fix}\ f{:}\tau.u : \tau}$$

Figure 5.4: Extensions to rgnURAL for `fix`

may mention the variable f to implement the recursion.

5.2.3 The rgnURAL Language

General recursion can be supported in rgnURAL in much the same manner as it is supported in $\mathsf{F}^{\mathsf{RGN}}$. This extension admits `fix`ing all forms of abstractions: functions, qualifier abstractions, pre-type abstractions, type abstractions, and region abstractions. Note that the typing rule for `fix` requires the type of the recursive abstraction to be unrestricted. While a less restrictive antecedent could be formulated, all practical uses of recursion suggest that the type should be unrestricted.

$$\mathbb{E}\left[\!\!\left[\frac{\Delta;\Gamma,f{:}\tau \vdash_{\text{exp}} u : \tau}{\Delta;\Gamma \vdash_{\text{exp}} \texttt{fix}\ f{:}\tau.u : \tau}\right]\!\!\right] \;=\; \texttt{fix}\ f{:}\mathbb{T}[\![\tau]\!].\mathbb{E}[\![e]\!]$$

Figure 5.5: Translation from $\mathsf{F}^{\mathsf{RGN}}$ to rgnURAL (`fix`)

Translation: From $\mathsf{F}^{\mathsf{RGN}}$ to rgnURAL

The extension of SEC with `fix` may be trivially translated into the extension of $\mathsf{F}^{\mathsf{RGN}}$ with `fix`. The translation of Section 4.3 is extended with the translations in Figure 5.5. Recall that all types in $\mathsf{F}^{\mathsf{RGN}}$ are translated to unrestricted types in rgnURAL. Therefore, the translation of a $\mathsf{F}^{\mathsf{RGN}}$ `fix` expression is a well-typed rgnURAL expression.

5.3 Region Reference Subtyping

We have noted previously that Cyclone extends the type-and-effect system of the Tofte-Talpin region calculus with a form of region subtyping: a reference in an older region can be safely treated as a reference in a younger region. Intuitively, this subtyping is safe, because the liveness of the younger region implies the liveness of the older region. Hence, whenever the type-and-effect system asserts that it is safe to access a reference in a younger region (by establishing that the younger region is live), it is also safe to access a reference in an older region (since the older region is necessarily live). In this section, we show how to support this form of region reference subtyping.

$\boxed{\Delta; \Gamma \vdash_{\text{exp}} e : \tau, \pi}$

$$\frac{\Delta; \Gamma \vdash_{\text{exp}} e : (\mu, \rho_1), \pi \qquad \Delta \vdash_{\text{rr}} \rho_2 \succeq \rho_1}{\Delta; \Gamma \vdash_{\text{exp}} e : (\mu, \rho_2), \pi}$$

Figure 5.6: Static semantics of SEC (region reference subtyping)

5.3.1 The Single Effect Calculus

This form of region reference subtyping may easily be handled in the Single Effect Calculus, by simply including a new type-and-effect rule for expressions that allows a boxed type in one region to be typed as a boxed type in a second region, when the first region is guaranteed to outlive the second (see Figure 5.6). Recall that the soundness of this rule is ensured by the LIFO stack of regions, which imposes a partial order on live regions. Operationally, the the coercion has no run-time effect; in particular, we do not copy the contents of the reference from one region to another.

5.3.2 The $\mathsf{F}^{\mathsf{RGN}}$ Language

We next turn our attention to $\mathsf{F}^{\mathsf{RGN}}$ and the extensions needed to support this form of region reference subtyping. Since we intend to extend the translation of Section 3.3 to handle this extension, we first consider what the translation must accomplish with respect to the new SEC type-and-effect rule.

Note that the translation of the new SEC type-and-effect rule must coerce a value of (translated) type $\mathsf{RGNRef}\ \mathbb{I}[\![\rho_1]\!]\ \mathbb{T}[\![\tau]\!]$ to a value of (translated) type $\mathsf{RGNRef}\ \mathbb{I}[\![\rho_2]\!]\ \mathbb{T}[\![\tau]\!]$. Abstracting, we can imagine a more general coercion function

with the type:

$$\forall \vartheta_1, \vartheta_2. \forall \beta. \, \mathsf{RGNRef}\ \vartheta_1\ \beta \to \mathsf{RGNRef}\ \vartheta_2\ \beta$$

Alternatively, we could introduce an abbreviation $\mathsf{RGNRefPf}(\theta_1 \preceq \theta_2)$ for the type of a function that coerces any reference in the region indexed by θ_1 into a reference in the region indexed by θ_2:

$$\mathsf{RGNRefPf}(\theta_1 \preceq \theta_2) \equiv \forall \beta. \, \mathsf{RGNRef}\ \theta_1\ \beta \to \mathsf{RGNRef}\ \theta_2\ \beta$$

Note that there is a strong similarity between the proposed $\mathsf{RGNRefPf}(\theta_1 \preceq \theta_2)$ and the $\mathsf{RGNPf}(\theta_1 \preceq \theta_2)$ already present in $\mathsf{F}^{\mathsf{RGN}}$. In fact, they both express a relationship between an older region index θ_1 and a younger region index θ_2; the only difference is that RGNPf specializes this relationship to region computations, while RGNRefPf specializes this relationship to region references.

Hence, rather than introduce RGNRefPf, we find it more convenient to make RGNPf an abstract type, and introduce new terms for specializing RGNPf to computations and to region references. We may consider $\mathsf{RGNPf}(\theta_1 \preceq \theta_2)$ to be a witness to the fact that the region stack indexed by θ_1 is a subtype of the region stack indexed by θ_2, corresponding to the fact that every region in the stack θ_1 is also in the stack θ_2. Figure 5.7 presents the typing rules for the new terms in $\mathsf{F}^{\mathsf{RGN}}$. The terms `reflRGNPf` and `transRGNPf` are combinators witnessing the reflexivity and transitivity of the outlives relation on regions. The terms `coerceRGN` and `coerceRGNRef` apply a proof of region subtyping to computations and region references, respectively. Operationally, both of these terms act as the identity function.

$\boxed{\Delta; \Gamma \vdash_{\text{exp}} e : \tau}$

$$\frac{\vdash_{\text{ctxt}} \Delta; \Gamma \qquad \Delta \vdash_{\text{index}} \theta}{\Delta; \Gamma \vdash_{\text{exp}} \texttt{reflRGNPf}\ [\theta] : \mathsf{RGNPf}(\theta \preceq \theta)}$$

$$\frac{\begin{array}{c}\Delta \vdash_{\text{index}} \theta_1 \qquad \Delta \vdash_{\text{index}} \theta_2 \qquad \Delta \vdash_{\text{index}} \theta_3 \\ \Delta; \Gamma \vdash_{\text{exp}} e_1 : \mathsf{RGNPf}(\theta_1 \preceq \theta_2) \\ \Delta; \Gamma \vdash_{\text{exp}} e_2 : \mathsf{RGNPf}(\theta_2 \preceq \theta_3)\end{array}}{\Delta; \Gamma \vdash_{\text{exp}} \texttt{transRGNPf}\ [\theta_1]\ [\theta_2]\ [\theta_3]\ e_1\ e_2 : \mathsf{RGNPf}(\theta_1 \preceq \theta_3)}$$

$$\frac{\begin{array}{c}\Delta \vdash_{\text{index}} \theta_1 \qquad \Delta \vdash_{\text{index}} \theta_2 \qquad \Delta \vdash_{\text{type}} \tau \\ \Delta; \Gamma \vdash_{\text{exp}} e_1 : \mathsf{RGNPf}(\theta_1 \preceq \theta_2) \qquad \Delta; \Gamma \vdash_{\text{exp}} e_2 : \mathsf{RGN}\ \theta_1\ \tau\end{array}}{\Delta; \Gamma \vdash_{\text{exp}} \texttt{coerceRGN}\ [\theta_1]\ [\theta_2]\ [\tau]\ e_1\ e_2 : \mathsf{RGN}\ \theta_2\ \tau}$$

$$\frac{\begin{array}{c}\Delta \vdash_{\text{index}} \theta_1 \qquad \Delta \vdash_{\text{index}} \theta_2 \qquad \Delta \vdash_{\text{type}} \tau \\ \Delta; \Gamma \vdash_{\text{exp}} e_1 : \mathsf{RGNPf}(\theta_1 \preceq \theta_2) \qquad \Delta; \Gamma \vdash_{\text{exp}} e_2 : \mathsf{RGNRef}\ \theta_1\ \tau\end{array}}{\Delta; \Gamma \vdash_{\text{exp}} \texttt{coerceRGNRef}\ [\theta_1]\ [\theta_2]\ [\tau]\ e_1\ e_2 : \mathsf{RGNRef}\ \theta_2\ \tau}$$

Figure 5.7: Static semantics of $\mathsf{F}^{\mathsf{RGN}}$ (region reference subtyping)

Witnesses

$$\mathbb{E}\left[\!\!\left[\frac{\vdash_{\text{rctxt}} \Delta \qquad (\varrho \succeq \{\rho_1, \ldots, \rho_i, \ldots, \rho_n\}) \in \Delta}{\Delta \vdash_{\text{rr}} \varrho \succeq \rho_i}\right]\!\!\right] \;=\; \texttt{sel}_i\ w_\varrho$$

$$\mathbb{E}\left[\!\!\left[\frac{\Delta \vdash_{\text{region}} \rho}{\Delta \vdash_{\text{rr}} \rho \succeq \rho}\right]\!\!\right] \;=\; \texttt{reflRGNPf}\ [\mathbb{I}[\![\rho]\!]]$$

$$\mathbb{E}\left[\!\!\left[\frac{\Delta \vdash_{\text{rr}} \rho_2 \succeq \rho' \qquad \Delta \vdash_{\text{rr}} \rho' \succeq \rho_1}{\Delta \vdash_{\text{rr}} \rho_2 \succeq \rho_1}\right]\!\!\right] \;=$$

$$\texttt{transRGNPf}\ [\mathbb{I}[\![\rho_1]\!]]\ [\mathbb{I}[\![\rho']\!]]\ [\mathbb{I}[\![\rho_2]\!]]\ \mathbb{E}[\![\rho' \succeq \rho_1]\!]\ \ \mathbb{E}[\![\rho_2 \succeq \rho']\!]$$

Expressions

$$\mathbb{E}\left[\!\!\left[\frac{\vdash_{\text{ctxt}} \Delta; \Gamma; \pi \qquad \Delta \vdash_{\text{region}} \rho \qquad \Delta \vdash_{\text{rr}} \pi \succeq \rho}{\Delta; \Gamma \vdash_{\text{exp}} \mathsf{i}\,\mathsf{at}\,\rho : (\mathsf{Int}, \rho), \pi}\right]\!\!\right] \;=$$

$$\texttt{coerceRGN}\ [\mathbb{I}[\![\rho]\!]]\ [\mathbb{I}[\![\pi]\!]]\ [\mathbb{T}[\![(\mathsf{Int}, \rho)]\!]]$$

$$\mathbb{E}[\![\pi \succeq \rho]\!]\ \ (\texttt{newRGNRef}\ [\mathbb{I}[\![\rho]\!]]\ [\mathbb{T}[\![\mathsf{Int}]\!]]\ \mathbb{E}[\![\rho]\!]\ \ \mathsf{i})$$

$\vdots$

$$\mathbb{E}\left[\!\!\left[\frac{\Delta; \Gamma \vdash_{\text{exp}} e : (\mu, \rho_1), \pi \qquad \Delta \vdash_{\text{rr}} \rho_2 \succeq \rho_1}{\Delta; \Gamma \vdash_{\text{exp}} e : (\mu, \rho_2), \pi}\right]\!\!\right] \;=$$

$$\texttt{bindRGN}\ r{:}\mathbb{T}[\![(\mu, \rho_1)]\!] \Leftarrow \mathbb{E}[\![e]\!]\ \ ;$$

$$\texttt{returnRGN}\ [\mathbb{I}[\![\rho_2]\!]]\ [\mathbb{T}[\![(\mu, \rho_2)]\!]]\ (\texttt{coerceRGNRef}\ [\mathbb{I}[\![\rho_1]\!]]\ [\mathbb{I}[\![\rho_2]\!]]\ [\mathbb{T}[\![\mu]\!]]$$

$$\mathbb{E}[\![\rho_2 \succeq \rho_1]\!]\ r)$$

Figure 5.8: Translation from SEC to $\mathsf{F}^{\mathsf{RGN}}$ (region reference subtyping)

Translation: From SEC to $\mathsf{F}^{\mathsf{RGN}}$

Figure 5.8 shows the revised and extended translation from SEC to $\mathsf{F}^{\mathsf{RGN}}$, now supporting region reference subtyping. We revise the translation from SEC outlives relations to $\mathsf{F}^{\mathsf{RGN}}$ witness terms; the reflexive and transitive rules are now translated to the `reflRGNPf` and `transRGNPf` terms. Because $\mathsf{RGNPf}(\theta_1 \preceq \theta_2)$ is now abstract, the translation of expressions must be revised to use `coerceRGN`; we give the translation of an integer constant as representative of this change.

Finally, the translation of the new SEC type-and-effect rule uses `coerceRGNRef`. Note that we must use `bindRGN` and `returnRGN` to sequence the pure `coerceRGNRef` in the RGN monad.

5.3.3 The rgnURAL Language

Encouragingly, we need to make no changes to rgnURAL to support region reference subtyping. Note that the insight (that the LIFO stack of regions, which imposes a partial order on live regions) used to ensure the soundness of region reference subtyping in SEC and $\mathsf{F}^{\mathsf{RGN}}$ does not apply to rgnURAL, since regions may be created and destroyed in an arbitrary order. Nonetheless, we may translate the region reference subtyping of $\mathsf{F}^{\mathsf{RGN}}$ into rgnURAL by revising and extending the translation of Section 4.3. The insight here is that the translation of the $\mathsf{RGNRef}\ \theta\ \tau$ uses an existentially bound region name (ϱ_r), to fix the region for the rgnURAL reference, and isomorphism to witness the fact that (the capability for) ϱ_r may be found within the stack represented by $\mathbb{T}[\![\theta]\!]$. The isomorphism expresses the fact that $\mathbb{T}[\![\theta]\!]$ may be coerced to and from ${}^{\mathsf{L}}(\beta \otimes {}^{\mathsf{L}}(\overline{\mathsf{Cap}}\ \varrho_r))$, for some "slack" β. Region reference subtyping in $\mathsf{F}^{\mathsf{RGN}}$ is translated to rgnURAL by constructing a new isomorphism with more "slack", corresponding to the additional live regions.

Translation: From $\mathsf{F}^{\mathsf{RGN}}$ to rgnURAL

Figures 5.9–5.12 revise and extend the translation of Section 4.3. Since the $\mathsf{RGNPf}(\theta_1 \preceq \theta_2)$ type is no longer an abbreviation, it requires a translation (Figure 5.9). Recall that $\mathsf{RGNPf}(\theta_1 \preceq \theta_2)$ is the type of witnesses to the fact that the region stack indexed by θ_1 is a subtype of the region stack indexed by θ_2. Hence, we translate it to a type that expresses the isomorphism between $\mathbb{T}[\![\theta_2]\!]$ and $\mathsf{L}(\mathbb{T}[\![\theta_1]\!] \otimes \beta)$, for some "slack" β. Recall that while the types $\mathbb{T}[\![\theta_2]\!]$, $\mathbb{T}[\![\theta_1]\!]$, and β may be linear, the pair of functions witnessing the isomorphism is unrestricted. This corresponds to the fact that the *proof* that θ_1 is a subtype of θ_2 is *persistent*, while the *existence* of the stacks θ_1 and θ_2 are *ephemeral*.

Figure 5.9 also revises the translation of `letRGN`. It is almost the same as the translation given in Section 4.3. The difference is that the general isomorphism *ppf* is passed to the inner computation, whereas before it was used to construct a witness (*pwit*, of a type corresponding to $\mathbb{T}[\![\forall\beta.\,\mathsf{RGN}\ \theta_1\ \beta \to \mathsf{RGN}\ \vartheta_2\ \beta]\!]$) which was passed to the inner computation.

Instead, this witness is constructed in the translation of `coerceRGN` (Figure 5.10). Note how the "slack" stack (stk_β) is split out and then combined in, bracketing the execution of the $\mathsf{RGN}\ \theta_1\ \tau$ computation.

The translation of `coerceRGNRef` (Figure 5.11) shows how the isomorphism for $\mathsf{RGNPf}(\theta_1 \preceq \theta_2)$, witnessing the fact that $\mathbb{T}[\![\theta_1]\!]$ is embedded in $\mathbb{T}[\![\theta_2]\!]$ with "slack" α, is combined with the isomorphism for $\mathsf{RGNRef}\ \theta_1\ \tau$, witnessing the fact that $\mathsf{L}(\overline{\mathsf{Cap}}\ \varrho)$ is embedded in $\mathbb{T}[\![\theta_1]\!]$ with "slack" β, to construct the isomorphism for $\mathsf{RGNRef}\ \theta_2\ \tau$, witnessing the fact that $\mathsf{L}(\overline{\mathsf{Cap}}\ \varrho)$ is embedded in $\mathbb{T}[\![\theta_2]\!]$ with "slack" $\mathsf{L}(\alpha \otimes \beta)$. Note that the translation neither reads from nor writes to the underlying reference *ref*; in fact, the reference need not even be allocated. Again,

Types

$$\mathbb{P}\left[\!\!\left[\frac{\Delta \vdash_{\text{index}} \theta_1 \qquad \Delta \vdash_{\text{index}} \theta_2}{\Delta \vdash_{\text{type}} \mathsf{RGNPf}(\theta_1 \preceq \theta_2)} \right]\!\!\right] \;=\; \overline{\exists}\beta.\, \mathbf{Iso}(\mathbb{T}[\![\theta_2]\!], {}^{\mathsf{L}}(\mathbb{T}[\![\theta_1]\!] \otimes \beta))$$

Expressions

$$\mathbb{E}\left[\!\!\left[\frac{\Delta \vdash_{\text{index}} \theta_1 \qquad \Delta \vdash_{\text{type}} \tau \qquad \Delta;\Gamma \vdash_{\text{exp}} v : \forall\vartheta.\, \mathsf{RGNPf}(\theta_1 \preceq \vartheta_2) \to \mathsf{RGNHnd}\ \vartheta_2 \to \mathsf{RGN}\ \vartheta_2\ \tau}{\Delta;\Gamma \vdash_{\text{exp}} \texttt{letRGN}\ [\theta_1]\ [\tau]\ v : \mathsf{RGN}\ \theta_1\ \tau} \right]\!\!\right] \;=$$

$$\begin{aligned}
&\texttt{let}\ f_v{:}\mathbb{T}[\![\forall\vartheta.\, \mathsf{RGNPf}(\theta_1 \preceq \vartheta_2) \to \mathsf{RGNHnd}\ \vartheta_2 \to \mathsf{RGN}\ \vartheta_2\ \tau]\!] = \mathbb{E}[\![v]\!]\ \texttt{in}\\
&{}^{\mathsf{U}}(\lambda stk_1{:}\mathbb{T}[\![\theta_1]\!].\, \texttt{let}\ \texttt{pack}(\varrho, \langle cap, hnd\rangle) = {}^{\mathsf{L},\mathsf{U}}\texttt{newrgn}\ \texttt{in}\\
&\qquad\qquad \texttt{let}\ id = {}^{\mathsf{U}}(\lambda stk_2{:}{}^{\mathsf{L}}(\mathbb{T}[\![\theta_1]\!] \otimes {}^{\mathsf{L}}(\overline{\mathsf{Cap}}\ \varrho)).\, stk_2)\ \texttt{in}\\
&\qquad\qquad \texttt{let}\ phnd = {}^{\mathsf{U}}\texttt{pack}(\varrho, {}^{\mathsf{U}}\langle {}^{\mathsf{U}}\texttt{pack}(\mathbb{T}[\![\theta_1]\!], \langle id, id\rangle), hnd\rangle)\ \texttt{in}\\
&\qquad\qquad \texttt{let}\ ppf = {}^{\mathsf{U}}\texttt{pack}({}^{\mathsf{L}}(\overline{\mathsf{Cap}}\ \varrho), \langle id, id\rangle)\ \texttt{in}\\
&\qquad\qquad \texttt{let}\ stk_2{:}{}^{\mathsf{L}}(\mathbb{T}[\![\theta_1]\!] \otimes {}^{\mathsf{L}}(\overline{\mathsf{Cap}}\ \varrho)) = {}^{\mathsf{L}}\langle stk_1, cap\rangle\ \texttt{in}\\
&\qquad\qquad \texttt{let}\ \langle stk_2, res\rangle = f_v\ [{}^{\mathsf{L}}(\mathbb{T}[\![\theta_1]\!] \otimes {}^{\mathsf{L}}(\overline{\mathsf{Cap}}\ \varrho))]\ ppf\ phnd\ stk_2\ \texttt{in}\\
&\qquad\qquad \texttt{let}\ \langle stk_1, cap\rangle = stk_2\ \texttt{in}\\
&\qquad\qquad \texttt{let}\ \langle\rangle = \texttt{freergn}\ {}^{\mathsf{L}}\langle cap, hnd\rangle\ \texttt{in}\\
&\qquad\qquad {}^{\mathsf{L}}\langle stk_1, res\rangle)
\end{aligned}$$

Figure 5.9: Translation from $\mathsf{F}^{\mathsf{RGN}}$ to rgnURAL (region reference subtyping (I))

Expressions

$$\mathbb{E}\left[\!\!\left[\frac{\Delta \vdash_{\text{index}} \theta_1 \qquad \Delta \vdash_{\text{index}} \theta_2 \qquad \Delta \vdash_{\text{type}} \tau \qquad \Delta;\Gamma \vdash_{\text{exp}} e_1 : \mathsf{RGNPf}(\theta_1 \preceq \theta_2) \qquad \Delta;\Gamma \vdash_{\text{exp}} e_2 : \mathsf{RGN}\ \theta_1\ \tau}{\Delta;\Gamma \vdash_{\text{exp}} \texttt{coerceRGN}\ [\theta_1]\ [\theta_2]\ [\tau]\ e_1\ e_2 : \mathsf{RGN}\ \theta_2\ \tau}\right]\!\!\right] =$$

$$\begin{aligned}
&\texttt{let}\ ppf{:}\mathbb{T}[\![\mathsf{RGNPf}(\theta_1 \preceq \theta_2)]\!] = \mathbb{E}[\![e_1]\!]\ \texttt{in}\\
&\texttt{let}\ f{:}\mathbb{T}[\![\mathsf{RGN}\ \theta_1\ \tau]\!] = \mathbb{E}[\![e_2]\!]\ \texttt{in}\\
&\texttt{let}\ \texttt{pack}(\alpha, \langle spl, cmb\rangle) = ppf\ \texttt{in}\\
&\lambda stk_2{:}\mathbb{T}[\![\theta_2]\!]\,.\,\texttt{let}\ \langle stk_1, stk_\alpha\rangle = spl\ stk_2\ \texttt{in}\\
&\qquad\qquad\texttt{let}\ \langle stk_1, res\rangle = f\ stk_1\ \texttt{in}\\
&\qquad\qquad\texttt{let}\ stk_2 = cmb\ {}^{\mathsf{L}}\langle stk_1, stk_\alpha\rangle\ \texttt{in}\\
&\qquad\qquad{}^{\mathsf{L}}\langle stk_2, res\rangle
\end{aligned}$$

Figure 5.10: Translation from $\mathsf{F}^{\mathsf{RGN}}$ to rgnURAL (region reference subtyping (II))

Expressions

$$\mathbb{E}\left[\!\!\left[\frac{\Delta \vdash_{\text{index}} \theta_1 \qquad \Delta \vdash_{\text{index}} \theta_2 \qquad \Delta \vdash_{\text{type}} \tau \qquad \Delta;\Gamma \vdash_{\text{exp}} e_1 : \mathsf{RGNPf}(\theta_1 \preceq \theta_2) \qquad \Delta;\Gamma \vdash_{\text{exp}} e_2 : \mathsf{RGNRef}\ \theta_1\ \tau}{\Delta;\Gamma \vdash_{\text{exp}} \texttt{coerceRGNRef}\ [\theta_1]\ [\theta_2]\ [\tau]\ e_1\ e_2 : \mathsf{RGNRef}\ \theta_2\ \tau}\right]\!\!\right] =$$

$$\begin{array}{l}
\texttt{let}\ \mathit{ppf}{:}\mathbb{T}[\![\mathsf{RGNPf}(\theta_1 \preceq \theta_2)]\!] = \mathbb{E}[\![e_1]\!]\ \texttt{in} \\
\texttt{let}\ \mathit{pref}{:}\mathbb{T}[\![\mathsf{RGNRef}\ \theta_1\ \tau]\!] = \mathbb{E}[\![e_2]\!]\ \texttt{in} \\
\texttt{let}\ \texttt{pack}(\alpha, \langle \mathit{spl}, \mathit{cmb}\rangle) = \mathit{ppf}\ \texttt{in} \\
\texttt{let}\ \texttt{pack}(\varrho, \langle \texttt{pack}(\beta, \langle \mathit{prj}, \mathit{inj}\rangle), \mathit{ref}\rangle) = \mathit{pref}\ \texttt{in} \\
\texttt{let}\ \mathit{prj}' = {}^{\mathsf{U}}\lambda \mathit{stk}_2{:}\mathbb{T}[\![\theta_2]\!]\,.\,\texttt{let}\ \langle \mathit{stk}_1, \mathit{stk}_\alpha\rangle = \mathit{spl}\ \mathit{stk}_2\ \texttt{in} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \texttt{let}\ \langle \mathit{stk}_\beta, \mathit{cap}_\varrho\rangle = \mathit{prj}\ \mathit{stk}_1\ \texttt{in} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad {}^{\mathsf{L}}\langle {}^{\mathsf{L}}\langle \mathit{stk}_\alpha, \mathit{stk}_\beta\rangle, \mathit{cap}_\varrho\rangle \qquad \texttt{in} \\
\texttt{let}\ \mathit{inj}' = {}^{\mathsf{U}}\lambda s{:}{}^{\mathsf{L}}({}^{\mathsf{L}}(\alpha \otimes \beta) \otimes {}^{\mathsf{L}}(\overline{\mathsf{Cap}}\ \varrho)).\,\texttt{let}\ \langle\langle \mathit{stk}_\alpha, \mathit{stk}_\beta\rangle, \mathit{cap}_\varrho\rangle = s\ \texttt{in} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \texttt{let}\ \mathit{stk}_1 = \mathit{inj}\ {}^{\mathsf{L}}\langle \mathit{stk}_\beta, \mathit{cap}_\varrho\rangle\ \texttt{in} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{cmb}\ {}^{\mathsf{L}}\langle \mathit{stk}_1, \mathit{stk}_\alpha\rangle \qquad \texttt{in} \\
{}^{\mathsf{U}}\texttt{pack}(\varrho, {}^{\mathsf{U}}\langle {}^{\mathsf{U}}\texttt{pack}({}^{\mathsf{L}}(\alpha \otimes \beta), {}^{\mathsf{U}}\langle \mathit{prj}', \mathit{inj}'\rangle), \mathit{ref}\rangle)
\end{array}$$

Figure 5.11: Translation from $\mathsf{F}^{\mathsf{RGN}}$ to rgnURAL (region reference subtyping (III))

this corresponds to the fact that this coercion does not actually copy the contents of the reference from one region to another.

Finally, Figure 5.12 shows the translations of `reflRGNPf` and `transRGNPf`. The translation of reflexivity simply uses a dummy $^{\mathsf{L}}\langle\rangle$ term as the "slack". The translation of transitivity is much like the translation of `coerceRGNRef`; it combines the isomorphism for $\mathsf{RGNPf}(\theta_1 \preceq \theta_2)$, with "slack" α, with the isomorphism for $\mathsf{RGNPf}(\theta_2 \preceq \theta_3$, with "slack" β, to construct the isomorphism for $\mathsf{RGNPf}(\theta_1 \preceq \theta_3)$, with "slack" $^{\mathsf{L}}(\alpha \otimes \beta)$.

5.4 Effect Polymorphism

Recall that effect polymorphism provides a means to abstract over an effect (a set of regions). Effect instantiation applies an effect abstraction to an effect. Effect polymorphism is especially useful for typing higher-order functions. For example, the type of the list `map` function should be polymorphic in the effect of the functional argument. We note that effect polymorphism is most useful in the presence of type polymorphism. While we have presented the region calculi in Chapter 2 as a monomorphic languages, adding type polymorphism is entirely orthogonal to the development thus far.

Our translation from the Single Effect Calculus to $\mathsf{F}^{\mathsf{RGN}}$ was simplified by using a single index (variable) for the RGN monad. We introduced the $\mathsf{RGNPf}(\theta_1 \preceq \theta_2)$ as a witness to the relationship between the indices θ_1 and θ_2; as such, it is closely related to the $\Delta \vdash_{\mathrm{rr}} \rho \succeq \rho'$ judgment. We noted in the previous section, that we could represent each of the rules for the $\Delta \vdash_{\mathrm{rr}} \rho \succeq \rho'$ judgment as an explicit coercion term in $\mathsf{F}^{\mathsf{RGN}}$, which in turn could be given a translation into $\mathsf{rgnURAL}$.

Expressions

$$\mathbb{E}\left[\!\!\left[\frac{\vdash_{\text{ctxt}} \Delta;\Gamma \qquad \Delta \vdash_{\text{index}} \theta}{\Delta;\Gamma \vdash_{\text{exp}} \texttt{reflRGNPf}\ [\theta] : \mathsf{RGNPf}(\theta \preceq \theta)}\right]\!\!\right] =$$

$$\begin{array}{l}
\texttt{let}\ spl = {}^{\mathsf{U}}\lambda stk{:}\mathbb{T}[\![\theta]\!]\,.\,\texttt{let}\ su = {}^{\mathsf{L}}\langle stk, {}^{\mathsf{L}}\langle\rangle\rangle\ \texttt{in}\ su\ \texttt{in} \\
\texttt{let}\ cmb = {}^{\mathsf{U}}\lambda su{:}{}^{\mathsf{L}}(\mathbb{T}[\![\theta]\!] \otimes {}^{\mathsf{L}}\mathbf{1}_{\otimes}).\,\texttt{let}\ \langle stk, \langle\rangle\rangle = su\ \texttt{in}\ stk\ \texttt{in} \\
{}^{\mathsf{U}}\mathsf{pack}({}^{\mathsf{L}}\mathbf{1}_{\otimes}, {}^{\mathsf{U}}\langle spl, cmb\rangle)
\end{array}$$

$$\mathbb{E}\left[\!\!\left[\frac{\begin{array}{c}\Delta \vdash_{\text{index}} \theta_1 \qquad \Delta \vdash_{\text{index}} \theta_2 \qquad \Delta \vdash_{\text{index}} \theta_3 \\ \Delta;\Gamma \vdash_{\text{exp}} e_1 : \mathsf{RGNPf}(\theta_1 \preceq \theta_2) \\ \Delta;\Gamma \vdash_{\text{exp}} e_2 : \mathsf{RGNPf}(\theta_2 \preceq \theta_3)\end{array}}{\Delta;\Gamma \vdash_{\text{exp}} \texttt{transRGNPf}\ [\theta_1]\ [\theta_2]\ [\theta_3]\ e_1\ e_2 : \mathsf{RGNPf}(\theta_1 \preceq \theta_3)}\right]\!\!\right] =$$

$$\begin{array}{l}
\texttt{let}\ ppf_1{:}\mathbb{T}[\![\mathsf{RGNPf}(\theta_1 \preceq \theta_2)]\!] = \mathbb{E}[\![e_1]\!]\ \texttt{in} \\
\texttt{let}\ ppf_2{:}\mathbb{T}[\![\mathsf{RGNPf}(\theta_2 \preceq \theta_3)]\!] = \mathbb{E}[\![e_2]\!]\ \texttt{in} \\
\texttt{let}\ \mathsf{pack}(\alpha, \langle spl_\alpha, cmb_\alpha\rangle) = ppf_1\ \texttt{in} \\
\texttt{let}\ \mathsf{pack}(\beta, \langle spl_\beta, cmb_\beta\rangle) = ppf_2\ \texttt{in} \\
\texttt{let}\ spl = {}^{\mathsf{U}}\lambda stk_3{:}\mathbb{T}[\![\theta_3]\!]\,.\,\texttt{let}\ \langle stk_2, stk_\beta\rangle = spl_\beta\ stk_3\ \texttt{in} \\
\qquad\qquad\qquad\qquad \texttt{let}\ \langle stk_1, stk_\alpha\rangle = spl_\alpha\ stk_2\ \texttt{in} \\
\qquad\qquad\qquad\qquad {}^{\mathsf{L}}\langle stk_1, {}^{\mathsf{L}}\langle stk_\alpha, stk_\beta\rangle\rangle \quad \texttt{in} \\
\texttt{let}\ cmb = {}^{\mathsf{U}}\lambda s{:}{}^{\mathsf{L}}(\mathbb{T}[\![\theta_1]\!] \otimes {}^{\mathsf{L}}(\alpha \otimes \beta)).\,\texttt{let}\ \langle stk_1, \langle stk_\alpha, stk_\beta\rangle\rangle = s\ \texttt{in} \\
\qquad\qquad\qquad\qquad\qquad\qquad \texttt{let}\ stk_2 = cmb_\alpha\ {}^{\mathsf{L}}\langle stk_1, stk_\alpha\rangle\ \texttt{in} \\
\qquad\qquad\qquad\qquad\qquad\qquad cmb_\beta\ {}^{\mathsf{L}}\langle stk_2, stk_\beta\rangle \quad \texttt{in} \\
{}^{\mathsf{U}}\mathsf{pack}({}^{\mathsf{L}}(\alpha \otimes \beta), {}^{\mathsf{U}}\langle spl, cmb\rangle)
\end{array}$$

Figure 5.12: Translation from $\mathsf{F}^{\mathsf{RGN}}$ to rgnURAL (region reference subtyping (IV))

If we were to adopt a source calculus with effects given by

$$\begin{array}{c}\text{Effects}\\ \phi \;::=\; \emptyset \mid \{\rho\} \mid \varepsilon \mid \phi_1 \cup \phi_2\end{array}$$

where effects may be any combination of regions and effect variables, then we would need to introduce judgments and rules to handle the various relationships between regions and effects; for example, the judgment $\Delta \vdash_{\text{ee}} \phi \supseteq \phi'$ would assert that all regions and effect variables in ϕ' are in ϕ.

Just as we represented the judgment $\Delta \vdash_{\text{rr}} \rho \succeq \rho'$ as the $\mathsf{F}^{\mathsf{RGN}}$ type $\mathsf{RGNPf}(\theta_1 \preceq \theta_2)$, we would represent the judgment $\Delta \vdash_{\text{ee}} \phi \supseteq \theta'$ as a new (abstract) $\mathsf{F}^{\mathsf{RGN}}$ type, say $\mathsf{RGNPf}(\Theta_1 \subseteq \Theta_2)$. And just as we represented each of the rules for the $\Delta \vdash_{\text{rr}} \rho \succeq \rho'$ judgment as an explicit coercion term in $\mathsf{F}^{\mathsf{RGN}}$, we would represent each of the rules for the $\Delta \vdash_{\text{rr}} \phi \supseteq \phi'$ judgment as new coercion terms in $\mathsf{F}^{\mathsf{RGN}}$.

For example, the rule

$$\frac{\Delta \vdash_{\text{ee}} \phi \supseteq \phi_1 \qquad \Delta \vdash_{\text{ee}} \phi \supseteq \phi_2}{\Delta \vdash_{\text{ee}} \phi \supseteq \phi_1 \cup \phi_2}$$

would be witnessed by a coercion term with the type:

$$\forall \varphi, \varphi_1, \varphi_2.\, \mathsf{RGNPf}(\varphi_1 \subseteq \varphi_2) \to \mathsf{RGNPf}(\varphi_2 \subseteq \varphi) \to \mathsf{RGNPf}(\varphi_1 \cup \varphi_2 \subseteq \varphi)$$

Simply put, in witnessing "sub-effecting" through explicit coercions, we need to introduce additional terms into the $\mathsf{F}^{\mathsf{RGN}}$ language. We note that the situation is really no better in a language with subtyping (e.g., $\mathsf{System\ F}_{\leq}$), as the subset relation is "richer" than the subtype relation on standard types (e.g., product types).

We believe that the translation into rgnURAL would realize each of these explicit coercions terms via various isomorphisms between different "views" of the set of allocated regions (represented as a collection of region capabilities). Making this intuition concrete is an interesting direction for future work.

5.5 High-Level Language Features of Cyclone

As we have noted previously, one of the goals of the Cyclone language has been to give programmers as much control over memory management as possible, while retaining safety through strong static typing. This goal has led to the development and integration of a number memory-management features, using regions as an organizing principle. While the efficacy of this design has been shown [44], an argument that justifies the soundness of the various memory-management features has proved elusive, due to sheer complexity.

In this section, we introduce a number of Cyclone's high-level features. We wish to show that the substructural language and type system of Chapter 4 is a simple target language into which we may translate the key features of Cyclone.

This will help establish rgnURAL as a core language that captures many features in a unified model.[1] Type soundness for rgnURAL then implies the soundness of more advanced features under suitable encodings. These encodings exhibit where high-level features compose well with one another and where they don't; in other words, the encodings help manage the complexity of related, but subtly different, features. Finally, these encodings help identify opportunities for new high-level features that emerge naturally from the lower-level core language.

[1] We remark that it is our intention that rgnURAL serve as a compiler intermediate language and as vehicle for formal reasoning, not as a high-level programming language.

We note that while this exercise is partially motivated by the fact that Cyclone's type system has not been formally proven sound, we do not expect to discover unsoundness in the type system. Rather, we wish to demonstrate the suitability of rgnURAL and to provide additional justification for the integration of features in Cyclone.

5.5.1 Key Features of Cyclone

The Cyclone language [17] is a type-safe dialect of C. Cyclone attempts to give programmers significant control over data representation, memory management, and performance (like C), while preventing buffer overflows, format-string attacks, and dangling-pointer dereferences (unlike C). Cyclone ensures the safety of programs through a combination of compile-time and run-time checks; the compile-time checks use a combination of programmer annotations, an advanced type-and-effect system, and a simple flow analysis. Cyclone's combination of performance, control, and safety make it a good language for writing low-level software, like runtime systems and device drivers.

In this section, we briefly introduce the key features of Cyclone that we wish to model in rgnURAL.

As with C programs, Cyclone programs make extensive use of pointers. Cyclone pointer types have the form `t*@region('r)`, where `t` is the type of the pointed to object, and `'r` is a region name describing the object's lifetime. The Cyclone type system tracks the set of live regions at each program point; dereferencing a pointer of type `t*@region('r)` requires that the region `'r` is in the set. (If `'r` is not in the set, a compile-time error signals a possible dangling pointer dereference.) Region polymorphism lets functions and data-structures abstract over the region

of their arguments and fields. Region parameters in Cyclone are indicated by annotations of the form `<‘r::R>` on functions and types, where `R` distinguishes region parameters from other kinds of parameters.

Cyclone provides a number of different kinds of regions, suitable for different allocation and deallocation patterns. *Stack regions* correspond to local-declaration blocks: entering a block creates a region and immediately allocates objects, while exiting the block destroys the region and deallocates the region's objects. Hence, a stack region has lexical scope, but the number and sizes of stack allocated objects is fixed at compile time. Pointers to stack allocated objects are assigned the name of the stack region, thereby preventing such pointers from being dereferenced outside the scope of the stack region.

Lexical regions are also created and deallocated according to program scoping, but a region handle allows objects to be allocated into the region throughout the region's lifetime. Cyclone uses the syntax

```
{ region<‘r> rh; ... }
```

to introduce lexical regions. This syntax defines the name (`‘r`) of the new lexical region and it defines the handle (`rh` of type `region_t<‘r>`) used to allocate memory in the new lexical region. The region is created when execution enters the `region` block and destroyed when execution exits the block. Note that a lexical region has lexical scope, but the number and sizes of region allocated objects are not fixed at compile time.

The Cyclone heap is a special region with the name `‘H`. All data allocated in the heap is managed by the Boehm-Demers-Weiser (BDW) conservative garbage collector [7]. Conceptually, the Cyclone heap is just a lexical region with global scope (which is never destroyed), and the global variable `heap_region` is its handle.

The lifetimes of stack and lexical regions follow the block structure of the program, being created and destroyed in a last-in-first-out (LIFO) discipline. As we have noted, such a discipline can be too restrictive, as it can not accommodate objects with overlapping, non-nested lifetimes. In many instances, the limitations associated with stack and lexical regions can be overcome by using the heap region, whose contents are periodically garbage collected. However, garbage collection is not suitable for all application domains. For example, embedded systems, OS kernels and device drivers, and network servers may require bounds on memory or real-time guarantees that are hard to achieve with garbage collection.

Hence, recent work has added *unique pointers* and *dynamic regions* to Cyclone [43, 77]. Unique pointers are based on insights from linear type systems and provide fine-grained memory management for individual objects. In particular, a the object pointed to by a unique pointer can be deallocated at any program point. On the other hand, unique pointers cannot be freely copied and there are further restrictions on their use in Cyclone programs. While there is much more to be said concerning Cyclone's unique pointers, for our purposes it suffices to reiterate that unique pointers are treated as linear objects, where the type system and a conventional flow analysis ensures that, at every program point, there is at most one usable copy of a value assigned a unique-pointer type.

Integrating unique pointers in Cyclone requires that we generalize the form of pointer types to `t*@region(‘r)@aqual(\q)`, where `\q` is an aliasability qualifier. A unique pointer has the qualifier `\U`, while a traditional (aliasable) pointer has the qualifier `\A`.[2]

[2]We note that the unique (`\U`) qualifier of Cyclone is related to the linear (L) and affine (A) qualifiers of rgnURAL, while the aliasable (`\A`) qualifier is related to the unrestricted (U) qualifier. We apologize for the unfortunate clash in notation.

Cyclone's general memory allocation routine takes the form `rqmalloc(rh, qh, sz)`, where `rh` is a region handle (of type `region_t<‘r>`), `qh` is an aliasability qualifier (of type `aqual_t<\q>`, drawn from the constants `unique_qual` and `alias_qual`), and `sz` is the amount of memory to allocate; the routine returns a pointer of type `t*@region(‘r)@aqual(\q)` (for an appropriate type `t`). Thus, both aliasable and unique objects may be allocated in any region. However, only unique pointers may be deallocated, using a deallocation routine of the form `rfree(rh, p)`, where `rh` is a region handle (of type `region_t<‘r>`) and `p` is a unique pointer (of type `t*@region(‘r)@aqual(\U)`). Hence, a Cyclone programmer may (explicitly) deallocate some objects in a region individually, and (implicitly) deallocate the rest of the objects when the region is destroyed. This strategy can improve the space/time overhead as compared to traditional regions; the term *reap* has been coined to describe regions that support this hybrid strategy [6, 42]

A dynamic region resembles a lexical region in many ways; the crucial difference is that a dynamic region can be created and destroyed at (almost) any point within a program. However, before accessing or allocating data within a dynamic region, the region must be *opened*. Opening a dynamic region adds the region to the set of live regions and prevents the region from being destroyed while it is open. The interface for creating and destroying dynamic regions is given by the following:

```
typedef struct DynamicRegion<‘r>*@aqual(\U)
   uregion_key_t<‘r::R>;
struct NewDynamicRegion { <‘r::R>
   uregion_key_t<‘r> key;
};
```

```
struct NewDynamicRegion new_ukey();
void free_ukey(uregion_key_t<`r> k);
```

A dynamic region is represented as a unique pointer to an abstract `struct DynamicRegion<`r>` (which is parameterized by the region `` `r `` and internally contains the handle to the region). This unique pointer is called the *key*, which serves as a run-time capability granting access to the region.

The `new_ukey` function creates a fresh dynamic region and returns the unique key for the region. (The `<`r::R>` annotation in the `struct NewDynamicRegion` type indicates that the region variable is existentially bound. Unpacking this existential type yields a region variable which does not conflict with any other region name. This is precisely the behavior we require for a function that creates a fresh region.) The `free_ukey` function destroys the key's region and the storage for the key. Since the key is unique, it must be used in a linear manner; the `free_ukey` function consumes the key.

Cyclone uses the syntax

```
{ region<`r> rh = open(k); ... }
```

to open a dynamic region. Within the scope, the region handle `rh` can be used to access `k`'s region; furthermore, `k` is temporarily consumed throughout the scope (preventing the region from being destroyed) and becomes accessible again when control leaves the scope.

5.5.2 The Cyc Language

Although the previous section has given an informal overview of the high-level features of Cyclone, it will be helpful to construct a more formal model of Cyclone

before sketching a translation into rgnURAL. In this section, we introduce the Cyc language, which may be seen as a rough approximation of Cyclone, cast in the spirit of the work presented thus far. In particular, we approximate Cyclone's type system with a novel type system that combines both monadic and substructural elements.

Figures 5.13 and 5.14 present the syntax of Cyc. From rgnURAL, we adopt the substructural qualifiers (q) and structure our types as a qualifier applied to a pre-type. From $\mathsf{F}^{\mathsf{RGN}}$, we adopt indices ($\theta$) and the types $\overline{\mathsf{RGN}}$ and $\overline{\mathsf{Pf}}$. We note that indices in Cyc abstractly represent a stack of regions. Hence, as in $\mathsf{F}^{\mathsf{RGN}}$, $\overline{\mathsf{RGN}}\ \theta\ \tau$ is the type of a computation which transforms a region stack indexed by θ and delivers a value of type τ; $\overline{\mathsf{Pf}}(\theta_1 \preceq \theta_2)$ is the type of a witness to the fact that the stack indexed by θ_1 is a subtype of the stack indexed by θ_2; $\overline{\mathsf{Ref}}\ \theta\ \tau$ is the type of references allocated in some region in the stack indexed by θ; $\overline{\mathsf{Hnd}}\ \theta\ \tau$ is the type of region handles for some region in the stack indexed by θ. We introduce $\overline{\mathsf{Key}}\ \theta$ as the type of a *dynamic-region key*, which serves as a capability for accessing some region in the stack indexed by θ.

As will become clear when we examine the typing rules for Cyc, this indirect representation of a region via a stack indexed by θ helps to model a number of high-level features of Cyclone. Essentially, the set of live regions at each program point, which is tracked by the Cyclone type system, is captured by the collection of $\overline{\mathsf{Pf}}$ terms and the index θ of a $\overline{\mathsf{RGN}}$ computation.

We note that a unique pointer in Cyclone (`t*@region(‘r)@aqual(\U)`) corresponds to the Cyc type $^{\mathsf{A}}(\overline{\mathsf{Ref}}\ \theta\ \tau)$ (since Cyclone allows a unique pointer to be implicitly discarded), while an aliasable pointer (`t*@region(‘r)@aqual(\A)`) corresponds to the type $^{\mathsf{U}}(\overline{\mathsf{Ref}}\ \theta\ \tau)$. Since Cyc assigns a substructural qualifier to all

Constant Qualifiers

$$\mathsf{q} \quad \in \quad \mathit{CQuals} = \{\mathsf{U}, \mathsf{R}, \mathsf{A}, \mathsf{L}\}$$

Qualifier variables

$$\xi \quad \in \quad \mathit{QVars}$$

Qualifiers

$$q \quad ::= \quad \xi \mid \mathsf{q}$$

Pre-type variables

$$\overline{\alpha} \quad \in \quad \mathit{PTVars}$$

Pre-types

$$\begin{aligned}\overline{\tau} \quad ::= \quad & \overline{\alpha} \mid \overline{\mathsf{Int}} \mid \overline{\mathsf{Bool}} \mid \tau_1 \multimap \tau_2 \mid \tau_1 \otimes \cdots \otimes \tau_n \mid \\ & \overline{\forall}\xi.\,\tau \mid \overline{\exists}\xi.\,\tau \mid \overline{\forall}\overline{\alpha}.\,\tau \mid \overline{\exists}\overline{\alpha}.\,\tau \mid \overline{\forall}\alpha.\,\tau \mid \overline{\exists}\alpha.\,\tau \mid \\ & \overline{\mathsf{RGN}}\ \theta\ \tau \mid \overline{\mathsf{Pf}}(\theta_1 \preceq \theta_2) \mid \overline{\mathsf{Ref}}\ \theta\ \tau \mid \overline{\mathsf{Hnd}}\ \theta \mid \overline{\mathsf{Key}}\ \theta \mid \overline{\forall}\vartheta.\,\tau \mid \overline{\exists}\vartheta.\,\tau\end{aligned}$$

Type variables

$$\alpha \quad \in \quad \mathit{TVars}$$

Types

$$\tau \quad ::= \quad \alpha \mid {}^{q}\overline{\tau}$$

Index variables

$$\vartheta \quad \in \quad \mathit{IVars}$$

Indices

$$\theta \quad ::= \quad \vartheta$$

Figure 5.13: Syntax of Cyc (I)

Integer constants

$$\mathfrak{i} \quad \in \quad \mathbb{Z}$$

Boolean constants

$$\mathfrak{b} \quad \in \quad \{\texttt{true}, \texttt{false}\}$$

Value variables

$$f, x \quad \in \quad \mathit{VVars}$$

Terms

$$\begin{aligned} e \quad ::= \quad & {}^{q}\mathfrak{i} \mid {}^{q}(e_1 \oplus e_2) \mid {}^{q}(e_1 \otimes e_2) \mid {}^{q}\mathfrak{b} \mid \texttt{if } e_b \texttt{ then } e_t \texttt{ else } e_f \mid \\ & x \mid {}^{q}\lambda x{:}\tau.\, e \mid e_1\ e_2 \mid \\ & {}^{q}\langle e_1, \ldots, e_n \rangle \mid \texttt{let } \langle x_1, \ldots, x_n \rangle = e_a \texttt{ in } e_b \mid \\ & {}^{q}\Lambda\xi.\, e \mid e\ [q] \mid {}^{q}\texttt{pack}(q, e) \mid \texttt{let pack}(\xi, x) = e_a \texttt{ in } e_b \mid \\ & {}^{q}\Lambda\overline{\alpha}.\, e \mid e\ [\overline{\tau}] \mid {}^{q}\texttt{pack}(\overline{\tau}, e) \mid \texttt{let pack}(\overline{\alpha}, x) = e_a \texttt{ in } e_b \mid \\ & {}^{q}\Lambda\alpha.\, e \mid e\ [\tau] \mid {}^{q}\texttt{pack}(\tau, e) \mid \texttt{let pack}(\alpha, x) = e_a \texttt{ in } e_b \mid \\ & \texttt{fix } f{:}\tau.u \mid \texttt{let } x = e_a \texttt{ in } e_b \mid \\ & \texttt{runRGN } e \mid \texttt{returnRGN } e \mid \texttt{thenRGN } e_a\ e_f \mid \texttt{reflPf} \mid \texttt{transPf } e_1\ e_2 \mid \\ & \texttt{coerceRGN } e_p\ e_k \mid \texttt{coerceRef } e_p\ e_r \mid \texttt{coerceHnd } e_p\ e_h \mid \\ & {}^{q_r}\texttt{new } e_h\ e_\star \mid \texttt{free } e_r \mid \texttt{read } e_r \mid \texttt{write } e_r\ e_\star \mid \texttt{swap } e_h\ e_\star \mid \\ & \texttt{letRGN } e \mid \texttt{newKey} \mid \texttt{freeKey } e_k \mid \texttt{openKey } e_k\ e \mid \\ & {}^{q}\Lambda\vartheta.\, e \mid e\ [\theta] \mid {}^{q}\texttt{pack}(\vartheta, e) \mid \texttt{let pack}(\theta, x) = e_a \texttt{ in } e_b \end{aligned}$$

Abstractions

$$u \quad ::= \quad {}^{q}\lambda x{:}\tau.\, e \mid {}^{q}\Lambda\xi.\, e \mid {}^{q}\Lambda\overline{\alpha}.\, e \mid {}^{q}\Lambda\alpha.\, e \mid {}^{q}\Lambda\vartheta.\, e$$

Figure 5.14: Syntax of Cyc (II)

pre-types (not just reference pre-types), we use the Cyc type $^{\mathsf{L}}(\overline{\mathsf{Key}}\ \theta)$ to represent the Cyclone type `uregion_key_t<'r>`.[3]

The term syntax for Cyc should be mostly familiar, as it adopts terms from $\mathsf{F}^{\mathsf{RGN}}$ and rgnURAL. Note that the reference primitives (`new`, `free`, `read`, `write`, and `swap`) do *not* take a capability argument (as they did in rgnURAL). Instead, the typing rules will assign these terms monadic types (as they were in $\mathsf{F}^{\mathsf{RGN}}$); running a computation of pre-type $\overline{\mathsf{RGN}}\ \theta\ \tau$ will require that all of the regions in the stack indexed by θ are live.

The only new terms are those dealing with dynamic regions. The term `newKey` creates a new dynamic region and returns its key; it corresponds to the Cyclone function `new_ukey`. The term `freeKey` destroys a dynamic region and consumes its key; it corresponds to the Cyclone function `free_ukey`. Finally, the term `openKey` temporarily consumes a dynamic-region key, makes the dynamic-region handle available to a $\overline{\mathsf{RGN}}$ computation, and finally returns the result of the nested computation along with the dynamic-region key.

We elide most of the static semantics for Cyc, as it follows directly from the static semantics for rgnURAL. Figures 5.15–5.18 present the interesting typing rules for the judgment $\Delta;\Gamma \vdash_{\text{exp}} e : \tau$. The typing rules for many of the terms adopted from $\mathsf{F}^{\mathsf{RGN}}$ are similar to the corresponding rules in the static semantics for $\mathsf{F}^{\mathsf{RGN}}$. However, note the ways in which substructural qualifiers are used in these rules. In particular, note that $\overline{\mathsf{RGN}}$ is always qualified with L, while $\overline{\mathsf{Pf}}$ is always qualified with U. A monadic computation $\overline{\mathsf{RGN}}$ is qualified with L because it denotes a suspended computation; hence, like abstractions, the types of the free

[3]Using the Cyc type $^{\mathsf{A}}(\overline{\mathsf{Key}}\ \theta)$ would be slightly more accurate, as Cyclone allows a unique region key to be implicitly discarded. However, using $^{\mathsf{L}}(\overline{\mathsf{Key}}\ \theta)$ makes the translation in the next section somewhat more uniform.

$\boxed{\Delta; \Gamma \vdash_{\text{exp}} e : \tau}$

$$\frac{\Delta; \Gamma \vdash_{\text{exp}} e : {}^{\mathsf{L}}(\overline{\forall}\vartheta.\, {}^{\mathsf{L}}(\overline{\mathsf{RGN}}\ \vartheta\ \tau))}{\Delta; \Gamma \vdash_{\text{exp}} \texttt{runRGN}\ e : \tau}$$

$$\frac{\Delta \vdash_{\text{index}} \theta \qquad \Delta; \Gamma \vdash_{\text{exp}} e : \tau}{\Delta; \Gamma \vdash_{\text{exp}} \texttt{returnRGN}\ e : {}^{\mathsf{L}}(\overline{\mathsf{RGN}}\ \theta\ \tau)}$$

$$\frac{\begin{array}{c}\Delta \vdash \Gamma \rightsquigarrow \Gamma_a \boxdot \Gamma_f \\ \Delta; \Gamma_a \vdash_{\text{exp}} e_a : {}^{\mathsf{L}}(\overline{\mathsf{RGN}}\ \theta\ \tau_a) \\ \Delta; \Gamma_f \vdash_{\text{exp}} e_f : {}^{q}(\tau_a \multimap {}^{\mathsf{L}}(\overline{\mathsf{RGN}}\ \theta\ \tau_a))\end{array}}{\Delta; \Gamma \vdash_{\text{exp}} \texttt{thenRGN}\ e_a\ e_f : {}^{\mathsf{L}}(\overline{\mathsf{RGN}}\ \theta\ \tau_b)}$$

$$\frac{\Delta \vdash_{\text{index}} \theta}{\Delta; \{\} \vdash_{\text{exp}} \texttt{reflPf} : {}^{\mathsf{U}}(\overline{\mathsf{Pf}}(\theta \preceq \theta))}$$

$$\frac{\begin{array}{c}\Delta \vdash \Gamma \rightsquigarrow \Gamma_1 \boxdot \Gamma_2 \\ \Delta; \Gamma_1 \vdash_{\text{exp}} e_1 : {}^{\mathsf{U}}(\overline{\mathsf{Pf}}(\theta_1 \preceq \theta_2)) \\ \Delta; \Gamma_2 \vdash_{\text{exp}} e_2 : {}^{\mathsf{U}}(\overline{\mathsf{Pf}}(\theta_2 \preceq \theta_3))\end{array}}{\Delta; \Gamma \vdash_{\text{exp}} \texttt{transPf}\ e_1\ e_2 : {}^{\mathsf{U}}(\overline{\mathsf{Pf}}(\theta_1 \preceq \theta_3))}$$

$$\frac{\begin{array}{c}\Delta \vdash \Gamma \rightsquigarrow \Gamma_p \boxdot \Gamma_k \\ \Delta; \Gamma_p \vdash_{\text{exp}} e_p : {}^{\mathsf{U}}(\overline{\mathsf{Pf}}(\theta_1 \preceq \theta_2)) \qquad \Delta; \Gamma_k \vdash_{\text{exp}} e_k : {}^{\mathsf{L}}(\overline{\mathsf{RGN}}\ \theta_1\ \tau)\end{array}}{\Delta; \Gamma \vdash_{\text{exp}} \texttt{coerceRGN}\ e_p\ e_k : {}^{\mathsf{L}}(\overline{\mathsf{RGN}}\ \theta_2\ \tau)}$$

$$\frac{\begin{array}{c}\Delta \vdash \Gamma \rightsquigarrow \Gamma_p \boxdot \Gamma_r \\ \Delta; \Gamma_p \vdash_{\text{exp}} e_p : {}^{\mathsf{U}}(\overline{\mathsf{Pf}}(\theta_1 \preceq \theta_2)) \qquad \Delta; \Gamma_r \vdash_{\text{exp}} e_r : {}^{q_r}(\overline{\mathsf{Ref}}\ \theta_1\ \tau)\end{array}}{\Delta; \Gamma \vdash_{\text{exp}} \texttt{coerceRef}\ e_p\ e_r : {}^{q_r}(\overline{\mathsf{Ref}}\ \theta_2\ \tau)}$$

$$\frac{\Delta \vdash \Gamma \rightsquigarrow \Gamma_p \boxdot \Gamma_h \qquad \Delta; \Gamma_p \vdash_{\text{exp}} e_p : {}^{\mathsf{U}}(\overline{\mathsf{Pf}}(\theta_1 \preceq \theta_2)) \qquad \Delta; \Gamma_h \vdash_{\text{exp}} e_h : {}^{q_h}(\overline{\mathsf{Hnd}}\ \theta_1)}{\Delta; \Gamma \vdash_{\text{exp}} \texttt{coerceHnd}\ e_p\ e_h : {}^{q_h}(\overline{\mathsf{Hnd}}\ \theta_2)}$$

Figure 5.15: Static semantics of Cyc (expressions (I))

$\boxed{\Delta; \Gamma \vdash_{\text{exp}} e : \tau}$

NEW(ANY)

$$\frac{\Delta \vdash_{\text{qual}} q_r \qquad \Delta \vdash \Gamma \rightsquigarrow \Gamma_h \boxdot \Gamma_a \qquad \Delta; \Gamma_h \vdash_{\text{exp}} e_h : {}^{q_h}(\overline{\mathsf{Hnd}}\ \theta) \qquad \Delta; \Gamma_a \vdash_{\text{exp}} e_\star : \tau \qquad \Delta \vdash \tau \preceq \mathsf{A}}{\Delta; \Gamma \vdash_{\text{exp}} {}^{q_r}\texttt{new}\ e_h\ e_\star : {}^{\mathsf{L}}(\overline{\mathsf{RGN}}\ \theta\ {}^{q_r}(\overline{\mathsf{Ref}}\ \theta\ \tau))}$$

NEW(R,L)

$$\frac{\Delta \vdash_{\text{qual}} q_r \qquad \Delta \vdash \Gamma \rightsquigarrow \Gamma_h \boxdot \Gamma_a \qquad \Delta; \Gamma_h \vdash_{\text{exp}} e_h : {}^{q_h}(\overline{\mathsf{Hnd}}\ \theta) \qquad \Delta; \Gamma_a \vdash_{\text{exp}} e_\star : \tau \qquad \Delta \vdash \mathsf{R} \preceq q_r}{\Delta; \Gamma \vdash_{\text{exp}} {}^{q_r}\texttt{new}\ e_h\ e_\star : {}^{\mathsf{L}}(\overline{\mathsf{RGN}}\ \theta\ \ {}^{q_r}(\overline{\mathsf{Ref}}\ \theta\ \tau))}$$

$$\frac{\Delta; \Gamma \vdash_{\text{exp}} e_r : {}^{q_r}(\overline{\mathsf{Ref}}\ \theta\ \tau) \qquad \Delta \vdash \mathsf{A} \preceq q_r}{\Delta; \Gamma \vdash_{\text{exp}} \texttt{free}\ e_r : {}^{\mathsf{L}}(\overline{\mathsf{RGN}}\ \theta\ \tau)}$$

$$\frac{\Delta; \Gamma \vdash_{\text{exp}} e_r : {}^{q_r}(\overline{\mathsf{Ref}}\ \rho\ \tau) \qquad \Delta \vdash \tau \preceq \mathsf{R}}{\Delta; \Gamma \vdash_{\text{exp}} \texttt{read}\ e_r : {}^{\mathsf{L}}(\overline{\mathsf{RGN}}\ \theta\ {}^{\mathsf{L}}({}^{q_r}(\overline{\mathsf{Ref}}\ \theta\ \tau) \otimes \tau))}$$

Figure 5.16: Static semantics of Cyc (expressions (II))

$\boxed{\Delta; \Gamma \vdash_{\text{exp}} e : \tau}$

WRITE(WEAK)

$$\frac{\Delta \vdash \Gamma \rightsquigarrow \Gamma_r \boxdot \Gamma_\star \qquad \Delta; \Gamma_r \vdash_{\text{exp}} e_r : {}^{q_r}(\overline{\mathsf{Ref}}\ \theta\ \tau) \qquad \Delta \vdash \tau \preceq \mathsf{A} \qquad \Delta; \Gamma_\star \vdash_{\text{exp}} e_\star : \tau}{\Delta; \Gamma \vdash_{\text{exp}} \texttt{write}\ e_r\ e_\star : {}^{\mathsf{L}}(\overline{\mathsf{RGN}}\ \theta\ {}^{q_r}(\overline{\mathsf{Ref}}\ \theta\ \tau))}$$

WRITE(STRONG)

$$\frac{\Delta \vdash \Gamma \rightsquigarrow \Gamma_r \boxdot \Gamma_\star \qquad \Delta; \Gamma_r \vdash_{\text{exp}} e_r : {}^{q_r}(\overline{\mathsf{Ref}}\ \theta\ \tau) \qquad \Delta \vdash \tau \preceq \mathsf{A} \qquad \Delta; \Gamma_\star \vdash_{\text{exp}} e_\star : \tau_\star \qquad \Delta \vdash \mathsf{A} \preceq q_r \qquad \Delta \vdash \tau_\star \preceq q_r}{\Delta; \Gamma \vdash_{\text{exp}} \texttt{write}\ e_r\ e_\star : {}^{\mathsf{L}}(\overline{\mathsf{RGN}}\ \theta\ {}^{q_r}(\overline{\mathsf{Ref}}\ \rho\ \tau_\star))}$$

SWAP(WEAK)

$$\frac{\Delta \vdash \Gamma \rightsquigarrow \Gamma_r \boxdot \Gamma_\star \qquad \Delta; \Gamma_r \vdash_{\text{exp}} e_r : {}^{q_r}(\overline{\mathsf{Ref}}\ \theta\ \tau) \qquad \Delta; \Gamma_\star \vdash_{\text{exp}} e_\star : \tau}{\Delta; \Gamma \vdash_{\text{exp}} \texttt{swap}\ e_r\ e_\star : {}^{\mathsf{L}}(\overline{\mathsf{RGN}}\ \theta\ {}^{\mathsf{L}}({}^{q_r}(\overline{\mathsf{Ref}}\ \rho\ \tau) \otimes \tau))}$$

SWAP(STRONG)

$$\frac{\Delta \vdash \Gamma \rightsquigarrow \Gamma_r \boxdot \Gamma_\star \qquad \Delta; \Gamma_r \vdash_{\text{exp}} e_r : {}^{q_r}(\overline{\mathsf{Ref}}\ \theta\ \tau) \qquad \Delta; \Gamma_\star \vdash_{\text{exp}} e_\star : \tau_\star \qquad \Delta \vdash \mathsf{A} \preceq q_r \qquad \Delta \vdash \tau_\star \preceq q_r}{\Delta; \Gamma \vdash_{\text{exp}} \texttt{swap}\ e_r\ e_\star : {}^{\mathsf{L}}(\overline{\mathsf{RGN}}\ \theta\ {}^{\mathsf{L}}({}^{q_r}(\overline{\mathsf{Ref}}\ \rho\ \tau_\star) \otimes \tau))}$$

Figure 5.17: Static semantics of Cyc (expressions (III))

variables in Γ should be bounded by the qualifier assigned to the computation. Since $\Delta \vdash \Gamma \preceq \mathsf{L}$ always hold, a simple means of achieving this to always qualify $\overline{\mathsf{RGN}}$ with L. On the other hand, a witness $\overline{\mathsf{Pf}}$ is qualified with U because it denotes a persistent fact about the relationship between two indices.

The rule for `runRGN` in Cyc is slightly different from the corresponding rule in $\mathsf{F}^{\mathsf{RGN}}$; the difference is that Cyc does not provide a region handle to the computation. This simply allows a computation, say opening and reading from a dynamic region, to be run without creating and destroying an extraneous lexical region.

Note that the rules for the reference primitives combine the corresponding rules from $\mathsf{F}^{\mathsf{RGN}}$ and rgnURAL. As in rgnURAL, the rules enforce the safe combinations of qualifiers for a reference and qualifiers for its contents. However, rather than take and return a capability argument (as in rgnURAL), the primitives are assigned $\overline{\mathsf{RGN}}$ types (as in $\mathsf{F}^{\mathsf{RGN}}$). The stack of regions implicitly threaded by the sequencing of monadic computations ensures that each of these primitives only access live regions.

The rules for `newKey` and `freeKey` are straight forward. Unsurprisingly, we may see that they are similar to the rules for `newrgn` and `freergn` in rgnURAL. Note that `newKey` has the type $^{\mathsf{L}}(\bar{\exists}\vartheta.\, ^{\mathsf{L}}(\overline{\mathsf{Key}}\ \vartheta))$; although we may read the type as asserting the existence of a stack of regions, with a key for accessing some region in the stack, the evaluation of `newKey` should create a new dynamic region and ϑ should denote the region stack with *only* the new dynamic region. We use a similar interpretation in the typing rule for `letRGN`: although the "inner" stack ϑ_i is universally quantified, we expect it to corresponds to the stack which extends the "outer" stack θ_o with one new region, to be destroyed at the end of the `letRGN`.

$\boxed{\Delta; \Gamma \vdash_{\text{exp}} e : \tau}$

$$\frac{\Delta; \Gamma \vdash_{\text{exp}} e : {}^{\mathsf{L}}(\overline{\forall}\vartheta_i.\, {}^{\mathsf{L}}(arg \multimap {}^{\mathsf{L}}(\overline{\mathsf{RGN}}\ \vartheta_i\ \tau))) \qquad arg \equiv {}^{\mathsf{L}}({}^{\mathsf{U}}(\overline{\mathsf{Pf}}(\theta_o \preceq \vartheta_i)) \otimes {}^{\mathsf{U}}(\overline{\mathsf{Hnd}}\ \vartheta_i))}{\Delta; \Gamma \vdash_{\text{exp}} \texttt{letRGN}\ e : {}^{\mathsf{L}}(\overline{\mathsf{RGN}}\ \theta_o\ \tau)}$$

$$\frac{}{\Delta; \{\} \vdash_{\text{exp}} \texttt{newKey} : {}^{\mathsf{L}}(\overline{\exists}\vartheta.\, {}^{\mathsf{L}}(\overline{\mathsf{Key}}\ \vartheta))} \qquad \frac{\Delta; \Gamma \vdash_{\text{exp}} e_k : {}^{\mathsf{L}}(\overline{\mathsf{Key}}\ \theta)}{\Delta; \Gamma \vdash_{\text{exp}} \texttt{freeKey}\ e_k : {}^{\mathsf{L}}\mathbf{1}_{\otimes}}$$

$$\frac{\Delta \vdash \Gamma \rightsquigarrow \Gamma_k \boxdot \Gamma_b \qquad \Delta; \Gamma_k \vdash_{\text{exp}} e_k : {}^{\mathsf{L}}(\overline{\mathsf{Key}}\ \theta) \qquad \Delta; \Gamma_b \vdash_{\text{exp}} e_b : {}^{\mathsf{L}}(\overline{\forall}\vartheta_i.\, {}^{\mathsf{L}}(arg \multimap {}^{\mathsf{L}}(\overline{\mathsf{RGN}}\ \vartheta_i\ \tau))) \qquad arg \equiv {}^{\mathsf{L}}({}^{\mathsf{U}}(\overline{\mathsf{Pf}}(\theta_o \preceq \vartheta_i)) \otimes {}^{\mathsf{U}}(\overline{\mathsf{Pf}}(\theta_d \preceq \vartheta_i)) \otimes {}^{\mathsf{U}}(\overline{\mathsf{Hnd}}\ \vartheta_d))}{\Delta; \Gamma \vdash_{\text{exp}} \texttt{openKey}\ e_k\ e_b : {}^{\mathsf{L}}(\overline{\mathsf{RGN}}\ \theta_o\ {}^{\mathsf{L}}({}^{\mathsf{L}}(\overline{\mathsf{Key}}\ \theta_d) \otimes \tau))}$$

$$\frac{\Delta \vdash_{\text{qual}} q \qquad \Delta \vdash \Gamma \preceq q \qquad \Delta, \vartheta; \Gamma \vdash_{\text{exp}} e : \tau}{\Delta; \Gamma \vdash_{\text{exp}} {}^{q}\Lambda\vartheta.\, e : {}^{q}(\overline{\forall}\vartheta.\, \tau)} \qquad \frac{\Delta; \Gamma \vdash_{\text{exp}} e_f : {}^{q}(\overline{\forall}\vartheta.\, \tau) \qquad \Delta \vdash_{\text{index}} \theta_a}{\Delta; \Gamma \vdash_{\text{exp}} e_f\ [\theta_a] : \tau[\theta_a/\vartheta]}$$

$$\frac{\Delta \vdash_{\text{qual}} q \qquad \Delta; \Gamma \vdash_{\text{exp}} e_2 : \tau[\theta_1/\vartheta] \qquad \Delta \vdash \tau[\theta_1/\vartheta] \preceq q}{\Delta; \Gamma \vdash_{\text{exp}} {}^{q}\texttt{pack}(\theta_1, e_2) : {}^{q}(\overline{\exists}\vartheta.\, \tau)}$$

$$\frac{\Delta \vdash \Gamma \rightsquigarrow \Gamma_a \boxdot \Gamma_b \qquad \Delta; \Gamma_a \vdash_{\text{exp}} e_a : {}^{q}(\overline{\exists}\vartheta.\, \tau_x) \qquad \Delta \vdash_{\text{type}} \tau \qquad \Delta, \vartheta; \Gamma_b, x{:}\tau_x \vdash e_b : \tau}{\Delta; \Gamma \vdash_{\text{exp}} \texttt{let pack}(\overline{\vartheta}, x) = e_a\ \texttt{in}\ e_b : \tau}$$

Figure 5.18: Static semantics of Cyc (expressions (IV))

The most interesting rule is the rule for `openKey`:

$$\frac{\begin{array}{c}\Delta \vdash \Gamma \rightsquigarrow \Gamma_k \boxdot \Gamma_b \qquad \Delta; \Gamma_k \vdash_{\text{exp}} e_k : {}^{\mathsf{L}}(\overline{\mathsf{Key}}\ \theta) \\ \Delta; \Gamma_b \vdash_{\text{exp}} e_b : {}^{\mathsf{L}}(\overline{\forall}\vartheta_i.\, {}^{\mathsf{L}}(arg \multimap {}^{\mathsf{L}}(\overline{\mathsf{RGN}}\ \vartheta_i\ \tau))) \\ arg \equiv {}^{\mathsf{L}}({}^{\mathsf{U}}(\overline{\mathsf{Pf}}(\theta_o \preceq \vartheta_i)) \otimes {}^{\mathsf{U}}(\overline{\mathsf{Pf}}(\theta_d \preceq \vartheta_i)) \otimes {}^{\mathsf{U}}(\overline{\mathsf{Hnd}}\ \vartheta_d))\end{array}}{\Delta; \Gamma \vdash_{\text{exp}} \texttt{openKey}\ e_k\ e_b : {}^{\mathsf{L}}(\overline{\mathsf{RGN}}\ \theta_o\ {}^{\mathsf{L}}({}^{\mathsf{L}}(\overline{\mathsf{Key}}\ \theta_d) \otimes \tau))}$$

Here, the "inner" stack ϑ_i corresponds to the stack which extends the "outer" stack θ_o with the stack of the dynamic region θ_d. The relationships between these stacks is witnessed by $\overline{\mathsf{Pf}}(\theta_o \preceq \theta_i)$ and $\overline{\mathsf{Pf}}(\theta_d \preceq \theta_i)$, which assert that the "inner" stack outlives both of the other stacks. Note that there is no relationship between θ_o and θ_d, since the dynamic region might be destroyed before or after the regions in θ_o. Exactly the same trick as was used in $\mathsf{F}^{\mathsf{RGN}}$ ensures that the index variable ϑ_i does not appear in the type τ, thereby ensuring that the result of the computation described by e_b does not depend upon the particular stack used to instantiate ϑ_i. In particular, the result will not leak any means of accessing the dynamic region (although it may contain references allocated in the dynamic region), thereby ensuring that the dynamic region may be destroyed at any point where it is not opened. Finally, note that the dynamic-region key is returned through the result of the `openKey` computation, making it available either to open the dynamic region again or to destroy the dynamic region.

A whole Cyclone program, with a global heap region may be expressed as:

$$\texttt{runRGN}\ ({}^{\mathsf{L}}(\Lambda\vartheta.\, \texttt{letRGN}\ ({}^{\mathsf{L}}(\Lambda\vartheta_{\mathcal{H}}.\, {}^{\mathsf{L}}(\lambda arg.\, \texttt{let}\ \langle \_, hnd_{\mathcal{H}} \rangle = arg\ \texttt{in}\ e)))))$$

where $\vartheta_{\mathcal{H}}$ approximates the Cyclone region `'H` and $hnd_{\mathcal{H}}$ approximates the Cyclone global variable `heap_region`.

5.5.3 Translation: From Cyc to rgnURAL

While Cyc appears to concisely capture many of the high-level features of Cyclone, it is by no means clear that there is a simple argument to directly prove the type soundness of the Cyc language. The combination of monadic and substructural elements in the type system would seem to suggest a rather complicated operational semantics along with an elaborate static semantics for the abstract machine configurations.

Instead, we will sketch a translation from Cyc to rgnURAL; hence, we will *define* the operational behavior of Cyc by its translation into rgnURAL. As we have already established the soundness of rgnURAL, a type-preserving translation will imply the soundness of Cyc. Since we will require no extensions to the rgnURAL language, this translation helps establish rgnURAL as a core language that captures many features in a unified model.

Before turning to the translation from Cyc to rgnURAL, we first demonstrate that lexical regions (introduced by `letRGN`) may be eliminated in favor of dynamic regions alone. In order to make the translation easier to read, we recall the `bindRGN` notation:

$$\texttt{bindRGN}\ x{:}\tau_a \Leftarrow e_a\ ;\ e_b \equiv \texttt{thenRGN}\ e_a\ {}^{\mathsf{L}}(\lambda x{:}\tau_a.\, e_b)$$

with the derived typing rule:

$$\frac{\Delta \vdash_{\text{exp}} \Gamma \rightsquigarrow \Gamma_a \boxdot \Gamma_b \qquad \Delta;\Gamma_a \vdash_{\text{exp}} e_a : {}^{\mathsf{L}}(\overline{\mathsf{RGN}}\ \theta\ \tau_a) \qquad \Delta;\Gamma_b, x{:}\tau_a \vdash_{\text{exp}} e_b : {}^{\mathsf{L}}(\overline{\mathsf{RGN}}\ \theta\ \tau_b)}{\Delta;\Gamma \vdash_{\text{exp}} \texttt{bindRGN}\ x{:}\tau_a \Leftarrow e_a\ ;\ e_b : {}^{\mathsf{L}}(\overline{\mathsf{RGN}}\ \theta\ \tau_b)}$$

Figure 5.19 shows how `letRGN` maybe implemented using dynamic regions. The translation is fairly straight forward: the creation and destruction of a new

$$
\begin{array}{l}
\texttt{letRGN}\ e\ \equiv \\
\quad \texttt{let pack}(\vartheta_d, key_d) = \texttt{newKey in} \\
\quad \texttt{bindRGN}\ kr{:}\mathsf{L}(\mathsf{A}(\overline{\mathsf{Key}}\ \vartheta_d) \otimes \tau) \\
\qquad\qquad \Leftarrow \texttt{openKey}\ key_d\ \mathsf{L}(\Lambda\vartheta_i.\,\mathsf{L}(\lambda arg. \\
\qquad\qquad\qquad\qquad \texttt{let}\ \langle pf_{o\preceq i}, pf_{d\preceq i}, hnd_d\rangle = arg\ \texttt{in} \\
\qquad\qquad\qquad\qquad e\ [\vartheta_i]\ \mathsf{L}\langle pf_{o\preceq i}, \texttt{coerceHnd}\ pf_{d\preceq i}\ hnd_d\rangle))\ ; \\
\quad \texttt{let}\ \langle res, key_d\rangle = kr\ \texttt{in} \\
\quad \texttt{let}\ \langle\rangle = \texttt{freeKey}\ key_d\ \texttt{in} \\
\quad \texttt{returnRGN}\ res
\end{array}
$$

Figure 5.19: Translation from Cyc to Cyc (`letRGN`)

dynamic region brackets the execution of the body of the `letRGN`. We coerce the handle for the dynamic region from the type $\mathsf{U}(\overline{\mathsf{Hnd}}\ \vartheta_d)$ to the type $\mathsf{U}(\overline{\mathsf{Hnd}}\ \vartheta_i)$, thereby making it available to the body of the `letRGN` at the appropriate type.

We next turn our attention to the translation of the remaining Cyc constructs to rgnURAL. In spirit, the translation is very similar to that of Section 4.3, as extended by Section 5.3.3. In particular, we translate $\overline{\mathsf{RGN}}\ \theta\ \tau$ to a stack passing interpretation of computations; $\overline{\mathsf{Pf}}(\theta_1 \preceq \theta_2)$ to an isomorphism between $\mathbb{T}[\![\theta_2]\!]$ and $\mathsf{L}(\mathbb{T}[\![\theta_1]\!] \otimes \beta)$, for some "slack" β; and $\overline{\mathsf{Ref}}\ \theta\ \tau$ and $\overline{\mathsf{Hnd}}\ \theta\ \tau$ to an existentially bound region along with a reference or handle and an isomorphism between $\mathbb{T}[\![\theta]\!]$ and $\mathsf{L}(\beta \otimes \mathsf{L}(\overline{\mathsf{Cap}}\ \varrho))$. Finally, a $\overline{\mathsf{Key}}\ \theta$ is translated to a stack (of type $\mathbb{T}[\![\theta]\!]$) along with a handle and a "clean-up" function (of type $\mathsf{U}(\mathbb{T}[\![\theta]\!] \multimap \mathsf{L}\mathbf{1}_{\otimes})$), to be invoked when the dynamic region is to be destroyed. Figure 5.20 summarizes the translation of Cyc types to rgnURAL types.

Translations yielding types

Types

$$\begin{aligned}
\mathbb{T}[\![\alpha]\!] &= \alpha \\
&\vdots \quad \vdots \quad \vdots \\
\mathbb{T}[\![^{q}(\overline{\mathsf{RGN}}\ \theta\ \tau)]\!] &= {}^{q}(\mathbb{T}[\![\theta]\!] \multimap {}^{\mathsf{L}}(\mathbb{T}[\![\theta]\!] \otimes \mathbb{T}[\![\tau]\!])) \\
\mathbb{T}[\![^{q}(\overline{\mathsf{Pf}}(\theta_1 \preceq \theta_2))]\!] &= {}^{q}(\overline{\exists}\beta.\, \mathbf{Iso}(\mathbb{T}[\![\theta_2]\!], {}^{\mathsf{L}}(\mathbb{T}[\![\theta_1]\!] \otimes \beta))) \\
\mathbb{T}[\![^{q}(\overline{\mathsf{Ref}}\ \theta\ \tau)]\!] &= \\
& {}^{q}(\overline{\exists}\varrho_r.\, {}^{q}({}^{\mathsf{U}}(\overline{\exists}\beta.\, \mathbf{Iso}(\mathbb{T}[\![\theta]\!], {}^{\mathsf{L}}(\beta \otimes {}^{\mathsf{L}}(\overline{\mathsf{Cap}}\ \varrho_r))))) \otimes {}^{q}(\overline{\mathsf{Ref}}\ \varrho_r\ \mathbb{T}[\![\tau]\!]))) \\
\mathbb{T}[\![^{q}(\overline{\mathsf{Hnd}}\ \theta)]\!] &= \\
& {}^{q}(\overline{\exists}\varrho_h.\, {}^{q}({}^{\mathsf{U}}(\overline{\exists}\beta.\, \mathbf{Iso}(\mathbb{T}[\![\theta]\!], {}^{\mathsf{L}}(\beta \otimes {}^{\mathsf{L}}(\overline{\mathsf{Cap}}\ \varrho_h))))) \otimes {}^{q}(\overline{\mathsf{Hnd}}\ \varrho_h))) \\
\mathbb{T}[\![^{q}(\overline{\mathsf{Key}}\ \theta)]\!] &= {}^{q}(\mathbb{T}[\![\theta]\!] \otimes \mathbb{T}[\![{}^{\mathsf{U}}(\overline{\mathsf{Hnd}}\ \theta)]\!] \otimes {}^{\mathsf{U}}(\mathbb{T}[\![\theta]\!] \multimap {}^{\mathsf{L}}\mathbf{1}_{\otimes})) \\
\mathbb{T}[\![^{q}(\overline{\forall}\vartheta.\, \tau)]\!] &= {}^{q}(\overline{\forall}\alpha_{\vartheta}.\, \mathbb{T}[\![\tau]\!]) \\
\mathbb{T}[\![^{q}(\overline{\exists}\vartheta.\, \tau)]\!] &= {}^{q}(\overline{\exists}\alpha_{\vartheta}.\, \mathbb{T}[\![\tau]\!])
\end{aligned}$$

Indices

$$\mathbb{T}[\![\vartheta]\!] = \alpha_{\vartheta}$$

Isomorphism Macro

$$\mathbf{Iso}(\tau_1, \tau_2) = {}^{\mathsf{U}}({}^{\mathsf{U}}(\tau_1 \multimap \tau_2) \otimes {}^{\mathsf{U}}(\tau_2 \multimap \tau_1))$$

Figure 5.20: Translation from $\mathsf{F}^{\mathsf{RGN}}$ to Cyc (I)

Translations yielding expressions

Expressions

$$\mathbb{E}\left[\!\!\left[\frac{}{\Delta;\{\}\vdash_{\text{exp}} \texttt{newKey} : {}^{\mathsf{L}}(\bar{\exists}\vartheta.\,{}^{\mathsf{L}}(\overline{\mathsf{Key}}\ \vartheta))}\right]\!\!\right] =$$

$$\begin{array}{l}
\texttt{let pack}(\varrho_d, \langle cap_d, hnd_d\rangle) = {}^{\mathsf{L,U}}\texttt{newrgn in} \\
\texttt{let } stk_d = {}^{\mathsf{L}}\langle{}^{\mathsf{L}}\langle\rangle, cap_d\rangle \texttt{ in} \\
\texttt{let } iso_d = \mathbf{IsoPairId}({}^{\mathsf{L}}\mathbf{1}_{\otimes}, {}^{\mathsf{L}}(\overline{\mathsf{Cap}}\ \varrho_d)) \texttt{ in} \\
\texttt{let } phnd_d = {}^{\mathsf{U}}\texttt{pack}(\varrho_d, {}^{\mathsf{U}}\langle{}^{\mathsf{U}}\texttt{pack}({}^{\mathsf{L}}\mathbf{1}_{\otimes}, iso_d), hnd_d\rangle) \texttt{ in} \\
\texttt{let } des_d = {}^{\mathsf{U}}(\lambda stk_d{:}{}^{\mathsf{L}}({}^{\mathsf{L}}\mathbf{1}_{\otimes} \otimes {}^{\mathsf{L}}(\overline{\mathsf{Cap}}\ \varrho_d)). \\
\qquad\qquad \texttt{let } \langle\langle\rangle, cap_d\rangle = stk_d \texttt{ in} \\
\qquad\qquad \texttt{freergn } cap_d\ hnd_d) \quad \texttt{in} \\
\texttt{let } key_d = {}^{\mathsf{L}}\langle stk_d, phnd_d, des_d\rangle \texttt{ in} \\
{}^{\mathsf{L}}\texttt{pack}({}^{\mathsf{L}}({}^{\mathsf{L}}\mathbf{1}_{\otimes} \otimes {}^{\mathsf{L}}(\overline{\mathsf{Cap}}\ \varrho_d)), key_d)
\end{array}$$

$$\mathbb{E}\left[\!\!\left[\frac{\Delta;\Gamma\vdash_{\text{exp}} e_k : {}^{\mathsf{L}}(\overline{\mathsf{Key}}\ \theta)}{\Delta;\Gamma\vdash_{\text{exp}} \texttt{freeKey } e_k : {}^{\mathsf{L}}\mathbf{1}_{\otimes}}\right]\!\!\right] = \begin{array}{l}
\texttt{let } key_d = \mathbb{E}[\![e]\!] \texttt{ in} \\
\texttt{let } \langle stk_d, phnd_d, des_d\rangle = key_d \texttt{ in} \\
des_d\ stk_d
\end{array}$$

Figure 5.21: Translation from $\mathsf{F}^{\mathsf{RGN}}$ to Cyc (II)

The translation of the purely functional Cyc terms to rgnURAL terms is very nearly an identity translation, and we elide them in this presentation. For the region and reference primitives adopted from $\mathsf{F}^{\mathsf{RGN}}$, the translation from Cyc to rgnURAL is very similar to the translation given in Section 4.3 (as extended in Section 5.3.3), and we elide these translations as well. The dynamic region primitives have the most interesting translations from Cyc to rgnURAL, and are given in Figures 5.21 and 5.22

Translations yielding expressions

Expressions

$$\mathbb{E}\left[\!\!\left[\frac{\begin{array}{c}\Delta \vdash \Gamma \rightsquigarrow \Gamma_k \boxdot \Gamma_b \qquad \Delta; \Gamma_k \vdash_{\text{exp}} e_k : {}^{\mathsf{L}}(\overline{\mathsf{Key}}\ \theta) \\ \Delta; \Gamma_b \vdash_{\text{exp}} e_b : {}^{\mathsf{L}}(\overline{\forall}\vartheta_i.\, {}^{\mathsf{L}}(arg \multimap {}^{\mathsf{L}}(\overline{\mathsf{RGN}}\ \vartheta_i\ \tau))) \\ arg \equiv {}^{\mathsf{L}}({}^{\mathsf{U}}(\overline{\mathsf{Pf}}(\theta_o \preceq \vartheta_i)) \otimes {}^{\mathsf{U}}(\overline{\mathsf{Pf}}(\theta_d \preceq \vartheta_i)) \otimes {}^{\mathsf{U}}(\overline{\mathsf{Hnd}}\ \vartheta_d))\end{array}}{\Delta; \Gamma \vdash_{\text{exp}} \texttt{openKey}\ e_k\ e_b : {}^{\mathsf{L}}(\overline{\mathsf{RGN}}\ \theta_o\ {}^{\mathsf{L}}({}^{\mathsf{L}}(\overline{\mathsf{Key}}\ \theta_d) \otimes \tau))}\right]\!\!\right] =$$

$\texttt{let}\ key_d{:}\mathbb{T}[\![{}^{\mathsf{L}}(\overline{\mathsf{Key}}\ \theta_d)]\!] = \mathbb{E}[\![e_k]\!]\ \texttt{in}$

$\texttt{let}\ f_b{:}\mathbb{T}[\![{}^{\mathsf{L}}(\overline{\forall}\vartheta_i.\, {}^{\mathsf{L}}(arg \multimap {}^{\mathsf{L}}(\overline{\mathsf{RGN}}\ \vartheta_i\ \tau)))]\!] = \mathbb{E}[\![e_b]\!]\ \texttt{in}$

${}^{\mathsf{L}}(\lambda stk_o{:}\mathbb{T}[\![\theta_o]\!]\,.\,\texttt{let}\ \langle stk_d, phnd_d, des_d\rangle = key_d\ \texttt{in}$

$\texttt{let}\ ppf_o = {}^{\mathsf{U}}\texttt{pack}(\mathbb{T}[\![\theta_d]\!], \mathbf{IsoPairId}(\mathbb{T}[\![\theta_o]\!], \mathbb{T}[\![\theta_d]\!]))\ \texttt{in}$

$\texttt{let}\ ppf_d = {}^{\mathsf{U}}\texttt{pack}(\mathbb{T}[\![\theta_o]\!], \mathbf{IsoPairSwap}(\mathbb{T}[\![\theta_o]\!], \mathbb{T}[\![\theta_d]\!]))\ \texttt{in}$

$\texttt{let}\ arg = {}^{\mathsf{U}}\langle ppf_o, ppf_d, phnd_d\rangle\ \texttt{in}$

$\texttt{let}\ stk_i{:}{}^{\mathsf{L}}(\mathbb{T}[\![\theta_o]\!] \otimes \mathbb{T}[\![\theta_d]\!]) = {}^{\mathsf{L}}\langle stk_o, stk_d\rangle\ \texttt{in}$

$\texttt{let}\ \langle stk_i, res\rangle = f_b\ [{}^{\mathsf{L}}(\mathbb{T}[\![\theta_o]\!] \otimes \mathbb{T}[\![\theta_d]\!])]\ arg\ stk_i\ \texttt{in}$

$\texttt{let}\ \langle stk_o, stk_d\rangle = stk_i\ \texttt{in}$

$\texttt{let}\ key_d = {}^{\mathsf{L}}\langle stk_d, phnd_d, des_d\rangle\ \texttt{in}$

${}^{\mathsf{L}}\langle stk_o, {}^{\mathsf{L}}\langle key_d, res\rangle\rangle)$

Figure 5.22: Translation from $\mathsf{F}^{\mathsf{RGN}}$ to Cyc (III)

Unsurprisingly, the translation of `newKey` uses `newrgn` to create a region. Surprisingly, the translation of `freeKey` does not (directly) use `freergn` to destroy the region. Rather, the translation of `newKey` constructs a function to destroy the region, when applied to the representation of the dynamic-region stack. The translation of `freeKey` needs only apply this function to the representation of the dynamic-region stack in order to destroy the region.

This translation ensures that only the translation of `newKey` needs to know the representation of the dynamic-region stack (stk_d). The macro **IsoPairId**(τ_1, τ_2) abbreviates a the isomorphism between the type $\llcorner(\tau_1 \otimes \tau_2)$ and the type $\llcorner(\tau_1 \otimes \tau_2)$. The isomorphism witnesses the embedding of the capability (cap_d) in the stack (stk_d). The translation of `newKey` uses the isomorphism to construct a packed handle for the dynamic region.

The translation of `openKey` is not significantly more complicated than the translation of `letRGN` in Section 4.3. The "inner" stack stk_i is formed by pairing the "outer" stack stk_o with the dynamic-region stack stk_d. The isomorphisms witnessing the embedding of the "outer" stack in the "inner" stack and the dynamic-region stack in the "inner" stack are trivial. The macro **IsoPairSwap**(τ_1, τ_2) abbreviates an isomorphism between the type $\llcorner(\tau_1 \otimes \tau_2)$ and the type $\llcorner(\tau_2 \otimes \tau_1)$. After running the "inner" computation, the dynamic-region key is reconstructed and paired with the result of the "inner" computation as the result of the "outer" computation.

5.5.4 Fused Regions

One of the advantages of translating Cyc to rgnURAL is that it exhibits opportunities for new high-level features that emerge naturally from the lower-level core language. In this section, we explore one such opportunity.

We first recall the translation for the type of a Cyc dynamic-region key:

$$\mathbb{T}[\![{}^{q}(\overline{\mathsf{Key}}\ \theta)]\!] \quad = \quad {}^{q}(\mathbb{T}[\![\theta]\!] \otimes \mathbb{T}[\![{}^{\mathsf{U}}(\overline{\mathsf{Hnd}}\ \theta)]\!] \otimes {}^{\mathsf{U}}(\mathbb{T}[\![\theta]\!] \multimap {}^{\mathsf{L}}\mathbf{1}_{\otimes}))$$

Note that this translation includes a "thunk" that remembers how destroy the dynamic region when `freeKey` is applied to the key. It should be clear that we can make a "bigger" key out of two keys by composing their thunks. For example, given key_1 of type $\mathbb{T}[\![{}^{\mathsf{L}}(\overline{\mathsf{Key}}\ \theta_1)]\!]$ and key_2 of type $\mathbb{T}[\![{}^{\mathsf{L}}(\overline{\mathsf{Key}}\ \theta_2)]\!]$, we can construct a new key as follows:

$$\begin{array}{l}
\texttt{let}\ \langle stk_1{:}\mathbb{T}[\![\theta_1]\!], phnd_1, des_1\rangle = key_1\ \texttt{in} \\
\texttt{let}\ \langle stk_2{:}\mathbb{T}[\![\theta_2]\!], phnd_2, des_2\rangle = key_2\ \texttt{in} \\
\texttt{let}\ stk = {}^{\mathsf{L}}\langle stk_1, stk_2\rangle\ \texttt{in} \\
\texttt{let}\ des = {}^{\mathsf{U}}(\lambda stk{:}{}^{\mathsf{L}}(\tau_1 \otimes \tau_2).\ \texttt{let}\ \langle stk_1, stk_2\rangle = stk\ \texttt{in} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \texttt{let}\ \langle\rangle = des_1\ stk_1\ \texttt{in} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \texttt{let}\ \langle\rangle = des_2\ stk_2\ \texttt{in} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad {}^{\mathsf{L}}\langle\rangle)\quad \texttt{in} \\
\texttt{let}\ iso_{1\preceq} = \mathbf{IsoPairId}(\mathbb{T}[\![\theta_1]\!], \mathbb{T}[\![\theta_2]\!])\ \texttt{in} \\
\texttt{let}\ iso_{2\preceq} = \mathbf{IsoPairSwap}(\mathbb{T}[\![\theta_1]\!], \mathbb{T}[\![\theta_2]\!])\ \texttt{in} \\
\texttt{let}\ \texttt{pack}(\varrho_1, \langle iso_1, hnd_1\rangle) = phnd_1\ \texttt{in} \\
\texttt{let}\ \texttt{pack}(\varrho_2, \langle iso_2, hnd_2\rangle) = phnd_2\ \texttt{in} \\
\texttt{let}\ phnd = \ldots\ \texttt{in} \\
\texttt{let}\ key = {}^{\mathsf{L}}\langle stk, phnd, des\rangle\ \texttt{in} \\
key
\end{array}$$

where the new packed handle $phnd$ can be constructed either from hnd_1 (by composing iso_1 with $iso_{1\prec}$ in the appropriate manner) or from hnd_2 (by composing iso_2 with $iso_{2\prec}$).

Having combined the "thunks" which destroy the dynamic regions, we know that the two dynamic regions will be destroyed at the same time. Therefore, in some sense, the liveness of one region implies the liveness of the other region, and vice-versa. In fact, we can make this notion more concrete by constructing not just a new key, but also witnesses to the relationship between the first and second stacks (stk_1 and stk_2) and the new stack (stk). We change the expression above to include:

$$\begin{array}{l}
\texttt{let } key = {}^{\mathsf{L}}\langle stk, phnd, des\rangle \texttt{ in} \\
\texttt{let } ppf_1 = {}^{\mathsf{U}}(\texttt{pack}(\mathbb{T}[\![\theta_2]\!], iso_{1\preceq})) \texttt{ in} \\
\texttt{let } ppf_2 = {}^{\mathsf{U}}(\texttt{pack}(\mathbb{T}[\![\theta_1]\!], iso_{2\preceq})) \texttt{ in} \\
{}^{\mathsf{L}}(\texttt{pack}({}^{\mathsf{L}}(\mathbb{T}[\![\theta_1]\!] \otimes \mathbb{T}[\![\theta_2]\!]), {}^{\mathsf{L}}\langle key, ppf_1, ppf_2\rangle))
\end{array}$$

This expression now has a type equivalent to:

$$\mathbb{T}[\![{}^{\mathsf{L}}(\overline{\exists}\vartheta.\, {}^{\mathsf{L}}({}^{\mathsf{L}}(\overline{\mathsf{Key}}\ \vartheta) \otimes \overline{\mathsf{Pf}}(\theta_1 \preceq \vartheta) \otimes \overline{\mathsf{Pf}}(\theta_2 \preceq \vartheta)))]\!]$$

Hence, we could take the expression above as the translation of a new Cyc term (`fuseKeys`) with the following typing rule:

$$\frac{\Delta \vdash \Gamma \rightsquigarrow \Gamma_1 \boxdot \Gamma_2 \qquad \Delta; \Gamma_1 \vdash_{\text{exp}} e_1 : {}^{\mathsf{L}}(\overline{\mathsf{Key}}\ \theta_1) \qquad \Delta; \Gamma_1 \vdash_{\text{exp}} e_2 : {}^{\mathsf{L}}(\overline{\mathsf{Key}}\ \theta_2)}{\Delta; \Gamma \vdash_{\text{exp}} \texttt{fuseKeys}\ e_1\ e_2 : {}^{\mathsf{L}}(\overline{\exists}\vartheta.\, {}^{\mathsf{L}}({}^{\mathsf{L}}(\overline{\mathsf{Key}}\ \vartheta) \otimes \overline{\mathsf{Pf}}(\theta_1 \preceq \vartheta) \otimes \overline{\mathsf{Pf}}(\theta_2 \preceq \vartheta)))}$$

Note that the result includes a "new" existentially bound index variable, along with proofs that allow references in either of the old dynamic regions to be coerced (via `coerceRef`) into the "new" dynamic region.

There are a some scenarios where the operational behavior of `fuseKeys` could be useful. For example, consider a routine that constructs a rooted data structure and, in many instances, the data structure is not needed in the future, but occasionally

it is necessary to use the structure in the future.[4] Since the structure might be needed in the future, the whole structure needs to be allocated in a region that outlives the current routine, but that wastes space in the cases where the structure is no longer needed. Using `fuseKeys`, we may instead allocate the structure in a dynamic region; when the structure is not needed, the dynamic region may be destroyed; when the structure is needed, the dynamic region may be fused with another dynamic region (which accumulates all of the dynamic regions for all of the needed data structures).

5.6 A Type-Safe Copying Garbage Collector

In this section, we consider an advanced application of region-based memory management: expressing a type-safe copying garbage collector. We are motivated to consider this application by the recognition that although many high-level, safe languages provide automatic memory management through a garbage collector, the interpreters and runtime systems for these high-level, safe languages are often written in low-level, unsafe languages. Providing interpreters and runtime systems are a common way of hosting applications on a dynamic platform (such as a web server). Yet, making use of a low-level, unsafe language to implement the interpreters and runtime systems raises the concern that a bug in the interpreter or runtime system may introduce a security hole that compromises the integrity of the entire platform. Hence, reducing or eliminating code written in low-level, unsafe languages from the implementation increases our confidence in the security of the platform as a whole.

[4]This kind of scenario often arises in assertion checking and theorem proving applications, where the rooted data structure corresponds to a logical formula.

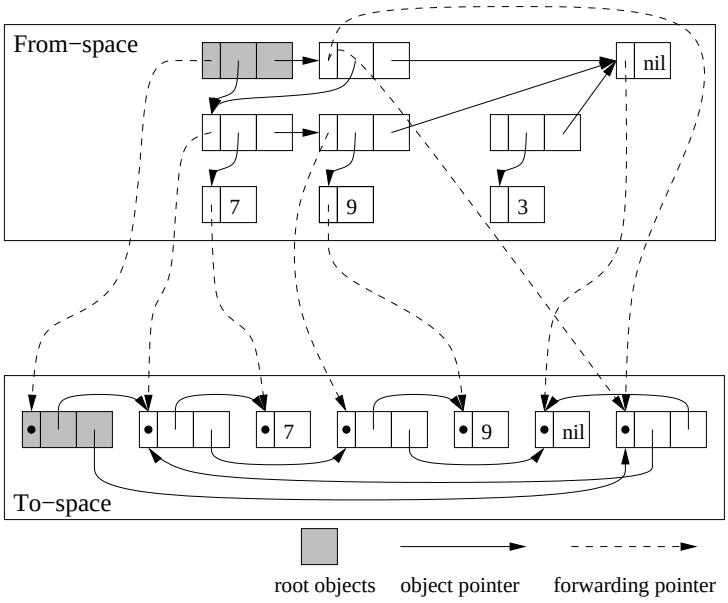

Figure 5.23: Copying garbage collector example

Implementing the core of an interpreter in a safe language is not itself a significant challenge. Rather, implementing the runtime system for the interpreter that provides automatic memory management is a challenge. Nonetheless, we sketch an implementation of a type-safe copying garbage collector, which uses region-based memory management to reclaim memory. Using region-based memory management to implement a type-safe garbage collector is not itself novel [93, 64, 38], but we believe that our implementation gives rise to a more natural typing for forwarding pointers.

Figure 5.23 illustrates a simple copying garbage collector collector. We assume that the interpreted program has been running, allocating objects in memory. At some point in time, the runtime system stops the interpreted program and starts the garbage collector. The collector begins with with a group of objects in the *From-space* and an empty *To-space*. The collector traverses the live objects (those objects reachable from the root objects) in From-space, copying each live object

into To-space when the object is first visited. The collector preserves sharing among objects by leaving a forwarding pointer in each From-space object when it is copied. The forwarding pointer points to the copied object in To-space. Whenever an object in From-space is visited, the copying function first examines the forwarding pointer. If it is non-NULL, then the copy function returns the forwarding address. Otherwise, space for the object is reserved in To-space, the forwarding pointer is set to the address of the reserved space, and the fields of the object are copied. After all live objects in From-space have been copied (depicted in Figure 5.23), the collector can free all memory in From-space and the program can continue execution, allocating new objects in To-space.

In the copying algorithm, the separation of managed memory into a From-space and a To-space suggests a natural correspondence with regions. Clearly, the LIFO discipline of lexical regions is insufficient for a copying garbage collector, since the lifetime of From-space should end after the beginning but before the end of To-space's lifetime. Hence, we turn our attention to $\mathsf{rgnURAL}$ with non-LIFO regions, where it appears that we have sufficient expressiveness to write a simple copying garbage collector (Figure 5.24).

In this simple example, the state of the interpreted program is a value of the type:

$$\mathsf{L}(\overline{\exists}\varrho.\,\mathsf{L}(\mathsf{L}(\overline{\mathsf{Cap}}\ \varrho) \otimes \mathsf{L}(\overline{\mathsf{Hnd}}\ \varrho) \otimes \mathsf{Prog}[\varrho]))$$

where the existentially bound region name ϱ fixes the region for the program's garbage collected memory, the capability $\mathsf{L}(\overline{\mathsf{Cap}}\ \varrho)$ and handle $\mathsf{L}(\overline{\mathsf{Hnd}}\ \varrho)$ provide access to the region, and the unspecified type $\mathsf{Prog}[\varrho]$ describes the evaluation of the program. The notation $[\varrho]$ indicates that $\mathsf{Prog}[\varrho]$ is a type parameterized by a region, in this case instantiated to the region ϱ. We intend that a value of type

$$
\begin{array}{l}
\texttt{let } doGC = \\
\quad {}^{\mathsf{U}}(\lambda state_{\text{old}}{:}{}^{\mathsf{L}}(\overline{\exists}\varrho.\, {}^{\mathsf{L}}({}^{\mathsf{L}}(\overline{\mathsf{Cap}}\ \varrho) \otimes {}^{\mathsf{L}}(\overline{\mathsf{Hnd}}\ \varrho) \otimes \mathsf{Prog}[\varrho])). \\
\qquad \textit{// unpack the old program state} \\
\qquad \texttt{let pack}(\varrho_f, \langle cap_f, hnd_f, prog_f\rangle) = state_{\text{old}}\ \texttt{in} \\
\qquad \textit{// create the to-space region} \\
\qquad \texttt{let pack}(\varrho_t, \langle cap_t, hnd_t\rangle) = {}^{\mathsf{L,U}}\texttt{newrgn in} \\
\qquad \textit{// copy the program} \\
\qquad \texttt{let } \langle cap_f, cap_t, prog_t\rangle = copyProg\ [\varrho_f, \varrho_t]\ {}^{\mathsf{L}}\langle cap_f, cap_t, hnd_t, prog_f\rangle\ \texttt{in} \\
\qquad \textit{// destroy the from-space region} \\
\qquad \texttt{let } \langle\rangle = \texttt{freergn}\ {}^{\mathsf{L}}\langle cap_f, hnd_f\rangle\ \texttt{in} \\
\qquad \textit{// package the new program state} \\
\qquad \texttt{let } state_{\text{new}} = {}^{\mathsf{L}}\texttt{pack}(\varrho_t, {}^{\mathsf{L}}\langle cap_t, hnd_t, prog_t\rangle)\ \texttt{in} \\
\qquad state_{\text{new}})
\end{array}
$$

Figure 5.24: Simple copying garbage collector in rgnURAL

$\mathsf{Prog}[\varrho]$ provides sufficient information for the garbage collector to identify the root objects in the program's memory.

The implementation of this simple garbage collector is relatively straight forward. The state of the interpreted program is unpacked, naming the From-space region ϱ_f. A new To-space is created with `newrgn`; the result is unpacked naming the To-space region ϱ_t. The program is copied using the *copyProg* function, which has the type:

$$^{\mathsf{U}}(\overline{\forall}\varrho_f.\,^{\mathsf{U}}(\overline{\forall}\varrho_t.\,^{\mathsf{U}}(^{\mathsf{L}}(^{\mathsf{L}}(\overline{\mathsf{Cap}}\ \rho_f) \otimes {}^{\mathsf{L}}(\overline{\mathsf{Cap}}\ \rho_t) \otimes {}^{\mathsf{L}}(\overline{\mathsf{Hnd}}\ \rho_f) \otimes \mathsf{Prog}[\varrho_f]) \multimap \\ {}^{\mathsf{L}}(^{\mathsf{L}}(\overline{\mathsf{Cap}}\ \rho_f) \otimes {}^{\mathsf{L}}(\overline{\mathsf{Cap}}\ \rho_t) \otimes \mathsf{Prog}[\varrho_t])))$$

Note that this function takes the capabilities for both From- and To-space and the handle for To-space, since it will need to read the program's old memory in From-space and write the program's new memory in To-space. The function returns the capabilities along with a new program (of type $\mathsf{Prog}[\varrho_t]$), which has root objects in To-space. Next, the From-space is destroyed, reclaiming memory that will no longer be accessed by the interpreted program. Finally, the new program is packed along with the capability and handle for To-space, yielding a new state of the interpreted program.

While this captures the spirit of the garbage collector, many details remain. In particular, we need to consider the representation of objects in the program's memory and the representation of forwarding pointers, used by the *copyProg* function to implement the copying of objects from From-space to To-space. A simple representation of objects pairs a forwarding pointer with some value corresponding to heap allocated data manipulated by the program (e.g., integer constants, pairs, etc.). The simple extension of rgnURAL with sum pre-types ($\tau_1 \oplus \tau_2$) and recursive

pre-types $\overline{\mu\alpha}.\,\tau$ would admit the following definitions:

$$\begin{aligned}
\mathsf{Obj}[\varrho] &\equiv {}^{\mathsf{U}}(?\mathsf{FwdPtr}? \otimes \mathsf{Val}[\varrho]) \\
\mathsf{Val}[\varrho] &\equiv {}^{\mathsf{U}}(\mathsf{Const} \oplus \mathsf{Pair}[\varrho] \oplus \cdots) \\
\mathsf{Const} &\equiv {}^{\mathsf{U}}\overline{\mathsf{Int}} \\
\mathsf{Pair}[\varrho] &\equiv {}^{\mathsf{U}}(\mathsf{ObjRef}[\varrho] \otimes \mathsf{ObjRef}[\varrho]) \\
\mathsf{ObjRef}[\varrho] &\equiv {}^{\mathsf{U}}(\overline{\mathsf{Ref}}\ \varrho\ \mathsf{Obj}[\varrho])
\end{aligned}$$

Note that a $\mathsf{Pair}[\varrho]$ value combines two object references ($\mathsf{ObjRef}[\varrho]$), thereby allowing both cycles and sharing in heap data manipulated by the program; these cycles and sharing should be preserved by the garbage collector.

We have yet to answer the question: "what is the type of the forwarding pointer?" The answer is (something along the lines of): "a pointer to an object in To-space, whose forwarding pointer is a pointer to an object in To-space's To-space, whose forwarding pointer ...." What we require is a name for all of the unwindings of the infinite sequence of pointers. The rgnURAL language (even with recursive types) cannot express this infinite sequence. Hence, we extend rgnURAL to provide such a name in the form of a region constructor, which maps region names to region names and generates an infinite *sequence* of region names:

Region variables

$$\varrho \quad \in \quad \mathit{RVars}$$

Regions

$$\rho \quad ::= \quad \varrho \mid \mathsf{Next}\ \rho$$

Note that although the region names ρ and $\mathsf{Next}\ \rho$ are related, the lifetimes of their corresponding regions are not. In a similar manner, $\mathsf{ObjRef}[\mathsf{Next}\ \varrho]$ will be a well-formed type anywhere in the scope of ϱ, even if the region corresponding to $\mathsf{Next}\ \rho$ has not been created. The original inspiration for the Next region constructor

comes from Hawblitzel *et al.* [38], where *type sequences* (mappings from integers to types) are used to index regions by a region number ("epoch"), yielding the connection between successive regions in a copying collector.

We can now give the type of a forwarding pointer as follows:

$$\begin{aligned}
\mathsf{Obj}[\varrho] &\equiv {}^{\mathsf{U}}(\mathsf{FwdPtr}[\varrho] \otimes \mathsf{Val}[\varrho]) \\
\mathsf{ObjRef}[\varrho] &\equiv {}^{\mathsf{U}}(\overline{\mathsf{Ref}}\ \varrho\ \mathsf{Obj}[\varrho]) \\
\mathsf{FwdPtr}[\varrho] &\equiv {}^{\mathsf{U}}(\overline{\mathsf{Ref}}\ \varrho\ {}^{\mathsf{U}}({}^{\mathsf{U}}\mathbf{1}_{\otimes} \oplus \mathsf{ObjRef}[\mathsf{Next}\ \varrho]))
\end{aligned}$$

Note that a forwarding pointer is a mutable reference containing either a dummy value (of type ${}^{\mathsf{U}}\mathbf{1}_{\otimes}$, indicating that the pointed to object hasn't yet been allocated) or a reference (of type $\mathsf{ObjRef}[\mathsf{Next}\ \varrho]$) to the pointed to object in the next region.

Operationally, we expect regions in a region sequence to behave much like normal regions, with access to a region in a region sequence mediated by a capability. In addition, we will have operations to create new region sequences and to create the next region in a region sequence. Furthermore, the operation to create the next region ($\mathsf{Next}\ \rho$) in a region sequence should produce the capability and handle for the next region ($\overline{\mathsf{Cap}}\ (\mathsf{Next}\ \rho)$ and $\overline{\mathsf{Hnd}}\ (\mathsf{Next}\ \rho)$). While this operation yields an infinite sequence of region capabilities, we need to ensure that the sequence is unique: that there is exactly one way to generate the capability for $\mathsf{Next}\ \rho$ for any ρ. The substructural qualifiers of rgnURAL provide exactly this uniqueness.

Hence, we extend rgnURAL with a type and operations to manage the generation of the sequence of regions:

Pre-types

$$\overline{\tau} \ ::= \ \cdots \mid \overline{\mathsf{Gen}}\ \rho$$

Terms

$$e \ ::= \ \cdots \mid {}^{q_g}\texttt{newrgnseq} \mid {}^{q_c,q_h}\texttt{nextrgn}$$

$$\frac{\Delta \vdash \mathsf{A} \preceq q_g}{\Delta; \{\} \vdash_{\text{exp}} {}^{q_g}\texttt{newrgnseq}^{\mathsf{L}}(\overline{\exists}\varrho.\, {}^{q_g}(\overline{\mathsf{Gen}}\ \varrho))}$$

$$\frac{\Delta; \Gamma \vdash_{\text{exp}} e_g : {}^{q_g}(\overline{\mathsf{Gen}}\ \rho)}{\Delta; \Gamma \vdash_{\text{exp}} {}^{q_c,q_h}\texttt{nextrgn}\ e_g : {}^{\mathsf{L}}({}^{q_c}(\overline{\mathsf{Cap}}\ \rho) \otimes {}^{q_h}(\overline{\mathsf{Hnd}}\ \rho) \otimes {}^{q_g}(\overline{\mathsf{Gen}}\ (\mathsf{Next}\ \rho)))}$$

Figure 5.25: Static semantics for rgnURAL with region sequences

The pre-type $\overline{\mathsf{Gen}}\ \rho$ (which will always be affine or linear) serves as a meta-capability: it is the capability to produce the capability for ρ and the next generator. The expression ${}^{q_g}\texttt{newrgnseq}$ creates a new region sequence (with no regions) by returning the first generator; the expression ${}^{q_c,q_h}\texttt{nextrgn}\ e_g$ creates the next region (capability and handle) in a region sequence when applied to a generator. The typing rules for the new expression forms are given in Figure 5.25.

Both `newrgnseq` and `nextrgn` are closely related to `newrgn`. Like `newrgn`, `newrgnseq` returns an existential package, hiding the name of a region, but rather than returning a capability and handle, it returns a generator. Hence, the existentially bound region name corresponds to the (as of yet uncreated) "first" region in the region sequence. Like `newrgn`, `nextrgn` returns a capability and handle, but rather than existentially hiding the region name, the region name is taken from the input generator. A generator for the "next" region is also returned, giving rise to an infinite sequence of (potential) regions. Because the generator is unique (affine or linear) and `nextrgn` consumes it, it follows that a program can only create one capability for $\mathsf{Next}\ \varrho$; hence, the sequence of regions is unique.

Operationally, we note that the physical region for any given region name need only be created when the capability and handle are created; that is, when `nextrgn`

```
let doGC =
  U(λstate_old:L(∃ϱ. L(L(Gen (Next ϱ)) ⊗ L(Cap ϱ) ⊗ L(Hnd ϱ) ⊗ Prog[ϱ])).
    // unpack the old program state
    let pack(ϱ_f, ⟨gen_f, cap_f, hnd_f, prog_f⟩) = state_old in
    // create the to-space region
    let ⟨gen_t, cap_t, hnd_t⟩) = L,Unextrgn gen_f in
    // copy the program
    let ⟨cap_f, cap_t, prog_t⟩ = copyProg [ϱ_f] L⟨cap_f, cap_t, hnd_t, prog_f⟩ in
    // destroy the from-space region
    let ⟨⟩ = freergn L⟨cap_f, hnd_f⟩ in
    // package the new program state
    let state_new = Lpack(Next ϱ_f, L⟨gen_t, cap_t, hnd_t, prog_t⟩) in
    state_new)
```

Figure 5.26: Simple copying garbage collector in rgnURAL with region sequences

is called. In particular, no regions need to be pre-created. Furthermore, we note that the rgnURAL primitive `freergn` suffices for destroying regions in a region sequence.

Figure 5.26 revises the simple copying garbage collector from Figure 5.24 to use region sequences. The program state is extended with a generator ($^{L}(\overline{\mathsf{Gen}}\ (\mathsf{Next}\ \varrho))$), in order to create the To-space. (Note that *copyProg* needs only ϱ_f, since the name of the To-space region is necessarily $\mathsf{Next}\ \varrho_f$.)

More details on this application, including a full description of an interpreter, runtime system, and copying garbage collector implemented in Cyclone, may be found in the paper *Implementation and Performance Evaluation of a Safe Runtime*

System in Cyclone [22]. Preliminary benchmarks demonstrated that we can indeed build a platform with reasonable performance when compared to other approaches that guarantee safety. More importantly, we could significantly reduce the amount of trusted, unsafe code needed to implement the system. The implementation relies crucially upon both dynamic regions and unique pointers, along with Cyclone analogues of Next, `newrgnseq`, and `nextrgn`.

Chapter 6

Conclusion

The central thesis of this dissertation has been that the type-and-effect systems that have traditionally been used to ensure the safety of region-based memory management are neither the only nor the simplest systems for this purpose. We have proposed that monadic and substructural type systems give rise to simpler, more expressive, and more uniform languages that continue to provide the power and safety of region-based memory management. In order to substantiate this claim, we defined two languages with novel type systems that ensure the safety of region-based memory management:

- the $\mathsf{F}^{\mathsf{RGN}}$ language with a monadic type system;
- the rgnURAL language with a substructural type system.

The first major technical contribution of this work has been the design of these languages and their type systems. The second major technical contribution of this work has been to demonstrate that we have lost no expressive power in adopting the type systems of $\mathsf{F}^{\mathsf{RGN}}$ and rgnURAL. In order to justify this claim, we showed how a region calculus with a traditional type-and-effect system may be translated to the $\mathsf{F}^{\mathsf{RGN}}$ language and we showed how the $\mathsf{F}^{\mathsf{RGN}}$ language may be translated to the rgnURAL language.

It is worth reviewing these translation before turning again to the claim that the monadic and substructural type systems are simpler than the type-and-effect systems.

We began with the Traditional Region Calculus (TRC), which corresponds directly to type-and-effect systems given in the literature. Its defining characteristics

are the form of the function type, the region abstraction type, and the type-and-effect judgment for expressions:

$$\tau_1 \xrightarrow{\phi} \tau_2 \qquad\qquad \forall \varrho.^{\phi}\, \tau \qquad\qquad \Delta; \Gamma \vdash_{\text{exp}} e : \tau, \phi$$

where the effect ϕ is a finite set of regions. We also introduced the Bounded Region Calculus (BRC), which extends TRC with a form of bounded region subtyping. Hence, the form of the function type, the region abstraction type, and the type-and-effect judgment for expressions are given as follows:

$$\tau_1 \xrightarrow{\phi} \tau_2 \qquad\qquad \forall \varrho \succeq \phi'.^{\phi}\, \tau \qquad\qquad \Delta; \Gamma \vdash_{\text{exp}} e : \tau, \phi$$

where the effect ϕ' serves as a lower bound on the lifetime of any region that instantiates ϱ. There is a trivial translation from TRC to BRC, whereby every region abstraction becomes a region abstraction with an empty bound.

The Single Effect Calculus (SEC) restricts BRC by admitting only a single region as the latent effect in a function or region abstraction type or in the type-and-effect judgment for expressions. Using the intuition that, given the partial order on live regions imposed by their nested lifetimes, a single region can serve as a witness for a set of regions, we showed how to translate from BRC to SEC. In terms of translating types and judgments, the key translations were the following:

$$\begin{array}{ccc}
\mathsf{TRC/BRC} & \rightsquigarrow & \mathsf{SEC} \\
(\tau_1 \xrightarrow{\phi} \tau_2, \rho) & \rightsquigarrow & (\forall \varpi \succeq \phi.^{\rho}\, (\tau_1^{\dagger} \xrightarrow{\varpi} \tau_2^{\dagger}, \rho), \rho) \\
(\forall \varrho \succeq \phi'.^{\phi}\, \tau, \rho) & \rightsquigarrow & (\forall \varrho \succeq \phi'.^{\rho}\, (\forall \varpi \succeq \phi.^{\varpi}\, \tau^{\dagger}, \rho), \rho) \\
\Delta; \Gamma \vdash^{\mathsf{BRC}}_{\text{exp}} e : \tau, \phi & \rightsquigarrow & \Delta^{\dagger}; \Gamma^{\dagger} \vdash^{\mathsf{SEC}}_{\text{exp}} e^{\dagger} : \tau^{\dagger}, \pi \quad \text{where } \pi \succeq \phi
\end{array}$$

The $\mathsf{F}^{\mathsf{RGN}}$ language introduced a monadic type system for region-based memory management. We used a monadic type, RGN θ τ, to represent computations

which transform a stack of regions indexed by θ and deliver a value of type τ. We also used region handles ($\mathsf{RGNHnd}\ \theta$) and region references ($\mathsf{RGNRef}\ \theta\ \tau$) to more accurately model the run-time behavior of region-based memory management. Finally, we introduced the type (abbreviation) $\mathsf{RGNPf}(\theta_1 \preceq \theta_2)$ to represent evidence of the fact that the stack of regions indexed by θ_1 is outlived by the stack of regions indexed by θ_2. Monadic encapsulation and parametric polymorphism provide sufficient "typing machinery" to ensure region safety; furthermore, all of the monadic commands for region-based memory management may be assigned conventional polymorphic types.

The translation from SEC to $\mathsf{F}^{\mathsf{RGN}}$ is primarily concerned with (1) eliminating region outlives relationships (using explicit evidence) and (2) sequencing computations using the monadic commands. The translation also introduces uses of region handles and region references. In terms of translating types and judgments, the key translations were the following:

$$
\begin{array}{rcl}
\mathsf{SEC} & \leadsto & \mathsf{F}^{\mathsf{RGN}} \\
(\tau_1 \xrightarrow{\pi} \tau_2, \rho) & \leadsto & \mathsf{RGNRef}\ \theta_\rho\ (\tau_1^\dagger \to \mathsf{RGN}\ \theta_\pi\ \tau_2^\dagger) \\
(\forall \varrho \succeq \{\rho'_1, \ldots \rho'_n\}.^{\pi}\, \tau, \rho) & \leadsto & \\
\multicolumn{3}{c}{\mathsf{RGNRef}\ \theta_\rho\ (\forall \vartheta_\varrho.\, (\mathsf{RGNPf}(\theta_{\rho'_1} \preceq \vartheta_\varrho) \times \cdots \times \mathsf{RGNPf}(\theta_{\rho'_n} \preceq \vartheta_\varrho))} \\
\multicolumn{3}{c}{\to \mathsf{RGNHnd}\ \vartheta_\varrho \to \mathsf{RGN}\ \theta_\pi\ \tau^\dagger)} \\
\Delta; \Gamma \vdash^{\mathsf{SEC}}_{\mathrm{exp}} e : \tau, \pi & \leadsto & \Delta^\dagger; \Gamma^\dagger \vdash^{\mathsf{F}^{\mathsf{RGN}}}_{\mathrm{exp}} e^\dagger : \mathsf{RGN}\ \theta_\pi\ \tau^\dagger
\end{array}
$$

The $\mathsf{rgnURAL}$ language introduced a substructural type system for region-based memory management. We used separate primitives for creating and destroying regions, which allows regions to have non-nested lifetimes. We continued to use region handles ($\overline{\mathsf{Hnd}}\ \rho$) and region references ($\overline{\mathsf{Ref}}\ \rho\ \tau$) to more accurately model the run-time behavior of region-based memory management. Finally, we intro-

duced the pre-type $\overline{\mathsf{Cap}}\ \rho$ to represent a capability that mediates access to a region (for allocating, reading, and writing references in the region and for destroying the region). A substructural type system (making use of the substructural qualifiers U, R, A, and L) provides sufficient "typing machinery" to ensure region safety; in particular, only regions with unique (A and L) capabilities may be destroyed, ensuring that there are no other copies of the capability in the program state to access the region.

The translation from $\mathsf{F}^{\mathsf{RGN}}$ to $\mathsf{rgnURAL}$ is primarily concerned with (1) exposing the stack-passing implementation of $\mathsf{RGN}\ \theta\ \tau$ computations and (2) eliminating explicit evidence terms by rearranging the representation of the region stack for a nested computation. It also handles the slight mismatch between the $\mathsf{RGNHnd}\ \theta$ and $\mathsf{RGNRef}\ \theta\ \tau$ types in $\mathsf{F}^{\mathsf{RGN}}$ and the $\overline{\mathsf{Hnd}}\ \rho$ and $\overline{\mathsf{Ref}}\ \rho\ \tau$ pre-types in $\mathsf{rgnURAL}$; the former types denote a handle for or reference in some region in the stack indexed by θ, while the latter pre-types denote a handle for or reference in a specific region named ρ. In terms of translating types and judgments, the key translations were the following:

$$
\begin{array}{rcl}
\mathsf{F}^{\mathsf{RGN}} & \rightsquigarrow & \mathsf{rgnURAL} \\
\mathsf{RGN}\ \theta\ \tau & \rightsquigarrow & {}^{\mathsf{U}}(\tau_\theta \multimap {}^{\mathsf{L}}(\tau_\theta \otimes \tau^\dagger)) \\
\mathsf{RGNHnd}\ \theta & \rightsquigarrow & \\
\multicolumn{3}{c}{{}^{\mathsf{U}}(\overline{\exists}\varrho.\, {}^{\mathsf{U}}({}^{\mathsf{U}}(\overline{\exists}\beta.\, \mathbf{Iso}(\tau_\theta, {}^{\mathsf{L}}(\beta \otimes {}^{\mathsf{L}}(\overline{\mathsf{Cap}}\ \varrho)))) \otimes {}^{\mathsf{U}}(\overline{\mathsf{Hnd}}\ \varrho)))} \\
\mathsf{RGNRef}\ \theta\ \tau & \rightsquigarrow & \\
\multicolumn{3}{c}{{}^{\mathsf{U}}(\overline{\exists}\varrho.\, {}^{\mathsf{U}}({}^{\mathsf{U}}(\overline{\exists}\beta.\, \mathbf{Iso}(\tau_\theta, {}^{\mathsf{L}}(\beta \otimes {}^{\mathsf{L}}(\overline{\mathsf{Cap}}\ \varrho)))) \otimes {}^{\mathsf{U}}(\overline{\mathsf{Ref}}\ \varrho\ \tau^\dagger)))} \\
\Delta; \Gamma \vdash^{\mathsf{F}^{\mathsf{RGN}}}_{\mathrm{exp}} e : \tau & \rightsquigarrow & \Delta^\dagger; \Gamma^\dagger \vdash^{\mathsf{rgnURAL}}_{\mathrm{exp}} e^\dagger : \tau^\dagger
\end{array}
$$

Thus far, we have considered the various translations in isolation. We can, of course, compose the translations, to yield a translation from $\mathsf{TRC}/\mathsf{BRC}$ to $\mathsf{rgnURAL}$.

TRC/BRC

$$(\tau_1^{\mathsf{TRC}} \xrightarrow{\{\rho_1,\ldots,\rho_n\}} \tau_2^{\mathsf{TRC}}, \rho)$$

SEC

$$(\forall\varpi \succeq \{\rho_1,\ldots,\rho_n\}.^{\rho}\, (\tau_1^{\mathsf{SEC}} \xrightarrow{\varpi} \tau_2^{\mathsf{SEC}}, \rho), \rho)$$

$\mathsf{F}^{\mathsf{RGN}}$

$$\begin{aligned}
&\mathsf{RGNRef}\ \theta_\rho\ (\forall\vartheta_\varpi.\,(\mathsf{RGNPf}(\theta_{\rho_1} \preceq \vartheta_\varpi) \times \cdots \times \mathsf{RGNPf}(\theta_{\rho_n} \preceq \vartheta_\varpi)) \rightarrow \\
&\qquad\qquad \mathsf{RGNHnd}\ \vartheta_\varpi \rightarrow \\
&\qquad\qquad \mathsf{RGN}\ \theta_\rho\ (\mathsf{RGNRef}\ \theta_\rho\ (\tau_1^{\mathsf{F}^{\mathsf{RGN}}} \rightarrow \mathsf{RGN}\ \vartheta_\varpi\ \tau_2^{\mathsf{F}^{\mathsf{RGN}}})))
\end{aligned}$$

rgnURAL

$$\begin{aligned}
&{}^{\mathsf{U}}(\overline{\exists}\varrho_a.\,{}^{\mathsf{U}}({}^{\mathsf{U}}(\overline{\exists}\beta_a.\,\mathbf{Iso}(\tau_{\theta_\rho}, {}^{\mathsf{L}}(\beta_a \otimes {}^{\mathsf{L}}(\overline{\mathsf{Cap}}\ \varrho_a)))) \otimes {}^{\mathsf{U}}(\overline{\mathsf{Ref}}\ \varrho_a\ \tau_Z))) \\
\text{where}\quad \tau_Z &\equiv {}^{\mathsf{U}}(\overline{\forall}\alpha_{\vartheta_\varpi}.\,{}^{\mathsf{U}}(\tau_Y \multimap {}^{\mathsf{U}}(\tau_X \multimap \tau_W))) \\
\tau_Y &\equiv {}^{\mathsf{U}}(\mathbb{T}[\![\mathsf{RGNPf}(\theta_{\rho_1} \preceq \vartheta_\varpi)]\!] \otimes \cdots \otimes \mathbb{T}[\![\mathsf{RGNPf}(\theta_{\rho_n} \preceq \vartheta_\varpi)]\!]) \\
\tau_X &\equiv {}^{\mathsf{U}}(\overline{\exists}\varrho_b.\,{}^{\mathsf{U}}({}^{\mathsf{U}}(\overline{\exists}\beta_b.\,\mathbf{Iso}(\alpha_{\vartheta_\varpi}, {}^{\mathsf{L}}(\beta_b \otimes {}^{\mathsf{L}}(\overline{\mathsf{Cap}}\ \varrho_b)))) \otimes {}^{\mathsf{U}}(\overline{\mathsf{Hnd}}\ \varrho_b))) \\
\tau_W &\equiv {}^{\mathsf{U}}(\tau_{\theta_\rho} \multimap {}^{\mathsf{L}}(\tau_{\theta_\rho} \otimes \tau_V)) \\
\tau_V &\equiv {}^{\mathsf{U}}(\overline{\exists}\varrho_c.\,{}^{\mathsf{U}}({}^{\mathsf{U}}(\overline{\exists}\beta_c.\,\mathbf{Iso}(\tau_{\theta_\rho}, {}^{\mathsf{L}}(\beta_c \otimes {}^{\mathsf{L}}(\overline{\mathsf{Cap}}\ \varrho_c)))) \otimes {}^{\mathsf{U}}(\overline{\mathsf{Ref}}\ \varrho_c\ \tau_U))) \\
\tau_U &\equiv {}^{\mathsf{U}}(\tau_1^{\mathsf{rgnURAL}} \multimap {}^{\mathsf{U}}(\alpha_{\vartheta_\varpi} \multimap {}^{\mathsf{L}}(\alpha_{\vartheta_\varpi} \otimes \tau_2^{\mathsf{rgnURAL}})))
\end{aligned}$$

Figure 6.1: Translation from TRC/BRC to rgnURAL (function type)

Figure 6.1 shows the translation of the TRC function boxed-type, through SEC and $\mathsf{F}^{\mathsf{RGN}}$, to a rgnURAL type. At first glance, this translation would appear to be a serious strike against the claim that the monadic and substructural type systems are simpler than the type-and-effect systems. It certainly appears that the TRC function boxed-type is simpler than its translation into $\mathsf{F}^{\mathsf{RGN}}$ and into rgnURAL.

However, the apparent complexity of the $\mathsf{F}^{\mathsf{RGN}}$ and rgnURAL types are actually evidence that the corresponding monadic and substructural type systems are simpler than the type-and-effect systems. To see this, we recall that the translations are type- and meaning-preserving. Hence, each of the types in Figure 6.1 represent values with the same computational behavior. The TRC function boxed-type combines many aspects of region-based memory management into a single type; hence, we must understand and reason about the TRC type as a whole. The $\mathsf{F}^{\mathsf{RGN}}$ and rgnURAL types in Figure 6.1, while complex in their entirety, are the composition of simpler individual types. For example, the $\mathsf{F}^{\mathsf{RGN}}$ type makes it clear that region handles and region references are distinct aspects of region-based memory management. Similarly, the $\mathsf{F}^{\mathsf{RGN}}$ type makes it clear that the effect and region subtyping in TRC may be factored out using explicit evidence of to the nested lifetimes of regions. The rgnURAL type exposes yet more distinct aspects of region-based memory management. For example, it demonstrates that the $\mathsf{F}^{\mathsf{RGN}}$ computations may be realized using a stack-passing implementation. Similarly, it demonstrates that region handles and references must identify a specific region within the implicit region stack. Hence, the translations show how much complexity is hidden "behind the scenes" in the deceptively simple TRC function boxed-type. We are able to explain this hidden complexity in terms of the distinguished components of the $\mathsf{F}^{\mathsf{RGN}}$ and rgnURAL languages. This helps demonstrate that the monadic

and substructural type systems are, at their core, simpler than the type-and-effect systems, but nonetheless are capable of expressing complex interactions among these components.

Another way of evaluating the various type systems for region-based memory management is to consider what type system features are used to support region-based memory management. Type-and-effect systems introduce *heavy-weight features* into the type system *exclusively* for supporting region-based memory management. For example, at the type level, they introduce a new syntactic class for effects, which are meant to be treated as sets of regions, so standard term equality does not suffice for type checking. The `letregion` construct requires a distinguished typing rule to account for the interplay of dangling pointers and affects. Similarly, a type-and-effect system requires special type-and-effect rules for functions and applications to account for latent effects. All of these features make a type-and-effect system both specialized to region-based memory management and distant from well-understood and widely-used type systems.

On the other hand, the monadic and substructural type systems introduce *light-weight primitives* and *reuse features* of the corresponding type system to *encode* proper region-based memory management. For example, we have noted that all of the $\mathsf{F}^{\mathsf{RGN}}$ monadic commands may be assigned conventional polymorphic types. A key aspect of the $\mathsf{F}^{\mathsf{RGN}}$ language is that it adopts the well-understood and widely-used type system of System F with no extensions. All that was required to support region-based memory management was to introduce the types RGN θ τ, RGNHnd θ, and RGNRef θ τ and the monadic commands with appropriate polymorphic types; the type system of System F required no changes (e.g., the typing rules for functions and applications are the same in System F and $\mathsf{F}^{\mathsf{RGN}}$).

Similarly, the substructural type system for the rgnURAL language needs no region specific "typing-machinery." In particular, we note that the majority of the typing rules for rgnURAL either follow directly from the corresponding rules in the λ^{URAL}-calculus or support the extension of the λ^{URAL}-calculus with universal and existential quantification. There are no extensions to the type system exclusively for supporting region-based memory management; rather, all that was required to support region-based memory management was to introduce the pre-types $\overline{\mathsf{Cap}}\ \rho$, $\overline{\mathsf{Hnd}}\ \rho$, and $\overline{\mathsf{Ref}}\ \rho\ \tau$ and the region and reference primitives. Although we may not assign conventional polymorphic types to the region and reference primitives (the appropriate constraints between types and qualifiers cannot be expressed in a polymorphic type), we note that the typing rules for the region and reference primitives need no special auxiliary judgments; rather, they may use of the judgments $\Delta \vdash q \preceq q'$ and $\Delta \vdash \tau \preceq q'$, which are used extensively throughout the rules for the other terms in the language.

Hence, we believe that we have established the claim that monadic and substructural type systems give rise to simpler, more expressive, and more uniform languages that continue to provide the power and safety of region-based memory management. Certainly, the rgnURAL language is more expressive than the Tofte-Talpin region calculus, as the region primitives of rgnURAL allow regions to have non-nested lifetimes. As we demonstrated in Section 5.5, the rgnURAL language is expressive enough to encode a number of advanced memory-management features of Cyclone. We may also see that the $\mathsf{F}^{\mathsf{RGN}}$ and rgnURAL languages are more uniform, in the sense that they uniformly represent evidence and capabilities as first-class values to be manipulated by the program, rather than leaving such aspects implicit in the type system.

6.1 Future Directions

When viewing the work in this dissertation as a whole, there are three major directions for future research, in addition to those considered previously in Sections 3.5 and 4.5.

The first is to note that while this dissertation has focused on region-based memory management, many of the themes explored here apply to *any* resource management problem. We began this dissertation by noting that memory is an essential resource used by computer programs. There are many other resources that may be acquired and released during the execution of a program: file handles, database connections, concurrency locks, graphics processor texture and shader units, etc. There are also less tangible, but important, "resources" that are used by a program, such as the current state within a network or cryptographic protocol.

Understanding how best to manage a variety of resources is an important direction for future research. As we continue to focus on static type systems, we see a need to move beyond types as "persistent" invariants towards types as "ephemeral" (or "dynamic") invariants; that is, invariants about the program or resources which are only true under some conditions or are only true for a limited duration. A substructural type system, like that considered in Chapter 4, would appear to be a good starting point, since the "persistent"/"ephemeral" distinction can be roughly approximated by the substructural qualifiers. Recall that in the translation from $\mathsf{F}^{\mathsf{RGN}}$ to rgnURAL, an unrestricted isomorphism corresponded to the fact that the proof that a capability is a member of a stack is persistent, while a linear capability corresponds to the fact that the proof that a region is live is ephemeral.

We believe that the uniformity of the monadic and substructural type systems discussed in the previous section should allow additional resources to be easily

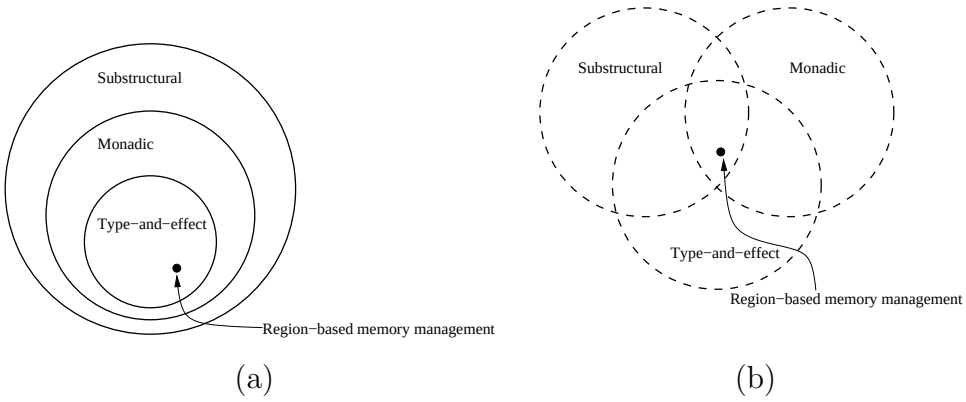

Figure 6.2: Relationships among three "flavors" of type systems

integrated into a language. In fact, others have adapted the ideas of the monadic type system for $\mathsf{F}^{\mathsf{RGN}}$ to provide a safe interface to file handles in Haskell [51].

A second major direction for future research is to further explore the relationships among type-and-effect systems, monadic type systems, and substructural type systems. Recall that we have demonstrated that one may successively encode the type-and-effect system of SEC into the monadic type system of $\mathsf{F}^{\mathsf{RGN}}$ and the monadic type system of $\mathsf{F}^{\mathsf{RGN}}$ into the substructural type system of rgnURAL. It is tempting to conclude from this result that we can order these three "flavors" of type systems by increasing expressiveness (Figure 6.2(a)). However, a more accurate picture is given by Figure 6.2(b), where region-based memory management falls into the intersection of these three "flavors" of type systems, as one feature that may be handled by all of them, and where the boundaries between these three "flavors" is indistinct (recall the hybrid monadic and substructural type system given in Section 5.5.

Hence, a particularly interesting direction for future research is to better understand what truly distinguishes one "flavor" of type system from another. As the work in this dissertation has demonstrated, we have a fairly good understanding

of the sorts of program behaviors that can be statically enforced in the intersections. However, we do not have as good an understanding of the sorts of program behaviors that can only be statically enforced by one "flavor" of type system and not by either of the others.

Perhaps surprisingly, we believe that the full range of type-and-effect systems have yet to be satisfactorily explored. As we noted in Chapter 2, variation amongst type-and-effect systems largely arises from the choice of effect language and the choice of auxiliary judgments that prove when one effect term is equivalent to or subsumed by another. Practically every type-and-effect system includes a number of atomic effects and an algebraic structure for combining effects. Effect terms as finite sets of atomic effects (as in the Tofte-Talpin region calculus) have been studied, but appear to have limited application. Exploring other algebraic structures should help illuminate the range of type-and-effect systems. At one end of the spectrum are program behaviors where it suffices to distinguish between the presence or absence of an effect (e.g., I/O interaction, non-termination, dynamic behavior). It may be possible to exploit the simplicity of these effect terms to yield simpler type-and-effect systems.

At the other end of the spectrum are applications where it is necessary for effect terms to accurately reflect run-time behavior of a program. For example, compilers are often conservative in the presence of effects like exceptions, because transformations that change the order of observed exceptions are not semantics preserving. Richer effect algebras offer a means by which program transformations may be enabled or disabled based on whether or not they preserve the effect of an expression. An important direction for future research is to hone in on suitable effect algebras that offer the right trade-offs between expressiveness and tractability.

A final major direction for future research is to explore how to best design high-level programming languages that integrate the monadic and substructural type systems explored in this dissertation. We have noted that it is our intention that $\mathsf{F}^{\mathsf{RGN}}$ and rgnURAL be considered as core languages, suitable for service as compiler intermediate languages or as a vehicles for formal reasoning; they are not suitable for service as high-level programming languages, since they lack many features that are essential in such a general-purpose programming language. It is also important to explore ways to minimize the burden placed on a programmer, who would otherwise need to account for a number of administrative details (e.g., composing and applying evidence terms in $\mathsf{F}^{\mathsf{RGN}}$, passing capabilities in rgnURAL). The investigation of inference algorithms and flow analyses should yield insights that make the type systems more palatable in high-level programming languages.

Appendix A

Type-and-Effect Systems: Technical Details

This appendix supplements the material in Chapter 2 with a number of technical details that would otherwise detract from that chapter's focus on the definition of the static semantics for th surface syntax of the Single Effect Calculus. In the following section, we revisit the presentation of the Single Effect Calculus, revising the static semantics to include judgment for the additional semantic objects introduced by the abstract machine configurations. These additional judgments are also necessary to support the proof of the correctness of the translation from SEC to $\mathsf{F}^{\mathsf{RGN}}$ in Appendix B.3.

In Appendix A.2, we sketch a syntactic proof of that evaluation in SEC preserves types. Yet more details, including complete proofs, may be found in the technical report *Monadic Regions: Formal Type Soundness and Correctness* [21].

A.1 The Single Effect Calculus

A.1.1 Static Semantics of SEC

Section 2.2.5 gave the static semantics for the surface syntax of SEC. However, the judgments given in that section are insufficient for carrying out a syntactic proof of type soundness, since there are no rules for `ref` terms (which arise during the evaluation of a program) and there is no typing judgment for stacks. This section extends the static semantics of Section 2.2.5 to overcome these deficiencies. In addition to the typing judgments for expressions and various well-formedness

$$
\begin{array}{lrcl}
\text{Region contexts} & \Delta & ::= & \cdot \mid \Delta, \varrho \succeq \phi \\
\text{Expression contexts} & \Gamma & ::= & \cdot \mid \Gamma, x{:}\tau \\
\text{Region domains} & \overline{\mathcal{R}} & ::= & \{\mathfrak{p}_1, \ldots, \mathfrak{p}_n\} \\
\text{Region types} & \mathcal{R} & ::= & \{\mathfrak{p}_1 \mapsto \omega_1, \ldots, \mathfrak{p}_n \mapsto \omega_n\} \\
\text{Stack domains} & \overline{\mathcal{S}} & ::= & \cdot \mid \overline{\mathcal{S}}, \mathfrak{r} \mapsto \overline{\mathcal{R}} \quad \text{(ordered domain)} \\
\text{Stack types} & \mathcal{S} & ::= & \cdot \mid \mathcal{S}, \mathfrak{r} \mapsto \mathcal{R} \quad \text{(ordered domain)}
\end{array}
$$

$$
\begin{array}{rcl}
\overline{\mathcal{S}} \sqsupseteq \overline{\mathcal{S}}' & \equiv & dom(\overline{\mathcal{S}}) = dom(\overline{\mathcal{S}}') \wedge \\
& & \quad \forall \mathfrak{r} \in dom(\overline{\mathcal{S}}').\ dom(\overline{\mathcal{S}}(\mathfrak{r})) \supseteq dom(\overline{\mathcal{S}}'(\mathfrak{r}))
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{S} \sqsupseteq \mathcal{S}' & \equiv & dom(\mathcal{S}) = dom(\mathcal{S}') \wedge \\
& & \quad \forall \mathfrak{r} \in dom(\mathcal{S}').\ dom(\mathcal{S}(\mathfrak{r})) \supseteq dom(\mathcal{S}'(\mathfrak{r})) \wedge \\
& & \quad\quad \forall \mathfrak{p} \in dom(\mathcal{S}'(\mathfrak{r})).\ \mathcal{S}(\mathfrak{r}, \mathfrak{p}) = \mathcal{S}'(\mathfrak{r}, \mathfrak{p})
\end{array}
$$

Figure A.1: Static semantics of SEC (definitions)

judgments for boxed types, types, and contexts given previously, we have typing judgments for values and storable values and judgments that check the type and well-formedness of stacks.

Definitions Figure A.1 presents additional definitions for syntactic objects that appear in the static semantics. Stack and region types mimic stacks and regions, recording the type of the value stored at each pointer. Stack and region domains are a technical device that records the live region and pointer names. Because proving the well-formedness of stack types requires proving the well-formedness of types, which requires verifying that region names are live, one cannot easily have stack types serve the dual purpose of recording live names. We tacitly assume that all domains are well-formed – containing distinct region names and pointer names.

$\boxed{\Delta; \overline{\mathbb{S}} \vdash_{\text{rr}} \rho_2 \succeq \rho_1}$

$$\frac{\overline{\mathbb{S}} \vdash_{\text{rctxt}} \Delta \qquad (\varrho \succeq \{\rho_1, \ldots, \rho_i, \ldots, \rho_n\}) \in \Delta}{\Delta; \overline{\mathbb{S}} \vdash_{\text{rr}} \varrho \succeq \rho_i} \qquad \frac{\Delta; \overline{\mathbb{S}} \vdash_{\text{region}} \rho}{\Delta; \overline{\mathbb{S}} \vdash_{\text{rr}} \rho \succeq \rho}$$

$$\frac{\Delta; \overline{\mathbb{S}} \vdash_{\text{rr}} \rho_2 \succeq \rho' \qquad \Delta; \overline{\mathbb{S}} \vdash_{\text{rr}} \rho' \succeq \rho_1}{\Delta; \overline{\mathbb{S}} \vdash_{\text{rr}} \rho_2 \succeq \rho_1}$$

$$\frac{\overline{\mathbb{S}} \vdash_{\text{rctxt}} \Delta \qquad \overline{\mathbb{S}} = \overline{\mathbb{S}}_1, \mathfrak{r}_1 \mapsto \overline{\mathcal{R}}_1, \overline{\mathbb{S}}_2, \mathfrak{r}_2 \mapsto \overline{\mathcal{R}}_2, \overline{\mathbb{S}}_3}{\Delta; \overline{\mathbb{S}} \vdash_{\text{rr}} \mathfrak{r}_2 \succeq \mathfrak{r}_1} \qquad \frac{\Delta; \overline{\mathbb{S}} \vdash_{\text{re}} \mathfrak{r} \succeq \{\mathfrak{r}_1, \ldots, \mathfrak{r}_n\}}{\Delta; \overline{\mathbb{S}} \vdash_{\text{rr}} \mathfrak{r} \succeq \mathfrak{r}_i}$$

$$\frac{\overline{\mathbb{S}} \vdash_{\text{rctxt}} \Delta \qquad \overline{\mathbb{S}} = \overline{\mathbb{S}}_1, \mathfrak{r}_1 \mapsto R_1, \overline{\mathbb{S}}_2}{\Delta; \overline{\mathbb{S}} \vdash_{\text{rr}} \bullet \succeq \mathfrak{r}_1}$$

$\boxed{\Delta; \overline{\mathbb{S}} \vdash_{\text{re}} \rho \succeq \phi}$

$$\frac{\overline{\mathbb{S}} \vdash_{\text{rctxt}} \Delta \qquad \Delta; \overline{\mathbb{S}} \vdash_{\text{rr}} \rho \succeq \rho_i \quad {}^{i \in 1 \ldots n}}{\Delta; \overline{\mathbb{S}} \vdash_{\text{re}} \rho \succeq \{\rho_1, \ldots, \rho_n\}}$$

Figure A.2: Static semantics of SEC (outlives judgments)

We define the relation $\sqsupseteq$ to describe extensions of stack domains and types. Note that we consider stack domains and types to have ordered domains. Hence, $dom(S) \sqsupseteq dom(S')$ indicates that the ordered domain of $dom(S')$ is a prefix of the ordered domain of $dom(S)$.

Outlives judgments Figure A.2 gives the judgments that formalize the liveness relationships between regions and effects. We summarize these judgments in the

following table:

Judgment	Meaning
$\Delta; \overline{\mathcal{S}} \vdash_{\text{rr}} \rho_2 \succeq \rho_1$	If region ρ_2 is live, then region ρ_1 is live. (Alt.: region ρ_1 outlives region ρ_2.)
$\Delta; \overline{\mathcal{S}} \vdash_{\text{re}} \rho \succeq \phi$	If region ρ is live, then all regions in ϕ are live. (Alt.: all regions in ϕ outlive region ρ.)

We note that the typing rules for the judgments $\vdash_{\text{rr}}$ and $\vdash_{\text{re}}$ simply formalize the reflexive, transitive closure of the syntactic constraints in Δ, each of which asserts a particular "outlived by" relation between a region variable and an effect set, and $\overline{\mathcal{S}}$, which asserts "outlived by" relations by explicit ordering of region names. Note that the judgment $\Delta; \overline{\mathcal{S}} \vdash_{rr} \rho_2 \succeq \rho_1$ is not syntax directed.

Terms Figures A.3–A.6 present the typing rules for the judgment $\Delta; \Gamma; \mathcal{S} : \overline{\mathcal{S}} \vdash_{\text{exp}} e : \tau, \pi$, which asserts that under the region context Δ, the value context Γ, and the stack type $\mathcal{S}$ with the stack domain $\overline{\mathcal{S}}$, the expression e has type τ and effects bounded by the region π. Figures A.3, A.4, and A.5 repeat the rules from Figures 2.22 and 2.23SECStaticSemanticsExpIII in order to demonstrate the manner in which stack types and stack domains are propagated through the rules given in Section 2.2.5. In particular, note that in every rule, stack types and stack domains are passed unmodified to sub-judgments.

Figure A.6 gives typing rules for reference expressions. The first rule asserts that a reference to a pointer in a dead region may be assigned any well-formed boxed type. The second rule ensures that any region name that appears in a reference is in scope; furthermore, a pointer in a live region denotes a value with the boxed type assigned by the stack type.

$\boxed{\Delta; \Gamma; \mathcal{S} : \overline{\mathcal{S}} \vdash_{\text{exp}} e : \tau, \pi}$

$$\frac{\vdash_{\text{ctxt}} \Delta; \Gamma; \mathcal{S} : \overline{\mathcal{S}}; \pi \qquad \Delta; \overline{\mathcal{S}} \vdash_{\text{region}} \rho \qquad \Delta; \overline{\mathcal{S}} \vdash_{\text{rr}} \pi \succeq \rho}{\Delta; \Gamma; \mathcal{S} : \overline{\mathcal{S}} \vdash_{\text{exp}} \mathtt{i}\,\mathtt{at}\,\rho : (\mathsf{Int}, \rho), \pi}$$

$$\frac{\begin{array}{c}\Delta; \Gamma; \mathcal{S} : \overline{\mathcal{S}} \vdash_{\text{exp}} e_1 : (\mathsf{Int}, \rho_1), \pi \qquad \Delta; \overline{\mathcal{S}} \vdash_{\text{rr}} \pi \succeq \rho_1 \\ \Delta; \Gamma; \mathcal{S} : \overline{\mathcal{S}} \vdash_{\text{exp}} e_2 : (\mathsf{Int}, \rho_2), \pi \qquad \Delta; \overline{\mathcal{S}} \vdash_{\text{rr}} \pi \succeq \rho_2 \\ \Delta; \overline{\mathcal{S}} \vdash_{\text{region}} \rho \qquad \Delta; \overline{\mathcal{S}} \vdash_{\text{rr}} \pi \succeq \rho\end{array}}{\Delta; \Gamma; \mathcal{S} : \overline{\mathcal{S}} \vdash_{\text{exp}} e_1 \oplus e_2\,\mathtt{at}\,\rho : (\mathsf{Int}, \rho), \pi}$$

$$\frac{\begin{array}{c}\Delta; \Gamma; \mathcal{S} : \overline{\mathcal{S}} \vdash_{\text{exp}} e_1 : (\mathsf{Int}, \rho_1), \pi \qquad \Delta; \overline{\mathcal{S}} \vdash_{\text{rr}} \pi \succeq \rho_1 \\ \Delta; \Gamma; \mathcal{S} : \overline{\mathcal{S}} \vdash_{\text{exp}} e_2 : (\mathsf{Int}, \rho_2), \pi \qquad \Delta; \overline{\mathcal{S}} \vdash_{\text{rr}} \pi \succeq \rho_2\end{array}}{\Delta; \Gamma \vdash_{\text{exp}} e_1 \otimes e_2 : \mathsf{Bool}, \pi} \qquad \frac{\vdash_{\text{ctxt}} \Delta; \Gamma; \mathcal{S} : \overline{\mathcal{S}}; \pi}{\Delta; \Gamma; \mathcal{S} : \overline{\mathcal{S}} \vdash_{\text{exp}} \mathfrak{b} : \mathsf{Bool}, \pi}$$

$$\frac{\begin{array}{c}\Delta; \Gamma; \mathcal{S} : \overline{\mathcal{S}} \vdash_{\text{exp}} e_b : \mathsf{Bool}, \pi \\ \Delta; \Gamma; \mathcal{S} : \overline{\mathcal{S}} \vdash_{\text{exp}} e_t : \tau, \pi \qquad \Delta; \Gamma; \mathcal{S} : \overline{\mathcal{S}} \vdash_{\text{exp}} e_f : \tau, \pi\end{array}}{\Delta; \Gamma; \mathcal{S} : \overline{\mathcal{S}} \vdash_{\text{exp}} \mathtt{if}\ e_b\ \mathtt{then}\ e_t\ \mathtt{else}\ e_f : \tau, \pi}$$

Figure A.3: Static semantics of SEC (expressions (I))

$\boxed{\Delta; \Gamma; \mathcal{S} : \overline{\mathcal{S}} \vdash_{\text{exp}} e : \tau, \pi}$

$$\frac{\vdash_{\text{ctxt}} \Delta; \Gamma; \mathcal{S} : \overline{\mathcal{S}}; \pi \qquad x \in dom(\Gamma) \qquad \Gamma(x) = \tau}{\Delta; \Gamma; \mathcal{S} : \overline{\mathcal{S}} \vdash_{\text{exp}} x : \tau, \pi}$$

$$\frac{\Delta; \Gamma, x{:}\tau_x; \mathcal{S} : \overline{\mathcal{S}} \vdash_{\text{exp}} e : \tau, \pi' \qquad \Delta; \overline{\mathcal{S}} \vdash_{\text{region}} \rho \qquad \Delta; \overline{\mathcal{S}} \vdash_{\text{rr}} \pi \succeq \rho}{\Delta; \Gamma; \mathcal{S} : \overline{\mathcal{S}} \vdash_{\text{exp}} \lambda x : \tau_x.^{\pi'}\, e\, \texttt{at}\, \rho : (\tau_x \xrightarrow{\pi'} \tau, \rho), \pi}$$

$$\frac{\begin{array}{c}\Delta; \Gamma; \mathcal{S} : \overline{\mathcal{S}} \vdash_{\text{exp}} e_f : (\tau_x \xrightarrow{\pi'_f} \tau, \rho_f), \pi \qquad \Delta; \overline{\mathcal{S}} \vdash_{\text{rr}} \pi \succeq \rho_f \\ \Delta; \Gamma; \mathcal{S} : \overline{\mathcal{S}} \vdash_{\text{exp}} e_a : \tau_x, \pi \qquad \Delta; \overline{\mathcal{S}} \vdash_{\text{rr}} \pi \succeq \pi'_f\end{array}}{\Delta; \Gamma; \mathcal{S} : \overline{\mathcal{S}} \vdash_{\text{exp}} e_f\ e_a : \tau, \pi}$$

$$\frac{\begin{array}{c}\Delta; \Gamma; \mathcal{S} : \overline{\mathcal{S}} \vdash_{\text{exp}} e_1 : \tau_1, \pi \qquad \cdots \qquad \Delta; \Gamma; \mathcal{S} : \overline{\mathcal{S}} \vdash_{\text{exp}} e_n : \tau_n, \pi \\ \Delta; \overline{\mathcal{S}} \vdash_{\text{region}} \rho \qquad \Delta; \overline{\mathcal{S}} \vdash_{\text{rr}} \pi \succeq \rho\end{array}}{\Delta; \Gamma; \mathcal{S} : \overline{\mathcal{S}} \vdash_{\text{exp}} \langle e_1, \ldots, e_n \rangle\, \texttt{at}\, \rho : (\tau_1 \times \cdots \times \tau_n, \rho), \pi}$$

$$\frac{\begin{array}{c}\Delta; \Gamma; \mathcal{S} : \overline{\mathcal{S}} \vdash_{\text{exp}} e : (\tau_1 \times \cdots \times \tau_n, \rho), \pi \\ \Delta; \overline{\mathcal{S}} \vdash_{\text{rr}} \pi \succeq \rho \qquad 0 \leq i \leq n\end{array}}{\Delta; \Gamma; \mathcal{S} : \overline{\mathcal{S}} \vdash_{\text{exp}} \texttt{sel}_i\ e : \tau_i, \pi}$$

Figure A.4: Static semantics of SEC (expressions (II))

$\boxed{\Delta; \Gamma; \mathcal{S} : \overline{\mathcal{S}} \vdash_{\text{exp}} e : \tau, \pi}$

$$\frac{\Delta; \overline{\mathcal{S}} \vdash_{\text{type}} \tau \qquad \vdash_{\text{ctxt}} \Delta; \Gamma; \mathcal{S} : \overline{\mathcal{S}}; \pi \qquad \Delta, \varrho \succeq \{\pi\}; \Gamma; \mathcal{S} : \overline{\mathcal{S}} \vdash_{\text{exp}} e_b : \tau, \varrho}{\Delta; \Gamma; \mathcal{S} : \overline{\mathcal{S}} \vdash_{\text{exp}} \texttt{letregion}\ \varrho\ \texttt{in}\ e_b : \tau, \pi}$$

$$\frac{\Delta, \varrho \succeq \phi; \Gamma; \mathcal{S} : \overline{\mathcal{S}} \vdash_{\text{exp}} u : \tau, \pi' \qquad \Delta; \overline{\mathcal{S}} \vdash_{\text{region}} \rho \qquad \Delta; \overline{\mathcal{S}} \vdash_{\text{rr}} \pi \succeq \rho}{\Delta; \Gamma; \mathcal{S} : \overline{\mathcal{S}} \vdash_{\text{exp}} \Lambda\varrho \succeq \phi.^{\pi'}\, u\, \texttt{at}\, \rho : (\forall \varrho \succeq \phi.^{\pi'} \tau, \rho), \pi}$$

$$\frac{\Delta; \Gamma; \mathcal{S} : \overline{\mathcal{S}} \vdash_{\text{exp}} e_f : (\forall \varrho \succeq \phi.^{\pi'_f} \tau, \rho_f), \pi \qquad \Delta; \overline{\mathcal{S}} \vdash_{\text{rr}} \pi \succeq \rho_f \qquad \Delta; \overline{\mathcal{S}} \vdash_{\text{region}} \rho_a \qquad \Delta; \overline{\mathcal{S}} \vdash_{\text{re}} \rho_a \succeq \phi \qquad \Delta; \overline{\mathcal{S}} \vdash_{\text{rr}} \pi \succeq \pi'_f[\rho_a/\varrho]}{\Delta; \Gamma; \mathcal{S} : \overline{\mathcal{S}} \vdash_{\text{exp}} e_f\ [\rho_a] : \tau[\rho_a/\varrho], \pi}$$

Figure A.5: Static semantics of SEC (expressions (III))

$\boxed{\Delta; \Gamma; \mathcal{S} : \overline{\mathcal{S}} \vdash_{\text{exp}} e : \tau, \pi}$

$$\frac{\vdash_{\text{ctxt}} \Delta; \Gamma; \mathcal{S} : \overline{\mathcal{S}}; \pi \qquad \Delta; \overline{\mathcal{S}} \vdash_{\text{btype}} \omega}{\Delta; \Gamma; \mathcal{S} : \overline{\mathcal{S}} \vdash_{\text{exp}} \texttt{ref}\ \bullet\ \mathfrak{p} : (\omega, \bullet), \pi}$$

$$\frac{\vdash_{\text{ctxt}} \Delta; \Gamma; \mathcal{S} : \overline{\mathcal{S}}; \pi \qquad \mathfrak{r} \in \overline{\mathcal{S}} \qquad \mathfrak{p} \in \overline{\mathcal{S}}(\mathfrak{r}) \qquad \mathcal{S}(\mathfrak{r}, \mathfrak{p}) = \omega}{\Delta; \Gamma; \mathcal{S} : \overline{\mathcal{S}} \vdash_{\text{exp}} \texttt{ref}\ \mathfrak{r}\ \mathfrak{p} : (\omega, \mathfrak{r}), \pi}$$

Figure A.6: Static semantics of SEC (expressions (IV))

$\boxed{\mathcal{S} : \overline{\mathcal{S}} \vdash_{\text{val}} v : \tau}$

$$\frac{\vdash_{\text{stype}} \mathcal{S} : \overline{\mathcal{S}}}{\mathcal{S} : \overline{\mathcal{S}} \vdash_{\text{val}} \mathfrak{b} : \mathsf{Bool}} \qquad \frac{\vdash_{\text{stype}} \mathcal{S} : \overline{\mathcal{S}} \qquad \cdot; \overline{\mathcal{S}} \vdash_{\text{btype}} \omega}{\mathcal{S} : \overline{\mathcal{S}} \vdash_{\text{val}} \texttt{ref} \bullet \mathfrak{p} : (\omega, \bullet)}$$

$$\frac{\vdash_{\text{stype}} \mathcal{S} : \overline{\mathcal{S}} \qquad \mathfrak{r} \in \overline{\mathcal{S}} \qquad \mathfrak{p} \in \overline{\mathcal{S}}(r) \qquad \mathcal{S}(\mathfrak{r}, \mathfrak{p}) = \omega}{\mathcal{S} : \overline{\mathcal{S}} \vdash_{\text{val}} \texttt{ref}\ \mathfrak{r}\ \mathfrak{p} : (\omega, \mathfrak{r})}$$

Figure A.7: Static semantics of SEC (values)

$\boxed{\mathcal{S} : \overline{\mathcal{S}} \vdash_{\text{sto}} w : \omega}$

$$\frac{\vdash_{\text{stype}} \mathcal{S} : \overline{\mathcal{S}}}{\mathcal{S} : \overline{\mathcal{S}} \vdash_{\text{sto}} \mathfrak{i} : \mathsf{Int}} \qquad \frac{\cdot; \cdot, x{:}\tau_x; \mathcal{S} : \overline{\mathcal{S}} \vdash_{\text{exp}} e : \tau, \pi'}{\mathcal{S} : \overline{\mathcal{S}} \vdash_{\text{sto}} \lambda x{:}\tau_x.^{\pi'}\, e : \tau_x \xrightarrow{\pi'} \tau}$$

$$\frac{\mathcal{S} : \overline{\mathcal{S}} \vdash_{\text{val}} v_1 : \tau_1 \qquad \cdots \qquad \mathcal{S} : \overline{\mathcal{S}} \vdash_{\text{val}} v_n : \tau_n}{\mathcal{S} : \overline{\mathcal{S}} \vdash_{\text{sto}} \langle v_1, \ldots, v_n \rangle : \tau_1 \times \cdots \times \tau_n} \qquad \frac{\cdot, \varrho \succeq \phi; \cdot; \mathcal{S} : \overline{\mathcal{S}} \vdash_{\text{exp}} u : \tau, \pi'}{\mathcal{S} : \overline{\mathcal{S}} \vdash_{\text{sto}} \Lambda\varrho \succeq \phi.^{\pi'}\, u : \forall \varrho \succeq \phi.^{\pi'}\, \tau}$$

Figure A.8: Static semantics of SEC (storable values)

$\boxed{\vdash_{\text{stype}} \mathcal{S} : \overline{\mathcal{S}}}$

$$\frac{\overline{\mathcal{S}} = dom(\mathcal{S}) \quad \forall \mathfrak{r} \in \overline{\mathcal{S}}.\ \overline{\mathcal{S}}(\mathfrak{r}) = dom(\mathcal{S}(\mathfrak{r})) \quad \forall \mathfrak{r} \in \overline{\mathcal{S}}.\ \forall \mathfrak{p} \in \overline{\mathcal{S}}(\mathfrak{r}).\ \cdot; \overline{\mathcal{S}} \vdash_{\text{btype}} \mathcal{S}(\mathfrak{r}, \mathfrak{p})}{\vdash_{\text{stype}} \mathcal{S} : \overline{\mathcal{S}}}$$

Figure A.9: Static semantics of SEC (stack types)

$\boxed{\vdash_{\text{stack}} S : \mathcal{S} : \overline{\mathcal{S}}}$

$$\frac{\vdash_{\text{stype}} \mathcal{S} : \overline{\mathcal{S}} \quad dom(\overline{\mathcal{S}}) = dom(\mathcal{S}) = dom(S) \quad \forall \mathfrak{r} \in dom(\overline{\mathcal{S}}).\ dom(\overline{\mathcal{S}}(\mathfrak{r})) = dom(\mathcal{S}(\mathfrak{r})) = dom(S(\mathfrak{r})) \quad \forall \mathfrak{r} \in dom(\overline{\mathcal{S}}).\ \forall \mathfrak{p} \in dom(\overline{\mathcal{S}}(\mathfrak{r})).\ \mathcal{S} : \overline{\mathcal{S}} \vdash_{\text{sto}} S(\mathfrak{r}, \mathfrak{p}) : (\mathcal{S}(\mathfrak{r}, \mathfrak{p}), \mathfrak{r})}{\vdash_{\text{stack}} S : \mathcal{S} : \overline{\mathcal{S}}}$$

Figure A.10: Static semantics of SEC (stacks)

Storable values and stacks Figures A.7 and A.8 give the typing rules for values and storable values, which follow directly from the typing rules for expressions. We require a separate judgments for these syntactic forms because the typing rules for expressions necessarily associate a single region bounding the effect of the expression. Since values and storable values are not evaluated, they have no effect.

Figures A.9 and A.10 presents typing rules that check the type and well-formedness of stacks. The judgment $\vdash_{\text{stype}} \mathcal{S} : \overline{\mathcal{S}}$ asserts that stack type $\mathcal{S}$ is well-formed with stack domain $\overline{\mathcal{S}}$. In particular, the judgment asserts that $\mathcal{S}$ has the domain specified by $\overline{\mathcal{S}}$ and each type in the range of $\mathcal{S}$ is well-formed. Finally, the

$\boxed{\Delta; \overline{\mathcal{S}} \vdash_{\text{region}} \rho}$

$$\frac{\overline{\mathcal{S}} \vdash_{\text{rctxt}} \Delta \qquad \varrho \in dom(\Delta)}{\Delta; \overline{\mathcal{S}} \vdash_{\text{region}} \varrho} \qquad \frac{\overline{\mathcal{S}} \vdash_{\text{rctxt}} \Delta \qquad \mathfrak{r} \in dom(\overline{\mathcal{S}})}{\Delta; \overline{\mathcal{S}} \vdash_{\text{region}} \mathfrak{r}} \qquad \frac{\overline{\mathcal{S}} \vdash_{\text{rctxt}} \Delta}{\Delta; \overline{\mathcal{S}} \vdash_{\text{region}} \bullet}$$

$\boxed{\Delta; \overline{\mathcal{S}} \vdash_{\text{eff}} \varphi}$

$$\frac{\overline{\mathcal{S}} \vdash_{\text{rctxt}} \Delta \qquad \Delta; \overline{\mathcal{S}} \vdash_{\text{region}} \rho_i \quad {}^{i \in 1 \ldots n}}{\Delta; \overline{\mathcal{S}} \vdash_{\text{eff}} \{\rho_1, \ldots, \rho_n\}}$$

Figure A.11: Static semantics of SEC (regions and effects)

judgment $\vdash_{\text{stack}} S : \mathcal{S} : \overline{\mathcal{S}}$ asserts that the stack S is well-formed with stack type $\mathcal{S}$ and stack domain $\overline{\mathcal{S}}$. Like the judgment $\vdash_{\text{stype}}$, it asserts that S has the domain specified by $\overline{\mathcal{S}}$ and each storable value in the range of S has the type specified by $\mathcal{S}$.

Regions, effects, boxed types, types, and contexts Figures A.11, A.12, and A.13 contain additional judgments for ensuring that regions ρ, effects ϕ, boxed types ω, types τ, region contexts Δ, and value contexts Γ are well-formed. Because regions, effects, boxed types, and types may contain region names, the judgments $\vdash_{\text{region}}$, $\vdash_{\text{effect}}$, $\vdash_{\text{btype}}$, $\vdash_{\text{type}}$, $\vdash_{\text{rctxt}}$, and $\vdash_{\text{vctxt}}$ require a stack domain $\overline{\mathcal{S}}$.

Surface programs and surface syntax Figure A.14 recalls the judgment $\vdash_{\text{prog}} e$. The rule for top-level surface programs requires that an expression evaluate to a boolean value in the context of distinguished region $\mathcal{H}$ that remains live throughout the execution of the program. It also serves as the single effect that bounds the effects of the entire program.

$\boxed{\Delta; \overline{\mathcal{S}} \vdash_{\text{btype}} \omega}$

$$\frac{\overline{\mathcal{S}} \vdash_{\text{rctxt}} \Delta}{\Delta; \overline{\mathcal{S}} \vdash_{\text{btype}} \mathsf{Int}} \qquad \frac{\Delta; \overline{\mathcal{S}} \vdash_{\text{type}} \tau_1 \qquad \Delta; \overline{\mathcal{S}} \vdash_{\text{region}} \pi' \qquad \Delta; \overline{\mathcal{S}} \vdash_{\text{type}} \tau_2}{\Delta; \overline{\mathcal{S}} \vdash_{\text{btype}} \tau_1 \xrightarrow{\pi'} \tau_2}$$

$$\frac{\Delta; \overline{\mathcal{S}} \vdash_{\text{type}} \tau_1 \qquad \cdots \qquad \Delta; \overline{\mathcal{S}} \vdash_{\text{type}} \tau_n}{\Delta; \overline{\mathcal{S}} \vdash_{\text{btype}} \tau_1 \times \cdots \times \tau_n}$$

$$\frac{\Delta; \overline{\mathcal{S}} \vdash_{\text{eff}} \phi \qquad \Delta, \varrho \succeq \phi; \overline{\mathcal{S}} \vdash_{\text{region}} \pi' \qquad \Delta, \varrho \succeq \phi; \overline{\mathcal{S}} \vdash_{\text{type}} \tau}{\Delta; \overline{\mathcal{S}} \vdash_{\text{btype}} \forall \varrho \succeq \phi.^{\pi'} \tau}$$

$\boxed{\Delta; \overline{\mathcal{S}} \vdash_{\text{type}} \tau}$

$$\frac{\overline{\mathcal{S}} \vdash_{\text{rctxt}} \Delta}{\Delta; \overline{\mathcal{S}} \vdash_{\text{type}} \mathsf{Bool}} \qquad \frac{\Delta; \overline{\mathcal{S}} \vdash_{\text{btype}} \omega \qquad \Delta; \overline{\mathcal{S}} \vdash_{\text{region}} \rho}{\Delta; \overline{\mathcal{S}} \vdash_{\text{type}} (\omega, \rho)}$$

Figure A.12: Static semantics of SEC (boxed types and types)

$\boxed{\overline{\mathcal{S}} \vdash_{\text{rctxt}} \Delta}$

$$\frac{}{\overline{\mathcal{S}} \vdash_{\text{rctxt}} \cdot} \qquad \frac{\overline{\mathcal{S}} \vdash_{\text{rctxt}} \Delta \qquad \varrho \notin dom(\Delta) \qquad \Delta; \overline{\mathcal{S}} \vdash_{\text{eff}} \phi}{\overline{\mathcal{S}} \vdash_{\text{rctxt}} \Delta, \varrho \succeq \phi}$$

$\boxed{\Delta; \overline{\mathcal{S}} \vdash_{\text{vctxt}} \Gamma}$

$$\frac{\overline{\mathcal{S}} \vdash_{\text{rctxt}} \Delta}{\Delta; \overline{\mathcal{S}} \vdash_{\text{vctxt}} \cdot} \qquad \frac{\Delta; \overline{\mathcal{S}} \vdash_{\text{vctxt}} \Gamma \qquad x \notin dom(\Gamma) \qquad \Delta; \overline{\mathcal{S}} \vdash_{\text{type}} \tau}{\Delta; \overline{\mathcal{S}} \vdash_{\text{vctxt}} \Gamma, x{:}\tau}$$

$\boxed{\vdash_{\text{ctxt}} \Delta; \Gamma; \mathcal{S} : \overline{\mathcal{S}}; \pi}$

$$\frac{\vdash_{\text{stype}} \mathcal{S} : \overline{\mathcal{S}} \qquad \Delta; \overline{\mathcal{S}} \vdash_{\text{vctxt}} \Gamma \qquad \Delta; \overline{\mathcal{S}} \vdash_{\text{region}} \pi}{\vdash_{\text{ctxt}} \Delta; \Gamma; \mathcal{S} : \overline{\mathcal{S}}; \pi}$$

Figure A.13: Static semantics of SEC (contexts)

$\boxed{\vdash_{\text{prog}} e}$

$$\frac{\cdot, \mathcal{H} \succeq \{\}; \cdot; \cdot : \cdot \vdash_{\text{exp}} e : \mathsf{Bool}, \mathcal{H}}{\vdash_{\text{prog}} e}$$

Figure A.14: Static semantics of SEC (programs)

We note that stack types and stack domains are purely technical devices that are used to prove type preservation and to support the proof of the correctness of the translation from SEC to $\mathsf{F}^{\mathsf{RGN}}$. In the static semantics, they simply collect the names of live regions and assign types to references. Note that in every rule, stack types and stack domains are passed unmodified to sub-judgments. Since the surface syntax does not admit explicitly named regions, we can type any surface expression with the judgment $\cdot, \mathcal{H} \succeq \{\}; \cdot; \cdot : \cdot \vdash_{\text{exp}} e : \tau, \mathcal{H}$ (as in the judgment $\vdash_{\text{prog}} e$). Pushing these empty stack types and stack domains through the rules leads to the following simplifications:

$\Delta; \Gamma; \mathcal{S} : \overline{\mathcal{S}} \vdash_{\text{exp}} e : \tau, \pi$	$\Longrightarrow$	$\Delta; \Gamma \vdash_{\text{exp}} e : \tau, \pi$
$\Delta; \overline{\mathcal{S}} \vdash_{\text{type}} \tau$	$\Longrightarrow$	$\Delta \vdash_{\text{btype}} \tau$
$\Delta; \overline{\mathcal{S}} \vdash_{\text{btype}} \omega$	$\Longrightarrow$	$\Delta \vdash_{\text{btype}} \omega$
$\Delta; \overline{\mathcal{S}} \vdash_{\text{eff}} \varphi$	$\Longrightarrow$	$\Delta \vdash_{\text{eff}} \varphi$
$\Delta; \overline{\mathcal{S}} \vdash_{\text{region}} \rho$	$\Longrightarrow$	$\Delta \vdash_{\text{region}} \rho$
$\Delta; \overline{\mathcal{S}} \vdash_{\text{re}} \rho \succeq \varphi$	$\Longrightarrow$	$\Delta \vdash_{\text{re}} \rho \succeq \varphi$
$\Delta; \overline{\mathcal{S}} \vdash_{\text{rr}} \rho_2 \succeq \rho_1$	$\Longrightarrow$	$\Delta \vdash_{\text{rr}} \rho_2 \succeq \rho_1$
$\overline{\mathcal{S}} \vdash_{\text{rctxt}} \Delta$	$\Longrightarrow$	$\vdash_{\text{rctxt}} \Delta$
$\Delta; \overline{\mathcal{S}} \vdash_{\text{vctxt}} \Gamma$	$\Longrightarrow$	$\Delta \vdash_{\text{vctxt}} \Gamma$
$\vdash_{\text{ctxt}} \Delta; \Gamma; \mathcal{S} : \overline{\mathcal{S}}; \theta$	$\Longrightarrow$	$\vdash_{\text{ctxt}} \Delta; \Gamma; \theta$

Hence, we recover the type system presented in Section 2.2.5, which is sufficient for type-checking surface programs.

A.2 Type Soundness for SEC

In this section, we sketch a syntactic proof of type soundness [96]. Since our ultimate goal was to demonstrate a type- and semantics-preserving translation

from the Single Effect Calculus to $\mathsf{F}^{\mathsf{RGN}}$, we forgo proving a Progress Theorem (such a proof would be very similar to the Progress Theorem for $\mathsf{F}^{\mathsf{RGN}}$ given in Appendix B.2).

The Preservation Theorem states that the terminating computation of a well-typed expression yields a well-typed extension of the stack and a value of the same type. Various substitution lemmas for dead regions are required to prove the cases where regions are deallocated.

Theorem A.1 (Preservation)

If

(a) $\vdash_{\text{stack}} S : \mathcal{S} : \overline{\mathcal{S}}$,

(b) $\cdot;\cdot;\mathcal{S} : \overline{\mathcal{S}} \vdash_{\text{exp}} e : \tau, \mathfrak{r}$, *and*

(c) $(S; e) \Downarrow (S'; v')$,

then there exists $\overline{\mathcal{S}}' \sqsupseteq \overline{\mathcal{S}}$ *and* $\mathcal{S}' \sqsupseteq \mathcal{S}$ *such that* $\vdash_{\text{stack}} S' : \mathcal{S}' : \overline{\mathcal{S}}'$ *and* $\cdot;\cdot;\mathcal{S}' : \overline{\mathcal{S}}' \vdash_{\text{val}} v' : \tau$.

Proof

By induction on the derivation (c) $(S; e) \Downarrow (S'; v')$.

Full details of this development are given in the technical report *Monadic Regions: Formal Type Soundness and Correctness* [21].

Appendix B

A Monadic Type System: Technical Details

This appendix supplements the material in Chapter 3 with a number of technical details that would otherwise detract from that chapter's focus on the translation from the Single Effect Calculus to $\mathsf{F}^{\mathsf{RGN}}$. In the following section, we revisit the presentation of the $\mathsf{F}^{\mathsf{RGN}}$ language, extending the dynamics to a *natural transition semantics* [84, 76], which models program execution in terms of transitions between *partial derivations*, and revising the static semantics to include judgment for the additional semantic objects introduced by the abstract machine configurations.

In Appendix B.2, we present a syntactic proof of type soundness for $\mathsf{F}^{\mathsf{RGN}}$. We adopt a proof method using natural transition semantics, which allows us to prove type soundness with the familiar progress and preservation theorems, without needing define a small-step operational semantics nor establish its correspondence with the large-step operational semantics. We remark further on this proof method at the end of Appendix B.2.

Finally, in Appendix B.3, we revisit the translation from SEC to $\mathsf{F}^{\mathsf{RGN}}$, extending the translation to accommodate the abstract machine configurations for SEC and expanding upon the proof of translation correctness. Yet more details, including complete proofs, may be found in the technical report *Monadic Regions: Formal Type Soundness and Correctness* [21].

B.1 The $\mathsf{F}^{\mathsf{RGN}}$ Language

B.1.1 Natural Transition Semantics of $\mathsf{F}^{\mathsf{RGN}}$

While the dynamic semantics presented in Section 3.2.2 suffices to describe the complete execution of a program, it cannot describe non-terminating executions or failed executions. To do so, we adopt a *natural transition semantics* [84, 76], which provides a notion of attempted or partial execution. The key idea is to model program execution as a sequence of *partial derivation trees*, which may or may not converge to a complete derivation. The advantage of the natural transition semantics is that it is directly related to the large-step operational semantics of the language, while being capable of describing the evaluation of programs that (a) diverge, (b) terminate with a value, and (c) "get stuck."

Before defining partial derivation trees, we distinguish between *complete judgments* ($(T; e) \Downarrow v$ and $(T, s \mapsto S; \kappa) \Downarrow_\kappa (S'; v)$, introduced in the dynamic semantics) and *pending judgments*, which are judgments of the form $(T; e) \Downarrow ?$ or $(T, \mathfrak{s} \mapsto S; \kappa) \Downarrow_\kappa ?$ and represent expressions and commands that need to be evaluated.

A *partial derivation tree* is an inductively defined structure given by the following grammar:

$$
\begin{array}{lrcl}
\text{Predicates} & P & & \\
\text{Complete derivations } ^\star & \mathfrak{J} & ::= & [(T; e) \Downarrow v] \mid [(T, s \mapsto S; \kappa) \Downarrow_\kappa (S'; v)] \mid P \\
\text{Partial derivation trees} & \mathfrak{D} & ::= & \mathfrak{J} \mid [(T; e) \Downarrow ?]() \mid [(T, s \mapsto S; \kappa) \Downarrow_\kappa ?]() \\
 & & & \mid [(T; e) \Downarrow ?](\mathfrak{J}_1, \ldots, \mathfrak{J}_{k-1}, \mathfrak{D}_k)\ ^\dagger \\
 & & & \mid [(T, s \mapsto S; \kappa) \Downarrow_\kappa ?](\mathfrak{J}_1, \ldots, \mathfrak{J}_{k-1}, \mathfrak{D}_k)\ ^\ddagger
\end{array}
$$

where

⋆ A complete derivation represents the entire derivation tree (comprised of instances of the evaluation rules) that terminates with the eponymous complete judgment.

† There is an instance of an evaluation rule with the form

$$\frac{J_1 \qquad \cdots \qquad J_n}{(T; e) \Downarrow v}$$

where $1 \leq k \leq n$ and

- for $i < k$, $\mathfrak{J}_i \equiv [J_i]$.
- if $J_k \equiv (T; e) \Downarrow v$, then $\mathfrak{D}_k = [(T; e) \Downarrow v]$ or $\mathfrak{D}_k = [(T; e) \Downarrow ?](\ldots)$.
- if $J_k \equiv (T, \mathfrak{s} \mapsto S; \kappa) \Downarrow_\kappa (S'; v)$, then $\mathfrak{D}_k = [(T, \mathfrak{s} \mapsto S; \kappa) \Downarrow_\kappa (S'; v)]$ or $\mathfrak{D}_k = [(T, \mathfrak{s} \mapsto S; \kappa) \Downarrow_\kappa ?](\ldots)$.
- if $J_k \equiv P$, then $\mathfrak{D}_k = P$.

‡ There is an instance of an evaluation rule with the form

$$\frac{J_1 \qquad \cdots \qquad J_n}{(T, s \mapsto S; \kappa) \Downarrow_\kappa (S'; v)}$$

where $1 \leq k \leq n$ and

- for $i < k$, $\mathfrak{J}_i \equiv [J_i]$.
- if $J_k \equiv (T; e) \Downarrow v$, then $\mathfrak{D}_k = [(T; e) \Downarrow v]$ or $\mathfrak{D}_k = [(T; e) \Downarrow ?](\ldots)$.
- if $J_k \equiv (T, \mathfrak{s} \mapsto S; \kappa) \Downarrow_\kappa (S'; v)$, then $\mathfrak{D}_k = [(T, \mathfrak{s} \mapsto S; \kappa) \Downarrow_\kappa (S'; v)]$ or $\mathfrak{D}_k = [(T, \mathfrak{s} \mapsto S; \kappa) \Downarrow_\kappa ?](\ldots)$.
- if $J_k \equiv P$, then $\mathfrak{D}_k = P$.

Note that the definition of a partial derivation tree requires that a node labeled with a pending judgment must have children that are "compatible" with the corresponding complete judgment. Furthermore, each node of a partial derivation tree can have at most one pending judgment amongst its children; the pending judgment must be the rightmost child and the parent node must also be a pending judgment.

Figures B.1 and B.2 gives (a representative sample of) the rules for the natural transition semantics. The rules are derived systematically from the judgments of Figures 3.5–3.8. In addition, there are two "congruence" rules given in Figure B.3. Finally, it should be clear that each transition moves a partial derivation tree "closer" to a complete judgment. Let $\longrightarrow^*$ be the reflexive, transitive closure of the $\longrightarrow$ relation.

The natural transition semantics enjoys soundness and completeness properties demonstrating that it accurately models the dynamic semantics in the case of terminating computations.

Lemma B.1

If $\mathfrak{D}$ is a partial derivation and $\mathfrak{D} \longrightarrow \mathfrak{D}'$, then $\mathfrak{D}'$ is a partial derivation.

Lemma B.2 (NTS Soundness)

(1) If $[(T; e) \Downarrow ?]() \longrightarrow^* \mathfrak{D}'$ and $\mathfrak{D}'$ contains no pending judgments, then $\mathfrak{D}'$ is a complete derivation for a judgment of the form $(T; e) \Downarrow v$ (i.e., $\mathfrak{D}' \equiv \mathfrak{J}' \equiv [(T; e) \Downarrow v]$).

(2) If $[(T, \mathfrak{s} \mapsto S; \kappa) \Downarrow_\kappa ?]() \longrightarrow^* \mathfrak{D}'$ and $\mathfrak{D}'$ contains no pending judgments, then $\mathfrak{D}'$ is a complete derivation for a judgment of the form $(T, \mathfrak{s} \mapsto S; \kappa) \Downarrow_\kappa (S'; v)$ (i.e., $\mathfrak{D}' \equiv \mathfrak{J}' \equiv [(T, \mathfrak{s} \mapsto S; \kappa) \Downarrow_\kappa S'; v]$).

$\boxed{\mathfrak{D} \longrightarrow \mathfrak{D}'}$

$$[(T; \lambda x{:}\tau.\, e) \Downarrow ?]() \longrightarrow \left[\frac{}{(T; \lambda x{:}\tau.\, e) \Downarrow \lambda x{:}\tau.\, e}\right]$$

$$[(T; e_f\ e_a) \Downarrow ?]() \longrightarrow [(T; e_f\ e_a) \Downarrow ?]([(T; e_f) \Downarrow ?]())$$

$$\frac{v_f \equiv \lambda x{:}\tau_x.\, e_b}{\begin{array}{l}[(T; e_f\ e_a) \Downarrow ?]([(T; e_f) \Downarrow v_f]) \longrightarrow \\ \quad [(T; e_f\ e_a) \Downarrow ?]([(T; e_f) \Downarrow v_f], v_f \equiv \lambda x{:}\tau_x.\, e_b)\end{array}}$$

$$\begin{array}{l}[(T; e_f\ e_a) \Downarrow ?]([(T; e_f) \Downarrow v_f], v_f \equiv \lambda x{:}\tau_x.\, e_b) \longrightarrow \\ \quad [(T; e_f\ e_a) \Downarrow ?]([(T; e_f) \Downarrow v_f], v_f \equiv \lambda x{:}\tau_x.\, e_b, [(T; e_a) \Downarrow ?]())\end{array}$$

$$\begin{array}{l}[(T; e_f\ e_a) \Downarrow ?]([(T; e_f) \Downarrow v_f], v_f \equiv \lambda x{:}\tau_x.\, e_b, [(T; e_a) \Downarrow v_a]) \longrightarrow \\ \qquad [(T; e_f\ e_a) \Downarrow ?] \left(\begin{array}{l}[(T; e_f) \Downarrow v_f], v_f \equiv \lambda x{:}\tau_x.\, e_b, \\ [(T; e_a) \Downarrow v_a], [(T; e_b[v_a/x]) \Downarrow ?]()\end{array}\right)\end{array}$$

$$\begin{array}{l}[(T; e_f\ e_a) \Downarrow ?] \left(\begin{array}{l}[(T; e_f) \Downarrow v_f], v_f \equiv \lambda x{:}\tau_x.\, e_b, \\ [(T; e_a) \Downarrow v_a], [(T; e_b[v_a/x]) \Downarrow v]\end{array}\right) \longrightarrow \\ \left[\dfrac{(T; e_f) \Downarrow v_f \qquad v_f \equiv \lambda x{:}\tau_x.\, e_b \qquad (T; e_a) \Downarrow v_a \qquad (T; e_b[v_a/x]) \Downarrow v}{(T; e_f\ e_a) \Downarrow v}\right]\end{array}$$

Figure B.1: Natural transition semantics of $\mathsf{F}^{\mathsf{RGN}}$ (abbreviated (I))

$\boxed{\mathfrak{D} \longrightarrow \mathfrak{D}'}$

$$\frac{\mathfrak{s} \notin dom(T)}{[(T; \texttt{runRGN}\ [\tau]\ v) \Downarrow ?]() \longrightarrow [(T; \texttt{runRGN}\ [\tau]\ v) \Downarrow ?](\mathfrak{s} \notin dom(T))}$$

$$\frac{\mathfrak{r} \notin dom(\cdot)}{\begin{array}{l}[(T; \texttt{runRGN}\ [\tau]\ v) \Downarrow ?](\mathfrak{s} \notin dom(T)) \longrightarrow \\ \quad [(T; \texttt{runRGN}\ [\tau]\ v) \Downarrow ?](\mathfrak{s} \notin dom(T), \mathfrak{r} \notin dom(\cdot))\end{array}}$$

$$[(T; \texttt{runRGN}\ [\tau]\ v) \Downarrow ?] \left(\begin{array}{l}\mathfrak{s} \notin dom(T), \mathfrak{r} \notin dom(\cdot), \\ [(T, \mathfrak{s} \mapsto (\cdot, \mathfrak{r} \mapsto \{\}); v\ [\mathfrak{s}\sharp\mathfrak{r}]\ (\texttt{hnd}\ \mathfrak{s}\sharp\mathfrak{r})) \Downarrow v'], v' \equiv \kappa', \\ [(T, \mathfrak{s} \mapsto (\cdot, \mathfrak{r} \mapsto \{\}); \kappa') \Downarrow_\kappa (S''; v'')], S'' \equiv \cdot, \mathfrak{r} \mapsto R''\end{array}\right) \longrightarrow$$

$$\left[\begin{array}{c}\mathfrak{s} \notin dom(T) \qquad \mathfrak{r} \notin dom(\cdot) \\ (T, \mathfrak{s} \mapsto (\cdot, \mathfrak{r} \mapsto \{\}); v\ [\mathfrak{s}\sharp\mathfrak{r}]\ (\texttt{hnd}\ \mathfrak{s}\sharp\mathfrak{r})) \Downarrow v' \\ \hline v' \equiv \kappa' \qquad (T, \mathfrak{s} \mapsto (\cdot, \mathfrak{r} \mapsto \{\}); \kappa') \Downarrow_\kappa S''; v'' \qquad S'' \equiv \cdot, \mathfrak{r} \mapsto R'' \\ \hline (T; \texttt{runRGN}\ [\tau]\ v) \Downarrow v''[\circ\sharp\bullet/\mathfrak{s}\sharp\mathfrak{r}]\end{array}\right]$$

Figure B.2: Natural transition semantics of $\mathsf{F}^{\mathsf{RGN}}$ (abbreviated (II))

$\boxed{\mathfrak{D} \longrightarrow \mathfrak{D}'}$

$$\frac{\mathfrak{D} \longrightarrow \mathfrak{D}'}{\begin{array}{l}[(T; e) \Downarrow ?](\mathfrak{J}_1, \ldots, \mathfrak{J}_k, \mathfrak{D}) \longrightarrow \\ \quad [(T; e) \Downarrow ?](\mathfrak{J}_1, \ldots, \mathfrak{J}_k, \mathfrak{D}')\end{array}} \qquad \frac{\mathfrak{D} \longrightarrow \mathfrak{D}'}{\begin{array}{l}[(T, s \mapsto S; \kappa) \Downarrow_\kappa ?](\mathfrak{J}_1, \ldots, \mathfrak{J}_k, \mathfrak{D}) \longrightarrow \\ \quad [(T, s \mapsto S; \kappa) \Downarrow_\kappa ?](\mathfrak{J}_1, \ldots, \mathfrak{J}_k, \mathfrak{D}')\end{array}}$$

Figure B.3: Natural transition semantics of $\mathsf{F}^{\mathsf{RGN}}$ (congruence)

Lemma B.3 (NTS Completeness)

(1) If $(T; e) \Downarrow v$ *and* $\mathfrak{D}$ *is a complete derivation for* $(T; e) \Downarrow v$*, then* $[(T; e) \Downarrow ?]() \longrightarrow^* \mathfrak{D}$.

(2) If $(T, \mathfrak{s} \mapsto S; \kappa) \Downarrow_\kappa (S'; v)$ *and* $\mathfrak{D}_\kappa$ *is a complete derivation for* $(T, \mathfrak{s} \mapsto S; \kappa) \Downarrow_\kappa (S'; v)$*, then* $[(T, \mathfrak{s} \mapsto S; \kappa) \Downarrow_\kappa ?]() \longrightarrow^* \mathfrak{D}_\kappa$.

For each tower T and expression e, we define an execution of e in T as a sequence

$$[(T; e) \Downarrow ?]() \longrightarrow \mathfrak{D}_1 \longrightarrow \mathfrak{D}_2 \longrightarrow \cdots$$

Thus, an execution has three possibilities:

(1) Suppose that for all $\mathfrak{D}_n$ such that $[(T; e) \Downarrow ?]() \longrightarrow^* \mathfrak{D}_n$, there exists $\mathfrak{D}_{n+1}$ such that $\mathfrak{D}_n \longrightarrow \mathfrak{D}_{n+1}$. Then, we say that e in T diverges.

(2) Suppose that there exists $\mathfrak{D}_n$ such that $[(T; e) \Downarrow ?]() \longrightarrow^* \mathfrak{D}_n$, such that there does not exist $\mathfrak{D}_{n+1}$ such that $\mathfrak{D}_n \longrightarrow \mathfrak{D}_{n+1}$.

   (a) Suppose $\mathfrak{D}_n$ contains no pending judgments. By Lemma B.2, $\mathfrak{D}_n \equiv [(T; e) \Downarrow v]$. Then, we say that e in T terminates with the value v.

   (b) Suppose $\mathfrak{D}_n$ contains pending judgments. Then, we say that e in T gets stuck.

By inspection of the rules in Figures B.1–B.3, it is clear that the stuck partial derivation trees correspond to trees in which predicates cannot be satisfied; all other transitions are unrestricted. Predicates like $v \equiv \lambda x{:}\tau.\, e$ and $v \equiv \kappa$ are traditional type errors, where expressions evaluate to values of the wrong form.

Predicates like $\mathfrak{s} \in dom(T)$ also correspond to type errors, where towers have the wrong form. The static semantics in Section 3.2.3 and the next section ensure that stuck partial derivation trees are not well-typed.

B.1.2 Static Semantics of $\mathsf{F}^{\mathsf{RGN}}$

Section 3.2.3 gave the static semantics for the surface syntax of $\mathsf{F}^{\mathsf{RGN}}$. However, the judgments given in that section are insufficient for carrying out a syntactic proof of type soundness, since there are no rules for `ref`, `hnd`, or `witnessRGN` terms (which arise during the evaluation of a program) and there is no typing judgment for towers. This section extends the static semantics of Section 3.2.3 to overcome these deficiencies. In addition to the typing judgments for expressions and various well-formedness judgments for types, indices, and contexts given previously, we have judgments that check the type and well-formedness of towers and the consistency of $\mathsf{F}^{\mathsf{RGN}}$ witness terms.

Definitions Figure B.4 presents additional definitions for syntactic objects that appear in the static semantics. Tower, stack, and region types mimic the structure of towers, stacks, and regions; they record the type of the value stored at each pointer. Tower, stack, and region domains are technical devices that record the live stack, region, and pointer names. Because proving the well-formedness of tower, stack, and region types requires proving the well-formedness of types, which requires verifying that stack and region names are live, one cannot easily have tower, stack, and region types serve the dual purpose of recording live names. We tacitly assume that all domains are well-formed – containing distinct stack names, region names, and pointer names.

$$
\begin{array}{lrcl}
\text{Type and index contexts} & \Delta & ::= & \cdot \mid \Delta, \alpha \mid \Delta, \vartheta \\
\text{Value contexts} & \Gamma & ::= & \cdot \mid \Gamma, x{:}\tau \\
\text{Region domains} & \overline{\mathcal{R}} & ::= & \{\mathfrak{p}_1, \ldots, \mathfrak{p}_n\} \\
\text{Region types} & \mathcal{R} & ::= & \{\mathfrak{p}_1 \mapsto \tau_1, \ldots, \mathfrak{p}_n \mapsto \tau_n\} \\
\text{Stack domains} & \overline{\mathcal{S}} & ::= & \cdot \mid \overline{\mathcal{S}}, \mathfrak{r} \mapsto \overline{\mathcal{R}} \quad \text{(ordered domain)} \\
\text{Stack types} & \mathcal{S} & ::= & \cdot \mid \mathcal{S}, \mathfrak{r} \mapsto \mathcal{R} \quad \text{(ordered domain)} \\
\text{Tower domains} & \overline{\mathcal{T}} & ::= & \cdot \mid \overline{\mathcal{T}}, \mathfrak{s} \mapsto \overline{\mathcal{S}} \quad \text{(ordered domain)} \\
\text{Tower types} & \mathcal{T} & ::= & \cdot \mid \mathcal{T}, \mathfrak{s} \mapsto \mathcal{S} \quad \text{(ordered domain)}
\end{array}
$$

$$
\mathsf{RGNPf}(\theta_1 \preceq \theta_2) \equiv \forall \beta.\, \mathsf{RGN}\ \theta_1\ \beta \to \mathsf{RGN}\ \theta_2\ \beta
$$

$$
\begin{array}{rcl}
\overline{\mathcal{S}} \sqsupseteq \overline{\mathcal{S}}' & \equiv & dom(\overline{\mathcal{S}}) = dom(\overline{\mathcal{S}}') \wedge \\
 & & \quad \forall \mathfrak{r} \in dom(\overline{\mathcal{S}}').\ dom(\overline{\mathcal{S}}(\mathfrak{r})) \supseteq dom(\overline{\mathcal{S}}'(\mathfrak{r}))
\end{array}
$$

$$
\overline{\mathcal{T}}|_{\mathfrak{s}} \equiv \overline{\mathcal{T}}', \mathfrak{s} \mapsto \overline{\mathcal{S}}' \quad \text{such that } \overline{\mathcal{T}} \equiv \overline{\mathcal{T}}', \mathfrak{s} \mapsto \overline{\mathcal{S}}', \overline{\mathcal{T}}''
$$

$$
\begin{array}{rcl}
\mathcal{S} \sqsupseteq \mathcal{S}' & \equiv & dom(\mathcal{S}) = dom(\mathcal{S}') \wedge \\
 & & \quad \forall \mathfrak{r} \in dom(\mathcal{S}').\ dom(\mathcal{S}(\mathfrak{r})) \supseteq dom(\mathcal{S}'(\mathfrak{r})) \wedge \\
 & & \quad\quad \forall \mathfrak{p} \in dom(\mathcal{S}'(\mathfrak{r}))\ldots\ \mathcal{S}(\mathfrak{r}, \mathfrak{p}) = \mathcal{S}'(\mathfrak{r}, \mathfrak{p})
\end{array}
$$

$$
\mathcal{T}|_{\mathfrak{s}} \equiv \mathcal{T}', \mathfrak{s} \mapsto \mathcal{S}' \quad \text{such that } \mathcal{T} \equiv \mathcal{T}', \mathfrak{s} \mapsto \mathcal{S}', \mathcal{T}''
$$

Figure B.4: Static semantics of $\mathsf{F}^{\mathsf{RGN}}$ (definitions)

$\boxed{\Delta; \Gamma; \mathcal{T} : \overline{\mathcal{T}} \vdash_{\text{exp}} e : \tau}$

$$\frac{\vdash_{\text{ctxt}} \Delta; \Gamma; \mathcal{T} : \overline{\mathcal{T}} \qquad x \in dom(\Gamma) \qquad \Gamma(x) = \tau}{\Delta; \Gamma; \mathcal{T} : \overline{\mathcal{T}} \vdash_{\text{exp}} x : \tau} \qquad \frac{\Delta; \Gamma, x{:}\tau_x; \mathcal{T} : \overline{\mathcal{T}} \vdash_{\text{exp}} e : \tau}{\Delta; \Gamma; \mathcal{T} : \overline{\mathcal{T}} \vdash_{\text{exp}} \lambda x{:}\tau_x.\, e : \tau_x \to \tau}$$

$$\frac{\Delta; \Gamma; \mathcal{T} : \overline{\mathcal{T}} \vdash_{\text{exp}} e_f : \tau_x \to \tau \qquad \Delta; \Gamma; \mathcal{T} : \overline{\mathcal{T}} \vdash_{\text{exp}} e_a : \tau_x}{\Delta; \Gamma; \mathcal{T} : \overline{\mathcal{T}} \vdash_{\text{exp}} e_f\ e_a : \tau} \qquad \frac{\vdash_{\text{ctxt}} \Delta; \Gamma; \mathcal{T} : \overline{\mathcal{T}} \qquad \Delta; \Gamma; \mathcal{T} : \overline{\mathcal{T}} \vdash_{\text{exp}} e_i : \tau_i \quad {}^{i \in 1 \ldots n}}{\Delta; \Gamma; \mathcal{T} : \overline{\mathcal{T}} \vdash_{\text{exp}} \langle e_1, \ldots, e_n \rangle : \tau_1 \times \cdots \times \tau_n}$$

$$\frac{\Delta; \Gamma; \mathcal{T} : \overline{\mathcal{T}} \vdash_{\text{exp}} e : \tau_1 \times \cdots \times \tau_n \qquad 1 \le i \le n}{\Delta; \Gamma; \mathcal{T} : \overline{\mathcal{T}} \vdash_{\text{exp}} \mathtt{sel}_i\ e : \tau_i} \qquad \frac{\vdash_{\text{ctxt}} \Delta; \Gamma; \mathcal{T} : \overline{\mathcal{T}} \qquad \Delta, \alpha; \Gamma; \mathcal{T} : \overline{\mathcal{T}} \vdash_{\text{exp}} e : \tau}{\Delta; \Gamma; \mathcal{T} : \overline{\mathcal{T}} \vdash_{\text{exp}} \Lambda\alpha.\, e : \forall\alpha.\, \tau}$$

$$\frac{\Delta; \Gamma; \mathcal{T} : \overline{\mathcal{T}} \vdash_{\text{exp}} e_f : \forall\alpha.\, \tau \qquad \Delta; \overline{\mathcal{T}} \vdash_{\text{type}} \tau_a}{\Delta; \Gamma; \mathcal{T} : \overline{\mathcal{T}} \vdash_{\text{exp}} e_f\ [\tau_a] : \tau[\tau_a/\alpha]} \qquad \frac{\Delta; \Gamma; \mathcal{T} : \overline{\mathcal{T}} \vdash_{\text{exp}} e_a : \tau_x \qquad \Delta; \Gamma, x{:}\tau_x; \mathcal{T} : \overline{\mathcal{T}} \vdash_{\text{exp}} e_b : \tau}{\Delta; \Gamma; \mathcal{T} : \overline{\mathcal{T}} \vdash_{\text{exp}} \mathtt{let}\ x = e_a\ \mathtt{in}\ e_b : \tau}$$

Figure B.5: Static semantics of $\mathsf{F}^{\mathsf{RGN}}$ (expressions (II revised))

We define the relation $\sqsupseteq$ to describe extensions of stack domains and types. Note that we consider tower and stack domains and types to have ordered domains. Hence, $dom(S) \sqsupseteq dom(S')$ indicates that the ordered domain of $dom(S')$ is a prefix of the ordered domain of $dom(S)$. Finally, we define restriction operators, which return a prefix of tower domains and types.

Terms Figures B.5–B.8 present the typing rules for the judgment $\Delta; \Gamma; \mathcal{T} : \overline{\mathcal{T}} \vdash_{\text{exp}} e : \tau$, which asserts that under the type and index context Δ, the value context Γ, and the tower type $\mathcal{T}$ with the tower domain $\overline{\mathcal{T}}$, the expression e has the type

$\boxed{\Delta; \Gamma; \mathcal{T} : \overline{\mathcal{T}} \vdash_{\text{exp}} e : \tau}$

$$\frac{\Delta; \overline{\mathcal{T}} \vdash_{\text{index}} \theta \qquad \Delta; \overline{\mathcal{T}} \vdash_{\text{type}} \tau \qquad \Delta; \Gamma; \mathcal{T} : \overline{\mathcal{T}} \vdash_{\text{exp}} v : \tau}{\Delta; \Gamma; \mathcal{T} : \overline{\mathcal{T}} \vdash_{\text{exp}} \texttt{returnRGN}\ [\theta]\ [\tau]\ v : \mathsf{RGN}\ \theta\ \tau}$$

$$\frac{\Delta; \overline{\mathcal{T}} \vdash_{\text{index}} \theta \qquad \Delta; \overline{\mathcal{T}} \vdash_{\text{type}} \tau_a \qquad \Delta; \overline{\mathcal{T}} \vdash_{\text{type}} \tau_b \qquad \Delta; \Gamma; \mathcal{T} : \overline{\mathcal{T}} \vdash_{\text{exp}} v_a : \mathsf{RGN}\ \theta\ \tau_a \qquad \Delta; \Gamma; \mathcal{T} : \overline{\mathcal{T}} \vdash_{\text{exp}} v_f : \tau_a \to \mathsf{RGN}\ \theta\ \tau_b}{\Delta; \Gamma; \mathcal{T} : \overline{\mathcal{T}} \vdash_{\text{exp}} \texttt{thenRGN}\ [\theta]\ [\tau_a]\ [\tau_b]\ v_a\ v_f : \mathsf{RGN}\ \theta\ \tau_b}$$

$$\frac{\Delta; \overline{\mathcal{T}} \vdash_{\text{index}} \theta_1 \qquad \Delta; \overline{\mathcal{T}} \vdash_{\text{type}} \tau \qquad \Delta; \Gamma; \mathcal{T} : \overline{\mathcal{T}} \vdash_{\text{exp}} v : \forall \vartheta.\, \mathsf{RGNPf}(\theta_1 \preceq \vartheta_2) \to \mathsf{RGNHnd}\ \vartheta_2 \to \mathsf{RGN}\ \vartheta_2\ \tau}{\Delta; \Gamma; \mathcal{T} : \overline{\mathcal{T}} \vdash_{\text{exp}} \texttt{letRGN}\ [\theta_1]\ [\tau]\ v : \mathsf{RGN}\ \theta_1\ \tau}$$

Figure B.6: Static semantics of $\mathsf{F}^{\mathsf{RGN}}$ (commands (I revised))

$\boxed{\Delta; \Gamma; \mathcal{T} : \overline{\mathcal{T}} \vdash_{\text{exp}} e : \tau}$

$$\frac{\vdash_{\text{ctxt}} \Delta; \Gamma; \mathcal{T} : \overline{\mathcal{T}} \qquad \Delta; \overline{\mathcal{T}} \vdash_{\text{type}} \tau}{\Delta; \Gamma; \mathcal{T} : \overline{\mathcal{T}} \vdash_{\text{exp}} \texttt{ref}\ \circ\sharp\bullet\ \mathfrak{p} : \mathsf{RGNRef}\ \circ\sharp\bullet\ \tau}$$

$$\frac{\vdash_{\text{ctxt}} \Delta; \Gamma; \mathcal{T} : \overline{\mathcal{T}} \qquad \mathfrak{s} \in \overline{\mathcal{T}} \qquad \Delta; \overline{\mathcal{T}} \vdash_{\text{type}} \tau}{\Delta; \Gamma; \mathcal{T} : \overline{\mathcal{T}} \vdash_{\text{exp}} \texttt{ref}\ \mathfrak{s}\sharp\bullet\ \mathfrak{p} : \mathsf{RGNRef}\ \mathfrak{s}\sharp\bullet\ \tau}$$

$$\frac{\vdash_{\text{ctxt}} \Delta; \Gamma; \mathcal{T} : \overline{\mathcal{T}} \qquad \mathfrak{s} \in \overline{\mathcal{T}} \qquad \mathfrak{r} \in \overline{\mathcal{T}}(\mathfrak{s}) \qquad \mathfrak{p} \in \overline{\mathcal{T}}(\mathfrak{s}, \mathfrak{r}) \qquad \mathcal{T}(\mathfrak{s}, \mathfrak{r}, \mathfrak{p}) = \tau}{\Delta; \Gamma; \mathcal{T} : \overline{\mathcal{T}} \vdash_{\text{exp}} \texttt{ref}\ \mathfrak{s}\sharp\mathfrak{r}\ \mathfrak{p} : \mathsf{RGNRef}\ \mathfrak{s}\sharp\mathfrak{r}\ \tau}$$

$$\frac{\vdash_{\text{ctxt}} \Delta; \Gamma; \mathcal{T} : \overline{\mathcal{T}}}{\Delta; \Gamma; \mathcal{T} : \overline{\mathcal{T}} \vdash_{\text{exp}} \texttt{hnd}\ \circ\sharp\bullet : \mathsf{RGNHnd}\ \circ\sharp\bullet} \qquad \frac{\vdash_{\text{ctxt}} \Delta; \Gamma; \mathcal{T} : \overline{\mathcal{T}} \qquad \mathfrak{s} \in \overline{\mathcal{T}}}{\Delta; \Gamma; \mathcal{T} : \overline{\mathcal{T}} \vdash_{\text{exp}} \texttt{hnd}\ \mathfrak{s}\sharp\bullet : \mathsf{RGNHnd}\ \mathfrak{s}\sharp\bullet}$$

$$\frac{\vdash_{\text{ctxt}} \Delta; \Gamma; \mathcal{T} : \overline{\mathcal{T}} \qquad \mathfrak{s} \in \overline{\mathcal{T}} \qquad \mathfrak{r} \in \overline{\mathcal{T}}(\mathfrak{s})}{\Delta; \Gamma; \mathcal{T} : \overline{\mathcal{T}} \vdash_{\text{exp}} \texttt{hnd}\ \mathfrak{s}\sharp\mathfrak{r} : \mathsf{RGNHnd}\ \mathfrak{s}\sharp\mathfrak{r}}$$

Figure B.7: Static semantics of $\mathsf{F}^{\mathsf{RGN}}$ (references and handles)

τ. Figures B.5 and B.6 repeat the rules from Figures 3.11 and 3.13 in order to demonstrate the manner in which tower types and tower domains are propagated through the rules given in Section 3.2.3; we elide the rules for other expression and command forms given in Section 3.2.3. We note that in every rule, tower types and tower domains are passed unmodified to sub-judgments.

Figure B.7 gives typing rules for reference and handle expressions. The rules ensure that stack and region names that appear in references and handles are in scope; furthermore, a pointer in a live stack and region denotes a value with the type assigned by the tower type.

$\boxed{\Delta; \Gamma; \mathcal{T} : \overline{\mathcal{T}} \vdash_{\text{exp}} e : \tau}$

$$\frac{\Delta; \overline{\mathcal{T}} \vdash_{\text{type}} \tau \qquad \Delta; \Gamma; \mathcal{T} : \overline{\mathcal{T}} \vdash_{\text{exp}} v : \mathsf{RGN}\ s\sharp r_1\ \tau \qquad \overline{\mathcal{T}} \vdash_{\text{cast}} s\sharp r_1 \rightsquigarrow s\sharp r_2}{\Delta; \Gamma; \mathcal{T} : \overline{\mathcal{T}} \vdash_{\text{exp}} \texttt{witnessRGN}\ s\sharp r_1\ s\sharp r_2\ [\tau]\ v : \mathsf{RGN}\ s\sharp r_2\ \tau}$$

Figure B.8: Static semantics of $\mathsf{F}^{\mathsf{RGN}}$ (commands (witness))

$\boxed{\overline{\mathcal{T}} \vdash_{\text{cast}} s\sharp r \rightsquigarrow s\sharp r'}$

$$\frac{}{\overline{\mathcal{T}} \vdash_{\text{cast}} \circ\sharp\bullet \rightsquigarrow \circ\sharp\bullet} \qquad \frac{\mathfrak{s} \in \overline{\mathcal{T}}}{\overline{\mathcal{T}} \vdash_{\text{cast}} \mathfrak{s}\sharp\bullet \rightsquigarrow \mathfrak{s}\sharp\bullet} \qquad \frac{\mathfrak{s} \in \overline{\mathcal{T}} \qquad \overline{\mathcal{T}}(\mathfrak{s}) \equiv \overline{\mathcal{S}}_1, \mathfrak{r}_1 \mapsto \overline{\mathcal{R}}_1, \overline{\mathcal{S}}_2}{\overline{\mathcal{T}} \vdash_{\text{cast}} \mathfrak{s}\sharp\mathfrak{r}_1 \rightsquigarrow \mathfrak{s}\sharp\bullet}$$

$$\frac{\mathfrak{s} \in \overline{\mathcal{T}} \qquad \overline{\mathcal{T}}(\mathfrak{s}) \equiv \overline{\mathcal{S}}_1, \mathfrak{r}_1 \mapsto \overline{\mathcal{R}}_1, \overline{\mathcal{S}}_2, \mathfrak{r}_2 \mapsto \overline{\mathcal{R}}_2, \overline{\mathcal{S}}_3}{\overline{\mathcal{T}} \vdash_{\text{cast}} \mathfrak{s}\sharp\mathfrak{r}_1 \rightsquigarrow \mathfrak{s}\sharp\mathfrak{r}_2}$$

Figure B.9: Static semantics of $\mathsf{F}^{\mathsf{RGN}}$ (casts)

Recall that the typing rule for `letRGN` requires that its function argument accepts a witness argument of type $\mathsf{RGNPf}(\theta_1 \preceq \vartheta_2)$. The witness argument is provided to the computation taking place in the stack with the inner/younger region allocated in order to coerce computations (such as allocating a new value in some outer/older region) from a computation indexed by the outer region to a computation indexed by the the inner region. This coercion is safe because every region in the stack denoted by θ_1 outlives every region in the stack denoted by ϑ_2. Operationally, such a witness function acts as the identity function.

The typing rule for `witnessRGN` in Figure B.8 formalizes this outlives argument: a `witnessRGN` term is well-typed whenever $s\sharp r_1$ can be cast to $s\sharp r_2$. The judgment

$\boxed{\vdash_{\text{ttype}} \mathcal{T} : \overline{\mathcal{T}}}$

$$\frac{\begin{array}{c}\overline{\mathcal{T}} = dom(\mathcal{T}) \\ \forall \mathfrak{s} \in \overline{\mathcal{T}}.\ \overline{\mathcal{T}}(\mathfrak{s}) = dom(\mathcal{T}(\mathfrak{s})) \\ \forall \mathfrak{s} \in \overline{\mathcal{T}}.\ \forall \mathfrak{r} \in \overline{\mathcal{T}}(\mathfrak{s}).\ \overline{\mathcal{T}}(\mathfrak{s}, \mathfrak{r}) = dom(\mathcal{T}(\mathfrak{s}, \mathfrak{r})) \\ \forall \mathfrak{s} \in \overline{\mathcal{T}}. \forall \mathfrak{r} \in \overline{\mathcal{T}}(\mathfrak{s}). \forall \mathfrak{p} \in \overline{\mathcal{T}}(\mathfrak{s}, \mathfrak{r}).\ \cdot; \overline{\mathcal{T}}|_{\mathfrak{s}} \vdash_{\text{type}} \mathcal{T}(\mathfrak{s}, \mathfrak{r}, \mathfrak{p})\end{array}}{\vdash_{\text{ttype}} \mathcal{T} : \overline{\mathcal{T}}}$$

Figure B.10: Static semantics of $\mathsf{F}^{\mathsf{RGN}}$ (tower types)

$\boxed{\vdash_{\text{tower}} T : \mathcal{T} : \overline{\mathcal{T}}}$

$$\frac{\begin{array}{c}\vdash_{\text{ttype}} \mathcal{T} : \overline{\mathcal{T}} \\ \overline{\mathcal{T}} = dom(\mathcal{T}) = dom(T) \\ \forall \mathfrak{s} \in \overline{\mathcal{T}}.\ \overline{\mathcal{T}}(\mathfrak{s}) = dom(\mathcal{T}(\mathfrak{s})) = dom(T(\mathfrak{s})) \\ \forall \mathfrak{s} \in \overline{\mathcal{T}}.\ \forall \mathfrak{r} \in \overline{\mathcal{T}}(\mathfrak{s}).\ \overline{\mathcal{T}}(\mathfrak{s}, \mathfrak{r}) = dom(\mathcal{T}(\mathfrak{s}, \mathfrak{r})) = dom(T(\mathfrak{s}, \mathfrak{r})) \\ \forall \mathfrak{s} \in \overline{\mathcal{T}}.\ \forall \mathfrak{r} \in \overline{\mathcal{T}}(\mathfrak{s}).\ \forall \mathfrak{p} \in \overline{\mathcal{T}}(\mathfrak{s}, \mathfrak{r}).\ \cdot; \cdot; \mathcal{T}|_{\mathfrak{s}} : \overline{\mathcal{T}}|_{\mathfrak{s}} \vdash_{\text{exp}} T(\mathfrak{s}, \mathfrak{r}, \mathfrak{p}) : \mathcal{T}(\mathfrak{s}, \mathfrak{r}, \mathfrak{p})\end{array}}{\vdash_{\text{tower}} T : \mathcal{T} : \overline{\mathcal{T}}}$$

Figure B.11: Static semantics of $\mathsf{F}^{\mathsf{RGN}}$ (towers)

$\overline{\mathcal{T}} \vdash_{\text{cast}} s \sharp r_1 \rightsquigarrow s \sharp r_2$ in Figure B.9 verifies the casts witnessed by `witnessRGN` terms. Note that the judgment $\overline{\mathcal{T}} \vdash_{\text{cast}} \mathfrak{s} \sharp \mathfrak{r}_1 \rightsquigarrow \mathfrak{s} \sharp \mathfrak{r}_2$ enforces the requirement that $\mathfrak{r}_1$ outlives $\mathfrak{r}_2$ in the stack $\mathfrak{s}$. The other $\vdash_{\text{cast}}$ judgments allow casts to deallocated regions, which can be introduced when deallocating a region at the end of a `runRGN` or `letRGN` computation. This is a technicality needed to ensure that programs remain closed and well-typed during their execution.

Towers Figures B.10 and B.11 presents typing rules that check the type and well-formedness of towers. The judgment $\vdash_{\text{ttype}} \mathcal{T} : \overline{\mathcal{T}}$ asserts that tower type $\mathcal{T}$ is well-formed with tower domain $\overline{\mathcal{T}}$. In particular, the judgment asserts that $\mathcal{T}$ has the domain specified by $\overline{\mathcal{T}}$ and each type in the range of $\mathcal{T}$ is well-formed. Note the use of the restriction operator; this ensures that types "lower" in the tower cannot reference stack and region names that appear "higher" in the tower. This corresponds to the fact that while `runRGN` computations can be nested, the inner computation must complete before executing a command in the outer computation. Hence, while an inner computation may have references to the outer computation, there can be no references from the outer computation to the inner computation. Finally, the judgment $\vdash_{\text{tower}} T : \mathcal{T} : \overline{\mathcal{T}}$ asserts that the tower T is well-formed with tower type $\mathcal{T}$ and tower domain $\overline{\mathcal{T}}$. Like the judgment $\vdash_{\text{ttype}}$, it asserts that T has the domain specified by $\overline{\mathcal{T}}$ and each value in the range of T has the type specified by $\mathcal{T}$. Again, restriction operators are used to assert that values "lower" in the tower cannot contain references to names "higher" in the tower.

Types, indices, and contexts Figures B.12 and B.13 contain additional judgments for ensuring that types τ, indices θ, and value contexts Γ are well-formed. Because types and indices may contain stack and region names, the judgments $\vdash_{\text{index}}$, $\vdash_{\text{type}}$, and $\vdash_{\text{vctxt}}$ require a tower domain $\overline{\mathcal{T}}$.

Surface syntax We note that tower types and tower domains are purely technical devices that are used to prove type soundness and to support the proof of the correctness of the translation from SEC to $\mathsf{F}^{\mathsf{RGN}}$. In the static semantics, they simply collect the names of live stacks and regions and assign types to references. Note that in every rule, tower types and tower domains are passed unmod-

$\boxed{\Delta; \overline{\mathcal{T}} \vdash_{\text{type}} \tau}$

$$\frac{\alpha \in dom(\Delta)}{\Delta; \overline{\mathcal{T}} \vdash_{\text{type}} \alpha} \qquad \frac{}{\Delta; \overline{\mathcal{T}} \vdash_{\text{type}} \mathsf{Int}} \qquad \frac{}{\Delta; \overline{\mathcal{T}} \vdash_{\text{type}} \mathsf{Bool}}$$

$$\frac{\Delta; \overline{\mathcal{T}} \vdash_{\text{type}} \tau_1 \qquad \Delta; \overline{\mathcal{T}} \vdash_{\text{type}} \tau_2}{\Delta; \overline{\mathcal{T}} \vdash_{\text{type}} \tau_1 \to \tau_2} \qquad \frac{\Delta; \overline{\mathcal{T}} \vdash_{\text{type}} \tau_i \quad {}^{i \in 1 \ldots n}}{\Delta; \overline{\mathcal{T}} \vdash_{\text{type}} \tau_1 \times \cdots \times \tau_n} \qquad \frac{\Delta, \alpha; \overline{\mathcal{T}} \vdash_{\text{type}} \tau}{\Delta; \overline{\mathcal{T}} \vdash_{\text{type}} \forall \alpha. \tau}$$

$$\frac{\Delta; \overline{\mathcal{T}} \vdash_{\text{index}} \theta \qquad \Delta; \overline{\mathcal{T}} \vdash_{\text{type}} \tau}{\Delta; \overline{\mathcal{T}} \vdash_{\text{type}} \mathsf{RGN}\ \theta\ \tau} \qquad \frac{\Delta; \overline{\mathcal{T}} \vdash_{\text{index}} \theta \qquad \Delta; \overline{\mathcal{T}} \vdash_{\text{type}} \tau}{\Delta; \overline{\mathcal{T}} \vdash_{\text{type}} \mathsf{RGNRef}\ \theta\ \tau}$$

$$\frac{\Delta; \overline{\mathcal{T}} \vdash_{\text{index}} \theta}{\Delta; \overline{\mathcal{T}} \vdash_{\text{type}} \mathsf{RGNHnd}\ \theta} \qquad \frac{\Delta, \vartheta; \overline{\mathcal{T}} \vdash_{\text{type}} \tau}{\Delta; \overline{\mathcal{T}} \vdash_{\text{type}} \forall \vartheta. \tau}$$

$\boxed{\Delta; \overline{\mathcal{T}} \vdash_{\text{index}} \theta}$

$$\frac{\vartheta \in dom(\Delta)}{\Delta; \overline{\mathcal{T}} \vdash_{\text{index}} \vartheta} \qquad \frac{}{\Delta; \overline{\mathcal{T}} \vdash_{\text{index}} \circ \sharp \bullet} \qquad \frac{\mathfrak{s} \in \overline{\mathcal{T}}}{\Delta; \overline{\mathcal{T}} \vdash_{\text{index}} \mathfrak{s} \sharp \bullet} \qquad \frac{\mathfrak{s} \in \overline{\mathcal{T}} \qquad \mathfrak{r} \in \overline{\mathcal{T}}(\mathfrak{s})}{\Delta; \overline{\mathcal{T}} \vdash_{\text{index}} \mathfrak{s} \sharp \mathfrak{r}}$$

Figure B.12: Static semantics of $\mathsf{F}^{\mathsf{RGN}}$ (types and indices)

$\boxed{\Delta; \overline{\mathcal{T}} \vdash_{\text{vctxt}} \Gamma}$

$$\frac{}{\Delta; \overline{\mathcal{T}} \vdash_{\text{vctxt}} \cdot} \qquad \frac{\Delta; \overline{\mathcal{T}} \vdash_{\text{vctxt}} \Gamma \qquad x \notin dom(\Gamma) \qquad \Delta; \overline{\mathcal{T}} \vdash_{\text{type}} \tau}{\Delta; \overline{\mathcal{T}} \vdash_{\text{vctxt}} \Gamma, x{:}\tau}$$

$\boxed{\vdash_{\text{ctxt}} \Delta; \Gamma; \mathcal{T} : \overline{\mathcal{T}}}$

$$\frac{\vdash_{\text{ttype}} \mathcal{T} : \overline{\mathcal{T}} \qquad \Delta; \overline{\mathcal{T}} \vdash_{\text{vctxt}} \Gamma}{\vdash_{\text{ctxt}} \Delta; \Gamma; \mathcal{T} : \overline{\mathcal{T}}}$$

Figure B.13: Static semantics of $\mathsf{F}^{\mathsf{RGN}}$ (contexts)

ified to sub-judgments. Since the surface syntax does not admit explicitly named stacks and regions, we can type any closed, surface expression with the judgment $\cdot; \cdot; \cdot : \cdot \vdash_{\text{exp}} e : \tau$. Pushing these empty tower types and tower domains through the rules leads to the following simplifications:

$\Delta; \Gamma; \cdot : \cdot \vdash_{\text{exp}} e : \tau$	$\Longrightarrow$	$\Delta; \Gamma \vdash_{\text{exp}} e : \tau$
$\Delta; \cdot \vdash_{\text{type}} \tau$	$\Longrightarrow$	$\Delta \vdash_{\text{type}} \tau$
$\Delta; \cdot \vdash_{\text{index}} \theta$	$\Longrightarrow$	$\Delta \vdash_{\text{index}} \theta$
$\Delta; \cdot \vdash_{\text{vctxt}} \Gamma$	$\Longrightarrow$	$\Delta \vdash_{\text{vctxt}} \Gamma$
$\vdash_{\text{ctxt}} \Delta; \Gamma; \cdot : \cdot$	$\Longrightarrow$	$\vdash_{\text{ctxt}} \Delta; \Gamma$

Hence, we recover the type system presented in Section 3.2.3, which is sufficient for type-checking surface programs.

B.2 Type Soundness for $\mathsf{F}^{\mathsf{RGN}}$

In this section, we sketch a syntactic proof of type soundness [96]. We wish to prove that a well-typed, closed initial program either succeeds (returning a value

of the correct type) or diverges. A preservation theorem and a progress theorem make this theorem an easy corollary.

The Preservation Theorem states that the terminating computation of a well-typed expression yields a value of the same type. Because the dynamic semantics are defined by two mutually inductive judgments, the Preservation Theorem also states that the terminating computation of a well-typed command yields a well-typed extension of the top stack and a value of the same type. Various substitution lemmas for dead stacks and regions are required to prove the cases where stacks and regions are deallocated.

Theorem B.4 (Preservation)

(1) If

(a) $\vdash_{\text{tower}} T : \mathcal{T} : \overline{\mathcal{T}}$,

(b) $\cdot;\cdot;\mathcal{T} : \overline{\mathcal{T}} \vdash_{\text{exp}} e : \tau$, *and*

(c) $(T; e) \Downarrow v$,

then $\cdot;\cdot;\mathcal{T} : \overline{\mathcal{T}} \vdash_{\text{exp}} v : \tau$.

(2) If

(a) $\vdash_{\text{tower}} T, \mathfrak{s} \mapsto S : \mathcal{T}, \mathfrak{s} \mapsto \mathcal{S} : \overline{\mathcal{T}}, \mathfrak{s} \mapsto \overline{\mathcal{S}}$,

(b) $\cdot;\cdot;\mathcal{T}, \mathfrak{s} \mapsto \mathcal{S} : \overline{\mathcal{T}}, \mathfrak{s} \mapsto \overline{\mathcal{S}} \vdash_{\text{exp}} \kappa : \mathsf{RGN}\ \mathfrak{s}\sharp\mathfrak{r}\ \tau$, *and*

(c) $(T, \mathfrak{s} \mapsto S; \kappa) \Downarrow_{\kappa} S'; v$,

then there exists $\overline{\mathcal{S}}' \sqsupseteq \overline{\mathcal{S}}$ *and* $\mathcal{S}' \sqsupseteq \mathcal{S}$ *such that*

$\vdash_{\text{tower}} T, \mathfrak{s} \mapsto S' : \mathcal{T}, \mathfrak{s} \mapsto \mathcal{S}' : \overline{\mathcal{T}}, \mathfrak{s} \mapsto \overline{\mathcal{S}}'$ *and*

$\cdot;\cdot;\mathcal{T}, \mathfrak{s} \mapsto \mathcal{S}' : \overline{\mathcal{T}}, \mathfrak{s} \mapsto \overline{\mathcal{S}}' \vdash_{\text{exp}} v : \tau$.

Proof

By mutual induction on the derivations (1c) $(T; e) \Downarrow v$ and (2c) $(T, \mathfrak{s} \mapsto S; \kappa) \Downarrow_\kappa (S'; v)$.

The Progress Theorem states that a partially evaluated expression can always move forward towards complete evaluation. Progress Theorems are notoriously difficult in a large-step operational semantics. Hence, we make use of the *natural transition semantics* [84, 76] introduced in Appendix B.1.1. The Progress Theorem states that any well-typed partial derivation that contains a pending judgment can transition to another well-typed partial derivation. As usual, the proof of the Progress Theorem depends on a Canonical Forms Lemma, which describes the forms of values of particular types.

Definition B.1

(1) A pending judgment $(T; e) \Downarrow ?$ *is well typed iff there exists* $\overline{\mathcal{T}}$, $\mathcal{T}$, *and* τ *such that* $\vdash_{\text{tower}} T : \mathcal{T} : \overline{\mathcal{T}}$ *and* $\cdot; \cdot; \mathcal{T} : \overline{\mathcal{T}} \vdash_{\text{exp}} e : \tau$.

(2) A pending judgment $(T, \mathfrak{s} \mapsto S; \kappa) \Downarrow ?$ *is well typed iff there exists* $\overline{\mathcal{T}}$, $\mathcal{T}$, $\overline{\mathcal{S}}$, $\mathcal{S}$, $\mathfrak{r} \in dom(\overline{\mathcal{S}})$, *and* τ *such that* $\vdash_{\text{tower}} T, \mathfrak{s} \mapsto S : \mathcal{T}, \mathfrak{s} \mapsto \mathcal{S} : \overline{\mathcal{T}}, \mathfrak{s} \mapsto \overline{\mathcal{S}}$ *and* $\cdot; \cdot; \mathcal{T}, \mathfrak{s} \mapsto \mathcal{S} : \overline{\mathcal{T}}, \mathfrak{s} \mapsto \overline{\mathcal{S}} \vdash_{\text{exp}} \kappa : \mathsf{RGN}\ \mathfrak{s} \sharp \mathfrak{r}\ \tau$.

(3) A partial derivation $\mathfrak{D}$ *is well typed iff every pending judgment in it is well typed.*

Theorem B.5 (Progress)

If $\mathfrak{D}$ *is a well-typed partial derivation with pending judgments, then there exists* $\mathfrak{D}'$ *such that* $\mathfrak{D} \longrightarrow \mathfrak{D}'$ *and* $\mathfrak{D}'$ *is well typed.*

Proof

Let $\mathfrak{N}$ be the uppermost node of $\mathfrak{D}$ that is labeled with a pending judgment, either $(T; e) \Downarrow ?$ or $(T, \mathfrak{s} \mapsto S; \kappa) \Downarrow_\kappa ?$. Any transition on $\mathfrak{D}$ must occur at this node. Proceed by considering all possible forms of pending judgments.

Theorem B.6 (Soundness)

If $\cdot; \cdot; \cdot :: \cdot \vdash_{\text{exp}} e : \tau$, then any execution of e (in $\cdot$) either terminates with a value v (such that $\cdot; \cdot; \cdot :: \cdot \vdash_{\text{exp}} v : \tau$) or diverges.

Proof

Let $[(\cdot; e) \Downarrow ?]() \longrightarrow \mathfrak{D}_1 \longrightarrow \mathfrak{D}_2 \longrightarrow \cdots$ be an execution of e. Note that $[(\cdot; e) \Downarrow ?]()$ is well-typed by $\vdash_{\text{tower}} \cdot :: \cdot :: \cdot$ and $\cdot; \cdot; \cdot :: \cdot \vdash_{\text{exp}} e : \tau$. By Progress, every $\mathfrak{D}_i$ is well typed.

(1) Suppose that for all $\mathfrak{D}_n$ such that $[(\cdot; e) \Downarrow ?]() \longrightarrow^* \mathfrak{D}_n$, there exists $\mathfrak{D}_{n+1}$ such that $\mathfrak{D}_n \longrightarrow \mathfrak{D}_{n+1}$. Then, e diverges.

(2) Suppose that there exists $\mathfrak{D}_n$ such that $[(\cdot; e) \Downarrow ?]() \longrightarrow^* \mathfrak{D}_n$, such that there does not exist $\mathfrak{D}_{n+1}$ such that $\mathfrak{D}_n \longrightarrow \mathfrak{D}_{n+1}$.

   (a) Suppose $\mathfrak{D}_n$ contains no pending judgments. By Lemma B.2, $\mathfrak{D}_n \equiv [(\cdot; e) \Downarrow v]$. Then, e terminates with the value v. By Preservation, $\cdot; \cdot; \cdot :: \cdot \vdash_{\text{exp}} v : \tau$.

   (b) Suppose $\mathfrak{D}_n$ contains pending judgments. By Progress, there exists $\mathfrak{D}'$ such that $\mathfrak{D}_n \longrightarrow \mathfrak{D}'$, contradicting the assumption that there does not exist $\mathfrak{D}_{n+1}$ such that $\mathfrak{D}_n \longrightarrow \mathfrak{D}_{n+1}$. Thus, e cannot get stuck.

Remarks As stated previously, our main reason for adopting a large-step operational semantics is to simplify the theorems and proofs of Appendix B.3. However, we believe that the technique of proving type soundness for languages described by natural transition semantics shows great promise, particularly when combined with a monadic treatment of effects. In many ways, natural transition semantics attempts to bridge the gap between large-step operational semantics and small-step operational semantics. Natural transition semantics incorporates the advantages of large-step operational semantics (namely, a concise semantics) and ameliorates some of the disadvantages of small-step operational semantics. First, there is no need to introduce intermediate terms to "mark" points of interest in an evaluating program. For example, Semmelroth and Sabry's account of monadic state in ML [72] requires a term $\texttt{sto}\ \Delta\ e$, to distinguish nested `runST` evaluations.

Second, there is no need to introduce evaluation contexts. While this may appear to be a minor point (since we have effectively defined the evaluation context by the path through a partial derivation tree to a pending judgment), it has broader implications, particularly in the monadic setting. For example, Semmelroth and Sabry's evaluation contexts are quite complex, requiring four separate contexts. This complexity is required to express the relative sequencing of pure and monadic operations; essentially, the contexts must find the $\texttt{sto}\ \Delta\ []$ that corresponds to the "active" monadic evaluation, then follow commands down to either the "active" monadic command or "active" pure expression. In the natural transition semantics, this is accomplished "automatically" by jumping to the pending judgment of the partial derivation tree. In our case, the fact that this pending judgment can take one of two forms (either a pure call-by-value System F judgment or an imperative monadic judgment), effectively eliminates the need to interleave contexts.

We also believe that the complete soundness proof using natural transition semantics is easier than the corresponding proof using small-step operational semantics (for example, the soundness proof for Cyclone's region system [30]). Eliminating intermediate terms and evaluation contexts are obvious savings. The proof flavor is also slightly different: where one was doing case analysis on the form of the active position of an evaluation context, now one is doing case analysis on the pending judgment's children.

B.3 Translation from SEC to $\mathsf{F}^{\mathsf{RGN}}$

Section 3.3 gave a translation from the surface syntax of SEC to the surface syntax of $\mathsf{F}^{\mathsf{RGN}}$. However, the translation given in that section is insufficient for carrying the proof of translation correctness, since there are no translations for SEC stacks and `ref` terms, which arise during the evaluation of a program. This section extends the translation of Section 3.3 with cases for the additional semantic objects in the abstract machine configurations for SEC.

Recall that a SEC program requires exactly one region stack for evaluation; we assume that the corresponding stack in the translated $\mathsf{F}^{\mathsf{RGN}}$ program is labeled by the stack name $\mathfrak{s}$. To the translation functions given in Section 3.3, we add $\mathbb{X}[\![\cdot]\!]$, which translates into towers, tower types, and tower domains.

Stacks Figures B.14 and B.15 give the translation of values and storable values, which follow directly from the translations of expressions. Figures B.16 and B.17 give the translation of stacks, where each stored value is translated according to the $\vdash_{\text{sto}}$ derivation implied by the $\vdash_{\text{stack}}$ derivation. There is one minor complication due to the fact that a Single Effect Calculus program has an implicit region stack,

Translations yielding closed values

Closed values

$$\mathbb{E}\left[\!\!\left[\frac{\vdash_{\text{stype}} \mathcal{S} : \overline{\mathcal{S}}}{\mathcal{S} : \overline{\mathcal{S}} \vdash_{\text{val}} \mathfrak{b} : \mathsf{Bool}}\right]\!\!\right] \;=\; \mathfrak{b}$$

$$\mathbb{E}\left[\!\!\left[\frac{\vdash_{\text{stype}} \mathcal{S} : \overline{\mathcal{S}} \qquad \mathfrak{r} \in \overline{\mathcal{S}} \qquad \mathfrak{p} \in \overline{\mathcal{S}}(r) \qquad \mathcal{S}(\mathfrak{r}, \mathfrak{p}) = \omega}{\mathcal{S} : \overline{\mathcal{S}} \vdash_{\text{val}} \mathtt{ref}\ \mathfrak{r}\ \mathfrak{p} : (\omega, \mathfrak{r})}\right]\!\!\right] \;=\; \mathtt{ref}\ \mathfrak{s}\sharp\mathfrak{r}\ \mathfrak{p}$$

$$\mathbb{E}\left[\!\!\left[\frac{\vdash_{\text{stype}} \mathcal{S} : \overline{\mathcal{S}} \qquad \cdot; \overline{\mathcal{S}} \vdash_{\text{btype}} \omega}{\mathcal{S} : \overline{\mathcal{S}} \vdash_{\text{val}} \mathtt{ref}\ \bullet\ \mathfrak{p} : (\omega, \bullet)}\right]\!\!\right] \;=\; \begin{cases} \mathtt{ref}\ \circ\sharp\bullet\ \mathfrak{p} & \text{if } \overline{\mathcal{S}} = \cdot \\ \mathtt{ref}\ \mathfrak{s}\sharp\bullet\ \mathfrak{p} & \text{otherwise} \end{cases}$$

Figure B.14: Translation from SEC to $\mathsf{F}^{\mathsf{RGN}}$ (closed values)

Translations yielding closed values

Storable values

$$\mathbb{E}\left[\!\!\left[\frac{\vdash_{\text{stype}} \mathcal{S} : \overline{\mathcal{S}}}{\mathcal{S} : \overline{\mathcal{S}} \vdash_{\text{sto}} \mathfrak{i} : \mathsf{Int}}\right]\!\!\right] \;=\; \mathfrak{i}$$

$$\mathbb{E}\left[\!\!\left[\frac{\cdot; \cdot, x{:}\tau_x; \mathcal{S} : \overline{\mathcal{S}} \vdash_{\text{exp}} e : \tau, \pi'}{\mathcal{S} : \overline{\mathcal{S}} \vdash_{\text{sto}} \lambda x{:}\tau_x.^{\pi'}\, e : \tau_x \xrightarrow{\pi'} \tau}\right]\!\!\right] \;=\; \lambda x{:}\mathbb{T}[\![\tau_1]\!]\,.\,\mathbb{E}[\![e]\!]$$

$$\mathbb{E}\left[\!\!\left[\frac{\mathcal{S} : \overline{\mathcal{S}} \vdash_{\text{val}} v_1 : \tau_1 \qquad \cdots \qquad \mathcal{S} : \overline{\mathcal{S}} \vdash_{\text{val}} v_n : \tau_n}{\mathcal{S} : \overline{\mathcal{S}} \vdash_{\text{sto}} \langle v_1, \ldots, v_n \rangle : \tau_1 \times \cdots \times \tau_n}\right]\!\!\right] \;=\; \langle \mathbb{E}[\![v_1]\!], \ldots, \mathbb{E}[\![v_n]\!] \rangle$$

$$\mathbb{E}\left[\!\!\left[\frac{\cdot, \varrho \succeq \phi; \cdot; \mathcal{S} : \overline{\mathcal{S}} \vdash_{\text{exp}} u : \tau, \pi'}{\mathcal{S} : \overline{\mathcal{S}} \vdash_{\text{sto}} \Lambda\varrho \succeq \phi.^{\pi'}\, u : \forall \varrho \succeq \phi.^{\pi'}\, \tau}\right]\!\!\right] \;=$$

$$\Lambda\vartheta_\varrho.\, \lambda w_\varrho{:}\mathbb{T}[\![\varrho \succeq \phi]\!]\,.\lambda h_\varrho{:}\mathsf{RGNHnd}\ \vartheta_\varrho.\, \mathbb{E}[\![u]\!]$$

Figure B.15: Translation from SEC to $\mathsf{F}^{\mathsf{RGN}}$ (storable values)

Translations yielding tower domains

$$\begin{array}{l}\text{Stack domains}\\ \mathbb{X}[\![\overline{\mathcal{S}}]\!] \;=\; \begin{cases} \cdot & \text{if } \overline{\mathcal{S}} = \cdot \\ \cdot, \mathfrak{s} \mapsto \overline{\mathcal{S}} & \text{otherwise} \end{cases}\end{array}$$

Translations yielding stack types

$$\begin{array}{l}\text{Stacks types}\\ \mathbb{X}\left[\!\!\left[\begin{array}{c} \overline{\mathcal{S}} = dom(\mathcal{S}) \\ \forall \mathfrak{r} \in \overline{\mathcal{S}}.\ \overline{\mathcal{S}}(\mathfrak{r}) = dom(\mathcal{S}(\mathfrak{r})) \\ \forall \mathfrak{r} \in \overline{\mathcal{S}}.\ \forall \mathfrak{p} \in \overline{\mathcal{S}}(\mathfrak{r}).\ \cdot; \overline{\mathcal{S}} \vdash_{\text{btype}} \mathcal{S}(\mathfrak{r}, \mathfrak{p}) \\ \hline \vdash_{\text{stype}} \mathcal{S} : \overline{\mathcal{S}} \end{array}\right]\!\!\right] \;=\; \mathcal{S}^*\end{array}$$

where $dom(S) = dom(S^*)$

$$\forall \mathfrak{r} \in dom(S).\ dom(S(\mathfrak{r})) = dom(S^*(\mathfrak{r}))$$

$$\forall \mathfrak{r} \in dom(S).\ \forall \mathfrak{p} \in dom(\overline{\mathcal{S}}(\mathfrak{r})).\ S^*(\mathfrak{r}, \mathfrak{p}) = \mathbb{T}[\![\mathcal{S}(\mathfrak{r}, \mathfrak{p})]\!]$$

Translations yielding tower types

$$\begin{array}{l}\text{Stack types}\\ \mathbb{X}[\![\vdash_{\text{stype}} \mathcal{S} : \overline{\mathcal{S}}]\!] \;=\; \begin{cases} \cdot & \text{if } \mathcal{S} = \cdot \\ \cdot, \mathfrak{s} \mapsto \mathbb{X}[\![\vdash_{\text{stype}} \mathcal{S} : \overline{\mathcal{S}}]\!] & \text{otherwise} \end{cases}\end{array}$$

Figure B.16: Translation from SEC to $\mathsf{F}^{\mathsf{RGN}}$ (stack domains and stack types)

Translations yielding stacks

Stacks

$$\mathbb{X}[\![\vdash_{\text{stack}} S : \mathfrak{S} : \overline{\mathfrak{S}}]\!] \;=\; S^*$$

where $dom(S) = dom(S^*)$

$$\forall \mathfrak{r} \in dom(S).\ dom(S(\mathfrak{r})) = dom(S^*(\mathfrak{r}))$$

$$\forall \mathfrak{r} \in dom(S).\ \forall \mathfrak{p} \in dom(\overline{\mathfrak{S}}(\mathfrak{r})).\ S^*(\mathfrak{r}, \mathfrak{p}) = \mathbb{E}[\![S(\mathfrak{r}, \mathfrak{p})]\!]$$

Translations yielding towers

Stacks

$$\mathbb{X}[\![\vdash_{\text{stack}} S : \mathfrak{S} : \overline{\mathfrak{S}}]\!] \;=\; \begin{cases} \cdot & \text{if } S = \cdot \\ \cdot, \mathfrak{s} \mapsto \mathbb{X}[\![\vdash_{\text{stack}} S : \mathfrak{S} : \overline{\mathfrak{S}}]\!] & \text{otherwise} \end{cases}$$

Figure B.17: Translation from SEC to $\mathsf{F}^{\mathsf{RGN}}$ (stacks)

while $\mathsf{F}^{\mathsf{RGN}}$ explicitly introduces (and eliminates) a region stack with the `runRGN` command. Hence, a stack domain, stack type, or stack may be translated to either an empty tower or a tower with a single stack. We make this choice based on whether or not *any* region is allocated in the stack.

Terms Figure B.18 gives the translation of `ref` terms. As with the translation of stacks, an issue arises with occurrences of $\bullet$ in the source program, which may be translated either to $\mathfrak{s}\sharp\bullet$, within the scope of the `runRGN`, where $\mathfrak{s}$ is the name of the stack introduced by the `runRGN`, or to $\circ\sharp\bullet$, outside the scope of the `runRGN`. Again, we make the choice of translation based on whether or not *any* region is in the $\overline{\mathfrak{S}}$ stack domain.

Translations yielding expressions

Expressions

$$\mathbb{E}\left[\!\!\left[\frac{\vdash_{\text{ctxt}} \Delta;\Gamma;\mathcal{S}:\overline{\mathcal{S}};\pi \qquad \mathfrak{r}\in\overline{\mathcal{S}} \qquad \mathfrak{p}\in\overline{\mathcal{S}}(\mathfrak{r}) \qquad \mathcal{S}(\mathfrak{r},\mathfrak{p})=\omega}{\Delta;\Gamma;\mathcal{S}:\overline{\mathcal{S}}\vdash_{\text{exp}} \texttt{ref}\ \mathfrak{r}\ \mathfrak{p}:(\omega,\mathfrak{r}),\pi}\right]\!\!\right] =$$

$$\texttt{returnRGN}\ [\![\mathbb{I}[\![\pi]\!]]\!]\ [\![\mathbb{T}[\![(\omega,\mathfrak{r})]\!]]\!]\ \texttt{ref}\ \mathfrak{s}\sharp\mathfrak{r}\ \mathfrak{p}$$

$$\mathbb{E}\left[\!\!\left[\frac{\vdash_{\text{ctxt}} \Delta;\Gamma;\mathcal{S}:\overline{\mathcal{S}};\pi \qquad \Delta;\overline{\mathcal{S}}\vdash_{\text{btype}}\omega}{\Delta;\Gamma;\mathcal{S}:\overline{\mathcal{S}}\vdash_{\text{exp}} \texttt{ref}\ \bullet\ \mathfrak{p}:(\omega,\bullet),\pi}\right]\!\!\right] =$$

$$\begin{cases} \texttt{returnRGN}\ [\![\mathbb{I}[\![\pi]\!]]\!]\ [\![\mathbb{T}[\![(\omega,\bullet)]\!]]\!]\ \texttt{ref}\ \circ\sharp\bullet\ \mathfrak{p} & \text{if } \overline{\mathcal{S}}=\cdot \\ \texttt{returnRGN}\ [\![\mathbb{I}[\![\pi]\!]]\!]\ [\![\mathbb{T}[\![(\omega,\bullet)]\!]]\!]\ \texttt{ref}\ \mathfrak{s}\sharp\bullet\ \mathfrak{p} & \text{otherwise} \end{cases}$$

Figure B.18: Translation from SEC to $\mathsf{F}^{\mathsf{RGN}}$ (references)

Translations yielding indices

Regions

$$\mathbb{I}\left[\!\!\left[\frac{\overline{\mathcal{S}}\vdash_{\text{rctxt}}\Delta \qquad \varrho\in dom(\Delta)}{\Delta;\overline{\mathcal{S}}\vdash_{\text{region}}\varrho}\right]\!\!\right] = \vartheta_{\varrho}$$

$$\mathbb{I}\left[\!\!\left[\frac{\overline{\mathcal{S}}\vdash_{\text{rctxt}}\Delta \qquad \mathfrak{r}\in dom(\overline{\mathcal{S}})}{\Delta;\overline{\mathcal{S}}\vdash_{\text{region}}\mathfrak{r}}\right]\!\!\right] = \mathfrak{s}\sharp\mathfrak{r}$$

$$\mathbb{I}\left[\!\!\left[\frac{\overline{\mathcal{S}}\vdash_{\text{rctxt}}\Delta}{\Delta;\overline{\mathcal{S}}\vdash_{\text{region}}\bullet}\right]\!\!\right] = \begin{cases} \circ\sharp\bullet & \text{if } \overline{\mathcal{S}}=\cdot \\ \mathfrak{s}\sharp\bullet & \text{otherwise} \end{cases}$$

Figure B.19: Translation from SEC to $\mathsf{F}^{\mathsf{RGN}}$ (regions (I))

Translations yielding expressions

Regions

$$\mathbb{E}\left[\!\!\left[\frac{\overline{\mathbb{S}} \vdash_{\text{rctxt}} \Delta \qquad \varrho \in dom(\Delta)}{\Delta; \overline{\mathbb{S}} \vdash_{\text{region}} \varrho}\right]\!\!\right] \;=\; h_{\varrho}$$

$$\mathbb{E}\left[\!\!\left[\frac{\overline{\mathbb{S}} \vdash_{\text{rctxt}} \Delta \qquad \mathfrak{r} \in dom(\overline{\mathbb{S}})}{\Delta; \overline{\mathbb{S}} \vdash_{\text{region}} \mathfrak{r}}\right]\!\!\right] \;=\; \texttt{hnd}\ \mathfrak{s}\sharp\mathfrak{r}$$

$$\mathbb{E}\left[\!\!\left[\frac{\overline{\mathbb{S}} \vdash_{\text{rctxt}} \Delta}{\Delta; \overline{\mathbb{S}} \vdash_{\text{region}} \bullet}\right]\!\!\right] \;=\; \begin{cases} \texttt{hnd}\ \circ\sharp\bullet & \text{if } \overline{\mathbb{S}} = \cdot \\ \texttt{hnd}\ \mathfrak{s}\sharp\bullet & \text{otherwise} \end{cases}$$

Figure B.20: Translation from SEC to $\mathsf{F}^{\mathsf{RGN}}$ (regions (II))

Regions Constant regions may also appear in the translation of SEC regions to $\mathsf{F}^{\mathsf{RGN}}$ indices (Figure B.19) and in the translation of SEC regions to $\mathsf{F}^{\mathsf{RGN}}$ handles (Figure B.20).

Outlives relations Figure B.21 extends the translation of the SEC outlives relation given in Figure 3.21. During the evaluation of a SEC program, the outlives relation is instantiated with constant regions (either region names or a dead region); under the translation, these outlives relations correspond to $\mathsf{F}^{\mathsf{RGN}}$ `witnessRGN` terms. Note that the translation ensures that an outlives relation is translated to a `witnessRGN` with a valid cast (see Figure B.9).

B.3.1 Translation Properties

As stated in Section 3.3.1, the translation is type preserving. To handle the extended translation, we refine the lemma as follows:

Translations yielding expressions

Witnesses

$$\mathbb{E}\left[\!\!\left[\frac{\overline{\mathcal{S}} \vdash_{\text{rctxt}} \Delta \qquad (\varrho \succeq \{\rho_1, \ldots, \rho_i, \ldots, \rho_n\}) \in \Delta}{\Delta; \overline{\mathcal{S}} \vdash_{\text{rr}} \varrho \succeq \rho_i}\right]\!\!\right] =$$

$$\Lambda\beta.\, \lambda k{:}\mathsf{RGN}\ \mathbb{I}[\![\rho_i]\!]\ \beta.\, \texttt{let}\ w = \texttt{sel}_i\ w_\varrho\ \texttt{in}\ w\ [\beta]\ k$$

$$\mathbb{E}\left[\!\!\left[\frac{\Delta; \overline{\mathcal{S}} \vdash_{\text{region}} \rho}{\Delta; \overline{\mathcal{S}} \vdash_{\text{rr}} \rho \succeq \rho}\right]\!\!\right] = \Lambda\beta.\, \lambda k{:}\mathsf{RGN}\ \mathbb{I}[\![\rho]\!]\ \beta.\, k$$

$$\mathbb{E}\left[\!\!\left[\frac{\Delta; \overline{\mathcal{S}} \vdash_{\text{rr}} \rho_2 \succeq \rho' \qquad \Delta; \overline{\mathcal{S}} \vdash_{\text{rr}} \rho' \succeq \rho_1}{\Delta; \overline{\mathcal{S}} \vdash_{\text{rr}} \rho_2 \succeq \rho_1}\right]\!\!\right] =$$

$$\Lambda\beta.\, \lambda k{:}\mathsf{RGN}\ \mathbb{I}[\![\rho_1]\!]\ \beta.\, \texttt{let}\ k' = \mathbb{E}[\![\rho' \succeq \rho_1]\!]\ [\beta]\ k\ \texttt{in}\ \mathbb{E}[\![\rho_2 \succeq \rho']\!]\ [\beta]\ k'$$

$$\mathbb{E}\left[\!\!\left[\frac{\overline{\mathcal{S}} \vdash_{\text{rctxt}} \Delta \qquad \Delta; \overline{\mathcal{S}} \vdash_{\text{rr}} \rho \succeq \rho_i \quad {}^{i \in 1 \ldots n}}{\Delta; \overline{\mathcal{S}} \vdash_{\text{re}} \rho \succeq \{\rho_1, \ldots, \rho_n\}}\right]\!\!\right] = (\mathbb{E}[\![\rho \succeq \rho_1]\!], \ldots, \mathbb{E}[\![\rho \succeq \rho_n]\!])$$

$$\mathbb{E}\left[\!\!\left[\frac{\overline{\mathcal{S}} \vdash_{\text{rctxt}} \Delta \qquad \overline{\mathcal{S}} = \overline{\mathcal{S}}_1, \mathfrak{r}_1 \mapsto \overline{\mathcal{R}}_1, \overline{\mathcal{S}}_2, \mathfrak{r}_2 \mapsto \overline{\mathcal{R}}_2, \overline{\mathcal{S}}_3}{\Delta; \overline{\mathcal{S}} \vdash_{\text{rr}} \mathfrak{r}_2 \succeq \mathfrak{r}_1}\right]\!\!\right] =$$

$$\Lambda\beta.\, \lambda k{:}\mathsf{RGN}\ \mathfrak{s}\sharp\mathfrak{r}_1\ \beta.\, \texttt{witnessRGN}\ \mathfrak{s}\sharp\mathfrak{r}_1\ \mathfrak{s}\sharp\mathfrak{r}_2\ [\beta]\ k$$

$$\mathbb{E}\left[\!\!\left[\frac{\Delta; \overline{\mathcal{S}} \vdash_{\text{re}} \mathfrak{r} \succeq \{\mathfrak{r}_1, \ldots, \mathfrak{r}_n\}}{\Delta; \overline{\mathcal{S}} \vdash_{\text{rr}} \mathfrak{r} \succeq \mathfrak{r}_i}\right]\!\!\right] =$$

$$\Lambda\beta.\, \lambda k{:}\mathsf{RGN}\ \mathbb{I}[\![\mathfrak{r}_i]\!]\ \beta.\, \texttt{let}\ w = \texttt{sel}_i\ \mathbb{E}[\![\mathfrak{r} \succeq \{\mathfrak{r}_1, \ldots, \mathfrak{r}_n\}]\!]\ \texttt{in}\ w\ [\beta]\ k$$

$$\mathbb{E}\left[\!\!\left[\frac{\overline{\mathcal{S}} \vdash_{\text{rctxt}} \Delta \qquad \overline{\mathcal{S}} = \overline{\mathcal{S}}_1, \mathfrak{r}_1 \mapsto R_1, \overline{\mathcal{S}}_2}{\Delta; \overline{\mathcal{S}} \vdash_{\text{rr}} \bullet \succeq \mathfrak{r}_1}\right]\!\!\right] =$$

$$\Lambda\beta.\, \lambda k{:}\mathsf{RGN}\ \mathfrak{s}\sharp\mathfrak{r}_1\ \beta.\, \texttt{witnessRGN}\ \mathfrak{s}\sharp\mathfrak{r}_1\ \mathfrak{s}\sharp\bullet\ [\beta]\ k$$

Figure B.21: Translation from SEC to $\mathsf{F}^{\mathsf{RGN}}$ (outlives relations (II))

Lemma B.7 (Translation Preserves Types)

(1) If $\overline{\mathcal{S}} \vdash^{\mathsf{SEC}}_{\text{rctxt}} \Delta$, *then* $\mathbb{D}\llbracket \overline{\mathcal{S}} \vdash^{\mathsf{SEC}}_{\text{rctxt}} \Delta \rrbracket$ *is well-formed.*

(2) If $\Delta; \overline{\mathcal{S}}^{\mathsf{SEC}} \vdash_{\text{region}} \rho$, *then* $\mathbb{D}\llbracket \overline{\mathcal{S}} \vdash^{\mathsf{SEC}}_{\text{rctxt}} \Delta \rrbracket ; \mathbb{X}\llbracket \overline{\mathcal{S}} \rrbracket \vdash^{\mathsf{F}^{\mathsf{RGN}}}_{\text{index}} \mathbb{I}\llbracket \Delta; \overline{\mathcal{S}} \vdash^{\mathsf{SEC}}_{\text{region}} \rho \rrbracket$.

(3) If $\overline{\mathcal{S}} \vdash^{\mathsf{SEC}}_{\text{rctxt}} \Delta$, *then* $\mathbb{D}\llbracket \overline{\mathcal{S}} \vdash^{\mathsf{SEC}}_{\text{rctxt}} \Delta \rrbracket ; \mathbb{X}\llbracket \overline{\mathcal{S}} \rrbracket \vdash^{\mathsf{F}^{\mathsf{RGN}}}_{\text{vctxt}} \mathbb{G}\llbracket \overline{\mathcal{S}} \vdash^{\mathsf{SEC}}_{\text{rctxt}} \Delta \rrbracket$.

(4) If $\Delta; \overline{\mathcal{S}} \vdash^{\mathsf{SEC}}_{\text{btype}} \omega$, *then* $\mathbb{D}\llbracket \overline{\mathcal{S}} \vdash^{\mathsf{SEC}}_{\text{rctxt}} \Delta \rrbracket ; \mathbb{X}\llbracket \overline{\mathcal{S}} \rrbracket \vdash^{\mathsf{F}^{\mathsf{RGN}}}_{\text{type}} \mathbb{T}\llbracket \Delta; \overline{\mathcal{S}} \vdash^{\mathsf{SEC}}_{\text{btype}} \omega \rrbracket$.

(5) If $\Delta; \overline{\mathcal{S}} \vdash^{\mathsf{SEC}}_{\text{type}} \tau$, *then* $\mathbb{T}\llbracket \overline{\mathcal{S}} \vdash^{\mathsf{SEC}}_{\text{rctxt}} \Delta \rrbracket ; \mathbb{T}\llbracket \vdash_{\text{sdom}} \overline{\mathcal{S}} \rrbracket \vdash^{\mathsf{F}^{\mathsf{RGN}}}_{\text{type}} \mathbb{T}\llbracket \Delta; \overline{\mathcal{S}} \vdash^{\mathsf{SEC}}_{\text{type}} \tau \rrbracket$.

(6) If $\vdash^{\mathsf{SEC}}_{\text{stype}} \mathcal{S} : \overline{\mathcal{S}}$, *then* $\vdash^{\mathsf{F}^{\mathsf{RGN}}}_{\text{ttype}} \mathbb{X}\llbracket \vdash^{\mathsf{SEC}}_{\text{stype}} \mathcal{S} : \overline{\mathcal{S}} \rrbracket : \mathbb{X}\llbracket \overline{\mathcal{S}} \rrbracket$.

(7) If $\Delta; \overline{\mathcal{S}} \vdash^{\mathsf{SEC}}_{\text{vctxt}} \Gamma$, *then* $\mathbb{D}\llbracket \overline{\mathcal{S}} \vdash^{\mathsf{SEC}}_{\text{rctxt}} \Delta \rrbracket ; \mathbb{X}\llbracket \overline{\mathcal{S}} \rrbracket \vdash^{\mathsf{F}^{\mathsf{RGN}}}_{\text{vctxt}} \mathbb{G}\llbracket \Delta; \overline{\mathcal{S}} \vdash^{\mathsf{SEC}}_{\text{vctxt}} \Gamma \rrbracket$.

(8) If $\Delta; \overline{\mathcal{S}} \vdash^{\mathsf{SEC}}_{\text{vctxt}} \Gamma$, *then*

$$\mathbb{D}\llbracket \overline{\mathcal{S}} \vdash^{\mathsf{SEC}}_{\text{rctxt}} \Delta \rrbracket ; \mathbb{X}\llbracket \overline{\mathcal{S}} \rrbracket \vdash^{\mathsf{F}^{\mathsf{RGN}}}_{\text{vctxt}} \mathbb{G}\llbracket \overline{\mathcal{S}} \vdash^{\mathsf{SEC}}_{\text{rctxt}} \Delta \rrbracket , \mathbb{G}\llbracket \Delta; \overline{\mathcal{S}} \vdash^{\mathsf{SEC}}_{\text{vctxt}} \Gamma \rrbracket .$$

(9) If $\Delta; \overline{\mathcal{S}} \vdash^{\mathsf{SEC}}_{\text{rr}} \rho_2 \succeq \rho_1$, *then*

$$\mathbb{D}\llbracket \overline{\mathcal{S}} \vdash^{\mathsf{SEC}}_{\text{rctxt}} \Delta \rrbracket ; \mathbb{X}\llbracket \overline{\mathcal{S}} \rrbracket \vdash^{\mathsf{F}^{\mathsf{RGN}}}_{\text{type}} \mathbb{T}\llbracket \Delta; \overline{\mathcal{S}} \vdash^{\mathsf{SEC}}_{\text{rr}} \rho_2 \succeq \rho_1 \rrbracket .$$

(10) If $\Delta; \overline{\mathcal{S}} \vdash^{\mathsf{SEC}}_{\text{re}} \rho \succeq \phi$, *then*

$$\mathbb{D}\llbracket \overline{\mathcal{S}} \vdash^{\mathsf{SEC}}_{\text{rctxt}} \Delta \rrbracket ; \mathbb{X}\llbracket \overline{\mathcal{S}} \rrbracket \vdash^{\mathsf{F}^{\mathsf{RGN}}}_{\text{type}} \mathbb{T}\llbracket \Delta; \overline{\mathcal{S}} \vdash^{\mathsf{SEC}}_{\text{re}} \rho \succeq \phi \rrbracket .$$

(11) If $\vdash^{\mathsf{SEC}}_{\text{stype}} \mathcal{S} : \overline{\mathcal{S}}$ *and* $\Delta; \overline{\mathcal{S}} \vdash^{\mathsf{SEC}}_{\text{rr}} \rho_2 \succeq \rho_1$, *then*

$$\mathbb{D}\llbracket \overline{\mathcal{S}} \vdash^{\mathsf{SEC}}_{\text{rctxt}} \Delta \rrbracket ; \mathbb{G}\llbracket \overline{\mathcal{S}} \vdash^{\mathsf{SEC}}_{\text{rctxt}} \Delta \rrbracket ; \mathbb{X}\llbracket \vdash^{\mathsf{SEC}}_{\text{stype}} \mathcal{S} : \overline{\mathcal{S}} \rrbracket : \mathbb{X}\llbracket \overline{\mathcal{S}} \rrbracket$$
$$\vdash^{\mathsf{F}^{\mathsf{RGN}}}_{\text{exp}} \mathbb{E}\llbracket \Delta; \overline{\mathcal{S}} \vdash^{\mathsf{SEC}}_{\text{rr}} \rho_2 \succeq \rho_1 \rrbracket : \mathbb{T}\llbracket \Delta; \overline{\mathcal{S}} \vdash^{\mathsf{SEC}}_{\text{rr}} \rho_2 \succeq \rho_1 \rrbracket .$$

(12) If $\vdash^{\mathsf{SEC}}_{\text{stype}} \mathcal{S} : \overline{\mathcal{S}}$ *and* $\Delta; \overline{\mathcal{S}} \vdash^{\mathsf{SEC}}_{\text{re}} \rho \succeq \phi$, *then*

$$\mathbb{D}\llbracket \overline{\mathcal{S}} \vdash^{\mathsf{SEC}}_{\text{rctxt}} \Delta \rrbracket ; \mathbb{G}\llbracket \overline{\mathcal{S}} \vdash^{\mathsf{SEC}}_{\text{rctxt}} \Delta \rrbracket ; \mathbb{X}\llbracket \vdash^{\mathsf{SEC}}_{\text{stype}} \mathcal{S} : \overline{\mathcal{S}} \rrbracket : \mathbb{X}\llbracket \overline{\mathcal{S}} \rrbracket$$
$$\vdash^{\mathsf{F}^{\mathsf{RGN}}}_{\text{type}} \mathbb{E}\llbracket \Delta; \overline{\mathcal{S}} \vdash^{\mathsf{SEC}}_{\text{re}} \rho \succeq \phi \rrbracket : \mathbb{T}\llbracket \Delta; \overline{\mathcal{S}} \vdash^{\mathsf{SEC}}_{\text{re}} \rho \succeq \phi \rrbracket .$$

(13) If $\vdash^{\mathsf{SEC}}_{\text{stype}} \mathcal{S} : \overline{\mathcal{S}}$ *and* $\Delta; \overline{\mathcal{S}} \vdash^{\mathsf{SEC}}_{\text{region}} \rho$, *then*

$$\mathbb{D}\llbracket \overline{\mathcal{S}} \vdash^{\mathsf{SEC}}_{\text{rctxt}} \Delta \rrbracket ; \mathbb{G}\llbracket \overline{\mathcal{S}} \vdash^{\mathsf{SEC}}_{\text{rctxt}} \Delta \rrbracket ; \mathbb{X}\llbracket \vdash^{\mathsf{SEC}}_{\text{stype}} \mathcal{S} : \overline{\mathcal{S}} \rrbracket : \mathbb{X}\llbracket \overline{\mathcal{S}} \rrbracket$$
$$\vdash^{\mathsf{F}^{\mathsf{RGN}}}_{\text{exp}} \mathbb{E}\llbracket \Delta; \overline{\mathcal{S}} \vdash^{\mathsf{SEC}}_{\text{region}} \rho \rrbracket : \mathsf{RGNHnd}\ \mathbb{I}\llbracket \Delta; \overline{\mathcal{S}} \vdash^{\mathsf{SEC}}_{\text{region}} \rho \rrbracket .$$

(14) If $\Delta; \Gamma; \mathcal{S} : \overline{\mathcal{S}} \vdash^{\mathsf{SEC}}_{\text{exp}} e : \tau, \pi$, *then*

$$\mathbb{D}\llbracket \overline{\mathcal{S}} \vdash^{\mathsf{SEC}}_{\text{rctxt}} \Delta \rrbracket \, ; \mathbb{G}\llbracket \overline{\mathcal{S}} \vdash^{\mathsf{SEC}}_{\text{rctxt}} \Delta \rrbracket \, , \mathbb{G}\llbracket \Delta; \overline{\mathcal{S}} \vdash^{\mathsf{SEC}}_{\text{vctxt}} \Gamma \rrbracket \, ; \mathbb{X}\llbracket \vdash^{\mathsf{SEC}}_{\text{stype}} \mathcal{S} : \overline{\mathcal{S}} \rrbracket : \mathbb{X}\llbracket \overline{\mathcal{S}} \rrbracket$$
$$\vdash^{\mathsf{F}^{\mathsf{RGN}}}_{\text{exp}} \mathbb{E}\llbracket \Delta; \Gamma; \mathcal{S} : \overline{\mathcal{S}} \vdash^{\mathsf{SEC}}_{\text{exp}} e : \tau, \pi \rrbracket$$
$$: \mathsf{RGN}\ \mathbb{I}\llbracket \Delta; \overline{\mathcal{S}} \vdash^{\mathsf{SEC}}_{\text{region}} \pi \rrbracket \ \ \mathbb{T}\llbracket \Delta; \overline{\mathcal{S}} \vdash^{\mathsf{SEC}}_{\text{type}} \tau \rrbracket \, .$$

(15) If $\mathcal{S} : \overline{\mathcal{S}} \vdash^{\mathsf{SEC}}_{\text{sto}} w : \omega$, *then*

$$\cdot; \cdot; \mathbb{X}\llbracket \vdash^{\mathsf{SEC}}_{\text{stype}} \mathcal{S} : \overline{\mathcal{S}} \rrbracket : \mathbb{X}\llbracket \overline{\mathcal{S}} \rrbracket$$
$$\vdash^{\mathsf{F}^{\mathsf{RGN}}}_{\text{exp}} \mathbb{E}\llbracket \mathcal{S} : \overline{\mathcal{S}} \vdash^{\mathsf{SEC}}_{\text{sto}} w : \omega \rrbracket : \mathbb{T}\llbracket \cdot; \overline{\mathcal{S}} \vdash^{\mathsf{SEC}}_{\text{btype}} \omega \rrbracket \, .$$

(16) If $\vdash^{\mathsf{SEC}}_{\text{stack}} S : \mathcal{S} : \overline{\mathcal{S}}$, *then*

$$\vdash^{\mathsf{F}^{\mathsf{RGN}}}_{\text{tower}} \mathbb{X}\llbracket \vdash^{\mathsf{SEC}}_{\text{stack}} S : \mathcal{S} : \overline{\mathcal{S}} \rrbracket : \mathbb{X}\llbracket \vdash^{\mathsf{SEC}}_{\text{stype}} \mathcal{S} : \overline{\mathcal{S}} \rrbracket : \mathbb{X}\llbracket \overline{\mathcal{S}} \rrbracket .$$

(17) If $\vdash^{\mathsf{SEC}}_{\text{prog}} p$, *then* $\cdot; \cdot; \cdot; \cdot \vdash^{\mathsf{F}^{\mathsf{RGN}}}_{\text{exp}} \mathbb{E}\llbracket \vdash^{\mathsf{SEC}}_{\text{prog}} p \rrbracket : \mathsf{Bool}$.

As before, the proof is by (mutual) induction on the structure of the typing judgments, making frequent appeals to various well-formedness lemmas.

As stated in Section 3.3.1, the translation is meaning preserving, with respect to the dynamic semantics of SEC and $\mathsf{F}^{\mathsf{RGN}}$:

Theorem B.8 (Translation Correctness (Programs))

If $\vdash^{\mathsf{SEC}}_{\text{prog}} e$ *and* $e \Downarrow^{\mathsf{SEC}}_{\text{prog}} \mathfrak{b}$ *and* $\mathbb{E}\llbracket \vdash^{\mathsf{SEC}}_{\text{prog}} e \rrbracket = e^{\dagger}$,
then $(\cdot; e^{\dagger}) \Downarrow^{\mathsf{F}^{\mathsf{RGN}}} \mathfrak{b}$.

We noted that the proof relies on a *coherence* lemma stating that the translation of SEC outlives relations to $\mathsf{F}^{\mathsf{RGN}}$ witness terms yields functions that are operationally equivalent to the identity function. The extended translations in this section make it possible to state the Coherence Lemma precisely:

Lemma B.9 (Coherence)

Suppose $\vdash^{\mathsf{SEC}}_{\text{stack}} S : \mathcal{S} : \overline{\mathcal{S}}$ *and* $\cdot; \overline{\mathcal{S}} \vdash^{\mathsf{SEC}}_{\text{rr}} \mathfrak{r} \succeq \mathfrak{r}_i$.

Let $\mathbb{X}[\![\overline{\mathcal{S}}]\!] = \cdot, s \mapsto \overline{\mathcal{S}}^\dagger$, $\mathbb{X}[\![\vdash^{\mathsf{SEC}}_{\text{stype}} \mathcal{S} : \overline{\mathcal{S}}]\!] = \cdot, \mathfrak{s} \mapsto \mathcal{S}^\dagger$,

$\mathbb{X}[\![\vdash^{\mathsf{SEC}}_{\text{stack}} S : \mathcal{S} : \overline{\mathcal{S}}]\!] = \cdot, \mathfrak{s} \mapsto S^\dagger$, *and* $\mathbb{E}[\![\cdot; \overline{\mathcal{S}} \vdash^{\mathsf{SEC}}_{\text{rr}} \mathfrak{r} \succeq \mathfrak{r}_i]\!] = v^\dagger_w$.

If $\cdot; \cdot; \cdot, \mathfrak{s} \mapsto \mathcal{S}^\dagger : \cdot, \mathfrak{s} \mapsto \overline{\mathcal{S}}^\dagger \vdash^{\mathsf{F}^{\mathsf{RGN}}}_{\text{exp}} \kappa : \mathsf{RGN}\ \mathfrak{s} \sharp \mathfrak{r}_i\ \tau$ *and*

$(\cdot, \mathfrak{s} \mapsto S^\dagger; \kappa) \Downarrow^{\mathsf{F}^{\mathsf{RGN}}}_{\kappa} (S'; v')$,

then $(\cdot, \mathfrak{s} \mapsto S^\dagger; v^\dagger_w\ [\tau]\ \kappa) \Downarrow^{\mathsf{F}^{\mathsf{RGN}}} \kappa'$ *and* $(\cdot, \mathfrak{s} \mapsto S^\dagger; \kappa') \Downarrow^{\mathsf{F}^{\mathsf{RGN}}}_{\kappa} (S'; v')$.

Coherence is used throughout the proof of correctness to show that every evaluation derivation for the source can be simulated by a derivation involving the translation of the source:

Theorem B.10 (Translation Preserves Semantics)

Suppose $\vdash^{\mathsf{SEC}}_{\text{stack}} S : \mathcal{S} : \overline{\mathcal{S}}$, $\cdot; \cdot; \mathcal{S} : \overline{\mathcal{S}} \vdash^{\mathsf{SEC}}_{\text{exp}} e : \tau, \mathfrak{r}'$, *and* $(S; e) \Downarrow^{\mathsf{SEC}} (S'; v')$.

Then there exists $\overline{\mathcal{S}}' \sqsupseteq^{\mathsf{SEC}} \overline{\mathcal{S}}$ *and* $\mathcal{S}' \sqsupseteq^{\mathsf{SEC}} \mathcal{S}$

such that $\vdash^{\mathsf{SEC}}_{\text{stack}} S' : \mathcal{S}' : \overline{\mathcal{S}}'$ *and* $\mathcal{S}' : \overline{\mathcal{S}}' \vdash^{\mathsf{SEC}}_{\text{cval}} v' : \tau$.

Let $\mathbb{X}[\![\overline{\mathcal{S}}]\!] = \cdot, \mathfrak{s} \mapsto \overline{\mathcal{S}}^\dagger$, $\mathbb{X}[\![\vdash^{\mathsf{SEC}}_{\text{stype}} \mathcal{S} : \overline{\mathcal{S}}]\!] = \cdot, \mathfrak{s} \mapsto \mathcal{S}^\dagger$,

$\mathbb{X}[\![\vdash^{\mathsf{SEC}}_{\text{stack}} S : \mathcal{S} : \overline{\mathcal{S}}]\!] = \cdot, \mathfrak{s} \mapsto S^\dagger$, *and* $\mathbb{E}[\![\cdot; \cdot; \mathcal{S} : \overline{\mathcal{S}} \vdash^{\mathsf{SEC}}_{\text{exp}} e : \tau, \mathfrak{r}']\!] = e^\dagger$.

Then $(\cdot, \mathfrak{s} \mapsto S^\dagger; e^\dagger) \Downarrow^{\mathsf{F}^{\mathsf{RGN}}} \kappa'$ *and* $(\cdot, \mathfrak{s} \mapsto S^\dagger; \kappa') \Downarrow^{\mathsf{F}^{\mathsf{RGN}}}_{\kappa} (S'^\dagger; v'^\dagger)$,

where $\mathbb{X}\left[\!\left[\overline{\mathcal{S}}'\right]\!\right] = \overline{\mathcal{S}}'^\dagger$, $\mathbb{X}\left[\!\left[\vdash^{\mathsf{SEC}}_{\text{stype}} \mathcal{S}' : \overline{\mathcal{S}}'\right]\!\right] = \mathcal{S}'^\dagger$, $\mathbb{X}\left[\!\left[\vdash^{\mathsf{SEC}}_{\text{stack}} S' : \mathcal{S}' : \overline{\mathcal{S}}'\right]\!\right] = S'^\dagger$, *and*

$\mathbb{E}\left[\!\left[\mathcal{S}' : \overline{\mathcal{S}}' \vdash^{\mathsf{SEC}}_{\text{cval}} v' : \tau\right]\!\right] = v'^\dagger$.

We note that the proof is greatly simplified by using large-step operational semantics for both the source and target languages, since for many expression forms, a single operational step in the source language is expanded to many operational steps in the target language.

Full details of this development are given in the technical report *Monadic Regions: Formal Type Soundness and Correctness* [21].

Appendix C

A Substructural Type System: Technical Details

This appendix supplements the material in Chapter 4 with a number of technical details that would otherwise detract from that chapter's focus on the translation from $\mathsf{F}^{\mathsf{RGN}}$ to rgnURAL. In the following section, we revisit the presentation of the rgnURAL language, extending the dynamics to an *allocation semantics*, which models the allocation (and deallocation) of every data structure in the program (not just regions and references), and revising the static semantics to include judgments for the additional semantic objects introduced by the abstract machine configurations.

In Appendix C.2, we discuss a syntactic proof of type soundness for rgnURAL. We have carried out and mechanically-verified the proof in the Twelf system [66] using its metatheorem checker [71, 36, 37]. We also consider the major differences between the static semantics of Appendix C.1.2 and the static semantics as encoded in the mechanized proof.

C.1 The rgnURAL Language

C.1.1 Allocation Semantics of rgnURAL

While the dynamic semantics presented in Section 4.2.2 accurately captures the allocation and deallocation of regions and the manipulation of references during the evaluation of the program, it does not capture the number of times that a (functional) data structure is used during the evaluation of the program. For

example, consider the following program:

$$
\begin{array}{l}
\texttt{let } i = {}^{\mathsf{L}}1 \texttt{ in} \\
\texttt{let } f = {}^{\mathsf{L}}(\lambda x{:}{}^{\mathsf{L}}\overline{\mathsf{Int}}.\, {}^{\mathsf{L}}(x \oplus i)) \texttt{ in} \\
\texttt{let } g = {}^{\mathsf{L}}(\lambda h{:}{}^{\mathsf{U}}({}^{\mathsf{L}}\overline{\mathsf{Int}} \multimap {}^{\mathsf{L}}\overline{\mathsf{Int}}).\, h\ (h\ {}^{\mathsf{L}}3)) \texttt{ in} \\
g\ f
\end{array}
$$

This program may not be assigned a typing derivation according to the static semantics presented in Section 4.2.3, but it may be evaluated (without error) by the dynamic semantics presented in Section 4.2.2, yielding the value ${}^{\mathsf{L}}5$. Although a static semantics is a conservative approximation of those programs that evaluate without error, it would be more satisfactory to have a dynamic semantics that accurately reflected that a linear qualified data structure must be used *exactly once*, an affine *at most once*, a relevant *at least once*, and an unrestricted an arbitrary number of times.

In order to capture these properties, we adopt an *allocation semantics*, where each value is allocated in a global store (distinct from the global heap), which records whether or not the value has been used.

Abstract Machine Configurations for rgnURAL

Figures C.1 and C.2 present abstract machines configurations for rgnURAL, which extend the syntax of Section 4.2.1 with semantic objects that appear in the allocation semantics.

Locations are used to represent indirections to store allocated values. The abstract machine syntax adds locations as a new expression form. Note that this formulation of the rgnURAL language does not include references, handles, or capabilities as expression forms.

Location names

$\mathfrak{l} \quad \in \quad \mathit{LNames}$

Abstract terms

$e \quad ::= \quad \ldots \mid \mathfrak{l}$

Pointer names

$\mathfrak{p} \quad \in \quad \mathit{PNames}$

Region names

$\mathfrak{r} \quad \in \quad \mathit{RNames}$

Abstract pre-values

$\overline{v} \quad ::= \quad \mathfrak{i} \mid \mathfrak{b} \mid \lambda x{:}\tau.\, e \mid \langle \mathfrak{l}_1, \ldots, \mathfrak{l}_n \rangle \mid$

$\Lambda\xi.\, e \mid \mathtt{pack}(q, \mathfrak{l}) \mid \Lambda\overline{\alpha}.\, e \mid \mathtt{pack}(\overline{\tau}, \mathfrak{l}) \mid \Lambda\alpha.\, e \mid \mathtt{pack}(\tau, \mathfrak{l}) \mid$

$\mathtt{cap} \mid \mathtt{hnd}\ \mathfrak{r} \mid \mathtt{ref}\ \mathfrak{r}\ \mathfrak{p} \mid \Lambda\varrho.\, e \mid \mathtt{pack}(\rho, \mathfrak{l})$

Abstract values

$v \quad ::= \quad {}^{\mathfrak{q}}\overline{v}$

Figure C.1: Abstract machine syntax of rgnURAL (I)

Flags

$$\mathfrak{f} \quad ::= \quad \mathsf{unused} \mid \mathsf{used}$$

Stores

$$\sigma \quad ::= \quad \{\mathfrak{l}_1 \mapsto (\mathfrak{f}_1, \upsilon_1), \ldots, \mathfrak{l}_n \mapsto (\mathfrak{f}_n, \upsilon_n)\}$$

Regions

$$R \quad ::= \quad \{\mathfrak{p}_1 \mapsto (\mathfrak{q}_1, \mathfrak{l}_1), \ldots, \mathfrak{p}_n \mapsto (\mathfrak{q}_n, \mathfrak{l}_n)\}$$

Region mark

$$\upsilon \quad ::= \quad \mathtt{^{q}live} \mid \mathtt{dead}$$

Heaps

$$H \quad ::= \quad \{\mathfrak{r}_1 \mapsto (\upsilon_1, R_1), \ldots, \mathfrak{r}_n \mapsto (\upsilon_n, R_n)\}$$

Abstract machine configurations

$$(H; \sigma; e)$$

Figure C.2: Abstract machine syntax of rgnURAL (II)

Value forms in rgnURAL are structured as a (constant) qualifier applied to a (closed) pre-value, which mirrors the structuring of types. Note that value forms are not a subset of expression forms; rather, they are a disjoint syntactic class representing store allocated data structures. Since we are assuming that *all* values are store allocated, pre-values include references, handles, and capabilities. Furthermore, the components of a tuple and of an existential package are required to be locations.

Figure C.2 gives the syntax of stores, regions, and heaps. Intuitively, values are associated with locations in stores σ; locations are associated with pointers

in regions R; regions are collected into heaps H. In order to support a syntactic proof of type soundness, the structure of stores, regions, and heaps includes some additional instrumentation.

A store σ maps locations $\mathfrak{l}$ to a pair of a flag $\mathfrak{f}$ and a value; the flag records whether or not the value has been used during the evaluation of the program.

A region R maps pointers $\mathfrak{p}$ to a pair of a qualifier $\mathfrak{q}$ and a location; the qualifier records the qualifier that annotated the `new` primitive that allocated the corresponding reference. A heap H maps region names $\mathfrak{r}$ to a pair of a region mark ν and a region; the region mark records whether the named region is allocated ($^{\mathfrak{q}_c}$`live`) or deallocated (`dead`). As in the operational semantics of Section 4.2.2, the allocation semantics will not allow the evaluation of a rgnURAL program to access a deallocated region. When a region is allocated, the region mark $^{\mathfrak{q}_c}$`live` records the qualifier of the capability associated with the region.

The notation $\sigma_1 \uplus \sigma_2$ (respectively, $H_1 \uplus H_2$ and $R_1 \uplus R_2$) denotes the disjoint union of the stores σ_1 and σ_2 (respectively, the heaps H_1 and H_2 and the regions R_1 and R_2); the operation is undefined if the domains of σ_1 and σ_2 (respectively, H_1 and H_2 and R_1 and R_2) are not disjoint.

Store and heap rules Before turning to the inductive judgment that defines the allocation semantics, we introduce a number of auxiliary judgments that factor out store and heap manipulations (see Figures C.3, C.4, and C.5).

The judgment $(\sigma; v) \xrightarrow{\mathsf{alloc}} (\sigma'; \mathfrak{l}')$ encapsulates the allocation of a new value in the store. The new value is assigned to a fresh location $\mathfrak{l}'$ and is flagged as unused.

Somewhat more interesting is the judgment $(\sigma; \mathfrak{l}) \xrightarrow{\mathsf{fetch}} (\sigma'; v)$, which fetches the contents of a location in the store. Since a value will only be fetched in order to be

$\boxed{(\sigma; v) \xrightarrow{\mathsf{alloc}} (\sigma'; \mathfrak{l}')}$

$$\frac{\mathfrak{l}' \notin dom(\sigma)}{(\sigma; v) \xrightarrow{\mathsf{alloc}} (\sigma, \mathfrak{l}' \mapsto (\mathsf{unused}, v); \mathfrak{l}')}$$

$\boxed{(\sigma; \mathfrak{l}) \xrightarrow{\mathsf{fetch}} (\sigma'; v)}$

$$\frac{\mathsf{q} \sqsubseteq \mathsf{R}}{(\sigma \uplus \{\mathfrak{l} \mapsto (\mathfrak{f}, {}^{\mathsf{q}}\overline{v})\}; \mathfrak{l}) \xrightarrow{\mathsf{fetch}} (\sigma \uplus \{\mathfrak{l} \mapsto (\mathsf{used}, {}^{\mathsf{q}}\overline{v})\}; {}^{\mathsf{q}}\overline{v})}$$

$$\frac{\mathsf{A} \sqsubseteq \mathsf{q}}{(\sigma \uplus \{\mathfrak{l} \mapsto (\mathsf{unused}, {}^{\mathsf{q}}\overline{v})\}; \mathfrak{l}) \xrightarrow{\mathsf{fetch}} (\sigma; {}^{\mathsf{q}}\overline{v})}$$

Figure C.3: Dynamic semantics of rgnURAL (store)

$\boxed{(H; \mathfrak{q}_c) \xrightarrow{\mathsf{newrgn}} (H'; \mathfrak{r}')}$

$$\frac{\mathfrak{r} \notin dom(H)}{(H; \mathfrak{q}_c) \xrightarrow{\mathsf{newrgn}} (H \uplus \{\mathfrak{r} \mapsto ({}^{\mathfrak{q}_c}\mathtt{live}, \{\})\}; \mathfrak{r})}$$

$\boxed{(H; \mathfrak{r}) \xrightarrow{\mathsf{freergn}} H'}$

$$\frac{}{(H \uplus \{\mathfrak{r} \mapsto ({}^{\mathfrak{q}_c}\mathtt{live}, R)\}; \mathfrak{r}) \xrightarrow{\mathsf{freergn}} H \uplus \{\mathfrak{r} \mapsto (\mathtt{dead}, R)\}}$$

Figure C.4: Dynamic semantics of rgnURAL (heap (I))

$$\boxed{(H; \mathfrak{r}; \mathfrak{q}_r; \mathfrak{l}_\star) \xrightarrow{\mathsf{new}} (H'; \mathfrak{p}')}$$

$$\frac{\mathfrak{p}' \notin dom(R)}{(H \uplus \{\mathfrak{r} \mapsto ({}^{\mathfrak{q}_c}\mathtt{live}, R)\}; \mathfrak{r}; \mathfrak{q}_r; \mathfrak{l}_\star) \xrightarrow{\mathsf{new}} (H \uplus \{\mathfrak{r} \mapsto ({}^{\mathfrak{q}_c}\mathtt{live}, R \uplus \{\mathfrak{p}' \mapsto (\mathfrak{q}_r, \mathfrak{l}_\star)\})\}; \mathfrak{p}')}$$

$$\boxed{(H; \mathfrak{r}; \mathfrak{p}) \xrightarrow{\mathsf{free}} (H'; \mathfrak{l})}$$

$$\frac{}{(H \uplus \{\mathfrak{r} \mapsto ({}^{\mathfrak{q}_c}\mathtt{live}, R \uplus \{\mathfrak{p} \mapsto (\mathfrak{q}_r, \mathfrak{l})\})\}; \mathfrak{r}; \mathfrak{p}) \xrightarrow{\mathsf{free}} (H \uplus \{\mathfrak{r} \mapsto ({}^{\mathfrak{q}_c}\mathtt{live}, R)\}; \mathfrak{l})}$$

$$\boxed{(H; \mathfrak{r}; \mathfrak{p}) \xrightarrow{\mathsf{read}} \mathfrak{l}}$$

$$\frac{}{(H \uplus \{\mathfrak{r} \mapsto ({}^{\mathfrak{q}_c}\mathtt{live}, R \uplus \{\mathfrak{p} \mapsto (\mathfrak{q}_r, \mathfrak{l})\})\}; \mathfrak{r}; \mathfrak{p}) \xrightarrow{\mathsf{read}} \mathfrak{l}}$$

$$\boxed{(H; \mathfrak{r}; \mathfrak{p}; \mathfrak{l}_\star) \xrightarrow{\mathsf{write}} H'}$$

$$\frac{}{(H \uplus \{\mathfrak{r} \mapsto ({}^{\mathfrak{q}_c}\mathtt{live}, R \uplus \{\mathfrak{p} \mapsto (\mathfrak{q}_r, \mathfrak{l})\})\}; \mathfrak{r}; \mathfrak{p}; \mathfrak{l}_\star) \xrightarrow{\mathsf{write}} H \uplus \{\mathfrak{r} \mapsto ({}^{\mathfrak{q}_c}\mathtt{live}, R \uplus \{\mathfrak{p} \mapsto (\mathfrak{q}_r, \mathfrak{l}_\star)\})\}}$$

$$\boxed{(H; \mathfrak{r}; \mathfrak{p}; \mathfrak{l}_\star) \xrightarrow{\mathsf{swap}} (H'; \mathfrak{l})}$$

$$\frac{}{(H \uplus \{\mathfrak{r} \mapsto ({}^{\mathfrak{q}_c}\mathtt{live}, R \uplus \{\mathfrak{p} \mapsto (\mathfrak{q}_r, \mathfrak{l})\})\}; \mathfrak{r}; \mathfrak{p}; \mathfrak{l}_\star) \xrightarrow{\mathsf{swap}} (H \uplus \{\mathfrak{r} \mapsto ({}^{\mathfrak{q}_c}\mathtt{live}, R \uplus \{\mathfrak{p} \mapsto (\mathfrak{q}_r, \mathfrak{l}_\star)\})\}; \mathfrak{l})}$$

Figure C.5: Dynamic semantics of rgnURAL (heap (II))

used, we reflect this fact in the updated store σ'. If the fetched value is unrestricted or relevant ($\mathsf{q} \sqsubseteq \mathsf{R}$), then (independent of the current flag) the location is flagged as used in the updated store. On the other hand, if the fetched value is affine or linear ($\mathsf{A} \sqsubseteq \mathsf{q}$), then the location is required to be currently unused and is removed from the updated store. This corresponds to the fact that an affine or linear value is required to be used *at most once* and *at least once*, respectively. By removing affine and linear values from the store, the semantics ensures that there would be an evaluation error if they were to be used more than once.

The judgments in Figure C.4 encapsulate the allocation and deallocation of regions, while the judgments in Figure C.4 encapsulate the allocation, deallocation, reading, writing, and swapping of references. They are straightforward manipulations of heaps and regions. Note that, like the semantics of Section 4.2.2, a region must be `live` in order to be accessed.

Evaluation rules An inductive judgment Figures C.6–C.12 defines the allocation semantics. We state without proof that the allocation semantics is deterministic, taking $(\sigma; H; e)$ configurations modulo α-conversion, including conversion of locations, region names, and pointers, which are uniquely bound in the store σ and heap H.

The judgment $(\sigma; H; e) \longmapsto (\sigma'; H'; e')$ asserts that one step of evaluation of the closed expression e in a store σ and heap H results in a new store σ' and heap H' and new expression e'.

The rules for $(\sigma; H; e) \longmapsto (\sigma'; H'; e')$ are very similar to the rules for $(H; e) \longmapsto (H'; e')$ given in Figures 4.9–4.13. The major difference is that each introduction form allocates a new value in the store, while each elimination form

$\boxed{(\sigma; H; e) \longmapsto (\sigma'; H'; e')}$

$$\frac{(\sigma; {}^{\mathfrak{q}}\mathfrak{i}) \xrightarrow{\mathsf{alloc}} (\sigma'; \mathfrak{l}')}{(\sigma; H; {}^{\mathfrak{q}}\mathfrak{i}) \longmapsto (\sigma'; H; \mathfrak{l}')}$$

$$\frac{(\sigma; \mathfrak{l}_1) \xrightarrow{\mathsf{fetch}} (\sigma_1; {}^{-}\mathfrak{i}_1) \qquad (\sigma_1; \mathfrak{l}_2) \xrightarrow{\mathsf{fetch}} (\sigma_2; {}^{-}\mathfrak{i}_2) \qquad \mathfrak{i}_1 \oplus \mathfrak{i}_2 = \mathfrak{i} \qquad (\sigma_2; {}^{\mathfrak{q}}\mathfrak{i}) \xrightarrow{\mathsf{alloc}} (\sigma'; \mathfrak{l}')}{(\sigma; H; {}^{\mathfrak{q}}(\mathfrak{l}_1 \oplus \mathfrak{l}_2)) \longmapsto (\sigma'; H; \mathfrak{l}')}$$

$$\frac{(\sigma; \mathfrak{l}_1) \xrightarrow{\mathsf{fetch}} (\sigma_1; {}^{-}\mathfrak{i}_1) \qquad (\sigma_1; \mathfrak{l}_2) \xrightarrow{\mathsf{fetch}} (\sigma_2; {}^{-}\mathfrak{i}_2) \qquad \mathfrak{i}_1 \oslash \mathfrak{i}_2 = \mathfrak{b} \qquad (\sigma_2; {}^{\mathfrak{q}}\mathfrak{b}) \xrightarrow{\mathsf{alloc}} (\sigma'; \mathfrak{l}')}{(\sigma; H; {}^{\mathfrak{q}}(\mathfrak{l}_1 \oslash \mathfrak{l}_2)) \longmapsto (\sigma'; H; \mathfrak{l}')}$$

$$\frac{(\sigma; {}^{\mathfrak{q}}\mathfrak{b}) \xrightarrow{\mathsf{alloc}} (\sigma'; \mathfrak{l}')}{(\sigma; H; {}^{\mathfrak{q}}\mathfrak{b}) \longmapsto (\sigma'; H; \mathfrak{l}')}$$

$$\frac{(\sigma; \mathfrak{l}) \xrightarrow{\mathsf{fetch}} (\sigma'; {}^{-}\texttt{true})}{(\sigma; H; \texttt{if } \mathfrak{l} \texttt{ then } e_t \texttt{ else } e_f) \longmapsto (\sigma'; H; e_t)}$$

$$\frac{(\sigma; \mathfrak{l}) \xrightarrow{\mathsf{fetch}} (\sigma'; {}^{-}\texttt{false})}{(\sigma; H; \texttt{if } \mathfrak{l} \texttt{ then } e_t \texttt{ else } e_f) \longmapsto (\sigma'; H; e_f)}$$

Figure C.6: Dynamic semantics of rgnURAL (expressions (I))

$\boxed{(\sigma; H; e) \longmapsto (\sigma'; H'; e')}$

$$\frac{(\sigma; {}^{\mathsf{q}}(\lambda x{:}\tau.\, e)) \xrightarrow{\mathsf{alloc}} (\sigma'; \mathfrak{l}')}{(\sigma; H; {}^{\mathsf{q}}(\lambda x{:}\tau.\, e)) \longmapsto (\sigma'; H; \mathfrak{l}')} \qquad \frac{(\sigma; \mathfrak{l}_f) \xrightarrow{\mathsf{fetch}} (\sigma'; {}^{-}(\lambda x{:}\tau_x.\, e_b)}{(\sigma; H; \mathfrak{l}_f\ \mathfrak{l}_a) \longmapsto (\sigma'; H; e_b[\mathfrak{l}_a/x])}$$

$$\frac{(\sigma; {}^{\mathsf{q}}\langle \mathfrak{l}_1, \ldots, \mathfrak{l}_n \rangle) \xrightarrow{\mathsf{alloc}} (\sigma'; \mathfrak{l}')}{(\sigma; H; {}^{\mathsf{q}}\langle \mathfrak{l}_1, \ldots, \mathfrak{l}_n \rangle) \longmapsto (\sigma'; H; \mathfrak{l}')}$$

$$\frac{(\sigma; \mathfrak{l}_a) \xrightarrow{\mathsf{fetch}} (\sigma'; {}^{-}\langle \mathfrak{l}_1, \ldots, \mathfrak{l}_n \rangle)}{(\sigma; H; \mathtt{let}\ \langle x_1, \ldots, x_n \rangle = \mathfrak{l}_a\ \mathtt{in}\ e_b) \longmapsto (\sigma'; H; e_b[\mathfrak{l}_1/x_1] \cdots [\mathfrak{l}_n/x_n])}$$

$$\frac{(\sigma; {}^{\mathsf{q}}(\Lambda\xi.\, e)) \xrightarrow{\mathsf{alloc}} (\sigma'; \mathfrak{l}')}{(\sigma; H; {}^{\mathsf{q}}(\Lambda\xi.\, e)) \longmapsto (\sigma'; H; \mathfrak{l}')} \qquad \frac{(\sigma; \mathfrak{l}_f) \xrightarrow{\mathsf{fetch}} (\sigma'; {}^{-}(\lambda\xi.\, e_b))}{(\sigma; H; \mathfrak{l}_f\ [q_a]) \longmapsto (\sigma'; H; e_b[q_a/\xi])}$$

$$\frac{(\sigma; {}^{\mathsf{q}}\mathtt{pack}(\mathfrak{q}, \mathfrak{l})) \xrightarrow{\mathsf{alloc}} (\sigma'; \mathfrak{l}')}{(\sigma; H; {}^{\mathsf{q}}\mathtt{pack}(\mathfrak{q}, \mathfrak{l})) \longmapsto (\sigma'; H; \mathfrak{l}')}$$

$$\frac{(\sigma; \mathfrak{l}_a) \xrightarrow{\mathsf{fetch}} (\sigma'; {}^{-}\mathtt{pack}(q, \mathfrak{l}_x))}{(\sigma; H; \mathtt{let}\ \mathtt{pack}(\xi, x) = \mathfrak{l}_a\ \mathtt{in}\ e_b) \longmapsto (\sigma'; H; e_b[\mathfrak{q}/\xi][\mathfrak{l}_x/x])}$$

Figure C.7: Dynamic semantics of rgnURAL (expressions (II))

$\boxed{(\sigma; H; e) \longmapsto (\sigma'; H'; e')}$

$$\frac{(\sigma; {}^{\mathsf{q}}(\Lambda\overline{\alpha}.\, e)) \xrightarrow{\mathsf{alloc}} (\sigma'; \mathfrak{l}')}{(\sigma; H; {}^{\mathsf{q}}(\Lambda\overline{\alpha}.\, e)) \longmapsto (\sigma'; H; \mathfrak{l}')} \qquad \frac{(\sigma; \mathfrak{l}_f) \xrightarrow{\mathsf{fetch}} (\sigma'; {}^{\mathsf{q}}(\lambda\overline{\alpha}.\, e_b))}{(\sigma; H; \mathfrak{l}_f\ [\overline{\tau}_a]) \longmapsto (\sigma'; H; e_b[\overline{\tau}_a/\overline{\alpha}])}$$

$$\frac{(\sigma; {}^{\mathsf{q}}\mathtt{pack}(\overline{\tau}, \mathfrak{l})) \xrightarrow{\mathsf{alloc}} (\sigma'; \mathfrak{l}')}{(\sigma; H; {}^{\mathsf{q}}\mathtt{pack}(\overline{\tau}, \mathfrak{l})) \longmapsto (\sigma'; H; \mathfrak{l}')}$$

$$\frac{(\sigma; \mathfrak{l}_a) \xrightarrow{\mathsf{fetch}} (\sigma'; {}^{-}\mathtt{pack}(\overline{\tau}, \mathfrak{l}_x))}{(\sigma; H; \mathtt{let}\ \mathtt{pack}(\overline{\alpha}, x) = \mathfrak{l}_a\ \mathtt{in}\ e_b) \longmapsto (\sigma'; H; e_b[\overline{\tau}/\overline{\alpha}][\mathfrak{l}_x/x])}$$

$$\frac{(\sigma; {}^{\mathsf{q}}(\Lambda\alpha.\, e)) \xrightarrow{\mathsf{alloc}} (\sigma'; \mathfrak{l}')}{(\sigma; H; {}^{\mathsf{q}}(\Lambda\alpha.\, e)) \longmapsto (\sigma'; H; \mathfrak{l}')} \qquad \frac{(\sigma; \mathfrak{l}_f) \xrightarrow{\mathsf{fetch}} (\sigma'; {}^{-}(\lambda\alpha.\, e_b))}{(\sigma; H; \mathfrak{l}_f\ [\tau_a]) \longmapsto (\sigma'; H; e_b[\tau_a/\alpha])}$$

$$\frac{(\sigma; {}^{\mathsf{q}}\mathtt{pack}(\tau, \mathfrak{l})) \xrightarrow{\mathsf{alloc}} (\sigma'; \mathfrak{l}')}{(\sigma; H; {}^{\mathsf{q}}\mathtt{pack}(\tau, \mathfrak{l})) \longmapsto (\sigma'; H; \mathfrak{l}')}$$

$$\frac{(\sigma; \mathfrak{l}_a) \xrightarrow{\mathsf{fetch}} (\sigma'; {}^{-}\mathtt{pack}(\tau, \mathfrak{l}_x))}{(\sigma; H; \mathtt{let}\ \mathtt{pack}(\alpha, x) = \mathfrak{l}_a\ \mathtt{in}\ e_b) \longmapsto (\sigma'; H; e_b[\tau/\alpha][\mathfrak{l}_x/x])}$$

$$\frac{}{(\sigma; H; \mathtt{let}\ x = \mathfrak{l}_a\ \mathtt{in}\ e_b) \longmapsto (\sigma'; H; e_b[\mathfrak{l}_a/x])}$$

Figure C.8: Dynamic semantics of rgnURAL (expressions (III))

$\boxed{(\sigma; H; e) \longmapsto (\sigma'; H'; e')}$

Evaluation contexts

$$
\begin{aligned}
E \quad ::= \quad & [\cdot] \mid {}^{q}(E_1 \oplus e_2) \mid {}^{q}(\mathfrak{l}_1 \oplus E_2) \mid {}^{q}(E_1 \otimes e_2) \mid {}^{q}(\mathfrak{l}_1 \otimes E_2) \mid \\
& \texttt{if } E_b \texttt{ then } e_t \texttt{ else } e_f \mid \\
& E_f\ e_a \mid \mathfrak{l}_f\ E_a \mid \\
& {}^{q}\langle \mathfrak{l}_1, \ldots, E_i, \ldots, e_n \rangle \mid \texttt{let } \langle x_1, \ldots, x_n \rangle = E_1 \texttt{ in } e_2 \mid \\
& E_f\ [q_a] \mid {}^{q}\texttt{pack}(q, E) \mid \texttt{let pack}(\xi, x) = E_a \texttt{ in } e_b \mid \\
& E_f\ [\overline{\tau}_a] \mid {}^{q}\texttt{pack}(\overline{\tau}, E) \mid \texttt{let pack}(\overline{\alpha}, x) = E_a \texttt{ in } e_b \mid \\
& E_f\ [\tau_a] \mid {}^{q}\texttt{pack}(\tau, E) \mid \texttt{let pack}(\alpha, x) = E_f \texttt{ in } e_a \mid \\
& \texttt{let } x = E_a \texttt{ in } e_b \mid \\
& \texttt{freergn } E_c\ e_h \mid \texttt{freergn } \mathfrak{l}_c\ E_h \mid \\
& {}^{q}\texttt{new } E_c\ e_h\ e_a \mid {}^{q}\texttt{new } \mathfrak{l}_c\ E_h\ e_a \mid {}^{q}\texttt{new } \mathfrak{l}_c\ \mathfrak{l}_h\ E_a \mid \\
& \texttt{free } E_c\ e_r \mid \texttt{free } \mathfrak{l}_c\ E_r \mid \\
& \texttt{read } E_c\ e_r \mid \texttt{read } \mathfrak{l}_c\ E_r \mid \\
& \texttt{write } E_c\ e_r\ e_a \mid \texttt{write } \mathfrak{l}_c\ E_r\ e_a \mid \texttt{write } \mathfrak{l}_c\ \mathfrak{l}_r\ E_a \mid \\
& \texttt{swap } E_c\ e_r\ e_a \mid \texttt{swap } \mathfrak{l}_c\ E_r\ e_a \mid \texttt{swap } \mathfrak{l}_c\ \mathfrak{l}_r\ E_a \mid \\
& E_f\ [\rho_a] \mid {}^{q}\texttt{pack}(\rho, E) \mid \texttt{let pack}(\varrho, x) = E_a \texttt{ in } e_b
\end{aligned}
$$

$$\frac{(\sigma; H; e) \longmapsto (\sigma'; H'; e')}{(\sigma; H; E[e]) \longmapsto (\sigma'; H'; E[e'])}$$

Figure C.9: Dynamic semantics of rgnURAL (contexts)

$\boxed{(\sigma; H; e) \longmapsto (\sigma'; H'; e')}$

$$\frac{(H; \mathfrak{q}_c) \xrightarrow{\mathsf{newrgn}} (H'; \mathfrak{r}') \quad (\sigma; {}^{\mathfrak{q}_c}(\mathtt{cap})) \xrightarrow{\mathsf{alloc}} (\sigma'_c; \mathfrak{l}'_c) \quad (\sigma'_c; {}^{\mathfrak{q}_h}(\mathtt{hnd}\ \mathfrak{r}')) \xrightarrow{\mathsf{alloc}} (\sigma'_h; \mathfrak{l}'_h) \quad (\sigma'_h; {}^{\mathsf{L}}\langle \mathfrak{l}_c, \mathfrak{l}_h \rangle) \xrightarrow{\mathsf{alloc}} (\sigma'_z; \mathfrak{l}'_z) \quad (\sigma'_z; {}^{\mathsf{L}}\mathtt{pack}(\mathfrak{r}', \mathfrak{l}'_z)) \xrightarrow{\mathsf{alloc}} (\sigma'; \mathfrak{l}')}{(\sigma; H; {}^{\mathfrak{q}_c, \mathfrak{q}_h}\mathtt{newrgn}) \longmapsto (\sigma'; H'; \mathfrak{l}')}$$

$$\frac{(\sigma; \mathfrak{l}_c) \xrightarrow{\mathsf{fetch}} (\sigma_c; {}^{\mathfrak{q}_c}(\mathtt{cap})) \quad (\sigma_c; \mathfrak{l}_h) \xrightarrow{\mathsf{fetch}} (\sigma_h; {}^{\mathfrak{q}_h}(\mathtt{hnd}\ \mathfrak{r})) \quad (H; \mathfrak{r}) \xrightarrow{\mathsf{freergn}} H' \quad (\sigma_h; {}^{\mathsf{L}}\langle\rangle) \xrightarrow{\mathsf{alloc}} (\sigma'; \mathfrak{l}')}{(\sigma; H; \mathtt{freergn}\ \mathfrak{l}_c\ \mathfrak{l}_h) \longmapsto (\sigma'; H'; \mathfrak{l}')}$$

Figure C.10: Dynamic semantics of rgnURAL (expressions (IV))

fetches a value from the store. The rules for expression forms other than the region and reference primitives all return the heap H unchanged. We use evaluation contexts E (Figure C.9) to lift the base rewriting rules to a standard, left-to-right, innermost-to-outermost, call-by-value interpretation of the language.

The rules for $(\sigma; H; e) \longmapsto (\sigma'; H'; e')$ for the region and reference primitives perform operations that side-effect the store and the heap. The rules mostly behave as their counterparts in Section 4.2.2, except that they use $(\sigma; v) \xrightarrow{\mathsf{alloc}} (\sigma'; \mathfrak{l}')$ and $(\sigma; \mathfrak{l}) \xrightarrow{\mathsf{fetch}} (\sigma'; v)$ to manipulate references, handles, and capabilities.

Recall that the operational behavior of `new`, `free`, `read`, `write`, and `swap` is to thread a ${}^{\mathfrak{q}_c}(\overline{\mathsf{Cap}}\ \mathfrak{r})$ value through the evaluation; furthermore, `read`, `write`, and `swap` thread a ${}^{\mathfrak{q}_r}(\overline{\mathsf{Ref}}\ \mathfrak{r}\ \tau)$ value through the evaluation. In Figures C.10 and C.11, we may see that this threading is accomplished by fetching the value from the argument location and reallocating the value, so that the value remains available for future use (at a fresh location).

$\boxed{(\sigma; H; e) \longmapsto (\sigma'; H'; e')}$

$$\frac{(\sigma; \mathfrak{l}_c) \xrightarrow{\mathsf{fetch}} (\sigma_c; {}^{\mathfrak{q}_c}(\mathtt{cap}))\qquad (\sigma_c; \mathfrak{l}_h) \xrightarrow{\mathsf{fetch}} (\sigma_h; {}^{\mathfrak{q}_h}(\mathtt{hnd}\ \mathfrak{r}))}{(H; \mathfrak{r}; \mathfrak{q}_r; \mathfrak{l}_a) \xrightarrow{\mathsf{new}} (H'; \mathfrak{p}')\qquad (\sigma_h; {}^{\mathfrak{q}_c}(\mathtt{cap})) \xrightarrow{\mathsf{alloc}} (\sigma'_c; \mathfrak{l}'_c)}{(\sigma'_c; {}^{\mathfrak{q}_r}(\mathtt{ref}\ \mathfrak{r}\ \mathfrak{p}')) \xrightarrow{\mathsf{alloc}} (\sigma'_p; \mathfrak{l}'_r)\qquad (\sigma'_p; {}^{\mathsf{L}}\langle \mathfrak{l}'_c, \mathfrak{l}'_r\rangle) \xrightarrow{\mathsf{alloc}} (\sigma'; \mathfrak{l}')}{(\sigma; H; {}^{\mathfrak{q}_r}\mathtt{new}\ \mathfrak{l}_c\ \mathfrak{l}_h\ \mathfrak{l}_a) \longmapsto (\sigma'; H'; \mathfrak{l}')}$$

$$\frac{(\sigma; \mathfrak{l}_c) \xrightarrow{\mathsf{fetch}} (\sigma_c; {}^{\mathfrak{q}_c}(\mathtt{cap}))\qquad (\sigma_c; \mathfrak{l}_r) \xrightarrow{\mathsf{fetch}} (\sigma_r; {}^{\mathfrak{q}_r}(\mathtt{ref}\ \mathfrak{r}\ \mathfrak{p}))}{(H; \mathfrak{r}; \mathfrak{p}) \xrightarrow{\mathsf{free}} (H'; \mathfrak{l})\qquad (\sigma_r; {}^{\mathfrak{q}_c}(\mathtt{cap})) \xrightarrow{\mathsf{alloc}} (\sigma'_c; \mathfrak{l}'_c)\qquad (\sigma'_c; {}^{\mathsf{L}}\langle \mathfrak{l}'_c, \mathfrak{l}\rangle) \xrightarrow{\mathsf{alloc}} (\sigma'; \mathfrak{l}')}{(\sigma; H; \mathtt{free}\ \mathfrak{l}_c\ \mathfrak{l}_r) \longmapsto (\sigma'; H'; \mathfrak{l}')}$$

$$\frac{\begin{array}{c}(\sigma; \mathfrak{l}_c) \xrightarrow{\mathsf{fetch}} (\sigma_c; {}^{\mathfrak{q}_c}(\mathtt{cap}))\\ (\sigma_c; \mathfrak{l}_r) \xrightarrow{\mathsf{fetch}} (\sigma_r; {}^{\mathfrak{q}_r}(\mathtt{ref}\ \mathfrak{r}\ \mathfrak{p}))\qquad (H; \mathfrak{r}; \mathfrak{p}) \xrightarrow{\mathsf{read}} \mathfrak{l}\qquad (\sigma_r; {}^{\mathfrak{q}_c}(\mathtt{cap})) \xrightarrow{\mathsf{alloc}} (\sigma'_c; \mathfrak{l}'_c)\\ (\sigma'_c; {}^{\mathfrak{q}_r}(\mathtt{ref}\ \mathfrak{r}\ \mathfrak{p})) \xrightarrow{\mathsf{alloc}} (\sigma'_r; \mathfrak{l}'_r)\qquad (\sigma'_r; {}^{\mathsf{L}}\langle \mathfrak{l}'_c, \mathfrak{l}'_r, \mathfrak{l}\rangle) \xrightarrow{\mathsf{alloc}} (\sigma'; \mathfrak{l}')\end{array}}{(\sigma; H; \mathtt{read}\ \mathfrak{l}_c\ \mathfrak{l}_r) \longmapsto (\sigma'; H; \mathfrak{l}')}$$

$$\frac{(\sigma; \mathfrak{l}_c) \xrightarrow{\mathsf{fetch}} (\sigma_c; {}^{\mathfrak{q}_c}(\mathtt{cap}))\qquad (\sigma_c; \mathfrak{l}_r) \xrightarrow{\mathsf{fetch}} (\sigma_r; {}^{\mathfrak{q}_r}(\mathtt{ref}\ \mathfrak{r}\ \mathfrak{p}))}{(H; \mathfrak{r}; \mathfrak{p}; \mathfrak{l}_\star) \xrightarrow{\mathsf{write}} H'\qquad (\sigma_r; {}^{\mathfrak{q}_c}(\mathtt{cap})) \xrightarrow{\mathsf{alloc}} (\sigma'_c; \mathfrak{l}'_c)}{(\sigma'_c; {}^{\mathfrak{q}_r}(\mathtt{ref}\ \mathfrak{r}\ \mathfrak{p})) \xrightarrow{\mathsf{alloc}} (\sigma'_r; \mathfrak{l}'_r)\qquad (\sigma'_c; {}^{\mathsf{L}}\langle \mathfrak{l}'_c, \mathfrak{l}'_r\rangle) \xrightarrow{\mathsf{alloc}} (\sigma'; \mathfrak{l}')}{(\sigma; H; \mathtt{write}\ \mathfrak{l}_c\ \mathfrak{l}_r\ \mathfrak{l}_\star) \longmapsto (\sigma'; H; \mathfrak{l}')}$$

$$\frac{(\sigma; \mathfrak{l}_c) \xrightarrow{\mathsf{fetch}} (\sigma_c; {}^{\mathfrak{q}_c}(\mathtt{cap}))\qquad (\sigma_c; \mathfrak{l}_r) \xrightarrow{\mathsf{fetch}} (\sigma_r; {}^{\mathfrak{q}_r}(\mathtt{ref}\ \mathfrak{r}\ \mathfrak{p}))}{(H; \mathfrak{r}; \mathfrak{p}; \mathfrak{l}_\star) \xrightarrow{\mathsf{swap}} (H'; \mathfrak{l})\qquad (\sigma_r; {}^{\mathfrak{q}_c}(\mathtt{cap})) \xrightarrow{\mathsf{alloc}} (\sigma'_c; \mathfrak{l}'_c)}{(\sigma'_c; {}^{\mathfrak{q}_r}(\mathtt{ref}\ \mathfrak{r}\ \mathfrak{p})) \xrightarrow{\mathsf{alloc}} (\sigma'_r; \mathfrak{l}'_r)\qquad (\sigma'_c; {}^{\mathsf{L}}\langle \mathfrak{l}'_c, \mathfrak{l}'_r, \mathfrak{l}\rangle) \xrightarrow{\mathsf{alloc}} (\sigma'; \mathfrak{l}')}{(\sigma; H; \mathtt{swap}\ \mathfrak{l}_c\ \mathfrak{l}_r\ \mathfrak{l}_\star) \longmapsto (\sigma'; H'; \mathfrak{l}')}$$

Figure C.11: Dynamic semantics of rgnURAL (expressions (V))

$\boxed{(\sigma; H; e) \longmapsto (\sigma'; H'; e')}$

$$\frac{(\sigma; {}^{\mathsf{q}}(\Lambda\varrho.\, e)) \xrightarrow{\mathsf{alloc}} (\sigma'; \mathfrak{l}')}{(\sigma; H; {}^{\mathsf{q}}(\Lambda\varrho.\, e)) \longmapsto (\sigma'; H; \mathfrak{l}')} \qquad \frac{(\sigma; \mathfrak{l}_f) \xrightarrow{\mathsf{fetch}} (\sigma'; {}^{\text{-}}(\lambda\varrho.\, e_b))}{(\sigma; H; \mathfrak{l}_f\ [\rho_2]) \longmapsto (\sigma'; H; e_b[\rho_2/\varrho])}$$

$$\frac{(\sigma; {}^{\mathsf{q}}\mathtt{pack}(\rho, \mathfrak{l})) \xrightarrow{\mathsf{alloc}} (\sigma'; \mathfrak{l}')}{(\sigma; H; {}^{\mathsf{q}}\mathtt{pack}(\rho, \mathfrak{l})) \longmapsto (\sigma'; H; \mathfrak{l}')}$$

$$\frac{(\sigma; \mathfrak{l}_a) \xrightarrow{\mathsf{fetch}} (\sigma'; {}^{\text{-}}\mathtt{pack}(\rho, \mathfrak{l}_x))}{(\sigma; H; \mathtt{let\ pack}(\varrho, x) = \mathfrak{l}_a\ \mathtt{in}\ e_b) \longmapsto (\sigma'; H; e_b[\rho/\varrho][\mathfrak{l}_x/x])}$$

Figure C.12: Dynamic semantics of rgnURAL (expressions (VI))

C.1.2 Static Semantics of rgnURAL

Section 4.2.3 gave the static semantics for the surface syntax of rgnURAL. However, the judgments given in that section are insufficient for carrying out a syntactic proof of type soundness, since there are no rules for $\mathfrak{l}$, `ref`, `hnd`, or `cap` terms (which arise during the evaluation of a program) and there are no typing judgments for stores, regions, or heaps. This section extends the static semantics of Section 4.2.3 to overcome these deficiencies. In addition to the typing judgments for expressions and various well-formedness judgments for qualifiers, pre-types, types, regions, and contexts given previously, we have judgments that check the type and consistency of stores, regions, and heaps.

Definitions Figure C.13 presents additional definitions for syntactic forms that appear in the static semantics. Store, region, and heap types mimic the corresponding objects from the dynamic semantics. A store type Σ records the type of

Qualifier, pre-type, type, region contexts

$$\begin{array}{lrcl}
 & \Delta & ::= & \cdot \mid \Delta, \xi \mid \Delta, \overline{\alpha} \mid \Delta, \alpha \mid \Delta, \varrho \\
\text{Value contexts} & \Gamma & ::= & \cdot \mid \Gamma, x{:}\tau \\
\text{Store types} & \Sigma & ::= & \{\mathfrak{l}_1 \mapsto \tau_1, \ldots, \mathfrak{l}_n \mapsto \tau_n\} \\
\text{Region types} & \mathcal{R} & ::= & \{\mathfrak{p}_1 \mapsto (\mathfrak{q}_1, \tau_1), \ldots, \mathfrak{p}_n \mapsto (\mathfrak{q}_n, \tau_n)\} \\
\text{Region tokens} & \Upsilon & ::= & {}^{\mathfrak{q}}\mathsf{pre} \mid \mathsf{abs} \\
\text{Heap types} & \mathcal{H} & ::= & \{\mathfrak{r}_1 \mapsto (\Upsilon_1, \mathcal{R}_1), \ldots, \mathfrak{r}_n \mapsto (\Upsilon_n, \mathcal{R}_n)\}
\end{array}$$

Figure C.13: Static semantics of rgnURAL (definitions)

the value stored at each location. A heap type $\mathcal{H}$ records, for each region name $\mathfrak{r}$, a region token Υ and a region type. The region token records the presence (pre) or absence (abs) of the capability for $\mathfrak{r}$ in an object type checked under $\mathcal{H}$; when type checking a heap, the region token must also match the region mark. Finally, a region type $\mathcal{R}$ records, for each pointer $\mathfrak{p}$, a qualifier for the corresponding `ref` $\mathfrak{r}$ $\mathfrak{p}$ pre-value and a type for the contents of the reference.

Qualifier order As before, in order to ensure the correct relationship between a data structure and its components, we extend the partial order on constant qualifiers to arbitrary qualifiers, types, and value contexts (see Figure C.14).

We must also extend the partial order to the other type-like syntactic objects: store types, region tokens, region types, and heap types. Figure C.15 presents the rule for the judgment $\Delta \vdash \Sigma \preceq q'$, which simply requires that each type τ in the range of Σ is bounded by q'.

Figure C.16 presents the rules for the judgments $\vdash \Upsilon \sqsubseteq \mathfrak{q}'$, $\vdash \mathcal{R} \sqsubseteq \mathfrak{q}'$, $\vdash (\Upsilon, \mathcal{R}) \sqsubseteq \mathfrak{q}'$, and $\vdash \mathcal{H} \sqsubseteq \mathfrak{q}'$. Since heap types will only be used to type check

$\boxed{\Delta \vdash q \preceq q'}$

$$\frac{\Delta \vdash_{\text{qual}} q}{\Delta \vdash \mathsf{U} \preceq q} \qquad \frac{\mathfrak{q}_1 \sqsubseteq \mathfrak{q}_2}{\Delta \vdash \mathfrak{q}_1 \preceq \mathfrak{q}_2} \qquad \frac{\Delta \vdash_{\text{qual}} q}{\Delta \vdash q \preceq \mathsf{L}}$$

$$\frac{\Delta \vdash_{\text{qual}} q}{\Delta \vdash q \preceq q} \qquad \frac{\Delta \vdash q_1 \preceq q_2 \qquad \Delta \vdash q_2 \preceq q_3}{\Delta \vdash q_1 \preceq q_2}$$

$\boxed{\Delta \vdash \tau \preceq q'}$

$$\frac{\Delta \vdash_{\text{type}} \tau}{\Delta \vdash \tau \preceq \mathsf{L}} \qquad \frac{\Delta \vdash q \preceq q' \qquad \Delta \vdash_{\text{ptype}} \overline{\tau}}{\Delta \vdash {}^{q}\overline{\tau} \preceq q'}$$

$\boxed{\Delta \vdash \Gamma \preceq q'}$

$$\frac{\Delta \vdash_{\text{qual}} q'}{\Delta \vdash \cdot \preceq q'} \qquad \frac{\Delta \vdash \tau \preceq q' \qquad \Delta \vdash \Gamma \preceq q'}{\Delta \vdash \Gamma, x{:}\tau \preceq q'}$$

Figure C.14: Static semantics of rgnURAL ($\preceq$ (I))

$\boxed{\Delta \vdash \Sigma \preceq q'}$

$$\frac{\Delta \vdash_{\text{qual}} q' \qquad \forall(\mathfrak{l} \mapsto \tau) \in \Sigma.\ \Delta \vdash \tau \preceq q'}{\Delta \vdash \Sigma \preceq q'}$$

Figure C.15: Static semantics of rgnURAL ($\preceq$ (II))

$\boxed{\vdash \Upsilon \sqsubseteq \mathfrak{q}'}$

$$\frac{\mathfrak{q} \sqsubseteq \mathfrak{q}'}{\vdash {}^{\mathfrak{q}}\mathsf{pre} \sqsubseteq \mathfrak{q}'} \qquad \frac{}{\vdash \mathsf{abs} \sqsubseteq \mathfrak{q}'}$$

$\boxed{\vdash \mathcal{R} \sqsubseteq \mathfrak{q}'}$

$$\frac{\forall(\mathfrak{p} \mapsto (\mathfrak{q}, \tau)) \in \mathcal{R}.\ \mathfrak{q} \sqsubseteq \mathfrak{q}'}{\vdash \mathcal{R} \sqsubseteq \mathfrak{q}'}$$

$\boxed{\vdash (\Upsilon, \mathcal{R}) \sqsubseteq \mathfrak{q}'}$

$$\frac{\vdash \Upsilon \sqsubseteq \mathfrak{q}' \qquad \vdash \mathcal{R} \sqsubseteq \mathfrak{q}'}{\vdash (\Upsilon, \mathcal{R}) \sqsubseteq \mathfrak{q}'}$$

$\boxed{\vdash \mathcal{H} \sqsubseteq \mathfrak{q}'}$

$$\frac{\forall \mathfrak{r} \in \mathit{dom}(\mathcal{H}).\ \vdash \mathcal{H}(r) \sqsubseteq \mathfrak{q}'}{\vdash \mathcal{H} \sqsubseteq \mathfrak{q}'}$$

Figure C.16: Static semantics of rgnURAL ($\sqsubseteq$ (III))

$\boxed{\Delta \vdash \Gamma \rightsquigarrow \Gamma_1 \boxdot \Gamma_2}$

$$\frac{}{\Delta \vdash \cdot \rightsquigarrow \cdot \boxdot \cdot} \qquad \frac{\Delta \vdash \Gamma \rightsquigarrow \Gamma_1 \boxdot \Gamma_2}{\Delta \vdash \Gamma, x{:}\tau \rightsquigarrow \Gamma_1, x{:}\tau \boxdot \Gamma_2} \qquad \frac{\Delta \vdash \Gamma \rightsquigarrow \Gamma_1 \boxdot \Gamma_2}{\Delta \vdash \Gamma, x{:}\tau \rightsquigarrow \Gamma_1 \boxdot \Gamma_2, x{:}\tau}$$

$$\textsc{Contr}\ \frac{\Delta \vdash \Gamma \rightsquigarrow \Gamma_1 \boxdot \Gamma_2 \qquad \Delta \vdash \tau \preceq \mathsf{R}}{\Delta \vdash \Gamma, x{:}\tau \rightsquigarrow \Gamma_1, x{:}\tau \boxdot \Gamma_2, x{:}\tau}$$

Figure C.17: Static semantics of rgnURAL ($\boxdot$ (I))

$\boxed{\Delta \vdash \Sigma \rightsquigarrow \Sigma_1 \boxdot \Sigma_2}$

$$\frac{\Sigma = \Sigma_1 \uplus \Sigma_2 \uplus \Sigma' \qquad \Delta \vdash \Sigma' \preceq \mathsf{R}}{\Delta \vdash \Sigma \rightsquigarrow (\Sigma_1 \uplus \Sigma') \boxdot (\Sigma_2 \uplus \Sigma')}$$

Figure C.18: Static semantics of rgnURAL ($\boxdot$ (II))

closed values and stores, the judgments are given without a context Δ. Note that in the judgment $\vdash \mathcal{R} \sqsubseteq \mathfrak{q}'$, the antecedent requires the qualifier of the reference (not the type of the contents of the reference) to be bounded by $\mathfrak{q}'$.

Context, store, heap, and region splitting Figure C.17 recalls the value-context splitting judgment given in Section 4.2.3. Splitting the context is necessary to ensure that variables are used appropriately by sub-expressions.

In addition, Figures C.18 and C.19 presents judgments that split store types, region tokens, region types, and heap types. As with value-context splitting, splitting these type-like syntactic objects is necessary to ensure that locations, references, handles, and capabilities are used appropriately by component objects in the typing rules.

$\boxed{\vdash \Upsilon \rightsquigarrow \Upsilon_1 \boxdot \Upsilon_2}$

$$\frac{\mathsf{q} \sqsubseteq \mathsf{R}}{\vdash {}^{\mathsf{q}}\mathsf{pre} \rightsquigarrow {}^{\mathsf{q}}\mathsf{pre} \boxdot {}^{\mathsf{q}}\mathsf{pre}} \qquad \frac{}{\vdash {}^{\mathsf{q}}\mathsf{pre} \rightsquigarrow \mathsf{abs} \boxdot {}^{\mathsf{q}}\mathsf{pre}} \qquad \frac{}{\vdash {}^{\mathsf{q}}\mathsf{pre} \rightsquigarrow {}^{\mathsf{q}}\mathsf{pre} \boxdot \mathsf{abs}}$$

$$\frac{}{\vdash \mathsf{abs} \rightsquigarrow \mathsf{abs} \boxdot \mathsf{abs}}$$

$\boxed{\vdash \mathcal{R} \rightsquigarrow \mathcal{R}_1 \boxdot \mathcal{R}_2}$

$$\frac{\mathcal{R} = \mathcal{R}_1 \uplus \mathcal{R}_2 \uplus \mathcal{R}' \qquad \vdash \mathcal{R}' \sqsubseteq \mathsf{R}}{\vdash \mathcal{R} \rightsquigarrow (\mathcal{R}_1 \uplus \mathcal{R}') \boxdot (\mathcal{R}_2 \uplus \mathcal{R}')}$$

$\boxed{\vdash (\Upsilon, \mathcal{R}) \rightsquigarrow (\Upsilon_1, \mathcal{R}_1) \boxdot (\Upsilon_2, \mathcal{R}_2)}$

$$\frac{\vdash \Upsilon \rightsquigarrow \Upsilon_1 \boxdot \Upsilon_2 \qquad \vdash \mathcal{R} \rightsquigarrow \mathcal{R}_1 \boxdot \mathcal{R}_2}{\vdash (\Upsilon, \mathcal{R}) \rightsquigarrow (\Upsilon_1, \mathcal{R}_1) \boxdot (\Upsilon_2, \mathcal{R}_2)}$$

$\boxed{\vdash \mathcal{H} \rightsquigarrow \mathcal{H}_1 \boxdot \mathcal{H}_2}$

$$\frac{\mathcal{H} = \mathcal{H}_1 \uplus \mathcal{H}_2 \uplus \mathcal{H}' \qquad dom(\mathcal{H}') = dom(\mathcal{H}'_1) = dom(\mathcal{H}'_2) \qquad \forall \mathfrak{r}' \in dom(\mathcal{H}').\ \vdash \mathcal{H}'(\mathfrak{r}') \rightsquigarrow \mathcal{H}'_1(\mathfrak{r}') \boxdot \mathcal{H}'_2(\mathfrak{r}')}{\vdash \mathcal{H} \rightsquigarrow (\mathcal{H}_1 \uplus \mathcal{H}'_1) \boxdot (\mathcal{H}_2 \uplus \mathcal{H}'_2)}$$

Figure C.19: Static semantics of rgnURAL ($\boxdot$ (III))

In the rule for store-type splitting, the antecedent $\Delta \vdash \Sigma' \preceq \mathsf{R}$ allows a set of unrestricted and relevant assumptions to be duplicated; hence, it acts like a strengthened **Contraction** property at the level of store types.

The rule for heap-type splitting is more subtle. The source heap type $\mathcal{H}$ is split into three disjoint components ($\mathcal{H}_1$, $\mathcal{H}_2$, and $\mathcal{H}'$); however, rather than sharing a common heap type $\mathcal{H}'$, the output heaps ($\mathcal{H}_1 \uplus \mathcal{H}'_1$ and $\mathcal{H}_2 \uplus \mathcal{H}'_2$) are defined so that on their common domains, they correspond to the pairwise splitting of the region tokens and region types from the heap type $\mathcal{H}'$.

Like the rule for store-type splitting, the rule for region-type splitting allows unrestricted and relevant assumptions to be duplicated; hence, it also acts like a strengthened **Contraction** property at the level of region types. Finally, the rules for the judgment $\vdash \Upsilon \rightsquigarrow \Upsilon_1 \boxdot \Upsilon_2$ ensure that an affine or linear pre token (corresponding to an affine or linear region capability) cannot be duplicated, but an unrestricted or linear pre token can be duplicated.

Qualifiers, pre-types, types, and regions Figure C.20 contains (completely standard) judgments for ensuring that qualifiers q, pre-types $\overline{\tau}$, types τ, and regions ρ are well-formed. These judgments simply enforce the invariant that no type or expression may depend upon unbound qualifier, pre-type, type, or region variables. Note that in the judgement $\Delta \vdash_{\text{region}} \rho$, any region name is considered well-formed.

Terms Figures C.21–C.28 present the typing rules for the judgment $\Delta; \Gamma; \Sigma \vdash_{\text{exp}} e : \tau$, which asserts that under the qualifier, pre-type, type, and region context Δ, the value context Γ, and the store type Σ, the expression e has the type τ. Recall that references, handles, and capabilities are not expression forms; hence, the judgment for expressions has no dependency on a heap type $\mathcal{H}$.

$\boxed{\Delta \vdash_{\text{qual}} q}$

$$\frac{\xi \in dom(\Delta)}{\Delta \vdash_{\text{qual}} \xi} \qquad \frac{}{\Delta \vdash_{\text{qual}} \mathfrak{q}}$$

$\boxed{\Delta \vdash_{\text{ptype}} \overline{\tau}}$

$$\frac{\overline{\alpha} \in dom(\Delta)}{\Delta \vdash_{\text{ptype}} \overline{\alpha}} \qquad \frac{}{\Delta \vdash_{\text{ptype}} \overline{\mathsf{Int}}} \qquad \frac{}{\Delta \vdash_{\text{ptype}} \overline{\mathsf{Bool}}} \qquad \frac{\Delta \vdash_{\text{type}} \tau_1 \qquad \Delta \vdash_{\text{type}} \tau_2}{\Delta \vdash_{\text{ptype}} \tau_1 \multimap \tau_2}$$

$$\frac{\Delta \vdash_{\text{type}} \tau_i \quad {}^{i \in 1 \ldots n}}{\Delta \vdash_{\text{ptype}} \tau_1 \otimes \cdots \otimes \tau_n} \qquad \frac{\Delta, \xi \vdash_{\text{type}} \tau}{\Delta \vdash_{\text{ptype}} \overline{\forall} \xi.\, \tau} \qquad \frac{\Delta, \xi \vdash_{\text{type}} \tau}{\Delta \vdash_{\text{ptype}} \overline{\exists} \xi.\, \tau} \qquad \frac{\Delta, \overline{\alpha} \vdash_{\text{type}} \tau}{\Delta \vdash_{\text{ptype}} \overline{\forall} \overline{\alpha}.\, \tau}$$

$$\frac{\Delta, \overline{\alpha} \vdash_{\text{type}} \tau}{\Delta \vdash_{\text{ptype}} \overline{\exists} \overline{\alpha}.\, \tau} \qquad \frac{\Delta, \alpha \vdash_{\text{type}} \tau}{\Delta \vdash_{\text{ptype}} \overline{\forall} \alpha.\, \tau} \qquad \frac{\Delta, \alpha \vdash_{\text{type}} \tau}{\Delta \vdash_{\text{ptype}} \overline{\exists} \alpha.\, \tau} \qquad \frac{\Delta \vdash_{\text{region}} \rho \qquad \Delta \vdash_{\text{type}} \tau}{\Delta \vdash_{\text{ptype}} \overline{\mathsf{Ref}}\ \rho\ \tau}$$

$$\frac{\Delta \vdash_{\text{region}} \rho}{\Delta \vdash_{\text{ptype}} \overline{\mathsf{Hnd}}\ \rho} \qquad \frac{\Delta \vdash_{\text{region}} \rho}{\Delta \vdash_{\text{ptype}} \overline{\mathsf{Cap}}\ \rho} \qquad \frac{\Delta, \varrho \vdash_{\text{type}} \tau}{\Delta \vdash_{\text{ptype}} \overline{\forall} \varrho.\, \tau} \qquad \frac{\Delta, \varrho \vdash_{\text{type}} \tau}{\Delta \vdash_{\text{ptype}} \overline{\exists} \varrho.\, \tau}$$

$\boxed{\Delta \vdash_{\text{type}} \tau}$

$$\frac{\alpha \in dom(\Delta)}{\Delta \vdash_{\text{type}} \alpha} \qquad \frac{\Delta \vdash_{\text{qual}} q \qquad \Delta \vdash_{\text{ptype}} \overline{\tau}}{\Delta \vdash_{\text{type}} {}^{q}\overline{\tau}}$$

$\boxed{\Delta \vdash_{\text{region}} \rho}$

$$\frac{\varrho \in dom(\Delta)}{\Delta \vdash_{\text{region}} \varrho} \qquad \frac{}{\Delta \vdash_{\text{region}} \mathfrak{r}}$$

Figure C.20: Static semantics of rgnURAL (qualifiers, pre-types, types, and regions)

$\boxed{\Delta; \Gamma; \Sigma \vdash_{\text{exp}} e : \tau}$

$$\frac{\Delta \vdash_{\text{qual}} q}{\Delta; \cdot; \{\} \vdash_{\text{exp}} {}^{q}\mathfrak{i} : {}^{q}\overline{\mathsf{Int}}}$$

$$\frac{\Delta \vdash_{\text{qual}} q \qquad \Delta \vdash \Gamma \rightsquigarrow \Gamma_1 \boxdot \Gamma_2 \qquad \Delta \vdash \Sigma \rightsquigarrow \Sigma_1 \boxdot \Sigma_2 \qquad \Delta; \Gamma_1; \Sigma_1 \vdash_{\text{exp}} e_1 : {}^{q_1}\overline{\mathsf{Int}} \qquad \Delta; \Gamma_2; \Sigma_2 \vdash_{\text{exp}} e_2 : {}^{q_2}\overline{\mathsf{Int}}}{\Delta; \Gamma; \Sigma \vdash_{\text{exp}} {}^{q}(e_1 \oplus e_2) : {}^{q}\overline{\mathsf{Int}}}$$

$$\frac{\Delta \vdash_{\text{qual}} q \qquad \Delta \vdash \Gamma \rightsquigarrow \Gamma_1 \boxdot \Gamma_2 \qquad \Delta \vdash \Sigma \rightsquigarrow \Sigma_1 \boxdot \Sigma_2 \qquad \Delta; \Gamma_1; \Sigma_1 \vdash_{\text{exp}} e_1 : {}^{q_1}\overline{\mathsf{Int}} \qquad \Delta; \Gamma_2; \Sigma_2 \vdash_{\text{exp}} e_2 : {}^{q_2}\overline{\mathsf{Int}}}{\Delta; \Gamma; \Sigma \vdash_{\text{exp}} {}^{q}(e_1 \lessgtr e_2) : {}^{q}\overline{\mathsf{Bool}}}$$

$$\frac{\Delta \vdash_{\text{qual}} q}{\Delta; \cdot; \{\} \vdash_{\text{exp}} {}^{q}\mathfrak{b} : {}^{q}\overline{\mathsf{Bool}}}$$

$$\frac{\Delta \vdash \Gamma \rightsquigarrow \Gamma_1 \boxdot \Gamma_2 \qquad \Delta \vdash \Sigma \rightsquigarrow \Sigma_1 \boxdot \Sigma_2 \qquad \Delta; \Gamma_1; \Sigma_1 \vdash_{\text{exp}} e_b : {}^{q}\overline{\mathsf{Bool}} \qquad \Delta; \Gamma_2; \Sigma_2 \vdash_{\text{exp}} e_t : \tau \qquad \Delta; \Gamma_2; \Sigma_2 \vdash_{\text{exp}} e_f : \tau}{\Delta; \Gamma; \Sigma \vdash_{\text{exp}} \texttt{if } e_b \texttt{ then } e_t \texttt{ else } e_f : \tau}$$

Figure C.21: Static semantics of rgnURAL (expressions (I))

$\boxed{\Delta; \Gamma; \Sigma \vdash_{\text{exp}} e : \tau}$

$$\frac{\Delta \vdash_{\text{type}} \tau}{\Delta; \cdot, x{:}\tau; \{\} \vdash_{\text{exp}} x : \tau}$$

WEAK

$$\frac{\Delta \vdash \Gamma \rightsquigarrow \Gamma_1 \boxdot \Gamma_2 \qquad \Delta \vdash \Gamma_1 \preceq \mathsf{A} \qquad \Delta; \Gamma_2; \Sigma \vdash e : \tau}{\Delta; \Gamma; \Sigma \vdash_{\text{exp}} e : \tau}$$

$$\frac{\Delta \vdash_{\text{qual}} q \qquad \Delta \vdash \Gamma \preceq q \qquad \Delta \vdash \Sigma \preceq q \qquad \Delta; \Gamma, x{:}\tau_x; \Sigma \vdash_{\text{exp}} e : \tau}{\Delta; \Gamma; \Sigma \vdash_{\text{exp}} {}^{q}\lambda x{:}\tau_x.\, e : {}^{q}(\tau_x \multimap \tau)}$$

$$\frac{\Delta \vdash \Gamma \rightsquigarrow \Gamma_f \boxdot \Gamma_a \qquad \Delta \vdash \Sigma \rightsquigarrow \Sigma_f \boxdot \Sigma_a \qquad \Delta; \Gamma_f; \Sigma_f \vdash_{\text{exp}} e_f : {}^{q}(\tau_x \multimap \tau) \qquad \Delta; \Gamma_a; \Sigma_a \vdash_{\text{exp}} e_a : \tau_x}{\Delta; \Gamma; \Sigma \vdash_{\text{exp}} e_f\ e_a : \tau}$$

$$\frac{\Delta \vdash_{\text{qual}} q \qquad \Delta \vdash \Gamma \rightsquigarrow \Gamma_1 \boxdot \cdots \boxdot \Gamma_n \qquad \Delta \vdash \Sigma \rightsquigarrow \Sigma_1 \boxdot \cdots \boxdot \Sigma_n \qquad \Delta; \Gamma_i; \Sigma_i \vdash_{\text{exp}} e_i : \tau_i \quad {}^{i \in 1 \ldots n} \qquad \Delta \vdash \tau_i \preceq q \quad {}^{i \in 1 \ldots n}}{\Delta; \Gamma; \Sigma \vdash_{\text{exp}} {}^{q}\langle e_1, \ldots, e_n \rangle : {}^{q}(\tau_1 \otimes \cdots \otimes \tau_n)}$$

$$\frac{\Delta \vdash \Gamma \rightsquigarrow \Gamma_a \boxdot \Gamma_b \qquad \Delta \vdash \Sigma \rightsquigarrow \Sigma_a \boxdot \Sigma_b \qquad \Delta; \Gamma_a; \Sigma_a \vdash e_a : {}^{q}(\tau_1 \otimes \cdots \otimes \tau_n) \qquad \Delta; \Gamma_b, x_1{:}\tau_1, \ldots, x_n{:}\tau_n; \Sigma_b \vdash_{\text{exp}} e_b : \tau}{\Delta; \Gamma; \Sigma \vdash_{\text{exp}} \texttt{let}\ \langle x_1, \ldots, x_n \rangle = e_a\ \texttt{in}\ e_b : \tau}$$

Figure C.22: Static semantics of rgnURAL (expressions (II))

$\boxed{\Delta; \Gamma; \Sigma \vdash_{\text{exp}} e : \tau}$

$$\frac{\Delta \vdash_{\text{qual}} q \qquad \Delta \vdash \Gamma \preceq q \qquad \Delta \vdash \Sigma \preceq q \qquad \Delta, \xi; \Gamma; \Sigma \vdash_{\text{exp}} e : \tau}{\Delta; \Gamma; \Sigma \vdash_{\text{exp}} {}^{q}\Lambda\xi.\, e : {}^{q}(\overline{\forall}\xi.\, \tau)}$$

$$\frac{\Delta; \Gamma; \Sigma \vdash_{\text{exp}} e_f : {}^{q}(\overline{\forall}\xi.\, \tau) \qquad \Delta \vdash_{\text{qual}} q_a}{\Delta; \Gamma; \Sigma \vdash_{\text{exp}} e_f\ [q_a] : \tau[q_a/\xi]}$$

$$\frac{\Delta \vdash_{\text{qual}} q \qquad \Delta; \Gamma; \Sigma \vdash_{\text{exp}} e_2 : \tau[q_1/\xi] \qquad \Delta \vdash \tau[q_1/\xi] \preceq q}{\Delta; \Gamma; \Sigma \vdash_{\text{exp}} {}^{q}\texttt{pack}(q_1, e_2) : {}^{q}(\overline{\exists}\xi.\, \tau)}$$

$$\frac{\begin{array}{c}\Delta \vdash \Gamma \rightsquigarrow \Gamma_a \boxdot \Gamma_b \qquad \Delta \vdash \Sigma \rightsquigarrow \Sigma_a \boxdot \Sigma_b \\ \Delta; \Gamma_a; \Sigma_a \vdash e_a : {}^{q}(\overline{\exists}\xi.\, \tau_x) \qquad \Delta \vdash_{\text{type}} \tau \qquad \Delta, \xi; \Gamma_b, x{:}\tau_x; \Sigma_b \vdash e_b : \tau\end{array}}{\Delta; \Gamma; \Sigma \vdash_{\text{exp}} \texttt{let pack}(\xi, x) = e_a\ \texttt{in}\ e_b : \tau}$$

$$\frac{\Delta \vdash_{\text{qual}} q \qquad \Delta \vdash \Gamma \preceq q \qquad \Delta \vdash \Sigma \preceq q \qquad \Delta, \overline{\alpha}; \Gamma; \Sigma \vdash_{\text{exp}} e : \tau}{\Delta; \Gamma; \Sigma \vdash_{\text{exp}} {}^{q}\Lambda\overline{\alpha}.\, e : {}^{q}(\overline{\forall}\overline{\alpha}.\, \tau)}$$

$$\frac{\Delta; \Gamma; \Sigma \vdash_{\text{exp}} e_f : {}^{q}(\overline{\forall}\overline{\alpha}.\, \tau) \qquad \Delta \vdash_{\text{ptype}} \overline{\tau}_a}{\Delta; \Gamma; \Sigma \vdash_{\text{exp}} e_f\ [\overline{\tau}_a] : \tau[\overline{\tau}_a/\overline{\alpha}]}$$

Figure C.23: Static semantics of rgnURAL (expressions (III))

$\boxed{\Delta; \Gamma; \Sigma \vdash_{\text{exp}} e : \tau}$

$$\frac{\Delta \vdash_{\text{qual}} q \qquad \Delta; \Gamma; \Sigma \vdash_{\text{exp}} e_2 : \tau[\overline{\tau}_1/\overline{\alpha}] \qquad \Delta \vdash \tau[\overline{\tau}_1/\overline{\alpha}] \preceq q}{\Delta; \Gamma; \Sigma \vdash_{\text{exp}} {}^{q}\texttt{pack}(\overline{\tau}_1, e_2) : {}^{q}(\overline{\exists}\overline{\alpha}.\, \tau)}$$

$$\frac{\begin{array}{c}\Delta \vdash \Gamma \rightsquigarrow \Gamma_a \boxdot \Gamma_b \qquad \Delta \vdash \Sigma \rightsquigarrow \Sigma_a \boxdot \Sigma_b \\ \Delta; \Gamma_a; \Sigma_a \vdash e_a : {}^{q}(\overline{\exists}\overline{\alpha}.\, \tau_x) \qquad \Delta \vdash_{\text{type}} \tau \qquad \Delta, \overline{\alpha}; \Gamma_b, x{:}\tau_x; \Sigma_b \vdash e_b : \tau\end{array}}{\Delta; \Gamma; \Sigma \vdash_{\text{exp}} \texttt{let pack}(\overline{\alpha}, x) = e_a \texttt{ in } e_b : \tau}$$

$$\frac{\Delta \vdash_{\text{qual}} q \qquad \Delta \vdash \Gamma \preceq q \qquad \Delta \vdash \Sigma \preceq q \qquad \Delta, \alpha; \Gamma; \Sigma \vdash_{\text{exp}} e : \tau}{\Delta; \Gamma; \Sigma \vdash_{\text{exp}} {}^{q}\Lambda\alpha.\, e : {}^{q}(\overline{\forall}\alpha.\, \tau)}$$

$$\frac{\Delta; \Gamma; \Sigma \vdash_{\text{exp}} e_f : {}^{q}(\overline{\forall}\alpha.\, \tau) \qquad \Delta \vdash_{\text{type}} \tau_a}{\Delta; \Gamma; \Sigma \vdash_{\text{exp}} e_f\ [\tau_a] : \tau[\tau_a/\alpha]}$$

$$\frac{\Delta \vdash_{\text{qual}} q \qquad \Delta; \Gamma; \Sigma \vdash_{\text{exp}} e_2 : \tau[\tau_1/\alpha] \qquad \Delta \vdash \tau[\tau_1/\alpha] \preceq q}{\Delta; \Gamma; \Sigma \vdash_{\text{exp}} {}^{q}\texttt{pack}(\tau_1, e_2) : {}^{q}(\overline{\exists}\alpha.\, \tau)}$$

$$\frac{\begin{array}{c}\Delta \vdash \Gamma \rightsquigarrow \Gamma_a \boxdot \Gamma_b \qquad \Delta \vdash \Sigma \rightsquigarrow \Sigma_a \boxdot \Sigma_b \\ \Delta; \Gamma_a; \Sigma_a \vdash_{\text{exp}} e_a : {}^{q}(\overline{\exists}\alpha.\, \tau_x) \qquad \Delta \vdash_{\text{type}} \tau \qquad \Delta, \alpha; \Gamma_b, x{:}\tau_x; \Sigma_b \vdash e_b : \tau\end{array}}{\Delta; \Gamma; \Sigma \vdash_{\text{exp}} \texttt{let pack}(\overline{\alpha}, x) = e_a \texttt{ in } e_b : \tau}$$

$$\frac{\begin{array}{c}\Delta \vdash \Gamma \rightsquigarrow \Gamma_a \boxdot \Gamma_b \\ \Delta \vdash \Sigma \rightsquigarrow \Sigma_a \boxdot \Sigma_b \qquad \Delta; \Gamma_a; \Sigma_a \vdash_{\text{exp}} e_a : \tau_x \qquad \Delta; \Gamma_b, x{:}\tau_x; \Sigma_b \vdash e_b : \tau\end{array}}{\Delta; \Gamma; \Sigma \vdash_{\text{exp}} \texttt{let } x = e_a \texttt{ in } e_b : \tau}$$

Figure C.24: Static semantics of rgnURAL (expressions (IV))

$\boxed{\Delta; \Gamma; \Sigma \vdash_{\text{exp}} e : \tau}$

$$\frac{\Delta \vdash_{\text{qual}} q_c \qquad \Delta \vdash_{\text{qual}} q_h}{\Delta; \cdot; \{\} \vdash_{\text{exp}} {}^{q_c,q_h}\texttt{newrgn} : {}^{\mathsf{L}}(\overline{\exists}\varrho.\, {}^{\mathsf{L}}({}^{q_c}(\overline{\mathsf{Cap}}\ \varrho) \otimes {}^{q_h}(\overline{\mathsf{Hnd}}\ \varrho)))}$$

$$\frac{\begin{array}{c}\Delta \vdash \Gamma \rightsquigarrow \Gamma_c \boxdot \Gamma_h \qquad \Delta \vdash \Sigma \rightsquigarrow \Sigma_c \boxdot \Sigma_h \\ \Delta; \Gamma_c; \Sigma_c \vdash_{\text{exp}} e_c : {}^{q_c}(\overline{\mathsf{Cap}}\ \rho) \qquad \Delta \vdash \mathsf{A} \preceq q_c \qquad \Delta; \Gamma_h; \Sigma_h \vdash_{\text{exp}} e_h : {}^{q_h}(\overline{\mathsf{Hnd}}\ \rho)\end{array}}{\Delta; \Gamma; \Sigma \vdash_{\text{exp}} \texttt{freergn}\ e_c\ e_h : {}^{\mathsf{L}}\overline{1}_{\otimes}}$$

NEW(ANY)

$$\frac{\begin{array}{c}\Delta \vdash_{\text{qual}} q_r \qquad \Delta \vdash \Gamma \rightsquigarrow \Gamma_c \boxdot \Gamma_h \boxdot \Gamma_a \\ \Delta \vdash \Sigma \rightsquigarrow \Sigma_c \boxdot \Sigma_h \boxdot \Sigma_a \qquad \Delta; \Gamma_c; \Sigma_c \vdash_{\text{exp}} e_c : {}^{q_c}(\overline{\mathsf{Cap}}\ \rho) \\ \Delta; \Gamma_h; \Sigma_h \vdash_{\text{exp}} e_h : {}^{q_h}(\overline{\mathsf{Hnd}}\ \rho) \qquad \Delta; \Gamma_a; \Sigma_a \vdash_{\text{exp}} e_\star : \tau \qquad \Delta \vdash \tau \preceq \mathsf{A}\end{array}}{\Delta; \Gamma; \Sigma \vdash_{\text{exp}} {}^{q_r}\texttt{new}\ e_c\ e_h\ e_\star : {}^{\mathsf{L}}({}^{q_c}(\overline{\mathsf{Cap}}\ \rho) \otimes {}^{q_r}(\overline{\mathsf{Ref}}\ \rho\ \tau))}$$

NEW(R,L)

$$\frac{\begin{array}{c}\Delta \vdash_{\text{qual}} q_r \qquad \Delta \vdash \Gamma \rightsquigarrow \Gamma_c \boxdot \Gamma_h \boxdot \Gamma_a \\ \Delta \vdash \Sigma \rightsquigarrow \Sigma_c \boxdot \Sigma_h \boxdot \Sigma_a \qquad \Delta; \Gamma_c; \Sigma_c \vdash_{\text{exp}} e_c : {}^{q_c}(\overline{\mathsf{Cap}}\ \rho) \\ \Delta; \Gamma_h; \Sigma_h \vdash_{\text{exp}} e_h : {}^{q_h}(\overline{\mathsf{Hnd}}\ \rho) \qquad \Delta; \Gamma_a; \Sigma_a \vdash_{\text{exp}} e_\star : \tau \qquad \Delta \vdash \mathsf{R} \preceq q_r\end{array}}{\Delta; \Gamma; \Sigma \vdash_{\text{exp}} {}^{q_r}\texttt{new}\ e_c\ e_h\ e_\star : {}^{\mathsf{L}}({}^{q_c}(\overline{\mathsf{Cap}}\ \rho) \otimes {}^{q_r}(\overline{\mathsf{Ref}}\ \rho\ \tau))}$$

$$\frac{\begin{array}{c}\Delta \vdash \Gamma \rightsquigarrow \Gamma_c \boxdot \Gamma_r \qquad \Delta \vdash \Sigma \rightsquigarrow \Sigma_c \boxdot \Sigma_r \\ \Delta; \Gamma_c; \Sigma_c \vdash_{\text{exp}} e_c : {}^{q_c}(\overline{\mathsf{Cap}}\ \rho) \qquad \Delta; \Gamma_r; \Sigma_r \vdash_{\text{exp}} e_r : {}^{q_r}(\overline{\mathsf{Ref}}\ \rho\ \tau) \qquad \Delta \vdash \mathsf{A} \preceq q_r\end{array}}{\Delta; \Gamma; \Sigma \vdash_{\text{exp}} \texttt{free}\ e_c\ e_r : {}^{\mathsf{L}}({}^{q_c}(\overline{\mathsf{Cap}}\ \rho) \otimes \tau)}$$

Figure C.25: Static semantics of rgnURAL (expressions (V))

$\boxed{\Delta; \Gamma; \Sigma \vdash_{\text{exp}} e : \tau}$

$$\frac{\Delta \vdash \Gamma \rightsquigarrow \Gamma_c \boxdot \Gamma_r \qquad \Delta \vdash \Sigma \rightsquigarrow \Sigma_c \boxdot \Sigma_r \qquad \Delta; \Gamma_c; \Sigma_c \vdash_{\text{exp}} e_c : {}^{q_c}(\overline{\mathsf{Cap}}\ \rho) \qquad \Delta; \Gamma_r; \Sigma_r \vdash_{\text{exp}} e_r : {}^{q_r}(\overline{\mathsf{Ref}}\ \rho\ \tau) \qquad \Delta \vdash \tau \preceq \mathsf{R}}{\Delta; \Gamma; \Sigma \vdash_{\text{exp}} \texttt{read}\ e_c\ e_r : {}^{\mathsf{L}}({}^{q_c}(\overline{\mathsf{Cap}}\ \rho) \otimes {}^{q_r}(\overline{\mathsf{Ref}}\ \rho\ \tau) \otimes \tau)}$$

WRITE(WEAK)

$$\frac{\Delta \vdash \Gamma \rightsquigarrow \Gamma_c \boxdot \Gamma_r \boxdot \Gamma_\star \qquad \Delta \vdash \Sigma \rightsquigarrow \Sigma_c \boxdot \Sigma_r \boxdot \Sigma_\star \qquad \Delta; \Gamma_c; \Sigma_c \vdash_{\text{exp}} e_c : {}^{q_c}(\overline{\mathsf{Cap}}\ \rho) \qquad \Delta; \Gamma_r; \Sigma_r \vdash_{\text{exp}} e_r : {}^{q_r}(\overline{\mathsf{Ref}}\ \rho\ \tau) \qquad \Delta \vdash \tau \preceq \qquad \Delta; \Gamma_\star; \Sigma_\star \vdash_{\text{exp}} e_\star : \tau}{\Delta; \Gamma; \Sigma \vdash_{\text{exp}} \texttt{write}\ e_c\ e_r\ e_\star : {}^{\mathsf{L}}({}^{q_c}(\overline{\mathsf{Cap}}\ \rho) \otimes {}^{q_r}(\overline{\mathsf{Ref}}\ \rho\ \tau))}$$

WRITE(STRONG)

$$\frac{\Delta \vdash \Gamma \rightsquigarrow \Gamma_c \boxdot \Gamma_r \boxdot \Gamma_\star \qquad \Delta \vdash \Sigma \rightsquigarrow \Sigma_c \boxdot \Sigma_r \boxdot \Sigma_\star \qquad \Delta; \Gamma_c; \Sigma_c \vdash_{\text{exp}} e_c : {}^{q_c}(\overline{\mathsf{Cap}}\ \rho) \qquad \Delta; \Gamma_r; \Sigma_r \vdash_{\text{exp}} e_r : {}^{q_r}(\overline{\mathsf{Ref}}\ \rho\ \tau) \qquad \Delta \vdash \tau \preceq \mathsf{A} \qquad \Delta; \Gamma_\star; \Sigma_\star \vdash_{\text{exp}} e_\star : \tau_\star \qquad \Delta \vdash \mathsf{A} \preceq q_r \qquad \Delta \vdash \tau_\star \preceq q_r}{\Delta; \Gamma; \Sigma \vdash_{\text{exp}} \texttt{write}\ e_c\ e_r\ e_\star : {}^{\mathsf{L}}({}^{q_c}(\overline{\mathsf{Cap}}\ \rho) \otimes {}^{q_r}(\overline{\mathsf{Ref}}\ \rho\ \tau_\star))}$$

Figure C.26: Static semantics of rgnURAL (expressions (VI))

$\boxed{\Delta; \Gamma; \Sigma \vdash_{\text{exp}} e : \tau}$

SWAP(WEAK)

$$\frac{\begin{array}{c}\Delta \vdash \Gamma \rightsquigarrow \Gamma_c \boxdot \Gamma_r \boxdot \Gamma_\star \\ \Delta \vdash \Sigma \rightsquigarrow \Sigma_c \boxdot \Sigma_r \boxdot \Sigma_\star \qquad \Delta; \Gamma_c; \Sigma_c \vdash_{\text{exp}} e_c : {}^{q_c}(\overline{\mathsf{Cap}}\ \rho) \\ \Delta; \Gamma_r; \Sigma_r \vdash_{\text{exp}} e_r : {}^{q_r}(\overline{\mathsf{Ref}}\ \rho\ \tau) \qquad \Delta; \Gamma_\star; \Sigma_\star \vdash_{\text{exp}} e_\star : \tau\end{array}}{\Delta; \Gamma; \Sigma \vdash_{\text{exp}} \mathtt{swap}\ e_c\ e_r\ e_\star : {}^{\mathsf{L}}({}^{q_c}(\overline{\mathsf{Cap}}\ \rho) \otimes {}^{q_r}(\overline{\mathsf{Ref}}\ \rho\ \tau) \otimes \tau)}$$

SWAP(STRONG)

$$\frac{\begin{array}{c}\Delta \vdash \Gamma \rightsquigarrow \Gamma_c \boxdot \Gamma_r \boxdot \Gamma_\star \qquad \Delta \vdash \Sigma \rightsquigarrow \Sigma_c \boxdot \Sigma_r \boxdot \Sigma_\star \\ \Delta; \Gamma_c; \Sigma_c \vdash_{\text{exp}} e_c : {}^{q_c}(\overline{\mathsf{Cap}}\ \rho) \qquad \Delta; \Gamma_r; \Sigma_r \vdash_{\text{exp}} e_r : {}^{q_r}(\overline{\mathsf{Ref}}\ \rho\ \tau) \\ \Delta; \Gamma_\star; \Sigma_\star \vdash_{\text{exp}} e_\star : \tau_\star \qquad \Delta \vdash \mathsf{A} \preceq q_r \qquad \Delta \vdash \tau_\star \preceq q_r\end{array}}{\Delta; \Gamma; \Sigma \vdash_{\text{exp}} \mathtt{swap}\ e_c\ e_r\ e_\star : {}^{\mathsf{L}}({}^{q_c}(\overline{\mathsf{Cap}}\ \rho) \otimes {}^{q_r}(\overline{\mathsf{Ref}}\ \rho\ \tau_\star) \otimes \tau)}$$

$$\frac{\Delta \vdash_{\text{qual}} q \qquad \Delta \vdash \Gamma \preceq q \qquad \Delta \vdash \Sigma \preceq q \qquad \Delta, \alpha; \Gamma; \Sigma \vdash_{\text{exp}} e : \tau}{\Delta; \Gamma; \Sigma \vdash_{\text{exp}} {}^{q}\Lambda\varrho.\, e : {}^{q}(\overline{\forall}\varrho.\, \tau)}$$

$$\frac{\Delta; \Gamma; \Sigma \vdash_{\text{exp}} e_f : {}^{q}(\overline{\forall}\varrho.\, \tau) \qquad \Delta \vdash_{\text{region}} \rho_a}{\Delta; \Gamma; \Sigma \vdash_{\text{exp}} e_f\ [\rho_a] : \tau[\rho_a/\varrho]}$$

$$\frac{\Delta \vdash_{\text{qual}} q \qquad \Delta; \Gamma; \Sigma \vdash_{\text{exp}} e_2 : \tau[\rho_1/\varrho] \qquad \Delta \vdash \tau[\rho_1/\varrho] \preceq q}{\Delta; \Gamma; \Sigma \vdash_{\text{exp}} {}^{q}\mathtt{pack}(\rho_1, e_2) : {}^{q}(\overline{\exists}\varrho.\, \tau)}$$

$$\frac{\begin{array}{c}\Delta \vdash \Gamma \rightsquigarrow \Gamma_a \boxdot \Gamma_b \qquad \Delta \vdash \Sigma \rightsquigarrow \Sigma_a \boxdot \Sigma_b \\ \Delta; \Gamma_a; \Sigma_a \vdash_{\text{exp}} e_a : {}^{q}(\overline{\exists}\varrho.\, \tau_x) \qquad \Delta \vdash_{\text{type}} \tau \qquad \Delta, \alpha; \Gamma_b, x{:}\tau_x; \Sigma_b \vdash e_b : \tau\end{array}}{\Delta; \Gamma; \Sigma \vdash_{\text{exp}} \mathtt{let}\ \mathtt{pack}(\overline{\varrho}, x) = e_a\ \mathtt{in}\ e_b : \tau}$$

Figure C.27: Static semantics of rgnURAL (expressions (VII))

$\boxed{\Delta; \Gamma; \Sigma \vdash_{\text{exp}} e : \tau}$

$$\frac{\Delta \vdash_{\text{type}} \tau}{\Delta; \cdot; \{\mathfrak{l} \mapsto \tau\} \vdash_{\text{exp}} \mathfrak{l} : \tau}$$

$$\frac{\Delta \vdash \Sigma \rightsquigarrow \Sigma_1 \boxdot \Sigma_2 \qquad \Delta \vdash \Sigma_1 \preceq \mathsf{A} \qquad \Delta; \Gamma; \Sigma_2 \vdash e : \tau}{\Delta; \Gamma; \Sigma \vdash_{\text{exp}} e : \tau}\ \textsc{Weak(Store)}$$

Figure C.28: Static semantics of rgnURAL (expressions (VIII))

Figures C.21–C.27 repeat the rules from Figures 4.17–4.24 in order to demonstrate the manner in which store types are propagated through the rules given in Section 4.2.3. In particular, note that the store type is treated in much the same manner as the value context: it is split in rules with multiple sub-expressions and it is bounded by a qualifier in rules that introduce closures. Like assumptions in the value context, splitting and bounding the store type ensures that each of the L and A values from the store is exclusively "owned" by exactly one sub-expression. Hence, we may see that the store type does not uniformly represent the type of the entire global store; rather, it represents the types of those locations in the global store which are "local" to an expression.

This interpretation is no clearer than in the typing rules for locations (see Figure C.28). Note that the rule demands that the store type expresses only the type of the location in question. Finally, the Weak(Store) rule demonstrates one more point of comparison with the treatment of the value context. This rule splits the store type into a sub-store type used to type the expression e ad a discardable sub-store type consisting of U and A locations that are not required to type the expression. Hence, the Weak(Store) rule acts as a strengthened **Weakening** property at the level of store types.

$\boxed{\Sigma; \mathcal{H} \vdash v : \tau}$

$$\frac{}{\{\}; \{\} \vdash_{\text{val}} {}^{\mathfrak{q}}\mathfrak{i} : {}^{\mathfrak{q}}\overline{\mathsf{Int}}} \qquad \frac{}{\{\}; \{\} \vdash_{\text{val}} {}^{\mathfrak{q}}\mathfrak{b} : {}^{\mathfrak{q}}\overline{\mathsf{Bool}}}$$

$$\frac{\cdot \vdash \Sigma \preceq \mathfrak{q} \qquad \cdot; \cdot, x{:}\tau_x; \Sigma \vdash_{\text{exp}} e : \tau}{\Sigma; \{\} \vdash_{\text{val}} {}^{\mathfrak{q}}(\lambda x{:}\tau_x.\, e) : {}^{\mathfrak{q}}(\tau_x \multimap \tau)}$$

$$\frac{\Delta \vdash \Sigma \rightsquigarrow \{\mathfrak{l}_1 \mapsto \tau_1\} \boxdot \cdots \boxdot \{\mathfrak{l}_n \mapsto \tau_n\} \qquad \cdot \vdash \tau_i \preceq \mathfrak{q}^{\ i \in 1 \ldots n}}{\Sigma; \{\} \vdash_{\text{val}} {}^{\mathfrak{q}}\langle \mathfrak{l}_1, \ldots, \mathfrak{l}_n \rangle : {}^{\mathfrak{q}}(\tau_1 \otimes \cdots \otimes \tau_n)}$$

$$\frac{\cdot \vdash \Sigma \preceq \mathfrak{q} \qquad \cdot, \xi; \cdot; \Sigma \vdash_{\text{exp}} e : \tau}{\Sigma; \{\} \vdash_{\text{val}} {}^{\mathfrak{q}}(\Lambda \xi.\, e) : {}^{\mathfrak{q}}(\overline{\forall} \xi.\, \tau)} \qquad \frac{\cdot \vdash \tau[q_1/\xi] \preceq \mathfrak{q}}{\{\mathfrak{l}_2 \mapsto \tau[q_1/\xi]\}; \{\} \vdash_{\text{val}} {}^{\mathfrak{q}}\mathtt{pack}(q_1, \mathfrak{l}_2) : {}^{\mathfrak{q}}(\overline{\exists} \xi.\, \tau)}$$

$$\frac{\cdot \vdash \Sigma \preceq \mathfrak{q} \qquad \cdot, \overline{\alpha}; \cdot; \Sigma \vdash_{\text{exp}} e : \tau}{\Sigma; \{\} \vdash_{\text{val}} {}^{\mathfrak{q}}(\Lambda \overline{\alpha}.\, e) : {}^{\mathfrak{q}}(\overline{\forall} \overline{\alpha}.\, \tau)} \qquad \frac{\cdot \vdash \tau[\overline{\tau}_1/\overline{\alpha}] \preceq \mathfrak{q}}{\{\mathfrak{l}_2 \mapsto \tau[\overline{\tau}_1/\overline{\alpha}]\}; \{\} \vdash_{\text{val}} {}^{\mathfrak{q}}\mathtt{pack}(\overline{\tau}_1, \mathfrak{l}_2)) : {}^{\mathfrak{q}}(\overline{\exists} \overline{\alpha}.\, \tau)}$$

$$\frac{\cdot \vdash \Sigma \preceq \mathfrak{q} \qquad \cdot, \alpha; \cdot; \Sigma \vdash_{\text{exp}} e : \tau}{\Sigma; \{\} \vdash_{\text{val}} {}^{\mathfrak{q}}(\Lambda \alpha.\, e) : {}^{\mathfrak{q}}(\overline{\forall} \alpha.\, \tau)} \qquad \frac{\cdot \vdash \tau[\tau_1/\alpha] \preceq \mathfrak{q}}{\{\mathfrak{l}_2 \mapsto \tau[\tau_1/\alpha]\}; \{\} \vdash_{\text{val}} {}^{\mathfrak{q}}\mathtt{pack}(\tau_1, \mathfrak{l}_2)) : {}^{\mathfrak{q}}(\overline{\exists} \alpha.\, \tau)}$$

Figure C.29: Static semantics of rgnURAL (values (I))

$\boxed{\Sigma; \mathcal{H} \vdash v : \tau}$

$$\frac{}{\{\}; \{\mathfrak{r} \mapsto (\mathsf{abs}, \{\mathfrak{p} \mapsto (\mathfrak{q}, \tau)\})\} \vdash_{\mathrm{val}} {}^{\mathfrak{q}}(\mathtt{ref}\ \mathfrak{r}\ \mathfrak{p}) : {}^{\mathfrak{q}}(\overline{\mathsf{Ref}}\ \mathfrak{r}\ \tau)}$$

$$\frac{}{\{\}; \{\mathfrak{r} \mapsto (\mathsf{abs}, \{\})\} \vdash_{\mathrm{val}} {}^{\mathfrak{q}}(\mathtt{hnd}\ \mathfrak{r}) : {}^{\mathfrak{q}}(\overline{\mathsf{Hnd}}\ \mathfrak{r})}$$

$$\frac{}{\{\}; \{\mathfrak{r} \mapsto ({}^{\mathfrak{q}}\mathsf{pre}, \{\})\} \vdash_{\mathrm{val}} {}^{\mathfrak{q}}(\mathtt{cap}) : {}^{\mathfrak{q}}(\overline{\mathsf{Cap}}\ \mathfrak{r})} \qquad \frac{\cdot \vdash \Sigma \preceq \mathfrak{q} \qquad \cdot, \varrho; \cdot; \Sigma \vdash_{\mathrm{exp}} e : \tau}{\Sigma; \{\} \vdash_{\mathrm{val}} {}^{\mathfrak{q}}(\Lambda \varrho.\, e) : {}^{\mathfrak{q}}(\overline{\forall} \varrho.\, \tau)}$$

$$\frac{\cdot \vdash \tau[\rho_1/\varrho] \preceq \mathfrak{q}}{\{\mathfrak{l}_2 \mapsto \tau[\rho_1/\varrho]\}; \{\} \vdash_{\mathrm{val}} {}^{\mathfrak{q}}\mathtt{pack}(\rho_1, \mathfrak{l}_2) : {}^{\mathfrak{q}}(\overline{\exists} \varrho.\, \tau)}$$

Figure C.30: Static semantics of rgnURAL (values (II))

Values The typing rules for values are, for the most part, simple adaptations of the typing rules for the corresponding expression forms. Figures C.29 and C.30 present the typing rules for the judgment $\Sigma; \mathcal{H} \vdash_{\mathrm{val}} v : \tau$, which asserts that under the store type Σ and heap type $\mathcal{H}$, the value v has the type τ. Recall that values are closed; hence, the judgment for values has no dependency on a qualifier, pre-type, type, and region context Δ or a value context Γ. Furthermore, values include references, handles, and capabilities; hence, the judgment for values has a dependency on a heap type $\mathcal{H}$.

One significant difference between the typing rules for values and the typing rules for expressions is that the judgment $\Sigma; \mathcal{H} \vdash_{\mathrm{val}} v : \tau$ does not include any rules for weakening the store type Σ or the heap type $\mathcal{H}$.

The typing rules for references, handles, and capabilities in Figure C.30 show that, like the store type Σ, the heap type $\mathcal{H}$ does not uniformly represent the type of

the entire global heap; rather, it represents the portion of the global heap which is "local" to a value. In particular, the rule for a reference $\mathsf{q}(\texttt{ref}\ \mathfrak{r}\ \mathfrak{p})$ demands a heap type of the form $\{\mathfrak{r} \mapsto (\mathsf{abs}, \{\mathfrak{p} \mapsto (\mathfrak{q}, \tau)\})\}$, which expresses only the type of the pointer in question and indicates that the capability for $\mathfrak{r}$ is absent from the value. The rule for a handle $\mathsf{q}(\texttt{hnd}\ \mathfrak{r})$ demands a heap type of the form $\{\mathfrak{r} \mapsto (\mathsf{abs}, \{\})\}$, which indicates that the capability for $\mathfrak{r}$ is absent from the value. Finally, the rule for a capability $\mathsf{q}(\texttt{cap})$ demands a heap of the form $\{\mathfrak{r} \mapsto (\mathsf{q}\mathsf{pre}, \{\})\}$, which indicates that the capability for $\mathfrak{r}$ is present in the value.

Program states, heaps, and stores The recursion between stores and heaps (namely, that stores map locations to values, including reference pointers, and heaps map region names to regions, which map pointers to locations) means that the judgments that assign store types to stores and heap types to heaps are interdependent. In order to keep the system comprehensible, we isolate this interdependence in the typing rule for program states (Figure C.31). Before examining this rule in detail, we introduce the typing judgments:

- $\mathcal{H}' \vdash \sigma : \Sigma$ asserts that, under the heap type $\mathcal{H}'$, the store σ has the store type Σ. As we have noted above, a store type does not uniformly represent the type of the entire store. Here, the store type Σ represents only the types of locations in the store σ that must be used to type check the heap and expression in the program state; in particular, it need not include the types of location used to type check other values in the store.

- $\Sigma' \vdash H : \mathcal{H}$ asserts that under the store type Σ', the heap H has the heap type $\mathcal{H}$. Here, the heap type $\mathcal{H}$ does represent the type of the entire heap that must be used to type check the store in the program state.

$\boxed{\vdash (\sigma; H) : \Sigma' \text{ and } \vdash (\sigma; H; e) : \tau}$

$$\frac{\mathcal{H}_\sigma \vdash \sigma : \Sigma \qquad \Sigma_H \vdash H : \mathcal{H}_\sigma \qquad \cdot \vdash \Sigma \rightsquigarrow \Sigma_H \boxdot \Sigma'}{\vdash (\sigma; H) : \Sigma'}$$

$$\frac{\vdash (\sigma; H) : \Sigma_e \qquad \cdot; \cdot; \Sigma_e \vdash_{\text{exp}} e : \tau}{\vdash (\sigma; H; e) : \tau}$$

Figure C.31: Static semantics of rgnURAL (program state)

- $\vdash (\sigma; \mathcal{H}) : \Sigma'$ asserts that the store and heap are consistent. Here, the store type Σ' represents only the types of locations in the store σ that must be used to type check the expression in the program state; in particular, it need include the types of locations used to type check other values in the store or other locations in the heap.

- $\vdash (\sigma; \mathcal{H}; e) : \tau$ asserts that a complete program state, consisting of a store, heap, and expression, is consistent; furthermore, the expression e has the type τ.

We first consider the typing rule for program states (Figure C.31). The first rule asserts the existence of a heap type ($\mathcal{H}_\sigma$) and three store types (Σ, Σ_H, and Σ'). The heap type $\mathcal{H}_\sigma$ is the type of the heap H and is used to type check the store σ. The store type Σ is the type of the store σ and is split into Σ_H and Σ'; the store type Σ_H is used to type check the heap H, while the store type Σ' is "left over", and will be used to type check the expression in a complete program state. Note that the use of $\mathcal{H}_\sigma$ and Σ_H as both type checking assumptions and as type checking results ensures that the store and heap are consistent.

The second rule simply requires that, in a complete program state, the store typing Σ_e that is "left over" from type checking the store σ and heap H is used to type check the expression e.

Next, we consider the typing rules for the judgment $\Sigma' \vdash H : \mathcal{H}$ and the supporting judgments $\Sigma' \vdash R : \mathcal{R}$, $\vdash \tau \downarrow \mathfrak{q}$, and $\vdash \upsilon : \Upsilon$ (Figure C.32).

The judgment $\vdash \upsilon : \Upsilon$ ensures that a region mark is consistent with its corresponding region token. If the region is allocated ($^{\mathfrak{q}}\texttt{live}$), then a capability for the region must be present (first rule), unless the `newrgn` primitive that allocated the region was annotated as unrestricted or affine ($\mathfrak{q} \sqsubseteq \mathsf{A}$), in which case, the capability for the region may be absent (second rule). If the region is deallocated (`dead`), then the capability for the region must be absent (third rule).

The judgment $\vdash \tau \downarrow \mathfrak{q}$ formalizes the safe combinations of qualifiers for a reference and type for its contents, as were given in Figure 4.22. In particular, it ensures that unrestricted and affine references may only store unrestricted and affine types, while relevant and linear references may store any type.

The judgment $\Sigma' \vdash R : \mathcal{R}$ asserts that, under the store type Σ', the region R has the region type $\mathcal{R}$. An empty region requires an empty store type and yields an empty region type (first rule). A non-empty region may be disjointly split into a region R and a distinguished binding $\{\mathfrak{p} \mapsto (\mathfrak{q}, \mathfrak{l})\}$ (second and third rules). In the second rule, the store type Σ' is split into Σ'_R, which is used to type check the region R, and $\{\mathfrak{l} \mapsto \tau\}$, which gives the type of the contents of the reference; furthermore, the qualifier for the reference and the type for its contents must be a safe combination. In the third rule, an unrestricted or affine reference may have been dropped; hence, we do not add a binding $\{\mathfrak{p} \mapsto (\mathfrak{q}, \tau)\}$ to the region type, since there will be no $^{\mathfrak{q}}(\texttt{ref}\ \mathfrak{r}\ \mathfrak{p})$ value in the program state to use the binding.

$\boxed{\vdash \upsilon : \Upsilon}$

$$\frac{}{\vdash {}^{\mathfrak{q}}\texttt{live} : {}^{\mathfrak{q}}\mathsf{pre}} \qquad \frac{\mathfrak{q} \sqsubseteq \mathsf{A}}{\vdash {}^{\mathfrak{q}}\texttt{live} : \mathsf{abs}} \qquad \frac{}{\vdash \texttt{dead} : \mathsf{abs}}$$

$\boxed{\vdash \tau \downarrow \mathfrak{q}}$

$$\frac{\cdot \vdash \tau \preceq \mathsf{A}}{\vdash \tau \downarrow \mathsf{U}} \qquad \frac{}{\vdash \tau \downarrow \mathsf{R}} \qquad \frac{\cdot \vdash \tau \preceq \mathsf{A}}{\vdash \tau \downarrow \mathsf{A}} \qquad \frac{}{\vdash \tau \downarrow \mathsf{L}}$$

$\boxed{\Sigma' \vdash R : \mathcal{R}}$

$$\frac{}{\{\} \vdash \{\} : \{\}} \qquad \frac{\cdot \vdash \Sigma' \rightsquigarrow \Sigma'_R \boxdot \{\mathfrak{l} \mapsto \tau\} \qquad \Sigma'_R \vdash R : \mathcal{R} \qquad \vdash \tau \downarrow \mathfrak{q}}{\Sigma' \vdash R \uplus \{\mathfrak{p} \mapsto (\mathfrak{q}, \mathfrak{l})\} : \mathcal{R} \uplus \{\mathfrak{p} \mapsto (\mathfrak{q}, \tau)\}}$$

$$\frac{\Sigma' \vdash R : \mathcal{R} \qquad \mathfrak{q} \sqsubseteq \mathsf{A}}{\Sigma' \vdash R \uplus \{\mathfrak{p} \mapsto (\mathfrak{q}, \mathfrak{l})\} : \mathcal{R}}$$

$\boxed{\Sigma' \vdash H : \mathcal{H}}$

$$\frac{}{\{\} \vdash \{\} : \{\}} \qquad \frac{\cdot \vdash \Sigma' \rightsquigarrow \Sigma'_H \boxdot \Sigma'_R \qquad \Sigma'_H \vdash H : \mathcal{H} \qquad \vdash \upsilon : \Upsilon \qquad \Sigma'_R \vdash R : \mathcal{R}}{\Sigma' \vdash H \uplus \{\mathfrak{r} \mapsto (\upsilon, R)\} : \mathcal{H} \uplus \{\mathfrak{r} \mapsto (\Upsilon, \mathcal{R})\}}$$

Figure C.32: Static semantics of rgnURAL (heap)

$\boxed{\mathcal{H}' \vdash \sigma : \Sigma}$

$$\frac{\forall \mathfrak{r}' \in dom(\mathcal{H}').\ \mathcal{H}'(\mathfrak{r}') = (\mathsf{abs}, \{\})}{\mathcal{H}' \vdash \{\} : \{\}}$$

$$\frac{\vdash \mathcal{H}' \rightsquigarrow \mathcal{H}'_\sigma \boxdot \mathcal{H}'_v \qquad \mathcal{H}'_\sigma \vdash \sigma : \Sigma_\sigma \qquad \vdash \Sigma_\sigma \rightsquigarrow \Sigma_v \boxdot \Sigma \qquad \Sigma_v; \mathcal{H}'_v \vdash_{\text{val}} v : \tau}{\mathcal{H}' \vdash \sigma \uplus \{\mathfrak{l} \mapsto (\mathfrak{f}, v)\} : \Sigma \uplus \{\mathfrak{l} \mapsto \tau\}}$$

$$\frac{\mathcal{H}' \vdash \sigma : \Sigma \qquad \mathfrak{q} \sqsubseteq \mathsf{A}}{\mathcal{H}' \vdash \sigma \uplus \{\mathfrak{l} \mapsto (\mathfrak{f}, {}^{\mathfrak{q}}\overline{v})\} : \Sigma} \qquad \frac{\mathcal{H}' \vdash \sigma : \Sigma \qquad \mathfrak{q} \sqsubseteq \mathsf{R}}{\mathcal{H}' \vdash \sigma \uplus \{\mathfrak{l} \mapsto (\mathsf{used}, {}^{\mathfrak{q}}\overline{v})\} : \Sigma}$$

Figure C.33: Static semantics of rgnURAL (store)

Finally, the judgment $\Sigma' \vdash H : \mathcal{H}$ asserts that under the store type Σ', the heap H has the heap type $\mathcal{H}$. An empty heap requires an empty store type and yields an empty heap type (first rule). In the second rule, a non-empty heap is disjointly split into a heap H and a distinguished binding $\{\mathfrak{r} \mapsto (\upsilon, R)\}$. The store type Σ' is split into Σ'_H, which is used to type check the heap H, and Σ'_R, which is used to type check the region R; furthermore, the region mark υ must be consistent with the region token Υ.

Note that the heap type $\mathcal{H}$ has a binding $\{\mathfrak{r} \mapsto (\Upsilon, \mathcal{R})\}$ for every binding $\{\mathfrak{r} \mapsto (\upsilon, R)\}$ in the heap H.

Lastly, we consider the typing rules for the judgment $\mathcal{H}' \vdash \sigma : \Sigma$ (Figure C.33). This judgment asserts that, under the heap type $\mathcal{H}'$, the store σ has the store type Σ. An empty store requires an (effectively) empty heap type and yields an empty store type (first rule). As noted above, since a heap type has a binding for every binding in the heap, the rule for an empty store may not require an empty heap

type $\{\}$. Rather, it requires a heap type that includes only bindings of the form $\{\mathfrak{r}' \mapsto (\mathsf{abs}, \{\})\}$, which is effectively empty.

A non-empty store may be disjointly split into a store σ and a distinguished binding $\{\mathfrak{l} \mapsto (\mathfrak{f}, v)\}$ (second, third, and fourth rules). In the second rule, the heap type $\mathcal{H}'$ is split into $\mathcal{H}'_\sigma$, which is used to type check the store σ, and $\mathcal{H}'_v$, which is used to type check the value v. However, type checking a value requires both a store typing and a heap typing. We find the store typing used to type check the value by splitting the store type Σ_σ (the store type of the store σ) into Σ_v and Σ. Splitting the store type in this manner ensures that an affine or linear value from the store σ is used *either* to type check the value v *or* is propagated to type check another portion of the program state. Hence, this rule ensures that locations for linear values in the store appear exactly once in the program state.

On the other hand, locations for affine values in the store must appear at most once in the program state and locations for unrestricted values may appear an arbitrary number of times (including zero times). The third rule covers this case: we do not add a binding $\{\mathfrak{l} \mapsto \tau\}$ to the store type, since there will be no $\mathfrak{l}$ in the program state to use the binding.

Similarly, locations for relevant values in the store must appear at least once in the program state *until the value is used*; once the value has been used at least once, it need not appear in the program state. The fourth rule covers this case: we do not add a binding $\{\mathfrak{l} \mapsto \tau\}$ to the store type, since there will be no $\mathfrak{l}$ in the program state to use the binding.

Note that this formulation of the static semantics, along with the allocation semantics in Appendix C.1.1, captures much of the behavior we expect from substructurally qualified data structures. In particular, the program state will not

type check unless locations bound to linear values appear exactly once in the program state, locations bound to affine values appear at most, locations bound to relevant values appear at least once until they are used (at which point, they may appear an arbitrary number of times), and locations abound to unrestricted values appear an arbitrary number of times.

Surface syntax We note that store types and heap types are purely technical devices that are used to prove type soundness. Since the surface syntax does not admit explicitly named locations, we can type any closed, surface expression with the judgment $\cdot;\cdot;\cdot \vdash_{\text{exp}} e : \tau$. Pushing this empty store type through the rules leads the type rules in Figures 4.17–4.24 and the type system presented in Section 4.2.3, which is sufficient for type-checking surface programs.

C.2 Type Soundness for rgnURAL

In this section, we sketch a syntactic proof of type soundness [96]. We wish to prove that a well-typed, closed initial program either terminates (returning a location of the correct type) or may continue to take evaluation steps. A progress theorem and a preservation (subject reduction) theorem make this theorem an easy corollary. The proofs are relatively straightforward, albeit tedious; we briefly discuss the most interesting components.

The Progress Theorem states that any well-typed program state, for which the expression is not already a location, may take an evaluation step.

Theorem C.1 (rgnURAL Progress)

If $\vdash (\sigma_1; H_1; e_1) : \tau$ *(i.e., if* $\vdash (\sigma_1; H_1) : \Sigma_1$ *and* $\cdot;\cdot;\Sigma_1 \vdash_{\text{exp}} e_1 : \tau$*), then*

either there exists $\mathfrak{l}$ *such that* $e_1 \equiv \mathfrak{l}$ *or there exists* σ_2 *and* H_2 *and* e_2 *such that* $(\sigma_1; H_1; e_1) \longmapsto (\sigma_2; H_2; e_2)$.

The proof is by induction on the derivation $\cdot; \cdot; \Sigma_1 \vdash_{\text{exp}} e : \tau$. In order to carry out the proof, we require a number of lemmas stating that various store and heap manipulations are implied by the well-typedness of the store and heap.

A representative sample of these lemmas are the following:

Lemma C.2

If $\mathcal{H} \vdash \sigma : \Sigma$, *then there exists* σ' *and* $\mathfrak{l}'$ *such that* $(\sigma; v) \xrightarrow{\mathsf{alloc}} (\sigma'; \mathfrak{l}')$.

Lemma C.3

If $\mathcal{H} \vdash \sigma : \Sigma$ *and* $\cdot \vdash \Sigma \rightsquigarrow \{\mathfrak{l} \mapsto \tau\} \boxdot \Sigma'$, *then there exists* σ', v, *such that* $(\sigma; \mathfrak{l}) \xrightarrow{\mathsf{fetch}} (\sigma'; v)$.

Lemma C.4

If $\Sigma \vdash H : \mathcal{H}'$, *then there exists* H' *and* $\mathfrak{r}'$ *such that* $(H; \mathfrak{q}_\mathfrak{c}) \xrightarrow{\mathsf{newrgn}} (H'; \mathfrak{r}')$.

Lemma C.5

If $\Sigma \vdash H : \mathcal{H}$ *and* $\vdash \mathcal{H} \rightsquigarrow \{\mathfrak{r} \mapsto ({}^{\mathfrak{q}_\mathfrak{c}}\mathsf{pre}, \{\})\} \boxdot \mathcal{H}'$, *then there exists* H' *such that* $(H; \mathfrak{r}) \xrightarrow{\mathsf{freergn}} H'$.

Lemma C.6

If $\Sigma \vdash H : \mathcal{H}$ *and* $\vdash \mathcal{H} \rightsquigarrow \{\mathfrak{r} \mapsto ({}^{\mathfrak{q}_\mathfrak{c}}\mathsf{pre}, \{\mathfrak{p} \mapsto (\mathfrak{q}_r, \tau)\})\} \boxdot \mathcal{H}'$, *then there exists* H' *and* $\mathfrak{l}'$ *such that* $(H; \mathfrak{r}; \mathfrak{p}; \mathfrak{l}_\star) \xrightarrow{\mathsf{swap}} (H'; \mathfrak{l}')$.

The proofs of each of these lemmas is either immediate or by induction on the store-typing derivation or the heap-typing derivation.

The Preservation Theorem states that the evaluation of a well-typed program state yields a program state of the same type.

Theorem C.7 (rgnURAL **Preservation)**

If $(\sigma_1; H_1; e_1) \longmapsto^* (\sigma_2; H_2; e_2)$ *and* $\vdash (\sigma_1; H_1; e_1) : \tau$ *(i.e., if* $\vdash (\sigma_1; H_1) : \Sigma_1$ *and* $\cdot; \cdot; \Sigma_1 \vdash_{\text{exp}} e_1 : \tau$*), then* $\vdash (\sigma_2; H_2; e_2) : \tau$ *(i.e., then* $\vdash (\sigma_2; H_2) : \Sigma_2$ *and* $\cdot; \cdot; \Sigma_2 \vdash_{\text{exp}} e_2 : \tau$*).*

The proof is by induction on the simultaneous order of the derivation $(\sigma_1; H_1; e_1) \longmapsto (\sigma_2; H_2; e_2)$ and the derivation $\cdot; \cdot; \Sigma_1 \vdash_{\text{exp}} e_1 : \tau$. (Induction over the expression typing judgment is only needed for the WEAK rule.) In order to carry out the proof, we require a number of lemmas.

The first group of supporting lemmas state that various store and heap manipulations preserve the well-typedness of the store and heap. A representative sample of these lemmas are the following:

Lemma C.8

If $\mathcal{H}_1 \vdash \sigma_1 : \Sigma_1$*,* $\Sigma_v; \mathcal{H}_v \vdash_{\text{val}} v : \tau$*,* $\cdot \vdash \Sigma_1 \rightsquigarrow \Sigma_v \boxdot \Sigma'$*,* $\vdash \mathcal{H}_2 \rightsquigarrow \mathcal{H}_v \boxdot \mathcal{H}_1$*, and* $(\sigma_1; v) \xrightarrow{\mathsf{alloc}} (\sigma_2; \mathfrak{l})$*, then there exists* Σ_2 *such that* $\mathcal{H}_2 \vdash \sigma_2 : \Sigma_2$ *and* $\cdot \vdash \Sigma_2 \rightsquigarrow \{\mathfrak{l} \mapsto \tau\} \boxdot \Sigma'$.

Lemma C.9

If $\mathcal{H}_1 \vdash \sigma_1 : \Sigma_1$*,* $\cdot \vdash \Sigma_1 \rightsquigarrow \{\mathfrak{l} \mapsto \tau\} \boxdot \Sigma'$*, and* $(\sigma_1; \mathfrak{l}) \xrightarrow{\mathsf{fetch}} (\sigma_2; v)$*, then there exists* $\mathcal{H}_2$*,* Σ_2*,* Σ_v*, and* $\mathcal{H}_v$*, such that* $\mathcal{H}_2 \vdash \sigma_2 : \Sigma_2$*,* $\Sigma_v; \mathcal{H}_v \vdash_{\text{val}} v : \tau$*,* $\cdot \vdash \Sigma_2 \rightsquigarrow \Sigma_v \boxdot \Sigma'$*, and* $\vdash \mathcal{H}_1 \rightsquigarrow \mathcal{H}_v \boxdot \mathcal{H}_2$.

Lemma C.10

If $\Sigma \vdash H_1 : \mathcal{H}_1$ *and* $(H_1; \mathfrak{q}_c) \xrightarrow{\mathsf{newrgn}} (H_2; \mathfrak{r})$*, then there exists* $\mathcal{H}_2$ *such that* $\Sigma \vdash H_2 : \mathcal{H}_2$ *and* $\cdot \vdash \mathcal{H}_2 \rightsquigarrow \{\mathfrak{r} \mapsto ({}^{\mathfrak{q}_c}\mathsf{pre}, \{\})\} \boxdot \mathcal{H}_1$*.*

Lemma C.11

If $\Sigma \vdash H_1 : \mathcal{H}_1$*,* $\mathsf{A} \sqsubseteq \mathfrak{q}_c$*,* $\vdash \mathcal{H}_1 \rightsquigarrow \{\mathfrak{r} \mapsto ({}^{\mathfrak{q}_c}\mathsf{pre}, \{\})\} \boxdot \mathcal{H}'$*, and* $(H_1; \mathfrak{r}) \xrightarrow{\mathsf{freergn}} H_2$*, then there exists* $\mathcal{H}_2$ *such that* $\Sigma \vdash H_2 : \mathcal{H}_2$ *and* $\vdash \mathcal{H}_2 \rightsquigarrow \{\mathfrak{r} \mapsto (\mathsf{abs}, \{\})\} \boxdot \mathcal{H}'$*.*

Lemma C.12

If $\Sigma_1 \vdash H_1 : \mathcal{H}_1$*,* $\mathsf{A} \sqsubseteq \mathfrak{q}_r$*,* $\vdash \tau_\star \downarrow \mathfrak{q}_r$*,*
$\vdash \mathcal{H}_1 \rightsquigarrow \{\mathfrak{r} \mapsto ({}^{\mathfrak{q}_c}\mathsf{pre}, \{\mathfrak{p} \mapsto (\mathfrak{q}_r, \tau)\})\} \boxdot \mathcal{H}'$*,* $\cdot \vdash \Sigma \rightsquigarrow \{\mathfrak{l}_\star \mapsto \tau_\star\} \boxdot \Sigma_1$*, and*
$(H_1; \mathfrak{r}; \mathfrak{p}; \mathfrak{l}_\star) \xrightarrow{\mathsf{swap}} (H_2; \mathfrak{l})$*, then there exists* Σ_2 *and* $\mathcal{H}_2$ *such that*
$\Sigma_2 \vdash H_2 : \mathcal{H}_2$*,* $\vdash \mathcal{H}_2 \rightsquigarrow \{\mathfrak{r} \mapsto ({}^{\mathfrak{q}_c}\mathsf{pre}, \{\mathfrak{p} \mapsto (\mathfrak{q}_r, \tau_\star)\})\} \boxdot \mathcal{H}'$*, and*
$\cdot \vdash \Sigma \rightsquigarrow \{\mathfrak{l} \mapsto \tau\} \boxdot \Sigma_2$*.*

Lemma C.13

If $\Sigma_1 \vdash H_1 : \mathcal{H}$*,* $\vdash \mathcal{H} \rightsquigarrow \{\mathfrak{r} \mapsto ({}^{\mathfrak{q}_c}\mathsf{pre}, \{\mathfrak{p} \mapsto (\mathfrak{q}_r, \tau)\})\} \boxdot \mathcal{H}'$*,*
$\cdot \vdash \Sigma \rightsquigarrow \{\mathfrak{l}_\star \mapsto \tau\} \boxdot \Sigma_1$*, and* $(H_1; \mathfrak{r}; \mathfrak{p}; \mathfrak{l}_\star) \xrightarrow{\mathsf{swap}} (H_2; \mathfrak{l})$*, then there exists* Σ_2 *such that* $\Sigma_2 \vdash H_2 : \mathcal{H}$*, and* $\cdot \vdash \Sigma \rightsquigarrow \{\mathfrak{l} \mapsto \tau\} \boxdot \Sigma_2$*.*

As with the corresponding progress lemmas, the proofs of each of these lemmas is either immediate or by induction on the store-typing derivation or the heap-typing derivation.

The next lemma required by the proof of the Preservation Theorem is a standard substitution lemma. Note that a general substitution (i.e., one that arbitrary expressions for variables), is not type preserving:

False Conjecture C.14

If $\Delta; \Gamma_{e+x}; \Sigma_e \vdash_{\text{exp}} e : \tau$, $\Delta; \Gamma_{e'}; \Sigma_{e'} \vdash_{\text{exp}} e' : \tau'$, $\Delta \vdash \Sigma \rightsquigarrow \Sigma_e \boxdot \Sigma_{e'}$,
$\Delta \vdash \Gamma_{e+e'} \rightsquigarrow \Gamma_e \boxdot \Gamma_{e+e'}$, $\Delta \vdash \Gamma_{e+x} \rightsquigarrow \Gamma_e \boxdot \cdot, x{:}\tau'$, *and* $x \notin dom(\Gamma_e)$, *then*
$\Delta; \Gamma_{e+e'}; \Sigma_{e+e'} \vdash_{\text{exp}} e[e'/x] : \tau$.

As a simple counter example, consider the following instantiation of the conjecture:

$$\cdot; \cdot, x{:}^{\mathsf{U}}\overline{\mathsf{Int}}; \{\} \vdash_{\text{exp}} {}^{\mathsf{L}}\langle x, x\rangle : {}^{\mathsf{L}}({}^{\mathsf{U}}\overline{\mathsf{Int}} \otimes {}^{\mathsf{U}}\overline{\mathsf{Int}})$$
$$\cdot; \cdot, f{:}^{\mathsf{L}}({}^{\mathsf{L}}\mathbf{1}_{\otimes} \multimap {}^{\mathsf{U}}\overline{\mathsf{Int}}); \{\} \vdash_{\text{exp}} f\ {}^{\mathsf{L}}\langle\rangle : {}^{\mathsf{U}}\overline{\mathsf{Int}}$$
$$\cdot \vdash \{\} \rightsquigarrow \{\} \boxdot \{\}$$
$$\cdot \vdash \cdot, f{:}^{\mathsf{L}}({}^{\mathsf{L}}\mathbf{1}_{\otimes} \multimap {}^{\mathsf{U}}\overline{\mathsf{Int}}) \rightsquigarrow \cdot \boxdot \cdot, f{:}^{\mathsf{L}}({}^{\mathsf{L}}\mathbf{1}_{\otimes} \multimap {}^{\mathsf{U}}\overline{\mathsf{Int}})$$
$$\cdot \vdash \cdot, x{:}^{\mathsf{U}}\overline{\mathsf{Int}} \rightsquigarrow \cdot \boxdot \cdot, x{:}^{\mathsf{U}}\overline{\mathsf{Int}}$$
$$x \notin dom(\cdot)$$

It should be clear that the corresponding consequent does not hold:

$$\cdot; \cdot, f{:}^{\mathsf{L}}({}^{\mathsf{L}}\mathbf{1}_{\otimes} \multimap {}^{\mathsf{U}}\overline{\mathsf{Int}}); \{\} \vdash_{\text{exp}} {}^{\mathsf{L}}\langle f\ {}^{\mathsf{L}}\langle\rangle, f\ {}^{\mathsf{L}}\langle\rangle\rangle : {}^{\mathsf{L}}({}^{\mathsf{U}}\overline{\mathsf{Int}} \otimes {}^{\mathsf{U}}\overline{\mathsf{Int}})$$

since the linear variable f is used more than once in expression ${}^{\mathsf{L}}\langle f\ {}^{\mathsf{L}}\langle\rangle, f\ {}^{\mathsf{L}}\langle\rangle\rangle$.

Instead, we require a more specific substitution lemma. We note that a location is the only expression form that is substituted for variables in the allocation semantics. Hence, we may state the substitution lemma for this specific case:

Lemma C.15

If $\Delta; \Gamma_{e+x}; \Sigma_e \vdash_{\text{exp}} e : \tau$, $\Delta \vdash \Sigma_{e+\mathfrak{l}} \rightsquigarrow \Sigma_e \boxdot \{\mathfrak{l} \mapsto \tau'\}$,
$\Delta \vdash \Gamma_{e+x} \rightsquigarrow \Gamma_e \boxdot \cdot, x{:}\tau'$, *and* $x \notin dom(\Gamma_e)$, *then* $\Delta; \Gamma_e; \Sigma_{e+\mathfrak{l}} \vdash_{\text{exp}} e[\mathfrak{l}/x] : \tau$.

The proof is by induction on the derivation $\Delta; \Gamma; \Sigma_e \vdash_{\text{exp}} e : \tau$.

Finally, the most interesting portion of the proof of the Preservation Theorem is a group of lemmas stating that typing derivations for stores and heaps with

"garbage" in the store and heap types may be transformed into derivations that omit the "garbage" from the store and heap types:

Lemma C.16

If $\mathcal{H} \vdash \sigma : \Sigma$, $\cdot \vdash \Sigma \rightsquigarrow \Sigma_g \boxdot \Sigma'$, *and* $\cdot \vdash \Sigma_g \preceq \mathsf{A}$, *then there exists* $\mathcal{H}_g$ *and* $\mathcal{H}'$ *such that* $\mathcal{H}' \vdash \sigma : \Sigma'$, $\vdash \mathcal{H} \rightsquigarrow \mathcal{H}_g \boxdot \mathcal{H}'$, *and* $\vdash \mathcal{H}_g \sqsubseteq \mathsf{A}$.

Lemma C.17

If $\Sigma \vdash H : \mathcal{H}$, $\vdash \mathcal{H} \rightsquigarrow \mathcal{H}_g \boxdot \mathcal{H}'$, *and* $\vdash \mathcal{H}_g \sqsubseteq \mathsf{A}$, *then there exists* Σ_g *and* Σ' *such that* $\Sigma' \vdash H : \mathcal{H}'$, $\cdot \vdash \Sigma \rightsquigarrow \Sigma_g \boxdot \Sigma'$, *and* $\cdot \vdash \Sigma_g \preceq \mathsf{A}$.

Lemma C.18

If $\vdash (\sigma; H) : \Sigma$, $\cdot \vdash \Sigma \rightsquigarrow \Sigma_g \boxdot \Sigma'$, *and* $\cdot \vdash \Sigma_g \preceq \mathsf{A}$, *then* $\vdash (\sigma; H) : \Sigma'$.

The proofs of the first and second lemmas are by induction on the derivations $\mathcal{H} \vdash \sigma : \Sigma$ and $\Sigma \vdash H : \mathcal{H}$, respectively.

The proof of the third lemma is much more subtle. Consider a naïve proof attempt.

Proof Attempt (Lemma C.18)

We assume that $\vdash (\sigma; H) : \Sigma_1$, $\cdot \vdash \Sigma_1 \rightsquigarrow \Sigma_{g1} \boxdot \Sigma$, and $\cdot \vdash \Sigma_{g1} \preceq \mathsf{A}$. We note that there is only one rule for the judgment $\vdash (\sigma; H) : \Sigma$; by inversion of this rule, there exists $\mathcal{H}_{\sigma 1}$, Σ'_1, and Σ_{H1} such that $\mathcal{H}_{\sigma 1} \vdash \sigma : \Sigma'_1$, $\Sigma_{H1} \vdash H : \mathcal{H}_{\sigma 1}$, and $\cdot \vdash \Sigma'_1 \rightsquigarrow \Sigma_{H1} \boxdot \Sigma_1$. By the associativity and commutativity of $\boxdot$, it follows that there exists Σ'_2 such that $\cdot \vdash \Sigma'_2 \rightsquigarrow \Sigma_{H1} \boxdot \Sigma$ and $\cdot \vdash \Sigma'_1 \rightsquigarrow \Sigma_{g1} \boxdot \Sigma'_2$.

Applying Lemma C.16 to $\mathcal{H}_{\sigma 1} \vdash \sigma : \Sigma'_1$, $\cdot \vdash \Sigma'_1 \rightsquigarrow \Sigma_{g1} \boxdot \Sigma'_2$, and $\cdot \vdash \Sigma_{g1} \preceq \mathsf{A}$, we conclude that there exists $\mathcal{H}_{g1}$ and $\mathcal{H}_{\sigma 2}$ such that $\mathcal{H}_{\sigma 2} \vdash \sigma : \Sigma'_2$, $\vdash \mathcal{H}_{\sigma 1} \rightsquigarrow \mathcal{H}_{g1} \boxdot \mathcal{H}_{\sigma 2}$ and $\vdash \mathcal{H}_{g1} \sqsubseteq \mathsf{A}$.

Applying Lemma C.17 to $\Sigma_{H1} \vdash H : \mathcal{H}_{\sigma 1}$, $\vdash \mathcal{H}_{\sigma 1} \rightsquigarrow \mathcal{H}_{g1} \boxdot \mathcal{H}_{\sigma 2}$, and $\vdash \mathcal{H}_{g1} \sqsubseteq \mathsf{A}$, we conclude that there exists Σ_{g2} and Σ_{H2} such that $\Sigma_{H2} \vdash H : \mathcal{H}_{\sigma 2}$, $\vdash \Sigma_{H1} \rightsquigarrow \Sigma_{g2} \boxdot \Sigma_{H2}$ and $\cdot \vdash \Sigma_{g2} \preceq \mathsf{A}$.

By the associativity and commutativity of $\boxdot$, it follows that there exists Σ_2 such that $\cdot \vdash \Sigma_2 \rightsquigarrow \Sigma_{g2} \boxdot \Sigma$ and $\cdot \vdash \Sigma_2' \rightsquigarrow \Sigma_{H2} \boxdot \Sigma_2$.

Note that we may construct the derivation:

$$\frac{\mathcal{H}_{\sigma 2} \vdash \sigma : \Sigma_2' \qquad \Sigma_{H2} \vdash H : \mathcal{H}_{\sigma 2} \qquad \cdot \vdash \Sigma_2' \rightsquigarrow \Sigma_{H2} \boxdot \Sigma_2}{\vdash (\sigma; H) : \Sigma_2}$$

We also have that $\cdot \vdash \Sigma_2 \rightsquigarrow \Sigma_{g2} \boxdot \Sigma$ and $\cdot \vdash \Sigma_{g2} \preceq \mathsf{A}$.

At this point in the proof, it appears as though we are in a position to inductively apply the lemma (as in a proof by induction). However, the constructed derivation $\vdash (\sigma; H) : \Sigma_2$ is not a sub-derivation of $\vdash (\sigma; H) : \Sigma_1$; nor it is not necessarily the case that Σ_{g2} is smaller than Σ_{g1}. Furthermore, we must establish that pushing "garbage" through the store type and heap typing eventually stops producing more "garbage".

The solution is to first prove a lemma relating the sizes of a store type and its splitting:

Lemma C.19

If $\cdot \vdash \Sigma \rightsquigarrow \Sigma_1 \boxdot \Sigma_2$, *then either* $\Sigma_2 = \Sigma$ *or* $|\Sigma_2| < |\Sigma|$.

On more auxiliary lemma exposes a sufficiently strong induction hypothesis:

Lemma C.20

If $\Sigma_{H1} \vdash H : \mathcal{H}_{\sigma 1}$, $\cdot \vdash \Sigma_2' \rightsquigarrow \Sigma_{H1} \boxdot \Sigma$, $\cdot \vdash \Sigma_1' \rightsquigarrow \Sigma_{g1} \boxdot \Sigma_2'$, $\mathcal{H}_{\sigma 1} \vdash \sigma : \Sigma_1'$, $\cdot \vdash \Sigma_{g1} \preceq \mathsf{A}$, *then there exists* Σ_{H2}, $\mathcal{H}_{\sigma 2}$, *and* Σ_3' *such that* $\Sigma_{H2} \vdash H : \mathcal{H}_{\sigma 2}$, $\cdot \vdash \Sigma_3' \rightsquigarrow \Sigma_{H2} \boxdot \Sigma$, $\mathcal{H}_{\sigma 2} \vdash \sigma : \Sigma_3'$.

Proof (Lemma C.20)

The proof is by strong induction on $|\Sigma_1'|$. Applying Lemma C.19 to $\cdot \vdash \Sigma_1' \rightsquigarrow \Sigma_{g1} \boxdot \Sigma_2'$, we conclude that either $\Sigma_2' = \Sigma_1'$ or $|\Sigma_2'| < |\Sigma_1'|$.

Suppose $\Sigma_2' = \Sigma_1'$. Take $\Sigma_{H2} = \Sigma_{H1}$, $\mathcal{H}_{\sigma 2} = \mathcal{H}_{\sigma 1}$, and $\Sigma_3' = \Sigma_2'$. Note that $\Sigma_{H1} \vdash H : \mathcal{H}_{\sigma 1}$, $\cdot \vdash \Sigma_2' \rightsquigarrow \Sigma_{H1} \boxdot \Sigma$, $\mathcal{H}_{\sigma 1} \vdash \sigma : \Sigma_2'$.

Suppose $|\Sigma_2'| < |\Sigma_1'|$. Applying Lemma C.16 to $\mathcal{H}_{\sigma 1} \vdash \sigma : \Sigma_1'$, $\cdot \vdash \Sigma_1' \rightsquigarrow \Sigma_{g1} \boxdot \Sigma_2'$, and $\cdot \vdash \Sigma_{g1} \preceq \mathsf{A}$, we conclude that there exists $\mathcal{H}_{g1}$ and $\mathcal{H}_{\sigma 2}$ such that $\mathcal{H}_{\sigma 2} \vdash \sigma : \Sigma_2'$, $\vdash \mathcal{H}_{\sigma 1} \rightsquigarrow \mathcal{H}_{g1} \boxdot \mathcal{H}_{\sigma 2}$ and $\vdash \mathcal{H}_{g1} \sqsubseteq \mathsf{A}$.

Applying Lemma C.17 to $\Sigma_{H1} \vdash H : \mathcal{H}_{\sigma 1}$, $\vdash \mathcal{H}_{\sigma 1} \rightsquigarrow \mathcal{H}_{g1} \boxdot \mathcal{H}_{\sigma 2}$, and $\vdash \mathcal{H}_{g1} \sqsubseteq \mathsf{A}$, we conclude that there exists Σ_{g2} and Σ_{H2} such that $\Sigma_{H2} \vdash H : \mathcal{H}_{\sigma 2}$, $\vdash \Sigma_{H1} \rightsquigarrow \Sigma_{g2} \boxdot \Sigma_{H2}$ and $\cdot \vdash \Sigma_{g2} \preceq \mathsf{A}$.

By the associativity and commutativity of $\boxdot$, it follows that there exists $\Sigma_\star$ such that $\cdot \vdash \Sigma_\star \rightsquigarrow \Sigma_{H2} \boxdot \Sigma$ and $\cdot \vdash \Sigma_2' \rightsquigarrow \Sigma_{g1} \boxdot \Sigma_\star$.

Applying the induction hypothesis to $\Sigma_{H2} \vdash H : \mathcal{H}_{\sigma 2}$, $\cdot \vdash \Sigma_\star \rightsquigarrow \Sigma_{H2} \boxdot \Sigma$, $\cdot \vdash \Sigma_2' \rightsquigarrow \Sigma_{g2} \boxdot \Sigma_\star$, $\mathcal{H}_{\sigma 2} \vdash \sigma : \Sigma_2'$, and $\cdot \vdash \Sigma_{g2} \preceq \mathsf{A}$, and noting that $|\Sigma_2'| < |\Sigma_1'|$, we conclude that there exists Σ_{H2}, $\mathcal{H}_{\sigma 2}$, and Σ_3' such that $\Sigma_{H2} \vdash H : \mathcal{H}_{\sigma 2}$, $\cdot \vdash \Sigma_3' \rightsquigarrow \Sigma_{H2} \boxdot \Sigma$, $\mathcal{H}_{\sigma 2} \vdash \sigma : \Sigma_3'$.

The proof of Lemma C.18 follows as a simple corollary of Lemma C.20.

We require these "garbage" lemmas in order to discard the store type left over from inverting the expression typing derivation for a location:

Lemma C.21

If $\cdot; \cdot; \Sigma \vdash \mathfrak{l} : \tau$, *then there exists* Σ_g *such that* $\cdot \vdash \Sigma \rightsquigarrow \Sigma_g \boxplus \{\mathfrak{l} \mapsto \tau\}$ *and* $\cdot \vdash \Sigma_g \preceq \mathsf{A}$.

The proof is by induction on $\cdot;\cdot;\Sigma \vdash \mathfrak{l} : \tau$, accumulating the "garbage" store type from instances of the WEAK(STORE) rule.

The Soundness theorem follows directly from the Progress and Preservation Theorems.

Theorem C.22 (rgnURAL **Soundness)**

If $\vdash (\sigma_1; H_1; e_1) : \tau$ *and* $(\sigma_1; H_1; e_1) \longmapsto^* (\sigma_2; H_2; e_2)$*, then either there exists* $\mathfrak{l}$ *such that* $e_2 \equiv \mathfrak{l}$ *or there exists* σ_3 *and* H_3 *and* e_3 *such that* $(\sigma_2; H_2; e_2) \longmapsto (\sigma_3; H_3; e_3)$.

Mechanized proof

We have carried out and mechanically-verified the soundness proof of rgnURAL in the Twelf system [66] using its metatheorem checker [71, 36, 37]. We consider the major differences between the presentation of the rgnURALlanguage in Appendix C.1 and its encoding in the mechanized proof.

Stores, heaps, and regions It turns out that modeling a program state with a store, considered as a partial map from (α-varying) locations to values (which may have free occurrences of those locations), is somewhat awkward to encode. The encoding becomes even more awkward in the presence of a heap of regions, as the store may have free occurrences of region names from the heap and pointer from the regions and the heap may have free occurrences of locations from the store.

Hence, to simplify the formulation, a store is represented as a list of location/flag/value triples, where locations are isomorphic to the natural numbers:

```
st      : type.   %name st S.

nil_st  : st.

cons_st : loc -> flag -> val -> st -> st.
```

which is essentially equivalent to the following:

$$\sigma \quad ::= \quad \bullet \mid \mathfrak{l} \mapsto (\mathfrak{f}, v), \sigma$$

The representations of regions and of heaps are analogous: a list of pointer/constant qualifier/location triples and a list of region name/region mark/region triples, respectively.

Our syntactic form for stores in Appendix C.1.1 implicitly required that a store does not contain duplicate bindings for the same location. A simple means of maintaining this invariant in the mechanized proof is to ensure that the locations in the store are kept in a sorted order, and to ensure that new locations are greater than any current location. We capture this invariant with a store well-formedness judgment:

```
st_wf           : st -> type.

st_wf_nil       : st_wf nil_st.

st_wf_cons_nil  : st_wf (cons_st L F V nil_st).

st_wf_cons_cons : st_wf (cons_st L' F' V' S') ->
                  loc_lt L' L ->
                     st_wf (cons_st L F V (cons_st L' F' V' S')).
```

which is equivalent the following rules:

$$\vdash \bullet \ \mathsf{wf} \qquad \vdash \mathfrak{l} \mapsto (\mathfrak{f}, v), \bullet \ \mathsf{wf} \qquad \frac{\vdash \mathfrak{l}' \mapsto (\mathfrak{f}', v'), \sigma' \ \mathsf{wf} \qquad \mathfrak{l}' <_{\mathsf{loc}} \mathfrak{l}}{\vdash \mathfrak{l} \mapsto (\mathfrak{f}', v'), \mathfrak{l}' \mapsto (\mathfrak{f}', v'), \sigma' \ \mathsf{wf}}$$

Note that that we could (almost) equally well take this as the definition of a store. However, it becomes somewhat cumbersome to do so, and there are relatively few situations where we really need the well-formedness requirement. Also, note that induction on stores (`st`) has two cases, while induction on store well-formedness judgments (`st_wf`) have three cases. Hence, where it is possible to reason about an arbitrary store, it is expedient to do so.

Store allocation $((\sigma; v) \xrightarrow{\mathsf{alloc}} (\sigma'; \mathfrak{l}'))$ is represented by a judgment:

```
st_alloc       : st -> val -> st -> loc -> type.
st_alloc_nil   : st_alloc nil_st V''
                          (cons_st zero_loc V'' unused_flag
                                   nil_st)
                          zero_loc.
st_alloc_cons  : st_alloc (cons_st L' V' F' S') V''
                          (cons_st (next_loc L') V'' unused_flag
                                   (cons_st L' V' F' S'))
                          (next_loc L').
```

which is equivalent the following rules:

$$(\bullet; v) \xrightarrow{\mathsf{alloc}} (0_{\mathsf{loc}} \mapsto (\mathsf{unused}, v), \bullet; 0_{\mathsf{loc}})$$

$$(\mathfrak{l}' \mapsto (\mathfrak{f}', v'), \sigma'; v) \xrightarrow{\mathsf{alloc}} (\mathfrak{l}' +_{\mathsf{loc}} 1_{\mathsf{loc}} \mapsto (\mathsf{unused}, v), (\mathfrak{f}', v'), \sigma'; \mathfrak{l}' +_{\mathsf{loc}} 1_{\mathsf{loc}})$$

Note that location 0_{loc} (`zero_loc`) is allocated when the input store is empty; otherwise the location $\mathfrak{l}' +_{\mathsf{loc}} 1_{\mathsf{loc}}$ (`next_loc L'`) is allocated when the head of the input store is location $\mathfrak{l}'$ (`L'`). A simple lemma proves that store allocation preserves store well-formedness.

Similar judgments represent the other store, heap, and region manipulations, each of which maintains well-formedness.

Store, heap, and region types The representation of a store type is analogous to the representation of a store; a store type is represented as a list of location/type pairs:

```
sttp      : type.   %name sttp ST.
nil_sttp  : sttp.
cons_sttp : loc -> tp -> sttp -> sttp.
```

which is essentially equivalent to the following:

$$\Sigma \ ::= \ \bullet \mid \iota \mapsto \tau, \Sigma$$

As with stores, we have a well-formedness judgment for store types, which requires that locations in the store type are kept in a sorted order.

This representation of a store type makes the representation of the judgment that splits store types somewhat more cumbersome than the corresponding the judgment given in Appendix C.1.2 (Figure C.18). Store type splitting ($\Delta \vdash \Sigma \rightsquigarrow \Sigma_1 \boxdot \Sigma_2$) is represented by a judgment:

```
sttp_split        : sttp -> sttp -> sttp -> type.
sttp_split_nil__* : sttp_split ST nil_sttp ST.
sttp_split_*__nil : sttp_split ST ST nil_sttp.
```

```
sttp_split_cons__cons_shared
                  : sttp_split ST ST1 ST2 ->
                    |-tpsq T rel_qual ->
                       sttp_split (cons_sttp L T ST)
                                  (cons_sttp L T ST1)
                                  (cons_sttp L T ST2).
sttp_split_cons__cons_l
                  : sttp_split ST ST1 (cons_sttp L2 T2 ST2) ->
                    loc_lt L2 L1 ->
                       sttp_split (cons_sttp L1 T1 ST)
                                  (cons_sttp L1 T1 ST1)
                                  (cons_sttp L2 T2 ST2).
sttp_split_cons__cons_r
                  : sttp_split ST (cons_sttp L1 T1 ST1) ST2 ->
                    loc_lt L1 L2 ->
                       sttp_split (cons_sttp L2 T2 ST)
                                  (cons_sttp L1 T1 ST1)
                                  (cons_sttp L2 T2 ST2).
```

which is equivalent the following rules:

$$\Delta \vdash \Sigma \rightsquigarrow \bullet \boxdot \Sigma \qquad \Delta \vdash \Sigma \rightsquigarrow \Sigma \boxdot \bullet \qquad \frac{\Delta \vdash \Sigma \rightsquigarrow \Sigma_1 \boxdot \Sigma_2 \qquad \Delta \vdash \tau \preceq \mathsf{R}}{\Delta \vdash \mathfrak{l} \mapsto \tau, \Sigma \rightsquigarrow \mathfrak{l} \mapsto \tau, \Sigma_1 \boxdot \mathfrak{l} \mapsto \tau, \Sigma_2}$$

$$\frac{\Delta \vdash \Sigma \rightsquigarrow \Sigma_1 \boxdot \ell_2 \mapsto \tau_2, \Sigma_2 \qquad \ell_2 <_{\mathsf{loc}} \ell_1}{\Delta \vdash \ell_1 \mapsto \tau_1, \Sigma \rightsquigarrow \ell_1 \mapsto \tau_1, \Sigma_1 \boxdot \ell_2 \mapsto \tau_2, \Sigma_2}$$

$$\frac{\Delta \vdash \Sigma \rightsquigarrow \ell_1 \mapsto \tau_1, \Sigma_1 \boxdot \Sigma_2 \qquad \ell_1 <_{\mathsf{loc}} \ell_2}{\Delta \vdash \ell_2 \mapsto \tau_2, \Sigma \rightsquigarrow \ell_1 \mapsto \tau_1, \Sigma_1 \boxdot \ell_2 \mapsto \tau_2, \Sigma_2}$$

Using the store type splitting judgment in the mechanized proof can be tedious. Induction on the store type splitting judgment requires five cases. Furthermore, we are required to state, prove, and make extensive use of "obvious" lemmas stating that the store splitting is a commutative and associative operation with the empty store as an identity.

The representation and treatment of heap types and region types is analogous. There are well-formedness judgments for heap types and region types, and the judgments for splitting of heap types and region types make use of the list representation.

We note that the judgments from Appendix C.1.2 that assign region types to regions ($\Sigma \vdash R : \mathcal{R}$), heap types to heaps ($\Sigma \vdash H : \mathcal{H}$), and store types to stores (Figures C.32 and C.33) are naturally handled by the list representation, since these rules are defined by a case for an empty object and a number of cases for an object factored into a sub-object and a distinguished binding.

Higher-order abstract syntax As is customary when representing an object language in the Twelf system, we representing binding of rgnURAL variables using higher-order abstract syntax. We use this encoding for value variables, qualifier variables, pre-type variables, type variables, and region variables.

This representation implicitly uses the LF context to represent the rgnURAL value context Γ and qualifier, pre-type, type, and region context Δ. Since the

LF context treats all bindings as unrestricted (i.e., the LF type system is not a substructural type system), we must use an auxiliary judgment to codify the substructural treatment of the value context. This auxiliary judgment asserts that meta-function (i.e., a piece of higher-order abstract syntax) respects the type of the abstracted variable (i.e., is linear, affine, relevant, or unrestricted in the variable). The encoding of the typing rules for value variable binding forms requires that the higher-order abstract syntax representing the binding respects the type of the bound variable. This auxiliary judgment is used in the formal proof of the substitution lemma (Lemma C.15).

We note that no such auxiliary judgment is needed for the qualifier, pre-type, type, and region context, since these variables may be treated as unrestricted.

BIBLIOGRAPHY

[1] Alex Aiken, Manuel Fähndrich, and Raph Levien. Better static memory management: Improving region-based analysis of higher-order languages. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'95)*, pages 174–185, June 1995.

[2] Alex Aiken, Jeffrey Foster, John Kodumal, and Tachio Terauchi. Checking and inferring local non-aliasing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'03)*, pages 129–140, June 2003.

[3] Paulo Almeida. Balloon types: Controlling sharing of state in data types. In *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP'97)*, pages 32–59, June 1997.

[4] Zena Ariola and Amyr Sabry. Correctness of monadic state: An imperative call-by-need calculus. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'98)*, pages 62–74, January 1998.

[5] Anindya Banerjee, Nevin Heintze, and Jon Riecke. Region analysis and the polymorphic lambda calculus. In *Proceedings of the 14th IEEE Symposium on Logic in Computer Science (LICS'99)*, pages 88–97, July 1999.

[6] Emery Berger, Benjamin Zorn, and Kathryn McKinley. Reconsidering custom memory allocation. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'02)*, pages 1–12, November 2002.

[7] Hans Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software – Practice and Experience*, 18(9):807–820, September 1988.

[8] Chandrasekhar Boyapati, Alexandru Sălcianu, William Beebee, and Martin Rinard. Ownership types for safe region-based memory management in real-time Java. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'03)*, pages 324–337, June 2003.

[9] John Boyland, James Noble, and William Retert. Capabilities for aliasing: A generalization of uniqueness and read-only. In *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP'01)*, pages 2–27, June 2001.

[10] Cristiano Calcagno. Stratified operational semantics for safety and correctness of the region calculus. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'01)*, pages 155–165, January 2001.

[11] Cristiano Calcagno, Simon Helsen, and Peter Thiemann. Syntactic type soundness results for the region calculus. *Information and Computation*, 173(2):199–332, March 2002.

[12] Chiyan Chen and Hongwei Xi. Combining programming with theorem proving. In *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming (ICFP'05)*, pages 66–77, September 2005.

[13] Dave Clarke and Tobias Wrigstad. External uniqueness is unique enough. In *Proceedings of the 17th European Conference on Object-Oriented Programming (ECOOP'03)*, pages 176–200, July 2003.

[14] David Clarke. *Object Ownership and Containment*. PhD thesis, School of Computer Science and Engineering, University of New South Wales, 2001.

[15] David Clarke, John Potter, and James Noble. Ownership types for flexible alias protection. In *Proceedings of the 13th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'98)*, pages 48–64, October 1998.

[16] Karl Crary, David Walker, and Greg Morrisett. Typed memory management in a calculus of capabilities. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'99)*, pages 262–275, January 1999.

[17] *Cyclone, version 1.0*, May 2006. `http://cyclone.thelanguage.org/`.

[18] Robert DeLine and Manuel Fähndrich. Enforcing high-level protocols in low-level software. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'01)*, pages 59–69, June 2001.

[19] Manuel Fähndrich, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen Hunt, James Larus, and Steven Levi. Language support for fast and reliable message-based communication in Singularity OS. In *Proceedings of the 1st ACM SIGOPS Eurosys Conference (EuroSys'06)*, April 2006.

[20] Manuel Fähndrich and Robert DeLine. Adoption and focus: Practical linear types for imperative programming. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'02)*, pages 13–24, June 2002.

[21] Matthew Fluet. Monadic regions: Formal type soundness and correctness. Technical Report TR2004-1936, Department of Computer Science, Cornell University, April 2004.

[22] Matthew Fluet and Daniel Wang. Implementation and performance evaluation of a safe runtime system in Cyclone. In *Informal Proceedings of the Semantics, Program Analysis, and Computing Environments for Memory Management Workshop (SPACE'04)*, January 2004.

[23] Steven Ganz. *Monadic Encapsulation of State*. PhD thesis, Indiana University, Bloomington, Indiana, forthcoming.

[24] David Gay and Alex Aiken. Memory management with explicit regions. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'98)*, pages 313–323, June 1998.

[25] David Gifford, Pierre Jouvelot, and Mark Sheldon. The FX-87 reference manual. Technical Report TR-407, Massachusetts Institute of Technology, September 1987.

[26] David Gifford, Pierre Jouvelot, Mark Sheldon, and James O'Toole. Report on the FX programming language. Technical Report TR-531, Massachusetts Institute of Technology, February 1992.

[27] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50(1):1–102, 1987.

[28] Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and Types*. Cambridge University Press, 1989.

[29] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in Cyclone. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'02)*, pages 282–293. June 2002.

[30] Dan Grossman, Greg Morrisett, Yanling Wang, Trevor Jim, Michael Hicks, and James Cheney. Formal type soundness for Cyclone's region system. Technical Report TR2001-1856, Department of Computer Science, Cornell University, November 2001.

[31] Christian Grothoff, Jens Palsberg, and Jan Vitek. Encapsulating objects with confined types. In *Proceedings of the 16th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'01)*, pages 241–255, October 2001.

[32] Jörgen Gustavsson and Josef Svenningsson. A usage analysis with bounded usage polymorphism and subtyping. In *Selected Papers of the 12th International Workshop on Implementation of Functional Languages (IFL'00)*, pages 140–157, September 2000.

[33] Kevin Hamlen, Greg Morrisett, and Fred Schneider. Certified in-lined reference monitoring on .NET. Technical Report TR2005-2003, Department of Computer Science, Cornell University, November 2005.

[34] Kevin Hamlen, Greg Morrisett, and Fred Schneider. Certfied in-lined reference monitoring on .NET. In *Proceedings of the ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS'06)*, pages 7–16, June 2006.

[35] R. Todd Hammel and David Gifford. FX-87 performance measurements: Dataflow implementation. Technical Report TR-421, Massachusetts Institute of Technology, September 1988.

[36] Robert Harper and Karl Crary. How to believe a Twelf proof. (Draft.), May 2005.

[37] Robert Harper and Daniel Licata. Mechanizing language definitions. (Submitted for publication.), April 2006.

[38] Chris Hawblitzel, Edward Wei, Heng Huang, Erik Krupski, and Lea Wittie. Low-level linear memory management. In *Informal Proceedings of the Semantics, Program Analysis, and Computing Environments for Memory Management Workshop (SPACE'04)*, January 2004.

[39] Simon Helsen and Peter Thiemann. Syntactic type soundness for the region calculus. In *Proceedings of the 4th International Workshop on Higher Order Operational Techniques in Semantics (HOOTS'00)*, pages 1–19, September 2000.

[40] Fritz Henglein, Henning Makholm, and Henning Niss. A direct approach to control-flow sensitive region-based memory management. In *Proceedings of the 3rd ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'01)*, pages 175–186, September 2001.

[41] Fritz Henglein, Henning Makholm, and Henning Niss. Effect types and region-based memory management. In Benjamin Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 3, pages 87–135. MIT Press, 2005.

[42] Matthew Hertz and Emery Berger. Quantifying the performance of garbage collection vs. explicit memory management. In *Proceedings of the 20th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'05)*, October 2005.

[43] Michael Hicks, Greg Morrisett, Dan Grossman, and Trevor Jim. Safe and flexible memory management in Cyclone. Technical Report CS-TR-4514, University of Maryland Department of Computer Science, July 2003.

[44] Michael Hicks, Greg Morrisett, Dan Grossman, and Trevor Jim. Experience with safe manual memory-management in Cyclone. In *Proceedings of the 4th International Symposium on Memory Management (ISMM'04)*, pages 73–84, October 2004.

[45] John Hogg. Islands: Aliasing protection in object-oriented languages. In *Proceedings of the 6th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'91)*, pages 271–285, November 1991.

[46] Galen Hunt, James Larus, Martí Abadi, Mark Aiken, Paul Barham, Manuel Fähndrich, Chris Hawblitzel, Orion Hodson, Steven Levi, Nick Murphy, Bjarne Steensgaard, David Tarditi, Ted Wobber, and Brian Zill. An overview of the Singularity project. Technical Report MSR-TR-2005-135, Microsoft Corporation, October 2005.

[47] Atsushi Igarashi and Naoki Kobayashi. Resource usage analysis. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'02)*, pages 331–342, January 2002.

[48] Koji Kagawa. Compositional references for stateful functional programming. In *Proceedings of the 2nd ACM SIGPLAN International Conference on Functional Programming (ICFP'97)*, pages 217–226. June 1997.

[49] Koji Kagawa. Monadic encapsulation with stack of regions. In *Proceedings of the 5th International Symposium on Functional and Logic Programming (FLOPS'01)*, pages 264–279, March 2001.

[50] Richard Kieburtz. Taming effects with monadic typing. In *Proceedings of the 3rd ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*, pages 51–62, September 1998.

[51] Oleg Kiselyov. Simple IO regions. `http://www.haskell.org/pipermail/haskell/2006-January/017410.html`, January 2006.

[52] Oleg Kiselyov, Ralf Lämmel, and Keean Schupke. Strongly typed heterogeneous collections. In *Proceedings of the ACM SIGPLAN Workshop on Haskell (Haskell'04)*, pages 96–107, September 2004.

[53] Naoki Kobayashi. Quasi-linear types. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'99)*, pages 29–42, January 1999.

[54] Viktor Kuncak, Patrick Lam, and Martin Rinard. Role analysis. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 17–32, January 2002.

[55] John Launchbury and Simon Peyton Jones. Lazy functional state threads. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'94)*, pages 24–35, June 1994.

[56] John Launchbury and Simon Peyton Jones. State in Haskell. *Lisp and Symbolic Computation*, 8(4):293–341, December 1995.

[57] John Launchbury and Amr Sabry. Monadic state: Axiomatization and type safety. In *Proceedings of the 2nd ACM SIGPLAN International Conference on Functional Programming (ICFP'97)*, pages 227–237, June 1997.

[58] John Lucassen and David Gifford. Polymorphic effect systems. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'88)*, pages 47–57, January 1988.

[59] Yitzhak Mandelbaum, David Walker, and Robert Harper. An effective theory of type refinements. In *Proceedings of the 8th ACM SIGPLAN International Conference on Functional Programming (ICFP'03)*, pages 213–225, September 2003.

[60] Torben Mogensen. Types for 0, 1 or many uses. In *Selected Papers of the 10th International Workshop on Implementation of Functional Languages (IFL'98)*, pages 112–122, September 1998.

[61] Eugino Moggi. Computational lambda calculus and monads. In *Proceedings of the 4th IEEE Symposium on Logic in Computer Science (LICS'89)*, pages 14–23, June 1989.

[62] Eugino Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, January 1991.

[63] Eugino Moggi and Amr Sabry. Monadic encapsulation of effects: a revised approach (extended version). *Journal of Functional Programming (JFP)*, 11(6):591–627, November 2001.

[64] Stefan Monnier, Bratin Saha, and Zhong Shao. Principled scavenging. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'01)*, pages 81–91, June 2001.

[65] Peter O'Hearn and John Reynolds. From Algol to polymorphic linear lambda-calculus. *Journal of the ACM (JACM)*, 47(1):167–223, January 2000.

[66] Frank Pfenning and Carsten Schürmann. System description: Twelf – a meta-logic framework for deductive systems. In *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, pages 202–206, July 1999.

[67] Benjamin Pierce. *Types and Programming Languages*. MIT Press, 2002.

[68] John Reynolds. Towards a theory of type structure. In *Proceedings of Colloque sur la Programmation (Programming Symposium)*, pages 408–425, April 1974.

[69] Jon Riecke and Ramesh Viswanathan. Isolating side effects in sequential languages. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'95)*, pages 1–12, January 1995.

[70] Amr Sabry and Philip Wadler. A reflection on call-by-value. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(6):916–941, November 1997.

[71] Carsten Schürmann and Frank Pfenning. A coverage checking algorithm for LF. In *Proceedings of the 16th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'03)*, pages 120–135, September 2003.

[72] Miley Semmelroth and Amr Sabry. Monadic encapsulation in ML. In *Proceedings of the 4th ACM SIGPLAN International Conference on Functional Programming (ICFP'99)*, pages 8–17, September 1999.

[73] Tim Sheard and Emir Pasalic. Meta-programming with built-in type equality (extended abstract). In *Proceedings of the Fourth International Workshop on Logical Frameworks and Meta-Languages (LFM'04)*, pages 106–124, July 2004.

[74] Sjaak Smetsers, Erik Barendsen, Marko van Eekelen, and Rinus Plasmeijer. Guaranteeing safe destructive updates through a type system with uniqueness information for graphs. In *Dagstuhl Seminar on Graph Transformations in Computer Science*, pages 358–379, January 1993.

[75] Fred Smith, David Walker, and Greg Morrisett. Alias types. In *Proceedings of the 9th European Symposium on Programming (ESOP'00)*, pages 366–381, March 2000.

[76] Geoffrey Smith and Dennis Volpano. A sound polymorphic type system for a dialect of C. *Science of Computer Programming*, 32(1-3):49–72, September 1998.

[77] Nikhil Swamy, Michael Hicks, Greg Morrisett, Dan Grossman, and Trevor Jim. Safe manual memory management in Cyclone. *Science of Computer Programming*, 62(2):95–204, October 2006. To appear.

[78] Mads Tofte, Lars Birkedal, Martin Elsman, Niels Hallenberg, Tommy Olesen, and Peter Sestoft. Programming with regions in the ML Kit (for version 4). Technical Report, IT University of Copenhagen, October 2002.

[79] Mads Tofte and Jean-Pierre Talpin. Implementation of the typed call-by-value λ-calculus using a stack of regions. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'94)*, pages 188–201, January 1994.

[80] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, February 1997.

[81] Stephen Tse and Steve Zdancewic. Translating dependency into parametricity. In *Proceedings of the 9th ACM SIGPLAN International Conference on Functional Programming (ICFP'04)*, pages 115–125, September 2004.

[82] David Turner, Philip Wadler, and Christian Mossin. Once upon a type. In *Proceedings of the 7th International Conference on Functional Programming Languages and Computer Architecture (FPCA'95)*, pages 1–11, June 1995.

[83] Jan Vitek and Boris Bokowski. Confined types in Java. *Software – Practice and Experience*, 31(6):507–532, May 2001.

[84] Dennis Volpano and Geoffrey Smith. Eliminating covert flows with minimum typings. In *Proceedings of the 10th IEEE Computer Security Foundations Workshop (CFSW'97)*, pages 156–168, June 1997.

[85] Philip Wadler. Linear types can change the world! In *Proceedings of the IFIP TC 2 Working Conference on Programming Concepts and Methods*, pages 561–581, April 1990.

[86] Philip Wadler. The essence of functional programming. In *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'92)*, pages 1–14, January 1992.

[87] Philip Wadler. The marriage of effects and monads. In *Proceedings of the 3rd ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*, pages 63–74, September 1995.

[88] Philip Wadler and Peter Thiemann. The marriage of effects and monads. *ACM Transactions on Computational Logic (TOCL)*, 4(1):1–32, January 2003.

[89] David Walker. Substructural type systems. In Benjamin Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 1, pages 3–43. MIT Press, 2005.

[90] David Walker, Karl Crary, and Greg Morrisett. Typed memory management in a calculus of capabilities. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 24(4):701–771, July 2000.

[91] David Walker and Greg Morrisett. Alias types for recursive data structures. In *Proceedings of the 3rd ACM SIGPLAN Workshop on Types in Compilation (TIC'00)*, pages 177–206, September 2000.

[92] David Walker and Kevin Watkins. On regions and linear types. In *Proceedings of the 6th ACM SIGPLAN International Conference on Functional Programming (ICFP'01)*, pages 181–192, September 2001.

[93] Daniel Wang and Andrew Appel. Type-preserving garbage collectors. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'-1)*, pages 166–178, January 2001.

[94] Keith Wansbrough and Simon Peyton Jones. Once upon a polymorphic type. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'99)*, pages 15–28, January 1999.

[95] Geoffrey Washburn and Stephanie Weirich. Boxes go bannanas: Encoding higher-order abstract syntax with parametric polymorphism. In *Proceedings of the 8th ACM SIGPLAN International Conference on Functional Programming (ICFP'03)*, pages 249–262, September 2003.

[96] Andrew Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, November 1994.