National Technical University of Athens
School of Electrical and Computer Engineering
Department of Computer Science

Static Safety Guarantees for Concurrent Programming Languages

DOCTORAL DISSERTATION

PRODROMOS GERAKIOS

Supervisor : Konstantinos Sagonas
Assoc. Professor N.T.U.A.

Athens, July 2012

Prodromos Gerakios

Abstract

Modern computers ranging from desktop PC to supercomputers have multi-core processors. Multi-core processors bring more computing power into a chip and software plays an essential role in utilizing this power.

Multithreaded programming is a widely used paradigm for exploiting concurrency in multi-core systems. An inherent side-effect of this paradigm is non-determinism of execution, which allows for invalid execution interleavings to occur. Such interleavings cannot be easily reproduced, thereby making the development of reliable concurrent software a hard task.

Data races and deadlocks are two well-known consequences of non-deterministic execution. Informally, a data race occurs when multiple threads concurrently access the same memory location without synchronization. A set of threads reaches a deadlocked state, when each thread in that set is waiting for a lock in cyclic order.

This dissertation shows that it is possible to increase the reliability of concurrent programs, by eliminating memory access violations, data races and deadlocks from multithreaded programs. In particular, we present the theory and implementation of a series of type systems and static analyses that provide absolute guarantees regarding multithreaded program execution.

In the theoretical parts, we present the formal semantics of the type systems and languages and give formal soundness proofs. In the implementation parts we discuss how we integrated these formal systems in real programming languages and the challenges that we met. We also provide extensive measurements for the performance of each analysis and compare alternative techniques.

Acknowledgements

First and foremost, I want to thank from the bottom of my heart my advisors, Nikos Papaspyrou and Kostis Sagonas, for all the guidance and support they have constantly given me. They have always dedicated their time and effort to provide me with advice and guidance. It has been a pleasure and an honor to work with and learn from such extraordinary individuals.

My way of thinking and the direction of my research has been greatly influenced by my studies at the University of Edinburgh. I would like to thank my advisor there, Stephen Gilmore, and the other members of the School of Informatics for directing me towards the formal study of programming languages.

The members of the Software Engineering Laboratory have always been good friends. In particular, I wish to thank Panagiotis Vekris, Giorgos Korfiatis, Michalis Papakyriakou, Maria Christakis, Xaris Nakos, Fanis Tziasios and Angelos Manousaridis for their outstanding support, the long hours of conversation and the good times that we have shared.

Last but not least, my wholehearted thanks go to my family, especially to my brother Kostas Gerakios, for their endless support in many ways.

My research was partially funded by the programme for supporting basic research (ΠΕΒΕ 2010) of the National Technical University of Athens under a project titled “Safety properties for concurrent programming languages.”

Prodromos Gerakios,

Athens, July 19, 2012

This thesis is also available as Technical Report CSD-SW-TR-1-12, National Technical University of Athens, School of Electrical and Computer Engineering, Department of Computer Science, Software Engineering Laboratory, July 2012.

URL: `http://www.softlab.ntua.gr/techrep/`
FTP: `ftp://ftp.softlab.ntua.gr/pub/techrep/`

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

1.1 Motivation

In the modern technological world we heavily depend on the continuous operation of complex and diverse software systems, which are of vital importance for our work or other purposes such as information, entertainment, mobility, etc. We rely on the proper functioning of computers, automobiles, aircrafts, telecommunications systems, databases and the Internet at all times. Computer system failures stemming from software hazards or violations from malicious attackers can have devastating consequences. One of the greatest challenges of our times is to develop techniques and tools for creating reliable software that will meet the growing demands of our society.

In the last years there has been a broad change in computer architecture that impacts every facet of our society as every electronic device from mobile phones to supercomputers needs to confront concurrency. More specifically, modern architectures have shifted from uniprocessor to multi-core configurations as device scaling predicted by Moore's Law is no longer a viable option for increasing the clock frequency of future uniprocessor systems.

A prominent execution model for exploiting the performance benefits of multi-core machines is explicit multithreading. Programming languages that retain the transparency and control of memory, such as C, are well-suited for explicit multithreading, except for the fact that programs written in them often compromise memory safety by allowing invalid memory accesses, buffer overruns, space leaks, etc., and become susceptible to data races and deadlocks by careless uses of locks. Thus, a challenge for programming language research is to design and implement multithreaded low-level languages providing static guarantees for the absence of memory violations and concurrency errors such as data races and deadlocks.

1.2 Thesis overview

The main objective of this thesis is to develop tools and techniques that improve reliability of concurrent software. Therefore, we present a number of static analyses that guarantee the absence of memory violations, data races and deadlocks from concurrent programs and impose minimal overheads at execution time.

While this work is primarily targeting low-level languages with unstructured locking,[1] and is applied to multithreaded Cyclone and C/pthreads programs, our formalisms are expressed as simple extensions of lambda calculus, thereby making the main ideas of our work *generic* and *language independent*.

First we propose two type systems that guarantee the absence of memory access violations and data races for a lambda calculus with region-based memory management and unstructured locking primitives. The first type system requires explicit type annotations, whereas the second type system can automatically infer type annotations and permits reader-writer locks. Second we propose two more type systems that guarantee absence of deadlocks for a lambda calculus with unstructured locking

[1] Hereon, we use the terms "unstructured locking", "non lexically-scoped locking" and "non block-structured locking" interchangeably.

primitives. The first type system also guarantees absence of memory access violations and data races, but requires explicit type annotations. The second type system solely guarantees absence of deadlocks and is able to automatically infer type annotations. We discuss the integration of our type systems to low-level languages such as Cyclone and C. Finally, we evaluate the effectiveness of each type system by testing our implementations over a number of benchmarks.

1.2.1 Hierarchical regions and locks for safe multithreading

Multithreaded programs that interact through shared memory allow concurrent memory accesses in a single address space. Threads, which are scheduled by a preemptive and weakly fair scheduler, generate random execution interleavings. Only a subset of these interleavings results in a consistent state. Inconsistent execution states occur in a multithreaded program when one thread accesses a memory location at the same time another thread attempts to write to it. In such interleavings we say that we have a *data race*. A basic correctness guarantee for multithreaded programs is absence of data races (i.e., *data race freedom*). In a shared memory system which does not support transactional memory [Harr03], threads must agree on the order of particular interleavings so that data races are avoided. More precisely, threads must somehow synchronize their actions. In this thesis, we only consider lock-based blocking synchronization. We also assume that memory accesses are sequentially consistent. That is, shared memory operations happen in the same order for all threads. Apart from inconsistent states resulting from data races, explicit memory management in a multithreaded environment may result in dangling pointer dereferences as one thread may deallocate a memory block while another thread attempts to access that block. Another important correctness guarantee is the absence of memory violations (i.e., *memory safety*).

We present the design, formal semantics, meta-theory and implementation of a strongly-typed language that employs advanced region-based management and hierarchical lock-based synchronization primitives and guarantees both memory safety and data race freedom.

A region [Toft94] can be thought of as a segment of memory. The key idea of region-based memory management is that each data object is allocated in some region. When a region is deallocated, all its objects are deallocated simultaneously. Region-based memory management has numerous benefits. The programmer has explicit control of the location and lifetime of memory objects as it is possible to allocate an object in any live region. In some languages it is also possible to perform early region deallocation. Moreover, it is more efficient to allocate objects in an existing memory segment or deallocate objects from it, rather than requesting a new memory area for each object individually. Our memory regions are organized in a hierarchical manner [Gay01], where each region is physically allocated within a single parent region and may contain multiple children regions. We propose a language that allows deallocation of complete subtrees in the presence of region sharing between threads and deallocation is allowed to occur at any program point.

Each region is associated with an implicit lock. Thus, locks also follow the hierarchical structure of regions and in this setting each region is protected by its own lock as well as the locks of all its ancestors. As opposed to the majority of type systems and analyses that guarantee race freedom for lexically-scoped locking constructs [Flan99a, Gros03, Boya02], our approach employs non-lexically scoped locking primitives, which are more suitable for languages at the C level of abstraction. Furthermore, the formal language allows regions and locks to be safely aliased and escape the lexical scope when passed to a new thread.

More importantly, our work is not just a theoretical design with some nice properties. We have integrated our analysis in Cyclone [Gros02], a strongly-typed dialect of C which preserves explicit control and representation of memory without sacrificing memory soundness. We have opted for Cyclone because it has a publicly available implementation but also because it is more than a safe variant of C. Cyclone offers modern programming language features such as first-class polymorphism, exceptions, tuples, namespaces, (extensible) algebraic data types, and region-based memory management. We will discuss how these features interact with our language and the additions that were needed to Cyclone's implementation.

We briefly outline the main design goals for our language, as well as some of the main design decisions that we made to serve these goals.

Low-level and concurrent. The language should be at the C level of abstraction and provide built-in constructs for concurrency similar to those currently used in e.g. C with the pthreads library (i.e., it should cater for non-lexically scoped mutual exclusion of concurrent threads).

Shared memory. The language should use shared memory as the means for intra-process communication. This happens both for efficiency reasons and because shared memory communication can easily be integrated in the existing region system of Cyclone, the language into which we implement our constructs.

Backwards compatibility. Sequential (Cyclone) code should work as expected with no modifications.

Static memory safety and thread safety guarantees. A static type and effect system should guarantee the absence of memory access violations and data races in well-typed programs, with minimal run-time overhead.

Safe and efficient region-based memory management. Traditional stack-based regions [Toft94] are limiting as they cannot be deallocated early. Furthermore, the stack-based discipline fails to model region lifetimes in concurrent languages, where the lifetime of a shared region depends on the lifetime of the longest-lived thread accessing that region. In contrast, we want regions that can be *deallocated early* and that can safely be *shared* between concurrent threads. We opt for a *hierarchical region* organization: each region is physically allocated within a single parent region and may contain multiple child regions. Early region deallocation in our multi-level hierarchy automatically deallocates the immediate subtree of a region without having to deallocate each region of the subtree recursively. The hierarchical region structure imposes the constraint that a child region is *live* only when its ancestors are live.

Race freedom. To prevent data races we use *lock-based* mutual exclusion. Instead of having a separate mechanism for locks, we opt for a uniform treatment of regions and locks: locks are placed in the same hierarchy as regions and enjoy similar properties. Each region is protected by its own private lock and inherits the access rights of its ancestors. The semantics of region locking is that the entire subtree of a region is *atomically locked* once the lock for that region has been acquired. Hierarchical locking can model complex synchronization strategies and lifts the burden of having to deal with explicit acquisition of multiple locks. Although deadlocks are possible, they can be eliminated by acquiring a single lock for a group of regions rather than acquiring multiple locks for each region separately or by more involved mechanisms. Additionally, we provide explicit locking primitives, which in turn allow a higher degree of concurrency than lexically-scoped locking, as some locks can be released early.

Region polymorphism and aliasing. Like Cyclone, we support functions which are generic with respect to regions (*region polymorphic*). This kind of polymorphism permits *region aliasing* as one actual region could be passed in the place of two distinct formal region parameters. In the presence of mutual exclusion and early region deallocation, aliasing is dangerous. We permit safe region aliasing with minimal restrictions. The mechanisms that we employ for this purpose also allow us to encode numerous useful idioms of concurrent programming, such as *region migration*, *lock ownership transfers*, *region sharing*, and *thread-local regions*.

In Chapter 2 we present a lambda calculus with region-based memory management, unstructured locking primitives and explicit type annotations. We also present its operational semantics, type system

```
59 struct dentry * efs_lookup(struct inode *dir, struct dentry *dentry) {
60   efs_ino_t inodenum;
61   struct inode * inode = NULL;
62   lock_kernel();
63   inodenum = efs_find_entry(dir, dentry->d_name.name, dentry->d_name.len);
64   if (inodenum) {
65     if (!(inode = iget(dir->i_sb, inodenum))) {
66       unlock_kernel();
67       return ERR_PTR(-EACCES);
68     }
69   }
70   unlock_kernel();
71   d_add(dentry, inode);
72   return NULL;
73 }
```

Listing 1.1: Code from Linux's EFS (`linux/fs/efs/namei.c`)

and the main theorems that guarantee the absence of memory violations and data races from well-typed programs. Chapter 3 presents a similar language that permits reader-writer locks and does not require explicit annotations as its type system is able to automatically infer them. Finally in Chapter 4, we integrate the latter type system to Cyclone and evaluate the performance of concurrent Cyclone programs implementation against C programs.

1.2.2 A hybrid approach to deadlock freedom

In shared memory concurrent programming, deadlocks typically occur as a consequence of cyclic lock acquisition between threads. Two or more threads are deadlocked when each of them is waiting for a lock that has been acquired and is held by another thread. As deadlocks are a serious problem, several methods to achieve deadlock freedom have so far been proposed. In particular, type-based approaches aim for static deadlock freedom guarantees. Most of the proposed type systems in this category [Flan99b, Koba06, Suen08, Vasc10] *prevent* deadlocks by imposing a strict (non-cyclic) lock acquisition order that must be respected throughout the entire program. However, insisting on a global lock ordering limits programming language expressiveness as many correct programs are rejected unnecessarily.

An alternative to deadlock prevention is to employ an approach that dynamically *avoids* deadlocks by utilizing information regarding future lock usage which is provided statically by program analysis. An interesting recent work in this direction is by Boudol [Boud09] who presented a type and effect system for deadlock avoidance when locking is block-structured (e.g. as in Java's `synchronized` blocks). Unfortunately, in Boudol's system the fact that locking is block-structured is a crucial assumption that prohibits the use of his method in many situations. For example, there is a lot of important existing code where locking is used in an unstructured way; cf. the code in Listing 1.1, which is a typical example of systems code. Furthermore note that in low-level languages such as C, even if the programmer adheres to block-structured locking, this is nothing more than a convention: at the source level, any tool needs to deal with separate lock and unlock primitives. Finally, in almost all languages, the restriction that locking is block-structured is usually lifted at the low-level language of the compiler for optimization purposes. This is the type of languages this work targets.

More specifically, we present a method to dynamically avoid deadlocks guided by information about the order of lock and unlock operations which is computed statically via program analysis. The analysis is based on a type and effect system that is general enough to be applicable regardless of how locking is used. Unstructured locking primitives and unrestricted lock aliasing introduce significant complexity to the type system compared with block-structured locking, where lock operations always match up with implicit unlock operations. The proposed type and effect systems guarantee locks are

safely released and acquired in the presence of unrestricted lock aliasing.

In Chapter 5 we review deadlock freedom techniques and provide a gentle introduction to the main ideas behind our deadlock avoidance technique. In Chapter 6 we present a lambda calculus with explicitly annotated functions, mutable references, explicit memory management constructs and unstructured locking primitives. We present its dynamic semantics, static semantics and the main theorems that guarantee the absence of memory violations, data races and deadlocks from well-typed programs. In Chapter 7 we present a lambda calculus with unstructured locking primitives that does not require explicit annotations as its type system is able to infer them. We present its dynamic semantics, static semantics and the main theorems that guarantee deadlock freedom from well-typed programs. In Chapter 8 we present a tool that uses a static analysis based on the type system of Chapter 7 to instrument multithreaded C programs and then links these programs with a run-time system that avoids possible deadlocks. We also report some very promising benchmark results which show that all possible deadlocks can automatically be avoided with only a small run-time overhead. More importantly, this is done without having to modify the original source program by altering the order of resource acquisition operations or by adding annotations.

Chapter 2

A unified hierarchy of regions and locks

2.1 Overview

In this chapter we present a multithreaded language that employs advanced region-based memory management. Regions are organized in a hierarchical manner such that each region is physically allocated within a single parent region and may contain multiple child regions. Implicit reentrant locks are used to protect regions from data races. This hierarchical structure imposes an ownership relation as well as lifetime constraints over regions: each region is protected by its own lock as well as the locks of all its ancestors. In addition, when a region is released, its subregions are automatically released as well. The language permits early release of regions or locks, thereby giving the programmer explicit control over the lifetime of regions and locks. Common multithreaded programming idioms are supported such as data migration and lock transfers between threads. In addition, data can alternate between "thread-local" and "shared" state. The static type system for this language guarantees that well-typed programs are free of memory access violations and data races. As opposed to the language presented in Chapter 3 that permits reader-writer locks and annotation inference, the language presented in this chapter requires explicit type annotations and supports only writer locks. The following section presents our language by example. We then present the formal language, its operational semantics, type system. The main theorems that guarantee the absence of memory violations and data races from well-typed programs are stated and proved.

2.2 Language features through examples

Our regions are lexically-scoped first-class citizens; they are manipulated via explicit handles. For instance, a region handle can be used for releasing a region early, for allocating references and regions within it, or for locking it. Our language uses a *type and effect system* to guarantee that regions and their contents are properly used. The details will be made clear in Section 2.3 and Section 2.5. Here, we present the main features of our language through examples. We try to avoid technical issues as much as possible; however, some characteristics of the type and effect system are revealed in this section and their presence is justified. Furthermore, to simplify the presentation in this section, we use abbreviations for a few language constructs that we expect the readers will find more intuitive. In particular, we use indentation instead of curly braces to denote functions and basic blocks.

We assume the existence of a global *heap* region ρ_H whose handle will be denoted by H. The heap is immortal (i.e., cannot be deallocated), and threads cannot lock it or allocate references to it. Instead, the heap is used only for allocating other regions into it.

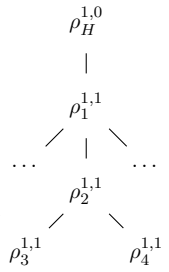
Example 2.1 (Simple region usage) This example shows a typical region use. New regions are allocated via the `region` construct. This construct requires a handle to an existing region (H in this case), in which the new region will be allocated, and introduces a type-level name (ρ) and a fresh handle (h) for the new region. The handle h is then used to allocate a new integer in region ρ; a reference to this integer (z) is created. Finally, the region is deallocated before the end of its lexical scope.

```
region<ρ> @ H;                          // {ρ^{1,1} ▷ ρ_H}
    let z = rnew(h) 42;
        ...
        *z = *z + 5;
        ...
        rfree(h);                       // { }    — empty effect, ρ is no longer alive
        ...
```

The comments on the right-hand side of the example's code show the current *effect*. An effect is roughly a set of *capabilities* that are held at a given program point. Right after creation of region ρ, the entry $\rho^{1,1} \triangleright \rho_H$ is added to the effect; this means that a capability ("$1,1$" — we will later explain what this means) is held for region ρ, which resides in the heap region (ρ_H). Regions start their life as local to a thread and their contents can be directly accessed. For instance, a reference z can be created in ρ, dereferenced and assigned a new value, as long as the type system can verify that a proper capability for ρ is present in the current effect. Deallocation of ρ removes the capability from the effect; once that is done, the region's contents become inaccessible.

Example 2.2 (Hierarchical regions) In the previous example a trivial hierarchy was created by allocating region ρ within the heap H. It is possible to construct richer region hierarchies. As in the previous example, the code below allocates a new region ρ_1 within the heap. Other regions can be then allocated within ρ_1, e.g. ρ_2; this can done by passing the handle of ρ_1 to the region creation construct. Similarly, regions ρ_3 and ρ_4 can be allocated within region ρ_2.

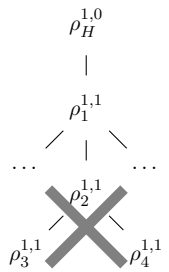
```
region<ρ1> h1 @ H;                      // {ρ1^{1,1} ▷ ρ_H}
    ...
    region<ρ2> h2 @ h1;                 // {ρ1^{1,1} ▷ ρ_H, ρ2^{1,1} ▷ ρ1}
        ...
        region<ρ3> h3 @ h2;             // {ρ1^{1,1} ▷ ρ_H, ρ2^{1,1} ▷ ρ1, ρ3^{1,1} ▷ ρ2}
            region<ρ4> h4 @ h2;         // {ρ1^{1,1} ▷ ρ_H, ρ2^{1,1} ▷ ρ1, ρ3^{1,1} ▷ ρ2, ρ4^{1,1} ▷ ρ2}
                ...
```



Our language allows regions to be allocated at any level of the hierarchy. For instance, it is possible to allocate more regions within region ρ_1, in the lexical scope of region ρ_4.

Example 2.3 (Bulk region deallocation) In the first example a single region was deallocated. That region was a *leaf* node in the hierarchy; it contained no sub-regions. In the general case, when a region is deallocated, the entire subtree below that region is also deallocated. This is what happens if, in the code of the previous example, we deallocate region ρ_2 within the innermost scope; regions ρ_3 and ρ_4 are also deallocated. They are all removed from the current effect and thus are no longer accessible.

```
region<ρ1> h1 @ H;                      // {ρ1^{1,1} ▷ ρ_H}
    ...
    region<ρ2> h2 @ h1;                 // {ρ1^{1,1} ▷ ρ_H, ρ2^{1,1} ▷ ρ1}
        ...
        region<ρ3> h3 @ h2;             // {ρ1^{1,1} ▷ ρ_H, ρ2^{1,1} ▷ ρ1, ρ3^{1,1} ▷ ρ2}
            region<ρ4> h4 @ h2;         // {ρ1^{1,1} ▷ ρ_H, ρ2^{1,1} ▷ ρ1, ρ3^{1,1} ▷ ρ2, ρ4^{1,1} ▷ ρ2}
                ...
                rfree(h2);              // {ρ1^{1,1} ▷ ρ_H}
                ...                     // ρ2, ρ3 and ρ4 are no longer alive
```



Example 2.4 (Region migration) A common multithreaded programming idiom is to use *thread-local* data. At any time, only one thread will have access to such data and therefore no locking is required. A thread can transfer thread-local data to another thread but, doing so, it loses access to the

data. This idiom is known as *migration*. Our language encodes thread-local data and data migration. As we have seen, newly created regions are considered thread-local; a capability for them is added to the current effect. We support data migration by allowing such capabilities to be transferred to other threads.

The following example illustrates region migration. A server thread is defined, which executes an infinite loop. In every iteration, a new region is created and is initialized with client data. The contents of the region are then processed and finally transferred to a newly created (spawned) thread.

```
void server ()
   while (true)
      region<ρ> h @ H;                    // {ρ^{1,1} ▷ ρ_H}
         let z = wait_data(h);            // region ρ is thread-local
            process(z);
            spawn output(h, z);           // { } — empty effect, ρ migrates to output
            ...                           // ρ cannot be accessed here
```

The server thread accepts the heap region and its handle. Within the infinite loop, it allocates a new region ρ in the heap. Its handle h is passed to function `wait_data`, which is supposed to fill the region ρ with client data (z). Function `process` is then called and works on the data. Until this point, region ρ is thread-local and accessible to the server thread, so no explicit locking is required. Now, let us assume that we want the processed data to be output by a different thread, e.g. to avoid an unnecessary delay on the server thread. A new thread `output` is spawned and receives the region handle h and the reference z to the client data. The capability $\rho^{1,1} \triangleright \rho_H$ is removed from the effect of `server` and is added to the input effect of thread `output`. Therefore, region ρ has now become thread-local to thread `output`, which can access it directly, while it is no longer accessible to the server thread.

Example 2.5 (Region sharing) In the previous examples, thread-local regions were associated with capability "$1, 1$". In general, a capability for a region consists of two natural numbers; the first denotes the *region count*, whereas the second denotes the *lock count*. When the region count is positive, the region is definitely alive. Similarly, when the lock count is positive, memory accesses to this region's contents are guaranteed to be race free. Capabilities with counts other than 1 can be used for *sharing* regions between threads.

Multithreaded programs often share data for communication purposes. In this example, a server thread almost identical to that of the previous example is defined. The programmer's intention here, however, is to process the data and display it in parallel. Therefore, the `output` thread is spawned first and then the server thread starts processing the data.

```
void server ()
   while (true)
      region<ρ> h @ H;                    // {ρ^{1,1} ▷ ρ_H}
         let z = wait_data(h);
            share(h); unlock(h);          // {ρ^{2,0} ▷ ρ_H}
            spawn output(h, z);           // {ρ^{1,0} ▷ ρ_H} — output consumes ρ^{1,0} ▷ ρ_H
            lock(h);                      // {ρ^{1,1} ▷ ρ_H}
            process(z);
            unlock(h);                    // {ρ^{1,0} ▷ ρ_H}
```

Function `share` increases the region count and function `unlock` decreases the lock count. As a consequence, starting with capability $\rho^{1,1} \triangleright \rho_H$, we end up with $\rho^{2,0} \triangleright \rho_H$. When `output` is spawned, it consumes "half" of this capability ($\rho^{1,0} \triangleright \rho_H$); the remaining "half" ($\rho^{1,0} \triangleright \rho_H$) is still held by the server thread. Region ρ is now shared between the two threads; however, none of them can access its data directly, as this may lead to a data race. The `lock` and `unlock` functions have to be used for explicitly locking and unlocking the region, before safely accessing its contents. The server thread avoids locking the region thus allowing the `output` thread to gain access to the region when needed.

Example 2.6 (Hierarchical locking) In the previous example, locking and unlocking was performed on a leaf region. In general, locking a region in the hierarchy has the effect of atomically locking its subregions as well. A region is accessible when it has been locked by the current thread or when at least one of its ancestors has been locked.

Hierarchical locking can be useful when a set of locks needs to be acquired atomically. In this example, we assume that two hash tables (tbl_1 and tbl_2) are used. An object with a given key must be removed from tbl_1, which resides in region ρ_1, and must be inserted in tbl_2, which resides in region ρ_2. We can atomically acquire access to both regions ρ_1 and ρ_2, by locking a common ancestor of theirs.

```
lock(h);                                          // the handle of a common ancestor of ρ1 and ρ2
let obj = hash_remove<ρ1>(tbl1, key);
    hash_insert<ρ2>(tbl2, key, obj);
unlock(h);
```

Example 2.7 (Region aliasing) An expressive language with regions will have to support region polymorphism, which invariably leads to *region aliasing*. This must be handled with caution, as a naïve approach may cause unsoundness. In the examples that follow, we discuss how region aliasing is used in our language as well as the restrictions that we impose to guarantee safety.

Function `swap`, which is polymoprhic in respect to regions ρ_1 and ρ_2 (polymorphic type variables ρ_1 and ρ_2 are specified next to the function name in the prototype declaration), accepts two integer references, residing in regions ρ_1 and ρ_2, and swaps their contents. It assumes that both regions are already locked and remain locked when the function returns.

```
// ρ1 and ρ2 are locked
void swap<ρ1, ρ2>(int * ρ1 x, int * ρ2 y)
    let z = *x;                                   // OK: ρ1 is locked
        *x = *y;                                  // OK: ρ1 and ρ2 are locked
        *y = z;                                   // OK: ρ2 is locked
```

In order to instantiate ρ_1 and ρ_2 with the same region ρ, we can create two lock capabilities by using the `lock` function twice on ρ's handle h. Of course, the second use of `lock` will succeed immediately, as the region has already been locked by the same thread.

```
...                                               // {ρ^{2,0} ▷ ρH}
lock(h); lock(h);                                 // {ρ^{2,2} ▷ ρH}
swap<ρ, ρ>(a, b);                                 // each ρ parameter requires ρ^{1,1} ▷ ρH
unlock(h); unlock(h);                             // {ρ^{2,0} ▷ ρH}
```

Example 2.8 (Reentrant locks) Region aliasing introduces the need for reentrant locks. To see this, let us change the swapping function of the previous example, so that it receives two references in unlocked regions. For swapping their contents, it will have to acquire locks for the two regions (and release them, when they are no longer needed).

```
// ρ1 and ρ2 are unlocked
void swap<ρ1, ρ2>(region<ρ1> h1, region<ρ2> h2, int * ρ1 x, int * ρ2 y)
    lock(h1);
    let z = *x;                                   // OK: ρ1 is locked
        lock(h2);
        *x = *y;                                  // OK: ρ1 and ρ2 are locked
        unlock(h1);
        *y = z;                                   // OK: ρ2 is locked
        unlock(h2);                               // all locks can be released
```

Suppose again that we are to instantiate ρ_1 and ρ_2 with the same region ρ.

```
...                                          // {ρ^{2,0}▷ ρ_H}
swap<ρ, ρ>(h, h, a, b);                      // each ρ parameter requires ρ^{1,0}▷ ρ_H
```

We can easily see, however, that the run-time system cannot use binary locks; in that case, `swap`$<\rho, \rho>$ would either come to a deadlock, waiting to obtain once more the lock that it has already acquired, or — worse — it might release the lock early (at `unlock`(h_1)) and allow a data race to occur. To avoid unsoundness, we use *reentrant locks*: lock counts are important both for static typing and for the run-time system. A lock with a positive run-time count can immediately be acquired again, if it was held by the same thread. Moreover, a lock is released only when its run-time count becomes zero.

Example 2.9 (Pure and impure capabilities) Unrestricted region aliasing leads to unsoundness. Consider function `bad`, which accepts two integer references (x and y) in regions ρ_1 and ρ_2, which are both locked. It lets ρ_1 migrate to a new thread and passes x as a parameter. It then assigns a value to y.

```
// ρ1 and ρ2 are locked
void bad<ρ1, ρ2>(int *ρ1 x, int *ρ2 y)
    spawn f(x);                              // ρ1 migrates to f while locked
    *y = 7;                                  // OK: ρ2 is still locked — WRONG!
```

A data race may occur if we call `bad` as follows; both threads have access to a, each holding a lock for ρ.

```
bad<ρ, ρ>(a, a);                             // each ρ parameter requires ρ^{1,1}▷ ρ_H
```

The cause of the unsoundness is that, in this last call to `bad`$<\rho, \rho>$, we allowed a single capability $\rho^{2,2} \triangleright \rho_H$ to be divided in two distinct capabilities $\rho^{1,1} \triangleright \rho_H$. More specifically, we divided the lock count in two and created two distinct lock capabilities, one of which escaped to a different thread through region migration. To resolve the unsoundness, we introduce the notion of *pure* (i.e., full) and *impure* (i.e., divided) capabilities. For instance, $\rho^{2,2} \triangleright \rho_H$ is a pure capability; when we divide it we obtain two impure halves, which we denote as $\overline{\rho^{1,1} \triangleright \rho_H}$. Impure capabilities cannot be given to newly spawned threads when their lock count is positive. In contrast with pure capabilities, they represent inexact knowledge of a region's counts.

2.3 Formal language

The language syntax is illustrated in Figure 2.1.[1] The core expressions include variables (x), constants (c), functions, and function application. Function application terms are annotated with a *calling mode* (ξ). The calling mode specifies whether a function application should be executed sequentially (`seq`) or in parallel (`par`). Monomorphic functions ($\lambda x.\, e$) must be annotated with their type (τ). Our language also includes region-polymorphic functions ($\Lambda\rho.\, f$) and region application ($e[\rho]$). Other constructs can be easily included, as long as conservative choices are made to ensure the soundness of the type system (e.g., in the standard `if-then-else` construct, both branches should produce the same effect).

The construct `newrgn`^{r} ρ, x @ e_1 `in` e_2 allocates a fresh region ρ at the region indicated by handle e_1, and binds x to the *handle* of ρ. Both ρ and x are lexically bound to the scope of e_2. The new region must be explicitly released within e_2. The region allocation construct is annotated with the parent region name r, which is only required for the type safety proof.

The constructs for manipulating references are standard. A newly allocated memory cell is returned by `new` e_1 @ e_2, where e_1 is an initializer expression for the new cell and handle e_2 indicates

[1] The constructs $\texttt{rgn}_\iota$, $\texttt{loc}_l$ and $\texttt{pop}_\gamma$ e are not considered part of the language. They are only introduced during program evaluation. We defer the discussion about them until Section 2.4.

$$
\begin{array}{lrcl}
\textbf{Expression} & e & ::= & x \mid \mathsf{c} \mid f \mid (e\ e)^{\xi} \mid e[r] \\
& & \mid & \mathtt{new}\, e \mathbin{@} e \mid e := e \mid \mathtt{deref}\, e \\
& & \mid & \mathtt{newrgn}^{r}\ \rho, x \mathbin{@} e\ \mathtt{in}\ e \\
& & \mid & \mathtt{cap}^{r}_{\eta}\ e \mid \mathtt{rgn}_{\iota} \mid \mathtt{loc}_{l} \mid \mathtt{pop}_{\gamma}\ e \\
\textbf{Function} & f & ::= & \lambda x.\, e\ \mathtt{as}\ \tau \xrightarrow{\gamma \to \gamma} \tau \mid \Lambda\rho.\, f \\
\textbf{Type} & \tau & ::= & b \mid \langle\rangle \mid \tau \xrightarrow{\gamma \to \gamma} \tau \mid \forall \rho.\, \tau \\
& & \mid & \mathtt{ref}(\tau, r) \mid \mathtt{rgn}(r) \\
\textbf{Effect} & \gamma & ::= & \emptyset \mid \gamma, r^{\kappa} \triangleright \pi
\end{array}
$$

$$
\begin{array}{lrcl}
\textbf{Capability kind} & \psi & ::= & \mathsf{rg} \mid \mathsf{lk} \\
\textbf{Capability op} & \eta & ::= & \psi+ \mid \psi- \\
\textbf{Region} & r & ::= & \rho \mid \iota \mid \iota @ n \\
\textbf{Capability} & \kappa & ::= & n, n \mid \overline{n, n} \\
\textbf{Region parent} & \pi & ::= & r \mid \bot \\
\textbf{Calling mode} & \xi & ::= & \mathsf{seq} \mid \mathsf{par}
\end{array}
$$

Figure 2.1: Syntax.

the region in which the new cell will be allocated. Standard assignment and dereference operators complete the picture. A region can be released either by deallocation or by transferring its ownership to another thread. At any given program point, each region is associated with a *capability* (κ). Capabilities consist of two natural numbers, the *capability counts*: the *region* count and *lock* count, which denote whether a region is live and locked respectively. When first allocated, a region starts with capability $(1, 1)$, meaning that it is live and locked, so that it can be accessed directly with no additional overhead. This is our equivalent of a thread-local region.

By using the construct $\mathtt{cap}^{r}_{\eta}\ e$, a thread can *increment* or *decrement* the capability counts of some region r whose handle is specified in e. The annotation (r) on the `cap` construct is only required for the type safety proof. The capability operator η can be, e.g., $\mathsf{rg}+$ (meaning that the region count is to be incremented) or $\mathsf{lk}-$ (meaning that the lock count is to be decremented). Incrementing counts is essential for sharing regions among threads and for region aliasing. Furthermore, incrementing a lock count from 0 to 1 amounts to acquiring a region lock, which may have to block the current thread if the lock is held by another thread. On the other hand, decrementing counts amounts to releasing capabilities. When a region count reaches zero, no subsequent operations can be performed on this region and the region may be physically deallocated (if no other threads are using it). When a lock count reaches zero, the region is unlocked, but it may still be *protected* by a locked ancestor region. As we explained, capability counts determine the validity of operations on regions and references. All memory-related operations require that the involved regions are live, i.e., the region count is greater than zero. Assignment and dereference can be performed only when the corresponding region is live and protected.

A capability of the form (n_1, n_2) is called a *pure* capability, whereas a capability of the form $(\overline{n_1, n_2})$ is called an *impure* capability. In both cases, it is implied that the current thread can decrement the region count n_1 times and the lock count n_2 times. Impure capabilities are obtained by splitting pure or other impure capabilities into several pieces, e.g., the pure capability $(3, 2)$ can be split into two impure capabilities $(\overline{2, 1})$ and $(\overline{1, 1})$, in the same spirit as *fractional capabilities* [Boyl03]. Splitting a linear resource into multiple pieces is particularly useful for region aliasing (e.g., the same region can be passed to a function in the place of two distinct region parameters). An impure capability implies that our knowledge of the region and lock counts held by the current thread is inexact. Under certain circumstances, the use of impure capabilities must be disallowed; e.g., an impure capability with a non-zero lock count cannot be passed to another thread, as it is unsound to allow two threads to simultaneously access the same region. Capability splitting takes place automatically with function application.

2.4 Operational semantics

We define a *small-step* operational semantics for our language, using two evaluation relations, at the level of *threads* and *expressions* (Figure 2.3 and Figure 2.4 on the next page). The thread evaluation

Stack	$\sigma ::= \emptyset \mid \sigma;\gamma$	$E ::= \Box \mid (E\ e)^{\xi} \mid (v\ E)^{\xi} \mid E\,[r]$
Hierarchy	$\delta ::= \emptyset \mid \delta, n \mapsto \sigma$	$\mid \ \texttt{newrgn}^{r}\ \rho, x\ @\ E\ \texttt{in}\ e \mid \texttt{cap}^{r}_{\eta}\ E$
Contents	$H ::= \emptyset \mid H, \ell \mapsto v$	$\mid \ \texttt{new}\ v\ @\ E \mid \texttt{deref}\ E \mid E := e$
Region list	$S ::= \emptyset \mid S, \imath \mapsto H$	$\mid \ v := E \mid \texttt{new}\ E\ @\ e \mid \texttt{pop}_{\gamma}\ E$
Threads	$T ::= \emptyset \mid T, n : e$	
Configuration	$C ::= \delta; S; T$	

Figure 2.2: Configuration, store, threads and evaluation contexts.

$$\frac{\begin{array}{l} v_1 \equiv \lambda x.\, e\ \mathsf{as}\ \tau_1 \xrightarrow{\gamma_1 \to \gamma_2} \tau_2 \qquad \delta = \delta'', n \mapsto \sigma;\gamma \qquad \text{fresh } n' \\ \mathsf{par} \vdash \gamma' = \emptyset \oplus (\gamma \ominus \gamma_1) \qquad \delta' = \delta'', n \mapsto \sigma;\gamma', n' \mapsto \emptyset;\gamma_1 \end{array}}{\delta; S; T, n : E[(v_1\ v)^{\mathsf{par}}] \rightsquigarrow \delta'; S; T, n : E[()], n' : (v_1\ v)^{\mathsf{seq}}} \quad \textit{(E-SN)}$$

$$\frac{\delta = \delta'', n \mapsto \sigma \qquad \sigma; S; e \to \sigma'; S'; e' \qquad \delta' = \delta'', n \mapsto \sigma' \qquad \vdash \delta'}{\delta; S; T, n : E[e] \rightsquigarrow \delta'; S'; T, n : E[e']} \quad \textit{(E-S)}$$

$$\frac{\delta = \delta', n \mapsto (\emptyset;\emptyset)}{\delta; S; T, n : () \rightsquigarrow \delta'; S; T} \quad \textit{(E-T)}$$

Figure 2.3: Thread evaluation relation $C \rightsquigarrow C'$.

relation transforms *configurations*. A configuration C (see Figure 2.2) consists of global hierarchy δ, an abstract *store* S and a thread map T.[2] The global hierarchy δ maps thread identifiers (n) to stacks (σ). A thread stack σ is a list of frames (γ) and represents a hierarchy of regions accessible to a thread. Each frame γ represents the portion of σ that is accessible to the function that is currently executed. Notice, that frames include region counts. A frame is a list of elements of the form $r^{\kappa}{\triangleright}\,\pi$, denoting that region r is associated with count κ and has parent π, which can be another region or $\bot$. Regions whose parents are $\bot$ are considered as roots in a region hierarchy. A store S maps region identifiers ($\imath$) to heaps (H). A heap H, maps memory locations to values. A thread map T associates thread identifiers to expressions (i.e., threads).

A *thread evaluation context* E (Figure 2.2) is defined as an expression with a *hole*, represented as $\Box$. The hole indicates the position where the next reduction step can take place. Our notion of evaluation context imposes a call-by-value evaluation strategy to our language. Subexpressions are evaluated in a left-to-right order.

We assume that concurrent reduction events can be totally ordered [Lamp79]. At each step, a random thread (n) is chosen from the thread list for evaluation (Figure 2.3). It should be noted that the thread evaluation rules are the only *non-deterministic* rules in the operational semantics of our language; in the presence of more than one active thread, our semantics does not specify which one will be selected for evaluation. Threads that have completed their evaluation, have released all regions used by them, and have been reduced to *unit* values, represented as $()$, are removed from the active thread list (rule *E-T*). Rule *E-S* reduces some thread n via the expression evaluation relation. Notice, that rule *E-S* only modifies the stack of thread n and requires that the resulting hierarchy δ' is *consistent* ($\vdash \delta'$ — defined in Figure 2.5): regions accessible to thread n should be inaccessible to other threads and regions having positive pure capabilities can only be live at a *single* stack frame of thread n.[3] Therefore, the operational semantics will get stuck if the mutual exclusion protocol is unsatisfied.

[2] The order of elements in comma-separated lists, e.g. in a store S or in a list of threads T, is unimportant; we consider all list permutations as equivalent.

[3] The second invariant ensures that regions with positive pure capabilities can safely be passed to other threads (e.g., locked). This is sound when the current thread has no more counts of such regions in other stack frames.

$$\frac{\sigma = \sigma'; \gamma \qquad \mathsf{seq} \vdash \gamma = \gamma_1 \oplus \gamma_r \qquad \sigma'' = \sigma'; \gamma_r; \gamma_1}{\sigma; S; ((\lambda x.\, e \;\mathsf{as}\; \tau_1 \xrightarrow{\gamma_1 \to \gamma_2} \tau_2)\ v)^{\mathsf{seq}} \to \sigma''; S; \mathsf{pop}_{\gamma_r}\ e[v/x]} \quad \textit{(E-A)}$$

$$\frac{\sigma = \sigma'; \gamma_r; \gamma' \qquad \mathsf{seq} \vdash \gamma'' = \gamma' \oplus (\gamma_r \ominus \emptyset) \qquad \sigma'' = \sigma'; \gamma''}{\sigma; S; \mathsf{pop}_{\gamma_r}\ v \to \sigma''; S; v} \quad \textit{(E-E)}$$

$$\frac{\sigma = \sigma'; \gamma \qquad \text{is\_live}(\gamma, r) \qquad \text{fresh } \imath \qquad \sigma'' = \sigma'; \gamma, \imath^{1,1} \triangleright r}{\sigma; S; \mathsf{newrgn}^r\ \rho, x\ @\ \mathsf{rgn}_{\bar{r}}\ \mathsf{in}\ e \to \sigma''; S, \imath \mapsto \emptyset; e[\imath/\rho][\mathsf{rgn}_\imath/x]} \quad \textit{(E-NG)}$$

$$\frac{\begin{array}{c}\sigma = \sigma'; \gamma \qquad \text{is\_live}(\gamma, r) \qquad \gamma = \gamma', r^{\kappa} \triangleright \pi \\ \kappa' = [\![\eta]\!]\,(\kappa) \qquad \sigma'' = \sigma'; \text{live}(\gamma', r^{\kappa'} \triangleright \pi)\end{array}}{\sigma; S; \mathsf{cap}^r_\eta\ \mathsf{rgn}_{\bar{r}} \to \sigma''; S; ()} \quad \textit{(E-C)}$$

$$\frac{\sigma = \sigma'; \gamma \qquad \text{is\_live}(\gamma, r) \qquad \text{fresh } \ell}{\sigma; S; \mathsf{new}\ v\ @\ \mathsf{rgn}_{\bar{r}} \to \sigma; S[\bar{r} \mapsto S(\bar{r}), \ell \mapsto v]; \mathsf{loc}_\ell} \quad \textit{(E-NR)}$$

$$\frac{\sigma = \sigma'; \gamma \qquad \text{is\_accessible}(\gamma, r) \qquad (\ell \mapsto v_1) \in S(\bar{r})}{\sigma; S; \mathsf{loc}_\ell := v \to \sigma; S(\bar{r})[\ell \mapsto v]; ()} \quad \textit{(E-AS)}$$

$$\frac{\sigma = \sigma'; \gamma \qquad \text{is\_accessible}(\gamma, r) \qquad (\ell \mapsto v) \in S(\bar{r})}{\sigma; S; \mathsf{deref}\ \mathsf{loc}_\ell \to \sigma; S; v} \quad \textit{(E-D)}$$

$$\frac{\text{fresh } n'}{\sigma; S; (\Lambda\rho.\, f)[r] \to \sigma; S; f[\bar{r}@n'/\rho]} \quad \textit{(E-RP)}$$

Figure 2.4: Expression evaluation relation $\sigma; S; e \to \sigma'; S'; e'$.

Our approach differs from related work, e.g. the work of Grossman [Gros03], where a special kind of value *junk*$_v$ is often used as an intermediate step when assigning a value v to a location, before the real assignment takes place, and type safety guarantees that no junk values are ever read.

When a parallel function application redex is detected within the evaluation context of a thread, a new thread is created (rule *E-SN*). The redex is replaced with a unit value in the currently executed thread and a new thread is added to the thread list, with a *fresh* thread identifier. The calling mode of the application term is changed from parallel to sequential. The topmost frame of the spawning thread (γ) is split into two frames γ' and γ_1 so that the intersection of regions locked in γ' and in γ_1 is empty.[4] If it is impossible to split γ, the thread evaluation relation gets stuck. Notice, that γ_1 is an effect annotation of the function abstraction. Frame γ is then replaced by γ' and γ_1 becomes the initial frame of the new thread.

The expression evaluation relation (defined in Figure 2.4) rewrites tuples of the form $\sigma; S; e$, where σ is a thread local stack, S is the global store, and e is an expression.

Constant regions may be of the form $\imath@n$, which is a constant region $\imath$ tagged with a unique identifier n. The region application (*E-RP*) rule introduces tags during substitution so as to prevent the existence of duplicate region names in function effects.

Hereon, the symbol γ means "the topmost frame of the currently executed thread n". The sequential function application (*E-A*) rule splits γ into two stack frames γ_1 and γ_r such that γ_1 matches the effect expected by the lambda abstraction, and substitutes the sequence of stack frames $\gamma_r; \gamma_1$ for γ. The function body is placed within a pop construct, which is annotated with frame γ_r. A pop con-

[4] The rules for splitting effects are defined in Figure 2.7 and discussed in Section 2.5. Until then, the judgement $\xi \vdash \gamma' = \gamma_2 \oplus (\gamma \ominus \gamma_1)$ should be interpreted as saying that the effect γ' is what we get if we start with γ, remove the input effect γ_1 of the function that we are calling and (when the function returns) add the function's output effect γ_2.

$$\frac{(r^{\kappa}\triangleright\bot)\in\gamma \quad \mathrm{rg}(\kappa)>0}{\mathrm{is\_live}(\gamma,r)} \qquad \frac{\gamma=\gamma',r^{\kappa}\triangleright r' \quad \mathrm{rg}(\kappa)>0 \quad \mathrm{is\_live}(\gamma',r')}{\mathrm{is\_live}(\gamma,r)}$$

$$\frac{(r^{\kappa}\triangleright\pi)\in\gamma \quad \mathrm{lk}(\kappa)>0 \quad \mathrm{is\_live}(\gamma,r)}{\mathrm{is\_accessible}(\gamma,r)}$$

$$\frac{\gamma=\gamma',r^{\kappa}\triangleright r' \quad \mathrm{lk}(\kappa)=0 \quad \mathrm{rg}(\kappa)>0 \quad \mathrm{is\_accessible}(\gamma',r')}{\mathrm{is\_accessible}(\gamma,r)}$$

$$\frac{\begin{array}{c}\sigma\simeq\sigma_1;\gamma,\imath^{\kappa}\triangleright\pi\Rightarrow\mathrm{is\_pure}(\kappa)\wedge\mathrm{rg}(\kappa)=0\wedge\\ \imath\notin\mathrm{dom}(\emptyset;\gamma)\wedge\sigma_1\neq\emptyset\Rightarrow\mathrm{zero\_pure}(\sigma_1,\imath)\end{array}}{\mathrm{zero\_pure}(\sigma,\imath)}$$

$$\frac{\vdash\delta \quad \sigma\vdash\delta}{\vdash\delta,n\mapsto\sigma} \qquad \frac{}{\vdash\emptyset}$$

$$\frac{\begin{array}{c}\sigma\vdash\delta\\ \forall\imath\in\mathrm{dom}(\sigma).\mathrm{is\_accessible}(\sigma,\imath)\Rightarrow\neg\mathrm{is\_accessible}(\sigma',\imath)\end{array}}{\sigma\vdash\delta,n\mapsto\sigma'}$$

$$\frac{\begin{array}{c}\forall\imath\in\mathrm{dom}(\sigma).\sigma\simeq\sigma_1;\gamma,\imath^{\kappa}\triangleright\pi+\sigma_2\wedge\mathrm{rg}(\kappa)>0\wedge\mathrm{is\_pure}(\kappa)\\ \Rightarrow\mathrm{zero\_pure}(\sigma_1,\imath)\wedge\imath\notin\mathrm{dom}(\sigma_2;\gamma)\end{array}}{\sigma\vdash\emptyset}$$

$$\begin{array}{lcl}
 & & \text{if } \eta\equiv\psi\pm\wedge\mathrm{is_pure}(\kappa)\Leftrightarrow\mathrm{is_pure}(\kappa')\wedge\\
[\![\eta]\!](\kappa) & = & \kappa' \quad (\psi=\mathsf{rg}\Rightarrow\mathrm{rg}(\kappa')=\mathrm{rg}\pm1\wedge\mathrm{lk}(\kappa')=\mathrm{lk}(\kappa))\wedge\\
 & & (\psi=\mathsf{lk}\Rightarrow\mathrm{lk}(\kappa')=\mathrm{lk}\pm1\wedge\mathrm{rg}(\kappa')=\mathrm{rg}(\kappa))\\
\bar{r} & = & \begin{cases}\imath & \text{if } r=\imath\\ \bar{r'} & \text{if } r=r'@n'\end{cases}\\
\mathrm{is_pure}(\kappa) & = & \exists n_1.\,\exists n_2.\,\kappa=n_1,n_2\\
\mathrm{rg}(\kappa) & = & n_1 \quad \text{if } \kappa=n_1,n_2\vee\kappa=\overline{n_1,n_2}\\
\mathrm{lk}(\kappa) & = & n_2 \quad \text{if } \kappa=n_1,n_2\vee\kappa=\overline{n_1,n_2}\\
\mathrm{live}(\gamma) & = & \{\, r^{\kappa}\triangleright\pi \mid (r^{\kappa}\triangleright\pi)\in\gamma\wedge\mathrm{is_live}(\gamma,r)\,\}
\end{array}$$

Figure 2.5: Auxiliary functions and predicates.

struct must not be contained in the original program, and must only appear during program evaluation. Rule *E-E* eliminates pop constructs, when the function body has been reduced to a value and the annotation γ_r of pop matches the frame preceding the topmost frame γ'. Frames γ_r are γ' are joined to form a new frame γ'', which replaces them on the current stack.

The remaining rules of Figure 2.4 make use of judgements $is\_live(\gamma,r)$ and $is\_accessible(\gamma,r)$ (auxiliary functions and predicates are defined in Figure 2.5) to establish that a region r is *live* and *accessible* in a frame γ. A region r is *live* in γ when the region count of each region in the path between r and the root region is positive. A region r is *accessible* in γ when it is live and there exists at least one region in the path between r and the root region with a positive lock count. We also define the following partial functions: $\bar{r}$ removes the unique identifier from a tagged region, $[\![\eta]\!](\kappa)$ decrements or increments the region or lock field of κ by one, according to operation specified by η, and finally $\mathrm{live}(\gamma)$ selects a subset of γ so that all regions in that subset are live.

Rule *E-NG* requires that region r is live in γ, adds a fresh and empty region $\imath$ to S and adds the dynamic effect of $\imath$ to γ, which specifies that r is the parent of $\imath$ and that $\imath$ has region and lock count of one. Rule *E-C* requires that region r is live in γ, substitutes $[\![\eta]\!](\kappa)$ for κ in γ at the exponent of

$$\frac{R;M;\Delta;\Gamma \vdash e_1 : \tau_1 \xrightarrow{\gamma_1 \to \gamma_2} \tau_2 \,\&\, (\gamma;\gamma') \qquad \xi = \mathsf{par} \Rightarrow \tau_2 = \langle\rangle \\ R;M;\Delta;\Gamma \vdash e_2 : \tau_1 \,\&\, (\gamma';\gamma'') \qquad \xi \vdash \gamma''' = \gamma_2 \oplus (\gamma'' \ominus \gamma_1)}{R;M;\Delta;\Gamma \vdash (e_1\ e_2)^{\xi} : \tau_2 \,\&\, (\gamma;\gamma''')} \textit{(T-AP)}$$

$$\frac{R;M;\Delta;\Gamma \vdash e : \mathtt{ref}(\tau, r) \,\&\, (\gamma;\gamma') \qquad \mathit{is\_accessible}(\gamma', r)}{R;M;\Delta;\Gamma \vdash \mathtt{deref}\ e : \tau \,\&\, (\gamma;\gamma')} \textit{(T-D)}$$

$$\frac{R;M;\Delta;\Gamma \vdash e_1 : \mathtt{rgn}(r) \,\&\, (\gamma;\gamma') \qquad \mathit{is\_live}(\gamma', r) \qquad R;\Delta \vdash \tau \\ R;M;\Delta,\rho;\Gamma,x : \mathtt{rgn}(\rho) \vdash e_2 : \tau \,\&\, (\gamma', \rho^{1,1} \triangleright r;\gamma'') \qquad \rho \notin \mathit{dom}(\gamma'')}{R;M;\Delta;\Gamma \vdash \mathtt{newrgn}^r\ \rho, x\ @\ e_1\ \mathtt{in}\ e_2 : \tau \,\&\, (\gamma;\gamma'')} \textit{(T-NG)}$$

$$\frac{R;M;\Delta;\Gamma \vdash e_1 : \tau \,\&\, (\gamma;\gamma') \qquad \mathit{is\_live}(\gamma'', r) \\ R;M;\Delta;\Gamma \vdash e_2 : \mathtt{rgn}(r) \,\&\, (\gamma';\gamma'')}{R;M;\Delta;\Gamma \vdash \mathtt{new}\ e_1\ @\ e_2 : \mathtt{ref}(\tau, r) \,\&\, (\gamma;\gamma'')} \textit{(T-NR)}$$

$$\frac{\mathit{is\_live}(\gamma', r^{\kappa} \triangleright \pi, r) \qquad \gamma'' = \mathit{live}(\gamma', r^{\kappa'} \triangleright \pi) \\ R;M;\Delta;\Gamma \vdash e_1 : \mathtt{rgn}(r) \,\&\, (\gamma;\gamma', r^{\kappa} \triangleright \pi) \qquad \kappa' = [\![\eta]\!]\,(\kappa)}{R;M;\Delta;\Gamma \vdash \mathtt{cap}^r_{\eta}\ e_1 : \langle\rangle \,\&\, (\gamma;\gamma'')} \textit{(T-CP)}$$

Figure 2.6: Selected typing rules.

$$\frac{\xi \vdash \gamma = \gamma_1 \oplus \gamma_r \qquad \xi \vdash \gamma' = \gamma_2 \oplus \gamma_r \\ \gamma'' = \mathit{live}(\gamma') \qquad \mathit{ok}(\gamma_1;\gamma_2) \qquad \xi = \mathsf{par} \Rightarrow \gamma_2 = \emptyset}{\xi \vdash \gamma'' = \gamma_2 \oplus (\gamma \ominus \gamma_1)} \textit{(ESJ)}$$

$$\frac{\xi \vdash \gamma, r^{\kappa_2} \triangleright \pi = \gamma_1 \oplus \gamma_2 \\ \xi \vdash \kappa = \kappa_1 + \kappa_2 \qquad \pi \simeq \pi' \qquad r' \simeq r}{\xi \vdash \gamma, r^{\kappa} \triangleright \pi = \gamma_1, r'^{\kappa_1} \triangleright \pi' \oplus \gamma_2} \textit{(ES-C)} \qquad \frac{}{\xi \vdash \gamma = \emptyset \oplus \gamma} \textit{(ES-N)}$$

$$\frac{\mathit{rg}(\kappa) = \mathit{rg}(\kappa_1) + \mathit{rg}(\kappa_2) \qquad \mathit{lk}(\kappa) = \mathit{lk}(\kappa_1) + \mathit{lk}(\kappa_2) \qquad \mathit{is\_pure}(\kappa) \Leftrightarrow \mathit{is\_pure}(\kappa_2) \\ \mathit{is\_pure}(\kappa_1) \Rightarrow \kappa = \kappa_1 \qquad \xi = \mathsf{par} \wedge \neg \mathit{is\_pure}(\kappa_1) \Rightarrow \mathit{lk}(\kappa_1) = 0}{\xi \vdash \kappa = \kappa_1 + \kappa_2} \textit{(CS)}$$

Figure 2.7: Effect and capability splitting.

r, and removes dead regions from the resulting frame. Rule *E-NR* requires that region r is live in γ and updates the heap of r with a fresh location ℓ mapping to value v. Notice, that r may be *unlocked*. Rules *E-AS* and *E-D* require that *some region* r, which contains the location (ℓ) being accessed, must be *accessible* in γ. Therefore, the semantics will get stuck when a thread attempts to access a memory location without having acquired an appropriate lock for this location.

2.5 Static semantics

We discuss the most interesting aspects of our type system. We employ a *type and effect system* to enforce memory and race safety invariants. Effects (γ) are used to statically track region capabilities.

The syntax of types has been defined in Figure 2.1. A collection of base types b is assumed; the syntax of values belonging to these types and operations upon such values are omitted. We assume the existence of a *unit* base type, which we denote by $\langle\rangle$. Region handle types $\mathtt{rgn}(r)$ and reference types $\mathtt{ref}(\tau, r)$ are associated with a type-level region r. Monomorphic function types carry an *input*

and an *output effect*. A well-typed expression e has a type τ under an input effect γ and results in an output effect γ'. The typing relation (see Figure 2.6) is denoted by $R; M; \Delta; \Gamma \vdash e : \tau \,\&\, (\gamma;\gamma')$ and uses four typing contexts: a set of region literals (R), a mapping of locations to types (M), a set of region variables (Δ), and a mapping of term variables to types (Γ).

The typing rule for function application (*T-AP*) splits the output effect of e_2 (γ'') by subtracting the function's input effect (γ_1). It then joins the remaining effect with the function's output effect (γ_2). In the case of parallel application, rule *T-AP* also requires that the return type is unit. The splitting and joining of effects is controlled by the judgement $\xi \vdash \gamma'' = \gamma_2 \oplus (\gamma \ominus \gamma_1)$, which is defined by rule *ESJ* in Figure 2.7. It uses the simpler judgement $\xi \vdash \gamma = \gamma_1 \oplus \gamma_2$ for just splitting (not joining) effects, which is defined by rules *ES-N* and *ES-C*. This, in turn, uses a third judgement $\xi \vdash \kappa = \kappa_1 \oplus \kappa_2$ for splitting capabilities, which is defined by rule *CS*. Some auxiliary functions and predicates are defined in Figure 2.5.

As defined in Figure 2.7 the rules for splitting and joining effects enforce the following properties:

- The effect after the join satisfies the liveness invariant, i.e., all regions that appear in it are live (their region counts and those of all their ancestors are positive). This is enforced by $\gamma'' = \mathit{live}(\gamma')$ in rule *ESJ*.
- For parallel application, the thread output effect must be empty. In other words, every thread is obliged to deallocate all its regions and release the locks that it holds.
- Regions with pure capabilities cannot appear more than once in a function's input or output effect. In other words, region aliasing is only allowed with impure capabilities. This is enforced by $\mathit{ok}(\gamma_1; \gamma_2)$ in rule *ESJ*.
- Capability and effect splitting is not symmetric. In $\xi \vdash \kappa = \kappa_1 \oplus \kappa_2$, capability κ_1 goes to the function being called and capability κ_2 is what "stays behind" (the same is true for effects). Under this light, the rule *CS* ensures three things:
  - Capabilities that stay behind are of the same purity as the original ones. This implies that capabilities do not change purity as a result of splitting and joining.
  - If a pure capability is passed to a function, nothing stays behind.
  - If an impure capability is passed to a spawned thread, the lock count must be zero.

  It is interesting to notice that the pure capability $(2, 1)$ can be split as $(\overline{1, 0}) \oplus (1, 1)$, but not as $(1, 1) \oplus (\overline{1, 0})$. In the case of parallel application, it cannot be split as $(\overline{1, 1}) \oplus (1, 0)$ either.
- In the effect that is passed to the called function, regions (and their parents) need not be identical to those in the original effect. Rule *ES-C* only requires that $r \simeq r'$ (and $\pi \simeq \pi'$), which means that equality is checked after erasing the region tags that are introduced by the operational semantics (i.e., replacing $\imath@n$ with $\imath$). This requirement is only important for the proof of type safety.

The typing rules for references are standard. In Figure 2.6 we only show the rules for dereference (*T-D*) and reference allocation (*T-NR*). The former checks that region r is *accessible*. The latter only checks that the region r is *live*. Notice that effect typing is left-to-right, which is consistent with the left-to-right evaluation in the operational semantics. The output effect of the rightmost sub-expression of each construct is always (except for rule *T-NG*) used for checking the liveness and accessibility invariants. The rule for creating new regions (*T-NG*) checks that e_1 is a handle for some live region r'. Expression e_2 is type checked in an extended typing context (i.e., ρ and $x : \mathtt{rgn}(\rho)$ are appended to Δ and Γ respectively) and an extended input effect (i.e., a new effect is appended to the input effect such that the new region is live and accessible to this thread). The rule also checks that the type and the output effect of e_2 do not contain any occurrence of region variable ρ. This implies that ρ must be *consumed* by the end of the scope of e_2. The capability manipulation rule (*T-CP*) checks that e is a

handle of a live region r. It then modifies the capability count of r as dictated by function $[\![\eta]\!]$, which increases or decreases the region or the lock count of its argument, according to the value of η. The dynamic semantics ensures that an operational step is performed if the updated hierarchy preserves the invariant that protected regions are accessible to a single thread at instance of time. For instance, if the lock of region r is held by some other executing thread, the evaluation of $\mathtt{cap}_{\mathsf{lk}+}$ must be suspended until the lock can be obtained. On the other hand, the evaluation of $\mathtt{cap}_{\mathsf{rg}-}$ does not need to suspend but may not be able to physically deallocate a region, as it may be used by other threads.

2.6 Type Safety

In this section we discuss the fundamental theorems that prove type safety of our language.[5] The type safety formulation is based on proving the *preservation* and *progress* lemmata. Informally, a program written in our language is safe when for each thread of execution an evaluation step can be performed or that thread is waiting for a lock (*blocked*). As discussed in Section 2.4, a thread may become stuck when it accesses a region that is not live or accessible (these are obviously the interesting cases in our concurrent setting; of course a thread may become stuck when it performs a non well-typed operation). Deadlocked threads are not considered to be stuck.

Definition 1 (Thread Typing) Let T be a collection of threads. Let $R; M; \delta$ be a global typing context. For each thread $n : e$ in T, we take $\delta(n)$ to be the input effect that corresponds to the evaluation of expression e. The following rules define *well-typed* threads.

$$
\mathit{pops}(\sigma : e) \quad = \quad \begin{cases}
\mathit{pops}(\sigma : e_1) \wedge \mathit{pops}(\emptyset; \emptyset : e_2) & \text{if } e \equiv (e_1\ e_2)^{\xi} \wedge e_1 \neq v \\
\mathit{pops}(\sigma : e_2) \wedge \mathit{pops}(\emptyset; \emptyset : v) & \text{if } e \equiv (v\ e_2)^{\xi} \\
\mathit{pops}(\sigma : e_1) & \text{if } e \equiv (e_1)\,[v] \\
\mathit{pops}(\sigma : e_1) \wedge \mathit{pops}(\emptyset; \emptyset : e_2) & \text{if } e \equiv \mathtt{newrgn}^{r}\ \rho, x\ @\ e_1\ \mathtt{in}\ e_2 \\
\mathit{pops}(\sigma : e_1) & \text{if } e \equiv \mathtt{cap}^{r'}_{\eta}\ e_1 \\
\mathit{pops}(\sigma : e_1) \wedge \mathit{pops}(\emptyset; \emptyset : v) & \text{if } e \equiv \mathtt{new}\ v\ @\ e_1 \\
\mathit{pops}(\sigma : e_1) \wedge \mathit{pops}(\emptyset; \emptyset : e_2) & \text{if } e \equiv \mathtt{new}\ e_1\ @\ e_2 \wedge e_1 \neq v \\
\mathit{pops}(\sigma : e_1) & \text{if } e \equiv \mathtt{deref}\ e_1 \\
\mathit{pops}(\sigma : e_1) \wedge \mathit{pops}(\emptyset; \emptyset : e_2) & \text{if } e \equiv e_1 := e_2 \wedge e_1 \neq v \\
\mathit{pops}(\sigma : e_1) \wedge \mathit{pops}(\emptyset; \emptyset : v) & \text{if } e \equiv v := e_1 \\
\mathit{pops}(\sigma' : e_1) & \text{if } e \equiv \mathtt{pop}_{\gamma_r}\ e_1 \wedge \sigma = \emptyset; \gamma_r + \sigma' \\
\sigma \equiv \emptyset; \gamma \wedge \mathit{pops}(\emptyset; \emptyset : e') & \text{if } e \equiv \lambda x.\, e'\ \mathtt{as}\ \tau \\
\sigma \equiv \emptyset; \gamma \wedge \mathit{pops}(\emptyset; \emptyset : f) & \text{if } e \equiv \Lambda\rho.\, f \\
\sigma \equiv \emptyset; \gamma & \text{if } e \in \{\mathtt{loc}_{\ell}, \mathtt{rgn}_{\iota}, ()\}
\end{cases}
$$

$$
\frac{}{R; M; \emptyset \vdash \emptyset} \qquad \frac{\begin{array}{c} R; M; \emptyset; \emptyset \vdash e : \langle\rangle \,\&\, (\gamma;\emptyset) \\ R; M; \delta' \vdash T \qquad \forall (n' : e') \in T. n' \neq n \\ \delta = \delta', n \mapsto \sigma; \gamma \qquad \mathit{pops}(\sigma; \gamma : e) \end{array}}{R; M; \delta \vdash T, n : e}
$$

Definition 2 (Store Typing) A store S is *well-typed* with respect to $R; M$ (we denote this by $R; M \vdash S$) when the following conditions are met:

- the set of region names in S is equal to R,
- the set of locations in M is equal to the set of locations in S, and
- each stored value $S(\iota)(\ell)$ is closed and has type $M(\ell)$ with empty effects, i.e., $R; M; \emptyset; \emptyset \vdash S(\iota)(\ell) : M(\ell) \,\&\, (\emptyset;\emptyset)$.

[5] Full proofs and a full formalization of our language are given in the Appendix A.

$$\frac{R = \{\imath \mid (\imath \mapsto H) \in S\}}{R \vdash S}$$

$$\frac{\bigcup_{(\imath \mapsto H) \in S} \{\ell \mid (\ell \mapsto v) \in H\} = \{\ell \mid (\ell \mapsto (\tau, \jmath)) \in M\}}{M \vdash S}$$

$$\frac{\begin{array}{cc} M \vdash S & R \vdash S \\ \forall (\ell \mapsto (\tau, \imath)) \in M. R; M; \emptyset; \emptyset \vdash S(\imath)(\ell) : \tau \,\&\, (\emptyset;\emptyset) \end{array}}{R; M \vdash S}$$

Definition 3 (Configuration Typing) A configuration $\delta; S; T$ is *well-typed* with respect to $R; M$ (we denote this by $R; M \vdash \delta; S; T$) when the collection of threads T is well-typed with respect to $R; M$, the store S is well-typed with respect to $R; M$ and the global hierarchy δ is *consistent* and well-typed with respect to R.

$$\frac{\forall (n \mapsto \sigma) \in \delta. \forall \gamma \in \sigma. \forall (r^{\kappa} \triangleright \pi) \in \gamma. \; \bar{r} \in R \wedge (\pi = r' \Rightarrow \bar{r'} \in R)}{R \vdash \delta} \qquad \frac{\vdash \delta \quad R; M \vdash S \quad R \vdash \delta}{R; M \vdash \delta; S}$$

$$\frac{R; M; \delta \vdash T \quad R; M \vdash \delta; S}{R; M \vdash \delta; S; T}$$

Definition 4 (Not stuck) A configuration $S; T$ is *not stuck* when each thread in T can take one of the evaluation steps in Figure 2.3 (*E-S*, *E-T* or *E-SN*) or it is waiting for a lock held by some other thread. Deadlocked threads are not considered to be stuck.

$$\begin{array}{rl} \mathit{nolock}(\delta, n, e) & \equiv e = E[\mathtt{cap}^{r}_{+\mathsf{lk}}\ \mathtt{rgn_j}] \wedge \exists \delta'', \pi, \kappa, \kappa'.\ \delta = \delta'', n \mapsto \sigma; \gamma, r^{\kappa} \triangleright \pi \wedge \\ & \quad \kappa' = [\![\mathsf{lk}+]\!]\,(\kappa) \wedge \neg \vdash \delta'', n \mapsto \sigma; \gamma, r^{\kappa'} \triangleright \pi \end{array}$$

$$\frac{\forall (n : e) \in T.\ (\delta; S; T \rightsquigarrow \delta; S'; T' \wedge (n : e) \notin T') \vee \mathit{nolock}(\delta, n, e)}{\vdash \delta; S; T}$$

Given these definitions, we can now present the main results of this paper. The *progress* and *preservation* lemmata are first formalized at the *program* level, i.e., for all concurrently executed threads.

Lemma 2.1 (Progress — Program) Let $S; T$ be a closed well-typed configuration with $R; M \vdash \delta; S; T$, then $S; T$ is not stuck ($\vdash S; T$).

Lemma 2.2 (Preservation — Program) Let $\delta; S; T$ be a well-typed configuration with $R; M \vdash \delta; S; T$. If the operational semantics takes a step $\delta; S; T \rightsquigarrow \delta'; S'; T'$, then there exist $R' \supseteq R$ and $M' \supseteq M$ such that the resulting configuration is well-typed with $R'; M' \vdash \delta'; S'; T'$.

Type safety is based on proving the *preservation* and *progress* lemmata.[6] The type safety theorem can be formulated as follows:

Theorem 2.1 (Type safety) Let T_0 be the program's initial thread list, containing only the `main` thread with identifier 1. Let δ_0 and S_0 be the program's initial hierarchy and region list, such that there is only the heap region $\imath_H$, with capability (1, 0) for thread 1. If the operational semantics takes any number of steps $\delta_0; S_0; T_0 \rightsquigarrow^* \delta; S; T$, then the resulting configuration $\delta; S; T$ is not stuck.

[6] Full proofs and a full formalization of our language are given in Appendix A.

Chapter 3

Inference for region hierarchies with reader-writer locks

3.1 Overview

This chapter combines ideas and material which we presented in the previous chapter, but at the same time it significantly extends this work. In particular, the language presented in the previous chapter is quite complicated and has several drawbacks: it requires *explicit* effect annotations, thereby restricting aliasing and requiring manual effort from the programmer. For instance, when aliasing occurs it may be necessary to manually create new capabilities, which entails a run-time overhead and makes programming less intuitive. Additionally, explicit count annotations as well as information about the "parent-of" relation limit polymorphism and result in code duplication.

In this chapter, we lift all these limitations. The type and effect system we will develop requires annotations *only* at thread creation points (i.e., at uses of the `spawn` operator) and all the remaining annotations are automatically inferred and checked by the analysis. Moreover, there are no annotations regarding region aliasing state (i.e., aliased and non-aliased regions). We also extend the formal language with permissions for *read-only* accesses to hierarchies. Such a feature is useful and increases concurrency when threads share regions without modifying them. Of course, a region can alternate between read-only and read/write or "no-access" states during its lifetime in a safe manner. The type system ensures this.

In short, the main features of the type system we present and the contributions of this chapter are as follows:

Hierarchical regions and reader/writer locks. We develop a region-polymorphic lambda calculus, where regions are organized in a hierarchy and are protected with reader/writer locks. When a reader/writer lock of a region is acquired, then its subregions atomically inherit the same access rights. In addition, read/write-protected hierarchies can migrate or be shared with new threads.

Effect inference. Functions need not be annotated with explicit effects and the system permits a higher degree of polymorphism as there are no explicit capabilities.

Formalisms and soundness. We provide an operational semantics for the proposed language and a static semantics that guarantees absence of memory violations and freedom from data races. In addition, we state safety theorems and provide proofs for the soundness of the core language. The operational semantics presented in Section 3.4 has been greatly simplified compared to the operational semantics of Section 2.4. For instance, the thread-local stack of hierachies (σ) has been simplified to a single hierarchy. In contrast with the semantics of Section 2.4, types can be erased and have no run-time significance. The language run-time system and compiler, which are implemented after the formal semantics, also benefit from these simplifications as there is less run-time overhead at function/spawn call and return points.

The next section presents the main features of our language by example. We then provide a description of the formal language, its operational semantics and static semantics, followed by a section where the main theorems that guarantee the absence of memory violations and data races from well-typed programs are stated and proved. The chapter ends with some concluding remarks.

3.2 Language features through examples

In this section, we present the main features of our language through examples. We try to avoid technical issues as much as possible (the details will be made clear in Section 3.3); however, some characteristics of the type and effect system are revealed in this section and their presence is justified.

Example 3.1 (Simple region usage) Our first example shows a typical use of regions and is identical to Example 2.1.

```
region<ρ> h @ H;                    // Live ρ_H
  let z = rnew(h) 42;               // Live ρ_H, Live ρ
    ...
    *z = *z + 5 ;                   // Live ρ_H, Live ρ, R ρ, W ρ,
    ...
    rfree(h);                       // Live ρ_H, Live ρ, R ρ, W ρ, Cap {ρ ↦ (−1, 0, 0)}
    ...
```

The comments on the right-hand side of the example's code show the current *effect*. An effect is an order of actions that abstract program behavior at each program point. The effect at each program point is a prefix of the effect of the succeeding program points. Therefore, we employ a *causal* type and effect system to achieve absence of memory violations and data races.

Once region ρ is created, the *constraint* $\texttt{Live}\,\rho_H$ is appended to the effect; this means that the parent region of ρ, the heap region ρ_H, must be *live* at this program point. The reference allocation operation appends a new constraint to the effect, namely $\texttt{Live}\,\rho$, which requires that ρ is *live*, but not necessarily *accessible* (i.e., protected by some lock). Both the region and reference allocation constructs enable a higher degree of concurrency as threads need not acquire exclusive access to a region to allocate some data within it.

The next command reads the value of the cell pointed by z, adds the value five to it and stores the result back to the cell. The constraints generated for the read and write operation are $\texttt{R}\,\rho$ and $\texttt{W}\,\rho$ respectively. Constraint $\texttt{R}\,\rho$ requires that region ρ is at *least* read-protected by some lock, whereas the second constraint $\texttt{W}\,\rho$ requires that this thread has exclusive access to ρ. At each program point, region ρ is associated with three counters represented as a three-element vector, representing its reference count, write lock count and read lock count. The constraint generated for the `rfree` operator is $\texttt{Cap}\,\{\rho \mapsto (-1,0,0))\}$, which states that the static *counts* of ρ should be incremented by $(-1,0,0)$.

When the entire effect for ρ is gathered, then two actions are performed:

- The first action validates each constraint of the effect against the initial count $(1,1,0)$ (i.e., ρ is initially *live* and this thread has exclusive/direct access to ρ, that is, ρ is *thread-local*). Once effect validation is completed, the resulting count must be of the form $(0,n_1,n_2)$, otherwise ρ may have been deallocated implicitly by deallocating some of its ancestors. In the latter case, the constraint $\neg\texttt{Live}\,\rho_H$ is added, which requires that at least one of the ancestors of ρ starting from its parent ρ_H is not live.

  E.g., given the effect $\texttt{Live}\,\rho_H, \texttt{Live}\,\rho, \texttt{R}\,\rho, \texttt{W}\,\rho, \texttt{Cap}\,\{\rho \mapsto (-1,0,0)\}$ and the initial count of $(1,1,0)$ for ρ it is easy to see that all constraints regarding ρ are satisfiable and the resulting count of ρ is $(0,1,0)$, therefore the region has been deallocated.

- The second action simplifies the current effect by *removing* satisfiable constraints regarding ρ and by *translating* unsatisfiable constraints to constraints regarding the ancestors of ρ. For instance, if $\texttt{W}\,\rho$ is *unsatisfiable*, then it is translated to $\texttt{W}\,\rho_H$. This implies that if ρ is not write-protected, then at least one of its ancestors must be write-protected.

Once both actions are completed the resulting effect is $\texttt{Live}\,\rho_H$.

In the examples that follow, we simplify the presentation of effects by showing the region counts at each step (e.g., $\rho^{1,1,0}$) as opposed to showing the entire effect. The construction of richer region hierarchies and bulk region deallocation occur in the same manner as in Examples 2.2 and 2.3 respectively.

Example 3.2 (Region migration) The following code illustrates region migration and is identical to Example 2.4 except for the the `spawn` operator, which is also passed the tuple $(h, 1, 1, 0)$ denoting that thread output *steals* $(1, 1, 0)$ counts from the server thread. Therefore, the counts for ρ in the server thread once `spawn` is executed are $(0, 0, 0)$, which implies that ρ is no longer accessible. In the output thread, ρ is directly accessible and no further lock operations are required.

```
void server ()
   while (true)
      region<ρ> h @ H ;                                   // ρ^{1,1,0}
         let z = wait_data(h) ;
            process(z);                                   // ρ^{1,1,0}
            spawn (h, 1, 1, 0) output(h, z);
            ...                                           // ρ^{0,0,0}
            // ρ cannot be accessed here!
```

Example 3.3 (Region sharing) The following code illustrates region sharing and is identical to Example 2.4 except for the explicit annotation at the spawn operation and `wr_lock`/`wr_unlock`, which are equivalent to `lock`/`unlock` operations of the previous chapter. Thread `output` steals $(1, 1, 0)$, thus the counts remaining for ρ in the server thread are $(1, 0, 0)$. Therefore region ρ is now shared between the two threads.

```
void server ()
   while (true)
      region<ρ> h @ H ;                                   // ρ^{1,1,0}
         let z = wait_data(h) ;
            share(h);                                     // ρ^{2,1,0}
            spawn(h, 1, 1, 0) output(h, z);               // ρ^{1,0,0}
            wr_lock(h);                                   // ρ^{1,1,0}
            process(z);
            wr_unlock(h);                                 // ρ^{1,0,0}
            ...
```

Example 3.4 (Region and lock sharing) In the previous example, region ρ was *shared* between two threads, but each of them had to acquire exclusive access to ρ. This approach limits the degree of concurrency, especially in the case where functions `process` and `output` do not modify the contents of ρ. Here, we extend the previous example so that both threads have simultaneous access to ρ.

```
void server ()
   while (true)
      region<ρ> h @ H ;                                   // ρ^{1,1,0}
         let z = wait_data(h) ;
            share(h);                                     // ρ^{2,1,0}
            wr_unlock(h);                                 // ρ^{2,0,0}
            rd_lock(h);                                   // ρ^{2,0,1}
            rd_lock(h);                                   // ρ^{2,0,2}
            spawn(h, 1, 0, 1) output(h, z);               // ρ^{1,0,1}
            process(z);
            ...
```

Operator `share` increases the region count, whereas the following three instructions release the write access permission to ρ and acquire two *read* access permissions to ρ. Consequently, the original count for ρ, namely $(1,1,0)$ is transformed to $(2,0,2)$. Thread `output` steals $(1,0,1)$, thus the counts remaining for ρ in the server thread are $(1,0,1)$. Region ρ is now shared between the two threads and both threads can concurrently *read* (only) the contents of ρ.

Example 3.5 (Hierarchical locking) The following code illustrates region sharing and is identical to Example 2.6 except for `wr_lock`/`wr_unlock`, which are equivalent to `lock`/`unlock` operations of the previous chapter.

```
wr_lock(h);                                   // the handle of a common ancestor of ρ1 and ρ2
let obj = hash_remove<ρ1>(tbl1, key);
    hash_insert<ρ2>(tbl2, key, obj);
wr_unlock(h);
```

Example 3.6 (Reentrant locks) Similarly to Example 2.8, this example shows that region aliasing introduces the need for reentrant locks in our language.

```
// ρ1 and ρ2 are unlocked
void swap<ρ1, ρ2>(region<ρ1> h1, region<ρ2> h2, int * ρ1 x, int * ρ2 y)
    wr_lock(h1);
    let z = *x ;                                  // OK: ρ1 is locked
        wr_lock(h2);
        *x = *y;                                  // OK: ρ1 and ρ2 are locked
        wr_unlock(h1);
        *y = z;                                   // OK: ρ2 is locked
        wr_unlock(h2);                            // all locks can be released

...
swap<ρ, ρ>(h, h, a, b);
```

Example 3.7 (Unsound sharing) In the code that follows, region ρ is shared with a new thread that accesses the contents of ρ, namely reference z. Region ρ is accessible in both the main and the new thread (the one executing function `f`).

```
region<ρ> h @ H ;                             // ρ^{1,1,0}
    let z = rnew(h) 42;                       // ρ^{1,1,0}
        share(h);                             // ρ^{2,1,0}
        wr_lock(h);                           // ρ^{2,2,0}
        spawn (h, 1, 1, 0) f(h, z);           // ρ^{1,1,0}
        *z = 17;                              // possible data race!
```

This program code will be rejected by our type system as `spawn` must consume either none or all write locks of ρ.

Example 3.8 (Unsound aliasing) In the previous example the data race bug was exposed in the main thread. However, data races may be introduced in nested function calls as a result of region aliasing. Consider function `bar`, which accepts a region handle h to region ρ_1, and two integer references (x and y) in regions ρ_1 and ρ_2, which are both locked.

```
void bar<ρ1, ρ2>(region<ρ1> h, int *ρ1 x, int *ρ2 y)     // ρ1^{1,1,0}, ρ2^{1,1,0}
    spawn (h, 1, 1, 0) f(x);                             // ρ2^{1,1,0}
    *y = 17;                                             // ρ2^{1,1,0}
```

The region counts calculated by the type system are shown on the right-hand side. As shown in the code below, if `bar` is invoked with the same reference z, a data race may occur since both main and the new thread will have access to ρ.

```
region<ρ> h @ H;                          // ρ^{1,1,0}
  let z = rnew(h) 42;                     // ρ^{1,1,0}
    share(h);                             // ρ^{2,1,0}
    wr_lock(h);                           // ρ^{2,2,0}
    bar<ρ, ρ>(h, z, z);                   // possible data race!
```

As mentioned in Example 3.1, the type system gathers the effect corresponding to the scope of a new region construct and then performs effect translation/validation. When type checking a function call, the formal regions of the function effect are substituted for the actual regions that instantiate the function. However, no checking is performed at the function call site. Therefore, region substitution and the deferred effect validation reduce invalid programs resulting from region aliasing to invalid programs resulting from invalid lock usage. As in the previous example, the type system will reject the above program as some but not all locks of ρ are passed to the new thread.

Example 3.9 (Negative constraints) As explained in Example 3.7, programs that grant region access to more than thread are susceptible to data races and are rejected by our type system. A region can also be protected by its own lock or the locks of its ancestors. Here we illustrate a program that is susceptible to data races as a result of hierarchical locking.

```
region<ρ1> h1 @ H;                                  // ρ1^{1,1,0}
  region<ρ2> h2 @ h2;                               // ρ1^{1,1,0}, ρ2^{1,1,0}
    let z = rnew(h2) 42;                            // ρ1^{1,1,0}, ρ2^{1,1,0}
      share(h1);                                    // ρ1^{2,1,0}, ρ2^{1,1,0}
      share(h2);                                    // ρ1^{2,1,0}, ρ2^{2,1,0}
      spawn {(h1, 1, 1, 0) (h2, 1, 0, 0)} f(x);     // ρ1^{1,0,0}, ρ2^{1,1,0}
      *z = 17;                                      // possible data race!
```

The main thread allocates a new region ρ_1 in the heap region and another region ρ_2 in region ρ_1. It then shares ρ_1 and ρ_2, by invoking the operation `share` on ρ_1 and ρ_2 respectively, and spawns a new thread `f` that has the lock for ρ_1 but not ρ_2. The main thread retains the lock for region ρ_2.

Due to the hierarchical relation of ρ_1 and ρ_2, the latter region is accessible in the new thread. Therefore, this program is susceptible to data races. The type system rejects such programs by introducing *negative constraints*. In particular, $\neg\texttt{RW}\,\rho_1$ is added in the effect of the new thread as its child region ρ_2 is accessible in the main thread. This constraint implies that neither ρ_1 nor its ancestors must be initially accessible in the new thread. Of course, this constraint is unsatisfiable and the program is rejected.

3.3 Formal language

The language syntax is illustrated in Figure 3.1. The core language includes variables (x), constants (`true`, `false` and $()$ — the unit value), functions (f), function application ($e_1\ e_2$), and conditional expressions (`if` e `then` e_1 `else` e_2). Functions can be monomorphic ($\lambda x.\, e$), region polymorphic ($\Lambda\rho.\, f$) where ρ is a region variable, and recursive (`fix` $x.\, f$). The application of region polymorphic functions is explicit ($e\,[r]$) where r is a metavariable ranging over region variables ρ and region constants $\imath$. We assume the existence of a special region constant denoted by $\bot$, which corresponds to the whole memory that is available to the program. This region cannot be manipulated (e.g., locked, released, etc.) by the program and only serves as the root of the region hierarchy.

The construct `newrgn` ρ, x @ e_1 `in` e_2 allocates a fresh region ρ, residing inside the region indicated by handle e_1, and binds x to the handle of ρ. Both ρ and x are lexically bound to the scope of e_2 and

Expression	e	$::= x \mid f \mid () \mid \texttt{true} \mid \texttt{false} \mid e\ e \mid e[r] \mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e$
		$\mid \texttt{newrgn}\ \rho, x\ @\ e\ \texttt{in}\ e \mid \texttt{new}\ e\ @\ e \mid e := e \mid \texttt{deref}\ e \mid \texttt{cap}_{\eta}\ e$
		$\mid \texttt{spawn}_{\xi}\ e \mid \texttt{loc}_{\ell} \mid \texttt{rgn}_{\iota}$
Function	f	$::= \lambda x.\, e \mid \Lambda\rho.\, f \mid \texttt{fix}\ x.\, f$
Value	v	$::= f \mid () \mid \texttt{true} \mid \texttt{false} \mid \texttt{loc}_{\ell} \mid \texttt{rgn}_{\iota}$
Region	r	$::= \rho \mid \iota$
Count vector	η	$::= (n, n, n)$
Spawn effect	ξ	$::= \emptyset \mid \xi, r \mapsto \eta$

Figure 3.1: Syntax.

the new region must be explicitly released within e_2. Each region is associated with a count vector η, consisting of three natural numbers (n_1, n_2, n_3):

- the reference count (n_1), which tells us whether the region is *live* in the current thread;
- the write lock count (n_2), which provides exclusive access to the region and tells us whether the current thread can assign values to locations in it; and
- the read lock count (n_3), which provides non-exclusive access to the region and tells us whether the current thread can read values from locations in it.

We use counts (natural numbers) instead of boolean values to support re-entrant locks and region aliasing, as explained in Section 3.2. Notice that the write lock (n_2) takes priority over the read lock (n_3): if $n_2 > 0$ then a thread has exclusive access to a region and is capable of writing and reading, otherwise if $n_3 > 0$ then a thread has non-exclusive access to a region and is only capable of reading, otherwise (if $n_2 = n_3 = 0$) a thread has no access to a region, i.e., it cannot write nor read. When first allocated, a region starts with $(1, 1, 0)$, meaning that it is live and exclusively locked by the current thread, so that it can be accessed directly with no additional overhead. This is our equivalent of a thread-local region.

The constructs for manipulating references are standard. A new memory cell is allocated by $\texttt{new}\ e_1\ @\ e_2$, where e_1 is an initializer expression for the new cell's contents and e_2 is a handle indicating the region in which the new cell will be allocated; the result is a reference to the newly allocated cell. Standard assignment ($e_1 := e_2$) and dereference ($\texttt{deref}\ e$) complete the picture. As we explained, capability counts determine the validity of operations on regions and references. All memory-related operations require that the involved regions are live. Assignment can be performed only when the corresponding region is live and write-protected, whereas dereference can be performed when the region is live and *at least* read-protected.

The construct $\texttt{cap}_{\eta}\ e$ formalizes the concept of incrementing or decrementing the counts for a region (i.e., acquiring or releasing capabilities). It is the formal counterpart of the constructs with the more descriptive names `share`, `rw_lock`, etc., that were used in Section 3.2. It requires a region handle e and a three-element vector η that denotes the relative counts to be added to the current counts of that region. In this vector η, a count is allowed to be negative, meaning that the current count is to be decreased. Incrementing a lock count (n_2 or n_3) from zero to a positive value amounts to acquiring a region lock and may have to block the current thread, if the lock is held by another thread. On the other hand, decrementing lock counts never blocks the current thread. Decrementing a region count (n_1) from a positive value to zero amounts to releasing the region; it may cause the region's contents (including any subregions residing in it) to be deallocated.

New threads can be created with the $\texttt{spawn}_{\xi}\ e$ construct, which starts evaluating expression e in parallel with the remaining computation in the current thread. It is annotated with a *spawn effect* ξ, which contains the list of regions that will be passed to the new thread and the exact counts that will

$$
\begin{array}{lrcl}
\textbf{Hierarchy} & \theta & ::= & \emptyset \mid \theta, \imath \mapsto (\eta, \imath) \\
\textbf{Heap} & H & ::= & \emptyset \mid H, \ell \mapsto v \\
\textbf{Store} & S & ::= & \emptyset \mid S, \imath \mapsto H \\
\textbf{Threads} & T & ::= & \emptyset \mid T, \langle \theta; e \rangle \\
\textbf{Configuration} & C & ::= & S; T \\
\textbf{Stack} & E & ::= & \Box \mid E[F] \\
\textbf{Frame} & F & ::= & \Box\ e \mid v\ \Box \mid \Box\,[r] \mid \texttt{if}\ \Box\ \texttt{then}\ e\ \texttt{else}\ e \mid \texttt{newrgn}\ \rho, x\ @\ \Box\ \texttt{in}\ e \\
 & & \mid & \texttt{new}\ \Box\ @\ e \mid \texttt{new}\ v\ @\ \Box \mid \Box := e \mid v := \Box \mid \texttt{deref}\ \Box \mid \texttt{cap}_{\eta}\ \Box
\end{array}
$$

Figure 3.2: Auxiliary syntax for operational semantics.

be consumed. To sum up, the counts of a region can be altered either by using the `cap` construct, or by transferring some to a newly spawned thread.

The two remaining constructs $\texttt{rgn}_{\imath}$ and $\texttt{loc}_{\ell}$ correspond to explicit region and location handles (the metavariables $\imath$ and ℓ range over region and location constants, respectively). They are not considered part of the source language, except for the special case $\texttt{rgn}_{\perp}$ which is the handle of the total memory and can be used to create new regions therein. With this exception, both constructs must not be used in the source program: they are only introduced during program evaluation, as discussed further in Section 3.4.

3.4 Operational semantics

We define a *small-step* operational semantics for our language in Figures 3.2 and 3.3. The thread evaluation relation $C \rightsquigarrow C'$ transforms configurations. A configuration C consists of an abstract store S and a thread list T.[1] A store S maps region identifiers ($\imath$) to heaps (H), which in turn map memory locations to values. Each thread in T is a pair containing a thread-local region hierarchy θ and an expression e to be evaluated. The hierarchy θ is a map indexed by region identifiers; for each region, this map gives us its count vector η (as known by some specific thread) and its parent region. A frame F is an expression with a *hole*, represented as $\Box$. The hole indicates the position where the next reduction step can take place. Our notion of *thread evaluation context* is defined as a stack of nested frames E, imposing a call-by-value evaluation strategy to our language. Subexpressions are evaluated in a left-to-right order. We assume that concurrent reduction events can be totally ordered [Lamp79]. Our evaluation rules are *non-deterministic*: the order in which different threads evaluate their expressions is not specified.

Threads that have been reduced to unit values are removed from the active thread list, as long as they have released all regions used by them (rule *E-T*). This is established by the premise $\mathsf{live}(\theta) = \emptyset$. Function live returns the set of all regions in θ that are live, i.e., their reference count as well as those of all their ancestors are positive. The formal definition of function live is given in Figure 3.4, together with the definitions of other auxiliary functions and predicates that are used in the operational semantics.

When a `spawn` redex is detected within a thread evaluation context, a new thread is created (rule *E-SP*). The redex is replaced with a unit value in the currently executed thread and a new thread is created to evaluate the given expression. The premise $\mathsf{merge}(\xi) \vdash \theta = \theta' \oplus \theta''$ splits the hierarchy of the current thread θ into θ' and θ'', corresponding to the hierarchy that will remain in the current thread and the hierarchy that will be passed to the new thread, respectively. The annotation ξ drives the splitting process by defining the counts that should be passed to the new thread. Function merge

[1] The order of elements in comma-separated lists, e.g. in a store S or in a list of threads T, is unimportant; we consider all list permutations as equivalent.

$$\frac{\mathsf{live}(\theta) = \emptyset}{S; T, \langle \theta; () \rangle \rightsquigarrow S; T} \quad \textit{(E-T)}$$

$$\frac{\mathsf{merge}(\xi) \vdash \theta = \theta' \oplus \theta'' \qquad \mathsf{dom}(\theta'') \subseteq \mathsf{live}(\theta)}{S; T, \langle \theta; E[\mathtt{spawn}_{\xi}\ e] \rangle \rightsquigarrow S; T, \langle \theta'; E[()] \rangle, \langle \theta''; \Box[e] \rangle} \quad \textit{(E-SP)}$$

$$\frac{f \equiv \lambda x.\, e}{S; T, \langle \theta; E[f\ \ v] \rangle \rightsquigarrow S; T, \langle \theta; E[e[v/x]] \rangle} \quad \textit{(E-A)}$$

$$\frac{f \equiv \Lambda \rho.\, f'}{S; T, \langle \theta; E[f\ [\imath]] \rangle \rightsquigarrow S; T, \langle \theta; E[f'[\imath/\rho]] \rangle} \quad \textit{(E-RP)}$$

$$\frac{f \equiv \mathtt{fix}\ x.\, f'}{S; T, \langle \theta; E[f\ \ v] \rangle \rightsquigarrow S; T, \langle \theta; E[f'[f/x]\ \ v] \rangle} \quad \textit{(E-FX)}$$

$$\frac{}{S; T, \langle \theta; E[\mathtt{if\ true\ then}\ e_1\ \mathtt{else}\ e_2] \rangle \rightsquigarrow S; T, \langle \theta; E[e_1] \rangle} \quad \textit{(E-IT)}$$

$$\frac{}{S; T, \langle \theta; E[\mathtt{if\ false\ then}\ e_1\ \mathtt{else}\ e_2] \rangle \rightsquigarrow S; T, \langle \theta; E[e_2] \rangle} \quad \textit{(E-IF)}$$

$$\frac{\jmath \in \mathsf{live}(\theta) \cup \{\bot\} \qquad \text{fresh}\ \imath \qquad \theta' = \theta, \imath \mapsto ((1,1,0), \jmath)}{S; T, \langle \theta; E[\mathtt{newrgn}\ \rho, x\ @\ \mathtt{rgn}_{\jmath}\ \mathtt{in}\ e] \rangle \rightsquigarrow S, \imath \mapsto \emptyset; T, \langle \theta'; E[e[\imath/\rho][\mathtt{rgn}_{\imath}/x]] \rangle} \quad \textit{(E-NR)}$$

$$\frac{\imath \in \mathsf{live}(\theta) \qquad \text{fresh}\ \ell}{S; T, \langle \theta; E[\mathtt{new}\ v\ @\ \mathtt{rgn}_{\imath}] \rangle \rightsquigarrow S[\imath \mapsto S(\imath), \ell \mapsto v]; T, \langle \theta; E[\mathtt{loc}_{\ell}] \rangle} \quad \textit{(E-NL)}$$

$$\frac{\ell \mapsto v' \in S(\imath) \qquad \imath \in \mathsf{wlocked}(\theta) \qquad \imath \notin \mathsf{rwlocked}(T)}{S; T, \langle \theta; E[\mathtt{loc}_{\ell} := v] \rangle \rightsquigarrow S[\imath \mapsto S(\imath)[\ell \mapsto v]]; T, \langle \theta; E[()] \rangle} \quad \textit{(E-AS)}$$

$$\frac{\ell \mapsto v \in S(\imath) \qquad \imath \in \mathsf{rwlocked}(\theta) \qquad \imath \notin \mathsf{wlocked}(T)}{S; T, \langle \theta; E[\mathtt{deref}\ \mathtt{loc}_{\ell}] \rangle \rightsquigarrow S; T, \langle \theta; E[v] \rangle} \quad \textit{(E-D)}$$

$$\frac{\imath \in \mathsf{live}(\theta) \qquad \theta' = \theta, \imath \mapsto (\eta + \eta', \jmath) \qquad \mathsf{mutex}(\{\theta'\} \cup \{\theta'' \mid \langle \theta''; e' \rangle \in T\})}{S; T, \langle \theta, \imath \mapsto (\eta, \jmath); E[\mathtt{cap}_{\eta'}\ \mathtt{rgn}_{\imath}] \rangle \rightsquigarrow S; T, \langle \theta'; E[()] \rangle} \quad \textit{(E-CP)}$$

Figure 3.3: Evaluation relation $C \rightsquigarrow C'$.

takes care of region aliasing, by merging the counts of entries in ξ that correspond to the same region. On the other hand, the premise $\mathsf{dom}(\theta'') \subseteq \mathsf{live}(\theta)$ ensures that all regions passed to the new thread are live.

The rules for evaluating the application of monomorphic functions (*E-A*), polymorphic functions (*E-RP*) and recursive functions (*E-FX*) are standard, as well as the rules for evaluating conditionals (*E-IT* and *E-IF*).

Rule *E-NR* requires that the parent region $\jmath$ is live or has the value $\bot$. The rule adds a fresh and empty region $\imath$ to the store S and associates it with the pair, $((1,1,0), \jmath)$ in the local hierarchy θ. Therefore, the new region is initially *live* and the current thread has exclusive access to it. Rule *E-NL* requires that region $\imath$ is live in θ and updates the heap of $\imath$ with a fresh location ℓ mapping to value v. Notice, that $\imath$ need not be protected.

Rule *E-AS* requires that location ℓ exists in some region $\imath$ of the global store S and requires that $\imath$ is exclusively owned by the current thread. This is established by $\imath \in \mathsf{wlocked}(\theta)$ and $\imath \notin \mathsf{rwlocked}(T)$. Function wlocked returns the set of live regions that can be write-accessed by the current thread, whereas function rwlocked returns the set of live regions that can be read or written by the remaining threads in T. If the current thread has no write-access to $\imath$ or any other thread has read or write access to $\imath$, then the evaluation will get stuck.

$$
\begin{array}{lcll}
\mathsf{merge}(\emptyset) & = & \emptyset & \\
\mathsf{merge}(\xi, r \mapsto \eta) & = & \mathsf{merge}(\xi), r \mapsto \eta & \text{if } r \notin \{r' \mid r' \mapsto \eta' \in \xi\} \\
\mathsf{merge}(\xi, r \mapsto \eta, r \mapsto \eta') & = & \mathsf{merge}(\xi, r \mapsto (\eta + \eta')) & \\
\mathsf{ok}(n_1, n_2, n_3) & = & n_1 \geq 0 \wedge n_2 \geq 0 \wedge n_3 \geq 0 & \\
(c_1, w_1, z_1) \oplus (c_2, w_2, z_2) & = & (c_1 + c_2, w_1 + w_2, z_1 + z_2) & \text{if } \mathsf{ok}(c_1, w_1, z_1) \wedge \mathsf{ok}(c_2, w_2, z_2) \wedge \\
 & & & \quad (w_1 = 0 \vee w_2 = 0) \wedge (c_2 > 0) \wedge \\
 & & & \quad (c_1 = 0 \implies w_1 = z_1 = 0) \wedge \\
 & & & \quad (w_1 > 0 \implies z_2 = 0) \wedge \\
 & & & \quad (w_2 > 0 \implies z_1 = 0)
\end{array}
$$

$$
\begin{array}{lcl}
\mathsf{ancestors}(\theta, \bot) & = & \emptyset \\
\mathsf{ancestors}(\theta, \imath) & = & \{\imath\} \cup \mathsf{ancestors}(\theta', \jmath) \quad \text{if } \theta = \theta', \imath \mapsto (\eta, \jmath) \\
\mathsf{ok}(\theta) & = & \forall \imath \mapsto (\eta, \jmath) \in \theta.\ \mathsf{ok}(\eta) \wedge \mathsf{ancestors}(\theta, \imath) \text{ defined} \\
\mathsf{live}(\theta) & = & \{\imath \mid \forall \jmath \in \mathsf{ancestors}(\theta, \imath).\ \exists \jmath \mapsto (\eta, \jmath') \in \theta.\ \mathsf{ok}(\eta - (1,0,0))\} \\
\mathsf{wlocked}(\theta) & = & \{\imath \mid \imath \in \mathsf{live}(\theta) \wedge \exists \jmath \mapsto (\eta, \jmath') \in \theta.\ \jmath \in \mathsf{ancestors}(\theta, \imath) \wedge \mathsf{ok}(\eta - (0,1,0))\} \\
\mathsf{rlocked}(\theta) & = & \{\imath \mid \imath \in \mathsf{live}(\theta) \wedge \exists \jmath \mapsto (\eta, \jmath') \in \theta.\ \jmath \in \mathsf{ancestors}(\theta, \imath) \wedge \mathsf{ok}(\eta - (0,0,1))\} \\
\mathsf{rwlocked}(\theta) & = & \mathsf{rlocked}(\theta) \cup \mathsf{wlocked}(\theta) \\
\mathsf{wlocked}(T) & = & \{\imath \mid \exists \langle \theta, e \rangle \in T.\ \imath \in \mathsf{wlocked}(\theta)\} \\
\mathsf{rwlocked}(T) & = & \{\imath \mid \exists \langle \theta, e \rangle \in T.\ \imath \in \mathsf{rwlocked}(\theta)\} \\
\mathsf{mutex}(\{\theta_1, \ldots, \theta_n\}) & = & \forall \imath \neq \jmath.\ \mathsf{rwlocked}(\theta_\imath) \cap \mathsf{wlocked}(\theta_\jmath) = \mathsf{wlocked}(\theta_\imath) \cap \mathsf{rwlocked}(\theta_\jmath) = \emptyset \\
\mathsf{hierarchy\_ok}(\theta_1; \theta_2) & = & \forall \imath \mapsto (\eta, \jmath) \in \theta_1.\ \exists\, \imath \mapsto (\eta', \jmath') \in \theta_2.\ (\jmath = \jmath' \vee (\jmath = \bot \wedge \jmath' \notin \mathsf{dom}(\theta_1)))
\end{array}
$$

$$
\frac{}{\emptyset \vdash \theta = \theta \oplus \emptyset}
\qquad
\frac{\begin{array}{cc} \eta = \eta_1 \oplus \eta_2 & \xi \vdash \theta = \theta_1 \oplus \theta_2 \\ \forall \imath' \in \mathsf{dom}(\xi).\ \imath \notin \mathsf{ancestors}(\theta, \imath') & \jmath' = \text{if } \jmath \in \mathsf{dom}(\xi) \text{ then } \jmath \text{ else } \bot \end{array}}{\xi, \imath \mapsto \eta_2 \vdash \theta, \imath \mapsto (\eta, \jmath) = \theta_1, \imath \mapsto (\eta_1, \jmath) \oplus \theta_2, \imath \mapsto (\eta_2, \jmath')}
$$

$$
\frac{}{\emptyset - \emptyset = \emptyset}
\qquad
\frac{\eta_1 \geq \eta_2 \qquad \theta_1 - \theta_2 = \theta'}{\theta_1, r \mapsto \eta_1 - \theta_2 \mapsto \eta_2 = \theta', r \mapsto \eta_1 - \eta_2}
$$

Figure 3.4: Auxiliary functions and predicates.

In contrast to the rule for assignment, rule *E-D* is more permissive as it admits simultaneous *read-only* access to region $\imath$ by more than one threads. It requires that the current thread has (possibly) non-exclusive access to region $\imath$ and that no other thread in T has exclusive access to it. Evaluation will get stuck if any of these two are violated.

Consequently, memory cells can be accessed only when there is an appropriate level of protection by the current thread and other threads do not violate mutual exclusion for write-locks. Our type safety results (see Section 3.7) guarantee that no thread can get stuck by violating mutual exclusion.

Rule *E-CP* requires that region $\imath$ is live in θ and adds the relative count vector η' to the current vector η for $\imath$. It is possible that the resulting hierarchy θ' does not preserve mutual exclusion with respect to the other hierarchies in T. For instance, suppose that $\imath$ is exclusively locked by some other thread in T and that the current thread tries to acquire a lock by incrementing its lock count. In this case, the current thread should block until the lock is released by the other thread. This is established by the premise $\mathsf{mutex}(\{\theta'\} \cup \{\theta'' \mid \langle \theta''; e' \rangle \in T\})$, which requires that the mutual exclusion invariant holds between all threads once θ is modified to θ'.

Type	τ	$::=$	$\texttt{unit} \mid \texttt{bool} \mid \tau \xrightarrow{\gamma} \tau \mid \forall\rho.\,\tau \mid \texttt{Ref}(\tau, r) \mid \texttt{Rgn}(r)$
Constraint	δ	$::=$	$\texttt{R} \mid \texttt{W} \mid \neg\texttt{RW} \mid \neg\texttt{W} \mid \texttt{Live} \mid \neg\texttt{Live}$
Event	ζ	$::=$	$\texttt{Cap}\,\xi \mid \delta\, r \mid \texttt{Spawn}\,\xi\,\gamma \mid \texttt{Join}\,\gamma\,\gamma$
Effect	γ	$::=$	$\emptyset \mid \zeta :: \gamma$
Type context	$\square$	$::=$	$\emptyset \mid \Gamma, x : \tau$
Region context	$\square$	$::=$	$\emptyset \mid \Delta, \rho$
Heap context	M	$::=$	$\emptyset \mid M, \ell \mapsto (\tau, \imath)$
Store context	R	$::=$	$\emptyset \mid R, \imath$

Figure 3.5: Syntax for types, effects and contexts.

3.5 Static semantics

We now present the type and effect system that we use to enforce memory safety and race freedom. The syntax of types and effects is given in Figure 3.5. Basic types consist of the unit type and the type of boolean values. Monomorphic function types ($\tau \xrightarrow{\gamma} \tau$) are annotated with the function's effect γ; effects are the most important aspect of this system and will be explained in detail in the rest of this section. Although this is not apparent in Figure 3.5, polymorphic types ($\forall\rho.\,\tau$) are restricted to functions. Region handle types $\texttt{Rgn}(r)$ and reference types $\texttt{Ref}(\tau, r)$ are associated with a type-level region r; the former is essentially a singleton type, the latter corresponds to the type of memory cells that reside in region r and whose contents have type τ.

Effects (γ) are used to statically track region state and accesses. They are ordered sequences of events (ζ) which correspond to behaviors that arise when evaluating expressions. Although in Figure 3.5 operator :: denotes the "cons" operation on effects, prepending an event to an effect, we will often abuse notation and use :: as an associative operator for appending effects, with $\emptyset$ as a zero element. We will also silently treat events as effects of length one.

There are four kinds of events. An event of the form $\texttt{Cap}\,\xi$ roughly corresponds to the evaluation of one or more $\texttt{cap}_\eta\ r$ expressions: it means that for each $r \mapsto \eta$ in ξ, the count vector of region r is incremented by η. On the other hand, an event of the form $\delta\, r$ is meant to impose a constraint δ on region r. There are several types of constraints, requiring that a region is readable (R), writable (W), not readable nor writable ($\neg$RW), not writable ($\neg$W), live (Live), and not live ($\neg$Live). Events of the form $\texttt{Cap}\,\xi$ and $\delta\, r$ are commonly called *atomic events*. On the other hand, we have two kinds of composite events. A spawn event $\texttt{Spawn}\,\xi\,\gamma$ means that a new thread is spawned, where ξ denotes the set of regions and corresponding count vectors that are passed to the new thread, and γ is the effect of the new thread's body. A conditional event $\texttt{Join}\,\gamma_1\,\gamma_2$ means that control flow branches and the effects of the two alternatives are γ_1 and γ_2.[2]

The typing relation, is denoted by $R; M; \Delta; \Gamma \vdash e : \tau \,\&\, \gamma$ which means that expression e has type τ and produces effect γ. It is defined in Figure 3.6 and uses four typing contexts (defined in Figure 3.5): a set of region constants (R), a mapping of locations to types and regions (M), a set of region variables (Δ), and a mapping of term variables to types (Γ). For brevity, we have omitted from our presentation in this section the definitions of several judgements referring either to the well-formedness of types, regions, etc., with respect to the typing contexts, or to the well formedness of the typing contexts themselves.

The typing rules *T-V*, *T-U*, *T-TR*, *T-FL*, *T-R*, *T-L*, *T-F*, *T-A*, *T-RF*, and *T-RP*, are more or less standard. Notice that the effects produced by values are always empty. The typing rule for function abstraction (*T-F*) annotates the function's type with the effect of the function's body. Moreover, the typing rule for function application (*T-A*) simply concatenates the effects of e_1 and e_2 (γ_1 and γ_2, re-

[2] The "append" operator distributes over "join". Semantically, the effects $\gamma :: \texttt{Join}\,\gamma_1\,\gamma_2 :: \gamma'$ and $\texttt{Join}\,(\gamma :: \gamma_1 :: \gamma')\,(\gamma :: \gamma_2 :: \gamma')$ are equivalent.

$$\frac{x : \tau \in \Gamma \quad \vdash R; M; \Delta; \Gamma}{R; M; \Delta; \Gamma \vdash x : \tau \,\&\, \emptyset} \textit{ (T-V)} \qquad \frac{\vdash R; M; \Delta; \Gamma}{R; M; \Delta; \Gamma \vdash () : \mathtt{unit} \,\&\, \emptyset} \textit{ (T-U)}$$

$$\frac{\vdash R; M; \Delta; \Gamma}{R; M; \Delta; \Gamma \vdash \mathtt{true} : \mathtt{bool} \,\&\, \emptyset} \textit{ (T-TR)} \qquad \frac{\vdash R; M; \Delta; \Gamma}{R; M; \Delta; \Gamma \vdash \mathtt{false} : \mathtt{bool} \,\&\, \emptyset} \textit{ (T-FL)}$$

$$\frac{\iota \in R \cup \{\bot\} \quad \vdash R; M; \Delta; \Gamma}{R; M; \Delta; \Gamma \vdash \mathtt{rgn}_\iota : \mathtt{Rgn}(\iota) \,\&\, \emptyset} \textit{ (T-R)} \qquad \frac{\ell \mapsto (\tau, \iota) \in M \quad \vdash R; M; \Delta; \Gamma}{R; M; \Delta; \Gamma \vdash \mathtt{loc}_\ell : \mathtt{Ref}(\tau, \iota) \,\&\, \emptyset} \textit{ (T-L)}$$

$$\frac{R; M; \Delta; \Gamma, x : \tau_1 \vdash e : \tau_2 \,\&\, \gamma}{R; M; \Delta; \Gamma \vdash \lambda x.\, e : \tau_1 \xrightarrow{\gamma} \tau_2 \,\&\, \emptyset} \textit{ (T-F)}$$

$$\frac{R; M; \Delta; \Gamma \vdash e_1 : \tau_1 \xrightarrow{\gamma} \tau_2 \,\&\, \gamma_1 \qquad R; M; \Delta; \Gamma \vdash e_2 : \tau_1 \,\&\, \gamma_2}{R; M; \Delta; \Gamma \vdash e_1 \;\; e_2 : \tau_2 \,\&\, \gamma_1 :: \gamma_2 :: \gamma} \textit{ (T-A)}$$

$$\frac{R; \Delta \vdash \Gamma \qquad R; M; \Delta, \rho; \Gamma \vdash f : \tau \,\&\, \emptyset}{R; M; \Delta; \Gamma \vdash \Lambda\rho.\, f : \forall \rho.\, \tau \,\&\, \emptyset} \textit{ (T-RF)}$$

$$\frac{R; M; \Delta; \Gamma \vdash e : \forall \rho.\, \tau \,\&\, \gamma \qquad R; \Delta \vdash r \qquad r \neq \bot}{R; M; \Delta; \Gamma \vdash e\,[r] : \tau[r/\rho] \,\&\, \gamma} \textit{ (T-RP)}$$

$$\frac{R; M; \Delta; \Gamma \vdash e : \mathtt{bool} \,\&\, \gamma \qquad R; M; \Delta; \Gamma \vdash e_1 : \tau \,\&\, \gamma_1 \qquad R; M; \Delta; \Gamma \vdash e_2 : \tau \,\&\, \gamma_2}{R; M; \Delta; \Gamma \vdash \mathtt{if}\ e\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 : \tau \,\&\, \gamma :: \mathtt{Join}\,\gamma_1\,\gamma_2} \textit{ (T-IF)}$$

$$\frac{R; \Delta \vdash \xi \qquad R; M; \Delta; \Gamma \vdash e : \mathtt{unit} \,\&\, \gamma \qquad \mathsf{dom}(\xi) = \mathsf{dom}(\gamma)}{R; M; \Delta; \Gamma \vdash \mathtt{spawn}_\xi\ e : \mathtt{unit} \,\&\, \mathtt{Spawn}\,\xi\,\gamma} \textit{ (T-SP)}$$

$$\frac{\begin{array}{c}\gamma_L = \{\mathtt{Live}\,r \mid r \in \mathsf{dom}(\phi(\emptyset))\} \qquad \gamma_s = \mathsf{summary}(\phi(\gamma_L)) \\ R; M; \Delta; \Gamma, x : \tau_1 \xrightarrow{\gamma_s} \tau_2 \vdash f : \tau_1 \xrightarrow{\phi(\gamma_s)} \tau_2 \,\&\, \emptyset\end{array}}{R; M; \Delta; \Gamma \vdash \mathtt{fix}\ x.\, f : \tau_1 \xrightarrow{\gamma_s} \tau_2 \,\&\, \emptyset} \textit{ (T-FX)}$$

$$\frac{R; M; \Delta; \Gamma \vdash e : \mathtt{Rgn}(r) \,\&\, \gamma \qquad r \neq \bot}{R; M; \Delta; \Gamma \vdash \mathtt{cap}_\eta\ e : \mathtt{unit} \,\&\, \gamma :: \mathtt{Cap}\,\{r \mapsto \eta\}} \textit{ (T-CP)}$$

$$\frac{R; M; \Delta; \Gamma \vdash e_1 : \mathtt{Ref}(\tau, r) \,\&\, \gamma_1 \qquad R; M; \Delta; \Gamma \vdash e_2 : \tau \,\&\, \gamma_2 \qquad r \neq \bot}{R; M; \Delta; \Gamma \vdash e_1 := e_2 : \mathtt{unit} \,\&\, \gamma_1 :: \gamma_2 :: \mathtt{W}\,r} \textit{ (T-AS)}$$

$$\frac{R; M; \Delta; \Gamma \vdash e : \mathtt{Ref}(\tau, r) \,\&\, \gamma \qquad r \neq \bot}{R; M; \Delta; \Gamma \vdash \mathtt{deref}\ e : \tau \,\&\, \gamma :: \mathtt{R}\,r} \textit{ (T-D)}$$

$$\frac{R; M; \Delta; \Gamma \vdash e_1 : \tau \,\&\, \gamma_1 \qquad R; M; \Delta; \Gamma \vdash e_2 : \mathtt{Rgn}(r) \,\&\, \gamma_2 \qquad r \neq \bot}{R; M; \Delta; \Gamma \vdash \mathtt{new}\ e_1\ @\ e_2 : \mathtt{Ref}(\tau, r) \,\&\, \gamma_1 :: \gamma_2 :: \mathtt{Live}\,r} \textit{ (T-NL)}$$

$$\frac{\begin{array}{ll}R; M; \Delta; \Gamma \vdash e_1 : \mathtt{Rgn}(r) \,\&\, \gamma_1 & R; M; \Delta, \rho; \Gamma, x : \mathtt{Rgn}(\rho) \vdash e_2 : \tau \,\&\, \gamma_2 \\ R; \Delta \vdash \tau & \mathsf{translate}(\gamma_2, \rho, (1,1,0), r) = \gamma_2'\end{array}}{R; M; \Delta; \Gamma \vdash \mathtt{newrgn}\ \rho, x\ @\ e_1\ \mathtt{in}\ e_2 : \tau \,\&\, \gamma_1 :: \mathtt{Live}\,r :: \gamma_2'} \textit{ (T-NR)}$$

Figure 3.6: Typing rules.

spectively) and the effect of the function's body (γ). This concatenation of effects is typical for several more constructs: the effects produced by subexpressions are first concatenated in the same order in which these subexpressions are evaluated (left to right), then (possibly) some effect that reflects the construct's behavior is appended. In the case of rule *T-A*, the additional effect is γ — the effect of the function's body. In the descriptions that follow, we will focus on the additional effects and ignore the effects that are propagated from the subexpressions.

For conditional expressions, the type system records the effects of the two branches without unifying them (rule *T-IF*), by adding the effect $\texttt{Join}\,\gamma_2\,\gamma_3$ which represents the two possible paths that will be executed at run-time. Similarly, the typing rule for thread creation (*T-SP*) adds the effect $\texttt{Spawn}\,\xi\,\gamma$ representing the spawn operation, where γ is the effect produced by the expression that will be evaluated in the new thread. Notice that the domains of ξ and γ should coincide: for all regions in the effect of the new thread, the `spawn` construct should determine the exact count vectors that will be passed from the current thread. Although, to simplify the type system, ξ is given as an annotation of the `spawn` construct, an implementation will be able to infer most of it. Because of the premise $\mathsf{dom}(\xi) = \mathsf{dom}(\gamma)$, after type checking the spawned expression, the domain of ξ is found. In general, it is not possible to infer unambiguously the lock counts of regions in ξ, however, it would be much simpler for programmers if they only had to annotate the `spawn` construct with the *locks* that are passed to the new thread.

If we ignore the effects on the function types, the rule for typing recursive functions (*T-FX*) is the standard one. However, in recursive functions it may be impossible to assign the recursive function x the same effect as the function's body f. Suppose that γ_s is the effect of the recursive function x. Then, the effect of the function's body may properly contain γ_s if there are recursive calls to x. In fact, the effect of the function's body is of the form $\phi(\gamma_s)$, where ϕ is a "compositional" function on events, i.e., a function that can only use its parameter as a sub-effect of the result. Our type system chooses an appropriate γ_s by using the function summary. We postpone the discussion on summaries and the restrictions that we impose on recursive functions until Section 3.6.

The typing rule for the capability manipulation construct (*T-CP*) adds the effect $\texttt{Cap}\,\{r \mapsto \eta\}$, representing change in the count vector of region r, whose handle is given by expression e and which must not be the special region $\perp$. Similarly, the typing rules for assignment and dereference (*T-AS* and *T-D*) add an effect with a constraint of type `W` and `R`, respectively. The typing rule for reference allocation (*T-NL*) is more relaxed, adding an effect with the constraint `Live`.

The most complicated typing rule is the one for creating new regions (*T-NR*). This is where the actual effect checking takes place, based on the events and constraints that have been added by the other rules. Assuming that e_1 is a handle for the parent region r, expression e_2 is type checked in an extended typing context that contains ρ and x. The type τ of e_2 should not mention ρ, i.e., the new region cannot escape in the result of e_2. The resulting effect contains the constraint that the parent region must be live. Furthermore, it contains γ_2', a modified version of γ_2 (the effect produced by e_2) which is computed with the partial function translate defined in Figure 3.7.

Function $\mathsf{translate}(\gamma, \rho, \eta, r)$ performs two tasks: (a) it *validates* the effect γ with respect to the specific region ρ, which starts with a count vector η and whose parent is r; and (b) if validation is successful, it produces a *transformed* effect in which all events mentioning ρ have either been removed, or replaced by appropriate events mentioning r (the parent of ρ). Validation keeps track of events modifying ρ's vector count and checks that all constraints are satisfied. Moreover, it checks that region ρ has been properly released at the end of effect γ. Transformation makes sure that ρ is not mentioned in the resulting effect.

Let us see this process with two simple examples. First, consider the effect $\gamma = \texttt{Live}\,\rho :: \texttt{W}\,\rho :: \texttt{Cap}\,\{\rho \mapsto (0,-1,0)\} :: \texttt{R}\,\rho :: \texttt{Cap}\,\{\rho \mapsto (-1,0,0)\}$ that could have been produced by the second line in the following program segment, which is erroneous because it tries to dereference z after it it has been unlocked:

$\texttt{region<}\rho\texttt{>}\ h\ @\ hr\ ;$
$\quad \texttt{let}\ z = \texttt{rnew}(h)\ 42;\ \ *z = 17;\ \texttt{wr\_unlock}(h);\ \ \textit{print}(*z);\ \texttt{rfree}(h)$

$$
\begin{array}{lcll}
\mathsf{rg}(n_1, n_2, n_3) & = & n_1 & \\
\mathsf{wr}(n_1, n_2, n_3) & = & n_2 & \\
\mathsf{rd}(n_1, n_2, n_3) & = & n_3 & \\
\\
\mathsf{bot}(\delta, \bot) & = & \emptyset & \text{if } \delta \notin \{\mathtt{R}, \mathtt{W}\} \\
\mathsf{bot}(\delta, r) & = & \delta\, r & \text{if } r \neq \bot \\
\\
\mathsf{solve}(\mathtt{R}, r, \eta) & = & \mathsf{bot}(\mathtt{Live}, r) & \text{if } \mathsf{ok}(\eta - (1,0,0)) \wedge \mathsf{wr}(\eta) + \mathsf{rd}(\eta) > 0 \\
\mathsf{solve}(\mathtt{R}, r, \eta) & = & \mathsf{bot}(\mathtt{R}, r) & \text{if } \mathsf{ok}(\eta - (1,0,0)) \wedge \mathsf{wr}(\eta) + \mathsf{rd}(\eta) = 0 \\
\mathsf{solve}(\mathtt{W}, r, \eta) & = & \mathsf{bot}(\mathtt{Live}, r) & \text{if } \mathsf{ok}(\eta - (1,0,0)) \wedge \mathsf{wr}(\eta) > 0 \\
\mathsf{solve}(\mathtt{W}, r, \eta) & = & \mathsf{bot}(\mathtt{W}, r) & \text{if } \mathsf{ok}(\eta - (1,0,0)) \wedge \mathsf{wr}(\eta) = 0 \\
\mathsf{solve}(\neg\mathtt{RW}, r, \eta) & = & \mathsf{bot}(\neg\mathtt{RW}, r) & \text{if } \mathsf{ok}(\eta - (1,0,0)) \wedge \mathsf{wr}(\eta) = \mathsf{rd}(\eta) = 0 \\
\mathsf{solve}(\neg\mathtt{W}, r, \eta) & = & \mathsf{bot}(\neg\mathtt{W}, r) & \text{if } \mathsf{ok}(\eta - (1,0,0)) \wedge \mathsf{wr}(\eta) = 0 \\
\mathsf{solve}(\mathtt{Live}, r, \eta) & = & \mathsf{bot}(\mathtt{Live}, r) & \text{if } \mathsf{ok}(\eta - (1,0,0)) \\
\mathsf{solve}(\neg\mathtt{Live}, r, \eta) & = & \emptyset & \text{if } \mathsf{ok}(\eta) \wedge \mathsf{rg}(\eta) = 0 \\
\mathsf{solve}(\neg\mathtt{Live}, r, \eta) & = & \neg\mathtt{Live}\, r & \text{if } \mathsf{ok}(\eta - (1,0,0)) \wedge r \neq \bot \\
\\
\mathsf{p\text{-}constraint}(r, \eta) & = & \mathsf{bot}(\neg\mathtt{RW}, r) & \text{if } \mathsf{wr}(\eta) > 0 \\
\mathsf{p\text{-}constraint}(r, \eta) & = & \mathsf{bot}(\neg\mathtt{W}, r) & \text{if } \mathsf{wr}(\eta) = 0 \wedge \mathsf{rd}(\eta) > 0 \\
\mathsf{p\text{-}constraint}(r, \eta) & = & \emptyset & \text{if } \mathsf{wr}(\eta) = \mathsf{rd}(\eta) = 0
\end{array}
$$

$$\frac{\mathsf{solve}(\neg\mathtt{Live}, r', \eta) = \gamma}{\mathsf{translate}(\emptyset, r, \eta, r') = \gamma} \textit{ (TR-E)} \qquad \frac{r \notin \mathsf{dom}(\xi) \qquad \mathsf{translate}(\gamma, r, \eta, r') = \gamma'}{\mathsf{translate}(\mathtt{Cap}\,\xi :: \gamma, r, \eta, r') = \mathtt{Cap}\,\xi :: \gamma'} \textit{ (TR-CN)}$$

$$\frac{\begin{array}{c}\mathsf{merge}(\xi) = \xi', r \mapsto \eta' \qquad \gamma_s = \mathsf{solve}(\mathtt{Live}, r', \eta) :: \mathtt{Cap}\,\xi' \\ \mathsf{translate}(\gamma, r, \eta + \eta', r') = \gamma' \qquad \mathsf{ok}(\eta + \eta')\end{array}}{\mathsf{translate}(\mathtt{Cap}\,\xi :: \gamma, r, \eta, r') = \gamma_s :: \gamma'} \textit{ (TR-CT)}$$

$$\frac{r_1 \neq r_2 \qquad \mathsf{translate}(\gamma, r_2, \eta, r') = \gamma'}{\mathsf{translate}(\delta\, r_1 :: \gamma, r_2, \eta, r') = \delta\, r_1 :: \gamma'} \textit{ (TR-DN)}$$

$$\frac{\mathsf{solve}(\delta, r', \eta) = \gamma_s \qquad \mathsf{translate}(\gamma, r, \eta, r') = \gamma'}{\mathsf{translate}(\delta\, r :: \gamma, r, \eta, r') = \gamma_s :: \gamma'} \textit{ (TR-DT)}$$

$$\frac{\mathsf{translate}(\gamma_1 :: \gamma, r, \eta, r') = \gamma_1' \qquad \mathsf{translate}(\gamma_2 :: \gamma, r, \eta, r') = \gamma_2'}{\mathsf{translate}(\mathtt{Join}\,\gamma_1\,\gamma_2 :: \gamma, r, \eta, r') = \mathtt{Join}\,\gamma_1'\,\gamma_2'} \textit{ (TR-J)}$$

$$\frac{r \notin \mathsf{dom}(\xi) \qquad \mathsf{translate}(\gamma, r, \eta, r') = \gamma'}{\mathsf{translate}(\mathtt{Spawn}\,\xi\,\gamma_s :: \gamma, r, \eta, r') = \mathtt{Spawn}\,\xi\,\gamma_s :: \gamma'} \textit{ (TR-SN)}$$

$$\frac{\begin{array}{c}\mathsf{merge}(\xi) = \xi', r \mapsto \eta_s \qquad \eta = \eta_r \oplus \eta_s \qquad r_s = \text{if } r' \in \mathsf{dom}(\xi) \text{ then } r' \text{ else } \bot \\ \mathsf{p\text{-}constraint}(r_s, \eta_r) = \gamma_s' \qquad \mathsf{translate}(\gamma_s, r, \eta_s, r_s) = \gamma_s'' \\ \mathsf{p\text{-}constraint}(r', \eta_s) = \gamma_r' \qquad \mathsf{translate}(\gamma, r, \eta_r, r') = \gamma_r'' \qquad \gamma_0 = \mathsf{solve}(\mathtt{Live}, r', \eta)\end{array}}{\mathsf{translate}(\mathtt{Spawn}\,\xi\,\gamma_s :: \gamma, r, \eta, r') = \gamma_0 :: \mathtt{Spawn}\,\xi'\,(\gamma_s' :: \gamma_s'') :: \gamma_r' :: \gamma_r''} \textit{ (TR-ST)}$$

Figure 3.7: Effect validation and transformation.

Starting with a count vector of $\eta = (1, 1, 0)$, the validation phase for γ first checks that the constraint $\texttt{Live}\,\rho$ is satisfied and then checks that the constraint $\texttt{W}\,\rho$ is satisfied. It then proceeds by decrementing the write lock count by 1, thus obtaining a count vector of $(1, 0, 0)$ for ρ. Subsequently, it checks if the constraint $\texttt{R}\,\rho$ is satisfied and finds that it is not, as ρ is now not protected for reading. It therefore generates a constraint $\texttt{R}\,r$ for the parent region; if the parent is locked for reading, then it is safe to access the contents of ρ as well. Validation will fail when translate is invoked for the parent region r, if this is not the case.

Let us now suppose that we fix the bug in the program, e.g., by removing the unlock operation or by moving the read operation before it. The resulting effect passes validation as all constraints are now satisfied and also, when we reach the end of the effect, region ρ has been released (its reference count is zero). The transformation effect translates all events that mention ρ to $\texttt{Live}\,r$. The intuition behind this is that, after validation was successful as far as ρ is concerned, it is only necessary to know that when ρ is mentioned its parent r is live. In this way, it is possible to reject erroneous programs such as the following:

$$\texttt{region}\langle\rho\rangle\ h\ @\ hr\ ;$$
$$\quad \texttt{let}\ z = \texttt{rnew}(h)\ 42;\ \ *z = 17;\ \texttt{rfree}(hr);\ \ \mathit{print}(*z);\ \texttt{rfree}(h)$$

where the effect for the second line is

$$\gamma = \texttt{Live}\,\rho :: \texttt{W}\,\rho :: \texttt{Cap}\,\{r \mapsto (0, -1, 0)\} :: \texttt{R}\,\rho :: \texttt{Cap}\,\{\rho \mapsto (-1, 0, 0)\}$$

In this case, with respect to ρ, validation succeeds and the transformed effect is $\gamma' = \texttt{Live}\,r :: \texttt{Live}\,r :: \texttt{Cap}\,\{r \mapsto (0, -1, 0)\} :: \texttt{Live}\,r :: \texttt{Live}\,r$. Notice that the event mentioning r in γ is not affected by the transformation. Now, when later this effect will be validated with respect to r, assuming that the initial reference count is equal to one, validation will fail.

In Figure 3.7, the definition of translate uses the partial function solve, which checks whether the atomic effect denoted by the first argument is valid with respect to a region that has a vector count given by the third argument. If this cannot be established unconditionally, the result is a constraint on the parent of this region that is given by the second argument. The definition of solve is straightforward. In the cases for R and W, if the constraint is found to be satisfied by the given count vector then the only requirement is that the parent region is live (except when the parent region is $\perp$ which is always considered live). Otherwise, the constraint is propagated to the parent region, e.g., if a region is not write-protected, a W constraint can only be satisfied if the same constraint is satisfied for its parent. A Live constraint for the parent is also generated in the case of Live. On the other hand, the negative constraints $\neg\texttt{RW}$ and $\neg\texttt{W}$ always propagate to the parent. The same happens with $\neg\texttt{Live}$, unless the region's reference count is zero.

The partial function $\textsf{translate}(\gamma, r, \eta, r')$ is defined with a case analysis on the first event in γ. If γ is empty, rule *TR-E* requires that region r is not live. Rules *TR-CN* and *TR-DN* handle the case of atomic events that refer to regions other than r; these remain unaffected. On the other hand, rules *TR-CT* and *TR-DT* use solve to validate and translate an atomic effect referring to r. Rule *TR-J* handles the case of "join" events: the two branches are translated separately, prepended to the rest of the effect, and the results are joined.

Rules *TR-SN* and *TR-ST* handle "spawn" events. The former is used when region r is not passed to the new thread. The latter is quite complicated. It determines the count vector η_s that is passed to the new thread and the count vector η_r that is left to the current thread.

It determines r_s, the parent of r as seen by the new thread (it is r', if r' is also passed to the new thread, otherwise it is $\perp$). It then translates the effects of the new thread and the current thread with the appropriate count vectors and parents. Finally, it prepends appropriate constraints, generated by function p-constraint, which guarantee mutual exclusion between the new and the current thread.

Let us suppose that a region r is shared between two threads, the first thread sees r' as the parent of r and the second thread sees η as the vector count for r. The purpose of $\textsf{p-constraint}(r', \eta)$ is to determine the constraints on r' that must be satisfied by first thread to guarantee mutual exclusion. If

$$\xi_1 - \xi_2 \;=\; \{r \mapsto \xi_1(r) - \xi_2(r)\} \quad \text{where } \xi(r) = \eta \text{ if } r \mapsto \eta \in \xi \text{ and } \xi(r) = (0,0,0) \text{ otherwise}$$
$$\mathsf{ok}(\xi) \;=\; \forall \iota \mapsto \eta \in \xi.\ \mathsf{ok}(\eta)$$

$$\frac{\mathsf{ok}(\xi)}{\mathsf{recursive}(\xi;\emptyset) = \xi}\ \textit{(R-E)} \qquad \frac{\delta \notin \{\neg\mathtt{RW}, \neg\mathtt{W}, \neg\mathtt{Live}\} \quad \mathsf{recursive}(\xi;\gamma) = \xi'}{\mathsf{recursive}(\xi;\delta\, r :: \gamma) = \xi'}\ \textit{(R-D)}$$

$$\frac{\mathsf{ok}(\xi) \quad \mathsf{recursive}(\xi - \xi';\gamma) = \xi''}{\mathsf{recursive}(\xi;\mathtt{Cap}\,\xi' :: \gamma) = \xi''}\ \textit{(R-C)}$$

$$\frac{\begin{array}{cc}\mathsf{ok}(\xi) & \xi_r = \xi - \xi_s \\ \forall r \mapsto \eta \in \xi_s.\ \mathsf{rd}(\eta) = \mathsf{wr}(\eta) = 0 & \mathsf{recursive}(\xi_r;\gamma) = \xi_r'\end{array}}{\mathsf{recursive}(\xi;\mathtt{Spawn}\,\xi_s\,\gamma_s :: \gamma) = \xi_r'}\ \textit{(R-S)}$$

$$\frac{\mathsf{recursive}(\xi;\gamma_1) = \xi' \quad \mathsf{recursive}(\xi;\gamma_2) = \xi' \quad \mathsf{recursive}(\xi';\gamma) = \xi''}{\mathsf{recursive}(\xi;\mathtt{Join}\,\gamma_1\,\gamma_2 :: \gamma) = \xi''}\ \textit{(R-J)}$$

$$\frac{\begin{array}{c}\mathsf{recursive}(\xi_1;\gamma) = \xi_1 \\ \xi_1 = \{r \mapsto (1,0,0) \mid r \in \mathsf{dom}(\gamma)\} \quad \xi_2 = \{r \mapsto (-1,0,0) \mid r \in \mathsf{dom}(\gamma)\}\end{array}}{\mathsf{summary}(\gamma) = \mathtt{Cap}\,\xi_1 :: \mathtt{Spawn}\,\xi_1\,(\gamma :: \mathtt{Cap}\,\xi_2)}\ \textit{(SUM)}$$

Figure 3.8: Summarized effects of recursive functions.

η has a positive write-lock count (in the second thread), then the region's parent r' must not be read- or write-protected (in the first thread). Otherwise, if η has a positive read-lock count (in the second thread), then the region's parent r' must not be write-protected (in the first thread). Otherwise, if both lock counts are zero in η, no additional constraints must be imposed on r'. Notice that p-constraint is used twice in rule *TR-ST* symmetrically.

3.6 Effects for recursive functions

Although the basics of how to type check recursive functions were explained in the previous section, we have not explained how to find the effect γ_s in rule *T-FX*. This is done with the partial function summary, defined in Figure 3.8, which imposes some restrictions on recursive functions and calculates the *summarized* effect. The restrictions imposed are the following and they apply to "external" regions, i.e., regions that exist before a recursive function is called (in contrast to regions that are created in a recursive function's body):

- When a recursive function returns, the counts of all external regions must be equal to the counts when the function was called. This is ensured by the premise $\mathsf{recursive}(\xi_1;\gamma) = \xi_1$ in rule *SUM*. It implies that a recursive function cannot deallocate any external regions.
- If a recursive function spawns new threads, it cannot pass to them any locks to external regions. This is ensured by rule *R-S*.
- A recursive function cannot presume any existing locks on external regions. This is ensured by the definition of ξ_1 and the spawn event in rule *SUM*.

After all this, we can now return to the explanation of rule *T-FX* on page 47. If ϕ is the compositional function on effects that corresponds to the recursive function's body, then $\phi(\emptyset)$ is the effect that one gets by completely ignoring the recursive calls. We just use this effect to identify the external regions that a recursive function uses and to construct the effect γ_L, which contains `Live` constraints for all those regions. Then we take the effect $\phi(\gamma_L)$ as the basis for our summary.

To summarize an effect, as shown in rule *SUM*, we essentially check that the constraints stated above are satisfied. The first two constraints are directly enforced by the partial function recursive,

whose definition is straightforward. Then, the effect γ to be summarized is *isolated* inside a `Spawn` effect, which enforces the third constraint.

3.7 Type safety

In this section we discuss the fundamental theorems that prove type safety of our language.[3] The type safety formulation is based on proving the *preservation* and *progress* lemmata. Informally, a program written in our language is safe when for each thread of execution an evaluation step can be performed or that thread is waiting for a lock (*blocked*). As discussed in Section 3.4, a thread may become stuck when it accesses a region that is not live or accessible — these are obviously the interesting cases in our concurrent setting; of course a thread may become stuck when it performs a non well-typed operation. Blocked threads and deadlocked threads are not considered to be stuck.

Definition 3.1 (Constraint validity) Predicate $\mathsf{cvalid}(\delta; \imath; \theta)$ is true when the constraint δ on r is consistent with the run-time hierarchy θ. It is defined as follows:

$$\frac{}{\mathsf{cvalid}(\mathtt{Live}; \bot; \theta)}\ \textit{(C-T)} \qquad \frac{\theta = \theta', \imath \mapsto (\eta, \jmath) \quad \mathsf{solve}(\delta, \jmath, \eta) = \emptyset}{\mathsf{cvalid}(\delta; \imath; \theta)}\ \textit{(C-B)}$$

$$\frac{\theta = \theta', \imath \mapsto (\eta, \jmath) \quad \mathsf{solve}(\delta, \jmath, \eta) = \delta'\ \jmath \quad \mathsf{cvalid}(\delta'; \jmath; \theta')}{\mathsf{cvalid}(\delta; \imath; \theta)}\ \textit{(C-R)}$$

Function solve is used in rules *C-B* and *C-R* to enable the validation of hierarchical constraints.

Definition 3.2 (Validity for count modification) The partial function $\mathsf{xvalid}(\xi; \theta)$ is defined as follows. We have $\mathsf{xvalid}(\xi; \theta) = \theta'$ when it is valid to apply the count modifications defined by ξ to the hierarchy θ and the result is θ'.

$$\frac{}{\mathsf{xvalid}(\emptyset; \theta) = \theta}\ \textit{(X-E)} \qquad \frac{\begin{array}{c}\mathsf{ok}(\eta + \eta') \quad \mathsf{cvalid}(\mathtt{Live}\ ; \imath; \theta, \imath \mapsto (\eta, \jmath)) \\ \mathsf{xvalid}(\xi; \theta, \imath \mapsto (\eta + \eta', \jmath)) = \theta'\end{array}}{\mathsf{xvalid}(\xi, \imath \mapsto \eta'; \theta, \imath \mapsto (\eta, \jmath)) = \theta'}\ \textit{(X-S)}$$

Definition 3.3 (Effect validity) The partial functions $\mathsf{evalid}(\zeta; \theta)$ and $\mathsf{gvalid}(\gamma; \theta)$, as well as the predicate $\mathsf{valid}(\gamma; \theta)$ are defined with the following rules. We have $\mathsf{evalid}(\zeta; \theta) = \theta'$ when the event ζ is valid in hierarchy θ and the result is θ'. Similarly, we have $\mathsf{gvalid}(\gamma; \theta) = \theta'$ when the effect γ is valid in hierarchy θ and the result is θ'. Finally, $\mathsf{gvalid}(\gamma; \theta)$ is true when the effect γ is valid in hierarchy θ and the result is a hierarchy with no live regions.

$$\frac{\mathsf{cvalid}(\delta; \imath; \theta)}{\mathsf{evalid}(\delta\ \imath; \theta) = \theta}\ \textit{(V-D)} \qquad \frac{\mathsf{xvalid}(\mathsf{merge}(\xi); \theta) = \theta'}{\mathsf{evalid}(\mathtt{Cap}\ \xi; \theta) = \theta'}\ \textit{(V-C)}$$

$$\frac{\mathsf{gvalid}(\gamma_1; \theta) = \theta' \quad \mathsf{gvalid}(\gamma_2; \theta) = \theta'}{\mathsf{evalid}(\mathtt{Join}\ \gamma_1\ \gamma_2; \theta) = \theta'}\ \textit{(V-J)}$$

$$\frac{\begin{array}{c}\forall \imath \in \mathsf{dom}(\theta_s).\ \mathsf{cvalid}(\mathtt{Live}\ ; \imath; \theta) \\ \mathsf{merge}(\xi) \vdash \theta = \theta_r \oplus \theta_s \quad \mathsf{valid}(\gamma_s; \theta_s) \quad \mathsf{mutex}(\{\theta_s, \theta_r\})\end{array}}{\mathsf{evalid}(\mathtt{Spawn}\ \xi\ \gamma_s; \theta) = \theta_r}\ \textit{(V-S)}$$

$$\frac{\mathsf{ok}(\theta)}{\mathsf{gvalid}(\emptyset; \theta) = \theta}\ \textit{(V-E)} \qquad \frac{\mathsf{ok}(\theta_1) \quad \mathsf{evalid}(\zeta; \theta_1) = \theta_2 \quad \mathsf{gvalid}(\gamma; \theta_2) = \theta_3}{\mathsf{gvalid}(\zeta :: \gamma; \theta_1) = \theta_3}\ \textit{(V-K)}$$

$$\frac{\mathsf{gvalid}(\gamma; \theta) = \theta' \quad \mathsf{live}(\theta') = \emptyset}{\mathsf{valid}(\gamma; \theta)}\ \textit{(V-V)}$$

[3] Full proofs and a full formalization of our language are given in Appendix B.

For the definition of evalid, the most interesting cases are rules *V-J* and *V-S*. In the former, a "join" effect is valid when both branches are valid and produce the same result. In the latter, for the validation of a "spawn" effect, θ is split into θ_s and θ_r, according to $\mathsf{merge}(\xi)$, and the lock counts of these two hierarchies must satisfy the mutual exclusion criteria; then, θ_s is the hierarchy of the new thread and γ_s must be valid for it, whereas θ_r is the hierarchy of the main thread and therefore the result.

Definition 3.4 (Thread typing) Let T be a collection of threads and $R; M$ be a global typing context. The relation $R; M \vdash T$ is defined as follows:

$$\frac{}{R; M \vdash \emptyset} \qquad \frac{\begin{array}{c} R; M \vdash T \qquad R; M; \emptyset; \emptyset \vdash e : \mathtt{unit} \,\&\, \gamma \qquad \mathsf{valid}(\gamma; \theta) \\ \forall \imath \mapsto (\eta, \jmath) \in \theta.\ \imath \in R \wedge \jmath \in R \cup \{\bot\} \end{array}}{R; M \vdash T, \langle \theta; e \rangle}$$

For each thread $\langle \theta, e \rangle$ in T, the effect γ produced by the closed expression e must be valid under θ, the regions contained in θ must be a subset of R, and their parents must be contained in R or $\bot$.

Definition 3.5 (Store typing) Let S be a store and $R; M$ be a global typing context. The relation $R; M \vdash S$ is defined as follows:

$$\frac{\begin{array}{c} R = \{\imath \mid \imath \mapsto H \in S\} \qquad \{(\ell, \imath) \mid \ell \mapsto (\tau, \imath) \in M\} = \{(\ell, \imath) \mid \ell \mapsto v \in H \wedge \imath \mapsto H \in S\} \\ \forall \ell \mapsto (\tau, \imath) \in M.\ R; M; \emptyset; \emptyset \vdash S(\imath)(\ell) : \tau \,\&\, \emptyset \end{array}}{R; M \vdash S}$$

The set of region names in S must be equal to R. The set of locations in M must be equal to the set of locations in all the heaps in S, and the regions in which these locations reside must coincide. Finally, for each location ℓ, the value stored in this location must be closed, must have the type mentioned by M and must produce an empty effect.

Definition 3.6 (Configuration typing) Let $S; T$ be a configuration and $R; M$ be a global typing context. The relation $R; M \vdash S; T$ is defined as follows:

$$\frac{R; M \vdash T \qquad R; M \vdash S \qquad \mathsf{mutex}(\{\theta \mid \langle \theta; e \rangle \in T\})}{R; M \vdash S; T}$$

A configuration $S; T$ is well-typed with respect to $R; M$ when both the collection of threads T and the store S are well-typed with respect to $R; M$. In addition the hierarchies of all threads in T must adhere to the mutual exclusion criteria (see predicate mutex in Figure 3.4 on page 45).

Definition 3.7 (Running) Let $\langle \theta; e \rangle$ be a thread, T be the remaining threads, and S be a store. The predicate $\mathsf{running}(S; T; \langle \theta; e \rangle)$ is defined as follows:

$$\frac{S; T, \langle \theta; e \rangle \rightsquigarrow S'; T' \qquad T \subseteq T'}{\mathsf{running}(S; T, \langle \theta; e \rangle; \langle \theta; e \rangle)}$$

A thread is running when it can take one of the evaluation steps in Figure 3.3.

Definition 3.8 (Blocked) Let $\langle \theta; e \rangle$ be a thread and T be the remaining threads. The predicate $\mathsf{blocked}(T; \langle \theta; e \rangle)$ is defined as follows:

$$\frac{\begin{array}{c} \imath \in \mathsf{live}(\theta, \imath \mapsto (\eta, \jmath)) \qquad \mathsf{mutex}(\{\theta, \imath \mapsto (\eta, \jmath)\} \cup \{\theta' \mid \langle \theta'; e' \rangle \in T\}) \\ \neg\mathsf{mutex}(\{\theta, \imath \mapsto (\eta + \eta', \jmath)\} \cup \{\theta' \mid \langle \theta'; e' \rangle \in T\}) \end{array}}{\mathsf{blocked}(T; \langle \theta, \imath \mapsto (\eta, \jmath); E[\mathtt{cap}_{\eta'}\ \mathtt{rgn}_\imath] \rangle)}$$

A thread is blocked when it attempts to acquire the lock of a live region that is locked by another thread.

Definition 3.9 (Not stuck) Let $S; T$ be a configuration. The relation $\vdash S; T$ is defined as follows:

$$\frac{\forall \langle \theta; e \rangle \in T.\ \mathsf{running}(S; T; \langle \theta; e \rangle) \vee \mathsf{blocked}(T; \langle \theta; e \rangle)}{\vdash S; T}$$

A configuration $S; T$ is *not stuck* when each thread in T is either running or blocked by some other thread.

Given these definitions, we can now present the main results of this chapter.

Lemma 3.1 (Progress) Let $R; M$ be a global typing context and $S; T$ be a well-typed configuration with $R; M \vdash S; T$. Then $\vdash S; T$, in other words $S; T$ is not stuck.

Proof sketch. By induction on the evaluation relation. Most cases can be trivially shown by using the invariants provided by predicate valid, which is obtained by inversion of the well formedness hypothesis for T.

Lemma 3.2 (Preservation) Let $R; M$ be a global typing context and $S; T$ be a well-typed configuration with $R; M \vdash S; T$. If the operational semantics takes a step $S; T \rightsquigarrow S'; T'$, then there exist $R' \supseteq R$ and $M' \supseteq M$ such that the resulting configuration is well-typed with $R'; M' \vdash S'; T'$.

Proof sketch. By induction on the evaluation relation. Most cases can be trivially shown by using the invariants provided by predicate valid, which is obtained by inversion of the well formedness hypothesis for T. The most interesting cases are *E-NR*, where it is must be shown that function $\mathsf{translate}$ entails effect validity, and *E-FX*, where it must be shown that function $\mathsf{summary}$ entails effect validity. These two are established by Lemmata 3.3 and 3.4, respectively.

Lemma 3.3 (Translate implies valid) If $\mathsf{valid}(\mathsf{translate}(\gamma, \imath, \eta, \jmath); \theta)$ for some region $\imath$ such that $\imath \notin \mathsf{dom}(\theta)$, then $\mathsf{valid}(\gamma; \theta, \imath \mapsto (\eta, \jmath))$.

Proof sketch. By induction on the structure of γ. The most interesting case is when γ is of the form $\mathsf{Spawn}\ \xi_s\ \gamma_s :: \gamma'$, $\mathsf{merge}(\xi_s) = \xi'_s, \imath \mapsto \eta_s$ and $\eta = \eta_r \oplus \eta_s$; it must be shown that if $\texttt{mutex}$ holds for the hierarchies of the child and parent thread, θ_s and θ_r respectively, then $\texttt{mutex}$ also holds when $\imath$ is added in the two hierarchies with counts η_s and η_r respectively. We employ the definitions of function $\mathsf{p} - \texttt{constraint}$ and $\eta = \eta_r \oplus \eta_s$ to show that when one of the threads has write access to $\imath$, then the other thread does not have access to $\imath$ and vice versa.

Lemma 3.4 (Recursion implies valid) If $\gamma_L = \{\texttt{Live}\, r \mid r \in \mathsf{dom}(\phi(\emptyset))\}$, $\gamma_s = \mathsf{summary}(\phi(\gamma_L))$, and $\mathsf{valid}(\gamma_s :: \gamma; \theta)$, then $\mathsf{valid}(\phi(\gamma_s) :: \gamma; \theta)$.

Proof sketch. Using a series of intermediate lemmata, the proof is reduced to showing that if $\mathsf{gvalid}(\gamma_s; \theta_0) = \theta_0$, $\mathsf{hierarchy\_ok}(\theta_1; \theta_0)$, and $\mathsf{gvalid}(\phi(\gamma_L); \theta_1) = \theta_2$, then $\mathsf{gvalid}(\phi(\gamma_s); \theta_1) = \theta_2$. This can be achieved by induction on the structure of the compositional function ϕ.

Now, assume that e is the expression that represents the initial program. Let $S_0 = \emptyset$ be the initial empty store and $T_0 = \emptyset, \langle \emptyset; e \rangle$ be the initial set of threads, consisting of just e with an empty region hierarchy. We are interested only in programs that are closed, well typed and whose effect is consistent with the initial empty region hierarchy. Our type safety theorem shows that such programs cannot become stuck.

Theorem 3.1 (Type safety) Let e be such that $\emptyset; \emptyset; \emptyset; \emptyset \vdash e : \texttt{unit}\ \&\ \emptyset$. If the operational semantics takes any number of steps $S_0; T_0 \rightsquigarrow^n S_n; T_n$, then the resulting configuration $S_n; T_n$ is not stuck.

Proof. Let $R_0 = \emptyset$ and $M_0 = \emptyset$. Using the assumptions it is easy to establish that $R_0; M_0 \vdash S_0; T_0$. Then, by induction on the number of steps n and using Lemma 3.2, we show that there exist $R_n \supseteq R_0$ and $M_n \supseteq M_0$ such that $R_n; M_n \vdash S_n; T_n$. Finally, Lemma 3.1 implies that $S_n; T_n$ is not stuck.

The empty contexts that are used when type-checking the initial program e guarantee that no explicit region values ($\texttt{rgn}_\imath$) or location values ($\texttt{loc}_\ell$) are used in the source of the initial program.

$$\frac{\vdash R; M; \Delta; \Gamma \qquad R; \Delta \vdash \tau}{R; M; \Delta; \Gamma \vdash \Box : \tau \xrightarrow{\tau} \emptyset\gamma_2} \textit{(E0)} \qquad \frac{\begin{array}{c} R; M; \Delta; \Gamma \vdash E : \tau_2 \xrightarrow{\tau_3} \gamma_2 \\ R; M; \Delta; \Gamma \vdash F : \tau_1 \xrightarrow{\tau_2} \gamma_1 \end{array}}{R; M; \Delta; \Gamma \vdash E[F] : \tau_1 \xrightarrow{\tau_3} \gamma_1 :: \gamma_2} \textit{(E1)}$$

$$\frac{\begin{array}{c} \tau \equiv \tau_1 \xrightarrow{\gamma_a} \tau_2 \qquad R; \Delta \vdash \tau \\ R; M; \Delta; \Gamma \vdash e_2 : \tau_1 \,\&\, \gamma_1 \end{array}}{R; M; \Delta; \Gamma \vdash \Box \;\; e_2 : \tau \xrightarrow{\tau_2} \gamma_1 :: \gamma_a} \textit{(F1)} \qquad \frac{\begin{array}{c} \tau \equiv \tau_1 \xrightarrow{\gamma_a} \tau_2 \\ R; M; \Delta; \Gamma \vdash v_1 : \tau \,\&\, \emptyset \end{array}}{R; M; \Delta; \Gamma \vdash v_1 \;\; \Box : \tau_1 \xrightarrow{\tau_2} \gamma_a\gamma_3} \textit{(F2)}$$

$$\frac{\begin{array}{c} R; M; \Delta, \rho; \Gamma, x : \mathtt{Rgn}(\rho) \vdash e_2 : \tau \,\&\, \gamma_1 \qquad R; \Delta \vdash r \\ R; \Delta \vdash \tau \qquad \mathtt{Live}\, r :: \mathsf{translate}(\gamma_1, \rho, (1,1,0), r) = \gamma_2 \end{array}}{R; M; \Delta; \Gamma \vdash \mathtt{newrgn}\ \rho, x\ @\ \Box\ \mathtt{in}\ e_2 : \mathtt{Rgn}(r) \xrightarrow{\tau} \gamma_2\gamma'} \textit{(F3)}$$

$$\frac{\vdash R; M; \Delta; \Gamma \qquad R; \Delta \vdash \forall\rho.\,\tau \qquad R; \Delta \vdash r \qquad r \neq \bot}{R; M; \Delta; \Gamma \vdash \Box\,[r] : \forall\rho.\,\tau \xrightarrow{\tau[r/\rho]} \emptyset} \textit{(F4)}$$

$$\frac{R; M; \Delta; \Gamma \vdash e_2 : \mathtt{Rgn}(r) \,\&\, \gamma_2 \qquad R; \Delta \vdash \tau \qquad r \neq \bot}{R; M; \Delta; \Gamma \vdash \mathtt{new}\ \Box\ @\ e_2 : \tau \xrightarrow{\mathtt{Ref}(\tau, r)} \gamma_2 :: \mathtt{Live}\, r\gamma'} \textit{(F5)}$$

$$\frac{R; \Delta \vdash r \qquad R; M; \Delta; \Gamma \vdash v : \tau \,\&\, \emptyset \qquad r \neq \bot}{R; M; \Delta; \Gamma \vdash \mathtt{new}\ v\ @\ \Box : \mathtt{Rgn}(r) \xrightarrow{\mathtt{Ref}(\tau, r)} \mathtt{Live}\, r\gamma'} \textit{(F6)}$$

$$\frac{R; \Delta \vdash r \qquad R; M; \Delta; \Gamma \vdash e_2 : \tau \,\&\, \gamma_1 \qquad r \neq \bot}{R; M; \Delta; \Gamma \vdash \Box := e_2 : \mathtt{Ref}(\tau, r) \xrightarrow{\mathtt{unit}} \&\, \gamma_1 :: \mathtt{W}\, r} \textit{(F7)}$$

$$\frac{R; M; \Delta; \Gamma \vdash \mathtt{loc}_\ell : \mathtt{Ref}(\tau, \imath) \,\&\, \emptyset \qquad \imath \neq \bot}{R; M; \Delta; \Gamma \vdash \mathtt{loc}_\ell := \Box : \tau \xrightarrow{\mathtt{unit}} \&\, \mathtt{W}\, \imath} \textit{(F8)}$$

$$\frac{\vdash R; M; \Delta; \Gamma \qquad R; \Delta \vdash \mathtt{Ref}(\tau, r) \qquad r \neq \bot}{R; M; \Delta; \Gamma \vdash \mathtt{deref}\ \Box : \mathtt{Ref}(\tau, r) \xrightarrow{\tau} \&\, \mathtt{R}\, r} \textit{(F9)}$$

$$\frac{\vdash R; M; \Delta; \Gamma \qquad R; \Delta \vdash \mathtt{Rgn}(r) \qquad r \neq \bot}{R; M; \Delta; \Gamma \vdash \mathtt{cap}_\eta\ \Box : \mathtt{Rgn}(r) \xrightarrow{\mathtt{unit}} \&\, \mathtt{Cap}\, \{r \mapsto \eta\}} \textit{(F10)}$$

$$\frac{R; M; \Delta; \Gamma \vdash e_1 : \tau \,\&\, \gamma_2 \qquad R; M; \Delta; \Gamma \vdash e_2 : \tau \,\&\, \gamma_3}{R; M; \Delta; \Gamma \vdash \mathtt{if}\ \Box\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 : \mathtt{bool} \xrightarrow{\tau} \&\, \mathtt{Join}\, \gamma_2\, \gamma_3} \textit{(F11)}$$

Figure 3.9: Evaluation context typing rules.

Chapter 4

Concurrent Cyclone

4.1 Overview

Cyclone is a safe variant of C that offers modern programming language features such as first-class polymorphism, exceptions, tuples, namespaces, (extensible) algebraic data types, and region-based memory management. However, Cyclone has no built-in support for concurrency. In this chapter we present the main features of Cyclone regarding memory management and argue that the safety guarantees of Cyclone are compromised by introducing concurrency to the language via external library support such as pthreads. We then discuss the integration of the type systems and language constructs described in the previous chapters to Cyclone as two distinct implementations, and argue about the decisions that we have made in order to guarantee memory safety and race freedom at the implementation level. We evalutate the performance of programs written in extended Cyclone (i.e. Concurrent Cyclone) against C/pthreads programs and present the performance results.

4.2 Cyclone: A memory-safe dialect of C

In this section we provide a brief overview of Cyclone before our additions. In particular, we first discuss memory management aspects of Cyclone: how "traditional" Cyclone regions are used and identify some shortcomings that are alleviated in the "extended" regions we described in this chapter. We also show through several examples that Cyclone's memory safety guarantees only hold for sequential programs. (All code excerpts in this chapter are shown using Cyclone syntax.)

4.2.1 Memory management in Cyclone

Cyclone employs a uniform treatment of different memory segments such as the main heap, the stack and individual regions. More specifically, memory segments are mapped into *logical memory partitions*. Each data object is allocated in a single memory segment, but references to objects may refer to multiple segments. Hereon, we overload the term "region" to mean a type-level logical memory partition, a run-time entity that enables fast allocation and bulk deallocation of objects, or a memory segment such as the heap and the stack.

For instance, a stack frame is treated as a region holding the values of variables declared in a lexical block. As another example, the main heap is an immortal region that contains all global variables. The type system of Cyclone tracks the set of *live* regions at each program point and verifies that the regions associated with each accessed object are indeed a subset of the live regions.

```
{ region<`r> h ;                          // live regions: {`r}
  int *`r z = rnew(h) 42;                 //               {`r}
      ...                                 //               {`r}
}                                         //               { }
```

The above example illustrates how a scoped region can be created and used in Cyclone: the first statement allocates a fresh memory segment, and associates this segment with a fresh type-level region

(i.e., ‘r). Following Cyclone's terminology, we use a leading backquote for type-level names, e.g. ‘r. (We will use the same name without the backquote for the corresponding region handle, which here is explicitly named h.) The comments on the right-hand side of the example's code show the live region set (i.e., the *effect*) at each program point.

The new region can be accessed via its *region handle* (h), which is given the *singleton type* `region`<‘r>. The second statement uses h to allocate memory for a single integer and initializes it to the value 42. The type of the fresh reference is annotated with region ‘r (i.e., `int` $*$ ‘r). The type system ensures that the reference can only be accessed when ‘r is in the current effect.

In Cyclone, the uniform treatment of memory allows for polymorphism over different kinds of memory segments.

```
void swap (int *‘r1 x , int *‘r2 y);
```

For instance, the above line of code declares a function that swaps the contents of the variables x and y located at regions ‘r_1 and ‘r_2 respectively. Both ‘r_1 and ‘r_2 are polymorphic and can be instantiated with any region. The following line of code invokes `swap` by explicitly instantiating both ‘r_1 and ‘r_2 to the same region ‘r.

```
{ region<‘r>h ;
  int *‘r z = rnew(h) 42;
  int *‘r y = rnew(h) 54;
  swap(z, y);                                   // effect of swap: {‘r, ‘r}
}
```

As shown in the comment, type-level regions can be freely aliased in a effect (e.g. {‘r, ‘r}). The downside of allowing unrestricted aliasing is that scoped regions can only be deallocated implicitly by the run-time system when a region's scope ends.

To ameliorate the situation, Cyclone's region system has been extended with three powerful features, namely *tracked types*, the notion of *borrowing* tracked types and *existential types*. Tracked types, which are closely related to linear types, disallow aliasing of *tracked* references. Borrowing can be used to convert a tracked reference to an aliasable reference within a particular scope. The aliasable reference is accessible within the scope, whereas the tracked reference becomes inaccessible for the duration of the scope. Finally, existential types serve as the means for overcoming lexically scoped region names, by permitting the on-demand concealment and disclosure of region names. Cyclone allows access and deallocation of non-lexically scoped (i.e., *dynamically* scoped) regions as follows:

- A request is made to the run-time system to allocate a fresh dynamic region.
- The run-time system returns an existential package containing some region name ‘r and a *key* (i.e., a tracked reference) to the handle of the fresh region. The handle is also annotated with ‘r.
- The existential package is unpacked and ‘r is brought into scope as well as the key.
- The program can immediately deallocate the new region by deallocating the key, or it may temporarily yield access to the key by allowing it to be *borrowed* within a scope. When this happens, ‘r is added to the effect and the region referred by the key is usable.

The following example illustrates a similar scenario:

```
void access_and_deallocate (NewDynRgn pr ) {
   let NewDynRgn{<‘r> key} = pr;                // open existential
     { region h = open(key);                    // borrow key for this scope
       let x = rnew(h) 42;
          ...
     }
```

```
    ...                                          // do some work
    free_ukey(key);                              // deallocate region
}
```

It should be noted that a dynamic region cannot be deallocated when its key has been *borrowed*. Additionally, Cyclone allows tracked references to leak and thus allows dynamic regions to leak as well. To tackle this issue, an intra-procedural analysis can be used to report tracked reference leaks. In practice, this analysis is impractical as it produces a large number of false positives [Swam06]. For instance, when a function call takes place between the allocation and deallocation point of a tracked reference, the analysis must report that the tracked reference may leak as an uncaught exception may be thrown during the call. For a detailed discussion about memory management aspects of Cyclone we refer the reader to the work of Swamy *et al.* [Swam06].

For sequential Cyclone programs, our extended regions disallow memory leaks in the presence of a complex shared memory management scheme with bulk region deallocation, allow region deallocation at *any program point* and simplify the process of creating, using and deallocating explicitly freeable regions.

4.2.2 Concurrency in Cyclone

Cyclone does not have language support for concurrency. Instead, it provides an interface to the pthreads library, which allows programmers to spawn new threads and use numerous synchronization primitives to control the interaction between threads. The interface to the pthreads library ensures that the run-time data structures are correctly initialized before a new thread runs.

To preserve memory safety (e.g., absence of dangling pointers), Cyclone requires that all memory regions passed to a new thread must live at least as long as the immortal (main) heap. This implies that threads can interact with other threads via dynamically allocated references that reside in the heap or in global variables. This restriction diminishes the explicit memory management benefits of Cyclone; in concurrent programs, aliasable heap references can only be garbage collected. The following definition has been extracted from Cyclone's interface to pthreads library:

$$\texttt{int pthread\_create}\,(\texttt{pthread\_t}\ @,\ \texttt{const pthread\_attr\_t}\ *,\\ \text{`}a(@\text{`}H)(\text{`}b),\ \text{`}b\ arg : \texttt{regions}(\text{`}b) \leq \text{`}H)$$

The most interesting part of the above definition is $\texttt{regions}(\text{`}b) \leq \text{`}H$, which says that all region names occurring in the type that will instantiate the type variable $\text{`}b$ must be live for at least as long as the immortal heap ($\text{`}H$). Tracked pointers cannot be passed to threads.

But the memory safety guarantees that Cyclone aims for can be compromised in other ways in the presence of multi-threading. Here we will only mention a few such cases.

Firstly, the data flow analysis performed for identifying where dynamic checks (e.g., null pointer and array bounds checks) should be inserted is unsound in a concurrent setting. Consider the following code fragment:

$$\texttt{void foo}\,(int\ *\text{`}r\ *\text{`}r\ x)\,\{\\ \quad \texttt{if}\ (x\ !{=}\ \texttt{NULL}\ \&\&\ *x\ !{=}\ \texttt{NULL})\ **x = 42;\\ \}$$

Assuming that x is a shared *possibly null* reference, then the analysis will deduce that $**x$ can be accessed within the conditional statement as x and $*x$ are definitely *not null*. This property does not hold for concurrent programs that share x, but do not synchronize their accesses to it.

Secondly, some features of Cyclone such as pattern matching, accesses to wide references (i.e., *fat pointers*) and *swap* operations between tracked references must be performed *atomically*. The lack of atomicity in swap operations and wide references can trivially compromise memory safety and cause dangling pointer dereferences and double "free" operations.

Last but not least, Cyclone's type system does not guard against data races. The absence of data races gives additional guarantees to the programmer and allows a thread-aware compiler to perform certain kinds of optimizations that should only be applied to sequential programs. As will be shown in the next two sections, we have solved some of these issues by implementing an adjusted version of our type system and operational semantics in Cyclone. We have also re-engineered the run-time system of Cyclone so that it is mostly non-blocking and thread-safe.

4.3 Common features

There are two distinct implementations for each of the type systems presented in the previous chapters. Both implementations have been integrated into the original compiler of Cyclone as distinct stages and are approximately twenty thousand lines of code. The two implementations have some features in common, such as the kind system, exception and re-entrant function annotations, the memory consistency model and are subject to the same restrictions such as their limited interaction with Cyclone's type polymorphism.

4.3.1 Extended regions and kind system

In contrast with traditional lexically scoped regions, which are allocated in a LIFO manner, our *extended* regions can be allocated at any extended region ancestor. We consider the main heap ($`H$) as the root of our region hierarchy. We have already shown a number of examples using extended regions in the previous two chapters. This form of allocation generalizes the stack-based region organization to a tree-based organization and enables finer-grained control of region lifetimes. As in the operational semantics, extended regions are *sharable*, but no synchronization is required for accessing its data, as an extended region is initially *accessible* to the thread allocating it.

In particular, our implementation draws a line between our extended regions and traditional Cyclone regions. In this way, we are able to restrict what *kinds* of regions can be shared. Traditional lexically scoped regions cannot be shared safely. For instance, the *stack frame* of a function is treated as a region and sharing the stack in a safe manner would have a severe impact on concurrency between threads. The type system of Cyclone uses *kinds* to group types. We therefore use two different kinds of region type variables: one for Traditional regions that cannot be shared among threads, and one for extended regions that are *sharable*.

4.3.2 Exceptions

Having static guarantees about the control flow of a program plays a crucial role in manual memory management. As mentioned in Section 4.2, in Cyclone the memory of tracked objects (e.g., dynamic regions) can only be safely reclaimed by the garbage collector.

Since we aim for a low-level language (cf. Section 2.1), we decided that the programmer should *always* be able to reclaim extended regions manually. Towards this goal, we have made it possible for the programmer to annotate Cyclone function declarations with uncaught exception names that may be thrown from a function's body.[1] In addition, exact knowledge of a function's control-flow graph is required to guarantee soundness. There exist three kinds of annotations for exceptions:

1. the `@throws(...)` enumerates all exceptions that may be thrown from a function body;
2. the `@nothrow` annotation is an abbreviation for `@throws()`; and
3. `@throwsany` acts as a wildcard for any exception that may be thrown. (This annotation is often useful for legacy library prototypes and code.)

[1] We have noticed that the implementation of Cyclone actually had a `@throws` clause but it is undocumented and not functioning.

The default annotation for functions is @throwsany. Exceptions may be thrown *explicitly* by the programmer or *implicitly* by the run-time system. Implicit exceptions arise in situations where:

- a *null* pointer is dereferenced;
- an *out of bounds* array access is performed;
- the run-time system has *insufficient memory* to fulfill an allocation request;
- a value *cannot be matched* against any of the available patterns.

The exception analysis takes into consideration both explicit and implicit exceptions and guaratees the definite deallocation of extended regions in case uncaught exceptions are raised.

4.3.3 Reentrant functions

Global data is implicitly shared by all threads and this may cause a data race. To preserve race freedom, we have constrained our language so that only extended regions can be shared between threads. (Although we allow reading global variables that are declared as constant.) Traditional Cyclone regions (or references) cannot be passed to threads.

To enforce this policy, we require that each explicitly spawned thread must be @re_entrant. A function annotated as @re_entrant yields access to global variables, the immortal heap, tracked objects and it can only invoke @re_entrant functions. Function main, is not @re_entrant. Global data and tracked objects can still be directly accessed by any non reentrant function invoked directly or indirectly by main. Therefore, sequential programs have full access to global data. Relaxing type checking so that tracked objects can be passed to threads, provided that these objects are consumed from the environment performing a spawn operation is left for future work.

4.3.4 Type polymorphism

Cyclone effects are not polymorphic. To allow the invocation of functions, which have polymorphic arguments (e.g., say $`a$), Cyclone programmers use the regions($`a$) operator. Its purpose is to defer effect checking until the function call is performed, where the calling environment must prove that all regions occurring in the type that instantiates $`a$ are present in the environment's effect:

void foo($`a$ ; regions($`a$));

In terms of extended regions, the regions operator would require that all regions occurring in $`a$ are *live* and *accessible* for the scope of the function call. However, this is beyond the scope of our type system. Furthermore, we cannot provably guarantee memory safety if this construct is used in the way described above. Therefore, the type checker disallows invalid uses of the regions operator.

This limitation could be improved in future work, but as a workaround we allow extended regions to interoperate with traditional regions. We explain this feature in the following subsection.

4.3.5 Interoperability with traditional regions

The distinction between traditional and extended regions may be limiting for programs that require both kinds of regions. To ameliorate the situation we introduce a language construct, which is similar to the alias and open constructs of Cyclone, that borrows a part (or a fraction) of an accessible extended region for a certain scope. Consider the following example:

```
{ region child @ parent;
    { region h = xopen(child);          // consume one write-lock capability
      ...
    }                                   // restore write-lock capability
    ...
}
```

The xopen construct *borrows* exactly one lock capability from the extended region $`child$ for the scope of the xopen construct. The type system requires that region $`child$ is *live* by the end of the xopen scope and creates a fresh logical region $`h$, which can be used as a traditional Cyclone region. It should be noted that $`child$ is *still* live and possibly accessible (if it has more than one lock capability) during the scope of xopen. On the downside, region $`child$ *must* remain locked for the scope of xopen.

4.3.6 Memory consistency

Our formal language semantics assumes a *sequentially consistent* memory model [Lamp79], which implies that concurrent read and write operations are viewed as an interleaving of *atomic* steps. Modern processors are implemented with much weaker memory consistency specifications, because sequential consistency restricts common compiler and hardware optimizations. Research on relaxed memory models [Ghar90, Adve90] has shown that *race-free* programs (i.e., programs where read and write operations to shared memory locations only occur within memory synchronization primitives) running on relaxed memory systems have a sequentially consistent view of memory operations.

Assuming that the compilation process preserves the original Cyclone code semantics, then we obtain *race-free* native code with sequential consistency guarantees. At the implementation level, we must guarantee that memory operations to extended regions cannot escape the scope of a "lock/unlock" primitive as locking operations *synchronize memory*. This situation may arise as a result of compiler optimizations such as *register promotion* [Boeh05]. We have taken the most conservative approach and require that extended region data objects are compiled down to C as `volatile`. According to the manual of GCC, which is invoked by the Cyclone compiler to generate native code, "*an implementation is free to reorder and combine volatile accesses which occur between sequence points, but cannot do so for accesses across a sequence point*" [Stal11, Section 6.40, page 357]. Our locking primitives introduce sequence points and thus the compilation process will not reorder volatile accesses in an unsafe manner.

4.4 Explicit annotations for Cyclone

In this section we discuss integration-specific details of the type system without inference to Cyclone such as the input/output effect annotations, hierarchy abstraction, the thread creation semantics and the encoding of capability-modifying operations as ordinary functions.

4.4.1 Traditional Cyclone effects

The *effect* system of Cyclone tracks the set of *accessible* regions at each program point. Functions are annotated with a *single* effect, which can be automatically inferred by calculating the union of region variables occurring in the types of function parameters. As mentioned, traditional regions cannot be deallocated once they have been added in a function's effect, as unrestricted region aliasing within an effect is admitted. Therefore, a function effect serves as both a precondition and a postcondition of the regions that are accessible before and after calling a function respectively.

The following example illustrates a function, which has been annotated explicitly with the effect $\{`r\}$. This effect implies that region $`r$ is both *live* and *accessible* for the entire scope of foo.

`void` *foo* (`region_t` $<$ `‘r` $>$; $\{`r\}$);

We have decided to place our effects in separate annotations as full effect inference for our regions is beyond the scope of this work and we wish to preserve backwards compatibility with traditional Cyclone programs that enjoy full inference.

Our effects are mutually exclusive with traditional effects so a region name may not exist in both effects. We expect that future implementations will integrate the two different kinds of effects into a single effect and will enjoy effect inference for both traditional and extended regions.

The following example shows how we can write function *foo* so that it uses extended regions:

$$\texttt{void}\,\textit{foo}\ (\ \texttt{region\_t} < \text{`}\texttt{r} > )\ @\texttt{ieffect}(\{\text{`}r, \texttt{i}(1,1), \text{`}H\})\ @\texttt{oeffect}(\{\text{`}r, \texttt{i}(1,1), \text{`}H\});$$

The @ieffect and @oeffect annotations denote the input and output effects of function *foo*. These effects consist solely of extended regions or $\text{`}H$.[2] The equivalent type of function *foo* in our formal type system would be:

$$\forall \text{`}r.\, \texttt{region\_t} < \text{`}r > \xrightarrow{\gamma \to \gamma} \langle\rangle \quad \text{where } \gamma \equiv \text{`}r^{\overline{1,1}} \triangleright \bot$$

The heap region is mapped to $\bot$ as it is immortal. Impure capabilities $\overline{n_1, n_2}$ are denoted by $\texttt{i}(n_1, n_2)$, whereas pure capabilities n_1, n_2 are denoted by $\texttt{p}(n_1, n_2)$. Pure capabilities are most useful when transferring lock capabilities to other threads. It is therefore expected that impure capabilities would be the common case. Therefore, the above definition can be abbreviated as follows:

$$\texttt{void}\,\textit{foo}(\ \texttt{region\_t} < \text{`}\texttt{r} > )\ @\texttt{ieffect}(\{\text{`}r, 1, 1, \text{`}H\})\ @\texttt{oeffect}(\{\text{`}r, 1, 1, \text{`}H\});$$

Finally, it is possible to omit the output effect annotation when a function consumes the regions declared at the input effect.

4.4.2 Hierarchy abstraction

In order to allow a function to access a region without having to pass all its ancestors explicitly, its ancestors can be abstracted from an effect for the duration of a function call. To maintain soundness, we require that abstracted parents are *live* before and after the call. Regions whose parent information has been abstracted cannot be passed to a new thread as this may be unsound. The definition of *foo* can be further simplified, by using hierarchy abstraction:

$$\texttt{void}\,\textit{foo}\ (\ \texttt{region\_t} < \text{`}\texttt{r} > )\ @\texttt{ieffect}(\{\text{`}r, 1, 1\})\ @\texttt{oeffect}(\{\text{`}r, 1, 1\});$$

4.4.3 Operating on capabilities

The cap operator of the formal semantics has been encoded as a set of library functions:

$$\begin{array}{ll}
\texttt{void}\,\textit{xdec}\ (\ \texttt{region_t} <\text{`}r>) & @\texttt{ieffect}(\{\text{`}r, 1, 0\}); \\
\texttt{void}\,\textit{xinc}\ (\ \texttt{region_t} <\text{`}r>) & @\texttt{ieffect}(\{\text{`}r, 1, 0\}) \\
 & @\texttt{oeffect}(\{\text{`}r, 2, 0\}); \\
\texttt{void}\,\textit{xldec}\ (\ \texttt{region_t} <\text{`}r>) & @\texttt{ieffect}(\{\text{`}r, 1, 1\}) \\
 & @\texttt{oeffect}(\{\text{`}r, 1, 0\}); \\
\texttt{void}\,\textit{xlinc}\ (\texttt{region_t} <\text{`}r>) & @\texttt{ieffect}(\{\text{`}r, 1, 0\}) \\
 & @\texttt{oeffect}(\{\text{`}r, 1, 1\});
\end{array}$$

For instance, the first function *xdec* encodes the operator $\texttt{cap}_{\texttt{rg}-}$ for any region $\text{`}r$. It requires that the calling context has at least one region capability. This invariant is encoded in its input effect. The output effect of *xdec* is empty, thus exactly one region capability is consumed. Similarly the remaining functions encode the remaining functionality of operator cap. It would be preferable to use dependent types to allow these functions to increment or decrement counts by more than one. To the best of our knowledge this is impossible to express at the type level, in Cyclone's type system. However, we plan on extending the type-level expressiveness in future versions.

[2] $\text{`}H$ can only occur as a parent annotation.

4.4.4 Thread creation

Threads can be explicitly created by the means of the `spawn` operator. This operator takes two expressions e_1 and e_2, i.e., `spawn` (e_1) e_2, and spawns a new thread. The first expression is a list of thread-specific parameters such as the stack size. The second expression e_2 must be a function call and the function must be annotated as @`re_entrant`@`nothrow` and its @`oeffect` annotation must be either empty or omitted. Furthermore, the traditional Cyclone effect must be *empty* so that unsharable regions cannot be used in the new thread. Both expressions e_1 and e_2 are evaluated from left to right. The spawning thread does not block and returns immediately.

4.5 Annotation inference for Cyclone

In this section we discuss integration-specific details of the type system with inference to Cyclone such as the *cap* operator and the thread creation semantics.

Operating on capabilities. The `cap` operator of the formal semantics has been implemented as is in our extension of Cyclone. Moreover, it is possible to encode various primitives using the macro preprocessor of Cyclone, as is shown in the following code excerpt.

```
#define rfree(h) cap (−1, 0, 0) (h)
region<‘r> h @ H;
    let z = rnew(h) 42;
        ...
        ∗z = ∗z + 17 ;
        ...
        rfree(h);
        ...
```

Extended region functions. To speed up the compilation process, we require that functions which modify or access in any way extended regions must be explicitly declared as @`xrgn`.

Thread creation. Threads can be explicitly created by the means of the `spawn` operator. This operator takes two expressions e_1 and e_2, i.e., `spawn` (e_1) e_2, and spawns a new thread. The first expression is a list of tuples of the form (h, n_1, n_2, n_3), where h is a region handle, and n_1, n_2 and n_3 represent the region counts passed to the new thread. The second expression e_2 must be a function call and the function must be annotated as @`re_entrant`@`nothrow`. Furthermore, the traditional Cyclone effect must be *empty* so that unsharable regions cannot be used in the new thread. Both expressions e_1 and e_2 are evaluated from left to right. The spawning thread does not block and returns immediately.

Example. To clarify the above features of extended Cyclone we provide the example of Figure 4.1. Lines $1 - 6$ are standard C macros and the statement at line 7 includes the namespace of `Core` library to the current namespace. The entry point of our program is at line 30, where function `main` is declared. The first two declarations in the body of `main` allocate two regions ‘`parent` and ‘`child` respectively so that ‘`child` is allocated within ‘`parent` and ‘`parent` is allocated within the heap region (‘`heap_region`). Lines $34 - 46$ define a $try/catch$ block that handles a possible memory allocation exception that may be implicitly thrown at line 34. The exception handler deallocates the entire hierarchy (line 44) by deallocating region ‘`parent`. The program would be rejected if line 44 were omitted as both extended regions are alive at that point. The fresh reference at line 34 is allocated in region ‘`child`, initialized to the value 25 and assigned to the stack variable `ref`. The next two lines yield access to regions ‘`parent` and ‘`child`. Lines 37-40 create reader threads that execute function `reader`, which takes that handle of region ‘`child`, the reference `ref` and a unique identifier i. At each iteration,

the region count of ‘child is incremented by one (line 38) and is then passed to the thread reader. Once all reader threads are created, the main thread calls function writer, which takes that handle of region ‘parent and the reference ref and terminates. Both reader and writer threads perform a fixed number of iterations and at each iteration each thread sleeps for a fixed interval so that we can obtain some interesting interleavings. Reader and writer threads release regions ‘child and ‘writer thread when they terminate. The actual regions are deallocated when their reference counts reach the value zero. The writer thread acquires and releases the writer lock on ‘parent region at lines 23 and 25 respectively and writes the iteraction counter i to the shared location ref. Notice, that the writer thread does not have to explicitly lock region ‘child as the write access on a region implies write access on its subregions. The reader thread acquires and releases a reader lock on ‘child region at lines 13 and 15 respectively and prints to the standarding error the unique thread identifier and the current value of location ref.

```
1  #include <core.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #define MONITOR_TIME 500000000
5  #define READERS 2
6  #define ITERATIONS 10000
7  using Core;
8
9  void reader( region_t<‘r0::X> h, int @‘r::X ref,int tid)
10    @xrgn @nothrow @re_entrant  {
11       for(int i=0;i<ITERATIONS;i++) {
12          nsleep(MONITOR_TIME);
13          cap(h,0,0,1); // acquire read lock
14          fprintf(stderr,"\n[%d] Reading ref : %d",tid, *ref);
15          cap(h,0,0,-1); // release read lock
16       }
17       cap(h,-1,0,0);
18    }
19
20 void writer(region_t<‘r::X> h, int @‘r0::X ref) @xrgn {
21    for(int i=0;i<ITERATIONS;i++) {
22       nsleep(MONITOR_TIME);
23       cap(h,0,1,0);
24       *ref = i;
25       cap(h,0,-1,0);
26    }
27    cap(h,-1,0,0);
28 }
29
30 int main(int argc, char *@fat *@fat argv) {
31    region parent @ heap_region;
32    region child @ parent;
33    try{
34       let ref = rnew(child) 25; // child is thread-local
35       cap(parent,0,-1,0); //yield write access to parent
36       cap(child,0,-1,0); // yield write access right to child
37       for(int i=0;i<READERS;i++) {
38          cap(child,1,0,0);
39          spawn ($(child,1,0,0)) reader(child,ref,i);
40       }
41       writer(parent,ref);
42    } catch {
43       case &Core::Failure(s):
44          cap(parent,-1,0,0);
45          return -1;
46    }
47    return 0;
48 }
```

Figure 4.1: A simple extended Cyclone program with reader/writer locks.

4.6 Implementation

Even though the analyses of the two type systems differ, the compiler implementation (Section 4.6.1) is similar for both implementations. The code generation pass and the run-time system implementation diverge significantly for each of the two implementations and are described separately in sections 4.7.1 and 4.7.2 respectively.

4.6.1 Compiler

We have implemented extended region checking as a separate compiler pass in Cyclone. First, the type well-formedness of our annotations (effects, exceptions, types) is checked. During type checking, we disregard control-flow and verify that the extended regions being accessed exist in a function's scope. This allows us to catch common errors early. Once type checking is finished, the compiler enters the static analysis stage where it performs data- and control-flow analyses and determines candidate program locations for dynamic check (e.g., array bounds checks) insertion.

The compiler may eliminate some candidate locations, by utilizing programmer-inserted checks (e.g, `if` $(i < \mathit{len})\ a[i] = 42$;). As illustrated in Section 4.2, some optimizations may be unsound. Ideally, the data-flow analysis should discard programmer-inserted assertions for memory accesses at some region r, when they are followed by an unlock operation on r. Our current implementation, is highly conservative and only allows dynamic check elimination for trivial cases of shared memory accesses.

The exception analysis considers compiler-inserted checks as *implicit* exceptions and performs a control-flow sensitive analysis to verify that uncaught exceptions that *may* be thrown from a function body are included in the function's @`throws` specification.

Finally, a control-flow sensitive effect analysis is performed. The analysis propagates effects through the control flow graph and performs the following actions in each of the two implementations respectively:

- **Explicit annotations**: it verifies that the output effect of the function body matches the function's @`oeffect` specification. Capability validation is performed on the immediately when encountering operations that modify capabilities.

- **Annotation inference**: appends new effects when appropriate to the overall effect without any additional checks. Effect *translation* and *validation* is performed when the entire effect of a region has been gathered.

It is worth noting that iterative and exception handling statements are treated *conservatively*. It is entirely possible to spawn a new thread, which consumes some locks or regions, in one of the branches of a conditional statement and not in the other. Of course, both branches must have the same overall capability counts for each region. For instance, the branch not containing the spawn operation could have a $xdec$ operation (or a cap operation in the case of the implementation with inference) so as to guarantee that the output effects of both branches match.

This analysis also utilizes function attributes, when checking function calls. For instance, effects are not propagated from function calls that never return to the calling context (i.e., annotated with `__attribute((noreturn))__`).

4.7 Code generation and run-time system

In the following subsections we discuss the code generation pass and the implementation of the run-time system for the implementation with explicit annotations and the implementation with inference respectively.

4.7.1 Implementation with explicit annotations

As discussed in Section 2.4, each thread maintains a local view of the hierarchy and each function call accesses a specific portion of the thread's view. This is a crucial feature for avoiding false region sharing, which in turn reduces parallelism, and memory leaks caused by bulk region deallocation.

Consider the case where a thread owning an unlocked region ρ *shares* that region with a new thread, which in turn allocates a fresh region ρ_1 at region ρ. The second thread uses ρ_1 locally and then deallocates it. Assuming that there is a global view of the hierarchy, ρ_1 is *owned* by the second thread. If the first thread attempts to lock ρ, then it will have to block until ρ_1 is deallocated by the second thread.

Deallocating regions *en masse* may temporarily cause a *region leak*, when there is an inexact correspondence between the dynamic hierarchy and the static hierarchy accessible to a function call. Let us assume that function calls access dynamically a single (local) view of the hierarchy, but each call views *statically* a specific *portion* of the hierarchy (including region and lock counts). At the type-level, when a non-leaf node of a hierarchy is removed from a function's effect, then the entire subtree of that node is also removed from the effect. The dynamic semantics can *only* decrement the count of the node being removed, but it cannot decrement the remaining nodes as it is unaware of the portion being removed. Thus, the subtree would temporarily leak until the non-leaf node is entirely removed. If the node being removed is pure, then it is safe to deallocate its subtree from the local hierarchy without requiring additional information. Otherwise, the compiler has two options: issue a warning about a possible leak or generate code that dynamically tracks the hierarchy passed to a function call. That is, the compiler could preserve an exact correspondence between run-time views and static effects. The advantage of the second approach is that it prevents temporary memory leaks. On the downside, it places an overhead for each function call that uses non-trivial hierarchies.

The current implementation strictly adheres to the formal semantics and implements the second option. As an optimization, we avoid code generation for function calls that use hierarchies of height one. However, it is entirely possible to allow the programmer to decide whether such leaks should be prevented, by adding annotations to functions (e.g., `@noleak`), or by introducing new compiler flags.

In the paragraphs that follow we discuss how the code generator assists the run-time system with type information so that it can prevent *false sharing* and *region leaks*. We also discuss about new features that have been added to the run-time system.

Code generation. We have altered the code generation pass so that we can perform the following tasks:

- Translate `spawn` statements to low-level primitives, which require (un)packing of function arguments and placing the call into a wrapper function, which acts as a glue between the call and thread that will execute it.

- Generate specialized code for allocating extended regions and references.

- Generate code for allowing "dynamic effect tracking" before *some* function calls. The sub-tree passed to a call is not actually copied. Instead we use a form of dynamic scoping (shallow binding in particular) so as to map type-level region names to nodes of the local tree. Each dynamic scope is pushed into the virtual stack frame of the run-time system. An additional `pop` statement is added after the call.

Run-time system. In order to maintain a local view of the global hierarchy, the run-time system performs the following tasks:

- It registers fresh regions to the local thread hierarchy in which they are allocated.

– When a subtree has to be deallocated, it uses the dynamic scoping structures to retrieve nodes of the local hierarchy and update their dynamic counts accordingly. Notice that all remaining region locks are released during the deallocation phase.

– The implementation of `spawn` uses a similar technique to construct the subtree passed to new thread and makes this tree accessible to the new thread. It also performs capability accounting tasks so that the dynamic trees of both threads match the static effects.

– Region locking is implemented in a straightforward manner by traversing the local hierarchy. To avoid deadlocks, subtrees are always locked in a top-down left-to-right manner.

– The region allocation subsystem has been re-engineered so that it can serve concurrent allocation requests in a non-blocking manner (i.e., using atomic operations).

4.7.2 Implementation with inference

Code generation. We have altered the code generation pass so that we can perform the following tasks:

– Translate `spawn` statements to low-level primitives, which require packing and unpacking of function arguments and placing the call into a wrapper function, which acts as a glue between the call and thread that will execute it.

– Generate specialized code for allocating extended regions and references.

Run-time system. In order to maintain a local view of the global hierarchy, the run-time system performs the following tasks:

– It registers fresh regions to the local thread hierarchy in which they are allocated.

– When a subtree has to be deallocated, the appropriate region counts of the local hierarchy are updated. Notice that all remaining region locks are released during the deallocation phase.

– The implementation of `spawn` uses a similar process based on the region count annotations of `spawn` operation to construct the subtree passed to new thread and makes this tree accessible to the new thread.

– Region locking is implemented in a straightforward manner by traversing the local hierarchy. To avoid deadlocks, subtrees are always locked in a top-down left-to-right manner.

– The region allocation subsystem has been re-engineered so that it can serve concurrent allocation requests in a non-blocking manner (i.e., using atomic operations).

4.8 Performance evaluation

We evaluated our implementation on concurrent benchmark programs taken from "The Computer Language Benchmarks Game."[3] As a basis for our evaluation we tried to use the fastest version of the programs in C, which we translated by hand to extended Cyclone as directly as possible. In case this was not possible (e.g., the programs could not be translated) we either used a slower version in C or (if no such version was available from the shootout) we picked a concurrent solution written in a different language (e.g., C#) and translated it both to C and to our language as directly as possible. In all cases, the two programs that we are comparing implement the same algorithm. The seven benchmark programs we used were:[4]

[3] In June 2011, its URL is `http://shootout.alioth.debian.org/u32q/`.

[4] Our implementation and the benchmark programs we used in this section are available from the URL: `http://www.softlab.ntua.gr/~pgerakios/cycinfer.tgz`.

binary-trees: a program that allocates, traverses and deallocates binary trees. The original program (#7) uses GCC's OpenMP library and, for efficiency, memory pools as implemented in the Apache Portable Runtime Library.

chameneos-redux: a program that simulates the interaction of a number of creatures, using symmetrical thread rendez-vous. Our basis for the comparison is the second fastest version in C (#2); it uses pthreads and mutex locks. The fastest version in C (#5) uses the processor's "compare and swap" instruction, instead of locks, and explicitly schedules threads to processor cores; it cannot be translated directly to our language. Besides, on our testing machine, it only produced the correct result when compiled with `-O2`.

fannkuch-redux: a program that performs indexed access to small sequences of integer numbers. The original program (#2) uses pthreads. However, on our testing machine, it only produced the correct result when compiled without optimizations and, for fairness, we did the same for our program. Since November 2009, this program has been removed from the shootout; there is currently no multithreaded solution in C for fannkuch-redux.

mandelbrot: a program that plots a bitmap of the Mandelbrot set. The basis of our comparison was the fastest C solution in November 2009 (#6); which uses pthreads and special SSE2 128-bit floating-point instructions. However, because SSE2 operations are not available in Cyclone, for fairness, we used the same algorithm but with normal double precision numbers. Since November 2009, two faster C solutions have appeared in the shootout: they both use atomic builtins for synchronization and the first (#4) uses OpenMP while the second (#3) uses pthreads.

regex-dna: a program that records the frequencies of DNA patterns, expressed as regular expressions. The input is provided by a file containing numerous DNA sequences, which are placed in a read-only array. The patterns are distributed to worker threads, which *simultaneously* access the read-only array. The basis of our comparison is the C# solution (#6). We should mention that the fastest C solution (#1), using a different algorithm which distributes workload dynamically, performed a bit slower than both translations of the C# version (in C and in Cyclone) for our 50MB input test.

spectral-norm: a program that calculates the spectral norm of an infinite matrix. The algorithm is based on iterative parallelism, where threads are synchronized at each loop with the use of barriers. Two vectors are used for storing intermediate results. At each loop iteration, the vectors become read-only so that they can be *simultaneously* accessed by all threads involved in the computation. The fastest C solution (#4) uses pthreads and special SSE2 128-bit floating-point instructions and, again, for fairness we used the same algorithm but with normal double precision numbers.

thread-ring: a program that creates a large number of threads, organized in a ring, and repeatedly passes a token from one thread to the next. The original program (#1) uses pthreads and mutex locks. (We should mention that at the time of this writing, the original C program performs very poorly, compared to versions in other languages.)

The testing machine we used is a quad-core 2.5GHz Intel (Q8300), with 4GB of RAM and 2x2MB of L2 cache, running a Linux 2.6.26-2 kernel. When running the benchmarks, the testing machine was in single-user mode and, besides the operating system, the only program running was the "bencher" program, which we took from the web site of the "Computer Language Benchmarks Game." The bencher program does repeated measurements (10 times) of program CPU time, elapsed time, resident memory usage, CPU load while the benchmark is running, and summarizes those measurements. Our implementation used GCC 4.3.2 as a back end, which was also used to compile the C programs. We used `-O3` (except for fannkuch-redux, as explained above). In our Cyclone implementation we disabled

benchmark	lang	CPU	memory	load per core (%)				elapsed	factor
binary-trees	c	13.281	100648	0	87	97	81	5.383	1.00
	cyc	15.725	121880	80	78	85	85	4.966	0.92
chameneos-redux	c	28.774	560	76	73	52	89	12.946	1.00
	cyc	241.679	720	88	87	78	85	73.199	5.65
fannkuch-redux	c	139.989	552	99	100	99	98	35.275	1.00
	cyc	171.499	716	100	99	100	100	42.947	1.22
mandelbrot	c	37.914	31868	81	100	85	97	10.485	1.00
	cyc	37.694	31924	99	83	84	99	10.354	0.99
regex-dna	c	6.500	832540	76	95	65	77	2.043	1.00
	cyc	6.568	832576	69	75	93	72	2.112	1.03
spectral-norm	c	10.609	616	100	99	98	99	2.686	1.00
	cyc	9.761	680	99	99	99	99	2.745	1.02
thread-ring	c	179.479	4520	8	42	5	40	121.565	1.00
	cyc	350.778	4748	44	45	1	5	188.443	1.55

Table 4.1: Performance overhead, compared to GCC, for benchmarks taken from "The Computer Language Benchmarks Game." All times are in seconds and memory sizes in KB.

the use of Boehm's garbage collector, which is only used for Cyclone's original regions and is not need for these benchmarks.

The results are summarized in Table 4.1. CPU and elapsed times are in seconds; memory is in KB. As shown in the table the benchmark programs fall in two categories. In the first category one finds Cyclone programs with approximately the same performance as the original C program. Programs in this category are: spectral-norm (2% slower), mandelbrot (1% faster), regex-dna (3% slower), and binary-trees (8% faster). In the case of regex-dna and spectral-norm we managed to achieve similar performance to the C/pthreads program by employing reader locks for read-only arrays. The case of binary-trees is particularly interesting as the two programs use the same algorithm and the original C program also uses a region-based memory management scheme (memory pools, implemented by the Apache Portable Runtime Library).

In the second category one finds programs that run slower in Cyclone compared to the ones in C: fannkuch-redex (22% slower), thread-ring (55% slower) and chameneos-redux (465% slower). In the former two benchmarks the overhead can be attributed to the fact that our implementation of locks is not as optimized as the pthreads library for lock-intensive applications. (Unfortunately, we could not use the pthreads library for implementing our locks, as the pthreads specification does not support lock transfers between threads.) There is a very heavy performance penalty in chameneos-redux. The original program uses one lock for the meeting place, where the creatures meet. In addition to this lock, our program also uses a second lock for the entire array holding the creatures' data. In our Cyclone implementation, the array must be locked because it is not possible to convince the type system that the creature waiting in the meeting room will *never* access its data, but instead this data will be updated by its peer and therefore no data race will occur. The creatures' array must also be locked even when accessing certain "thread-local" fields of the creature structure. The performance penalty is mainly imposed by the second lock, which only allows one creature to make progress.

The overhead of the chameneos-redux program compared to the original C program reveals an inherent limitation of the granularity of locking supported by our type system. In Cyclone as well as in all region-based languages where regions annotate types, each array must reside in a single region whose name is present in the array's type. Therefore one cannot have an array whose elements reside in different regions. For our system, this means that concurrent access to array elements is necessarily coarse: a thread has to acquire the lock corresponding to the region, therefore locking the whole array for writing or reading. In other words, it is impossible to have two different threads writing concurrently to different parts of the same array. This limitation could be lifted by introducing existential types over regions, but this is technically quite involved and is a topic for future work.

benchmark	total (c)	total (cyc)	statements (cyc)	annotations (cyc)
binary-trees	129	183	12	14
chameneos-redux	301	333	16	30
fannkuch-redux	173	257	7	19
mandelbrot	169	232	6	13
regex-dna	306	417	6	11
spectral-norm	238	307	10	29
thread-ring	75	103	9	12

Table 4.2: Total lines of code, extended Cyclone statements and annotations compared to C, for benchmarks taken from "The Computer Language Benchmarks Game."

In terms of memory consumption, our implementation uses more or less the same amount of memory as the original programs. The only benchmark with noticeable difference in memory is binary-trees. Notice however that the implementation technology behind the two programs here is different: the original program uses OpenMP, whereas our program uses pthreads. These two implementation technologies cannot be directly compared.

Syntactic overhead The safety guarantees provided by extended Cyclone's hierarchical region system with reader/writer locks require the use of new programming constructs and annotations as discussed in earlier sections. We have assessed the syntactic overhead of writing programs in extended Cyclone compared to the corresponding C programs by counting the lines of code in each benchmark. For the extended Cyclone programs, we have also measured the number of lines that contain additional statements required by our type system (e.g., region sharing and locking primitives, etc.) or type annotations (e.g., when spawning new threads). The results are summarized in Table 4.2. Extended Cyclone programs have 34% more lines of code compared to C programs. However, only a small percentage of the additional lines of code is due to extended Cyclone features. In particular, the lines corresponding to additional statements and annotations account for 7.6% and 4.3% of the additional lines of code respectively.

Chapter 5

Effects for deadlock freedom

5.1 Overview

The possibility to run into a deadlock is an annoying and commonly occurring hazard associated with the concurrent execution of programs. This chapter serves as an introduction to type-based deadlock avoidance for languages with non block-structured locking primitives as a technique for guaranteeing deadlock freedom. In the section that follows, we briefly review existing type-based approaches to deadlock freedom We then explain why his approach cannot guarantee deadlock freedom in the presence of unstructured locking and describe informally how our approach manages to avoid deadlocks when unstructured locking is used. We also give the intuition behind the effect system employed to guarantee deadlock freedom.

5.2 Introduction

Lock-based synchronization may give rise to deadlocks. Two or more threads are deadlocked when each of them is waiting for a lock that is acquired by another thread. According to Coffman *et al.* [Coff71], a set of threads reaches a *deadlocked state* when the following conditions hold:

- *Mutual exclusion*: Threads claim exclusive control of the locks that they acquire.
- *Hold and wait*: Threads already holding locks may request (and wait for) new locks.
- *No preemption*: Locks cannot be forcibly removed from threads; they must be released explicitly by the thread that acquired them.
- *Circular wait*: Two or more threads form a circular chain, where each thread waits for a lock held by the next thread in the chain.

Coffman has identified three strategies that guarantee deadlock-freedom by denying at least one of the above conditions *before* or *during* program execution:

- *Deadlock prevention*: At each point of execution, *ensure* that at least one of the above conditions is not satisfied. Thus, programs that fall into this category are correct by design.
- *Deadlock detection and recovery*: A dedicated observer thread *determines* whether the above conditions are satisfied and preempts some of the deadlocked threads, releasing (some of) their locks, so that the remaining threads can make progress.
- *Deadlock avoidance*: Using information that is computed in advance regarding thread resource allocation, *determine* whether granting a lock will bring the program to an *unsafe* state, i.e., a state which can result in deadlock, and only grant locks that lead to safe states.

Several type systems have been proposed that guarantee deadlock freedom, the majority of which is based on the first two strategies. In the deadlock prevention category, one finds type and effect systems that guarantee deadlock freedom by statically enforcing a global lock acquisition order that must be respected by all threads [Flan99b, Boya02, Koba06, Suen08, Vasc10]. Using a strict lock acquisition order is a constraint we want to avoid, as it unnecessarily rejects many correct programs. Our work follows the third strategy (deadlock avoidance). It is based on an idea put forward recently

```
let  f = λx. λy. λz.   lock_{y} x; x := x + 1;
                       lock_{z} y; y := y + x;
                       unlock x;
                       lock_∅ z;   z := z + y;
                       unlock z;
                       unlock y
in   f a a b
```

(a) before substitution

```
lock_{a} a;   a := a + 1;
lock_{b} a;   a := a + a;
unlock a;
lock_∅ b;     b := b + a;
unlock b;
unlock a
```

(b) after substitution

Figure 5.1: An example program, which is well typed before substitution (a) but not after (b).

by Boudol, who proposed a type system for deadlock avoidance that is more permissive than existing approaches [Boud09]. However, his system is suitable for programs that use *exclusively* lexically-scoped locking primitives. Our approach ensures deadlock freedom for the proposed language by preserving exact information about the order of events, both statically and dynamically. In the next section, we informally describe Boudol's idea and present an informal overview of our type and effect system.

5.3 Deadlock avoidance

Recently, Boudol developed a type and effect system for deadlock freedom [Boud09], which is based on *deadlock avoidance*. The effect system calculates for each expression the set of acquired locks and annotates lock operations with the "future" lockset. The runtime system utilizes the inserted annotations so that each lock operation can only proceed when its "future" lockset is unlocked. The main advantage of Boudol's type system is that it allows a larger class of programs to type check and thus increases the programming language expressiveness as well as concurrency by allowing arbitrary locking schemes.

The previous example can be rewritten in Boudol's language as follows, assuming that the only lock operations in the two threads are those visible:

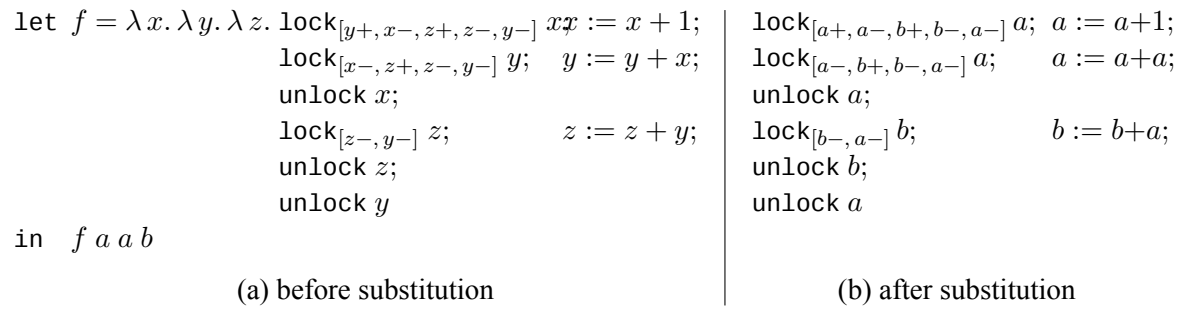
$$(\texttt{lock}_{\{y\}}\ x\ \texttt{in}\ \ldots\ \texttt{lock}_{\emptyset}\ y\ \texttt{in}\ \ldots) \ ||\ (\texttt{lock}_{\{x\}}\ y\ \texttt{in}\ \ldots\ \texttt{lock}_{\emptyset}\ x\ \texttt{in}\ \ldots)$$

This program is accepted by Boudol's type system which, in general, allows locks to be acquired in *any* order. At runtime, the first lock operation of the first thread must ensure that y has not been acquired by the second (or any other) thread, before granting x (and symmetrically for the second thread). The second lock operations need not ensure anything special, as the future locksets are empty.

The main disadvantage of Boudol's work is that locking operations have to be lexically-scoped. Even if his language had `lock`/`unlock` constructs, instead of `lock` ... `in` ..., Boudol's type system is not sufficient to guarantee deadlock freedom. The example program in Figure 5.1(a) will help us see why: It updates the values of three shared variables, x, y and z, making sure at each step that only the strictly necessary locks are held.[1]

In our naïvely extended (and broken, as will be shown) version of Boudol's type and effect system, the program in Figure 5.1(a) will type check. The future lockset annotations of the three locking operations in the body of f are $\{y\}$, $\{z\}$ and $\emptyset$, respectively. (This can be easily verified by observing the lock operations between a specific `lock` and `unlock` pair.) Now, function f is used by instantiating both x and y with the same variable a, and instantiating z with a different variable b. The result of this substitution is shown in Figure 5.1(b). The first thing to notice is that, if we want this program to work in this case, locks have to be *re-entrant*. This roughly means that if a thread holds some lock, it can try to acquire the same lock again; this will immediately succeed, but then the thread will have to release the lock *twice*, before it is actually released.

[1] To simplify presentation, we assume here that there is one implicit lock per variable, which has the same name.



(a) before substitution (b) after substitution

Figure 5.2: The program of Figure 5.1 with continuation effect annotations; now well typed in both cases.

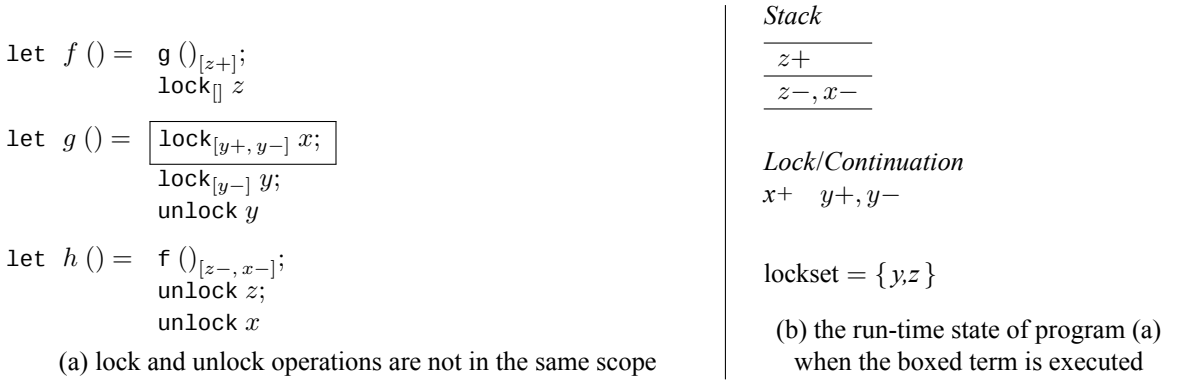
Even with re-entrant locks, however, it is easy to see that the program in Figure 5.1(b) does not type check with the present annotations. The first `lock` for a now matches with the *last* (and not the first) `unlock`; this means that a will remain locked during the whole execution of the program. In the meantime b is locked, so the future lockset annotation of the first `lock` should contain b, but it does not. (The annotation of the second `lock` contains b, but blocking there if lock b is not available does not prevent a possible deadlock; lock a has already been acquired.) So, the technical failure of our naïvely extended language is that the preservation lemma breaks. From a more pragmatic point of view, if a thread running in parallel already holds b and, before releasing it, is about to acquire a, a deadlock can occur. The naïve extension also fails for another reason: Boudol's system is based on the assumption that calling a function cannot affect the set of locks that are held. This is obviously not true, if non lexically-scoped locking operations are to be supported.

The type and effect system proposed in this chapter supports unstructured locking, by preserving more information at the effect level. Instead of treating effects as unordered collections of locks, our type system precisely tracks effects as an order of `lock` and `unlock` operations, without enforcing a strict lock-acquisition order. The *continuation effect* of a term represents the effect of the function code succeeding that term. In our approach, lock operations are annotated with a continuation effect. When a `lock` operation is evaluated, the future lockset is calculated by inspecting its continuation effect. The `lock` operation succeeds only when both the lock and the future lockset are available.

Figure 5.2 illustrates the same program as in Figure 5.1, except that locking operations are now annotated with continuation effects. For example, the annotation $[y+,\, x-,\, z+,\, z-,\, y-]$ at the first `lock` operation means that in the future (i.e., after this lock operation) y will be acquired, then x will be released, and so on.[2] If x and y were different, the runtime system would deduce that between this `lock` operation on x and the corresponding `unlock` operation, only y is locked, so the future lockset in Boudol's sense would be $\{y\}$. On the other hand, if x and y are instantiated with the same a, the annotation becomes $[a+,\, a-,\, b+,\, b-,\, a-]$ and the future lockset that is calculated is now the correct $\{a, b\}$. In a real implementation, there are several optimizations that can be performed (e.g., pre-calculation of effects) but we do not deal with them in this chapter.

There are three issues that must be faced, before we can apply this approach to a full programming language. First, we need to consider continuation effects in an interprocedural manner as lock operations and their matching unlock operations may not occur in the same scope. Figure 5.3(a) illustrates a program where the first lock operation in the body of function g matches with the last unlock operation in the body of function h after the point where g is called. In this case, the future lockset of the first lock operation of h contains lock z that is not visible in the body of h. We choose to compute function effects intraprocedurally and to annotate each application term with a continuation effect, which represents the effect of the code succeeding the application term in the calling function's body, as shown in Figure 5.3(a). At each function call, the associated continuation effect is pushed on the

[2] In the examples of this section, a simplified version of effects is used, to make presentation easier. In the formalism of Section 6.2, the plus and minus signs would be encoded as differences in lock counts, e.g., $y+$ would be encoded by a $y^{1,0}$ (an unlocked y) followed in time by a $y^{1,1}$ (a locked y).



(a) lock and unlock operations are not in the same scope

(b) the run-time state of program (a) when the boxed term is executed

Figure 5.3: Iterprocedural effects and run-time lockset computation.

stack for the duration of the call. Figure 5.3(b) shows the run-time state of the program when control reaches the lock operation on x: the run-time stack (which grows upwards in the figure) contains the continuation effects of the `f` and `g` calls. The future lockset computation algorithm is straightforward given the continuation effect of x and the run-time stack: the algorithm starts with an empty future lockset and traverses the continuation effect until the matching $x-$ is found. While traversing the effect, other locations being locked are added to the future lockset. For instance, in Figure 5.3(b) the algorithm adds y to the future lockset of x and then considers the continuation effects on the stack, from top to bottom. Thus, z is added to the future lockset and the matching unlock operation is found on the next element of the stack. The resulting lockset is $\{y, z\}$. During program execution the lock operation unblocks when lockset $\{x, y, z\}$ is available. Notice that we grant only a *single lock* for each lock operation (only x in this case) as opposed to acquiring prematurely the entire lockset with it ($\{y, z\}$), which would damage the program's degree of parallelism.

Second, we need to support conditional statements. The tricky part here is that, even in a simple conditional statement such as

`if` c `then` (`lock` x; ... `unlock` x) `else` (`lock` y; ... `unlock` y)

the two branches have different effects: $[x+,\ x-]$ and $[y+,\ y-]$, respectively. A typical type and effect system would have to reject this program, but this would be very restrictive in our case. We resolve this issue by requiring that the *overall* effect of both alternatives is the same. This (very roughly) means that, after the plus and minus signs cancel each other out, we have equal numbers of plus or minus signs for each lock in both alternatives. Furthermore, we assign the *combined* effect of the two alternatives to the conditional statement, thus keeping track of the effect of both branches; in the example above, the combined effect is denoted by $[x+,\ x-]\,?\,[y+,\ y-]$.

The third and most complicated issue that we need to face is support for recursive functions.

Consider the following example:

$$\begin{array}{l}\texttt{letrec f} = \lambda\,\texttt{x}.\,\lambda\,\texttt{y}.\,\lambda\,\texttt{z}.\\ \quad \texttt{if z} > 0\ \texttt{then}\ (\texttt{lock}_{\gamma}\ \texttt{x};\ \texttt{f x y}\ (\texttt{z} - 1);\ \texttt{unlock x})\\ \qquad\qquad \texttt{else}\ (\texttt{lock}_{\emptyset}\ \texttt{y}; \ldots \texttt{unlock y};)\end{array}$$

In this case, if we employ the usual typing for `letrec`, the effect of f must equal the effect of its body. However, this is impossible, as the two effects cannot be structurally equivalent: in fact, the effect of f is *contained* in the effect of its body, due to the recursive call.

Let us briefly see how our system can infer the effect of function f in the example above. Suppose that γ_f is the (unknown) effect of f. Then, the effect of *the body* of f as a function of γ_f is expressed as

$$\gamma_b(\gamma_f) = ([x+]\,?\,\gamma_f\,?\,[x-])\,?\,[y+,\ y-]$$

where $\gamma_1\,?\,\gamma_2$ denotes the appending of two effects γ_1 and γ_2. We are looking for a solution to the equation

$$\gamma_f = \mathsf{summary}(\gamma_b(\gamma_f))$$

At this point, we can start with $\gamma_0 = \emptyset$ (noticing that function f has no unmatched lock or unlock operations) and look for the limit of the sequence $\gamma_{n+1} = \mathsf{summary}(\gamma_b(\gamma_n))$ in other words, for a fixed point of the summarized function's body. We have

$$\gamma_1 = \mathsf{summary}(\gamma_b(\gamma_0)) = \mathsf{summary}([x+,\, x-] \,?\, [y+,\, y-])$$

Although we have not formally defined what function summary does, a possible (but conservative) choice here would be to "merge" the effects of the two branches in the summary. Therefore,

$$\gamma_1 = [x+,\, x-,\, y+,\, y-]$$

We can then proceed in the same way and take

$$\gamma_2 = \mathsf{summary}([x+,\, x+,\, x-,\, y+,\, y-,\, x-] \,?\, [y+,\, y-])$$

If we are outside function f, we don't care if inside f lock x is taken more than once. Nor do we care if x is held or not, at the moment when y is taken. We are just happy to know that x and y are taken and released. Therefore, by merging again:

$$\gamma_2 = [x+,\, x-,\, y+,\, y-] = \gamma_1$$

We reached a fixed point and we can take γ_1 as the summarized effect of function f. Therefore, the effect γ in the annotation of the first lock operation in the example is equal to $[x+,\, x-,\, y+,\, y-,\, x-]$.

5.4 Concluding remarks

This chapter served as an introduction to type-based deadlock avoidance for languages with non block-structured locking primitives as a technique for guaranteeing deadlock freedom. We presented informally our effect system, defined the notion of "continuation effects" and discussed how different language features are treated using effects such lock operations, function calls, conditional expressions and recursion.

The following two chapters present two type systems for deadlock avoidance. The first type system is more complex as it provides additional guarantees for well-typed programs such as memory-safety and race freedom, but requires type annotations. Thus, it is more suitable for languages such as Cyclone. The second type system only focuses on deadlock freedom, but is able to infer all effects. This makes it suitable for multithreaded C programs which are usually hard to annotate. The third chapter following the present chapter discusses the analysis and the tool that we have developed for multithreaded C programs.

Chapter 6

Explicit effects for deadlock freedom

6.1 Overview

In this chapter we present a simple language with explicitly annotated functions, mutable references, explicit (de-)allocation constructs and unstructured locking primitives. We also present its operational semantics and a type system that ensures that well typed programs are *memory safe* (safe memory accesses and definite release of allocated memory), *race free* (and definite release of locks and absence of dangling unlock operations) and *deadlock free*. The chapter ends with the presentation of the main soundness results and a few concluding remarks.

6.2 Formalism

The syntax of our language is illustrated in Figure 6.1, where x and ρ range over term and "region" variables, respectively. A region is thought of as a memory unit that can be shared between threads and whose contents can be atomically locked. In this chapter, we make the simplistic assumption that there is a one-to-one correspondence between regions and memory cells (locations), but this is of course not necessary.

The language core comprises of variables (x), constants (the unit value, `true` and `false`), functions (f), and function application. Functions can be location polymorphic ($\Lambda\rho.\, f$) and location application is explicit ($e[\rho]$). Monomorphic functions ($\lambda x.\, e$) must be annotated with their type. The application of monomorphic functions is annotated with a *calling mode* (ξ), which is `seq` for normal (sequential) application and `par` for parallel application. Notice that sequential application terms are annotated with γ, the *continuation effect* as mentioned earlier. The semantics of parallel application is that once the application term is evaluated to a redex, then it is moved to a new thread of execution and the spawning thread can proceed with the remaining computation in parallel with the new thread. The term $\mathtt{pop}_\gamma\ e$ encloses a function body e and can only appear during evaluation. The same applies to constant locations $\imath@n$, which cannot exist at the source-level. The construct $\mathtt{let}\ \rho, x = \mathtt{ref}\ e_1\ \mathtt{in}\ e_2$ allocates a fresh cell, initializes it to e_1, and associates it with variables ρ and x within expression e_2. As in other approaches, we use ρ as the type-level representation of the new cell's location. The reference variable x has the singleton type $\mathtt{ref}(\rho, \tau)$, where τ is the type of the cell's contents. This allows the type system to connect x and ρ and thus to statically track uses of the new cell. As will be explained later, the cell can be consumed either by deallocation or by transferring its ownership to another thread. Assignment and dereference operators are standard. The value $\mathtt{loc}_\imath$ represents a reference to a location $\imath$ and is introduced during evaluation. Source programs cannot contain $\mathtt{loc}_\imath$.

At any given program point, each cell is associated with a *capability* (κ). Capabilities consist of two natural numbers, the *capability counts*: the *cell reference* count, which denotes whether the cell is live, and the *lock* count, which denotes whether the cell has been locked to provide the current thread with exclusive access to its contents. Capability counts determine the validity of operations on cells. When first allocated, a cell starts with capability $(1, 1)$, meaning that it is live and locked, which provides exclusive access to the thread which allocated it. (This is our equivalent of thread-local data.) Capabilities can be either *pure* (n_1, n_2) or *impure* $(\overline{n_1, n_2})$. In both cases, it is implied that the

$$
\begin{array}{lrcl}
\textbf{Expression} & e & ::= & x \mid f \mid (e\ e)^{\xi} \mid (e)\,[r] \mid e := e \\
& & \mid & \texttt{deref}\ e \mid \texttt{let}\ \rho, x = \texttt{ref}\ e\ \texttt{in}\ e \\
& & \mid & \texttt{share}\ e \mid \texttt{release}\ e \mid \texttt{lock}_{\gamma}\ e \\
& & \mid & \texttt{unlock}\ e \mid () \mid \texttt{pop}_{\gamma}\ e \mid \texttt{loc}_{\iota} \\
& & \mid & \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \mid \texttt{true} \mid \texttt{false} \\
\textbf{Value} & v & ::= & f \mid () \mid \texttt{loc}_{\iota} \mid \texttt{true} \mid \texttt{false} \\
\textbf{Function} & f & ::= & \lambda x.\, e\ \texttt{as}\ \tau \xrightarrow{\gamma} \tau \mid \Lambda\rho.\, f \mid \texttt{fix}\ x : \tau.\, f
\end{array}
$$

$$
\begin{array}{lrcl}
\textbf{Type} & \tau & ::= & \langle\rangle \mid \tau \xrightarrow{\gamma} \tau \mid \forall\rho.\, \tau \\
& & \mid & \texttt{ref}(\tau, r) \mid \texttt{bool} \\
\textbf{Location} & r & ::= & \rho \mid \iota@n \mid \rho@n \\
\textbf{Calling mode} & \xi & ::= & \mathsf{seq}(\gamma) \mid \mathsf{par} \\
\textbf{Capability} & \kappa & ::= & n, n \mid \overline{n, n} \\
\textbf{Effect} & \gamma & ::= & \emptyset \mid \gamma, r^{\kappa} \mid \gamma, \gamma\,?\,\gamma
\end{array}
$$

Figure 6.1: Language syntax.

$$
\begin{array}{lrcl}
\textbf{Configuration} & C & ::= & S; T \\
\textbf{Store} & S & ::= & \emptyset \mid S, \iota \mapsto v \\
\textbf{Threads} & T & ::= & \emptyset \mid T, n : \theta; e \\
\textbf{Locations} & \epsilon & ::= & \emptyset \mid \epsilon, \iota \\
\textbf{Access Lists} & \theta & ::= & \emptyset \mid \theta, \iota \mapsto n; n; \epsilon; \epsilon
\end{array}
$$

$$
\begin{array}{lrcl}
\textbf{Stack} & E & ::= & \Box \mid E[F] \\
\textbf{Frame} & F & ::= & (\Box\ e)^{\xi} \mid (v\ \Box)^{\xi} \mid (\Box)\,[r] \\
& & \mid & \texttt{let}\ \rho, x = \texttt{ref}\ \Box\ \texttt{in}\ e \\
& & \mid & \texttt{deref}\ \Box \mid \Box := e \mid v := \Box \\
& & \mid & \texttt{share}\ \Box \mid \texttt{release}\ \Box \\
& & \mid & \texttt{lock}_{\gamma}\ \Box \mid \texttt{unlock}\ \Box \mid \texttt{pop}_{\gamma}\ \Box \\
& & \mid & \texttt{if}\ \Box\ \texttt{then}\ e_1\ \texttt{else}\ e_2
\end{array}
$$

Figure 6.2: Operational semantics, semantic domains.

current thread can decrement the cell reference count n_1 times and the lock count n_2 times. Similarly to *fractional permissions* [Boyl03], impure capabilities denote that a location may be aliased. Our type system requires aliasing information so as to determine whether it is safe to pass lock capabilities to new threads.

The remaining language constructs (`share` e, `release` e, $\texttt{lock}_{\gamma}\ e$ and `unlock` e) operate on a reference e. The first two constructs *increment* and *decrement* the cell reference count of e respectively. Similarly, the latter two constructs *increment* and *decrement* the lock count of e. As mentioned earlier, the runtime system inspects the lock annotation γ to determine whether it is safe to lock e.

6.3 Operational semantics

We define a *small-step* operational semantics for our language in Figure 6.3 and Figure 6.2.[1] The evaluation relation transforms *configurations*. A configuration C consists of an abstract *store* S and a thread map T.[2] A store S maps constant locations (ι) to values (v). A thread map T associates thread identifiers to expressions (i.e., threads) and access lists. An *access list* θ maps location identifiers to *reference* and *lock* counts.

A *frame* F is an expression with a *hole*, represented as $\Box$. The hole indicates the position where the next reduction step can take place. A *thread evaluation context* E, is defined as a stack of nested frames. Our notion of evaluation context imposes a call-by-value evaluation strategy to our language. Subexpressions are evaluated in a left-to-right order. We assume that concurrent reduction events can be totally ordered [Lamp79]. At each step, a *random* thread (n) is chosen from the thread list for evaluation. Therefore, the evaluation rules are *non-deterministic*.

When a parallel function application redex is detected within the evaluation context of a thread, a new thread is created (rule *E-SN*). The redex is replaced with a unit value in the currently executed thread and a new thread is added to the thread list, with a *fresh* thread identifier. The calling mode of

[1] Some of the functions and judgements that are used by the operational and (later) the static semantics are not formally defined in this chapter. Verbal descriptions and a full formalization are given in Appendix C.

[2] The order of elements in comma-separated lists, e.g., in a store S or in a list of threads T, is unimportant; we consider all list permutations as equivalent.

$$\frac{v' \equiv \lambda x.\, e_1 \text{ as } \tau_1 \xrightarrow{\gamma_a} \tau_2 \qquad \text{fresh } n' \qquad (\theta_1, \theta_2) = \mathsf{split}(\theta\ ,\ \mathsf{max}(\gamma_a))}{S; T, n:\theta; E[(v'\ \ v)^{\mathsf{par}}] \rightsquigarrow S; T, n:\theta_1; E[()], n':\theta_2; \square[(v'\ \ v)^{\mathsf{seq}(\mathsf{min}(\gamma_a))}]} \quad \textit{(E-SN)}$$

$$\frac{\forall \imath.\, \theta(\imath) = (0,0)}{S; T, n:\theta; ()\ \rightsquigarrow S; T} \quad \textit{(E-T)}$$

$$\frac{v' \equiv \lambda x.\, e_1 \text{ as } \tau'}{S; T, n:\theta; E[(v'\ \ v)^{\mathsf{seq}(\gamma_b)}]\ \rightsquigarrow S; T, n:\theta; E[\mathsf{pop}_{\gamma_b}\ e_1[v/x]]} \quad \textit{(E-A)}$$

$$\frac{}{S; T, n:\theta; E[\mathsf{pop}_{\gamma}\ v]\ \rightsquigarrow S; T, n:\theta; E[v]} \quad \textit{(E-PP)}$$

$$\frac{\text{fresh } n_2}{S; T, n:\theta; E[(\Lambda\rho.\, f)[\imath@n_1]]\ \rightsquigarrow S; T, n:\theta; E[f[\imath@n_2/\rho]]} \quad \textit{(E-RP)}$$

$$\frac{}{S; T, n:\theta; E[(\mathtt{fix}\ x:\tau.\, f\ \ v)^{\mathsf{seq}(\gamma_a)}]\ \rightsquigarrow S; T, n:\theta; E[(f[\mathtt{fix}\ x:\tau.\, f/x]\ \ v)^{\mathsf{seq}(\gamma_a)}]} \quad \textit{(E-FX)}$$

$$\frac{}{S; T, n:\theta; E[\mathtt{if\ true\ then}\ e_1\ \mathtt{else}\ e_2]\ \rightsquigarrow S; T, n:\theta; E[e_1]} \quad \textit{(E-IT)}$$

$$\frac{}{S; T, n:\theta; E[\mathtt{if\ false\ then}\ e_1\ \mathtt{else}\ e_2]\ \rightsquigarrow S; T, n:\theta; E[e_2]} \quad \textit{(E-IF)}$$

$$\frac{\text{fresh } \imath@n_1 \qquad S' = S, \imath \mapsto v \qquad \theta' = \theta, \imath \mapsto 1; 1; \emptyset; \emptyset}{S; T, n:\theta; E[\mathtt{let}\ \rho, x = \mathtt{ref}\ v\ \mathtt{in}\ e_2]\ \rightsquigarrow S'; T, n:\theta'; E[e_2[\imath@n_1/\rho][\mathtt{loc}_\imath/x]]} \quad \textit{(E-NG)}$$

$$\frac{\theta(\imath) \geq (1,1) \qquad \imath \notin \mathsf{locked}(T)}{S; T, n:\theta; E[\mathtt{loc}_\imath := v]\ \rightsquigarrow S[\imath \mapsto v]; T, n:\theta; E[()]} \quad \textit{(E-AS)}$$

$$\frac{\theta(\imath) \geq (1,1) \qquad \imath \notin \mathsf{locked}(T)}{S; T, n:\theta; E[\mathtt{deref}\ \mathtt{loc}_\imath]\ \rightsquigarrow S; T, n:\theta; E[S(\imath)]} \quad \textit{(E-D)}$$

$$\frac{\theta(\imath) \geq (1,0) \qquad \theta' = \theta\ +_\imath\ (1,0)}{S; T, n:\theta; E[\mathtt{share}\ \mathtt{loc}_\imath]\ \rightsquigarrow S; T, n:\theta'; E[()]} \quad \textit{(E-SH)}$$

$$\frac{\begin{array}{c}\theta(\imath) \geq (1,0) \qquad \theta(\imath) = (n_1, n_2) \\ n_1 = 1 \Rightarrow n_2 = 0 \qquad \theta' = \theta\ +_\imath\ (-1,0)\end{array}}{S; T, n:\theta; E[\mathtt{release}\ \mathtt{loc}_\imath]\ \rightsquigarrow S; T, n:\theta'; E[()]} \quad \textit{(E-RL)}$$

$$\frac{\begin{array}{c}\epsilon = \mathsf{lockset}(\imath, 1, E[\mathsf{pop}_{\gamma_1}\ \square]) \qquad \theta = \theta'', \imath \mapsto n_1; 0; \epsilon_1; \epsilon_2 \\ \theta' = \theta'', \imath \mapsto n_1; 1; \mathsf{dom}(S); \epsilon \qquad n_1 \geq 1 \qquad \mathsf{locked}(T) \cap \epsilon = \emptyset\end{array}}{S; T, n:\theta; E[\mathtt{lock}_{\gamma_1}\ \mathtt{loc}_\imath]\ \rightsquigarrow S; T, n:\theta'; E[()]} \quad \textit{(E-LK0)}$$

$$\frac{\theta(\imath) \geq (1,1) \qquad \theta' = \theta\ +_\imath\ (0,1)}{S; T, n:\theta; E[\mathtt{lock}_{\gamma_1}\ \mathtt{loc}_\imath]\ \rightsquigarrow S; T, n:\theta'; E[()]} \quad \textit{(E-LK1)}$$

$$\frac{\theta(\imath) \geq (1,1) \qquad \theta' = \theta\ +_\imath\ (0,-1)}{S; T, n:\theta; E[\mathtt{unlock}\ \mathtt{loc}_\imath]\ \rightsquigarrow S; T, n:\theta'; E[()]} \quad \textit{(E-UL)}$$

Figure 6.3: Operational semantics, reduction relation.

$$\mathsf{set}(\gamma) \quad = \quad \forall \alpha,\, \gamma_1,\, \gamma_2.\, \gamma = (\gamma_1, \alpha) :: \gamma_2 \Rightarrow \alpha = r^\kappa \,\wedge\, r \notin \mathsf{dom}(\gamma_1) \cup \mathsf{dom}(\gamma_2)$$

$$\frac{}{\gamma :: \emptyset = \gamma} \qquad \frac{\gamma :: \gamma' = \gamma''}{\gamma :: \gamma', r^\kappa = \gamma'', r^\kappa} \qquad \frac{\gamma :: \gamma' = \gamma''}{\gamma :: \gamma', (\gamma_1 \,?\, \gamma_2) = \gamma'', (\gamma_1 \,?\, \gamma_2)}$$

$$\frac{\mathsf{dom}(\gamma) = \epsilon}{\mathsf{dom}(\gamma, r^\kappa) = \epsilon \cup \{r\}} \qquad \frac{\mathsf{dom}(\gamma) = \epsilon \quad \mathsf{dom}(\gamma') = \epsilon' \quad \mathsf{dom}(\gamma'') = \epsilon''}{\mathsf{dom}(\gamma, \gamma' \,?\, \gamma'') = \epsilon \cup \epsilon' \cup \epsilon''} \qquad \frac{}{\mathsf{dom}(\emptyset) = \emptyset}$$

$$\frac{}{\mathsf{max}(\emptyset) = \emptyset} \qquad \frac{\gamma_2 = \{r'^\kappa \in \mathsf{max}(\gamma_1) \mid r' \neq r\}}{\mathsf{max}(\gamma_1, r^\kappa) = \gamma_2, r^\kappa} \qquad \frac{\gamma_4 = \mathsf{max}(\gamma_1 :: \gamma_2) = \mathsf{max}(\gamma_1 :: \gamma_3)}{\mathsf{max}(\gamma_1, \gamma_2 \,?\, \gamma_3) = \gamma_4}$$

$$\frac{\mathsf{set}(\gamma_1) \quad \gamma_2 = \gamma_1 :: \gamma_3 \quad \mathsf{dom}(\gamma_2) = \mathsf{dom}(\gamma_1)}{\mathsf{min}(\gamma_2) = \gamma_1} \qquad \frac{\gamma_1 = \gamma_2 :: \gamma_3}{\gamma_2 \lhd \gamma_1} \qquad \frac{\mathsf{max}(\gamma) = (\gamma_1, r^\kappa) :: \gamma_2}{\gamma(r) = \kappa}$$

$$\begin{array}{lcl}
\mathsf{locked}(T) & = & \{\, \imath \mid (n : \theta; e) \in T \,\wedge\, \theta(\imath) \geq (1,1)\} \\
\mathsf{lk}(\kappa) & = & n_2 \quad \text{if } \kappa = (n_1, n_2) \\
\theta \,+_\imath\, (n_1, n_2) & = & \theta[\imath \mapsto n_1 + n_3; n_2 + n_4; \epsilon_1; \epsilon_2] \quad \text{if } (\imath \mapsto n_3; n_4; \epsilon_1; \epsilon_2) \in \theta \\
\theta(\imath) & = & (n_1, n_2) \quad \text{if } (\imath \mapsto n_1; n_2; \epsilon_1; \epsilon_2) \in \theta
\end{array}$$

$$\frac{\kappa = (n_1, n_2) \qquad (\theta_a, \theta_b) = \mathsf{split}(\theta\,,\, \gamma) \qquad (\theta_c, \theta_d) = (\theta_a \,+_\imath\, (-n_1, -n_2)\,,\, \theta_b \,+_\imath\, (n_1, n_2))}{(\theta_c, \theta_d) = \mathsf{split}(\theta\,,\, \gamma, (\imath @ n_0)^\kappa)} \;\textit{(A1)} \qquad \frac{}{(\theta, \emptyset) = \mathsf{split}(\theta\,,\, \emptyset)} \;\textit{(A2)}$$

$$\frac{}{(\emptyset, n) = \mathsf{frame\_lockset}(\imath, n,\, \emptyset)} \;\textit{(W0)} \qquad \frac{}{(\emptyset, 0) = \mathsf{frame\_lockset}(\imath, 0,\, \gamma)} \;\textit{(W1)}$$

$$\frac{n_1 > 0 \qquad n_2 = \mathsf{lk}(\kappa) - \mathsf{lk}(\gamma(\imath @ n_0)) \qquad (\epsilon, n_3) = \mathsf{frame\_lockset}(\imath, n_1 + n_2,\, \gamma)}{(\epsilon, n_3) = \mathsf{frame\_lockset}(\imath, n_1,\, \gamma, (\imath @ n_0)^\kappa)} \;\textit{(W2)}$$

$$\frac{n_1 > 0 \qquad (\epsilon, n_2) = \mathsf{frame\_lockset}(\imath, n_1,\, \gamma_1 :: \gamma_2) \qquad (\epsilon', n_2) = \mathsf{frame\_lockset}(\imath, n_1,\, \gamma_1 :: \gamma_3)}{(\epsilon \cup \epsilon', n_2) = \mathsf{frame\_lockset}(\imath, n_1,\, \gamma_1, \gamma_2 \,?\, \gamma_3)} \;\textit{(W3)}$$

$$\frac{n_1 > 0 \qquad (\epsilon, n_2) = \mathsf{frame\_lockset}(\imath, n_1,\, \gamma) \qquad \mathtt{j} \neq \imath \qquad \epsilon' = \{\mathtt{j} \mid \mathsf{lk}(\kappa) - \mathsf{lk}(\gamma(\mathtt{j} @ n_0)) < 0\}}{(\epsilon \,\cup\, \epsilon', n_2) = \mathsf{frame\_lockset}(\imath, n_1,\, \gamma, (\mathtt{j} @ n_0)^\kappa)} \;\textit{(W4)}$$

$$\frac{}{\{\imath\} = \mathsf{lockset}(\imath, n, \Box)} \;\textit{(L1)}$$

$$\frac{n_1 > 0 \qquad \epsilon' = \mathsf{lockset}(\imath, n_2, E) \qquad (\epsilon, n_2) = \mathsf{frame\_lockset}(\imath, n_1,\, \gamma)}{\epsilon \cup \epsilon' = \mathsf{lockset}(\imath, n_1, E[\mathsf{pop}_\gamma\, \Box])} \;\textit{(L2)}$$

$$\frac{E \neq \Box}{\{\imath\} = \mathsf{lockset}(\imath, 0, E)} \;\textit{(L3)} \qquad \frac{F \neq \mathsf{pop}_\gamma\, \Box \qquad \epsilon = \mathsf{lockset}(\imath, n_1, E) \qquad n_1 > 0}{\epsilon = \mathsf{lockset}(\imath, n_1, E[F])} \;\textit{(L4)}$$

Figure 6.4: Operational semantics, helper relations.

the application term is changed from parallel to sequential. The continuation effect associated with the sequential annotation equals the resulting effect of the function being applied (i.e., $\mathsf{min}(\gamma_a)$). Notice, that θ is divided into two lists θ_1 and θ_2 using the new thread's initial effect $\mathsf{max}(\gamma_a)$ as a reference for consuming the appropriate number of counts from θ. The formal definitions of helper functions such as min, max and $split$ are available in Figure 6.4. When evaluation of a thread reduces to a unit value, the thread is removed from the thread list (rule *E-T*). This is successfully only if the thread has previously released all of its resources.

The rule for sequential function application (*E-A*) reduces an application redex to a pop expression, which contains the body of the function and is annotated with the same effect as the application term. Evaluation propagates through pop expressions (rule *E-PP*), which are only useful for calculating future locksets in rule *E-LK0*. The rules for evaluating the application of polymorphic functions (*E-RP*) and recursive functions (*E-FX*) are standard, as well as the rules for evaluating conditionals (*E-IT* and *E-IF*).

The rules for reference allocation, assignment and dereference are straightforward. Rule *E-NG* appends a fresh location $\imath$ (with initial value v) and the dynamic count $(1, 1)$ to S and θ respectively. Rules *E-AS* and *E-D* require that the location ($\imath$) being accessed is both live and accessible and no other thread has access to $\imath$ (see Figure 6.4 for the definition of function $locked$). Therefore dangling memory location accesses as well as unsynchronized accesses cause the evaluation to get *stuck*. Furthermore, the rules *E-SH*, *E-RL* and *E-UL* manipulate a cell's reference or lock count. They are also straightforward, simply checking that the cell is live and (in the case of *E-UL*) locked. Rule *E-RL* makes sure that a cell is unlocked before its reference count can be decremented to zero.

The most interesting rule is *E-LK0*, which applies when the reference being locked ($\imath$) is initially unlocked. The future lockset (ϵ) is dynamically computed, by inspecting the preceding stack frames (E) as well as the lock annotation (γ_1). The lockset ϵ is a list of locations (and thus locks). (see Figure 6.4 for the definition of function $lockset$). The reference $\imath$ must be live and no other thread must hold either $\imath$ or any of the locations in ϵ. Upon success, the lock count of $\imath$ is incremented by one. On the other hand, rule *E-LK1* applies when $\imath$ has already been locked by the current thread (that tries to lock it again). This immediately succeeds and the lock count is incremented by one. The definition of function $+_\imath$ is available in Figure 6.4.

6.4 Static semantics

In this section we present our type and effect system and discuss the most interesting parts. Effects are used to statically track the capability of each cell. An effect (γ) is an *ordered list* of elements of the form r^κ and summarizes the sequence of operations (e.g., locking or sharing) on references. The syntax of types in Figure 6.1 (on page 80) is more or less standard: Atomic types consist of base types (the unit type, denoted by $\langle\rangle$, and `bool`); reference types $\texttt{ref}(\tau, r)$ are associated with a type-level cell name r and monomorphic function types carry an *effect*. Figure 6.5 contains the well-formedness rules for the typing environment, types and effects. Figure 6.7 and Figure 6.8 contain the typing rules and Figure 6.6 contains auxiliary typing rules. The typing relation is denoted by $R; M; \Delta; \Gamma \vdash e : \tau \,\&\, (\gamma;\gamma')$, where $R; M; \Delta; \Gamma$ is the typing context, e is an expression, τ is the type attributed to e, γ is the *input effect*, and γ' is the *output effect*. In the typing context, M is a mapping of constant locations to types, Δ is a set of cell variables, and Γ is a mapping of term variables to types. Lock operations and sequential application terms are annotated with the continuation effect. This imposes the restriction that effects must flow backwards. The input effect γ to an expression e is indeed the continuation effect; it represents the operations that follow the evaluation of e. On the other hand, the output effect γ' represents the combined operations of e and its continuation. The typing relation guarantees that the input effect is always a *prefix* of the output effect.

The typing rules *T-U*, *T-TR*, *T-FL*, *T-V*, *T-L*, *T-RF* and *T-RP* are almost standard, except for the occasional premise $\tau \simeq \tau'$ which allows the type system to ignore the identifiers used for location aliasing and, for example, treat the types $\imath@n_1$ and $\imath@n_2$ as equal. The typing rule *T-F* checks that,

$$\frac{}{M;\Delta \vdash \emptyset} \qquad \frac{M;\Delta \vdash r \quad M;\Delta \vdash \gamma_1}{M;\Delta \vdash \gamma_1, r^{\kappa}} \qquad \frac{M;\Delta \vdash \gamma_1 \quad M;\Delta \vdash \gamma_2 \quad M;\Delta \vdash \gamma_3}{M;\Delta \vdash \gamma_1, \gamma_2 \,?\, \gamma_3}$$

$$\frac{r \in \Delta \cup \mathsf{dom}(M)}{M;\Delta \vdash r} \qquad \frac{M;\Delta \vdash \iota}{M;\Delta \vdash \iota@n} \qquad \frac{M;\Delta \vdash \rho}{M;\Delta \vdash \rho@n}$$

$$\frac{\vdash M \quad M;\Delta \vdash \Gamma \quad M;\Delta \vdash \gamma_1 \quad M;\Delta \vdash \gamma_2 \quad \gamma_1 \lhd \gamma_2 \quad \mathsf{seq}(\emptyset) \vdash \gamma_2}{\vdash M;\Delta;\Gamma;\gamma_1;\gamma_2}$$

$$\frac{}{M;\Delta \vdash \mathtt{bool}} \qquad \frac{M;\Delta,\rho \vdash \tau}{M;\Delta \vdash \forall\rho.\,\tau} \qquad \frac{M;\Delta \vdash \tau \quad M;\Delta \vdash r}{M;\Delta \vdash \mathtt{ref}(\tau, r)}$$

$$\frac{\mathsf{min}(\gamma_1)\ \text{defined} \qquad M;\Delta \vdash \tau_1 \quad M;\Delta \vdash \gamma_1 \quad M;\Delta \vdash \tau_2}{M;\Delta \vdash \tau_1 \xrightarrow{\gamma_1} \tau_2} \qquad \frac{}{M;\Delta \vdash \langle\rangle}$$

$$\frac{}{M;\Delta \vdash \emptyset} \qquad \frac{M;\Delta \vdash \tau \quad x \notin \mathsf{dom}(\Gamma) \quad M;\Delta \vdash \Gamma}{M;\Delta \vdash \Gamma, x:\tau}$$

$$\frac{}{\vdash \emptyset} \qquad \frac{\vdash M \quad \iota \notin \mathsf{dom}(M) \quad M;\emptyset \vdash \tau}{\vdash M, \iota \mapsto \tau}$$

Figure 6.5: Well-formedness rules.

if the effect γ_b that is annotated in the function's type is well formed, it is indeed the effect of the function's body. On the other hand, the typing rule *T-A* for function application has a lot more work to do. It joins the input effect γ (i.e., the continuation effect) and the function's effect γ_a, which contains the entire history of events occurring in the function body; this is performed by the premise $\xi \vdash \gamma_2 = \gamma \oplus \gamma_a$, which performs all the necessary checks to ensure that all the capabilities required in the function's effect γ_a are available, that pure capabilities are not aliased, and, in the case of parallel application, that no lock capabilities are split and that the resulting capability of each location is zero. Rule *T-PP* works as a bridge between the body of a function that is being executed and its calling environment. Rule *T-FX* uses the function summary to summarize the effect of the function's body and to check that the type annotation indeed contains the right summary. The effect summary is conservatively computed as the set of locks that are acquired within the function body; the unmatched lock/unlock operations are also taken into account.

Rule *T-NG* for creating new cells passes the input effect γ to e_2, the body of `let`, augmented by $\rho^{0,0}$. This means that, upon termination of e_2, both references and locks of ρ must have been consumed. The output effect of e_2 is a γ_1 such that ρ has capability $(1,1)$, which implies that when e_2 starts being evaluated ρ is live and locked. The input effect of the cell initializer expression e_1 is equal to the output effect of e_2 without any occurrences of ρ. Rules *T-AS* and *T-D* check that, before dereferencing or assigning to cells, a capability of at least $(1,1)$ is held. Rules *T-SH*, *T-RL*, *T-LK* and *T-UL* are the ones that modify cell capabilities. In each rule, κ is the capability after the operation has been executed. In the case of *T-RL*, if the reference count for a cell is decremented to zero, then all locks must have previously been released. The last rule in Figure 6.8, and probably the least intuitive, is *T-IF*. Suppose γ is the input (continuation) effect to a conditional expression. Then γ is passed as the input effect to both branches. We know that the outputs of both branches will have γ as a common prefix; if γ_2 and γ_3 are the suffixes, respectively, then $\gamma_2 \,?\, \gamma_3$ is the combined suffix, which is passed as the input effect to the condition e_1.

$$\frac{}{r \simeq r}\ \textit{(S0)} \qquad \frac{r' \simeq r}{r \simeq r'}\ \textit{(S1)} \qquad \frac{r \simeq r'}{r \simeq r'@n_2}\ \textit{(S2)} \qquad \frac{}{\emptyset \simeq \emptyset}\ \textit{(S3)}$$

$$\frac{r_1 \simeq r_2 \qquad \gamma_1 \simeq \gamma_2}{\gamma_1, {r_1}^{\kappa} \simeq \gamma_2, {r_2}^{\kappa}}\ \textit{(S4)} \qquad \frac{\gamma_1 \simeq \gamma_4 \qquad \gamma_2 \simeq \gamma_5 \qquad \gamma_3 \simeq \gamma_6}{\gamma_1, \gamma_2 \,?\, \gamma_3 \simeq \gamma_4, \gamma_5 \,?\, \gamma_6}\ \textit{(S5)} \qquad \frac{}{\tau \simeq \tau}\ \textit{(S6)}$$

$$\frac{\begin{array}{c}\tau_3 \simeq \tau_4 \\ r_1 \simeq r_2\end{array}}{\mathtt{ref}(\tau_3, r_1) \simeq \mathtt{ref}(\tau_4, r_2)}\ \textit{(S7)} \qquad \frac{\begin{array}{c}\text{fresh } \rho_1@n \\ \tau_1[\rho_1@n/\rho] \simeq \tau_2[\rho_1@n/\rho']\end{array}}{\forall\rho.\,\tau_1 \simeq \forall\rho'.\,\tau_2}\ \textit{(S8)}$$

$$\frac{\begin{array}{cc}\tau_1 \simeq \tau_3 & \tau_2 \simeq \tau_4 \\ \gamma_1 \simeq \gamma_3 & \gamma_2 \simeq \gamma_4\end{array}}{\tau_1 \xrightarrow{\gamma_1} \tau_2 \simeq \tau_3 \xrightarrow{\gamma_3} \tau_4}\ \textit{(S9)}$$

$$\mathsf{is\_pure}(\kappa) = \exists n_1.\ \exists n_2.\ \kappa = n_1, n_2$$

$$\frac{\begin{array}{c}\forall r^{\kappa} \in \gamma.\, \mathsf{is\_pure}(\kappa) \Rightarrow \forall r'^{\kappa'} \in \gamma.\, r' \neq r \Rightarrow \neg(r \simeq r') \\ \xi = \mathsf{par} \Rightarrow \forall r^{\kappa}.(r^{\kappa} \in \mathsf{min}(\gamma) \Rightarrow \kappa = (0,0)) \wedge (r^{\kappa} \in \mathsf{max}(\gamma) \wedge \neg\mathsf{is\_pure}(\kappa) \Rightarrow \mathsf{lk}(\kappa) = 0)\end{array}}{\xi \vdash \gamma}\ \textit{(OK)}$$

$$\frac{r' \simeq r \qquad \gamma' = \gamma \setminus r'}{\gamma' = \gamma, r^{\kappa} \setminus r'}\ \textit{(M0)} \qquad \frac{\neg(r' \simeq r) \qquad \gamma' = \gamma \setminus r'}{\gamma', r^{\kappa} = \gamma, r^{\kappa} \setminus r'}\ \textit{(M1)} \qquad \frac{}{\emptyset = \emptyset \setminus r}\ \textit{(M2)}$$

$$\frac{\begin{array}{c}\mathsf{is\_pure}(\kappa_3) \Rightarrow \kappa_2 = (0,0) \wedge \mathsf{is\_pure}(\kappa_1) \qquad \mathsf{is\_pure}(\kappa_1) \Leftrightarrow \mathsf{is\_pure}(\kappa_2) \\ \kappa_1 = (n_3 + n_5, n_4 + n_6) \qquad \kappa_3 = (n_5, n_6) \qquad \kappa_2 = (n_3, n_4)\end{array}}{\kappa_1 = \kappa_2 + \kappa_3}\ \textit{(K1)}$$

$$\frac{}{\gamma = \mathsf{subtract}(\gamma\,,\,\emptyset)}\ \textit{(ES1)} \qquad \frac{\gamma_2 = \mathsf{subtract}(\gamma, r^{\kappa_2}\,,\,\gamma_1) \qquad \kappa = \kappa_2 + \kappa_1}{\gamma_2 = \mathsf{subtract}((\gamma, r^{\kappa})\,,\,(\gamma_1, r^{\kappa_1}))}\ \textit{(ES2)}$$

$$\frac{\kappa = \gamma(r) + \kappa_2 \qquad \gamma'' = \mathsf{add}(\gamma\,,\,\gamma')}{\gamma'', r^{\kappa} = \mathsf{add}(\gamma\,,\,(\gamma', r^{\kappa_2}))}\ \textit{(AD1)} \qquad \frac{}{\emptyset = \mathsf{add}(\gamma\,,\,\emptyset)}\ \textit{(AD2)}$$

$$\frac{\begin{array}{c}\gamma_2 = \mathsf{add}(\mathsf{subtract}(\mathsf{max}(\gamma)\,,\,\mathsf{min}(\gamma_1))\,,\,\gamma_1) \\ \gamma_2 = \mathsf{min}(\gamma_2) :: \gamma_3 \qquad \mathsf{seq}(\emptyset) \vdash \gamma\end{array}}{\mathsf{seq}(\gamma) \vdash \gamma :: \gamma_3 = \gamma \oplus \gamma_1}\ \textit{(D0)}$$

$$\frac{\begin{array}{c}\mathsf{par} \vdash \gamma_1 \\ \gamma_2 = \mathsf{add}(\mathsf{subtract}(\mathsf{max}(\gamma)\,,\,\mathsf{min}(\gamma_1))\,,\,\mathsf{max}(\gamma_1))\end{array}}{\mathsf{par} \vdash \gamma :: \gamma_2 = \gamma \oplus \gamma_1}\ \textit{(D1)}$$

$$\frac{\mathsf{locked}(\gamma, 1, r) \qquad \mathsf{lk}(\kappa) = 0}{\mathsf{locked}((\gamma, r^{\kappa}), 0, r)}\ \textit{(X1)} \qquad \frac{\mathsf{lk}(\kappa) > 0}{\mathsf{locked}((\gamma, r^{\kappa}), 1, r)}\ \textit{(X2)}$$

$$\frac{r \neq r' \qquad \mathsf{locked}(\gamma, n, r)}{\mathsf{locked}((\gamma, r'^{\kappa}), n, r)}\ \textit{(X3)}$$

$$\frac{\mathsf{locked}((\gamma :: \gamma_1), n, r) \vee \mathsf{locked}((\gamma :: \gamma_2), n, r)}{\mathsf{locked}((\gamma, \gamma_1 \,?\, \gamma_2), n, r)}\ \textit{(X4)}$$

$$\frac{\gamma_b = \mathsf{min}(\gamma_a) \qquad \gamma_c = \mathsf{max}(\gamma_a) \qquad \gamma_d = \{r^{\gamma_c(r)+(0,1)} \mid \mathsf{locked}(\gamma_a, 0, r)\}}{\gamma_b :: \gamma_d :: \gamma_c = \mathsf{summary}(\gamma_a)}\ \textit{(L0)}$$

Figure 6.6: Auxiliary functions.

$$\frac{\vdash M;\Delta;\Gamma;\gamma;\gamma}{M;\Delta;\Gamma \vdash () : \langle\rangle \,\&\, (\gamma;\gamma)}\ \textit{(T-U)} \qquad \frac{\vdash M;\Delta;\Gamma;\gamma;\gamma}{M;\Delta;\Gamma \vdash \texttt{true} : \texttt{bool} \,\&\, (\gamma;\gamma)}\ \textit{(T-TR)}$$

$$\frac{\vdash M;\Delta;\Gamma;\gamma;\gamma}{M;\Delta;\Gamma \vdash \texttt{false} : \texttt{bool} \,\&\, (\gamma;\gamma)}\ \textit{(T-FL)} \qquad \frac{\vdash M;\Delta;\Gamma;\gamma;\gamma \quad (x:\tau') \in \Gamma \quad \tau \simeq \tau'}{M;\Delta;\Gamma \vdash x : \tau \,\&\, (\gamma;\gamma)}\ \textit{(T-V)}$$

$$\frac{\vdash M;\Delta;\Gamma;\gamma;\gamma \quad \tau' \equiv \tau_1 \xrightarrow{\gamma_b} \tau_2 \quad M;\Delta \vdash \tau' \quad \tau \simeq \tau' \quad \mathsf{seq}(\emptyset) \vdash \gamma_b \Rightarrow M;\Delta;\Gamma, x:\tau_1 \vdash e_1 : \tau_2 \,\&\, (\mathsf{min}(\gamma_b);\gamma_b)}{M;\Delta;\Gamma \vdash \lambda x.\, e_1 \;\mathsf{as}\; \tau' : \tau \,\&\, (\gamma;\gamma)}\ \textit{(T-F)}$$

$$\frac{M;\Delta,\rho;\Gamma \vdash f : \tau \,\&\, (\gamma;\gamma)}{M;\Delta;\Gamma \vdash \Lambda\rho.\, f : \forall\rho.\, \tau \,\&\, (\gamma;\gamma)}\ \textit{(T-RF)} \qquad \frac{M;\Delta \vdash r \quad M;\Delta \vdash \tau[r/\rho] \quad M;\Delta;\Gamma \vdash e_1 : \forall\rho.\, \tau \,\&\, (\gamma;\gamma')}{M;\Delta;\Gamma \vdash (e_1)\,[r] : \tau[r/\rho] \,\&\, (\gamma;\gamma')}\ \textit{(T-RP)}$$

$$\frac{M;\Delta;\Gamma \vdash e_1 : \tau_1 \xrightarrow{\gamma_a} \tau_2 \,\&\, (\gamma_3;\gamma') \quad \xi \vdash \gamma_2 = \gamma \oplus \gamma_a \quad M;\Delta;\Gamma \vdash e_2 : \tau_1 \,\&\, (\gamma_2;\gamma_3) \quad \xi = \mathsf{par} \Rightarrow \tau_2 = \langle\rangle}{M;\Delta;\Gamma \vdash (e_1\ e_2)^{\xi} : \tau_2 \,\&\, (\gamma;\gamma')}\ \textit{(T-A)}$$

$$\frac{M;\Delta;\Gamma \vdash e : \tau' \,\&\, (\mathsf{min}(\gamma_b);\gamma_b) \quad \gamma_b \simeq \gamma_b' \quad \mathsf{seq}(\gamma) \vdash \gamma' = \gamma \oplus \gamma_b' \quad \tau' \simeq \tau \quad \vdash M;\Delta;\Gamma;\gamma;\gamma'}{M;\Delta;\Gamma \vdash \mathsf{pop}_{\gamma}\ e : \tau \,\&\, (\gamma;\gamma')}\ \textit{(T-PP)}$$

$$\frac{\tau \equiv \tau_1 \xrightarrow{\gamma_b} \tau_2 \quad \tau' \equiv \tau_1' \xrightarrow{\gamma_a'} \tau_2' \quad \tau \simeq \tau' \quad \gamma_a \simeq \gamma_a' \quad M;\Delta;\Gamma, x:\tau \vdash f : \tau' \,\&\, (\gamma;\gamma) \quad \gamma_b = \mathsf{summary}(\gamma_a)}{M;\Delta;\Gamma \vdash \mathtt{fix}\ x:\tau.\, f : \tau \,\&\, (\gamma;\gamma)}\ \textit{(T-FX)}$$

Figure 6.7: Typing rules *(part I)*.

6.5 Type safety

In this section we present proof sketches for the fundamental theorems that prove type safety of our language.[3] The type safety formulation is based on proving *progress*, *deadlock freedom* and *preservation* lemmata. Informally, a program written in our language is safe when for each thread of execution either an evaluation step can be performed, or the thread is waiting to acquire a lock (*blocked*). In addition, there must not exist any threads that have reached a deadlocked state. As discussed in Section 6.3, a thread may become stuck when it performs an illegal operation, or when it references a location that has been deallocated, or when it accesses a location that has not been locked.

Definition 6.1 (Thread Typing.) Let $E[e]$ be the body of a thread and let θ be the thread's *access list*. Thread typing is defined by the rule:

$$\frac{M;\Delta;\Gamma \vdash e : \tau \,\&\, (\gamma_a;\gamma_b) \quad M;\Delta;\Gamma \vdash E : \tau \xrightarrow{\gamma_a;\gamma_b} \langle\rangle \,\&\, (\gamma_1;\gamma_2) \quad \forall r^{\kappa} \in \gamma_1.\, \kappa = (0,0) \quad \mathsf{counts\_ok}(E[\mathsf{pop}_{\gamma_b}\,\square], \theta) \quad \mathsf{lockset\_ok}(E[\mathsf{pop}_{\gamma_b}\,\square], \theta)}{M;\Delta;\Gamma \vdash_t \theta; E[e] : \langle\rangle \,\&\, (\gamma_1;\gamma_2)}\ \textit{(EA)}$$

First of all, thread typing implies the typing of $E[e]$.

Secondly, thread typing establishes an exact correspondence between counts of the access list θ and counts of pop expression annotations that reside in the evaluation context $E[\mathsf{pop}_{\gamma_b}\,\square]$ (i.e., $\mathsf{counts\_ok}(E[\mathsf{pop}_{\gamma_b}\,\square], \theta)$ — defined in Figure 6.9). The typing derivations of e and E establish an

[3] The complete proofs are given in the Appendix C.

$$\frac{\vdash M;\Delta;\Gamma;\gamma;\gamma \qquad (\imath \mapsto \tau') \in M \qquad \tau \simeq \mathtt{ref}(\tau',\imath)}{M;\Delta;\Gamma \vdash \mathtt{loc}_\imath : \tau \,\&\, (\gamma;\gamma)} \;\textit{(T-L)}$$

$$\frac{\begin{array}{c} M;\Delta;\Gamma \vdash e_1 : \tau_1 \,\&\, (\gamma_1 \setminus \rho;\gamma') \qquad\qquad \gamma_1(\rho) = (1,1) \\ M;\Delta \vdash \tau \qquad M;\Delta,\rho;\Gamma,x : \mathtt{ref}(\tau_1,\rho) \vdash e_2 : \tau \,\&\, (\gamma,\rho^{0,0};\gamma_1) \end{array}}{M;\Delta;\Gamma \vdash \mathtt{let}\ \rho, x = \mathtt{ref}\ e_1\ \mathtt{in}\ e_2 : \tau \,\&\, (\gamma;\gamma')} \;\textit{(T-NG)}$$

$$\frac{\begin{array}{c} M;\Delta;\Gamma \vdash e_1 : \mathtt{ref}(\tau,r) \,\&\, (\gamma_1;\gamma') \\ M;\Delta;\Gamma \vdash e_2 : \tau \,\&\, (\gamma;\gamma_1) \qquad \gamma(r) \geq (1,1) \end{array}}{M;\Delta;\Gamma \vdash e_1 := e_2 : \langle\rangle \,\&\, (\gamma;\gamma')} \;\textit{(T-AS)}$$

$$\frac{M;\Delta;\Gamma \vdash e_1 : \mathtt{ref}(\tau,r) \,\&\, (\gamma;\gamma') \qquad \gamma(r) \geq (1,1)}{M;\Delta;\Gamma \vdash \mathtt{deref}\ e_1 : \tau \,\&\, (\gamma;\gamma')} \;\textit{(T-D)}$$

$$\frac{M;\Delta;\Gamma \vdash e : \mathtt{ref}(\tau,r) \,\&\, (\gamma, r^{\kappa-(1,0)};\gamma') \qquad \kappa \geq (2,0) \qquad \gamma(r) = \kappa}{M;\Delta;\Gamma \vdash \mathtt{share}\ e : \langle\rangle \,\&\, (\gamma;\gamma')} \;\textit{(T-SH)}$$

$$\frac{\begin{array}{c} M;\Delta;\Gamma \vdash e : \mathtt{ref}(\tau,r) \,\&\, (\gamma, r^{\kappa+(1,0)};\gamma') \\ \kappa = (n_1,n_2) \qquad n_1 = 0 \Rightarrow n_2 = 0 \qquad \gamma(r) = \kappa \end{array}}{M;\Delta;\Gamma \vdash \mathtt{release}\ e : \langle\rangle \,\&\, (\gamma;\gamma')} \;\textit{(T-RL)}$$

$$\frac{M;\Delta;\Gamma \vdash e : \mathtt{ref}(\tau,r) \,\&\, (\gamma, r^{\kappa-(0,1)};\gamma') \qquad \kappa \geq (1,1) \qquad \gamma(r) = \kappa}{M;\Delta;\Gamma \vdash \mathtt{lock}_\gamma\ e : \langle\rangle \,\&\, (\gamma;\gamma')} \;\textit{(T-LK)}$$

$$\frac{M;\Delta;\Gamma \vdash e : \mathtt{ref}(\tau,r) \,\&\, (\gamma, r^{\kappa+(0,1)};\gamma') \qquad \kappa \geq (1,0) \qquad \gamma(r) = \kappa}{M;\Delta;\Gamma \vdash \mathtt{unlock}\ e : \langle\rangle \,\&\, (\gamma;\gamma')} \;\textit{(T-UL)}$$

$$\frac{\begin{array}{c} M;\Delta;\Gamma \vdash e_1 : \mathtt{bool} \,\&\, (\gamma,\gamma_2\,?\,\gamma_3;\gamma') \qquad \mathsf{max}(\gamma :: \gamma_2) = \mathsf{max}(\gamma :: \gamma_3) \\ M;\Delta;\Gamma \vdash e_2 : \tau \,\&\, (\gamma;\gamma :: \gamma_2) \qquad M;\Delta;\Gamma \vdash e_3 : \tau \,\&\, (\gamma;\gamma :: \gamma_3) \end{array}}{M;\Delta;\Gamma \vdash \mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3 : \tau \,\&\, (\gamma;\gamma')} \;\textit{(T-IF)}$$

Figure 6.8: Typing rules *(part II)*.

exact correspondence between the annotations of pop expressions and static effects. Therefore, for each location $\imath$ in θ, the dynamic reference and lock counts of $\imath$ are identical to the static counts of $\imath$ deduced by the type system.

Thirdly, thread typing enforces the invariant that the future lockset of an acquired lock at any program point is *always* a subset of the future lockset computed when the lock was initially acquired (i.e., $\mathsf{lockset\_ok}(E[\mathtt{pop}_{\gamma_b}\ \square], \theta)$ — defined in Figure 6.9). This invariant is essential for establishing deadlock freedom. Finally, all locations must be deallocated and released when a thread terminates ($\forall r^\kappa \in \gamma_1.\, \kappa = (0,0)$).

Definition 6.2 (Process Typing.) A collection of threads T is well typed if each thread in T is well typed and thread identifiers are distinct:

$$\frac{}{M \vdash \emptyset} \qquad \frac{M;\emptyset;\emptyset \vdash_t \theta; e : \langle\rangle \,\&\, (\gamma;\gamma') \qquad M \vdash T \qquad n \notin \mathsf{dom}(T)}{M \vdash T, n : \theta; e}$$

$$\begin{array}{ll} \mathsf{pure}(\gamma) & = \{\iota \mid \iota \simeq r \wedge r^{n_1,n_2} \in \gamma \wedge n_1 + n_2 > 0\} \\ \mathsf{counts\_ok}(E,\theta) & = \mathsf{counts\_rok}(E,\theta,\emptyset,\emptyset) \end{array}$$

$$\frac{\theta', \iota \mapsto n_1; n_2; \epsilon_1; \epsilon_2 = \theta - \gamma \qquad \kappa = (n_3, n_4) \qquad \iota \simeq r \qquad (n_1, n_2) \geq (n_3, n_4) \qquad \mathsf{is\_pure}(\kappa) \Rightarrow n_1 = n_3 \wedge n_2 = n_4}{\theta', \iota \mapsto n_1 - n_3; n_2 - n_4; \epsilon_1; \epsilon_2 = \theta - \gamma, r^{\kappa}} \textit{ (B0)} \qquad \frac{}{\theta = \theta - \emptyset} \textit{ (B1)}$$

$$\frac{\mathsf{counts\_rok}(E, \theta', \gamma, \epsilon \cup \epsilon') \qquad \epsilon' = \mathsf{pure}(\gamma'') \qquad \theta' = \theta - \gamma'' \qquad \epsilon \cap \epsilon' = \emptyset \qquad \gamma'' = \mathsf{subtract}(\mathsf{max}(\gamma)\,,\, \mathsf{max}(\gamma_1))}{\mathsf{counts\_rok}(E[\mathsf{pop}_{\gamma}\ \square], \theta, \gamma_1, \epsilon)} \textit{ (C1)}$$

$$\frac{\forall \iota.\theta(\iota) = (0,0)}{\mathsf{counts\_rok}(\square, \theta, \gamma, \epsilon)} \textit{ (C0)} \qquad \frac{F \neq \mathsf{pop}_{\gamma'}\ \square \qquad \mathsf{counts\_rok}(E, \theta, \gamma, \epsilon)}{\mathsf{counts\_rok}(E[F], \theta, \gamma, \epsilon)} \textit{ (C2)}$$

$$\frac{\mathsf{lockset\_ok}(E, \theta) \qquad \mathsf{lockset}(\iota, n_2, E) \cap \epsilon_1 \subseteq \epsilon_2}{\mathsf{lockset\_ok}(E\,,\ \theta, \iota \mapsto n_1; n_2; \epsilon_1; \epsilon_2)} \textit{ (DL0)} \qquad \frac{}{\mathsf{lockset\_ok}(E, \emptyset)} \textit{ (DL1)}$$

Figure 6.9: Type safety validity relations.

Definition 6.3 (Store Typing.) A store S is well typed if there is a one-to-one correspondence between S and M and all stored values are closed and well typed:

$$\frac{\mathsf{dom}(M) = \mathsf{dom}(S) \qquad \forall (\iota \mapsto \tau) \in M.M; \emptyset; \emptyset \vdash S(\iota) : \tau \,\&\, (\emptyset;\emptyset)}{M \vdash S}$$

Definition 6.4 (Configuration Typing.) A configuration $S; T$ is *well typed* when both T and S are well typed, and locks are acquired by at most one thread (i.e., $\mathsf{mutex}(T)$ holds).

$$\mathsf{mutex}(T) \equiv \forall T_1, n : \theta; E[e].T = T_1, n : \theta; E[e] \Rightarrow \forall \iota.\theta(\iota) \geq (1,1) \Rightarrow \iota \notin \mathsf{locked}(T_1)$$

$$\frac{M \vdash T \qquad M \vdash S \qquad \mathsf{mutex}(T)}{M \vdash S; T}$$

Definition 6.5 (Deadlocked State.) A set of threads $n_0, \ldots, n_k$, where $k > 0$, has reached a *deadlocked state*, when each thread n_ι has acquired lock $\ell_{(\iota+1) \bmod (k+1)}$ and is waiting for lock ℓ_ι.

$$\begin{array}{rcl} \mathsf{deadlocked}(T) & \equiv & T \supseteq T_1, n_0 : \theta_0; E[\mathtt{lock}_{\gamma_0}\ \mathtt{loc}_{\iota_0}], \ldots n_k : \theta_k; E_k[\mathtt{lock}_{\gamma_k}\ \mathtt{loc}_{\iota_k}] \wedge k > 0 \Rightarrow \\ & & \forall m_1 \in [0,k].m_2 = (m_1 + 1)\ \mathit{mod}(k+1) \wedge \theta_{m_1}(\iota_{m_2}) \geq (1,1) \end{array}$$

Definition 6.6 (Not Stuck.) A configuration $S; T$ is *not stuck* when each thread in T can take one of the evaluation steps in Figure 6.3 or it is trying to acquire a lock which (either itself or its future lockset) is unavailable (i.e., $\mathsf{blocked}(T, n)$ holds).

$$\begin{array}{rcl} \mathsf{blocked}(T, n) & \equiv & T = T_1, n : \theta; E[\mathtt{lock}_{\gamma_2}\ \mathtt{loc}_\iota] \wedge \theta(\iota) = (n_1, n_2) \wedge n_1 > 0 \wedge n_2 = 0 \wedge \\ & & \mathsf{locked}(T_1) \cap \mathsf{lockset}(\iota, 1, E[\mathsf{pop}_{\gamma_2}\ \square]) \neq \emptyset \end{array}$$

$$\frac{\forall T', n : \theta; e.T = T', n : \theta; e \Rightarrow (T' \subseteq T'' \wedge S; T \rightsquigarrow S'; T'') \vee \mathsf{blocked}(T, n)}{\vdash S; T}$$

Given these definitions, we can now present the main results of this chapter. *Progress*, *deadlock freedom* and *preservation* are formalized at the *program* level, i.e., for all concurrently executed threads.

Lemma 6.1 (Deadlock Freedom) If the initial configuration takes n steps, where each step is well typed, then the resulting configuration has not reached a deadlocked state.

Proof. Let us assume that z threads have reached a deadlocked state and let $m \in [0, z-1]$, $k = (m+1)$ *mod* z and $o = (k+1)$ *mod* z. According to definition of *deadlocked state*, thread m acquires lock $\imath_k$ and waits for lock $\imath_m$, whereas thread k acquires lock $\imath_o$ and waits for lock $\imath_k$. Assume that m is the first of the z threads that acquires a lock so it acquires lock $\imath_k$, before thread k acquires lock $\imath_o$.

Let us assume that $S_y; T_y$ is the configuration once $\imath_o$ is acquired by thread k for the first time, ϵ_{1y} is the corresponding lockset of $\imath_o$ ($\epsilon_{1y} = \mathsf{lockset}(\imath_o, 1, E[\mathsf{pop}_{\gamma_y}\ \Box])$) and ϵ_{2y} is the set of all heap locations ($\epsilon_{2y} = \mathsf{dom}(S_y)$) at the time $\imath_o$ is acquired. Then, $\imath_k$ does not belong to ϵ_{1y}, otherwise thread k would have been blocked at the lock request of $\imath_o$ as $\imath_k$ is already owned by thread m.

Let us assume that when thread k attempts to acquire $\imath_k$, the configuration is of the form $S_x; T_x$. According to the assumption of this lemma that all configurations are well typed so $S_x; T_x$ is well-typed as well. By inversion of the typing derivation of $S_x; T_x$, we obtain the typing derivation of thread $n_k : \theta_k; E_k[\mathtt{lock}_{\gamma'_k}\ \mathtt{loc}_{\imath_k}]$: $\mathtt{lock}_{\gamma'_k}\ \mathtt{loc}_{\imath_k}$ is well-typed with input-output effect $(\gamma'_k; \gamma''_k)$, where $\kappa = \gamma'_k(\imath_k @ n')$, $\kappa \geq (1,1)$, $\gamma'' = \gamma'_k, (\imath_k @ n')^{\kappa - (1,0)}$, and $\mathsf{lockset\_ok}(E_k[\mathsf{pop}_{\gamma''_k}\ \Box], \theta_k)$ holds, where θ_k is the access list of thread k. $\mathsf{lockset\_ok}\ (E_k[\mathsf{pop}_{\gamma''_k}\ \Box], \theta_k)$ implies $\mathsf{lockset}(\imath_o, n_2, E_k[\mathsf{pop}_{\gamma''_k}\ \Box]) \cap \epsilon_1 \subseteq \epsilon_2$, where $\theta_k = \theta'_k, \imath_o \mapsto n_1; n_2; \epsilon_1; \epsilon_2$ (notice that n_2 is positive, $\epsilon_2 = \epsilon_{1y}$ and $\epsilon_1 = \epsilon_{2y}$ — this is immediate by the operational steps from $S_y; T_y$ to $S_x; T_x$ and rule *E-LK0*).

We have assumed that m is the first thread to lock $\imath_k$ at some step before $S_y; T_y$, thus $\imath_k \in \mathsf{dom}(S_y)$ (the store can only grow — this is immediate by observing the operational semantics rules). By the definition of $lockset$ function and the definition of γ''_k we have that $\imath_k \in \mathsf{lockset}(\imath_o, n_2, E_k[\mathsf{pop}_{\gamma''_k}\ \Box])$. Therefore, $\imath_k \in \mathsf{lockset}(\imath_o, n_2, E_k[\mathsf{pop}_{\gamma''_k}\ \Box]) \cap \mathsf{dom}(S_y) \subseteq \epsilon_{1y}$, which is a contradiction.

Lemma 6.2 (Progress) If $S; T$ is a well typed configuration, then $S; T$ is not stuck.

Proof. It suffices to show that for any thread in T, a step can be performed or $block$ predicate holds for it. Let n be an arbitrary thread in T such that $T = T_1, n : \theta; e$ for some T_1. By inversion of the typing derivation of $S; T$ we have that $M; \emptyset; \emptyset \vdash_t \theta; e : \langle\rangle(\gamma;\gamma')$, $\mathsf{mutex}(T)$, and $M \vdash S$.

If e is a *value* then by inversion of $M; \emptyset; \emptyset \vdash_t \theta; e : \langle\rangle\ \&\ (\gamma;\gamma')$, we obtain that $\gamma = \gamma'$, $E[e] = \Box[()]$ and $\forall \imath. \theta(\imath) = (0,0)$, as a consequence of $\forall r^\kappa \in \gamma.\kappa = (0,0)$ and $\mathsf{counts\_ok}\ (\Box[\mathsf{pop}_\gamma\ \Box], \theta)$. Thus, rule *E-T* can be applied.

If e is not a value then it can be trivially shown (by induction on the typing derivation of e) that there exists a redex u and an evaluation context E such that $e = E[u]$. By inversion of the thread typing derivation for e we obtain that $M; \emptyset; \emptyset \vdash u : \tau(\gamma_a;\gamma_b)$, $M; \emptyset; \emptyset \vdash E : \tau \xrightarrow{\gamma_a;\gamma_b} \langle\rangle(\gamma;\gamma')$ (the evaluation context typing rules are provided in Figure 6.10 and Figure 6.11), $\mathsf{counts\_ok}(E[\mathsf{pop}_{\gamma_b}\ \Box], \theta)$ hold.

Then, we proceed by performing a case analysis on u (we only consider the most interesting cases):

Case $(\lambda x.\, e'\ \mathtt{as}\ \tau\ v)^{\mathsf{par}}$: it suffices to show that $(\theta_1, \theta_2) = \mathsf{split}(\theta\ ,\ \mathsf{max}(\gamma_c))$ is defined, where γ_c is the annotation of type τ. If $\mathsf{max}(\gamma_c)$ is empty, then the proof is immediate from the base case of split function. Otherwise, we must show that for all $\imath$, the count $\theta(\imath)$ is greater than or equal to the sum of all $(\imath @ n)^\kappa$ in $\mathsf{max}(\gamma_c)$. This can be shown by considering $\mathsf{par} \vdash \gamma_b = \gamma_a \oplus \gamma_c$ (i.e., the max counts in γ_c are less than or equal to the max counts in γ_b), which can be obtained by inversion of the typing derivation of $(\lambda x.\, e'\ \mathtt{as}\ \tau\ v)^{\mathsf{par}}$, and the exact correspondence between static (γ_b) and dynamic counts (i.e, $\mathsf{counts\_ok}(E[\mathsf{pop}_{\gamma_b}\ \Box], \theta)$). Thus, rule *E-SN* can be applied to perform a single step.

Case $\mathtt{share}\ \mathtt{loc}_\imath$: $\mathsf{counts\_ok}(E[\mathsf{pop}_{\gamma_b}\ \Box], \theta)$ establishes an exact correspondence between dynamic and static counts. The typing derivation implies that $\gamma_a(\imath @ n_1) \geq (2,0)$, for some n_1 existentially bound in the premise of the derivation. Therefore, $\theta(\imath) \geq (1,0)$. It is possible to perform a single step using rule *E-SH*. The cases for $\mathtt{release}\ \mathtt{loc}_\imath$ and $\mathtt{unlock}\ \mathtt{loc}_\imath$ can be shown in a similar manner.

Case $\mathtt{lock}_{\gamma_a}\ \mathtt{loc}_\imath$: similarly to the case we can show that $\theta(\imath) = (n_1, n_2)$ and n_1 is positive. If n_2 is positive, rule *E-LK1* can be applied. Otherwise, n_2 is zero. Let ϵ be equal to $\mathsf{locked}(T_1) \cap \mathsf{lockset}(\imath, 1, E[\mathsf{pop}_{\gamma_a}\ \Box])$. If ϵ is empty then rule *E-LK0* can be applied in order to perform a single step. Otherwise, $\mathsf{blocked}(T, n)$ predicate holds and the configuration is not stuck.

$$\frac{\vdash M;\Delta;\Gamma;\gamma_1;\gamma_2 \qquad M;\Delta \vdash \tau}{M;\Delta;\Gamma \vdash \Box : \tau \xrightarrow{\gamma_1;\gamma_2} \tau \,\&\, (\gamma_1;\gamma_2)} \;(E0) \qquad \frac{\begin{array}{c} M;\Delta;\Gamma \vdash E : \tau_2 \xrightarrow{\gamma_5;\gamma_6} \tau_3 \,\&\, (\gamma_1;\gamma_2) \\ M;\Delta;\Gamma \vdash F : \tau_1 \xrightarrow{\gamma_3;\gamma_4} \tau_2 \,\&\, (\gamma_5;\gamma_6) \end{array}}{M;\Delta;\Gamma \vdash E[F] : \tau_1 \xrightarrow{\gamma_3;\gamma_4} \tau_3 \,\&\, (\gamma_1;\gamma_2)} \;(E1)$$

$$\frac{\begin{array}{c} \gamma_3 \lhd \gamma_4 \qquad \gamma_2 = \gamma_1 \oplus \gamma_a \\ M;\Delta;\Gamma \vdash e_2 : \tau_1 \,\&\, (\gamma_2;\gamma_3) \qquad M;\Delta \vdash \tau_1 \xrightarrow{\gamma_a} \tau_2 \\ \xi \vdash \gamma_a \qquad \xi = \mathsf{seq}(\gamma_1) \vee (\xi = \mathsf{par} \wedge \tau_2 = \langle\rangle) \end{array}}{M;\Delta;\Gamma \vdash (\Box\ e_2)^{\xi} : (\tau_1 \xrightarrow{\gamma_a} \tau_2) \xrightarrow{\gamma_3;\gamma_4} \tau_2 \,\&\, (\gamma_1;\gamma_4)} \;(F1)$$

$$\frac{\begin{array}{c} M;\Delta;\Gamma \vdash v_1 : \tau_1 \xrightarrow{\gamma_a} \tau_2 \,\&\, (\gamma_3;\gamma_3) \\ \gamma_2 = \gamma_1 \oplus \gamma_a \qquad \gamma_2 \lhd \gamma_3 \qquad M;\Delta \vdash \gamma_3 \\ \xi \vdash \gamma_a \qquad \xi = \mathsf{seq}(\gamma_1) \vee (\xi = \mathsf{par} \wedge \tau_2 = \langle\rangle) \end{array}}{M;\Delta;\Gamma \vdash (v_1\ \Box)^{\xi} : \tau_1 \xrightarrow{\gamma_2;\gamma_3} \tau_2 \,\&\, (\gamma_1;\gamma_3)} \;(F2)$$

$$\frac{\begin{array}{c} \vdash M;\Delta;\Gamma;\gamma;\gamma' \qquad \vdash M;\Delta;\Gamma;\gamma_1;\gamma_2 \qquad M;\Delta \vdash \tau \\ \gamma' = \gamma \oplus \gamma_2 \qquad e = v \Rightarrow \gamma_1 = \mathsf{min}(\gamma_1) \end{array}}{M;\Delta;\Gamma \vdash \mathsf{pop}_{\gamma}\ \Box : \tau \xrightarrow{\gamma_1;\gamma_2} \tau \,\&\, (\gamma;\gamma')} \;(F3)$$

$$\frac{\begin{array}{c} \gamma_3 = \gamma_2 \setminus \rho \qquad \gamma_3 \lhd \gamma' \qquad \gamma_1(\rho) = (1,1) \qquad M;\Delta \vdash \tau_1 \qquad M;\Delta \vdash \tau \\ \vdash M;\Delta;\Gamma;\gamma;\gamma' \qquad M;\Delta,\rho;\Gamma,x : \mathtt{ref}(\tau_1,\rho) \vdash e_2 : \tau \,\&\, (\gamma,\rho^{0,0};\gamma_1) \end{array}}{M;\Delta;\Gamma \vdash \mathtt{let}\ \rho, x = \mathtt{ref}\ \Box\ \mathtt{in}\ e_2 : \tau_1 \xrightarrow{\gamma_3;\gamma'} \tau \,\&\, (\gamma;\gamma')} \;(F4)$$

$$\frac{\begin{array}{c} \vdash M;\Delta;\Gamma;\gamma;\gamma' \qquad M;\Delta \vdash \mathtt{ref}(\tau,r) \\ M;\Delta;\Gamma \vdash e_2 : \tau \,\&\, (\gamma;\gamma_1) \qquad \gamma(r) \geq (1,1) \end{array}}{M;\Delta;\Gamma \vdash \Box := e_2 : \mathtt{ref}(\tau,r) \xrightarrow{\gamma_1;\gamma'} \langle\rangle \,\&\, (\gamma;\gamma')} \;(F5)$$

$$\frac{\begin{array}{c} \gamma \lhd \gamma' \qquad \gamma(r) \geq (1,1) \\ M;\Delta;\Gamma \vdash \mathtt{loc}_i : \mathtt{ref}(\tau,r) \,\&\, (\gamma';\gamma') \end{array}}{M;\Delta;\Gamma \vdash \mathtt{loc}_i := \Box : \tau \xrightarrow{\gamma;\gamma'} \langle\rangle \,\&\, (\gamma;\gamma')} \;(F6)$$

$$\frac{\vdash M;\Delta;\Gamma;\gamma;\gamma' \qquad \gamma(r) \geq (1,1) \qquad M;\Delta \vdash \mathtt{ref}(\tau,r)}{M;\Delta;\Gamma \vdash \mathtt{deref}\ \Box : \mathtt{ref}(\tau,r) \xrightarrow{\gamma;\gamma'} \tau \,\&\, (\gamma;\gamma')} \;(F7)$$

$$\frac{\begin{array}{c} \vdash M;\Delta;\Gamma;\gamma_1;\gamma' \qquad M;\Delta \vdash \mathtt{ref}(\tau,r) \\ \kappa \geq (2,0) \qquad \gamma(r) = \kappa \qquad \gamma_1 = \gamma, r^{\kappa-(1,0)} \end{array}}{M;\Delta;\Gamma \vdash \mathtt{share}\ \Box : \mathtt{ref}(\tau,r) \xrightarrow{\gamma_1;\gamma'} \langle\rangle \,\&\, (\gamma;\gamma')} \;(F8)$$

$$\frac{\begin{array}{c} n_1 = 0 \Rightarrow n_2 = 0 \qquad \gamma_1 = \gamma, r^{\kappa+(1,0)} \qquad \kappa = (n_1,n_2) \\ \vdash M;\Delta;\Gamma;\gamma_1;\gamma' \qquad M;\Delta \vdash \mathtt{ref}(\tau,r) \qquad \gamma(r) = \kappa \end{array}}{M;\Delta;\Gamma \vdash \mathtt{release}\ \Box : \mathtt{ref}(\tau,r) \xrightarrow{\gamma_1;\gamma'} \langle\rangle \,\&\, (\gamma;\gamma')} \;(F9)$$

$$\frac{\begin{array}{c} \vdash M;\Delta;\Gamma;\gamma_1;\gamma' \qquad M;\Delta \vdash \mathtt{ref}(\tau,r) \qquad \kappa \geq (1,0) \\ \gamma(r) = \kappa \qquad \gamma_1 = \gamma, r^{\kappa+(0,1)} \end{array}}{M;\Delta;\Gamma \vdash \mathtt{unlock}\ \Box : \mathtt{ref}(\tau,r) \xrightarrow{\gamma_1;\gamma'} \langle\rangle \,\&\, (\gamma;\gamma')} \;(F10)$$

$$\frac{\begin{array}{c} \vdash M;\Delta;\Gamma;\gamma_1;\gamma' \qquad M;\Delta \vdash \mathtt{ref}(\tau,r) \\ \kappa \geq (1,1) \qquad \gamma(r) = \kappa \qquad \gamma_1 = \gamma, r^{\kappa-(0,1)} \end{array}}{M;\Delta;\Gamma \vdash \mathtt{lock}_{\gamma}\ \Box : \mathtt{ref}(\tau,r) \xrightarrow{\gamma_1;\gamma'} \langle\rangle \,\&\, (\gamma;\gamma')} \;(F11)$$

Figure 6.10: Evaluation context typing rules *(part I)*.

$$\frac{\gamma_3 = \gamma, \gamma_1 \,?\, \gamma_2 \quad M;\Delta \vdash \gamma' \quad \gamma_3 \lhd \gamma' \quad \mathsf{max}(\gamma :: \gamma_1) = \mathsf{max}(\gamma :: \gamma_2) \quad M;\Delta;\Gamma \vdash e_2 : \tau \,\&\, (\gamma;\gamma :: \gamma_1) \quad M;\Delta;\Gamma \vdash e_3 : \tau \,\&\, (\gamma;\gamma :: \gamma_2)}{M;\Delta;\Gamma \vdash \texttt{if}\ \square\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \texttt{bool} \xrightarrow{\gamma_3;\gamma'} \tau \,\&\, (\gamma;\gamma')} \textit{(F12)}$$

$$\frac{M;\Delta;\Gamma \vdash e : \tau \,\&\, (\gamma_a;\gamma_b) \quad M;\Delta;\Gamma \vdash E : \tau \xrightarrow{\gamma_a;\gamma_b} \langle\rangle \,\&\, (\gamma_1;\gamma_2) \quad \forall r^\kappa \in \gamma_1.\, \kappa = (0,0) \quad \mathsf{counts\_ok}(E[\mathsf{pop}_{\gamma_b}\ \square], \theta) \quad \mathsf{lockset\_ok}(E[\mathsf{pop}_{\gamma_b}\ \square], \theta)}{M;\Delta;\Gamma \vdash_t \theta; E[e] : \langle\rangle \,\&\, (\gamma_1;\gamma_2)} \textit{(EA)}$$

Figure 6.11: Evaluation context typing rules *(part II)*.

Case $\texttt{deref}\ \texttt{loc}_\imath$: it can be trivially shown (as in the previous case of $share$ that we proved $\theta(\imath) \geq (1,0)$), that $\theta(\imath) \geq (1,1)$ and since $\mathsf{mutex}(T_1, n:\theta; E[\texttt{deref}\ \texttt{loc}_\imath])$ holds, then $\imath \notin \mathsf{locked}(T_1)$ and thus rule *E-D* can be used to perform a step. The case of $\texttt{loc}_\imath := v$ can be shown in a similar manner. □

Lemma 6.3 (Preservation) Let $S;T$ be a well-typed configuration with $M \vdash S;T$. If the operational semantics takes a step $S;T \rightsquigarrow S';T'$, then there exists $M' \supseteq M$ such that the resulting configuration is well-typed with $M' \vdash S';T'$.

Proof. We proceed by case analysis on the thread evaluation relation (we only consider a few cases due to space limitations):

Case *E-A*: Rule *E-A* implies $S' = S,\ \ T' = T, n:\theta; E[\mathsf{pop}_{\gamma_a}\ e_1[v/x]]$ and $e = (\lambda x.\, e_1\ \mathsf{as}\ \tau_0\ v)^{\mathsf{seq}}$, where $\tau_0 \equiv \tau_1 \xrightarrow{\gamma_c} \tau_2$. By inversion of the configuration typing assumption we have that $\mathsf{mutex}(T, n:\theta; E[e])$ and $M;\emptyset;\emptyset \vdash_t \theta; E[e] : \langle\rangle \,\&\, (\gamma;\gamma')$ hold. It suffices to show that $\mathsf{mutex}(T, n:\theta; E[\mathsf{pop}_{\gamma_a}\ e_1[v/x]])$ and $M;\emptyset;\emptyset \vdash_t \theta; E[\mathsf{pop}_{\gamma_a}\ e_1[v/x]] : \langle\rangle \,\&\, (\gamma;\gamma')$ hold. The former is immediate from $\mathsf{mutex}(T, n:\theta; E[e])$ as no new locks are acquired. Now we proceed with the latter, which can be shown by proving that $M;\emptyset;\emptyset \vdash \mathsf{pop}_{\gamma_a}\ e_1[v/x] : \tau_2' \,\&\, (\gamma_a;\gamma_b)$ holds. By inversion on the thread typing derivation $E[e]$ we have $M;\emptyset;\emptyset \vdash v : \tau_1'(\gamma_b;\gamma_b)$, $\mathsf{seq} \vdash \gamma_b = \gamma_a \oplus \gamma_c'$ and $M;\emptyset;\emptyset \vdash \lambda x.\, e_1\ \mathsf{as}\ \tau_1 \xrightarrow{\gamma_c} \tau_2 : \tau_1' \xrightarrow{\gamma_c'} \tau_2'(\gamma_b;\gamma_b)$, where $\tau_1' \xrightarrow{\gamma_c'} \tau_2' \simeq \tau_1 \xrightarrow{\gamma_c} \tau_2$. We can use proof by induction on the expression typing relation to show that if v is well typed with τ_1', then it is also well typed with τ_1 provided that $\tau_1 \simeq \tau_1'$. Therefore, $M;\emptyset;\emptyset \vdash v : \tau_1(\gamma_b;\gamma_b)$ holds. By inversion of the function typing derivation we obtain that $\mathsf{seq} \vdash \gamma_c \Rightarrow M;\emptyset;\emptyset, x : \tau_1 \vdash e_1 : \tau_2 \,\&\, (\mathsf{min}(\gamma_c);\gamma_c).\ \mathsf{seq} \vdash \gamma_c'$ (premise of $\mathsf{seq} \vdash \gamma_b = \gamma_a \oplus \gamma_c'$) and $\gamma_c \simeq \gamma_c'$ imply that $\mathsf{seq} \vdash \gamma_c$ holds, thus $M;\emptyset;\emptyset, x : \tau_1 \vdash e_1 : \tau_2 \,\&\, (\mathsf{min}(\gamma_c);\gamma_c)$ holds. By applying the standard value substitution lemma on the new typing derivation of v we obtain that $M;\emptyset;\emptyset \vdash e_1[v/x] : \tau_2 \,\&\, (\mathsf{min}(\gamma_c);\gamma_c)$ holds. The application of rule *T-PP* implies that $M;\emptyset;\emptyset \vdash \mathsf{pop}_{\gamma_a}\ e_1[v/x] : \tau_2' \,\&\, (\gamma_a;\gamma_b)$ holds.

Case *E-LK0*, *E-LK1*, *E-UL*, *E-SH* and *E-RL*: these rules generate side-effects as they modify the reference/lock count of location $\imath$. We provide a single proof for all cases. Hence, we are assuming here that u (i.e. in $E[u]$) has one of the following forms: $\texttt{lock}_{\gamma_1}\ \texttt{loc}_\imath$, $\texttt{unlock}\ \texttt{loc}_\imath$ $\texttt{share}\ \texttt{loc}_\imath$ or $\texttt{release}\ \texttt{loc}_\imath$. Rules *E-LK0*, *E-LK1*, *E-UL*, *E-SH* and *E-RL* imply that $S' = S$, $T' = T, n:\theta'; E[()]$, where $()$ replaces u in context E and θ differs with respect to θ' only in the one of the counts of $\imath$ (i.e., $\theta' = \theta[\imath \mapsto \theta(\imath) + (n_1, n_2)]$ and $\gamma_a(r) - \kappa = (n_1, n_2)$ — γ_a is the input effect of $E[u]$).

By inversion of the configuration typing assumption we have that:

- $\mathsf{mutex}(T, n:\theta; E[u])$: In the case of *E-UL*, *E-SH*, *E-LK1* and *E-RL* no new locks are acquired. Thus, $\mathsf{mutex}(T, n:\theta'; E[()])$ holds. In the case of rule *E-LK0*, a new lock $\imath$

is acquired (i.e., when the lock count of $\imath$ is zero) the precondition of *E-LK0* suggests that no other thread holds $\imath$: $\mathsf{locked}(T) \cap \mathsf{lockset}(\imath, 1, E[\mathsf{pop}_{\gamma_a}\ \square]) = \emptyset$. Thus, $\mathsf{mutex}(T, n : \theta'; E[()])$ holds.

- $M; \emptyset; \emptyset \vdash_t \theta; E[u] : \langle\rangle$ & $(\gamma;\gamma')$: By inversion we have that $M; \emptyset; \emptyset \vdash E : \langle\rangle \xrightarrow{\gamma_a;\gamma_b} \langle\rangle(\gamma;\gamma')$ and $M; \emptyset; \emptyset \vdash u : \langle\rangle(\gamma_a;\gamma_b)$, where $\gamma_b = \gamma_a, (\imath@n')^\kappa$ for some n'. It can be trivially shown from the latter derivation that $M; \emptyset; \emptyset \vdash () : \langle\rangle(\gamma_a;\gamma_a)$. We can obtain from the typing derivation of E (proof by induction) that $M; \emptyset; \emptyset \vdash E : \langle\rangle \xrightarrow{\gamma_a;\gamma_a} \langle\rangle(\gamma;\gamma'')$, where $\gamma' = \gamma'', (\imath@n')^\kappa$.
- $\mathsf{lockset\_ok}(E[\mathsf{pop}_{\gamma_b}\ \square], \theta)$ and $\mathsf{counts\_ok}(E[\mathsf{pop}_{\gamma_b}\ \square], \theta)$: By the definition of $lockset$ function it can be shown that $\mathsf{lockset}(\mathtt{j}, n_b, E[\mathsf{pop}_{\gamma_a}\ \square]) \subseteq \mathsf{lockset}(\mathtt{j}, n_b, E[\mathsf{pop}_{\gamma_b}\ \square])$ for all $\mathtt{j} \neq \imath$ in the domain of θ' (n_b is the lock count of $\mathtt{j}$ in θ). The same applies for $\mathtt{j} = \imath$ in the case of rules *E-SH*, *E-RL* as the lock count of $\imath$ is not affected. In the case of rules *E-LK0*, *E-LK1*, *E-UL* we have $\mathsf{lockset}(\imath, n_b \pm 1, E[\mathsf{pop}_{\gamma_a}\ \square])$, but this is identical to $\mathsf{lockset}(\imath, n_b, E[\mathsf{pop}_{\gamma_b}\ \square])$ by the definition of $lockset$. Therefore $\mathsf{lockset\_ok}(E[\mathsf{pop}_{\gamma_a}\ \square], \theta')$ holds. The predicate $counts\_ok$ $(E[\mathsf{pop}_{\gamma_b}\ \square], \theta)$ enforces the invariant that the static counts are identical to the dynamic counts (θ) of $\imath$. The lock count of θ is modified by ± 1 and γ_a differs with respect to γ_b by $(\imath@n')^\kappa$. We can use this fact to show that $\mathsf{counts\_ok}(E[\mathsf{pop}_{\gamma_a}\ \square], \theta')$. □

Lemma 6.4 (Multi-step Program Preservation) Let $S_0; T_0$ be a *closed well-typed configuration* for some M_0 and assume that $S_0; T_0$ evaluates to $S_n; T_n$ in n steps. Then for all $\imath \in [0, n]$ $M_\imath \vdash S_\imath; T_\imath$ holds.

Proof. Proof by induction on the number of steps n using Lemma 6.3.

Theorem 6.1 (Type Safety) Let expression e be the initial program and let the initial typing context M_0 and the initial program configuration $S_0; T_0$ be defined as follows: $M_0 = \emptyset$, $S_0 = \emptyset$, and $T_0 = \{0 : \emptyset; e\}$. If $S_0; T_0$ is well-typed in M_0 and the operational semantics takes any number of steps $S_0; T_0 \rightsquigarrow^n S_n; T_n$, then the resulting configuration $S_n; T_n$ is not stuck and T_n has not reached a deadlocked state.

Proof. The application of Lemma 6.4 to the typing derivation of $S_0; T_0$ implies that for all steps from zero to n there exists an $M_\imath$ such that $M_\imath \vdash S_\imath; T_\imath$. Therefore, Lemma 6.1 implies that $\neg\mathsf{deadlocked}(T_n)$ and Lemma 6.2 implies $S_n; T_n$ is not stuck.

Typing the initial configuration $S_0; T_0$ with the empty typing context M_0 guarantees that all functions in the program are closed and that no explicit location values ($\mathtt{loc}_\imath$) are used in the source of the original program.

6.6 Concluding remarks

The main contribution of this work is type-based deadlock avoidance for a language with unstructured locking primitives and the meta-theory for the proposed semantics. The type system presented in this chapter guarantees that well-typed programs will not deadlock at execution time. This is possible by statically verifying that program annotations reflect the order of future lock operations and using the annotations at execution time to avoid deadlocks. The main advantage over purely static approaches to deadlock freedom is that our type system accepts a wider class of programs as it does not enforce a total order on lock acquisition. The main disadvantages of our approach is that it imposes an additional run-time overhead induced by the future lockset computation and blocking time (i.e., both the requested lock and its future lockset must be available). Additionally, in some cases threads may unnecessarily block because our type and effect system is conservative. For example, when a

thread locks x and executes a lengthy computation (without acquiring other locks) before releasing x, it would be safe to allow another thread to lock y even if x is in its future lockset.

We have shown that this is a non-trivial extension for existing type systems based on deadlock avoidance. There are three significant sources of complexity: (i) lock acquisition and release operations may not be properly nested, (ii) lock-unlock pairs may span multiple contexts: function calls that contain lock operations may not always *increase* the size of lockset, but instead *limit* the lockset size. In addition, future locksets must be computed in a context-sensitive manner (stack traversal in our case), and (iii) in the presence of location (lock) polymorphism and aliasing, it is very difficult for a static type system even to detect the previous two sources of complexity. To address lock aliasing without imposing restrictions statically, we defer lockset resolution until run-time.

Chapter 7

Effect inference for deadlock freedom

7.1 Overview

The type and effect system presented in the previous chapter, requires explicit and complex type annotations from the programmer. Its effects contain elements that are pairs (n_1, n_2) associating memory cells with two capability counts: n_1 is a cell reference count, denoting whether the cell is live, while n_2 is the lock count, denoting how many times the cell has been locked (as locks are re-entrant). In addition, capabilities can be either unique or possibly aliased: the type system requires aliasing information so as to determine whether it is safe to pass lock capabilities to new threads. As a result, that type system is probably unsuitable for a language like C/pthreads; instead, it is relevant for a language like Cyclone [Gros02] where it is commonplace for functions to have annotations.

In contrast, the type and effect system we develop in this chapter is much simpler. It focuses on deadlock avoidance only, captures the temporal order of lock and unlock operations, and imposes no restrictions with respect to aliasing. More importantly, its implementation is amenable to effect *inference*, and there is no requirement that functions are annotated with explicit effects. Instead, the type and effect system gathers effects and validates them at the beginning of the lexical scope of each lock. This simpler system is thus directly applicable to C/pthreads programs.

7.2 Formal semantics and metatheory

The syntax of our language is illustrated in Figure 7.1, where x and ρ range over term and lock variables, respectively, and $\imath$ ranges over lock constants. In this chapter, to make the presentation as simple as possible, we do not include any mutable shared state in our language. In other words, we study locks in isolation: locks do not serve any other purpose than thread synchronization (mutual exclusion). Without shared mutable references, locks may seem a bit pointless. However, our primary goal is to develop a simple and understandable type and effect system that guarantees deadlock avoidance. Including shared memory and achieving other interesting properties, such as memory safety and data race freedom, are goals which are more or less orthogonal to deadlock freedom and have already been presented in the previous chapter.

The language core comprises of constants (`true`, `false` and $()$ — the "unit" value), functions (f), and function application. Functions can be monomorphic ($\lambda x.\, e$), lock polymorphic ($\Lambda\rho.\, f$), and recursive (`fix` $x.\, f$). The application of lock polymorphic functions is explicit ($e[r]$, where r is a metavariable ranging over lock constants and variables). The application of monomorphic functions is annotated with a *calling mode* (ξ), which is seq for normal sequential application and par for parallel application.[1] The semantics of parallel application is that, once the application term is evaluated to a redex, it is moved to a new thread of execution and the spawning thread can proceed with the remaining computation in parallel with the new thread. Conditional expressions (`if` e `then` e_1 `else` e_2) are standard.

The construct `newlock` ρ, x `in` e allocates a fresh lock, which is initially unlocked, and associates it with variables ρ and x within expression e. The type variable ρ is bound to the type-level representation

[1] Notice that sequential application terms are annotated with γ, the *continuation effect*.

$$
\begin{array}{lll}
\textbf{Expression} & e & ::= x \mid v \mid (e\ e)^{\xi} \mid (e)\,[r] \mid \mathtt{pop}_{\gamma}\ e \\
& & \quad \mid \mathtt{newlock}\ \rho, x\ \mathtt{in}\ e \\
& & \quad \mid \mathtt{lock}_{\gamma}\ e \mid \mathtt{unlock}\ e \\
& & \quad \mid \mathtt{if}\ e\ \mathtt{then}\ e\ \mathtt{else}\ e \\
\textbf{Value} & v & ::= () \mid \mathtt{true} \mid \mathtt{false} \mid f \mid \mathtt{lk}_{\iota} \\
\textbf{Function} & f & ::= \lambda x.\, e \mid \Lambda\rho.\, f \mid \mathtt{fix}\ x.\, f \\
\textbf{Type} & \tau & ::= \langle\rangle \mid \mathtt{Bool} \mid \mathtt{Lk}(r) \mid \tau \xrightarrow{\gamma} \tau \mid \forall\rho.\, \tau
\end{array}
\qquad
\begin{array}{lll}
\textbf{Lock} & r & ::= \rho \mid \iota \\
\textbf{Calling mode} & \xi & ::= \mathsf{seq}(\gamma) \mid \mathsf{par} \\
\textbf{Operation} & \kappa & ::= + \mid - \\
\textbf{Effect} & \gamma & ::= \emptyset \mid r^{\kappa}, \gamma \mid \gamma\,?\,\gamma, \gamma
\end{array}
$$

Figure 7.1: Language and type syntax.

$$
\begin{array}{lll}
\textbf{Lock Store} & S & ::= \emptyset \mid S, \iota \mapsto n; n; \epsilon; \epsilon \\
\textbf{Threads} & T & ::= \emptyset \mid T, n : e \\
\textbf{Configuration} & C & ::= S; T \\
\textbf{Lockset} & \epsilon & ::= \emptyset \mid \epsilon, \iota
\end{array}
\qquad
\begin{array}{lll}
\textbf{Context} & E & ::= \Box \mid E[F] \\
\textbf{Frame} & F & ::= (\Box\ e)^{\xi} \mid (v\ \Box)^{\xi} \mid (\Box)\,[r] \mid \mathtt{pop}_{\gamma}\ \Box \\
& & \quad \mid \mathtt{lock}_{\gamma_1}\ \Box \mid \mathtt{unlock}\ \Box \\
& & \quad \mid \mathtt{if}\ \Box\ \mathtt{then}\ e\ \mathtt{else}\ e
\end{array}
$$

Figure 7.2: Operational semantics syntax and evaluation context.

of the fresh lock and allows the type system to statically track uses of it, whereas the term variable x is bound to the fresh lock's handle. Handles can be used as arguments in operations $\mathtt{lock}_{\gamma}\, e$ and $\mathtt{unlock}\, e$. It is worth noting that run-time locks are re-entrant, so each lock is associated with a count which is modified after each successful lock/unlock operation. As mentioned, the run-time system inspects the lock annotation γ to determine whether it is safe to lock e.

The term $\mathtt{pop}_{\gamma}\ e$ encloses a function body e and cannot exist at the source-level; it only appears during evaluation. The same applies to constant lock handles $\mathtt{lk}_{\iota}$.

The syntax of types is more or less standard; a function's type is annotated with the function's effect. Effects (γ) are sequences of events. An atomic event can either be r^{+} or r^{-}, representing acquire and release operations on a lock handle of type $\mathtt{Lk}(r)$. Events also include $\gamma_1\,?\,\gamma_2$, where γ_1 and γ_2 are the continuation effects corresponding to the two branches of a conditional expression.

7.3 Operational semantics

We define a *small-step* operational semantics for our language in Figure 7.2 and 7.3.[2] The evaluation relation transforms *configurations*. A configuration C consists of an abstract *lock store* S and a thread map T.[3] A store S maps constant locks (ι) to tuples of the form $(n_1; n_2; \epsilon_1; \epsilon_2)$. The first two elements of the tuple are natural numbers representing the thread identifier that owns ι and the count of ι, respectively. The remaining two elements are locksets; they bear no operational significance but are necessary for the type safety proof. The first lockset (ϵ_1) represents the set of all locks in S when ι was last locked (when its n_2 went from zero to one). The second lockset (ϵ_2) represents the future lockset of ι when it was last locked.

A thread map T associates thread identifiers to expressions (i.e., threads). A *frame* F is an expression with a *hole*, represented as $\Box$. The hole indicates the position where the next reduction step can take place. A *thread evaluation context* E is defined as a stack of nested frames. Our notion of evaluation context imposes a call-by-value evaluation strategy to our language. Subexpressions are evaluated in a left-to-right order. We assume that concurrent reduction events can be totally ordered [Lamp79]. At each step, a *random* thread (n) is chosen from the thread list for evaluation. Therefore, the evaluation rules are *non-deterministic*.

[2] A full formalization is given in Appendix D.

[3] The order of elements in comma-separated lists, e.g., in a store S or in a list of threads T, is unimportant; we consider all list permutations as equivalent. However, in sequences (e.g., effects), order is important.

$$\frac{\text{fresh } n'}{S;T,n:E[(v'\ \ v)^{\mathsf{par}}] \rightsquigarrow S;T,n:E[()],n':\Box[(v'\ \ v)^{\mathsf{seq}(\emptyset)}]} \quad \textit{(E-SN)}$$

$$\frac{}{S;T,n:\Box[()] \rightsquigarrow S;T} \quad \textit{(E-T)}$$

$$\frac{}{S;T,n:E[((\lambda x.\, e_1)\ \ v)^{\mathsf{seq}(\gamma)}] \rightsquigarrow S;T,n:E[\mathsf{pop}_\gamma\ e_1[v/x]]} \quad \textit{(E-A)}$$

$$\frac{}{S;T,n:E[\mathsf{pop}_\gamma\ v] \rightsquigarrow S;T,n:E[v]} \quad \textit{(E-PP)}$$

$$\frac{}{S;T,n:E[(\Lambda\rho.\, f)[\imath]] \rightsquigarrow S;T,n:E[f[\imath/\rho]]} \quad \textit{(E-RP)}$$

$$\frac{v' = \mathtt{fix}\ x.\, f}{S;T,n:E[(v'\ \ v)^{\mathsf{seq}(\gamma)}] \rightsquigarrow S;T,n:E[(f[v'/x]\ \ v)^{\mathsf{seq}(\gamma)}]} \quad \textit{(E-FX)}$$

$$\frac{}{S;T,n:E[\mathtt{if\ true\ then}\ e_1\ \mathtt{else}\ e_2] \rightsquigarrow S;T,n:E[e_1]} \quad \textit{(E-IT)}$$

$$\frac{}{S;T,n:E[\mathtt{if\ false\ then}\ e_1\ \mathtt{else}\ e_2] \rightsquigarrow S;T,n:E[e_2]} \quad \textit{(E-IF)}$$

$$\frac{\text{fresh } \imath \qquad S' = S,\imath \mapsto n;0;\emptyset;\emptyset}{S;T,n:E[\mathtt{newlock}\ \rho,x\ \mathtt{in}\ e_1] \rightsquigarrow S';T,n:E[e_1[\imath/\rho][\mathtt{lk}_\imath/x]]} \quad \textit{(E-NG)}$$

$$\frac{\begin{array}{c}S(\imath) = n_1;0;\epsilon_1;\epsilon_2 \qquad S' = S[\imath \mapsto n;1;\mathsf{dom}(S);\epsilon] \\ \epsilon = \mathsf{run}(\mathsf{stack}(E[\mathsf{pop}_{\gamma_1}\ \Box]),\imath,1) \qquad \epsilon \cup \{\imath\} \subseteq \mathsf{available}(S,n)\end{array}}{S;T,n:E[\mathtt{lock}_{\gamma_1}\ \mathtt{lk}_\imath] \rightsquigarrow S';T,n:E[()]} \quad \textit{(E-LK0)}$$

$$\frac{S(\imath) = n;n_2;\epsilon_1;\epsilon_2 \qquad n_2 > 0 \qquad S' = S[\imath \mapsto n;n_2+1;\epsilon_1;\epsilon_2]}{S;T,n:E[\mathtt{lock}_{\gamma_1}\ \mathtt{lk}_\imath] \rightsquigarrow S';T,n:E[()]} \quad \textit{(E-LK1)}$$

$$\frac{S(\imath) = n;n_2;\epsilon_1;\epsilon_2 \qquad n_2 > 0 \qquad S' = S[\imath \mapsto n;n_2-1;\epsilon_1;\epsilon_2]}{S;T,n:E[\mathtt{unlock}\ \mathtt{lk}_\imath] \rightsquigarrow S';T,n:E[()]} \quad \textit{(E-UL)}$$

Figure 7.3: Operational semantics.

When a parallel function application redex is detected within the evaluation context of a thread, a new thread is created (rule *E-SN*). The redex is replaced with the unit value in the currently executed thread and a new thread is added to the thread list, with a *fresh* thread identifier. The calling mode of the application term is changed from parallel to sequential, with an empty continuation effect. When evaluation of a thread reduces to a unit value, the thread is removed from the thread list (rule *E-T*). The sequential function application rule (*E-A*) reduces an application redex to a `pop` expression, which contains the body of the function and is annotated with the same effect as the application term. `Pop` expressions are used to form the run-time stack of continuation effects. When the expression contained within a `pop` has been reduced to a value, then enclosing `pop` is removed and the value is returned to the context (rule *E-PP*). The rules for evaluating the application of polymorphic functions (*E-RP*) and recursive functions (*E-FX*) are standard, as well as the rules for evaluating conditionals (*E-IT* and *E-IF*). Rule *E-NG* appends to S a fresh lock $\imath$, which is initially unlocked.

The most interesting rule is *E-LK0*, which dynamically computes the future lockset (ϵ) of lock $\imath$. To achieve this, function stack (defined in Figure 7.4) assembles the overall (stacked) continuation

$$
\begin{array}{lcl}
\mathsf{run}(\gamma, \imath, n) & = & \begin{cases}
\emptyset & \text{if } n = 0 \\
\mathsf{run}(\gamma', \imath, n+1) & \text{if } \gamma = \imath^+, \gamma' \text{ and } n > 0 \\
\mathsf{run}(\gamma', \imath, n-1) & \text{if } \gamma = \imath^-, \gamma' \text{ and } n > 0 \\
\mathsf{run}(\gamma', \imath, n) \cup \{\mathtt{j}\} & \text{if } \gamma = \mathtt{j}^+, \gamma' \text{ and } n > 0 \\
\mathsf{run}(\gamma', \imath, n) & \text{if } \gamma = \mathtt{j}^-, \gamma' \text{ and } n > 0 \\
\mathsf{run}((\gamma_1 :: \gamma'), \imath, n) \cup \mathsf{run}((\gamma_2 :: \gamma'), \imath, n) & \text{if } \gamma = \gamma_1 \,?\, \gamma_2, \gamma' \text{ and } n > 0
\end{cases} \\
\mathsf{stack}(E) & = & \begin{cases}
\emptyset & \text{if } E = \Box \\
\mathsf{stack}(E') & \text{if } E = E'[F] \text{ and } F \neq \mathsf{pop}_{\gamma'}\ \Box \\
\gamma' :: \mathsf{stack}(E') & \text{if } E = E'[\mathsf{pop}_{\gamma'}\ \Box]
\end{cases} \\
\mathsf{available}(S, n) & = & \begin{cases}
\emptyset & \text{if } S = \emptyset \\
\mathsf{available}(S', n) \cup \{\imath\} & \text{if } S = S', \imath \mapsto n_1; n_2; \epsilon_1; \epsilon_2 \text{ and } n_1 = n \text{ or } n_2 = 0 \\
\mathsf{available}(S', n) & \text{if } S = S', \imath \mapsto n_1, n_2; \epsilon_1; \epsilon_2 \text{ and } n_1 \neq n \text{ and } n_2 > 0
\end{cases} \\
\mathsf{dom}(S) & = & \{\imath \mid \imath \mapsto n_1; n_2; \epsilon_a; \epsilon_b \in S\}
\end{array}
$$

Figure 7.4: Operational semantics helper relations.

effect by concatenating the continuation effect annotations of pop expressions that are found in the stack of the evaluation context. The lockset computation is modeled by function *run*$(\gamma, \imath, k)$ (defined in Figure 7.4), which accepts the stacked effect γ, the lock $\imath$ whose lockset is to be computed and the number k of unmatched unlock events ($\imath^-$) in the stack. It returns a subset of the lock events (r^+) located in the stack, such that each element of the subset is locked *before* the last unmatched unlock operation of $\imath$. Function run is defined only when all unlock events for $\imath$ are found in the stacked effect. The future lockset of $\imath$ (ϵ) is equal to *run*$(\gamma, \imath, 1)$. Rule *E-LK0* also requires that both $\imath$ and its future lockset are available — $\epsilon \cup \{\imath\} \subseteq$ *available*(S, n). Function available (also defined in Figure 7.4) takes as input a lock store S and a thread identifier n and returns a set of locks, such that each element of the set can be acquired by thread n (i.e., locks whose thread identifier either equals n or their count is zero). If the availability premise holds, the lock count of $\imath$ is set to one and the thread identifier is set to n. In addition, both ϵ_1 and ϵ_2 (the last two elements of $S(\imath)$) are replaced with *dom*(S) (all locks allocated in the program) and ϵ, respectively.

The rules for acquiring or releasing a held lock (*E-LK1* or *E-UL*) require that the count of that lock is positive and that it is owned by the thread that is performing the unlock/lock operation. Otherwise, the semantics will get stuck. We will soon present a type system for this language and also the type safety formulation that guarantees that well-typed programs cannot reach a stuck state.

Note that, although rule *E-LK0* ensures that all locks in the future lockset ϵ are available before proceeding, our semantics only acquires the requested lock $\imath$ and not any of the locks in ϵ. As a possible optimization, an implementation could choose to acquire additionally some subset $\epsilon' \subseteq \epsilon$. These locks are all available at this point and an implementation might not want to recheck for their availability and more importantly risk having to wait for them at the time they are needed, in case some other thread has got hold of them until then. Pre-acquisition of locks, however, may reduce parallelism and an implementation should use it only when an analysis shows that the locks will definitely be needed, and not "too late" in the future. (Additional information could statically be placed in the effects to guide such an implementation.) The type safety of our system, stated in Section 7.6, can be proved even if the semantics pre-acquires a subset of the future lockset in this rule.

$$\frac{M;\Delta \vdash \Gamma \qquad M;\Delta \vdash \gamma}{M;\Delta;\Gamma \vdash () : \langle\rangle \,\&\, (\gamma;\gamma)} \textit{(T-U)} \qquad \frac{x : \tau \in \Gamma \qquad M;\Delta \vdash \Gamma \qquad M;\Delta \vdash \gamma}{M;\Delta;\Gamma \vdash x : \tau \,\&\, (\gamma;\gamma)} \textit{(T-V)}$$

$$\frac{M;\Delta \vdash \Gamma \qquad M;\Delta \vdash \gamma}{M;\Delta;\Gamma \vdash \texttt{true} : \texttt{Bool} \,\&\, (\gamma;\gamma)} \textit{(T-T)} \qquad \frac{M;\Delta \vdash \Gamma \qquad M;\Delta \vdash \gamma}{M;\Delta;\Gamma \vdash \texttt{false} : \texttt{Bool} \,\&\, (\gamma;\gamma)} \textit{(T-F)}$$

$$\frac{\begin{array}{c} M;\Delta \vdash \Gamma \qquad M;\Delta \vdash \gamma \qquad \tau \equiv \tau_1 \xrightarrow{\gamma_b} \tau_2 \\ M;\Delta \vdash \tau \qquad M;\Delta;\Gamma, x : \tau_1 \vdash e_1 : \tau_2 \,\&\, (\emptyset;\gamma_b) \end{array}}{M;\Delta;\Gamma \vdash \lambda x.\, e_1 : \tau \,\&\, (\gamma;\gamma)} \textit{(T-FN)}$$

$$\frac{M;\Delta,\rho;\Gamma \vdash f : \tau \,\&\, (\gamma;\gamma)}{M;\Delta;\Gamma \vdash \Lambda\rho.\, f : \forall\rho.\, \tau \,\&\, (\gamma;\gamma)} \textit{(T-RF)}$$

$$\frac{r \in M \cup \Delta \qquad M;\Delta;\Gamma \vdash e_1 : \forall\rho.\, \tau \,\&\, (\gamma;\gamma')}{M;\Delta;\Gamma \vdash (e_1)\,[r] : \tau[r/\rho] \,\&\, (\gamma;\gamma')} \textit{(T-RP)}$$

$$\frac{M;\Delta \vdash \gamma \qquad M;\Delta;\Gamma \vdash e : \tau \,\&\, (\emptyset;\gamma')}{M;\Delta;\Gamma \vdash \mathtt{pop}_\gamma\ e : \tau \,\&\, (\gamma;\gamma' :: \gamma)} \textit{(T-PP)}$$

$$\frac{M;\Delta \vdash \Gamma \qquad M;\Delta \vdash \gamma \qquad \iota \in M}{M;\Delta;\Gamma \vdash \mathtt{lk}_\iota : \mathtt{Lk}(\iota) \,\&\, (\gamma;\gamma)} \textit{(T-L)} \qquad \frac{M;\Delta;\Gamma \vdash e : \mathtt{Lk}(r) \,\&\, (r^+,\gamma;\gamma')}{M;\Delta;\Gamma \vdash \mathtt{lock}_\gamma\, e : \langle\rangle \,\&\, (\gamma;\gamma')} \textit{(T-LK)}$$

$$\frac{M;\Delta;\Gamma \vdash e : \mathtt{Lk}(r) \,\&\, (r^-,\gamma;\gamma')}{M;\Delta;\Gamma \vdash \mathtt{unlock}\, e : \langle\rangle \,\&\, (\gamma;\gamma')} \textit{(T-UL)} \qquad \frac{\begin{array}{c} M;\Delta;\Gamma \vdash e_1 : \tau_1 \xrightarrow{\gamma_a} \tau_2 \,\&\, (\gamma_1;\gamma') \\ M;\Delta;\Gamma \vdash e_2 : \tau_1 \,\&\, (\gamma_a :: \gamma;\gamma_1) \end{array}}{M;\Delta;\Gamma \vdash (e_1\ e_2)^{\mathsf{seq}(\gamma)} : \tau_2 \,\&\, (\gamma;\gamma')} \textit{(T-SA)}$$

$$\frac{\begin{array}{c} \forall r \in \mathsf{dom}(\gamma_a).\, r;0 \vdash_{ok} \gamma_a \\ M;\Delta;\Gamma \vdash e_1 : \tau_1 \xrightarrow{\gamma_a} \langle\rangle \,\&\, (\gamma_1;\gamma') \qquad M;\Delta;\Gamma \vdash e_2 : \tau_1 \,\&\, (\gamma;\gamma_1) \end{array}}{M;\Delta;\Gamma \vdash (e_1\ e_2)^{\mathsf{par}} : \langle\rangle \,\&\, (\gamma;\gamma')} \textit{(T-PA)}$$

$$\frac{\begin{array}{c} M;\Delta \vdash \tau \qquad \rho \notin \mathsf{dom}(\gamma) \qquad \rho;0 \vdash_{ok} \gamma' \\ M;\Delta,\rho;\Gamma, x : \mathtt{Lk}(\rho) \vdash e_1 : \tau \,\&\, (\gamma;\gamma') \end{array}}{M;\Delta;\Gamma \vdash \mathtt{newlock}\ \rho, x\ \mathtt{in}\ e_1 : \tau \,\&\, (\gamma;\gamma' \setminus \rho)} \textit{(T-NG)}$$

$$\frac{\begin{array}{c} R;M;\Delta;\Gamma, x : \tau_a \vdash f : \tau_b \,\&\, (\gamma;\gamma) \\ \tau_a \equiv \tau_1 \xrightarrow{\gamma_a} \tau_2 \qquad \tau_b \equiv \tau_1 \xrightarrow{\gamma_b} \tau_2 \qquad \gamma_a = \mathsf{summary}(\gamma_b) \end{array}}{R;M;\Delta;\Gamma \vdash \mathtt{fix}\ x.\, f : \tau_a \,\&\, (\gamma;\gamma)} \textit{(T-FX)}$$

$$\frac{\begin{array}{c} M;\Delta;\Gamma \vdash e_1 : \mathtt{Bool} \,\&\, (\gamma_1\,?\,\gamma_2,\gamma;\gamma') \\ M;\Delta;\Gamma \vdash e_2 : \tau \,\&\, (\gamma;\gamma_1 :: \gamma) \qquad M;\Delta;\Gamma \vdash e_3 : \tau \,\&\, (\gamma;\gamma_2 :: \gamma) \end{array}}{M;\Delta;\Gamma \vdash \mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3 : \tau \,\&\, (\gamma;\gamma')} \textit{(T-IF)}$$

Figure 7.5: Typing rules.

$$\frac{}{M;\Delta \vdash \emptyset} \qquad \frac{r \in M \cup \Delta \quad M;\Delta \vdash \gamma_1}{M;\Delta \vdash \gamma_1, r^{\kappa}} \qquad \frac{M;\Delta \vdash \gamma_1 \quad M;\Delta \vdash \gamma_2 \quad M;\Delta \vdash \gamma_3}{M;\Delta \vdash \gamma_1, \gamma_2 \,?\, \gamma_3}$$

$$\frac{}{M;\Delta \vdash \texttt{Bool}} \qquad \frac{M;\Delta, \rho \vdash \tau}{M;\Delta \vdash \forall \rho.\, \tau} \qquad \frac{r \in M \cup \Delta}{M;\Delta \vdash \texttt{Lk}(r)}$$

$$\frac{M;\Delta \vdash \tau_1 \quad M;\Delta \vdash \gamma_1 \quad M;\Delta \vdash \tau_2}{M;\Delta \vdash \tau_1 \xrightarrow{\gamma_1} \tau_2} \qquad \frac{}{M;\Delta \vdash \langle\rangle}$$

$$\frac{}{M;\Delta \vdash \emptyset} \qquad \frac{M;\Delta \vdash \tau_1 \quad x \notin \mathsf{dom}(\Gamma_1) \quad M;\Delta \vdash \Gamma_1}{M;\Delta \vdash \Gamma_1, x : \tau_1}$$

Figure 7.6: Well formedness.

7.4 Static semantics

The syntax of types and effects is given in Figure 7.1 (on page 96). Basic types consist of the boolean and the unit type, denoted by $\langle\rangle$; lock handle types $\texttt{Lk}(r)$ are singleton types parameterized by a type-level lock name r; and monomorphic function types carry the function's effect. Effects (γ) are used to statically track lock ownership information; they are ordered *sequences* of events, which can be either r^{κ} or $\gamma_1 \,?\, \gamma_2$.

The typing relation is denoted by $R; M; \Delta; \Gamma \vdash e : \tau \,\&\, (\gamma;\gamma')$. It takes an expression e, the typing context $R; M; \Delta; \Gamma$, and an input effect γ, and produces the type τ assigned to expression e as well as an output effect γ'. Here, M is a set of lock constants, Δ is a set of lock variables, and Γ is a mapping of term variables to types.

As lock operations and application terms are annotated with their continuation effect, it is natural that effects *flow backwards* through the type system: the input effect to an expression e represents the events that follow in the future of e, that is, after e is evaluated. On the other hand, the output effect represents the combined sequence of events caused by e and its future. In fact, the typing relation does not modify the input effect but rather appends to it: the input effect is always a *suffix* of the output effect, in chronological order. (This is ensured by the typing relation and the typing context well-formedness.) The typing rules are given in Figure 7.5.[4] The typing rules *T-U*, *T-T*, *T-F*, *T-V*, *T-L*, *T-RF*, *T-RP* and *T-FN* are standard. Notice, that in the case of rule *T-FN*, the input effect of the function's body e_1 is empty. The typing rule for sequential function application (*T-SA*) appends the input effect γ to the function's effect γ_a and propagates the new effect to expression e_2, which in turn propagates its output effect to e_1. The output effect of the sequential function application is the output effect of expression e_1. The annotation of the application must match with the input effect γ. Rule *T-PP* acts as a bridge between the body of a function that is being executed and its calling environment, by appending the continuation effect to the effect of the function's body. The rule for parallel application (*T-PA*) is similar to the sequential application rule, except that the function's effect (γ_a) is not combined with the input effect (as the function will be evaluated in a new thread) and the function's return type must be unit. In addition, all locks in the function's effect must be released before and after the function's execution — $\forall r.\, r; 0 \vdash_{ok} \gamma_a$. The relation $r; n \vdash_{ok} \gamma$ checks that there exist exactly n unmatched unlock events in γ for lock r (it is used at the same time to make sure that r is never released more times than it has been acquired).

The rule for typing recursive functions (*T-FX*) is the standard one, if we ignore the effects γ_a and γ_b on the function types. As mentioned in Section 5.3, it may be impossible to assign the recursive function variable x the same effect as the function body f (i.e., γ_b). The intuition here is that x must be assigned an effect γ_a that *summarizes* γ_b, and this effect can be computed as a least fixed point with

[4] A complete formalization appears in the Appendix D.

the procedure that was sketched in Section 5.3. We postpone the discussion on summaries for a little longer, until Section 7.5.

The rule for creating new locks (*T-NG*) passes the input effect γ to e_1, the body of `let`, assigns the lock handle variable x the singleton type $\texttt{Lk}(\rho)$ and adds ρ to the lock variable context for the scope of e_1. The output effect of the lock creation construct is equal to the output effect of e_1 minus any events of the form ρ^κ. The rule also requires that ρ is unlocked before and after the execution of $e_1 \text{ — } \rho; 0 \vdash_{ok} \gamma'$. Rule *T-LK* prepends r^+ to the input effect and propagates the resulting effect to e_1. Notice, that the input effect must match the lock annotation (the continuation effect of the lock operation must be valid). The typing rule *T-UL* prepends r^- to the input effect and propagates the resulting effect to e_1.

The typing rule for conditional expressions (*T-IF*) propagates the input effect of the conditional expression to its branches e_2 and e_3 respectively. We know that γ is a common suffix of the output effect of e_2 and e_3. Let us assume that γ_1 and γ_2 are the prefixes of the two branches respectively. Thus, the input effect of the guard expression e_1 is $\gamma_1 \,?\, \gamma_2, \gamma$, which tells us that the type system records the effects of both branches but it does not unify them.

The typing rules *T-SA*, *T-LK* and *T-PP* ensure that the effect annotations in sequential applications, `lock` and `pop` expressions are equal to the expression's input effect. This means that, even in this language (and much more so in a language like C), programmers are not really expected to explicitly annotate such expressions: it is easy for the type and effect system to infer the annotations.

7.5 Summarizing recursive functions

We have already discussed why it is necessary to summarize the effects of recursive functions. However, the function summary can be correctly defined in different ways. In principle, any possible definition will do, as long as it satisfies Lemmata 7.1 and 7.2.

Lemma 7.1 (Consistency of Summary) Let σ be a substitution of lock variables with lock constants and γ_s be a continuation effect. If $\gamma_a = \mathit{summary}(\gamma_b)$ then for all $\imath$ and n we have

$$\mathit{run}(\sigma(\gamma_b :: \gamma_s), \imath, n) \subseteq \mathit{run}(\sigma(\gamma_a :: \gamma_s), \imath, n)$$

Before we proceed to Lemma 7.2, we provide an informal definition for function startup. This function takes an effect γ and finds all unmatched lock and unlock operations in γ. It produces an effect γ' which has all the unmatched lock operations, followed by all the unmatched unlock operations. E.g.

$$\mathsf{startup}([x+,\, x-] \,?\, [y+,\, y-]) = \emptyset$$
$$\mathsf{startup}([z-,\, y+] \,?\, [x+,\, y+,\, x-,\, z-]) = [y+,\, z-]$$

We can also define the notion of compositionality for functions on effects. Informally, a function $F(\gamma)$ is *compositional* if γ can only be used as a sub-effect in the result (i.e. in the way that our type and effect system uses effects).

Lemma 7.2 (Fixed Point of Summary) Let $F(\gamma)$ be a compositional function, $\gamma_0 = \mathsf{startup}(F(\emptyset))$, and $\gamma_{n+1} = \mathsf{summary}(F(\gamma_n))$. Then there exists a k such that for all $n > k$ we have $\gamma_k = \gamma_n$.

If a summary function satisfies Lemma 7.2, then the procedure described in Section 5.3 can be used to compute the fixed point of all recursive functions. This fixed point can be used by the type system to determine type τ_a in rule *T-FX*. Furthermore, if a summary function satisfies Lemma 7.1, then it is safe for the run-time system to use the summarized effect in the place of the real effect of a function's body. In all cases, the future lockset that will be computed based on the summary will be a superset of the future lockset that would be computed based on the body's real effect.

We use the conservative function summary (see Figure 7.7) that can be shown to satisfy Lemmata 7.1 and 7.2. For any effect γ, we define $\mathsf{summary}(\gamma)$ as follows. We take $\mathsf{startup}(\gamma)$ and split it

in two components: γ_+, which contains the unmatched lock operations, and γ_-, which contains the unmatched unlock operations. We reorder the events in γ_+ and γ_- using any total order relation on lock variables ρ. (This *normalization* is required for ensuring that a fixed point exists — Lemma 7.2.) We then build a third component: γ_0, which contains one pair of $[x+,\, x-]$ for each lock x that is acquired at any time in γ, excluding the ones that are in γ_+. Again, we normalize γ_0 by reordering the events that it contains. Finally, we take $\mathsf{summary}(\gamma) = \gamma_+ :: \gamma_0 :: \gamma_-$.

As an example, consider the following conditional statement:

$$\begin{array}{l}\texttt{if}\ e\ \texttt{then}\ (\texttt{lock}\ x;\ \ldots\ \texttt{lock}\ y;\ \ldots\ \texttt{unlock}\ y)\\ \qquad\texttt{else}\ (\texttt{lock}\ z;\ \ldots\ \texttt{lock}\ x;\ \ldots\ \texttt{unlock}\ z)\end{array}$$

The corresponding effect is:

$$\gamma = [x+,\, y+,\, y-]\,?\,[z+,\, x+,\, z-]$$

There is one unmatched lock operation (for x, which occurs in both branches of the conditional), therefore $\mathsf{startup}(\gamma) = [x+]$. We take $\gamma_+ = [x+]$ and $\gamma_- = \emptyset$. Then, we build $\gamma_0 = [y+,\, y-,\, z+,\, z-]$, by taking one matching pair for each of the lock operations that occur in γ and are not contained in γ_+ (these are $y+$ and $z+$, and we order lock variables lexicographically). Thus:

$$\mathsf{summary}(\gamma) = [x+,\, y+,\, y-,\, z+,\, z-]$$

More accurate summary functions can also be constructed, not merging branching effects and respecting the nested structure of lock/unlock operations. However, we are not convinced of their practical importance and, in particular, whether the future locksets $\mathit{run}(\sigma(\gamma_a :: \gamma_s), \imath, n)$ that they produce are indeed more accurate.

As a last note here, summarization is not only necessary for dealing with recursive functions. It is useful for reducing the size of the effects of non-recursive functions, to improve the performance of the run-time system.

7.6 Type safety and deadlock freedom

In this section we present the fundamental theorems that prove type safety for our language, together with very brief proof sketches.[5] Type safety, which in this system implies deadlock freedom, is based on proving the *preservation*, *deadlock freedom* and *progress* lemmata. Informally, a program written in our language is safe when each thread of execution can perform an evaluation step or is waiting for a lock (*blocked*). In addition, there must not exist threads that have reached a deadlocked state.

As discussed in Section 7.3, a thread may become stuck when it performs an ill-typed operation, or when it attempts to compute the future lockset of a malformed stack, or when it attempts to acquire a non-existing lock, or when it attempts to release a lock whose count has already reached the value zero, and so on.

Definition 7.1 (Thread Effect Consistency) The following rules define *effect-consistent* threads.

$$\frac{\imath; n_1 \vdash_{ok} \gamma \qquad \epsilon_3 = \mathsf{run}(\gamma, \imath, n_1) \quad n; \gamma \vdash S \qquad \epsilon_1 \cap \epsilon_3 \subseteq \epsilon_2}{n; \gamma \vdash S, \imath \mapsto n; n_1; \epsilon_1; \epsilon_2} \qquad \frac{\imath; 0 \vdash_{ok} \gamma \qquad n \neq n_1 \qquad n; \gamma \vdash S}{n; \gamma \vdash S, \imath \mapsto n_1; n_2; \epsilon_1; \epsilon_2} \qquad \frac{}{n; \gamma \vdash \emptyset}$$

Thread effect consistency (denoted by $n; \gamma \vdash S$) ensures that any lock acquired by thread n will be released before thread n terminates. Furthermore, it establishes an exact correspondence between locks in γ and S. In particular, for each lock $\imath$ in the domain of γ, $\imath; n_1 \vdash_{ok} \gamma$ (see Figure 7.7) must hold, where n_1 must equal the reference count of $\imath$ in S for each thread n. Notice that only one thread

[5] A full formalization of our language and complete proofs are given in Appendix D.

$$\frac{\gamma' = \gamma \setminus r}{\gamma' = r^{\kappa}, \gamma \setminus r}\ (M0) \qquad \frac{r' \neq r \quad \gamma' = \gamma \setminus r'}{r^{\kappa}, \gamma' = r^{\kappa}, \gamma \setminus r'}\ (M1) \qquad \frac{}{\emptyset = \emptyset \setminus r}\ (M3)$$

$$\frac{\gamma_1' = \gamma_1 \setminus r' \quad \gamma_2' = \gamma_2 \setminus r' \quad \gamma_3' = \gamma_3 \setminus r'}{\gamma_2' \,?\, \gamma_3', \gamma_1' = \gamma_2 \,?\, \gamma_3, \gamma_1 \setminus r}\ (M4)$$

$$\frac{0 \leq n \quad r; n+1 \vdash_{ok} \gamma}{r; n \vdash_{ok} r^{+}, \gamma}\ (OK1) \qquad \frac{r; n-1 \vdash_{ok} \gamma \quad n > 0}{r; n \vdash_{ok} r^{-}, \gamma}\ (OK2)$$

$$\frac{0 \leq n \quad r; n \vdash_{ok} \gamma \quad r \neq r'}{r; n \vdash_{ok} r'^{\kappa}, \gamma}\ (OK3)$$

$$\frac{}{r; 0 \vdash_{ok} \emptyset}\ (OK4) \qquad \frac{0 \leq n \quad r; n \vdash_{ok} \gamma_1 :: \gamma \quad r; n \vdash_{ok} \gamma_2 :: \gamma}{r; n \vdash_{ok} \gamma_1 \,?\, \gamma_2, \gamma}\ (OK5)$$

$$\frac{\begin{array}{c} r; n_a \vdash_{ok} \gamma_a :: (r^{-})^{n_b} \quad r \in \mathsf{dom}(\gamma_a) \quad \forall n_c.\, \neg\, (r; n_a - 1 \vdash_{ok} \gamma_a :: (r^{-})^{n_c}) \\ \gamma_3 = \{r^{+}, r^{-} \mid r^{+} \in \gamma_a\} \qquad \mathsf{rsummary}(\gamma_a \setminus r) = \gamma_1; \gamma_2; \gamma_0 \end{array}}{\mathsf{rsummary}(\gamma_a) = \gamma_3 :: \gamma_1; (r^{+})^{n_b} :: \gamma_2; (r^{-})^{n_a} :: \gamma_0}\ (PX0)$$

$$\frac{}{\mathsf{rsummary}(\emptyset) = \emptyset; \emptyset; \emptyset}\ (PX1)$$

$$\mathsf{summary}(\gamma_a) = \ \gamma_1 :: \gamma_2 :: \gamma_3 \qquad \text{if } \mathsf{rsummary}(\gamma_a) = \gamma_1; \gamma_2; \gamma_3$$

Figure 7.7: Summarization relation.

can have a positive reference count for $\imath$. It also establishes that the future lockset of an acquired lock at any program point (ϵ_3 — modulo the locations that have been created *after* the lock was initially acquired) is *always* a subset of the future lockset computed when the lock was initially acquired (ϵ_2).

Definition 7.2 (Thread Typing) The following rules define *well typed* threads.

$$\frac{}{S; M \vdash \emptyset} \qquad \frac{\begin{array}{c} M; \emptyset; \emptyset \vdash e : \langle\rangle \,\&\, (\emptyset; \gamma) \quad S; M \vdash T \\ n \notin \mathsf{dom}(T) \qquad n; \gamma \vdash S \end{array}}{S; M \vdash T, n : e}$$

A collection of threads T is well typed w.r.t. a lock store S and a set of lock identifiers M, if for each thread $n : e$, expression e is well-typed with an empty input effect and some output effect γ and the lock store is consistent w.r.t. n and γ.

Definition 7.3 (Configuration Typing) A configuration $S; T$ is *well typed* w.r.t. M (we denote this by $M \vdash S; T$) when $S; M \vdash T$ and $M = \mathit{dom}(S)$.

$$\frac{S; M \vdash T \quad M = \mathsf{dom}(S)}{M \vdash S; T}$$

Definition 7.4 (Deadlocked State) A configuration has reached a *deadlocked state* when there exist a set of threads $n_0, \ldots, n_k$, for $k > 0$, and a set of locks $\ell_0, \ldots, \ell_k$, such that each thread $n_\imath$ has acquired lock $\ell_{(\imath+1) \bmod (k+1)}$ and is waiting for lock $\ell_\imath$.

$$\begin{array}{lcl} \mathsf{locks}(S, \imath, n) & = & \begin{cases} n_2 & \text{if } S(\imath) = (n; n_2; \epsilon_1; \epsilon_2) \\ 0 & \text{if } S(\imath) = (n_1; n_2; \epsilon_1; \epsilon_2) \wedge n_1 \neq n \end{cases} \\ \mathsf{deadlocked}(T) & \equiv & T \supseteq T_1, n_0 : E[\mathtt{lock}_{\gamma_0}\ \mathtt{lk}_{\imath_0}], \ldots, n_k : E_k[\mathtt{lock}_{\gamma_k}\ \mathtt{lk}_{\imath_k}] \ \wedge\ k > 0 \wedge \\ & & \forall m_1 \in [0, k].\, m_2 = (m_1 + 1)\ \mathit{mod}(k+1)\ \wedge\ \mathsf{locks}(S, \imath_{m_2}, m_1) > 0 \end{array}$$

Definition 7.5 (Not Stuck) A configuration $S;T$ is *not stuck* when each thread in T can take one of the evaluation steps in Figure 7.3 or is waiting for a lock held by some other thread.

Given these definitions, we can now present the main results of this chapter. The *progress*, *deadlock freedom* and *preservation* lemmata are formalized at the *program* level, i.e., for all concurrently executed threads. Let expression e be the initial program. The initial program configuration $S_0;T_0$ is defined by taking $S_0 = \emptyset$, and $T_0 = \{\emptyset; e\}$.

Lemma 7.3 (Deadlock Freedom) If the initial configuration takes n steps, where each step is well-typed for some M, then the resulting configuration has not reached a deadlocked state.

Proof. Let us assume that z threads have reached a deadlocked state and let $m \in [0, z-1]$, $k = (m+1)$ *mod* z and $o = (k+1)$ *mod* z. According to definition of *deadlocked state*, thread m acquires lock $\imath_k$ and waits for lock $\imath_m$, whereas thread k acquires lock $\imath_o$ and waits for lock $\imath_k$. Assume that m is the first of the z threads that acquires a lock, so it acquires lock $\imath_k$ before thread k acquires lock $\imath_o$.

Let us assume that $S_y;T_y$ is the configuration once $\imath_o$ is acquired by thread k for the first time, ϵ_{1y} is the corresponding lockset of $\imath_o$ ($\epsilon_{1y} = \mathit{run}(\mathit{stack}(E[\mathtt{pop}_{\gamma_y}\ \Box]), 1, \imath_o)$) and ϵ_{2y} is the set of all heap locations ($\epsilon_{2y} = \mathit{dom}(S_y)$) at the time $\imath_o$ is acquired. Then, $\imath_k$ does not belong to ϵ_{1y}, otherwise thread k would have been blocked at the lock request of $\imath_o$ as $\imath_k$ is already owned by thread m.

Let us assume that when thread k attempts to acquire $\imath_k$, the configuration is of the form $S_x;T_x$. According to the assumption of this lemma that all configurations are well typed so $S_x;T_x$ is well-typed as well. By inversion of the typing derivation of $S_x;T_x$, we obtain the typing derivation of thread $n_k : E_k[\mathtt{lock}_{\gamma'_k}\ \mathtt{lk}_{\imath_k}]$: $\mathtt{lock}_{\gamma'_k}\ \mathtt{lk}_{\imath_k}$ is well-typed with input-output effect $(\gamma'_k;\gamma''_k)$, where $\gamma''_k = {\imath_k}^+, \gamma'_k$, $E_k[\mathtt{lock}_{\gamma'_k}\ \mathtt{lk}_{\imath_k}]$ is well typed with input-output effect $(\emptyset;\gamma_k)$, where $\gamma_k = {\imath_k}^+, \gamma'''_k$ (for some γ'''_k), and $n_k;\gamma_k \vdash S_x$ holds. The latter derivation implies that $\mathit{run}(\gamma_k, \imath_o, n_2) \cap \epsilon_1 \subseteq \epsilon_2$, where $S_x = S'_x, \imath_o \mapsto n_k; n_2; \epsilon_1; \epsilon_2$ (notice that n_2 is positive, $\epsilon_2 = \epsilon_{1y}$ and $\epsilon_1 = \epsilon_{2y}$ — this is immediate by the operational steps from $S_y;T_y$ to $S_x;T_x$ and rule *E-LK0*).

We have assumed that m is the first thread to lock $\imath_k$ at some step before $S_y;T_y$, thus $\imath_k \in \mathit{dom}(S_y)$ (the store can only grow — this is immediate by observing the operational semantics rules). By the definition of function run and the definition of γ''_k we have that $\imath_k \in \mathit{run}(\gamma_k, \imath_o, n_2) = \mathit{run}(\gamma'''_k, \imath_o, n_2) \cup \{\imath_k\}$. Therefore, $\imath_k \in \mathit{run}(\gamma_k, \imath_o, n_2) \cap \mathit{dom}(S_y) \subseteq \epsilon_{1y}$, which is a contradiction. □

Lemma 7.4 (Progress) dlslemma:thread-progress If $S;T$ is a closed well-typed configuration with $M \vdash S;T$, then $S;T$ is not stuck.

Proof. It suffices to show that for any thread in T, a step can be performed or the thread is blocked. Let n be an arbitrary thread in T such that $T = T_1, n : e$ for some T_1. By inversion of the configuration typing derivation we have that $S; M \vdash T_1, n : e$, and $M = \mathit{dom}(S)$. By inversion of the former derivation we obtain that $n;\gamma \vdash S$ and $M;\emptyset;\emptyset \vdash e : \langle\rangle$ & $(\emptyset;\gamma)$. If e is a value then it can only be the unit value and a step can be performed using rule *E-T*. If e is not value then e is of the form $E[u]$ (this can be shown by induction on the typing derivation of e). The application of the context decomposition lemma (proof by induction on the shape of E) to the typing derivation of $E[u]$ yields that: $M;\emptyset;\emptyset \vdash u : \tau$ & $(\gamma_a;\gamma_b)$ and $M;\emptyset;\emptyset \vdash E' : \tau \xrightarrow{\gamma_a;\gamma_b} \langle\rangle$ & & $(\emptyset;\gamma)$ (the evaluation context typing rules are provided in Figure 7.8). We proceed by a case analysis on u (we only consider the most interesting cases):

Case $(v'\ v)^{\mathsf{seq}}$: the typing derivation of u implies that v' is of the form $\lambda x.\, e'$ or $\mathtt{fix}\ x.\, e'$. In the first case rule *E-A* can be applied, whereas in the second case rule *E-FX* can be applied.

Case $\mathtt{unlock}\ v$: the typing derivation of u implies that v is a lock handle (i.e., $v = \mathtt{lk}_\imath$). It is possible to derive that $\gamma = \imath^-, \gamma'$, for some γ'. By inversion of the store typing premise ($n;\gamma \vdash S$) of the derivation for thread n we have that $\imath; n_2 \vdash_{ok} \imath^-, \gamma'$, where n_2 is the reference count of lock $\imath$. By inversion of the latter derivation (rule *OK2*) n_2 is positive. Combined with the store

typing derivation, this tells us that the thread identifier of $\imath$ is n. Therefore, a single step can be performed via rule *E-UL*.

Case $\texttt{lock}_{\gamma_a}\ v$: the typing derivation of u implies that v is a lock handle (i.e., $v = \texttt{lk}_\imath$). If the reference count (n_2) of lock $\imath$ is positive then the proof is similar to the case of $\texttt{unlock}\ v$ and a step can be performed via rule *E-LK1*. If $n_2 = 0$, it is possible to derive $\gamma = \imath^+, \gamma_a :: \gamma'$ for some γ'. By inversion of the store typing premise ($n; \gamma \vdash S$) of the derivation for thread n we have that $\imath; 0 \vdash_{ok} \imath^+, \gamma_a :: \gamma'$ and that the thread identifier of $\imath$ is n. Therefore $\imath; 0 \vdash_{ok} \imath^+, \gamma_a :: \gamma'$ implies $\epsilon = \mathit{run}(\mathit{stack}(E[\texttt{pop}_{\gamma_a}\ \Box]), \imath, 1)$ is defined (here we are using the fact that the typing derivation implies that $\gamma_a :: \gamma' = \mathit{stack}(E[\texttt{pop}_{\gamma_a}\ \Box])$ and also the fact than when *ok* is defined so is run — this can be trivially shown). Now, if $\epsilon \cup \{\imath\} \subseteq \mathit{available}(S, n)$, then rule *E-LK0* can be applied. Otherwise, the thread is considered to be blocked *but not stuck* (see the third rule of judgement stuck). □

Lemma 7.5 (Preservation) Let $S; T$ be a well-typed configuration with $M \vdash S; T$. If the operational semantics takes a step $S; T \rightsquigarrow S'; T'$, then there exists $M' \supseteq M$ such that the resulting configuration is well-typed with $M' \vdash S'; T'$.

Proof. By induction on the thread evaluation relation (we only consider the most interesting cases):

Case *E-A*: this rule is side-effect free so $S' = S$ and $T' = T$. Therefore, it suffices to show that $E[\texttt{pop}_{\gamma_a}\ e_1[v/x]]$ is well-typed with the same effect as $E[u]$, where u equals $(v'\ v)^{\mathsf{seq}}$ and v' is equal to $\lambda x.\, e_1$. By inversion of the configuration typing we have that $M; \emptyset; \emptyset \vdash E[e] : \langle\rangle \,\&\, (\emptyset;\gamma)$. The application of the context decomposition lemma (proof by induction on the shape of E) to the typing derivation of $E[u]$ yields that: $M; \emptyset; \emptyset \vdash E : \tau_2' \xrightarrow{\gamma_a;\gamma_b} \langle\rangle \,\&\, (\emptyset;\gamma)$ and $M; \emptyset; \emptyset \vdash e : \tau_2' \,\&\, (\gamma_a;\gamma_b)$. By inversion of the latter derivation we have that $M; \emptyset; \emptyset \vdash v : \tau_1' \,\&\, (\gamma_b;\gamma_b)$, and $M; \emptyset; \emptyset \vdash \lambda x.\, e_1 : \tau_1' \xrightarrow{\gamma_c'} \tau_2' \,\&\, (\gamma_b;\gamma_b)$, where $\gamma_b = \gamma_c' :: \gamma_a$. We can apply inversion to the latter derivation to obtain $M; \emptyset; \emptyset, x : \tau_1' \vdash e_1 : \tau_2' \,\&\, (\emptyset;\gamma_c')$. The standard substitution lemma implies that $M; \emptyset; \emptyset \vdash e_1[v/x] : \tau_2' \,\&\, (\emptyset;\gamma_c')$ holds. The application of rule *T-PP* yields $M; \emptyset; \emptyset \vdash \texttt{pop}_{\gamma_a}\ e_1[v/x] : \tau_2' \,\&\, (\gamma_a;\gamma_b)$ holds. Finally, the context composition lemma yields $M; \emptyset; \emptyset \vdash E[\texttt{pop}_{\gamma_a}\ e_1[v/x]] : \langle\rangle \,\&\, (\emptyset;\gamma)$.

Case *E-FX*: as in the previous case, this rule is side-effect free. Redex u is equal to $(\texttt{fix}\ x.\, f\ v)^{\mathsf{seq}}$. By inversion of the configuration typing derivation we obtain $M; \emptyset; \emptyset \vdash E[e] : \langle\rangle \,\&\, (\emptyset;\gamma)$. The context decomposition lemma yields $M; \emptyset; \emptyset \vdash E : \tau_2' \xrightarrow{\gamma_a;\gamma_b} \langle\rangle \,\&\, (\emptyset;\gamma)$ and $M; \emptyset; \emptyset \vdash e : \tau_2' \,\&\, (\gamma_a;\gamma_b)$. By inversion of the latter derivation we have that $M; \emptyset; \emptyset \vdash v : \tau_1' \,\&\, (\gamma_b;\gamma_b)$, and $M; \emptyset; \emptyset \vdash \texttt{fix}\ x.\, f : \tau_1' \xrightarrow{\gamma_c'} \tau_2' \,\&\, (\gamma_b;\gamma_b)$, where $\gamma_b = \gamma_c' :: \gamma_a'$. By inversion of the typing derivation of $\texttt{fix}\ x.\, f$ we obtain that $M; \emptyset; \emptyset, x : \tau_1' \xrightarrow{\gamma_c'} \tau_2' \vdash f : \tau_1' \xrightarrow{\gamma_c''} \tau_2' \,\&\, (\gamma_b;\gamma_b)$ and $\mathit{summary}(\gamma_c'') = \gamma_c'$. The variable substitution lemma yields $M; \emptyset; \emptyset \vdash f[\texttt{fix}\ x.\, f/x] : \tau_1' \xrightarrow{\gamma_c''} \tau_2' \,\&\, (\gamma_b;\gamma_b)$ holds. The effects of E can be strengthened and then weakened (proof by induction on the shape of E) so that $M; \emptyset; \emptyset \vdash E : \tau_2' \xrightarrow{\gamma_a;\gamma_c'' :: \gamma_a} \langle\rangle \,\&\, (\emptyset;\gamma_c'' :: \gamma')$ holds and γ' is such that $\gamma = \gamma_c' :: \gamma'$. Values, such as $f[\texttt{fix}\ x.\, f/x]$ can be assigned any well-formed effect provided that the input effect is identical to the output effect (proof by induction on the expression typing derivation). Therefore, $M; \emptyset; \emptyset \vdash f[\texttt{fix}\ x.\, f/x] : \tau_1' \xrightarrow{\gamma_c''} \tau_2' \,\&\, (\gamma_c'' :: \gamma_a;\gamma_c'' :: \gamma_a)$ and $M; \emptyset; \emptyset \vdash v : \tau_1' \,\&\, (\gamma_c'' :: \gamma_a;\gamma_c'' :: \gamma_a)$ hold. The application of rule *T-SA* yields $M; \emptyset; \emptyset \vdash (f[\texttt{fix}\ x.\, f/x]\ v)^{\mathsf{seq}} : \tau_2' \,\&\, (\gamma_a;\gamma_c'' :: \gamma_a)$. The application context composition lemma gives $M; \emptyset; \emptyset \vdash E[(f[\texttt{fix}\ x.\, f/x]\ v)^{\mathsf{seq}}] : \langle\rangle \,\&\, (\emptyset;\gamma_c'' :: \gamma')$. Given that $n; \gamma \vdash S$ and $\gamma_c' = \mathit{summary}(\gamma_c'')$ hold it is possible to show that $n; \gamma_c'' :: \gamma' \vdash S$. The key idea in this proof is to show that summary preserves unmatched lock acquisition or release operations and that the future lockset of any lock in γ_c' is a superset of the corresponding lockset in γ_c''.

Case *E-LK0*: rule *E-LK0* implies that $T' = T, n : E[()]$, where $()$ replaces u ($u = \texttt{lock}_{\gamma_a}\,\texttt{lk}_\imath$) in context E. It also implies that $\epsilon = \mathit{run}(\mathit{stack}(E[\texttt{pop}_{\gamma_a}\ \square]), \imath, 1)$, $\epsilon \cup \{\imath\} \subseteq \mathit{available}(S, n)$ and $S' = S[\imath \mapsto n; 1; \mathit{dom}(S); \epsilon]$. It suffices to show that $M = \mathit{dom}(S')$, $n; \gamma'_n \vdash S'$, $\forall n' \in \mathit{dom}(T).\, n'; \gamma_{n'} \vdash S'$, where $\gamma_{n'}$ is the output effect of thread n' and $M; \emptyset; \emptyset \vdash E[()] : \langle\rangle \,\&\, (\emptyset; \gamma'_n)$, where γ_n is the output effect of thread n and is defined as $\gamma_n = \imath^+, \gamma'_n$. $M = \mathit{dom}(S')$ is immediate from $M = \mathit{dom}(S)$ and the definition of S'. $n; \gamma'_n \vdash S'$ can be shown by a case analysis on location $\texttt{j}$ of S'. If $\texttt{j} \neq \imath$, then all premises of $n; \gamma_n \vdash S$ also hold for S' and γ'_n. If $\texttt{j} = \imath$, then premise $\imath; 0 \vdash_{ok} \imath^+, \gamma'_n$ implies $\imath; 1 \vdash_{ok} \gamma'_n$. In addition, $\mathit{stack}(E[\texttt{pop}_{\gamma_a}\ \square]) = \gamma'_n$, thus $\epsilon = \mathit{run}(\gamma'_n, \imath, 1)$. The invariant $n'; \gamma_{n'} \vdash S'$ holds for all threads $n' \neq n$ as $S(\texttt{j}) = S'(\texttt{j})$ for all $\texttt{j} \neq \imath$ and $\imath$ is not locked by n' in S'. By inversion of the thread typing derivation we have that: $M; \emptyset; \emptyset \vdash E[u] : \langle\rangle \,\&\, (\emptyset; \gamma)$. The application of the context decomposition lemma (proof by induction on the shape of E) to the typing derivation of $E[u]$ yields that: $M; \emptyset; \emptyset \vdash E : \langle\rangle \xrightarrow{\gamma_a; \imath^+; \gamma_a} \langle\rangle \,\&\, (\emptyset; \gamma_n)$ and $M; \emptyset; \emptyset \vdash u : \langle\rangle \,\&\, (\gamma_a; \imath^+, \gamma_a)$. The effects of E can be strengthened (proof by induction on the shape of E) so that $M; \emptyset; \emptyset \vdash E : \langle\rangle \xrightarrow{\gamma_a; \gamma_a} \langle\rangle \,\&\, (\emptyset; \gamma'_n)$ holds. Rule *T-U* implies $M; \emptyset; \emptyset \vdash () : \langle\rangle \,\&\, (\gamma_a; \gamma_a)$. The application of the context composition lemma on the derivations of E and $()$ yields $M; \emptyset; \emptyset \vdash E[()] : \langle\rangle \,\&\, (\emptyset; \gamma'_n)$. Cases *E-UL* and *E-LK1* can be shown in a similar manner. □

Lemma 7.6 (Multi-step Preservation) Let $S_0; T_0$ be a *closed well-typed configuration* for some M_0 and assume that $S_0; T_0$ evaluates to $S_n; T_n$ in n steps. Then for all $\imath \in [0, n]$ $M_\imath \vdash S_\imath; T_\imath$ holds.

Proof. Proof by induction on the number of steps n using Lemma 7.5.

Theorem 7.1 (Type Safety) If the initial configuration $S_0; T_0$ is closed and well-typed ($\emptyset \vdash S_0; T_0$) and the operational semantics takes any number of steps $S_0; T_0 \rightsquigarrow^n S_n; T_n$, then the resulting configuration $S_n; T_n$ is not stuck and T_n has not reached a deadlocked state.

Proof. The application of Lemma 7.6 to the typing derivation of $S_0; T_0$ implies that for all steps from zero to n there exists an $M_\imath$ such that $M_\imath \vdash S_\imath; T_\imath$. Therefore, Lemma 7.3 implies that T_n has not reached a deadlocked state and Lemma 7.4 implies $S_n; T_n$ is not stuck.

Using an empty typing context for typing the initial configuration $S_0; T_0$ guarantees that all functions in the program are closed and that no explicit lock values ($\texttt{lk}_\imath$) are used in the source of the original program.

7.7 Concluding remarks

Similarly to the previous chapter, we utilize statically computed information regarding lock usage at execution time in order to avoid deadlocks. As opposed to the language presented in the previous chapter, we proposed a language that does not require explicit and complex annotations from the programmer. Additionally, the proposed type and effect system is less complicated and can accept a wider class of programs as it imposes no restrictions with respect to aliasing. More importantly, it is able to automatically infer program effects and therefore it is more suitable for a language like C. We also presented a semantics for the proposed language, a sound type and effect system that guarantees that well-typed programs cannot reach a deadlocked state, and a proof sketch for the type safety theorem and related lemmata.

Sub-effect $\gamma \lhd \gamma' \equiv \exists \gamma''.\gamma' = \gamma'' :: \gamma$

$$\frac{M;\Delta \vdash \Gamma \quad M;\Delta \vdash \gamma_1 \quad M;\Delta \vdash \gamma_2 \quad M;\Delta \vdash \tau}{M;\Delta;\Gamma \vdash \Box : \tau \xrightarrow{\gamma_1;\gamma_2} \tau \,\&\, (\gamma_1;\gamma_2)} \textit{(E0)}$$

$$\frac{\begin{array}{c} M;\Delta;\Gamma \vdash E : \tau_2 \xrightarrow{\gamma_5;\gamma_6} \tau_3 \,\&\, (\gamma_1;\gamma_2) \\ M;\Delta;\Gamma \vdash F : \tau_1 \xrightarrow{\gamma_3;\gamma_4} \tau_2 \,\&\, (\gamma_5;\gamma_6) \end{array}}{M;\Delta;\Gamma \vdash E[F] : \tau_1 \xrightarrow{\gamma_3;\gamma_4} \tau_3 \,\&\, (\gamma_1;\gamma_2)} \textit{(E1)}$$

$$\frac{\begin{array}{c} M;\Delta \vdash \gamma_2 \quad \gamma_1 \lhd \gamma_2 \quad M;\Delta \vdash \tau_2 \\ M;\Delta;\Gamma \vdash e_2 : \tau_1 \,\&\, (\gamma_a :: \gamma;\gamma_1) \end{array}}{M;\Delta;\Gamma \vdash (\Box\ e_2)^{\mathsf{seq}(\gamma)} : (\tau_1 \xrightarrow{\gamma_a} \tau_2) \xrightarrow{\gamma_1;\gamma_2} \tau_2 \,\&\, (\gamma;\gamma_2)} \textit{(F1)}$$

$$\frac{\begin{array}{c} \gamma_2 \lhd \gamma_3 \quad M;\Delta \vdash \tau_1 \xrightarrow{\gamma_a} \langle\rangle \\ M;\Delta;\Gamma \vdash e_2 : \tau_1 \,\&\, (\gamma_1;\gamma_2) \end{array}}{M;\Delta;\Gamma \vdash (\Box\ e_2)^{\mathsf{par}} : (\tau_1 \xrightarrow{\gamma_a} \langle\rangle) \xrightarrow{\gamma_2;\gamma_3} \langle\rangle \,\&\, (\gamma_1;\gamma_3)} \textit{(F2)}$$

$$\frac{\begin{array}{c} \gamma_2 = \gamma_1 :: \gamma_a \quad M;\Delta \vdash \gamma_3 \quad \gamma_2 \lhd \gamma_3 \\ M;\Delta;\Gamma \vdash v_1 : \tau_1 \xrightarrow{\gamma_a} \tau_2 \,\&\, (\gamma_3;\gamma_3) \end{array}}{M;\Delta;\Gamma \vdash (v_1\ \Box)^{\mathsf{seq}(\gamma_1)} : \tau_1 \xrightarrow{\gamma_2;\gamma_3} \tau_2 \,\&\, (\gamma_1;\gamma_3)} \textit{(F3)}$$

$$\frac{\begin{array}{c} M;\Delta \vdash \gamma_2 \quad \gamma_1 \lhd \gamma_2 \\ M;\Delta;\Gamma \vdash v_1 : \tau_1 \xrightarrow{\gamma_a} \langle\rangle \,\&\, (\gamma_2;\gamma_2) \end{array}}{M;\Delta;\Gamma \vdash (v_1\ \Box)^{\mathsf{par}} : \tau_1 \xrightarrow{\gamma_1;\gamma_2} \langle\rangle \,\&\, (\gamma_1;\gamma_2)} \textit{(F4)}$$

$$\frac{M;\Delta \vdash \tau \quad M;\Delta \vdash \gamma' \quad \gamma' = \gamma_2 :: \gamma}{M;\Delta;\Gamma \vdash \mathsf{pop}_\gamma\ \Box : \tau \xrightarrow{\emptyset;\gamma_2} \tau \,\&\, (\gamma;\gamma')} \textit{(F5)}$$

$$\frac{\begin{array}{c} M;\Delta \vdash \Gamma \quad M;\Delta \vdash \gamma' \\ M;\Delta \vdash \mathtt{Lk}(r) \quad \gamma_1 = r^-, \gamma \quad \gamma_1 \lhd \gamma' \end{array}}{M;\Delta;\Gamma \vdash \mathtt{unlock}\,\Box : \mathtt{Lk}(r) \xrightarrow{\gamma_1;\gamma'} \langle\rangle \,\&\, (\gamma;\gamma')} \textit{(F6)}$$

$$\frac{\begin{array}{c} M;\Delta \vdash \Gamma \quad M;\Delta \vdash \gamma' \\ M;\Delta \vdash \mathtt{Lk}(r) \quad \gamma_1 = r^+, \gamma \quad \gamma_1 \lhd \gamma' \end{array}}{M;\Delta;\Gamma \vdash \mathtt{lock}_\gamma\ \Box : \mathtt{Lk}(r) \xrightarrow{\gamma_1;\gamma'} \langle\rangle \,\&\, (\gamma;\gamma')} \textit{(F7)}$$

$$\frac{\begin{array}{c} \gamma_3 = \gamma_1\,?\,\gamma_2, \gamma \quad M;\Delta \vdash \gamma' \quad \gamma_3 \lhd \gamma' \\ M;\Delta;\Gamma \vdash e_2 : \tau \,\&\, (\gamma;\gamma_1 :: \gamma) \quad M;\Delta;\Gamma \vdash e_3 : \tau \,\&\, (\gamma;\gamma_2 :: \gamma) \end{array}}{M;\Delta;\Gamma \vdash \mathtt{if}\ \Box\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3 : \mathtt{Bool} \xrightarrow{\gamma_3;\gamma'} \tau \,\&\, (\gamma;\gamma')} \textit{(F8)}$$

$$\frac{M;\Delta \vdash \gamma' \quad M;\Delta \vdash \forall\rho.\,\tau \quad r \in M \cup \Delta \quad \gamma \lhd \gamma'}{M;\Delta;\Gamma \vdash (\Box)\,[r] : \forall\rho.\,\tau \xrightarrow{\gamma;\gamma'} \tau[r/\rho] \,\&\, (\gamma;\gamma')} \textit{(F9)}$$

Figure 7.8: Evaluation context typing rules.

Chapter 8

Deadlock avoidance tool

8.1 Overview

Deadlocks can have devastating effects in systems code. In the previous chapters we presented two type and effect systems that provably avoid deadlocks and in this chapter we present a tool that uses a sound static analysis based on the type system of Chapter 7 to instrument multithreaded C programs and then links these programs with a run-time system that avoids possible deadlocks. In contrast to most other purely static tools for deadlock freedom, our tool does not insist that programs adhere to a strict lock acquisition order or use lock primitives in a block-structured way, thus it is appropriate for systems code and OS applications. We also report some very promising benchmark results which show that all possible deadlocks can automatically be avoided with only a small run-time overhead. More importantly, this is done without having to modify the original source program by altering the order of resource acquisition operations or by adding annotations.

8.2 Deadlock avoidance analysis

Our analysis is performed in two phases. The first phase takes place at compile-time and performs a field-sensitive and context-sensitive pointer analysis followed by continuation effect (as defined in Chapter 5) inference, and source code instrumentation with effects; finally, the instrumented program is linked with the runtime system. The second phase is purely dynamic and takes place when the original program requests a lock. The future lockset (as defined in Chapter 5) of the requested lock is computed by utilizing the inserted effects and the lock is only granted when both the lock and its future lockset are available. In this chapter, we extend the analysis for deadlock avoidance for multithreaded C programs presented in the previous chapter, so that it can handle locks residing in data structures allocated in the heap or the stack and introduce new optimizations that reduce the size of effects, thereby reducing the size of future locksets and the blocking time of lock operations.

8.2.1 Static analysis

The analysis takes as input the program's abstract syntax tree and constructs a call-graph, which is visited bottom-up. There are four main stages involved in the analysis of function declarations (i.e., nodes of the call graph) described in the following paragraphs.

Pointer analysis. First, a standard, off-the-shelf field-sensitive intra-procedural pointer analysis based on symbolic execution [Voun07] is employed so as to formulate an abstract heap and stack state at each program point. We have customized the analysis so that it treats heap allocation in a context-sensitive manner. At the end of the first stage, we obtain a mapping for each expression to a set of abstract locations. Each abstract location r can be a formal parameter, a global variable or a heap-allocated location.

Effect inference. The effect for each function is computed by running a standard forward data flow algorithm on the function's control flow graph. Each node in the control flow graph is associated with an input, a current and an output effect. The input effect of a node is formulated by *joining* effects flowing from all its *front* edges. For instance, a node associated with effects $\gamma_1 \ldots \gamma_k$, will be assigned an input effect $\gamma_1 \,?\, \ldots \,?\, \gamma_k$, which denotes a choice between the alternative effects $\gamma_1 \ldots \gamma_k$. A distinction is made when encountering back edges; we defer the discussion regarding the treatment of back edges until the next paragraph.

The current effect of a node can be r^+ or r^-, when a lock or unlock operation is found, respectively. It can be `malloc` ρ, when a new reference is allocated dynamically and bound to the variable ρ. It can also be `call` $r\,(\,r_1, \ldots, r_n\,) : r'$ when a function is called, where r is a reference to the function, $r_1 \ldots r_n$ are references to the function's arguments, and r' is a reference that corresponds to the function's result. In this case, if our standard points-to analysis cannot determine a unique target for r (e.g., if a function is called indirectly through a pointer), then the original effect is replaced by a joined effect consisting of several alternative branches of the form `call` $f_i\,(\,r_1, \ldots, r_n\,) : r'_i$, for each alternative target function f_i.

The output effect is computed by appending the current effect to the input effect and is propagated to a node's successors until a fixed point is reached. A function's effect is computed by joining the output effects of nodes having no successors.

Loops. Effects flowing from back edges must be equivalent (with respect to the lock counts) to the input effect of the same node. This restriction allows us to soundly encode loop effects: a loop may have any number of lock or unlock operations provided that upon exit of a loop the counts of each lock match the counts before the loop was executed. Assuming that the effect of the loop body is γ, then we take $(\gamma \,?\, \emptyset),\ (\gamma \,?\, \emptyset)$ as the effect of the entire loop. The empty effect on both branches compensates for the case where a loop is not executed. The duplication of $(\gamma \,?\, \emptyset)$ is required so that lock operations can match between successive loop iterations. In cases where a loop effect does not contain unmatched lock operations, this duplication may be optimized away.

Effect optimizations. While computing effects, several optimizations are performed so as to compact/elide effects and in general minimize the repetitions of the identical effect segments in a function's effect. One of the optimizations for compacting an effect computes the common prefix and suffix of the effects included in a join operator, to decrease the size of branches. Another kind of optimization is to flatten effects that consist of nested join operators. For example, an effect of the form $(\gamma_1 \,?\, \gamma_2) \,?\, (\gamma_3 \,?\, \gamma_4)$ can be reduced to $\gamma_1 \,?\, \gamma_2 \,?\, \gamma_3 \,?\, \gamma_4$. In addition, multiple occurrences of the *empty effect* in alternative effects $\gamma_1 \ldots \gamma_n$ are substituted with a single empty effect. These two optimizations are run alternately until a fixed point is reached in the size of the effect.

Eventually, call effects are substituted by a *summary* of the effect corresponding to the function being called. In summarized effects multiple lock/unlock pairs for the same reference are redundant. The intuition behind this optimization is that the future lockset that will be computed dynamically will be the same, regardless of the number of times that a lock operation occurs.

Another important optimization attempts to minimize the size of the runtime effect stack, so that the future lockset calculation algorithm visits as few stack frames as possible. One way to achieve this is by disabling code instrumentation for functions that do not directly perform any lock operations and do not contain calls to functions that will visit their effect frame at run time. Finally, we invoke the data flow algorithm, which is CPU-intensive, only for functions that are known to contain lock operations (by performing an in-advance linear search), to avoid additional overheads.

8.2.2 Code generation

Our main goal was to minimize the overhead induced by "effect accounting". A naïve implementation of the informal description provided in earlier sections would simply allocate and initialize effect

frames for each function call or lock operation, which would be unacceptable in terms of performance. The code generation phase statically creates a *single* block of initialization code for the effect of each function and inserts effect index update instructions (i.e., a single assignment) before each call and lock operation. Therefore, the overhead imposed for such operations is minimal. Each function is also instrumented with instructions for pushing and popping effects from the run-time stack at function entry and exit points respectively. This imposes a constant overhead to function calls.

Finally, *mappings* for stack and heap pointers are generated at run-time as such locations cannot be known statically. A mapping binds an abstract location to a run-time address. An inverse mapping is also maintained for abstract heap locations. When a deallocation operation is performed such as `free`, the inverse mapping is searched using the physical address to be deallocated and the binding between the abstract heap location and the physical address is removed from the heap mapping. In this way, our analysis is able to deal with locks that are dynamically deallocated and thereby avoid invalid accesses to deallocated locks.

8.2.3 Current limitations

Non C code. Our analysis can strictly handle the C language. Library code cannot be analyzed as it is not C code. We have assumed that by default library functions have an empty effect. However, it is possible to provide user-defined effect annotations for library functions. The analysis cannot deal with non-local jumps (including signals) and inline assembly.

Pointer analysis. The off-the-shelf pointer analysis module fails when encountering programs with pointer arithmetic involving locks (including arrays) and recursive data structures that contain or point to locks. Even though our analysis extends the standard pointer analysis with context-sensitive tracking of fresh heap locations, it fails to track heap allocation (for data structures containing or pointing to locks) at recursive functions and loops. This limitation is dual to the aforementioned limitation regarding unbounded data structures. In addition, expressions passed in lock functions must be assigned a *unique* abstract location. Finally, we require that lock pointers are mutated only before they are shared between threads, and that locks referenced with at least two levels of indirection (e.g., via double pointers) are not aliased at function calls.

Conditional execution. The analysis also currently rejects programs in which lock and their matching unlock operations are conditionally executed in distinct conditional statements having equivalent guards. For instance, the following program is rejected:

```
if (condition) lock(z);
if (condition) unlock(z);
```

8.2.4 Runtime system

The runtime system overrides the standard implementation of locking functions, e.g., the pthreads functions `pthread_mutex_lock` and `pthread_cond_wait`.[1] If a lock is already held by the requesting thread then the lock's count is simply incremented. (This occurs only when re-entrant locks are used; however, re-entrant locks are needed in languages that support unrestricted lock aliasing.) Otherwise, the runtime system performs two steps: it computes the future lockset of the requested lock and verifies that all locks in the future lockset are available when the lock is acquired.

The future lockset calculation algorithm uses the effect index inserted by the instrumentation phase to calculate the future lockset of the requested lock. When the matching unlock operation is not found

[1] Currently our tool has built-in support only for overriding the functions of the pthreads library, but it can easily be extended to support other locks. Its implementation and the set of benchmarks we used are available from `http://www.softlab.ntua.gr/~pgerakios/deadlocks/`.

benchmark	run in	user	system	elapsed	ratio
curlftpfs	C	0.002	0.758	33.450	0.982
	C+da	0.000	0.680	32.862	
flam3	C	63.660	3.910	49.050	1.003
	C+da	67.860	3.640	49.200	
migrate-n	C	5545.311	4631.341	4138.070	1.118
	C+da	5334.921	5020.346	4625.670	
ngorca	C	124.846	0.126	8.270	0.996
	C+da	124.467	0.126	8.240	
sshfs-fuse	C	0.000	0.890	20.880	1.000
	C+da	0.000	0.950	20.880	
tgrep	C	13.238	11.639	5.190	1.191
	C+da	14.801	11.655	6.180	

Table 8.1: Performance of C vs. C+da (C plus deadlock avoidance).

in the function's effect, the algorithm visits the effects on the runtime stack. Locks held by the requesting thread are excluded from the lockset. Effect traversal is performed efficiently as the majority of effects are represented by arrays. Each atomic effect is represented by two machine words.

The next step is to acquire the lock provided that all locks in its future lockset are available. We have implemented three different strategies for dealing with unavailable locks. The first one employs *futexes* [Fran02], which in general allow a thread to wait in the kernel for an event. The remaining two strategies employ *busy waiting* or *yield* control to other threads. As the performance of futexes was clearly superior in almost all benchmarks, we quickly focused on this strategy and used it exclusively for all the results that we present in the next section.

Our algorithm initially checks if all locks in the future lockset are available. If some lock is unavailable, we perform a wait operation on it (using one of the above strategies) and retry. If all locks in the future lockset are available, we *tentatively acquire* the requested lock. Then, we check again the future lockset. If any lock in the future lockset is unavailable, we *release* the acquired lock, we perform a wait operation on the unavailable lock and repeat all the steps from the beginning. Otherwise, the lock acquisition operation is considered successful. This approach allows our locking algorithm to be more permissive compared to versioning schemes that check future locksets atomically and thereby yields a higher degree of concurrency for instrumented programs.

8.3 Performance evaluation

We describe some experimental results, aiming to demonstrate that our approach can achieve deadlock freedom with low run-time overhead. The experiments were performed on a machine with four Intel Xeon E7340 CPUs (2.40 GHz), having a total of 16 cores and 16 GB of RAM, running Linux 2.6.26-2-amd64 and GCC 4.3.2.

We used a total of seven benchmark programs. The first one is written by us (this is a program from the literature known to exhibit deadlocks), whereas the remaining six are real applications, whose source code is publicly available. Their performance results are shown in Table 8.1; the order is alphabetical.

dining philosophers: a program implementing the obvious and deadlock-prone attempt to solve the classic multi-process synchronization problem. Each philosopher first picks up the stick on his left, then the stick on his right. The original program deadlocks with a probability that decreases as the number of philosophers increases (for five philosophers, the probability for deadlock was roughly 70%) but increases again when the number of philosophers exceeds the number of available cores. For the performance comparison that we discuss below, we only used the deadlock-free runs of the original program.

n	original	instrumented	improvement
5	126,536	126,961	0.34%
10	224,536	230,981	2.87%
15	284,150	298,563	5.07%
31	536,889	587,051	9.34%
63	1,080,322	1,193,509	10.48%
127	1,603,880	2,219,022	38.35%
255	1,480,183	4,220,603	185.14%

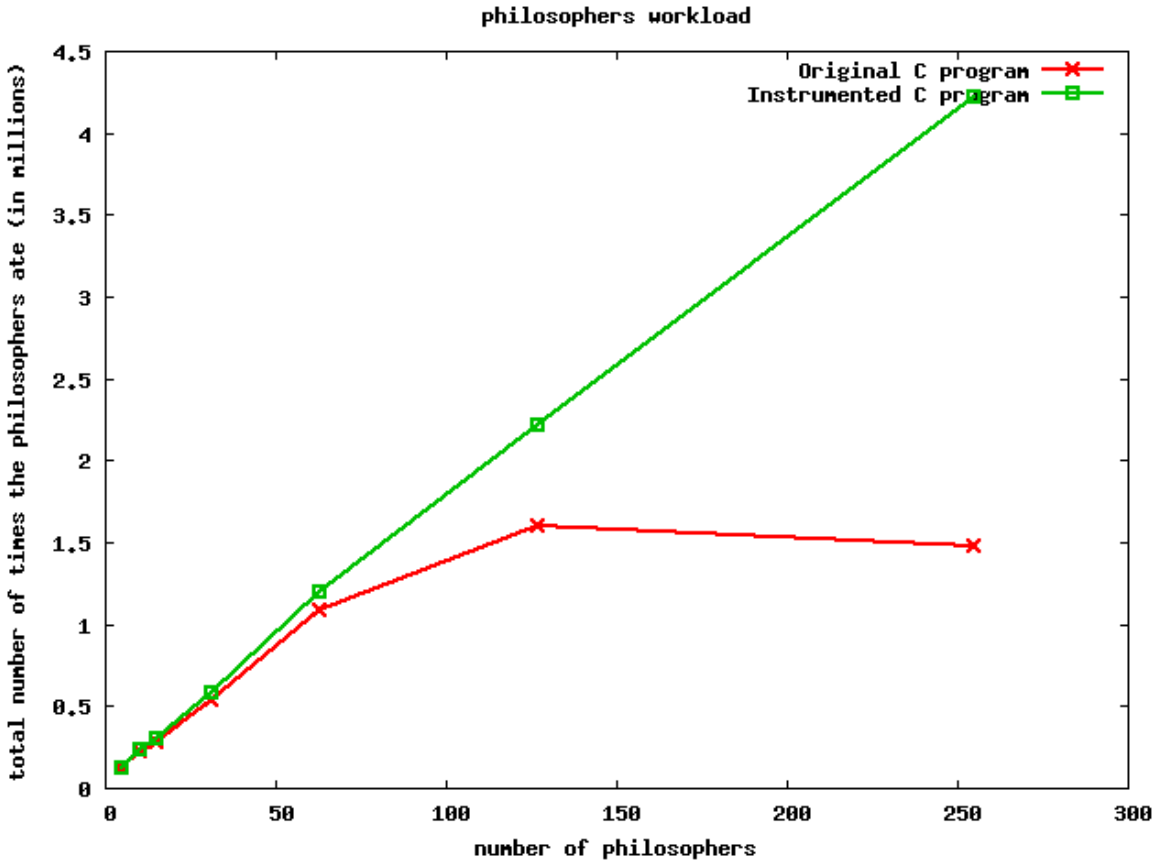


Figure 8.1: Performance comparison for the *dining philosophers*. We measure the total number of times that the n philosophers ate.

The performance of the original and the instrumented versions are shown in Figure 8.1. For a given elapsed time (2 secs) we measured the total number of times that the philosophers ate (using a per-thread random number generator that was identical in both versions). It is interesting to see that in the instrumented program, the number grows linearly with the number of philosophers, i.e., each philosopher eats for a (more or less) constant number of times during the 2 secs (this number is determined by the ratio of eating time versus sleeping time, which was chosen to be 0.1 in both programs). On the other hand, in the original program, the linear growth seems to last only as long as the number of philosophers is small and we do not run out of cores.

curlftpfs [CURL] and sshfs-fuse [SSHF]: are file system clients that access hosts via the FTP and SSH network protocol respectively. Both applications create threads on demand so as to serve concurrent read and write requests to the file systems, using two and three distinct locks respectively to synchronize data structures, logging and access to non thread-safe functions. In our experiments, we mount a remote directory over the corresponding file system and start a fixed

number of concurrent threads, each of which is trying to download a number of large files. The total volume of data that is copied over the file systems is linear w.r.t. to the number of threads. In both cases, the instrumented program has almost identical performance to the original program. Both programs have approximately zero user time as they mainly invoke the kernel space API of FUSE [FUSE].

flam3: a multithreaded program which creates "cosmic recursive fractal flames", i.e., (animations consisting of) algorithmically generated images based on fractals [flam]. A single lock is used to synchronize access to a shared bucket accumulator that merges computations of distinct threads. We measured the time required to generate a long sequence of fractal images. The results again were almost identical for the original and the instrumented version of flam3.

migrate-n: estimates population parameters, effective population sizes and migration rates of n populations using genetic data [MIGR]. The program maintains a work list of Markov chains and uses a thread pool to execute tasks until the work list becomes empty. Two locks are employed for implementing the thread pool and for accessing shared variables. It is worth mentioning that locks are dynamically allocated and several billion lock operations were executed during the program run. The instrumented program ran 11% slower than the original.

ngorca: a multithreaded password recovery tool using exhaustive key search for DES-encrypted passwords in Oracle databases [NGOR]. The program achieves speedup by splitting the search space of each encrypted password across threads, using multiple locks for implementing logging, counters and condition variables. The results again were almost identical for the original and the instrumented version of ngorca.

tgrep: a multithreaded version of the utility program grep which is part of the SUNWdev suite of Solaris 10 [tgre]. The program achieves speedup by splitting the search space across threads, using multiple locks for implementing thread-safe queues, logging and counters. In our experiment, we looked for an occurrence of a six-letter word in a directory tree containing 100,000 files. The instrumented program is 19% slower than the original program. This is due to the fact that tgrep is not only lock-intensive (about 1.5 million lock operations were executed in our test run), but also it is by far the benchmark with the longest effects that we could find. The maximum effect size for a function is 54 and the average effect size is 19.5, which are both about five times higher than the second next benchmark (ngorca). Furthermore, the program employs seven distinct global locks; the dynamically calculated future lockset had a maximum size of five elements and an average size of 1.3, which again were about five times higher than the second next benchmark.

8.4 Concluding remarks

Deadlocks are an important problem especially for systems code written in languages that employ non block-structured locking. In this chapter, we presented implementation aspects of a novel tool that dynamically avoids deadlock states for multithreaded C programs. The key idea is to utilize statically computed information regarding lock usage at runtime in order to avoid deadlocks. We described the main aspects of our static analysis and its obvious limitations: it is necessarily imprecise and cannot support unbounded data structures containing locks. However, we showed that our approach is applicable to several multithreaded C programs containing systems code and our evaluation results reveal that it imposes only a modest runtime overhead, induced by the future lockset computation and by blocking threads more often (i.e., when the requested lock is available but something in its future lockset is not). Nevertheless, we think that its runtime overhead is reasonable for guaranteed deadlock avoidance.

Chapter 9

Related work

The work presented in this dissertation is related to multiple research areas such as safe systems programming languages, safe region-based memory management, data race freedom and deadlock avoidance techniques. In the following sections, we discuss related works in each of these areas.

9.1 Safe systems programming languages

Systems programming languages (SPLs), such as C, can be differentiated from high-level languages with respect to their high run-time performance, the explicit memory management and the low-level memory representation. The lack of strong safety guarantees regarding the behavior of SPL programs is responsible for a large number of vulnerabilities and serious security breaches (e.g., system crashes and/or intrusions). Austin *et al.* have identified two kinds of memory access errors in sequential programs [Aust94]: spatial access errors (e.g., dereferencing a pointer or a subscripted array reference outside the object's bounds) and temporal access errors (e.g., dereferencing a memory location outside the lifetime of the referent). Weakly-typed SPLs, which enable arbitrary type casts and unsafe use of memory, accept programs which generate memory access errors. Multithreaded systems programs, which produce random execution interleavings, may introduce additional memory access errors as well as concurrency hazards such as deadlocks and data races.

Research has focused on providing strong static guarantees for safe SPLs. Two promising approaches towards this goal are the use of type systems and the use of explicit logics, e.g., separation logic [Reyn02]. Advanced type systems have been employed so as to provide safety for SPLs at the C-level of abstraction. For instance, Cyclone adopts static region-based memory management to guarantee the absence of temporal access errors [Jim02, Gros02]. CQual uses type qualifiers to enrich the C language type system in a user-extensible way to achieve similar goals [Fost02]. Vault supports linear types to track object usages at compile time [DeLi01]. Furthermore, Vault's linear type system also allows state to be attached to object types. Such state can statically enforce resource management protocols on the tracked objects. CCured performs a whole-program analysis to infer pointer kinds and then optimizes away run-time checks when it can statically guarantee that a pointer use will not cause memory access errors [Necu05]. Deputy, the successor of CCured, uses a dependent type system to enable modular type checking and maintains explicit memory object representations (i.e. avoids the inclusion of meta-data information along with pointers) by taking advantage of the dependencies generated by the type system [Cond07].

However, state-of-the-art safe SPLs achieve absence of memory access errors by sacrificing explicit memory representations, thereby preventing interoperability with legacy code and deteriorating run-time performance. The type systems of safe SPLs are often unable to encode common C idioms and therefore make automated porting of legacy code hard. None of the above safe SPLs have full support for safe concurrency. For instance, the type systems of both Cyclone [Jim02] and Vault [DeLi01] are unsound in concurrent program settings. There are numerous languages at the C level of abstraction with explicit concurrency features [Frig98, Gay03, Ande08a] but to the best of our knowledge none of them provides both memory safety and race freedom guarantees.

In contrast with type systems, separation logic permits finer-grained program specifications at the

cost of undecidability. For instance, Concurrent C minor [Hobo08] is a concurrent version of C with threads, shared memory and first-class locks, which uses a variant of separation logic to reason about programs.

9.2 Region-based memory management

The first statically checked stack-based region system was developed by Tofte and Talpin [Toft94]. Since then, several memory-safe systems enabling early region deallocation were proposed [Heng01, Walk01, Flue06, DeLi01]. These works use linear types to guarantee memory safety. The proposed memory safety guarantees only hold for sequential languages, place severe restrictions on memory aliasing and ignore issues such as minimizing annotations or providing control to the user. In contrast with these works, we achieve a higher degree of aliasing and extend memory safety guarantees to multi-threaded programs such that shared memory can be concurrently accessed or released by any thread.

RC [Gay01] and Cyclone [Gros02] were the first imperative languages to allow safe region-based management with explicit constructs. Both allowed early region deallocation and RC also introduced the notion of multi-level region hierarchies. RC employs reference counting to dynamically detect that no external references to a region exist, when that region is deallocated. RC programs can make use of more memory-management idioms as all checks are dynamic. On the downside, this imposes a run-time overhead as such programs may throw region-related exceptions as opposed to our approach which is purely static. For instance, RC's region-creation construct requires a parent region. The invariant that the parent region must live longer than its children is enforced dynamically via the means of run-time checks. RC does not guarantee memory safety or race freedom for multithreaded programs.

The Real-Time Specification for Java [Boll02] extends Java with regions in the form of libraries, using "ScopedMemory" objects. To ensure soundness, dynamic checks are performed when accessing regions or references to regions. Statically checked region systems have also been proposed for real-time Java to rule out dynamic checks imposed by its specification. Boyapati *et al.* have introduced hierarchical regions in ownership types [Boya03]. In contrast with our work, Boyapati *et al.* do not enable early release of regions or locks and provide no support for data migration or lock transfers between threads. In addition, variables must be declared either as "thread-local" or "shared" but cannot alternate between the two states. Additionally, their type system only allows sub-regions for *shared* regions, whereas our approach does not have this limitation. In previous work, Boyapati *et al.* also proposed an ownership-based type system that prevents deadlocks and data races [Boya02]. In contrast to that system, we support locking of arbitrary nodes in the region hierarchy.

Zhao *et al.* [Zhao04] proposed Scoped Types, where lexically-scoped regions are wrapped by special kind of objects called *scopes*. Static region hierarchies (depth-wise) can be used by nesting scoped objects. The main advantage of their approach is that programs require fewer annotations compared to programs with explicit region constructs, scopes can be shared at any program point and region handles are implicit. In the same track, Zhao *et al.* proposed implicit ownership annotations for regions [Zhao08]. Thus, classes that have no explicit owner can be allocated in any static region without requiring explicit owner annotations. This is a form of *existential ownership*. In contrast with Zhao's *et al.* proposals, the proposed type system of Chapter 3 allows regions to be released before the end of their lexical scope and provides additional data-race freedom guarantees for programs that concurrently access shared memory by employing reader/writer locks.

9.3 Safe concurrency

9.3.1 Data race freedom

Several tools and techniques have been developed for detecting data races using static analysis, dynamic analysis or a combination of the two. The main advantage of dynamic data race analyses [Sava97, Agar05, OCal03, Choi02] is that they do not require manual intervention from the user. On the other hand, dynamic analyses incur runtime overheads as they instrument the original program with additional code, and can only identify *some* but not *all* possible data races.

Several type systems have been developed for checking race freedom statically [Flan00, Flan99b, Boya01, Boya02, Flan02] in Java programs. Such systems associate lock handles with singleton types or unique abstract locations and effect systems in order to track their uses throughout the program. Most of these type systems require the insertion of explicit type annotations in programs. In the same spirit, a type system for safe multi-threading in Cyclone [Gros03] was proposed by Grossman. His type system does not enable early release of regions and provides no support for data migration or lock transfers between threads. In addition, variables must be declared either as "thread-local" or "shared" but cannot alternate between the two states. In the context of Java, Cunningham *et al.* proposed a universe type system to guarantee race freedom in a calculus of objects [Cunn07]. Similarly to our system, object hierarchies can be atomically locked at any level. Unlike our system, the work of Cunningham *et al.* does not support early lock releases and lock ownership transfers between threads.

More recently, tools that automatically infer type annotations for data race freedom have been developed for Java [Flan01] and C [Prat06, Voun07]. The latter two analyses use context and flow sensitive propagation of locksets and verify that all accessed locations are consistently protected by the same lock through the program. Like our analyses, they handle unstructured locking, but do not validate/match lock and unlock operations. Consequently, data races may occur as a result of undefined behavior of pthreads library. In addition, static analysis tools for C can only recognize certain locking primitives, thus data races may occur when unknown primitives are encountered. In contrast, our approach integrates locking primitives in the language so it is impossible to allow a data race to occur.

Terauchi [Tera08] proposed a technique that reduces the problem of race detection to linear programming. Other static methods for finding data races are proposed in the literature such as abstract interpretation [Gots07] or model checking [Henz04, Qade04]. Other tools such as RacerX [Engl03] are unsound for common C programming idioms such as the use of wrapper functions that perform lock operations.

Data races may result from unintentional data sharing. Recent work on static analysis for C programs [Ande08b, Ande09] infers data sharability and therefore it can report programs that share data unintentionally. More recently, Bocchino *et al.* proposed a type and effect system for DPJ (Deterministic Parallel Java [Bocc09]) that partitions the heap into distinct hierarchical segments and uses those segments to disambiguate accesses to distinct objects. DPJ's type system ensures non-interference by enforcing the invariant that concurrent accesses are read-only or they must refer to disjoint locations for write operations. The segment disjointness invariant places significant constraints on segment aliasing, which is not permitted at the level of types. In contrast, in our system region aliasing is possible at all times. In order to allow race-free writes and reads, the work on DJP was recently extended with non deterministic constructs such as `atomic`, which provides strong isolation guarantees [Bocc11]. The "read-only" and "read-write" region capabilities of our system can encode the non-interference constraints as well as the ability to mutate shared data without introducing data races. (However, our region locking operations have blocking semantics.) Moreover, low-level languages such as the one we are proposing, often constitute the target language of high-level languages like Java and DPJ. In this respect, our type system is not bound to any specific programming paradigm, but instead it is applicable to any language at its implementation level.

Matsakis and Gross recently proposed another variant of Java with static race freedom guarantees using the notion of *intervals* [Mats10]. Intervals are first-class objects representing time spans in which a certain piece of code executes. Intervals can be partially ordered and/or hierarchically nested.

Similarly to our regions, hierarchically nested intervals inherit access rights to data from their ancestor intervals. Concurrent read operations that happen after write operations to shared data as well as concurrent write operations to shared data protected by a lock are permitted. However, to guarantee type safety, the type system requires explicit lock specifications as well as happens-before annotations. Locks are also first-class objects but are never acquired explicitly. Instead, the run-time system uses lock specifications of methods to implicitly acquire locks required by an interval.

9.3.2 Deadlock freedom

The majority of literature for language-based approaches to deadlock freedom falls under deadlock prevention and deadlock detection and recovery. In the deadlock prevention category, one finds type and effect systems [Flan99b, Boya02, Koba06, Suen08, Vasc10] that guarantee deadlock freedom by statically enforcing a global lock acquisition ordering, which must be respected by all threads. In this setting, starting with the influential work of Flanagan and Abadi [Flan99b], lock handles are associated with type-level lock names via the use of singleton types. Thus, handle lk_i is of type $\texttt{lk}(i)$. The same applies to lock handle variables. The effect system tracks the order of lock operations on handles or variables and determines whether all threads acquire locks in the same order. This can be too restrictive for some programs, where the analysis may be imprecise. It is not hard to come up with an example that shows this point.:

$$(\texttt{lock}\ x\ \texttt{in}\ \ldots\ \texttt{lock}\ y\ \texttt{in}\ \ldots) \ ||\ (\texttt{lock}\ y\ \texttt{in}\ \ldots\ \texttt{lock}\ x\ \texttt{in}\ \ldots)$$

In a few words, there are two parallel threads which acquire two distinct locks, x and y, in reverse order. When trying to find a partial order $\leq$ on locks for this program, the type system or static analysis tool will deduce that $x \leq y$ must be true, because of the first thread, and that $y \leq x$ must be true, because of the second. Thus, the program will be rejected, both in the system of Flanagan and Abadi which requires annotations [Flan99b] and in the system of Kobayashi which employs inference [Koba06] as there is no single lock order for *both* threads. Similar considerations apply to the more recent works of Suenaga [Suen08] and Vasconcelos *et al.* [Vasc10] dealing with unstructured locking primitives. A notable exception in the deadlock prevention research is the type system of Boyapati *et al.* [Boya02] which allows for some form of dynamism. Namely, it allows programmers to partition locks into a fixed number of equivalence classes (lock levels), use recursive tree-based data structures to describe their partial order, and also perform a limited set of mutations to these data structures which can change the partial order of locks *within* a given lock level at runtime. Even in this system though, to guarantee soundness, the partial order between lock levels is fixed statically. In contrast, our system does not impose any partial order on locks at compile time, but instead naturally grants locks of different threads during runtime based on the actual program needs and lock contention.

Purely static tools to deadlock prevention employ flow-sensitive static analysis [Engl03] and theorem proving [Flan02] to identify places in the code where programs do not adhere to some global lock acquisition order for all threads. In theory, such static approaches are attractive because they do not incur run-time overhead. In practice however, adhering to a strict lock acquisition order is rarely easy and seems unsuitable for systems programming. Even in simpler application domains, experience has shown that a global lock ordering is inflexible and difficult to enforce in complex, multi-layered software written by large teams of programmers. More importantly, because purely static approaches are by definition conservative, they often reject programs unnecessarily or result in a large number of false alarms.

Recently, Boudol developed a type and effect system for deadlock freedom [Boud09], which is based on *deadlock avoidance*. The effect system calculates for each expression the set of acquired locks and annotates lock operations with the "future" lockset. The run-time system utilizes the inserted annotations so that each lock operation can only proceed when its "future" lockset is unlocked. The main advantage of Boudol's type system is that it allows a larger class of programs to type check compared to purely static approaches to deadlock freedom and thus increases the programming language expressiveness as well as concurrency by allowing arbitrary locking schemes.

The previous example can be rewritten in Boudol's language as follows, assuming that the only lock operations in the two threads are those visible:

$$(\texttt{lock}_{\{y\}}\ x\ \texttt{in}\ \ldots\ \texttt{lock}_{\emptyset}\ y\ \texttt{in}\ \ldots) \ ||\ (\texttt{lock}_{\{x\}}\ y\ \texttt{in}\ \ldots\ \texttt{lock}_{\emptyset}\ x\ \texttt{in}\ \ldots)$$

This program is accepted by Boudol's type system which, in general, allows locks to be acquired in *any* order. At run-time, the first lock operation of the first thread must ensure that y has not been acquired by the second (or any other) thread, before granting x. The second lock operation need not ensure anything special, as its future lockset is empty. (The handling is symmetric for the second thread.). However, Boudol's work does not handle programs with unstructured locking.

From deadlock avoidance approaches, besides Boudol's proposal, a tool that is quite similar to ours is Gadara [Wang08]. Gadara employs whole program analysis to model programs and discrete control theory to synthesize a concurrent logic that avoids deadlocks at run time [Wang09]. Like our work, Gadara targets C/pthreads programs and is claimed to avoid deadlocks quite efficiently because it performs the majority of its deadlock avoidance computations offline. (The tool is not publicly available.) Similarly to our future locksets, Gadara uses the notion of *control places* to decide whether it is safe to admit a lock acquisition. More precisely, a lock acquisition can only proceed when all the control places associated with the lock are available. The mostly static approach followed by Gadara, as well as the lack of alias analysis, results in an over-approximation of the set of run-time locks associated with a control place.

Purely dynamic approaches to deadlock detection [Qin07, Jula08] do not suffer from false positives, but they are often inflexible because when a deadlock is detected it is quite often too late to react on or recover from it. (The programs may have already performed some irrevocable operations such as I/O.)

9.3.3 Other approaches to safe concurrency

Transactional memory. Concurrency errors, such as data races can be eliminated in shared memory systems by the means of transactional memory [Harr03, Harr05, Herl03, Ring05, Welc04] implemented with optimistic concurrency techniques. In addition, lock-related deadlocks can be avoided as the use locks is not required in systems supporting transactional memory. They main idea of transactional memory is that memory accesses within a transaction are logged and the write operations are committed when the log is consistent with the current state of memory. Otherwise, the transaction is rolled back and restarted. Transactions do not interact well with I/O operations, and in some cases, performance can be worse than locking due to the overhead introduced by logging and rollback.

Message passing. Message passing is an alternative form of communication to shared memory. The key idea is that all necessary synchronization is implicitly handled by the message passing abstraction and relieves the programmer from explicit managing synchronization via low-level primitives. Concurrency errors such as data races are eliminated by design in the message passing communication model, but it should be noted that deadlocks cannot be eliminated.

The most simple, and most coarse grain, form of message passing is separate, sequential processes which exchange data via a communication channel. A minimalistic formal description of those communicating sequential processes is the π-calculus [Miln99].

In general the support for spawning new processes is not directly included in mainstream programming languages such as C and is rather provided via libraries such as the Message Passing Interface [Grop96]. Such libraries define a rich set of communication abstractions, ranging from synchronous and asynchronous point-to-point operations to complex collective operations.

Erlang [Arms07] is one of the few programming languages that has built-in support for process creation and communication channel between processes. The language specification prohibits shared memory communication, but it seems that the language implementation and libraries may introduce data races [Chri10].

Chapter 10

Conclusion

In this thesis, we proposed a number of sound static analyses and type systems that guarantee absence of memory access violations and data races for well-typed multithreaded programs. We also proposed a technique that combines a static analysis and dynamic checks to avoid deadlocks in well-typed programs. Let us summarize the main aspects and contributions of this work:

- We presented a multithreaded language that requires explicit type annotations and employs advanced hierarchical region-based management and lock-based synchronization primitives. That language gives explicit control over the lifetime of regions and locks. We then introduced an improved language, which does not require type annotations in source programs and employs reader-writer locks for region hierarchies. Both languages support common multithreaded programming idioms such as data migration and lock transfers between threads. In addition, data can alternate between "thread-local" and "shared" state. We specified an operational semantics and a type system for these languages and proved the theorems that guarantee well-typed programs are free of memory access violations and data races. To the best of our knowledge, this is the first approach that soundly combines the aforementioned features in a language.

- We integrated our formalisms in two concurrent variants of Cyclone and compared the performance of concurrent Cyclone programs against C/pthreads programs. In most cases, concurrent Cyclone programs had negligible overheads.

- We presented an explicitly annotated language with unstructured locking primitives, recursion, mutable references and explicit deallocation primitives. We then presented a simplified language with unstructured locking primitives and recursion that does not require explicit annotations. We specified an operational semantics and a type system for the languages discussed above that guarantees deadlock freedom for well-typed programs. In the case of the former language the type system guarantees that well-typed programs are also memory safe and race free. To the best of our knowledge, this is the first type-based deadlock avoidance technique that soundly guarantees deadlock freedom for languages with unstructured locking.

- We implemented our type system for deadlock avoidance as a static analysis for C/pthreads, and described the design decisions, adjustments and optimizations that we found necessary. Our evaluation results show modest runtime overhead in instrumented C programs.

Overall, we supported the thesis that software reliability can be improved by eliminating invalid memory accesses and concurrency errors such as data races and deadlocks by the means of mostly static analyses in a practical and efficient way. There are possible extensions and improvements to the theories and implementations presented in this thesis that are subject of future work:

Recursive effect inference: regarding the region system, there are several restrictions applying to "external" regions (i.e. regions that exist before a recursive function is called as opposed to regions that are created in a recursive function's body), when computing recursive function effects. For instance, recursive functions are unable to deallocate external regions and in fact when a recursive function returns, the counts of all external regions must be equal to the counts

when the function was called. If a recursive function spawns new threads, it cannot pass to them any locks to external regions and finally recursive functions cannot presume any existing locks on external regions. Similarly to the region system, the deadlock avoidance system requires recursive functions to return the same lock counts as the lock counts prior to the function call. During the benchmarking process, where we ported several programs from C to Cyclone, we did not encounter any programs that could not be handled by our inference algorithm. It would be interesting to further investigate if it is possible to develop a new inference algorithm that lifts the aforementioned restrictions.

Concurrent Cyclone there are plenty of optimizations and improvements that could be done to our implementation. Here, we identify the three most important ones according to our current benchmarking experiences. First, a lexically-scoped Cilk-like [Frig98] construct for allowing parent threads to wait for the children threads to terminate would be highly desirable:

```
join {
      for (int i = 0 ; i < size ; i++)
            spawn worker(a[i]);
    }
```

Second, for certain applications it would be preferable to associate locks to individual references as opposed to regions. It is possible to extend our system to support finer-grained locking by blurring the distinction between regions and references. That is, a fresh region effect could be assigned to new lockable references. In turn, this could be implemented by introducing a new language construct or utilizing existing methods such as tracked pointers and existential types (i.e. a similar mechanism to dynamic regions). At run-time, the new reference would be allocated in the parent region's space and explicit deallocation would still be possible by using reaps [Swam06]. Third, in some of the C benchmarks that outperformed our implementations such as chameneos-redux, distinct parts of shared arrays were concurrently mutated. Extending our type system and run-time region system so that a single array or a recursive data structure can be distributed over a set of regions would be an interesting extension of our work.

Type-based deadlock avoidance for C/pthreads: the process of benchmarking and testing our tool for deadlock avoidance revealed that in a large number of real-world C language projects, locks were used in arrays and recursive data structures. The type systems for deadlock avoidance presented in this thesis are insufficient for programs with recursive data structures and arrays that contain or point to locks. A conservative solution to this problem would require future locksets to contain all possible locks in the program. (Dynamically allocated locks in arrays and recursive data structures can be tracked by overriding the memory allocation functions at run-time.). Less conservative solutions require further investigation. Second, the run-time overhead induced by our technique can be attributed to the elapsed time during lockset computation and the blocking time when a lockset is unavailable. In benchmarks using a large number of nested locks, the lockset computation runtime amounts to half of the overall overhead. The worst-case lockset computation runtime depends on the effect size of the top-level function performing the lock operation as well as the effect sizes of the functions on the call stack. One way of eliminating this overhead is by caching locksets that have already been computed. Another possibility is to statically compute locksets in a context-sensitive manner. The key idea is to maintain a mapping from contexts (i.e. call stacks) to locksets: given the signature of a call stack it would be possible to recover the statically computed lockset. Of course, recursive functions need special treatment and this is subject of future work. Third, our pointer analysis could be improved so that pointer mutations from other threads are tracked. Currently, our analysis requires that pointers are mutated only before they are shared between threads. Finally, systems code involves inline assembly, non-local jumps and special instructions for performing locking. Extending our tool to fully support systems code is target of future work.

Bibliography

[Adve90] Sarita V. Adve and Mark D. Hill, "Weak Ordering – A New Definition", in *Proceedings of the Annual International Symposium on Computer Architecture*, pp. 2–14, New York, NY, USA, 1990, ACM.

[Agar05] Rahul Agarwal, Amit Sasturkar, Liqiang Wang and Scott D. Stoller, "Optimized run-time race detection and atomicity checking using partial discovered types", in *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, pp. 233–242, New York, NY, USA, 2005, ACM.

[Ande08a] Todd Anderson, Neal Glew, Peng Guo, Brian T. Lewis, Wei Liu, Zhanglin Liu, Leaf Petersen, Mohan Rajagopalan, James M. Stichnoth, Gansha Wu and Dan Zhang, "Pillar: A Parallel Implementation Language", in *Proceedings of the International Workshop on Languages and Compilers for Parallel Computing*, vol. 5234 of *LNCS*, pp. 141–155, Springer, 2008.

[Ande08b] Zachary R. Anderson, David Gay, Robert Ennals and Eric Brewer, "SharC: Checking data sharing strategies for multithreaded C", in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 149–158, New York, NY, USA, 2008, ACM.

[Ande09] Zachary R. Anderson, David Gay and Mayur Naik, "Lightweight Annotations for Controlling Sharing in Concurrent Data Structures", in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 98–109, New York, NY, USA, 2009, ACM.

[Arms07] Joe Armstrong, "Erlang - Software for a Concurrent World", in *European Conference on Object-Oriented Programming*, pp. 1–1, Springer, 2007.

[Aust94] Todd M. Austin, Scott E. Breach and Gurindar S. Sohi, "Efficient detection of all pointer and array access errors", in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 290–301, ACM, 1994.

[Bocc09] Robert L. Bocchino, Jr., Vikram S. Adve, Danny Dig, Sarita V. Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, Hyojin Sung and Mohsen Vakilian, "A Type and Effect System for Deterministic Parallel Java", in *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pp. 97–116, New York, NY, USA, 2009, ACM.

[Bocc11] Robert L. Bocchino, Jr., Stephen Heumann, Nima Honarmand, Sarita V. Adve, Vikram S. Adve, Adam Welc and Tatiana Shpeisman, "Safe Nondeterminism in a Deterministic-by-default Parallel Language", in *Conference Record of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 535–548, New York, NY, USA, 2011, ACM.

[Boeh05] Hans-Juergen Boehm, "Threads cannot be implemented as a library", in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 261–268, New York, NY, USA, 2005, ACM Press.

[Boll02] Gregory Bollella and James Gosling, "The real-time specification for Java", *Computer*, vol. 33, no. 6, pp. 47–54, 2002.

[Boud09] Gérard Boudol, "A Deadlock-Free Semantics for Shared Memory Concurrency", in Martin Leucker and Carroll Morgan, editors, *Proceedings of the International Colloquium on Theoretical Aspects of Computing*, vol. 5684 of *LNCS*, pp. 140–154, Springer, 2009.

[Boya01] Chandrasekhar Boyapati and Martin Rinard, "A Parameterized Type System for Race-free Java Programs", in *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pp. 56–69, New York, NY, USA, 2001, ACM Press.

[Boya02] Chandrasekhar Boyapati, Robert Lee and Martin Rinard, "Ownership Types for Safe Programming: Preventing Data Races and Deadlocks", in *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pp. 211–230, New York, NY, USA, November 2002, ACM Press.

[Boya03] Chandrasekhar Boyapati, Alexandru Salcianu, William S. 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*, pp. 324–337, New York, NY, USA, June 2003, ACM Press.

[Boyl03] John Boyland, "Checking Interference with Fractional Permissions", in Radhia Cousot, editor, *Static Analysis: Proceedings of the 10th International Symposium*, vol. 2694 of *LNCS*, pp. 55–72, Springer, 2003.

[Choi02] Jong-Deok Choi, Keunwoo Lee, Alexey Loginov, Robert O'Callahan, Vivek Sarkar and Manu Sridharan, "Efficient and precise datarace detection for multithreaded object-oriented programs", in *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pp. 258–269, ACM, 2002.

[Chri10] Maria Christakis and Konstantinos Sagonas, "Static Detection of Race Conditions in Erlang", in *Practical Aspects of Declarative Languages*, vol. 5937 of *LNCS*, pp. 119–133, Springer, 2010.

[Coff71] Edward G. Coffman, Jr., Michael J. Elphick and Arie Shoshani, "System Deadlocks", *ACM Comput. Surv.*, vol. 3, no. 2, pp. 67–78, 1971.

[Cond07] Jeremy Condit, Matthew Harren, Zachary R. Anderson, David Gay and George C. Necula, "Dependent Types for Low-level Programming", in Rocco De Nicola, editor, *Programming Language and Systems: Proceedings of the European Symposium on Programming*, vol. 4421 of *LNCS*, pp. 520–535, Springer, 2007.

[Cunn07] David Cunningham, Sophia Drossopoulou and Susan Eisenbach, "Universes for Race Safety", in *Proceedings of the Workshop on Verification and Analysis of Multi-threaded Java-like Programs*, pp. 20–51, 2007.

[CURL] "A FTP filesystem based on cURL and FUSE", `http://curlftpfs.sourceforge.net/`.

[DeLi01] 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*, pp. 59–69, New York, NY, USA, 2001, ACM Press.

[Engl03] Dawson Engler and Ken Ashcraft, "RacerX: effective, static detection of race conditions and deadlocks", in *Proceedings of ACM Symposium on Operating Systems Principles*, pp. 237–252, New York, NY, USA, 2003, ACM.

[flam] flam3.com, "Cosmic Recursive Fractal Flames", `http://flam3.com/`.

[Flan99a] Cormac Flanagan and Martín Abadi, "Object Types Against Races", in Jos C. M. Baeten and Sjouke Mauw, editors, *International Conference on Concurrency Theory*, vol. 1664 of *LNCS*, pp. 288–303, Springer, 1999.

[Flan99b] Cormac Flanagan and Martín Abadi, "Types for Safe Locking", in *Programming Language and Systems: Proceedings of the European Symposium on Programming*, vol. 1576 of *LNCS*, pp. 91–108, Springer, 1999.

[Flan00] Cormac Flanagan and Stephen N. Freund, "Type-based race detection for Java", in *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pp. 219–232, ACM, 2000.

[Flan01] Cormac Flanagan and K. Rustan M. Leino, "Houdini, an Annotation Assistant for ESC/-Java", in *Formal Methods Europe*, vol. 2021 of *LNCS*, pp. 500–517, Springer, 2001.

[Flan02] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe and Raymie Stata, "Extended Static Checking for Java", in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 234–245, New York, NY, USA, 2002, ACM.

[Flue06] Matthew Fluet, Gregory Morrisett and Amal Ahmed, "Linear Regions Are All You Need", in Peter Sestoft, editor, *Programming Language and Systems: Proceedings of the European Symposium on Programming*, vol. 3924 of *LNCS*, pp. 7–21, Springer, 2006.

[Fost02] Jeffrey S. Foster, *Type qualifiers: lightweight specifications to improve software quality*, Ph.D. thesis, Berkeley, 2002.

[Fran02] Hubertus Franke, Rusty Russell and Matthew Kirkwood, "Fuss, Futexes and Furwocks: Fast Userlevel Locking in Linux", in *Proceedings of the Ottawa Linux Summit*, pp. 479–495, 2002.

[Frig98] Matteo Frigo, Charles E. Leiserson and Keith H. Randall, "The Implementation of the Cilk-5 Multithreaded Language", in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 212–223, New York, NY, USA, 1998, ACM Press.

[FUSE] "A filesystem in userspace", `http://fuse.sourceforge.net/`.

[Gay01] David Gay and Alexander Aiken, "Language Support for Regions", in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 70–80, New York, NY, USA, 2001, ACM Press.

[Gay03] David Gay, Philip Levis, J. Robert von Behren, Matt Welsh, Eric A. Brewer and David E. Culler, "The nesC Language: A Holistic Approach to Networked Embedded Systems", in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 1–11, New York, NY, USA, 2003, ACM.

[Ghar90] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta and John Hennessy, "Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors", in *Proceedings of the Annual International Symposium on Computer Architecture*, pp. 15–26, New York, NY, USA, 1990, ACM Press.

[Gots07] Alexey Gotsman, Josh Berdine, Byron Cook and Mooly Sagiv, "Thread-modular shape analysis", in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 266–277, ACM, 2007.

[Grop96] William Gropp, Ewing L. Lusk, Nathan E. Doss and Anthony Skjellum, "A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard", *Parallel Computing*, vol. 22, no. 6, pp. 789–828, 1996.

[Gros02] Dan Grossman, Gregory 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*, pp. 282–293, New York, NY, USA, 2002, ACM Press.

[Gros03] Dan Grossman, "Type-safe Multithreading in Cyclone", in *Proceedings of the ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*, pp. 13–25, New York, NY, USA, 2003, ACM Press.

[Harr03] Timothy L. Harris and Keir Fraser, "Language Support for Lightweight Transactions", in *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pp. 388–402, New York, NY, USA, 2003, ACM Press.

[Harr05] Tim Harris, Simon Marlow, Simon L. Peyton Jones and Maurice Herlihy, "Composable Memory Transactions", in *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 48–60, ACM, 2005.

[Heng01] Fritz Henglein, Henning Makholm and Henning Niss, "A Direct Approach to Control-flow Sensitive Region-based Memory Management", in *Proceedings of the International Conference on Principles and Practice of Declarative Programming*, pp. 175–186, New York, NY, USA, 2001, ACM.

[Henz04] Thomas A. Henzinger, Ranjit Jhala and Rupak Majumdar, "Race Checking by Context Inference", in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 1–13, ACM, 2004.

[Herl03] Maurice Herlihy, Victor Luchangco, Mark Moir and William N. Scherer III, "Software transactional memory for dynamic-sized data structures", in *Proceedings of the Symposium on Principles of Distributed Computing*, pp. 92–101, ACM, 2003.

[Hobo08] Aquinas Hobor, Andrew W. Appel and Francesco Zappa Nardelli, "Oracle Semantics for Concurrent Separation Logic", in *Programming Language and Systems: Proceedings of the European Symposium on Programming*, vol. 4960 of *LNCS*, pp. 353–367, Springer, 2008.

[Jim02] Trevor Jim, Greg Morrisett, Dan Grossman, Michael Hicks, James Cheney and Yanling Wan, "Cyclone: A safe dialect of C", in *Usenix Annual Technical Conference*, USENIX Association, 2002.

[Jula08] Horatiu Jula, Daniel Tralamazza, Cristian Zamfir and George Candea, "Deadlock Immunity: Enabling Systems to Defend Against Deadlocks", in Richard Draves and Robbert van Renesse, editors, *Proceedings of the Symposium on Operating Systems Design and Implementation*, pp. 295–308, USENIX Association, 2008.

[Koba06] Naoki Kobayashi, "A New Type System for Deadlock-Free Processes", in C. Baier and H. Hermanns, editors, *International Conference on Concurrency Theory*, vol. 4137 of *LNCS*, pp. 233–247, Springer, 2006.

[Lamp79] Leslie Lamport, "A New Approach to Proving the Correctness of Multiprocess Programs", *ACM Transactions on Programming Languages and Systems*, vol. 1, no. 1, pp. 84–97, 1979.

[Mats10] Nicholas D. Matsakis and Thomas R. Gross, "A time-aware type system for data-race protection and guaranteed initialization", in *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pp. 634–651, New York, NY, USA, 2010, ACM.

[MIGR] "A tool that estimates population size and migration rate", `http://popgen.sc.fsu.edu/Migrate/Migrate-n.html`.

[Miln99] Robin Milner, *Communicating and mobile systems - the Pi-calculus*, Cambridge University Press, 1999.

[Necu05] George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak and Westley Weimer, "CCured: Type-safe Retrofitting of Legacy Software", *ACM Transactions on Programming Languages and Systems*, vol. 27, no. 3, pp. 477–526, 2005.

[NGOR] "A password recovery tool for Oracle Database", `http://code.google.com/p/ngorca/`.

[OCal03] Robert O'Callahan and Jong-Deok Choi, "Hybrid Dynamic Data Race Detection", in *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 167–178, New York, NY, USA, 2003, ACM Press.

[Prat06] Polyvios Pratikakis, Jeffrey S. Foster. and Michael Hicks, "Locksmith: context-sensitive correlation analysis for race detection", in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 320–331, New York, NY, USA, 2006, ACM.

[Qade04] Shaz Qadeer and Dinghao Wu, "KISS: Keep It Simple and Sequential", in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 14–24, ACM, 2004.

[Qin07] Feng Qin, Joseph Tucek, Yuanyuan Zhou and Jagadeesan Sundaresan, "Rx: Treating bugs as allergies — a safe method to survive software failures", *ACM Transactions on Computer Systems*, vol. 25, no. 3, p. 7/2, 2007.

[Reyn02] John C. Reynolds, "Separation logic: A logic for shared mutable data structures", in *Proceedings of the IEEE Symposium on Logic in Computer Science*, pp. 55–74, IEEE Computer Society Press, 2002.

[Ring05] Michael F. Ringenburg and Dan Grossman, "AtomCaml: first-class atomicity via rollback", in *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*, pp. 92–104, ACM, 2005.

[Sava97] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro and Thomas E. Anderson, "Eraser: A Dynamic Data Race Detector for Multi-Threaded Programs", in *Proceedings of ACM Symposium on Operating Systems Principles*, pp. 27–37, ACM, 1997.

[SSHF] "SSH FileSystem", `http://fuse.sourceforge.net/sshfs.html`.

[Stal11] Richard M. Stallman and the GCC Developer Community, *Using the GNU Compiler Collection*, 2011. `http://gcc.gnu.org/onlinedocs/` (version 4.6.0).

[Suen08] Kohei Suenaga, "Type-Based Deadlock-Freedom Verification for Non-Block-Structured Lock Primitives and Mutable References", in G. Ramalingam, editor, *Asian Symposium on Programming Languages and Systems*, vol. 5356 of *LNCS*, pp. 155–170, Springer, 2008.

[Swam06] Nikhil Swamy, Michael Hicks, Greg Morrisett, Dan Grossman and Trevor Jim, "Safe Manual Memory Management in Cyclone", *Science of Computer Programming*, vol. 62, no. 2, pp. 122–144, 2006.

[Tera08] Tachio Terauchi, "Checking race freedom via linear programming", in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 1–10, ACM, 2008.

[tgre] "Multithreaded `grep`", Part of Sun Microsystems' *Multithreaded Programming Guide*, available at `http://docs.sun.com/app/docs/doc/806-5257`.

[Toft94] Mads Tofte and Jean-Pierre Talpin, "Implementation of the Typed Call-by-value λ-calculus using a Stack of Regions", in *Conference Record of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 188–201, New York, NY, USA, 1994, ACM Press.

[Vasc10] Vasco Vasconcelos, Francisco Martin and Tiago Cogumbreiro, "Type Inference for Deadlock Detection in a Multithreaded Polymorphic Typed Assembly Language", in Alastair R. Beresford and Simon Gay, editors, *Proceedings of the Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software*, vol. 17 of *EPTCS*, pp. 95–109, 2010.

[Voun07] Jan Wen Voung, Ranjit Jhala and Sorin Lerner, "RELAY: static race detection on millions of lines of code", in *Proceedings of the joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pp. 205–214, New York, NY, USA, 2007, ACM.

[Walk01] David Walker and Kevin Watkins, "On Regions and Linear Types", in *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*, pp. 181–192, New York, NY, USA, October 2001, ACM Press.

[Wang08] Yin Wang, Terence Kelly, Manjunath Kudlur, Stéphane Lafortune and Scott Mahlke, "Gadara: Dynamic Deadlock Avoidance for Multithreaded Programs", in Richard Draves and Robbert van Renesse, editors, *Proceedings of the Symposium on Operating Systems Design and Implementation*, pp. 281–294, USENIX Association, 2008.

[Wang09] Yin Wang, Stéphane Lafortune, Terence Kelly, Manjunath Kudlur and Scott Mahlke, "The theory of deadlock avoidance via discrete control", in *Conference Record of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 252–263, New York, NY, USA, 2009, ACM.

[Welc04] Adam Welc, Suresh Jagannathan and Antony L. Hosking, "Transactional Monitors for Concurrent Objects", in *European Conference on Object-Oriented Programming*, pp. 519–542, Springer, 2004.

[Zhao04] Tian Zhao, James Noble and Jan Vitek, "Scoped Types for Real-Time Java", in *Proceedings of the 25th IEEE International Real-Time Systems Symposium*, pp. 241–251, IEEE Computer Society, 2004.

[Zhao08] Tian Zhao, Jason Baker, James Hunt, James Noble and Jan Vitek, "Implicit ownership types for memory management", *Science of Computer Programming*, vol. 71, no. 3, pp. 213–241, 2008.

Appendix A

Formal semantics and proof of soundness for Chapter 2

A.1 Language syntax

Value	$v \;::=\; f \mid c \mid \mathtt{rgn}_\iota \mid \mathtt{loc}_l$
Expression	$e \;::=\; x \mid \mathsf{c} \mid f \mid (e\ e)^\xi \mid e[r]$
	$\quad\mid\; \mathtt{new}\, e \mathbin{@} e \mid e := e \mid \mathtt{deref}\, e$
	$\quad\mid\; \mathtt{newrgn}^r\ \rho, x \mathbin{@} e\ \mathtt{in}\ e$
	$\quad\mid\; \mathtt{cap}^r_\eta\, e \mid \mathtt{rgn}_\iota \mid \mathtt{loc}_l \mid \mathtt{pop}_\gamma\ e$
Capability kind	$\psi \;::=\; \mathsf{rg} \mid \mathsf{lk}$
Capability op	$\eta \;::=\; \psi+ \mid \psi-$
Capability	$\kappa \;::=\; n, n \mid \overline{n, n}$
Region parent	$\pi \;::=\; r \mid \bot$
Effect	$\gamma \;::=\; \emptyset \mid \gamma, r^\kappa \triangleright \pi$
Type	$\tau \;::=\; b \mid \langle\rangle \mid \tau \xrightarrow{\gamma \to \gamma} \tau \mid \forall \rho.\, \tau$
	$\quad\mid\; \mathtt{ref}(\tau, r) \mid \mathtt{rgn}(r)$
Function	$f \;::=\; \lambda x.\, e\ \mathsf{as}\ \tau \xrightarrow{\gamma \to \gamma} \tau \mid \Lambda \rho.\, f$
Calling mode	$\xi \;::=\; \mathsf{seq} \mid \mathsf{par}$
Region	$r \;::=\; \rho \mid \iota \mid \iota @ n$

A.2 Operational semantics

Auxiliary syntax for operational semantics

Stack	$\sigma \;::=\; \emptyset \mid \sigma; \gamma$
Hierarchy	$\delta \;::=\; \emptyset \mid \delta, n \mapsto \sigma$
Contents	$H \;::=\; \emptyset \mid H, \ell \mapsto v$
Region list	$S \;::=\; \emptyset \mid S, \iota \mapsto H$
Threads	$T \;::=\; \emptyset \mid T, n : e$
Configuration	$C \;::=\; \delta; S; T$
Evaluation context	$E \;::=\; \square \mid (E\ e)^\xi \mid (v\ E)^\xi \mid E\,[r]$
	$\quad\mid\; \mathtt{newrgn}^r\ \rho, x \mathbin{@} E\ \mathtt{in}\ e \mid \mathtt{cap}^r_\eta\, E$
	$\quad\mid\; \mathtt{new}\, v \mathbin{@} E \mid \mathtt{deref}\, E \mid E := e$
	$\quad\mid\; v := E \mid \mathtt{new}\, E \mathbin{@} e \mid \mathtt{pop}_\gamma\ E$

Evaluation relation $C \to C'$

$$\frac{\delta = \delta'', n \mapsto \sigma \qquad \sigma; S; e \to \sigma'; S'; e' \qquad \delta' = \delta'', n \mapsto \sigma' \qquad \vdash \delta'}{\delta; S; T, n : E[e] \rightsquigarrow \delta'; S'; T, n : E[e']} \textit{(E-S)}$$

$$\frac{\begin{array}{c} v_1 \equiv \lambda x.\, e \;\mathsf{as}\; \tau_1 \xrightarrow{\gamma_1 \to \gamma_2} \tau_2 \qquad \delta = \delta'', n \mapsto \sigma; \gamma \qquad \text{fresh } n' \\ \mathsf{par} \vdash \gamma' = \emptyset \oplus (\gamma \ominus \gamma_1) \qquad \delta' = \delta'', n \mapsto \sigma; \gamma', n' \mapsto \emptyset; \gamma_1 \end{array}}{\delta; S; T, n : E[(v_1\ v)^{\mathsf{par}}] \rightsquigarrow \delta'; S; T, n : E[()], n' : (v_1\ v)^{\mathsf{seq}}} \textit{(E-SN)}$$

$$\frac{\begin{array}{c} \sigma = \sigma'; \gamma \qquad \textit{is\_live}(\gamma, r) \qquad \gamma = \gamma', r^{\kappa} \triangleright \pi \\ \kappa' = [\![\eta]\!](\kappa) \qquad \sigma'' = \sigma'; \textit{live}(\gamma', r^{\kappa'} \triangleright \pi) \end{array}}{\sigma; S; \mathsf{cap}^r_\eta\ \mathtt{rgn}_{\bar{r}} \to \sigma''; S; ()} \textit{(E-C)}$$

$$\frac{\delta = \delta', n \mapsto (\emptyset; \emptyset)}{\delta; S; T, n : () \rightsquigarrow \delta'; S; T} \textit{(E-T)}$$

$$\frac{\sigma = \sigma'; \gamma \qquad \mathsf{seq} \vdash \gamma = \gamma_1 \oplus \gamma_r \qquad \sigma'' = \sigma'; \gamma_r; \gamma_1}{\sigma; S; ((\lambda x.\, e \;\mathsf{as}\; \tau_1 \xrightarrow{\gamma_1 \to \gamma_2} \tau_2)\ v)^{\mathsf{seq}} \to \sigma''; S; \mathsf{pop}_{\gamma_r}\ e[v/x]} \textit{(E-A)}$$

$$\frac{\text{fresh } n'}{\sigma; S; (\Lambda\rho.\, f)[r] \to \sigma; S; f[\bar{r}@n'/\rho]} \textit{(E-RP)}$$

$$\frac{\sigma = \sigma'; \gamma \qquad \textit{is\_accessible}(\gamma, r) \qquad (\ell \mapsto v) \in S(\bar{r})}{\sigma; S; \mathtt{deref}\ \mathtt{loc}_\ell \to \sigma; S; v} \textit{(E-D)}$$

$$\frac{\sigma = \sigma'; \gamma \qquad \textit{is\_live}(\gamma, r) \qquad \text{fresh } \ell}{\sigma; S; \mathtt{new}\ v\ @\ \mathtt{rgn}_{\bar{r}} \to \sigma; S[\bar{r} \mapsto S(\bar{r}), \ell \mapsto v]; \mathtt{loc}_\ell} \textit{(E-NR)}$$

$$\frac{\sigma = \sigma'; \gamma \qquad \textit{is\_accessible}(\gamma, r) \qquad (\ell \mapsto v_1) \in S(\bar{r})}{\sigma; S; \mathtt{loc}_\ell := v \to \sigma; S(\bar{r})[\ell \mapsto v]; ()} \textit{(E-AS)}$$

$$\frac{\sigma = \sigma'; \gamma \qquad \textit{is\_live}(\gamma, r) \qquad \text{fresh } \imath \qquad \sigma'' = \sigma'; \gamma, \imath^{1,1} \triangleright r}{\sigma; S; \mathtt{newrgn}^r\ \rho, x\ @\ \mathtt{rgn}_{\bar{r}}\ \mathtt{in}\ e \to \sigma''; S, \imath \mapsto \emptyset; e[\imath/\rho][\mathtt{rgn}_\imath/x]} \textit{(E-NG)}$$

$$\frac{\sigma = \sigma'; \gamma_r; \gamma' \qquad \mathsf{seq} \vdash \gamma'' = \gamma' \oplus (\gamma_r \ominus \emptyset) \qquad \sigma'' = \sigma'; \gamma''}{\sigma; S; \mathsf{pop}_{\gamma_r}\ v \to \sigma''; S; v} \textit{(E-E)}$$

Helper predicates and functions

$$\frac{(r^{\kappa} \triangleright \bot) \in \gamma \qquad \textit{rg}(\kappa) > 0}{\textit{is\_live}(\gamma, r)} \qquad \frac{\gamma = \gamma', r^{\kappa} \triangleright r' \qquad \textit{rg}(\kappa) > 0 \qquad \textit{is\_live}(\gamma', r')}{\textit{is\_live}(\gamma, r)}$$

$$\frac{(r^{\kappa} \triangleright \pi) \in \gamma \qquad \textit{lk}(\kappa) > 0 \qquad \textit{is\_live}(\gamma, r)}{\textit{is\_accessible}(\gamma, r)}$$

$$\frac{\gamma = \gamma', r^{\kappa}\triangleright r' \quad \mathit{lk}(\kappa) = 0 \quad \mathit{rg}(\kappa) > 0 \quad \mathit{is\_accessible}(\gamma', r')}{\mathit{is\_accessible}(\gamma, r)}$$

$$\frac{\begin{array}{c}\sigma \simeq \sigma_1; \gamma, \imath^{\kappa}\triangleright \pi + \sigma_2 \\ \mathit{lk}(\kappa) > 0 \vee (\pi = r \wedge \mathit{is\_accessible}(\sigma, \bar{r}))\end{array}}{\mathit{is\_accessible}(\sigma, \imath)}$$

$$\frac{\begin{array}{c}\sigma \simeq \sigma_1; \gamma, \imath^{\kappa}\triangleright \pi \Rightarrow \mathit{is\_pure}(\kappa) \wedge \mathit{rg}(\kappa) = 0 \wedge \\ \imath \notin \mathit{dom}(\emptyset; \gamma) \wedge \sigma_1 \neq \emptyset \Rightarrow \mathit{zero\_pure}(\sigma_1, \imath)\end{array}}{\mathit{zero\_pure}(\sigma, \imath)}$$

$$\frac{\vdash \delta \quad \sigma \vdash \delta}{\vdash \delta, n \mapsto \sigma} \qquad \frac{}{\vdash \emptyset}$$

$$\frac{\begin{array}{c}\sigma \vdash \delta \\ \forall \imath \in \mathit{dom}(\sigma).\mathit{is\_accessible}(\sigma, \imath) \Rightarrow \neg \mathit{is\_accessible}(\sigma', \imath)\end{array}}{\sigma \vdash \delta, n \mapsto \sigma'}$$

$$\frac{\begin{array}{c}\forall \imath \in \mathit{dom}(\sigma).\sigma \simeq \sigma_1; \gamma, \imath^{\kappa}\triangleright \pi + \sigma_2 \wedge \mathit{rg}(\kappa) > 0 \wedge \mathit{is\_pure}(\kappa) \\ \Rightarrow \mathit{zero\_pure}(\sigma_1, \imath) \wedge \imath \notin \mathit{dom}(\sigma_2; \gamma)\end{array}}{\sigma \vdash \emptyset}$$

$$\begin{array}{lcl}
\bar{r} & = & \begin{cases} \imath & \text{if } r = \imath \\ \bar{r'} & \text{if } r = r'@n' \end{cases} \\
\sigma \simeq \sigma' & = & \begin{cases} \sigma_1 \simeq \sigma_2 \wedge \gamma_1 \simeq \gamma_2 & \text{if } \sigma = \sigma_1; \gamma_1 \wedge \sigma' = \sigma_2; \gamma_2 \\ \sigma_1 \equiv \sigma_2 & \text{otherwise} \end{cases} \\
\mathit{rg}(\kappa) & = & n_1 \quad \text{if } \kappa = n_1, n_2 \vee \kappa = \overline{n_1, n_2} \\
\mathit{lk}(\kappa) & = & n_2 \quad \text{if } \kappa = n_1, n_2 \vee \kappa = \overline{n_1, n_2} \\
\mathit{live}(\gamma) & = & \{\, r^{\kappa}\triangleright \pi \mid (r^{\kappa}\triangleright \pi) \in \gamma \wedge \mathit{is_live}(\gamma, r) \,\} \\
\sigma_1 + \sigma_2 & = & \begin{cases} (\sigma_1 + \sigma); \gamma_n & \text{if } \sigma_2 \equiv \sigma; \gamma_n \\ \sigma_1 & \text{if } \sigma_2 \equiv \emptyset \end{cases} \\
\gamma_1 + \gamma_2 & = & \begin{cases} (\gamma_1 + \gamma), r^{\kappa}\triangleright \pi & \text{if } \gamma_2 \equiv \gamma, r^{\kappa}\triangleright \pi \\ \gamma_1 & \text{if } \gamma_2 \equiv \emptyset \end{cases} \\
\mathit{dom}(\gamma) & = & \{\, r \mid (r^{\kappa}\triangleright \pi) \in \gamma \,\} \\
\mathit{dom}(\sigma) & = & \begin{cases} \{\bar{r} \mid (r^{\kappa}\triangleright \pi) \in \gamma\} \cup \mathit{dom}(\sigma') & \text{if } \sigma = \sigma'; \gamma \\ \emptyset & \text{if } \sigma = \emptyset \end{cases}
\end{array}$$

A.3 Static semantics

Static semantics syntax

Region List $R ::= \emptyset \mid R, \imath$

Type variable list $\square ::= \emptyset \mid \Delta, \rho$

Memory List $M ::= \emptyset \mid M, \ell \mapsto (\tau, \imath)$

Variable list $\square ::= \emptyset \mid \Gamma, x : \tau$

Helper functions and predicates

$$
\begin{array}{lcl}
\mathit{is\_pure}(\kappa) & = & \exists n_1.\, \exists n_2.\, \kappa = n_1, n_2 \\
 & & \quad \text{if } \eta \equiv \psi \pm 1 \wedge \mathit{is\_pure}(\kappa) \Leftrightarrow \mathit{is\_pure}(\kappa') \wedge \\
[\![\eta]\!]\,(\kappa) & = & \kappa' \quad (\psi = \mathsf{rg} \Rightarrow \mathit{rg}(\kappa') = \mathit{rg} \pm 1 \wedge \mathit{lk}(\kappa') = \mathit{lk}(\kappa)) \wedge \\
 & & \quad (\psi = \mathsf{lk} \Rightarrow \mathit{lk}(\kappa') = \mathit{lk} \pm 1 \wedge \mathit{rg}(\kappa') = \mathit{rg}(\kappa)) \\
\mathit{valid\_pure}(\gamma) & = & \forall r^{\kappa} \triangleright \pi \in \gamma. \exists \gamma_1. \gamma = \gamma_1, r^{\kappa} \triangleright \pi \wedge \mathit{is\_pure}(\kappa) \Rightarrow \forall r' \simeq r. r' \notin \mathit{dom}(\gamma_1) \\
\mathit{ok}(\gamma_1; \gamma_2) & = & \mathit{valid\_pure}(\gamma_1) \wedge \mathit{valid\_pure}(\gamma_2) \\
\mathit{valid}(\gamma_1; \gamma_2) & = & (\forall (r^{\kappa} \triangleright \pi) \in \gamma_1.\, \forall (r^{\kappa'} \triangleright \pi') \in \gamma_2.\, \pi = \pi' \wedge (\mathit{is\_pure}(\kappa) \Leftrightarrow \\
 & & \mathit{is\_pure}(\kappa'))) \wedge \mathit{live}(\gamma_1) = \gamma_1 \wedge \mathit{live}(\gamma_2) = \gamma_2 \wedge \mathit{dom}(\gamma_2) \subseteq \mathit{dom}(\gamma_1) \\
r_1 \simeq r_2 & \equiv & \begin{cases} \iota_1 \equiv \iota_2 & \text{if } r_1 \equiv \iota_1 @ n_1 \wedge r_2 \equiv \iota_2 @ n_2 \\ r_1 \equiv r_2 & \text{otherwise} \end{cases} \\
\pi_1 \simeq \pi_2 & \equiv & \begin{cases} r_1 \simeq r_2 & \text{if } \pi_1 \equiv r_1 \wedge \pi_1 \equiv r_2 \\ \pi_1 \equiv \pi_2 & \text{otherwise} \end{cases} \\
\gamma_1 \simeq \gamma_2 & \equiv & \begin{cases} \gamma_3 \simeq \gamma_4 \wedge r_1 \simeq r_2 \wedge \pi_1 \simeq \pi_2 & \text{if } \gamma_1 = \gamma_3, r_1^{\kappa} \triangleright \pi_1 \wedge \gamma_2 = \gamma_4, r_2^{\kappa} \triangleright \pi_2 \\ \gamma_1 \equiv \gamma_2 & \text{otherwise} \end{cases} \\
\tau_1 \simeq \tau_2 & \equiv & \begin{cases} \tau_1 \equiv \tau_2 & \text{if } (\{\tau_1\} \cup \{\tau_2\}) \cap \{\langle\rangle, b\} \\ \tau_3 \simeq \tau_4 \wedge r_1 \simeq r_2 & \text{if } \tau_1 = \mathtt{ref}(\tau_3, r_1) \wedge \tau_2 = \mathtt{ref}(\tau_4, r_2) \\ r_1 \simeq r_2 & \text{if } \tau_1 = \mathtt{rgn}(r_1) \wedge \tau_2 = \mathtt{rgn}(r_2) \\ \tau_3 \simeq \tau_5 \wedge \tau_4 \simeq \tau_6 \wedge \gamma_1 \simeq \gamma_3 \wedge \gamma_2 \simeq \gamma_4 & \text{if } \tau_1 \equiv \tau_3 \xrightarrow{\gamma_1 \to \gamma_2} \tau_4 \wedge \tau_2 \equiv \tau_5 \xrightarrow{\gamma_3 \to \gamma_4} \tau_6 \\ \tau_3[\rho/\rho_1] \simeq \tau_4[\rho/\rho_2] & \text{if } \tau_1 \equiv \forall \rho_1.\, \tau_3 \wedge \tau_2 \equiv \forall \rho_2.\, \tau_4 \wedge \text{fresh } \rho \end{cases} \\
\mathit{set}(\gamma; \gamma') & = & (\forall (r^{\kappa} \triangleright \pi) \in \gamma. \exists \gamma_1. \gamma = \gamma_1, r^{\kappa} \triangleright \pi \wedge r \notin \mathit{dom}(\gamma_1)) \\
 & & \wedge (\forall (r^{\kappa} \triangleright \pi) \in \gamma'. \exists \gamma_1. \gamma' = \gamma_1, r^{\kappa} \triangleright \pi \wedge r \notin \mathit{dom}(\gamma_1))
\end{array}
$$

Well-formedness rules

$$\frac{}{R; \Delta \vdash \emptyset} \qquad \frac{R; \Delta \vdash \gamma_1 \quad R; \Delta \vdash r_1 \quad r_1 \neq \pi \quad \pi = r_2 \Rightarrow R; \Delta \vdash r_2}{R; \Delta \vdash \gamma_1, r_1^{\kappa} \triangleright \pi}$$

$$\frac{r \in \Delta \uplus R}{R; \Delta \vdash r} \qquad \frac{R; \Delta \vdash \iota}{R; \Delta \vdash \iota @ n}$$

$$\frac{}{R; \Delta \vdash b} \qquad \frac{R; \Delta \vdash r}{R; \Delta \vdash \mathtt{rgn}(r)} \qquad \frac{R; \Delta, \rho \vdash \tau}{R; \Delta \vdash \forall \rho.\, \tau}$$

$$\frac{R; \Delta \vdash \tau \quad R; \Delta \vdash r}{R; \Delta \vdash \mathtt{ref}(\tau, r)} \qquad \frac{}{R; \Delta \vdash \langle\rangle}$$

$$\frac{\mathit{valid}(\gamma_1; \gamma_2) \qquad R; \Delta \vdash \tau_1 \quad R; \Delta \vdash \gamma_1 \quad R; \Delta \vdash \tau_2 \quad R; \Delta \vdash \gamma_2}{R; \Delta \vdash \tau_1 \xrightarrow{\gamma_1 \to \gamma_2} \tau_2}$$

$$\frac{R; \Delta \vdash \tau_1 \quad x \notin \mathit{dom}(\Gamma_1) \quad R; \Delta \vdash \Gamma_1}{R; \Delta \vdash \Gamma_1, x : \tau_1} \qquad \frac{R \vdash M_1 \quad \ell \notin \mathit{dom}(M_1) \quad R; \emptyset \vdash \mathtt{ref}(\tau_1, \iota)}{R \vdash M_1, \ell \mapsto (\tau_1, \iota)}$$

$$\frac{R \vdash M \qquad R;\Delta \vdash \Gamma \qquad \mathit{set}(\gamma;\gamma') \quad R;\Delta \vdash \gamma \qquad \mathit{ok}(\gamma;\gamma') \quad R;\Delta \vdash \gamma'}{\vdash R;M;\Delta;\Gamma;\gamma;\gamma'}$$

Typing rules

$$\frac{\vdash R;M;\Delta;\Gamma;\gamma;\gamma \qquad (x:\tau) \in \Gamma \qquad \tau \simeq \tau'}{R;M;\Delta;\Gamma \vdash x : \tau' \,\&\, (\gamma;\gamma)} \textit{ (T-V)} \qquad \frac{\vdash R;M;\Delta;\Gamma;\gamma;\gamma}{R;M;\Delta;\Gamma \vdash \mathtt{c} : b \,\&\, (\gamma;\gamma)} \textit{ (T-I)}$$

$$\frac{\vdash R;M;\Delta;\Gamma;\gamma;\gamma}{R;M;\Delta;\Gamma \vdash () : \langle\rangle \,\&\, (\gamma;\gamma)} \textit{ (T-U)} \qquad \frac{\vdash R;M;\Delta;\Gamma;\gamma;\gamma \qquad R;\Delta \vdash \imath \qquad r \simeq \imath}{R;M;\Delta;\Gamma \vdash \mathtt{rgn}_\imath : \mathtt{rgn}(r) \,\&\, (\gamma;\gamma)} \textit{ (T-R)}$$

$$\frac{\vdash R;M;\Delta;\Gamma;\gamma;\gamma \qquad (\ell \mapsto (\tau,\imath)) \in M \qquad \tau' \simeq \mathtt{ref}(\tau,\imath)}{R;M;\Delta;\Gamma \vdash \mathtt{loc}_l : \tau' \,\&\, (\gamma;\gamma)} \textit{ (T-L)}$$

$$\frac{R;M;\Delta,\rho;\Gamma \vdash f : \tau \,\&\, (\gamma;\gamma)}{R;M;\Delta;\Gamma \vdash \Lambda\rho.\, f : \forall\rho.\,\tau \,\&\, (\gamma;\gamma)} \textit{ (T-RF)}$$

$$\frac{\vdash R;M;\Delta;\Gamma;\gamma;\gamma \qquad \tau \equiv \tau_1 \xrightarrow{\gamma_1 \to \gamma_2} \tau_2 \qquad \mathit{set}(\gamma_1;\gamma_2) \qquad \tau \simeq \tau' \qquad \mathit{ok}(\gamma_1;\gamma_2) \Rightarrow R;M;\Delta;\Gamma,x:\tau_1 \vdash e : \tau_2 \,\&\, (\gamma_1;\gamma_2)}{R;M;\Delta;\Gamma \vdash \lambda x.\, e \;\mathtt{as}\; \tau : \tau' \,\&\, (\gamma;\gamma)} \textit{ (T-F)}$$

$$\frac{R;M;\Delta;\Gamma \vdash e_1 : \tau_1 \xrightarrow{\gamma_1 \to \gamma_2} \tau_2 \,\&\, (\gamma;\gamma') \qquad \xi = \mathsf{par} \Rightarrow \tau_2 = \langle\rangle \qquad R;M;\Delta;\Gamma \vdash e_2 : \tau_1 \,\&\, (\gamma';\gamma'') \qquad \xi \vdash \gamma''' = \gamma_2 \oplus (\gamma'' \ominus \gamma_1)}{R;M;\Delta;\Gamma \vdash (e_1\ e_2)^\xi : \tau_2 \,\&\, (\gamma;\gamma''')} \textit{ (T-AP)}$$

$$\frac{R;\Delta \vdash r \qquad R;M;\Delta;\Gamma \vdash e : \forall\rho.\,\tau \,\&\, (\gamma;\gamma')}{R;M;\Delta;\Gamma \vdash e\ [r] : \tau[r/\rho] \,\&\, (\gamma;\gamma')} \textit{ (T-RP)}$$

$$\frac{R;M;\Delta;\Gamma \vdash e_1 : \mathtt{rgn}(r) \,\&\, (\gamma;\gamma') \qquad \mathit{is\_live}(\gamma',r) \qquad R;\Delta \vdash \tau \qquad R;M;\Delta,\rho;\Gamma,x:\mathtt{rgn}(\rho) \vdash e_2 : \tau \,\&\, (\gamma',\rho^{1,1} \triangleright r;\gamma'') \qquad \rho \notin \mathit{dom}(\gamma'')}{R;M;\Delta;\Gamma \vdash \mathtt{newrgn}^r\ \rho, x\ @\ e_1\ \mathtt{in}\ e_2 : \tau \,\&\, (\gamma;\gamma'')} \textit{ (T-NG)}$$

$$\frac{R;M;\Delta;\Gamma \vdash e_1 : \tau \,\&\, (\gamma;\gamma') \qquad \mathit{is\_live}(\gamma'',r) \qquad R;M;\Delta;\Gamma \vdash e_2 : \mathtt{rgn}(r) \,\&\, (\gamma';\gamma'')}{R;M;\Delta;\Gamma \vdash \mathtt{new}\ e_1\ @\ e_2 : \mathtt{ref}(\tau,r) \,\&\, (\gamma;\gamma'')} \textit{ (T-NR)}$$

$$\frac{R;M;\Delta;\Gamma \vdash e_1 : \mathtt{ref}(\tau,r) \,\&\, (\gamma;\gamma') \qquad R;M;\Delta;\Gamma \vdash e_2 : \tau \,\&\, (\gamma';\gamma'') \qquad \mathit{is\_accessible}(\gamma'',r)}{R;M;\Delta;\Gamma \vdash e_1 := e_2 : \langle\rangle \,\&\, (\gamma;\gamma'')} \textit{ (T-A)}$$

$$\frac{\mathsf{seq} \vdash \gamma' = \gamma_2 \oplus (\gamma_r \ominus \emptyset) \qquad R;\Delta \vdash \gamma_r \qquad \mathit{ok}(\gamma_r;\emptyset) \qquad R;M;\Delta;\Gamma \vdash e : \tau' \,\&\, (\gamma_1;\gamma_2) \qquad \tau \simeq \tau' \qquad \mathit{set}(\gamma_r;\emptyset)}{R;M;\Delta;\Gamma \vdash \mathtt{pop}_{\gamma_r}\ e : \tau \,\&\, (\gamma_1;\gamma')} \textit{ (T-E)}$$

$$\frac{\begin{array}{c}\mathit{is\_live}(\gamma', r^{\kappa} \triangleright \pi, r) \qquad \gamma'' = \mathit{live}(\gamma', r^{\kappa'} \triangleright \pi) \\ R; M; \Delta; \Gamma \vdash e_1 : \mathtt{rgn}(r) \,\&\, (\gamma;\gamma', r^{\kappa} \triangleright \pi) \qquad \kappa' = [\![\eta]\!]\,(\kappa)\end{array}}{R; M; \Delta; \Gamma \vdash \mathtt{cap}^{r}_{\eta}\ e_1 : \langle\rangle \,\&\, (\gamma;\gamma'')} \textit{ (T-CP)}$$

$$\frac{R; M; \Delta; \Gamma \vdash e : \mathtt{ref}(\tau, r) \,\&\, (\gamma;\gamma') \qquad \mathit{is\_accessible}(\gamma', r)}{R; M; \Delta; \Gamma \vdash \mathtt{deref}\ e : \tau \,\&\, (\gamma;\gamma')} \textit{ (T-D)}$$

Auxiliary typing rules

$$\frac{\begin{array}{c}\xi \vdash \gamma = \gamma_1 \oplus \gamma_r \qquad \xi \vdash \gamma' = \gamma_2 \oplus \gamma_r \\ \gamma'' = \mathit{live}(\gamma') \qquad \mathit{ok}(\gamma_1;\gamma_2) \qquad \xi = \mathsf{par} \Rightarrow \gamma_2 = \emptyset\end{array}}{\xi \vdash \gamma'' = \gamma_2 \oplus (\gamma \ominus \gamma_1)} \textit{ (ESJ)}$$

$$\frac{\begin{array}{c}\xi \vdash \gamma, r^{\kappa_2} \triangleright \pi = \gamma_1 \oplus \gamma_2 \\ \xi \vdash \kappa = \kappa_1 + \kappa_2 \qquad \pi \simeq \pi' \qquad r' \simeq r\end{array}}{\xi \vdash \gamma, r^{\kappa} \triangleright \pi = \gamma_1, r'^{\kappa_1} \triangleright \pi' \oplus \gamma_2} \textit{ (ES-C)}$$

$$\frac{}{\xi \vdash \gamma = \emptyset \oplus \gamma} \textit{ (ES-N)}$$

$$\frac{\begin{array}{c}\mathit{rg}(\kappa) = \mathit{rg}(\kappa_1) + \mathit{rg}(\kappa_2) \qquad \mathit{lk}(\kappa) = \mathit{lk}(\kappa_1) + \mathit{lk}(\kappa_2) \qquad \mathit{is\_pure}(\kappa) \Leftrightarrow \mathit{is\_pure}(\kappa_2) \\ \mathit{is\_pure}(\kappa_1) \Rightarrow \kappa = \kappa_1 \qquad \xi = \mathsf{par} \wedge \neg \mathit{is\_pure}(\kappa_1) \Rightarrow \mathit{lk}(\kappa_1) = 0\end{array}}{\xi \vdash \kappa = \kappa_1 + \kappa_2} \textit{ (CS)}$$

A.4 Type safety

Type safety rules

$$\mathit{pops}(\sigma : e) \quad = \quad \begin{cases} \mathit{pops}(\sigma : e_1) \wedge \mathit{pops}(\emptyset;\emptyset : e_2) & \text{if } e \equiv (e_1\ e_2)^{\xi} \wedge e_1 \neq v \\ \mathit{pops}(\sigma : e_2) \wedge \mathit{pops}(\emptyset;\emptyset : v) & \text{if } e \equiv (v\ e_2)^{\xi} \\ \mathit{pops}(\sigma : e_1) & \text{if } e \equiv (e_1)\,[v] \\ \mathit{pops}(\sigma : e_1) \wedge \mathit{pops}(\emptyset;\emptyset : e_2) & \text{if } e \equiv \mathtt{newrgn}^{r}\ \rho, x\ @\ e_1\ \mathtt{in}\ e_2 \\ \mathit{pops}(\sigma : e_1) & \text{if } e \equiv \mathtt{cap}^{r'}_{\eta}\ e_1 \\ \mathit{pops}(\sigma : e_1) \wedge \mathit{pops}(\emptyset;\emptyset : v) & \text{if } e \equiv \mathtt{new}\ v\ @\ e_1 \\ \mathit{pops}(\sigma : e_1) \wedge \mathit{pops}(\emptyset;\emptyset : e_2) & \text{if } e \equiv \mathtt{new}\ e_1\ @\ e_2 \wedge e_1 \neq v \\ \mathit{pops}(\sigma : e_1) & \text{if } e \equiv \mathtt{deref}\ e_1 \\ \mathit{pops}(\sigma : e_1) \wedge \mathit{pops}(\emptyset;\emptyset : e_2) & \text{if } e \equiv e_1 := e_2 \wedge e_1 \neq v \\ \mathit{pops}(\sigma : e_1) \wedge \mathit{pops}(\emptyset;\emptyset : v) & \text{if } e \equiv v := e_1 \\ \mathit{pops}(\sigma' : e_1) & \text{if } e \equiv \mathtt{pop}_{\gamma_r}\ e_1 \wedge \sigma = \emptyset;\gamma_r + \sigma' \\ \sigma \equiv \emptyset;\gamma \wedge \mathit{pops}(\emptyset;\emptyset : e') & \text{if } e \equiv \lambda x.\, e'\ \mathtt{as}\ \tau \\ \sigma \equiv \emptyset;\gamma \wedge \mathit{pops}(\emptyset;\emptyset : f) & \text{if } e \equiv \Lambda\rho.\, f \\ \sigma \equiv \emptyset;\gamma & \text{if } e \in \{\mathtt{loc}_{\ell}, \mathtt{rgn}_{\iota}, ()\} \end{cases}$$

$$\begin{array}{lll} \mathit{nolock}(\delta, n, e) & \equiv e = E[\mathtt{cap}^{r}_{+\mathsf{lk}}\ \mathtt{rgn}_{\mathtt{j}}] \wedge \exists \delta'', \pi, \kappa, \kappa'.\ \delta = \delta'', n \mapsto \sigma;\gamma, r^{\kappa} \triangleright \pi \wedge \\ & \quad \kappa' = [\![\mathsf{lk}+]\!]\,(\kappa) \wedge \neg \vdash \delta'', n \mapsto \sigma;\gamma, r^{\kappa'} \triangleright \pi \\ \mathit{redex}(e) & = (\exists\, \sigma, \sigma', S, S', e', n.\ \ \sigma; S; e \to \sigma'; S'; e') \vee (\exists v_1, v_2, \gamma_1. e = (v_1\ v_2)^{\mathsf{par}}) \end{array}$$

$$\frac{\bigcup_{(\iota \mapsto H) \in S} \{\ell \mid (\ell \mapsto v) \in H\} = \{\ell \mid (\ell \mapsto (\tau, \jmath)) \in M\}}{M \vdash S}$$

$$\frac{R = \{\iota \mid (\iota \mapsto H) \in S\}}{R \vdash S}$$

$$\frac{\begin{array}{c}\forall (n \mapsto \sigma) \in \delta. \forall \gamma \in \sigma. \forall (r^{\kappa} \triangleright \pi) \in \gamma. \\ \bar{r} \in R \wedge (\pi = r' \Rightarrow \bar{r'} \in R)\end{array}}{R \vdash \delta}$$

$$\frac{\begin{array}{c}M \vdash S \qquad R \vdash S \\ \forall (\ell \mapsto (\tau, \iota)) \in M. R; M; \emptyset; \emptyset \vdash S(\iota)(\ell) : \tau \,\&\, (\emptyset;\emptyset)\end{array}}{R; M \vdash S} \qquad \frac{\vdash \delta \qquad R; M \vdash S \qquad R \vdash \delta}{R; M \vdash \delta; S}$$

$$\frac{}{R; M; \emptyset \vdash \emptyset} \qquad \frac{\begin{array}{c}R; M; \emptyset; \emptyset \vdash e : \langle\rangle \,\&\, (\gamma;\emptyset) \\ R; M; \delta' \vdash T \qquad \forall (n' : e') \in T. n' \neq n \\ \delta = \delta', n \mapsto \sigma; \gamma \qquad \mathit{pops}(\sigma; \gamma : e)\end{array}}{R; M; \delta \vdash T, n : e} \qquad \frac{R; M; \delta \vdash T \qquad R; M \vdash \delta; S}{R; M \vdash \delta; S; T}$$

$$\frac{\forall (n : e) \in T.\ (\delta; S; T \rightsquigarrow \delta; S'; T' \wedge (n : e) \notin T') \vee \mathit{nolock}(\delta, n, e)}{\vdash \delta; S; T}$$

Multi-step evaluation rules

$$\frac{\begin{array}{c}n > 0 \qquad \delta; S; T \rightsquigarrow^{n-1} \delta_{n-1}; S_{n-1}; T_{n-1} \\ \delta_{n-1}; S_{n-1}; T_{n-1} \rightsquigarrow \delta_n; S_n; T_n\end{array}}{\delta; S; T \rightsquigarrow^{n} \delta_n; S_n; T_n} \textit{ (E-M1)} \qquad \frac{}{\delta; S; T \rightsquigarrow^{0} \delta; S; T} \textit{ (E-M2)}$$

A.5 Proof of soundness

Definition A.1 Type Safety Initial Environment

$$\begin{array}{lcl} R_0 & = & \{\iota_H\} \\ \delta_0 & = & \{1 \mapsto \iota_H^{1,0} \triangleright \bot\} \\ S_0 & = & \{\iota_H \mapsto \emptyset\} \\ T_0 & = & \{1 : (\mathtt{main}[\iota_H]\ \mathtt{rgn}_{\iota_H})^{\mathsf{seq}}\} \end{array}$$

Theorem A.1 (Type safety) Let R_0, δ_0, S_0 and T_0 be defined as in Definition A.1. If the operational semantics takes any number of steps $\delta_0; S_0; T_0 \rightsquigarrow^n \delta_n; S_n; T_n$, then the resulting configuration $\delta_n; S_n; T_n$ is not stuck.

Proof. The proof is trivial: Lemma A.1 is applied to the assumptions $\delta; S; T$ is well-typed and the operational semantics performs n steps, to obtain that $\delta_n; S_n; T_n$ is well-typed for some $R_n; M_n$. Then, lemma A.53 is applied to the latter fact to prove that $\delta_n; S_n; T_n$ is not stuck.

Lemma A.1 (Multi-step Program Preservation) Let $\delta; S; T$ be a *closed well-typed configuration* such that $R; M \vdash \delta; S; T$ for some R;M. If the operational semantics evaluates $\delta; S; T$ to $\delta'; S'; T'$ in n steps then there exists a *closed well-typed configuration* such that $R'; M' \vdash \delta'; S'; T'$, where R' and M' are supersets of R and M respectively.

Proof. Proof by induction on the number of steps n. When no steps are performed the proof is immediate from the assumption. If n steps are performed, we have that $\delta; S; T \rightsquigarrow^{n} \delta'; S'; T'$ or $\delta; S; T \rightsquigarrow^{n-1} \delta_{n-1}; S_{n-1}; T_{n-1}$ and $\delta_{n-1}; S_{n-1}; T_{n-1} \rightsquigarrow \delta'; S'; T'$. By applying the induction hypothesis on the fact that $\delta; S; T$ is well-typed and that $n-1$ steps are performed we obtain that there exist $R_{n-1}; M_{n-1}$ such that $R_{n-1}; M_{n-1} \vdash \delta_{n-1}; S_{n-1}; T_{n-1}$. We complete the proof by applying lemma A.2 on the latter fact and $\delta_{n-1}; S_{n-1}; T_{n-1} \rightsquigarrow \delta'; S'; T'$.

Lemma A.2 (Preservation — Program) Let $\delta; S; T$ be a well-typed configuration with $R; M \vdash \delta; S; T$. If the operational semantics takes a step $\delta; S; T \rightsquigarrow \delta'; S'; T'$, then there exist $R' \supseteq R$ and $M' \supseteq M$ such that the resulting configuration is well-typed with $R'; M' \vdash \delta'; S'; T'$.

Proof. By case analysis on the thread evaluation relation:

Case *E-T*: The premise of this rule are $T_1, n : () = T$ and $\delta_1, n \mapsto (\emptyset; \emptyset) = \delta$, for some δ_1 and T_1. By applying lemma A.3 to the configuration typing assumption we have that $R; M \vdash \delta_1, n \mapsto (\emptyset; \emptyset); S; T_1, n : ()$ holds.

By inversion of the latter configuration typing derivation we obtain the store $(R; M \vdash \delta_1, n \mapsto (\emptyset; \emptyset); S)$, and thread $(R; M; \delta_1, n \mapsto (\emptyset; \emptyset) \vdash T_1, n : ())$ typing derivations. By inversion of the thread typing derivation, we have that $R; M; \delta_1 \vdash T_1$ is well-typed. Lemma A.9 is applied to the store typing derivation $(R; M \vdash \delta_1, n \mapsto (\emptyset; \emptyset); S)$ to obtain that $R; M \vdash \delta_1; S$ holds. The new store and thread typing derivations give us $R; M \vdash \delta_1; S; T_1$.

Case *E-S* : The premises of this rule are $T_1, n : E[e] = T$, $\delta = \delta_1, n \mapsto \sigma, \vdash \delta'$, $\delta' = \delta_1, n \mapsto \sigma'$ and $\sigma; S; e \to \sigma'; S'; e'$. The resulting configuration is $\delta'; S'; T_1, n : E[e']$. By applying lemma A.3 to the latter configuration typing derivation, we have that $R; M \vdash \delta; S; T_1, n : E[e]$ holds. By inverting the *configuration typing* we obtain that $R; M; \delta \vdash T_1, n : E[e]$ and $R; M \vdash \delta; S$ holds. By inversion of the thread typing derivation we have that $R; M; \emptyset; \emptyset \vdash E[e] : \langle\rangle \& (\gamma; \emptyset)$, $\sigma = \sigma''; \gamma$, $pops(\sigma''; \gamma : E[e])$ holds, and $R; M; \delta_1 \vdash T_1$ holds. By applying lemma A.11 to the typing derivation of $E[e]$ we obtain that $R; M; \emptyset; \emptyset \vdash e : \tau, (\gamma; \gamma')$ for some γ' and τ. By applying lemma A.29 to $\vdash \delta[n \mapsto \sigma']$ (rule *E-S*), the typing derivation of e, the *expression evaluation* step $(\sigma; S; e \to \sigma'; S'; e')$ and the store typing derivation $(R; M \vdash \delta; S)$, we obtain that e' is also well-typed $(R'; M'; \emptyset; \emptyset \vdash e' : \tau \& (\gamma''; \gamma'))$, where γ'' is the top stack frame for thread n ($\sigma' = \sigma'''; \gamma''$ for some σ'''), for some $R \subseteq R'$, $M \subseteq M'$, and the resulting store $\delta[n \mapsto \sigma']; S'$ is also well-typed $(R'; M' \vdash \delta[n \mapsto \sigma']; S')$. By applying lemma A.12 to the typing derivation of e' we have that $\vdash R'; M'; \emptyset; \emptyset; \gamma''; \gamma'$. By inversion of the latter derivation we have that $R' \vdash M'$. By applying lemma A.4 to $R; M; \delta \vdash T_1, n : E[e]$, $R \subseteq R'$, $M \subseteq M'$ and $R' \vdash M'$, we have that $R'; M'; \delta \vdash T_1, n : E[e]$ holds. By inversion of the latter derivation we have that $R'; M'; \emptyset; \emptyset \vdash E[e] : \langle\rangle \& (\gamma; \emptyset)$. By lemma A.23 we can substitute e' for e in the evaluation context E (all well-typed in $R'; M'$) to obtain $R'; M'; \emptyset; \emptyset \vdash E[e'] : \langle\rangle \& (\gamma''; \emptyset)$.

The application of lemma A.6 to $pops((\sigma''; \gamma) : E[e])$ implies that $\exists \sigma_1, \sigma_2 . \sigma''; \gamma = \sigma_1 + \sigma_2 \wedge pops(\sigma_2 : e))$. The application of lemma A.7 to $pops((\sigma''; \gamma) : E[e])$, $(\sigma''; \gamma); S; e \to \sigma'; S'; e'$, $\sigma''; \gamma = \sigma_1 + \sigma_2$, and $pops(\sigma_2 : e)$, gives us that $pops(\sigma_1 + \sigma_2' : E[e'])$, where $\sigma' = \sigma_1 + \sigma_2'$.

By inversion of $R'; M'; \delta \vdash T_1, n : E[e]$ we have that $R'; M'; \delta_1 \vdash T_1$ and $\forall (n' : e') \in T_1 . n' \neq n$. We can reconstruct a similar derivation by using the latter derivations along with $pops(\sigma_1 + \sigma_2' : E[e'])$, $\delta' = \delta_1, n \mapsto \sigma_1 + \sigma_2'$ and $R'; M'; \emptyset; \emptyset \vdash E[e'] : \langle\rangle \& (\gamma''; \emptyset)$: $R'; M'; \delta' \vdash T_1, n : E[e']$. Both $R'; M' \vdash \delta'; S'$ and $R'; M'; \delta' \vdash T_1, n : E[e']$ imply that $R'; M' \vdash \delta'; S'; T_1, n : E[e']$ holds.

Case *E-SN*: The *program evaluation* assumption gives us that $e \equiv E[e']$, such that e' is a parallel application redex, and its premise asserts that e' is moved to a new thread as a local application redex e''. It also gives that $T_1, n : E[e'] = T$ and that n' is *fresh*. The resulting store map δ' is equal to $\delta'', n \mapsto \sigma; \gamma', n' \mapsto \gamma_1$, where δ equals $\delta'', n \mapsto \sigma; \gamma$ and $\mathsf{par} \vdash \gamma' = \emptyset \oplus (\gamma \ominus \gamma_1)$ holds. By applying lemma A.3 to the configuration typing derivation, $R; M \vdash \delta; S; T, T_1, n : E[e'] = T$ and $\delta = \delta'', n \mapsto \sigma; \gamma$, we have that $R; M \vdash \delta'', n \mapsto \sigma; \gamma; S; T_1, n : E[e']$ holds.

We need to prove that $R; M \vdash \delta'; S; T_1, n : E[()], n' : e''$ holds. It suffices to prove that $R; M; \delta' \vdash T_1, n : E[()], n' : e''$ and $R; M \vdash \delta'; S$.

Thread typing: The following obligations must be proved:

- $R; M; \emptyset; \emptyset \vdash E[()] : \langle\rangle \,\&\, (\gamma';\emptyset)$: we must prove that $R; M; \delta'', n \mapsto \sigma; \gamma \vdash T_1, n : E[e']$. By inversion of this obligation it suffices to prove that $R; M; \emptyset; \emptyset \vdash E[e'] : \langle\rangle \& (\gamma; \emptyset)$ holds. By applying lemma A.11 to the typing derivation of $E[e']$, we obtain that e' is well-typed in the context $R; M; \emptyset; \emptyset$ with effect $(\gamma; \gamma'')$ for some γ''. The application of lemma A.10 to the latter derivation implies that $\mathsf{par} \vdash \gamma'' = \emptyset \oplus (\gamma \ominus \gamma_1')$, where $\gamma_1' \simeq \gamma_1$. The application of lemma A.33 implies that $\mathsf{par} \vdash \gamma'' = \emptyset \oplus (\gamma \ominus \gamma_1)$ holds. The capability addition derivation rule is deterministic thus, that γ'' is equal to γ'. Thus, e' is well-typed with effect $(\gamma; \gamma')$.

  By applying lemma A.12 to the typing derivation of e', we have that $\vdash R; M; \emptyset; \emptyset; \gamma; \gamma'$. Thus, $\vdash R; M; \emptyset; \emptyset; \gamma'; \gamma'$ also holds (trivial). Consequently, by rule *T-U* we have that $R; M; \emptyset; \emptyset \vdash () : \langle\rangle \& (\gamma'; \gamma')$ holds. Now we can substitute the well-typed unit value described above for e' in the evaluation context E, by using lemma A.23, to obtain that $E[()]$ is well-typed in the typing context $R; M; \emptyset; \emptyset$ with effect $(\gamma'; \emptyset)$.
- $\forall (n'' : e''') \in T_1 . n'' \neq n$: immediate from the thread typing assumption (premise), which can be obtained by inversion of the original configuration typing derivation.
- $\delta = \delta'', n \mapsto \sigma; \gamma'$: immediate.
- $\mathit{pops}(\sigma; \gamma' : E[()])$: this is immediate by the application of lemma A.8 to the fact that $\mathit{pops}(\sigma; \gamma : E[e'])$.
- $R; M; \emptyset; \emptyset \vdash e'' : \langle\rangle \,\&\, (\gamma_1;\emptyset)$: the application of lemma A.24 to the fact that e' is well-typed in the context $R; M; \emptyset; \emptyset$ with effect $(\gamma; \gamma')$, yields that e'' is well-typed in the context $R; M; \emptyset; \emptyset$ with effect $(\gamma_1; \emptyset)$.
- $\forall (n'' : e''') \in T_1 . n'' \neq n'$ and $n' \neq n$: immediate from the fact that n' is *fresh*.
- $\delta = \delta'', n \mapsto \sigma; \gamma', n' \mapsto \emptyset; \gamma_1$: immediate.
- $\mathit{pops}(\emptyset; \gamma_1 : e'')$: the assumption that $\mathit{pops}(\sigma; \gamma : E[e'])$ and lemma A.7 imply that $\sigma; \gamma = \sigma_a + \sigma_b$ and $\mathit{pops}(\sigma_b : e')$. Expression e' comprises of values thus by the definition of pops we have that $\sigma_b = \emptyset; \gamma$. The definition of pops also allows us to derive $\mathit{pops}(\emptyset; \gamma_1 : e')$. Thus, $\mathit{pops}(\emptyset; \gamma_1 : e'')$ also holds.

Store typing: by applying lemma A.12 to the typing derivation (as shown earlier) of e' implies that $\mathit{ok}(\gamma; \gamma')$ holds. The proof is immediate by the application of lemma A.28 to the fact that $\mathit{ok}(\gamma; \gamma')$ holds, $R; M \vdash \delta; S$ holds, δ' is equal to $\delta'', n \mapsto \sigma; \gamma', n' \mapsto \gamma_1$, $\mathit{live}(\gamma_1) = \gamma_1$ (by inversion of the function type well-formedness derivation, which is a premise of the function typing derivation) and $\mathsf{par} \vdash \gamma' = \emptyset \oplus (\gamma \ominus \gamma_1)$.

Lemma A.3 (Reordering) $R; M \vdash \delta; S; T \wedge T' = T \wedge \delta' = \delta \Rightarrow R; M \vdash \delta'; S; T'$

Proof. Trivial.

Lemma A.4 (Thread Weakening) $R; M; \delta \vdash T \wedge R \subseteq R' \wedge M \subseteq M' \wedge R' \vdash M' \Rightarrow R'; M'; \delta \vdash T$

Proof. Proof by induction on the shape of T.

- $\emptyset$: $R'; M'; \delta \vdash \emptyset$ trivially holds.
- $T', n : e$: By inversion of this derivation we have that
  - $R; M; \emptyset; \emptyset \vdash e : \langle\rangle \,\&\, (\gamma;\emptyset)$: The application of lemma A.21 to $R \subseteq R'$ and the typing derivation of e gives us $R'; M; \emptyset; \emptyset \vdash e : \langle\rangle \,\&\, (\gamma;\emptyset)$. The application of lemma A.22 to the latter derivation, $M \subseteq M'$ and $R' \vdash M'$ gives us $R'; M'; \emptyset; \emptyset \vdash e : \langle\rangle \,\&\, (\gamma;\emptyset)$.
  - $R; M; \delta' \vdash T'$: By the induction hypothesis $R'; M'; \delta' \vdash T'$ holds.
  - $\forall (n' : e') \in T'.n' \neq n$
  - $\delta = \delta', n \mapsto \sigma; \gamma$
  - $\mathit{pops}(\sigma; \gamma : e)$

  We can use the above facts to derive $R'; M'; \delta \vdash T', n : e$ holds.

Lemma A.5 (pops expression preservation) $\sigma_0; S; e \to \sigma_0'; S'; e' \wedge \mathit{pops}(\sigma_2 : e) \wedge \sigma_0 = \sigma_1 + \sigma_2 \Rightarrow \exists \sigma_2'. \wedge \sigma_0' = \sigma_1 + \sigma_2' \wedge \mathit{pops}(\sigma_2' : e')$

Proof. Proof by case analysis on the operational rules.

Case *E-C*: the premises of this rule tell us that $\sigma_0 = \sigma; \gamma$ and $\sigma_0' = \sigma; \mathit{live}(\gamma', r^{\kappa'} \triangleright \pi)$. Thus, $\sigma; \gamma = \sigma + \emptyset; \gamma$ and $\sigma_0' = \sigma + \emptyset; \mathit{live}(\gamma', r^{\kappa'} \triangleright \pi)$. By the definition of predicate pops we can derive $\mathit{pops}(\emptyset; \mathit{live}(\gamma', r^{\kappa'} \triangleright \pi) : ())$.

Case *E-A*: the premises of this rule tell us that $\sigma_0 = \sigma; \gamma$ and $\sigma_0' = \sigma; \gamma_r; \gamma_1$. Thus, $\sigma; \gamma = \sigma + \emptyset; \gamma$ and $\sigma_0' = \sigma + \emptyset; \gamma_r; \gamma_1$. By inversion of the assumption that $\mathit{pops}(\sigma : (\lambda x.\, e\ \mathtt{as}\ \tau\ v)^{\mathsf{seq}})$ holds we have that $\mathit{pops}(\emptyset; \emptyset : e)$. It is trivial to show that $\mathit{pops}(\emptyset; \gamma_1 : e)$ also holds by induction on predicate pops. We combine the latter fact with $\emptyset; \gamma_r; \gamma_1 = \emptyset; \gamma_r + \emptyset; \gamma_1$ to derive $\mathit{pops}(\emptyset; \gamma_r; \gamma_1 : \mathtt{pop}_{\gamma_r}\ e)$.

Case *E-NG*: the premises of this rule tell us that $\sigma_0 = \sigma; \gamma$ and $\sigma_0' = \sigma; \gamma, r^{1,1} \triangleright r'$. Thus, $\sigma; \gamma = \sigma + \emptyset; \gamma$ and $\sigma_0' = \sigma + \emptyset; \gamma, r^{1,1} \triangleright r'$. By inversion of the assumption that $\mathit{pops}(\sigma : \mathtt{newrgn}^{r'}\ \rho, x\ @\ \mathtt{rgn}_{r'}\ \mathtt{in}\ e)$ holds we have that $\mathit{pops}(\emptyset; \emptyset : e)$. It is trivial to show that $\mathit{pops}(\emptyset; \gamma, r^{1,1} \triangleright r' : e)$ also holds by induction on predicate pops. Thus, the proof is completed if σ_2 is equal to $\emptyset; \gamma, r^{1,1} \triangleright r'$.

Case *E-E*: the premises of this rule tell us that $\sigma_0 = \sigma; \gamma_r; \gamma'$ and $\sigma_0' = \sigma; \gamma''$. Thus, $\sigma; \gamma = \sigma + \emptyset; \gamma_r; \gamma'$ and $\sigma_0' = \sigma + \emptyset; \gamma''$. By inversion of the assumption $\mathit{pops}(\emptyset; \gamma_r; \gamma' : \mathtt{pop}_{\gamma_r}\ v)$ we have that $\mathit{pops}(\emptyset; \gamma' : v)$ holds. By the definition of predicate pops we can rewrite the latter fact as $\mathit{pops}(\emptyset; \gamma'' : v)$. Thus, the proof is completed if σ_2 is equal to $\emptyset; \gamma''$.

Case *E-RP*: this rule implies that $\sigma_0 = \sigma_0' = \sigma$. The assumption that $\mathit{pops}(\sigma_2 : (f)\,[r])$ holds tells us that $\sigma_2 = \emptyset; \gamma$ for some γ. Thus, $\sigma = \sigma_1 + \emptyset; \gamma$. $\sigma_0' = \sigma = \sigma_1 + \emptyset; \gamma$. Thus $\sigma_2' = \emptyset; \gamma$. The assumption that $\mathit{pops}(\sigma_2 : (f)\,[r])$ also tells us that $\mathit{pops}(\emptyset; \emptyset : f)$ holds. By the definition of pops, $\mathit{pops}(\emptyset; \gamma : f)$ also holds. Thus, $\mathit{pops}(\sigma_2' : f)$ holds.

Case *E-D*,*E-NR*,*E-AS*: similar to the previous case.

Lemma A.6 (pops implication) $\mathit{pops}(\sigma : E[e]) \Rightarrow \exists \sigma_1, \sigma_2.\sigma = \sigma_1 + \sigma_2 \wedge \mathit{pops}(\sigma_2 : e))$

Proof. We perform induction on the shape of E:

Case $\Box[e]$: Let σ_1 and σ_2 be equal to $\emptyset$ and σ respectively. By the latter facts and the assumption $\mathit{pops}(: E[e])$

we have that $\mathit{pops}(\sigma : e)$ (assumption) holds.

Case $((E'\ e_2)^{\xi})[e]$: this is equivalent to $(E'[e]\ e_2)^{\xi}$. By inversion of the assumption we have that $\mathit{pops}(\sigma : E'[e])$ holds ($E'[e]$ is not value; this is dealt with in the next case). By the induction hypothesis there exists σ_1 and σ_2 such that $\sigma = \sigma_1 + \sigma_2$ and $\mathit{pops}(\sigma_2 : e)$.

Case $((v_1\ E')^{\xi})[e]$: this is equivalent to $(v_1\ E'[e])^{\xi}$. By inversion of the assumption we have that $\mathit{pops}(\sigma : E'[e])$ holds By the induction hypothesis there exists σ_1 and σ_2 such that $\sigma = \sigma_1 + \sigma_2$ and $\mathit{pops}(\sigma_2 : e)$.

Case $(\mathtt{pop}_{\gamma}\ E'))[e]$,$(E'\ [r])[e]$: this is equivalent to $\mathtt{pop}_{\gamma_r}\ E'[e]$. By inversion of the assumption we have that $\sigma = 0; \gamma_r + \sigma'$ and $\mathit{pops}(\sigma' : E'[e])$. By applying the induction hypothesis we have that $\sigma' = \sigma'_1 + \sigma_2$ and $\mathit{pops}(\sigma_2 : e)$. Thus, the proof is completed if σ_1 equals $\emptyset; \gamma_r + \sigma'_1$.

Case $(\mathtt{cap}^r_{\eta}\ E')[e]$,$(\mathtt{deref}\ E')[e]$,$(E' := e_2)[e]$, $(\mathtt{loc}_{\ell} := E')[e]$, $(\mathtt{new}\ E'\ @\ e_2))[e]$, $(\mathtt{new}\ v\ @\ E'))[e]$, $(\mathtt{newrgn}^r\ \rho, x\ @\ E'\ \mathtt{in}\ e_2)[e]$: Similar to the above proof structure.

Lemma A.7 (*pops* evaluation context preservation) $\mathit{pops}(\sigma_1 + \sigma_2 : E[e]) \wedge \mathit{pops}(\sigma_2 : e) \wedge \sigma_1 + \sigma_2; S; e \rightarrow \sigma'; S'; e' \Rightarrow \exists \sigma_3 . \sigma' = \sigma_1 + \sigma_3 \wedge \mathit{pops}(\sigma' : E[e'])$.

Proof. We proceed by induction on the structure of E:

Case $\Box[e]$: the application of lemma A.5 to $\mathit{pops}(\sigma_2 : e)$ and $\sigma_1 + \sigma_2; S; e \rightarrow \sigma'; S'; e'$ implies that there exists an σ_3 such that $\sigma' = \sigma_1 + \sigma_3$ and $\mathit{pops}(\sigma_3 : e')$ holds. The assumption implies that $\mathit{pops}(\sigma_1 + \sigma_2 : \Box[e])$ and $\mathit{pops}(\sigma_2 : e)$. This can only hold if $\sigma_1 = \emptyset$. We have shown that $\mathit{pops}(\sigma_3 : e')$ holds. Thus, $\mathit{pops}(\sigma_1 + \sigma_3 : \Box[e'])$ holds.

Case $((E'\ e_2)^{\xi})[e]$: By the definition of the evaluation context and the assumption that $\mathit{pops}(\sigma_1 + \sigma_2 : E[e])$ holds, we have that $\mathit{pops}(\sigma_1 + \sigma_2 : (E'[e]\ e_2)^{\xi})$ holds. $E'[e]$ is not a value as e is a redex (operational step assumption). Therefore, by inversion of $\mathit{pops}(\sigma_1 + \sigma_2 : (E'[e]\ e_2)^{\xi})$ and the latter fact we obtain that $\mathit{pops}(\sigma_1 + \sigma_2 : E'[e])$ and $\mathit{pops}(\emptyset; \emptyset : e_2)$. By applying the induction hypothesis we have that there exists an σ'_3 such that $\mathit{pops}(\sigma' : E'[e'])$ and $\sigma' = \sigma_1 + \sigma'_3$ holds. Therefore, we can combine $\mathit{pops}(\sigma' : E'[e'])$ and $\mathit{pops}(\emptyset; \emptyset : e_2)$ to derive $\mathit{pops}(\sigma' : (E'[e]\ e_2)^{\xi})$.

Case $((v_1\ E')^{\xi})[e]$: By the definition of the evaluation context and the assumption that $\mathit{pops}(\sigma_1 + \sigma_2 : E[e])$ holds, we have that $\mathit{pops}(\sigma_1 + \sigma_2 : (v_1\ E'[e])^{\xi})$ holds. By inversion of $\mathit{pops}(\sigma_1 + \sigma_2 : (v_1\ E'[e])^{\xi})$ and the latter fact we obtain that $\mathit{pops}(\sigma_1 + \sigma_2 : E'[e])$ and $\mathit{pops}(\emptyset; \emptyset : v_1)$. By applying the induction hypothesis we have that there exists an σ'_3 such that $\mathit{pops}(\sigma' : E'[e'])$ and $\sigma' = \sigma_1 + \sigma'_3$ holds. Therefore, we can combine $\mathit{pops}(\sigma' : E'[e'])$ and $\mathit{pops}(\emptyset; \emptyset : v_1)$ to derive $\mathit{pops}(\sigma' : (v_1\ E'[e'])^{\xi})$.

Case $(\mathtt{pop}_{\gamma}\ E'))[e]$: By the definition of the evaluation context and the assumption that $\mathit{pops}(\sigma_1 + \sigma_2 : E[e])$ holds, we have that $\mathit{pops}(\sigma_1 + \sigma_2 : \mathtt{pop}_{\gamma_r}\ E'[e])$ holds. By inversion of the latter fact we obtain that $\sigma_1 + \sigma_2 = \emptyset; \gamma_r + \sigma_x$ and $\mathit{pops}(\sigma_x; E'[e])$. σ_2 is a postfix of σ_x as $\mathit{pops}(\sigma_2 : e)$ would not hold otherwise (definition of *pops* predicate). Thus, there exists an σ'_x such that $\sigma_x = \sigma'_x + \sigma_2$ and $\sigma_1 = 0; \gamma_r + \sigma'_x$. By applying the induction hypothesis we obtain that $\mathit{pops}(\sigma'_x + \sigma_3 : E'[e'])$ for some σ_3. We can combine the latter fact with $\sigma_1 = 0; \gamma_r + \sigma'_x$ to derive $\mathit{pops}(\sigma_1 + \sigma_3 : E'[e'])$.

Case $(E'\ [r])[e]$,$(\mathtt{cap}^r_{\eta}\ E')[e]$,$(\mathtt{deref}\ E')[e]$,$(E' := e_2)[e]$, $(\mathtt{loc}_{\ell} := E')[e]$, $(\mathtt{new}\ E'\ @\ e_2))[e]$, $(\mathtt{new}\ v\ @\ E'))[e]$, $(\mathtt{newrgn}^r\ \rho, x\ @\ E'\ \mathtt{in}\ e_2)[e]$: Similar to the above proof structure.

Lemma A.8 (*pops* — Replace value) $\mathit{pops}(\sigma;\gamma : E[(v\ v')^{\xi}]) \Rightarrow \forall\gamma'.\mathit{pops}(\sigma;\gamma' : E[()])$

Proof. Proof by induction on the shape of E:

Case $\Box$: this is immediate by the definition of *pops* for the unit value.

Case $(E'\ e_2)^{\xi'}$: By the definition of the evaluation context and the assumption that $\mathit{pops}(\sigma;\gamma : E[(v\ v')^{\xi'}])$ holds, $\mathit{pops}(\sigma;\gamma : (E'[(v\ v')^{\xi}]\ e_2)^{\xi'})$ also holds. By inversion of the latter judgement and the fact that the hole does not contain a value we obtain that $\mathit{pops}(\sigma;\gamma : E'[(v\ v')^{\xi}])$ and $\mathit{pops}(\emptyset;\emptyset : e_2)$. By applying the induction hypothesis we obtain that $\forall\gamma'.\mathit{pops}(\sigma;\gamma' : E'[()])$. We can use the latter fact and $\mathit{pops}(\emptyset;\emptyset : e_2)$ to derive $\forall\gamma'.\mathit{pops}(\sigma;\gamma' : (E'[()]\ e_2)^{\xi'})$.

Case $(v_1\ E')^{\xi'}$: By the definition of the evaluation context and the assumption that $\mathit{pops}(\sigma;\gamma : E[(v\ v')^{\xi'}])$ holds, $\mathit{pops}(\sigma;\gamma : (v_1\ E'[(v\ v')^{\xi}])^{\xi'})$ also holds. By inversion of the latter judgement we obtain that $\mathit{pops}(\sigma;\gamma : E'[(v\ v')^{\xi}])$ and $\mathit{pops}(\emptyset;\emptyset : v_1)$. By applying the induction hypothesis we obtain that $\forall\gamma'.\mathit{pops}(\sigma;\gamma' : E'[()])$. We can use the latter fact and $\mathit{pops}(\emptyset;\emptyset : v_1)$ to derive $\forall\gamma'.\mathit{pops}(\sigma;\gamma' : (v_1\ E'[()])^{\xi'})$.

Case $(\mathtt{pop}_{\gamma}\ E'))[e]$: By the definition of the evaluation context and the assumption that $\mathit{pops}(\sigma;\gamma : E[(v\ v')^{\xi'}])$ holds, $\mathit{pops}(\sigma;\gamma : \mathtt{pop}_{\gamma_r}\ E'[(v\ v')^{\xi}])$ also holds. By inversion of the latter judgement we obtain that $\sigma;\gamma = \emptyset;\gamma_r + \sigma'$ and $\mathit{pops}(\sigma' : E'[(v\ v')^{\xi}])$. $\sigma;\gamma = \emptyset;\gamma_r + \sigma'$ implies that there exists a σ'' such that $\sigma' = \sigma'';\gamma$. Thus, $\mathit{pops}(\sigma' : E'[(v\ v')^{\xi}])$ becomes $\mathit{pops}(\sigma'';\gamma : E'[(v\ v')^{\xi}])$. The application of the induction hypothesis to the latter fact gives us that $\forall\gamma'.\mathit{pops}(\sigma'';\gamma' : E'[(v\ v')^{\xi}])$. Thus, we can derive from the above facts that $\mathit{pops}(\sigma;\gamma' : \mathtt{pop}_{\gamma_r}\ E'[()])$ holds.

Case $(E'\ [r])[e]$,$(\mathtt{cap}^{r}_{\eta}\ E')[e]$,$(\mathtt{deref}\ E')[e]$,$(E' := e_2)[e]$, $(\mathtt{loc}_{\ell} := E')[e]$, $(\mathtt{new}\ E'\ @\ e_2))[e]$, $(\mathtt{new}\ v\ @\ E'))[e]$, $(\mathtt{newrgn}^{r}\ \rho, x\ @\ E'\ \mathtt{in}\ e_2)[e]$: Similar to the above proof structure.

Lemma A.9 (Store Strengthening — Empty γ) If store $\delta, n \mapsto (\emptyset;\emptyset); S$ is well-typed in the context $R; M$, then $\delta; S$ is also well-typed in the same context.

Proof. By inversion of the store typing assumption we have that

- $R; M \vdash S$
- $R \vdash \delta, n \mapsto (\emptyset;\emptyset)$
- $\vdash \delta, n \mapsto (\emptyset;\emptyset)$

$R \vdash \delta$ trivially holds by observing the premise of $R \vdash \delta, n \mapsto (\emptyset;\emptyset)$. By inversion of $\vdash \delta, n \mapsto (\emptyset;\emptyset)$, we obtain that $\vdash \delta$ holds.

Therefore, the latter facts and $R; M \vdash S$ imply that $R; M \vdash \delta; S$ also holds.

Lemma A.10 (Inversion)
$R; M; \Delta; \Gamma \vdash x : \tau' \,\&\, (\gamma_a;\gamma_b) \Rightarrow \gamma_a = \gamma_b = \gamma \wedge \vdash R; M; \Delta; \Gamma; \gamma; \gamma\ \wedge\ (x : \tau) \in \Gamma\ \wedge\ \tau \simeq \tau'$
$\wedge$
$R; M; \Delta; \Gamma \vdash \mathtt{c} : \tau \,\&\, (\gamma_a;\gamma_b) \Rightarrow \gamma_a = \gamma_b = \gamma \wedge \vdash R; M; \Delta; \Gamma; \gamma; \gamma\ \wedge\ \tau = b$
$\wedge$
$R; M; \Delta; \Gamma \vdash () : \tau \,\&\, (\gamma_a;\gamma_b) \Rightarrow \gamma_a = \gamma_b = \gamma \wedge \vdash R; M; \Delta; \Gamma; \gamma; \gamma\ \wedge\ \tau = \langle\rangle$
$\wedge$

$R; M; \Delta; \Gamma \vdash \mathtt{rgn}_\iota : \tau \,\&\, (\gamma_a;\gamma_b) \Rightarrow \gamma_a = \gamma_b = \gamma \wedge \vdash R; M; \Delta; \Gamma; \gamma; \gamma \;\wedge\; \tau \simeq \mathtt{rgn}(\iota) \;\wedge\; R; \Delta \vdash \iota$
$\wedge$
$R; M; \Delta; \Gamma \vdash \mathtt{loc}_l : \tau \,\&\, (\gamma_a;\gamma_b) \Rightarrow \gamma_a = \gamma_b = \gamma \wedge \vdash R; M; \Delta; \Gamma; \gamma; \gamma \;\wedge\; \tau \simeq \mathtt{ref}\; M(\ell)$
$\wedge$
$R; M; \Delta; \Gamma \vdash \mathtt{cap}^r_\eta\; e_1 : \tau \,\&\, (\gamma;\gamma'') \Rightarrow \tau = \langle\rangle \;\wedge\; R; M; \Delta; \Gamma \vdash e_1 : \mathtt{rgn}(r) \,\&\, (\gamma;\gamma', r^\kappa \triangleright \pi) \;\wedge\; \kappa' = [\![\eta]\!]\,(\kappa) \;\wedge\; \gamma'' = \mathit{live}(\gamma', r^{\kappa'} \triangleright \pi) \;\wedge\; \mathit{is\_live}(\gamma', r^\kappa \triangleright \pi)$
$\wedge$
$R; M; \Delta; \Gamma \vdash \lambda x.\, e \;\mathtt{as}\; \tau : \tau' \,\&\, (\gamma_a;\gamma_b) \Rightarrow \gamma_a = \gamma_b = \gamma \wedge \vdash R; M; \Delta; \Gamma; \gamma; \gamma \;\wedge\; R; \Delta \vdash \tau \;\wedge\; \tau \equiv \tau_1 \xrightarrow{\gamma_1 \to \gamma_2} \tau_2 \wedge \gamma' = \gamma_1 \wedge \mathit{set}(\gamma_1;\gamma_2) \wedge (\mathit{ok}(\gamma_1;\gamma_2) \Rightarrow R; M; \Delta; \Gamma, x : \tau_1 \vdash e : \tau_2 \,\&\, (\gamma_1;\gamma_2)) \wedge \tau' \simeq \tau$
$\wedge$
$R; M; \Delta; \Gamma \vdash \Lambda\rho.\, f : \tau \,\&\, (\gamma_a;\gamma_b) \Rightarrow \gamma_a = \gamma_b = \gamma \;\wedge\; \tau = \forall\rho.\, \tau' \;\wedge\; R; M; \Delta, \rho; \Gamma \vdash f : \tau' \,\&\, (\gamma;\gamma)$
$\wedge$
$R; M; \Delta; \Gamma \vdash e\;[r] : \tau \,\&\, (\gamma;\gamma') \Rightarrow \tau = \tau'[r/\rho] \;\wedge\; R; \Delta \vdash r \;\wedge\; R; M; \Delta; \Gamma \vdash e : \forall\rho.\, \tau' \,\&\, (\gamma;\gamma')$
$\wedge$
$R; M; \Delta; \Gamma \vdash (e_1\; e_2)^\xi : \tau_2 \,\&\, (\gamma;\gamma_5) \Rightarrow R; M; \Delta; \Gamma \vdash e_1 : \tau_1 \xrightarrow{\gamma_1 \to \gamma_2} \tau_2 \,\&\, (\gamma;\gamma_3) \;\wedge\; \mathsf{par} \Rightarrow \tau_2 = \langle\rangle \;\wedge\; R; M; \Delta; \Gamma \vdash e_2 : \tau_1 \,\&\, (\gamma_3;\gamma_4) \;\wedge\; \xi \vdash \gamma_5 = \gamma_2 \oplus (\gamma_4 \ominus \gamma_1)$
$\wedge$
$R; M; \Delta; \Gamma \vdash \mathtt{newrgn}^{r'}\; \rho, x \;@\; e_1 \;\mathtt{in}\; e_2 : \tau \,\&\, (\gamma;\gamma'') \Rightarrow R; M; \Delta; \Gamma \vdash e_1 : \mathtt{rgn}(r) \,\&\, (\gamma;\gamma') \;\wedge\; r' \equiv r \wedge \mathit{is\_live}(\gamma', r) \wedge R; \Delta \vdash \tau \wedge R; M; \Delta, \rho; \Gamma, x : \mathtt{rgn}(\rho) \vdash e_2 : \tau \,\&\, (\gamma', \rho^{1,1} \triangleright r; \gamma'') \wedge \rho \notin \mathit{dom}(\gamma'')$
$\wedge$
$R; M; \Delta; \Gamma \vdash \mathtt{new}\; e_1 \;@\; e_2 : \tau \,\&\, (\gamma;\gamma'') \Rightarrow \tau = \mathtt{ref}(\tau', r) \;\wedge\; R; M; \Delta; \Gamma \vdash e_1 : \tau' \,\&\, (\gamma;\gamma') \;\wedge\; \mathit{is\_live}(\gamma'', r) \;\wedge\; R; M; \Delta; \Gamma \vdash e_2 : \mathtt{rgn}(r) \,\&\, (\gamma';\gamma'')$
$\wedge$
$R; M; \Delta; \Gamma \;\vdash\; e_1 := e_2 \;:\; \tau' \,\&\, (\gamma;\gamma_2) \;\Rightarrow\; \tau' \;=\; \langle\rangle \;\wedge\; R; M; \Delta; \Gamma \;\vdash\; e_1 \;:\; \mathtt{ref}(\tau, r) \,\&\, (\gamma;\gamma') \;\wedge\; R; M; \Delta; \Gamma \vdash e_2 : \tau \,\&\, (\gamma';\gamma'') \;\wedge\; \mathit{is\_accessible}(\gamma'', r)$
$\wedge$
$R; M; \Delta; \Gamma \vdash \mathtt{deref}\; e : \tau \,\&\, (\gamma;\gamma') \Rightarrow R; M; \Delta; \Gamma \vdash e : \mathtt{ref}(\tau, r) \,\&\, (\gamma;\gamma') \;\wedge\; \mathit{is\_accessible}(\gamma', r)$
$\wedge$
$R; M; \Delta; \Gamma \vdash \mathtt{pop}_{\gamma_r}\; e : \tau \,\&\, (\gamma;\gamma') \Rightarrow \mathit{ok}(\gamma_r;\emptyset) \;\wedge\; R; M; \Delta; \Gamma \vdash e : \tau \,\&\, (\gamma_1;\gamma_2) \;\wedge\; \mathsf{seq} \vdash \gamma' = \gamma_2 \oplus (\gamma_r \ominus \emptyset) \;\wedge\; R; \Delta \vdash \gamma_r \;\wedge\; \tau \simeq \tau' \;\wedge\; \mathit{set}(\gamma_r;\emptyset)$

Proof. Straightforward pattern matching on the typing derivations.

Lemma A.11 (Context Inversion) If $E[e]$ is a well-typed expression in the typing context $R; M; \Delta; \Gamma$ with effect $(\gamma_1;\gamma_2)$, then e is also a well-typed expression for some type τ, in the same typing context with effect $(\gamma_1;\gamma_3)$ for some γ_3.

Proof. By straightforward induction on the shape of the evaluation context. The

Case $\Box[e]$ then proof is immediate.

Case $((E'\; e_2)^\xi)[e]$: An equivalent expression for this case is $(E'[e]\; e_2)^\xi$. By the assumption, $(E'[e]\; e_2)^\xi$ is a well-typed application term. Lemma A.10 implies that $E[e]$ is well-typed in the same typing context with effect $(\gamma_1;\gamma')$, where γ' is its output effect. The application of the induction hypothesis to the the latter typing derivation yields that e is a well-typed term in the same typing context with effect $(\gamma_1;\gamma'')$ for some γ''.

Case $((v_1\; E')^\xi)[e]$: An equivalent expression for this case is $(v_1\; E'[e])^\xi$. Lemma A.10 implies that $(v_1\; E'[e])^\xi$, $E'[e]$ and v_1 are well-typed. In addition, v_1 is a value with effect $(\gamma_1;\gamma_1)$ (this is immediate by performing a case analysis on v and applying lemma A.10). Thus, the input effect of $E'[e]$ is γ_1. The application of the induction hypothesis to the latter fact implies that e is well-typed for some type τ with effect $(\gamma_1;\gamma_3)$, for some γ_3.

Case $(\mathtt{cap}^r_\eta\ E')[e]$,$(\mathtt{deref}\ E')[e]$, $(E' := e_2)[e]$, $(\mathtt{loc}_\ell := E')[e]$, $(\mathtt{new}\ E' \mathbin{@} e_2))[e]$, $(\mathtt{new}\ v \mathbin{@} E'))[e]$, $(\mathtt{pop}_\gamma\ E'))[e]$,$(E'\ [r])[e]$, $(\mathtt{newrgn}^r\ \rho, x \mathbin{@} E'\ \mathtt{in}\ e_2)[e]$:
Similar to the above proof structure.

Lemma A.12 (Well-Formedness) If an expression e is well-typed in the typing context $R; M; \Delta; \Gamma$, with effect $(\gamma; \gamma')$, then $\vdash R; M; \Delta; \Gamma; \gamma; \gamma'$ holds.

Proof. Straightforward proof by induction on the expression typing derivation. The most interesting cases are the ones of rules *T-AP* and *T-E*:

- *T-A*: By applying lemma A.10 to the typing derivation of e we have that e_1 is well-typed with effect $(\gamma; \gamma_x)$, e_2 is well-typed with effect $(\gamma_x; \gamma_y)$ and $\xi \vdash \gamma' = \gamma_2 \oplus (\gamma \ominus \gamma_y)$. By applying the induction hypothesis to e_1 and e_2 we obtain that $\vdash R; M; \Delta; \Gamma; \gamma; \gamma_x$ and $\vdash R; M; \Delta; \Gamma; \gamma_x; \gamma_y$ respectively. It suffices to prove the following obligations:
  - $R \vdash M$: immediate by inversion of $\vdash R; M; \Delta; \Gamma; \gamma; \gamma_x$.
  - $R; \Delta \vdash \Gamma$: immediate by inversion of $\vdash R; M; \Delta; \Gamma; \gamma; \gamma_x$.
  - $R; \Delta \vdash \gamma$: immediate by inversion of $\vdash R; M; \Delta; \Gamma; \gamma; \gamma_x$.
  - $R; \Delta \vdash \gamma'$: the effect addition assumption implies that the regions of γ' is a subset of the regions of γ. Thus, $R; \Delta \vdash \gamma'$ follows from the fact that $R; \Delta \vdash \gamma$ holds as shown earlier.
  - *set*$(\gamma; \gamma')$: by inversion of $\vdash R; M; \Delta; \Gamma; \gamma; \gamma_x$ we obtain that *set*$(\gamma; \emptyset)$ holds. The effect addition assumption implies that the regions of γ' are contained in the regions of γ. Thus, *set*$(\gamma'; \emptyset)$ is immediate from the fact that *set*$(\gamma; \emptyset)$. Hence *set*$(\gamma; \gamma')$.
  - *ok*$(\gamma; \gamma')$: by inversion of $\vdash R; M; \Delta; \Gamma; \gamma; \gamma_x$ we obtain that *ok*$(\gamma; \emptyset)$ holds. The effect addition assumption implies that the regions of γ' are contained in the regions of γ and the *purity* of each atomic effect of γ' is identical to the *purity* of the same effect in γ. Thus, *ok*$(\gamma; \gamma')$ holds.

Case *T-E*: By applying lemma A.10 to the typing derivation of e we obtain that e_1 is well-typed with effect $(\gamma; \gamma_2)$, $\mathsf{seq} \vdash \gamma' = \gamma_2 \oplus (\gamma_r \ominus \emptyset)$, *ok*$(\gamma_r; \emptyset)$, *set*$(\gamma_r; \emptyset)$ and $R; \Delta \vdash \gamma_r$. By applying the induction hypothesis to e_1 we obtain that $\vdash R; M; \Delta; \Gamma; \gamma; \gamma_2$. It suffices to prove the following obligations:

- $R \vdash M$: immediate by inversion of $\vdash R; M; \Delta; \Gamma; \gamma; \gamma_2$.
- $R; \Delta \vdash \Gamma$: immediate by inversion of $\vdash R; M; \Delta; \Gamma; \gamma; \gamma_2$.
- $R; \Delta \vdash \gamma$: immediate by inversion of $\vdash R; M; \Delta; \Gamma; \gamma; \gamma_2$.
- $R; \Delta \vdash \gamma'$: the effect addition assumption implies that the regions of γ' is a subset of the regions of γ_r. Thus, $R; \Delta \vdash \gamma'$ follows from the fact that $R; \Delta \vdash \gamma_r$.
- *set*$(\gamma; \gamma')$: the effect addition assumption implies that the regions of γ' are contained in the regions of γ_r. Thus, *set*$(\gamma'; \emptyset)$ is immediate from the fact that *set*$(\gamma_r; \emptyset)$. By inversion of $\vdash R; M; \Delta; \Gamma; \gamma; \gamma_2$ we obtain that *set*$(\gamma; \emptyset)$ holds. Hence *set*$(\gamma; \gamma')$.
- *ok*$(\gamma; \gamma')$: we have shown that *ok*$(\gamma_r; \emptyset)$ holds. The effect addition assumption implies that the regions of γ' are contained in the regions of γ_r and the *purity* of each atomic effect of γ' is identical to the *purity* of the same effect in γ_r. Hence, *ok*$(\gamma'; \emptyset)$ holds. By inversion of $\vdash R; M; \Delta; \Gamma; \gamma; \gamma_2$ we obtain that *ok*$(\gamma; \emptyset)$ holds. Thus, *ok*$(\gamma; \gamma')$ holds.

Lemma A.13 (Value-Effect — Using well-formedness) If value v is well-typed in the typing context $R; M; \Delta; \Gamma$, with effect $(\gamma; \gamma)$ and $\vdash R; M; \Delta; \Gamma; \gamma_1; \gamma_2$, then v is well-typed in the same typing context with effect $(\gamma_1; \gamma_1)$ and $(\gamma_2; \gamma_2)$.

Proof. The proof is trivial, but we provide the key steps behind the proof. The assumption implies that $\vdash R; M; \Delta; \Gamma; \gamma_1; \gamma_1$ and also $\vdash R; M; \Delta; \Gamma; \gamma_2; \gamma_2$ hold (trivial). By lemma A.10 we obtain the well-formedness derivation as well as some other premises (in the case of rules *T-L,T-R,T-V,T-F*). We may use the latter premises of value typing, which *still hold* (same typing context), along with the latter two well-formedness derivations to formulate the new value typing derivations with effect $(\gamma_1; \gamma_1)$ and $(\gamma_2; \gamma_2)$ respectively. The case for rule *T-RF* can be shown trivially by induction (the base case is the same as for rule *T-F*).

Lemma A.14 (Value-Effect) If value v is well-typed in the typing context $R; M; \Delta; \Gamma$, with effect $(\gamma; \gamma)$, and e is well-typed in the same typing context with effect $(\gamma'; \gamma'')$, then v is well-typed in the same typing context with effect $(\gamma''; \gamma'')$ and $(\gamma'; \gamma')$.

Proof. By applying lemma A.10 to the typing derivation of v, we have that $\vdash R; M; \Delta; \Gamma; \gamma; \gamma$. Similarly, the application of lemma A.12 to the typing derivation of e implies that $\vdash R; M; \Delta; \Gamma; \gamma'; \gamma''$. The proof is completed by applying lemma A.13.

Lemma A.15 (R Well-Formedness Weakening) $R; \Delta \vdash r \wedge R \subseteq R' \Rightarrow R'; \Delta \vdash r$

Proof. We proceed by performing a case analysis on r:

- $\imath@n$: By inversion of this derivation we have that $R; \Delta \vdash \imath$. We can use the induction hypothesis to complete the proof.
- $r \neq \imath@n$: By inversion of this derivation $\bar{r} \in R \uplus \Delta$ holds. Thus, $\bar{r} \in R' \uplus \Delta$ also holds.

Lemma A.16 (Effect Well-formedness Weakening) $R; \Delta \vdash \gamma \wedge R \subseteq R' \Rightarrow R'; \Delta \vdash \gamma$

Proof. We proceed by performing a case analysis on γ:

- $\emptyset$: $R'; \Delta \vdash \emptyset$ trivially holds.

  $R; \Delta \vdash \gamma', r^{\kappa} \triangleright \pi$: $R'; \Delta \vdash \gamma'$ holds by the induction hypothesis. $R'; \Delta \vdash r$ holds by lemma A.15. If $\pi = r'$, then $R'; \Delta \vdash r'$ holds by lemma A.15.

Lemma A.17 (Type Context Well-formedness Weakening) $R; \Delta \vdash \tau \wedge R \subseteq R' \Rightarrow R'; \Delta \vdash \tau$

Proof. We proceed by performing a case analysis on τ:

- b: $R'; \Delta \vdash b$ trivially holds.
- $\langle\rangle$: $R'; \Delta \vdash \langle\rangle$ trivially holds.
- $\texttt{rgn}(r)$: $R'; \Delta \vdash r$ holds by lemma A.15.
- $\texttt{ref}(\tau', r)$: $R'; \Delta \vdash r$ holds by lemma A.15. $R'; \Delta \vdash \tau'$ holds by the induction hypothesis.
- $\forall \rho.\, \tau'$: $R'; \Delta, \rho \vdash \tau'$ holds by the induction hypothesis.

- $\tau' \xrightarrow{\gamma_1 \to \gamma_2} \tau''$: $R'; \Delta \vdash \tau'$ holds by the induction hypothesis. $R'; \Delta \vdash \tau''$ holds by the induction hypothesis. $R'; \Delta \vdash \gamma_1$ holds by lemma A.16. $R'; \Delta \vdash \gamma_2$ holds by lemma A.16.

Lemma A.18 (Variable context well-formedness weakening) $R; \Delta \vdash \Gamma \wedge R \subseteq R' \Rightarrow R'; \Delta \vdash \Gamma$

Proof. We proceed by performing a case analysis on Γ:

- $\emptyset$: $R'; \Delta \vdash \emptyset$ trivially holds.

  $R; \Delta \vdash \Gamma', x : \tau$: $R'; \Delta \vdash \Gamma'$ holds by the induction hypothesis. $R'; \Delta \vdash \tau$ holds by lemma A.17.

Lemma A.19 (Memory Context Well-formedness Weakening — R) $R \vdash M \wedge R \subseteq R' \Rightarrow R' \vdash M$

Proof. We proceed by performing a case analysis on M:

- $\emptyset$: $R' \vdash \emptyset$ trivially holds.

  $R \vdash M', \ell \mapsto (\tau, \imath)$: $R' \vdash M'$ holds by the induction hypothesis. $R'; \emptyset \vdash \texttt{ref}(\tau, \imath)$ holds by lemma A.17.

Lemma A.20 (Typing Context Well-formedness Weakening) $\vdash R; M; \Delta; \Gamma; \gamma_1; \gamma_2 \wedge R \subseteq R' \Rightarrow \vdash R'; M; \Delta; \Gamma; \gamma_1; \gamma_2$

Proof. Immediate by lemmas A.19, A.18, A.16.

Lemma A.21 (Typing Context Weakening — R) If expression e is well-typed in the typing context $R; M; \Delta; \Gamma$ and R' is a superset of R, then e is well-typed in the context $R'; M; \Delta; \Gamma$ with the same type and effect.

Proof. By applying lemma A.12 to the typing derivation of e we have that $\vdash R; M; \Delta; \Gamma; \gamma_1; \gamma_2$. Lemma A.20 implies that $\vdash R'; M; \Delta; \Gamma; \gamma_1; \gamma_2$ holds.

- *T-I*: Immediate by applying rule *T-I* to $\vdash R'; M; \Delta; \Gamma; \gamma_1; \gamma_2$.
- *T-U*: Immediate by applying rule *T-U* to $\vdash R'; M; \Delta; \Gamma; \gamma_1; \gamma_2$.
- *T-R*: By applying lemma A.10 to this derivation we have that $R; \Delta \vdash \imath$ and $r \simeq \imath$. Lemma A.15 implies that $R'; \Delta \vdash \imath$ holds. Thus, we can apply rule *T-R* to the latter fact, $r \simeq \imath$ and $\vdash R'; M; \Delta; \Gamma; \gamma_1; \gamma_2$ to complete the proof.
- *T-L*: By applying lemma A.10 to this derivation we have that $(\ell \mapsto (\tau', \imath)) \in M$ and $\texttt{ref}\, M(\ell) \simeq \tau$. Thus, we can apply rule *T-L* to $\vdash R'; M; \Delta; \Gamma; \gamma_1; \gamma_2$, $(\ell \mapsto (\tau', \imath)) \in M$ and $\texttt{ref}\, M(\ell) \simeq \tau$ to complete the proof.
- *T-V*: By applying lemma A.10 to this derivation we have that $(x : \tau') \in \Gamma$ and $\tau' \simeq \tau$. Thus, we can apply rule *T-V* to $\vdash R'; M; \Delta; \Gamma; \gamma_1; \gamma_2$, $(x : \tau') \in \Gamma$ and $\tau' \simeq \tau$ to complete the proof.
- *T-F*: By applying lemma A.10 to this derivation we have that
  - $\vdash R; M; \Delta; \Gamma; \gamma; \gamma$: We have shown that $\vdash R'; M; \Delta; \Gamma; \gamma_1; \gamma_2$ holds.

– $R; \Delta \vdash \tau$: $R'; \Delta \vdash \tau$ holds by lemma A.17.
– $\tau' \simeq \tau$
– $\tau \equiv \tau_1 \xrightarrow{\gamma_1 \to \gamma_2} \tau_2$
– $\mathit{set}(\gamma_1; \gamma_2)$
– $\mathit{ok}(\gamma_1; \gamma_2) \Rightarrow R; M; \Delta; \Gamma, x : \tau_1 \vdash e' : \tau_2 \,\&\, (\gamma_1;\gamma_2)$: Assuming that $\mathit{ok}(\gamma_1; \gamma_2)$ holds, we apply the induction hypothesis to the derivation of e' to derive that $R'; M; \Delta; \Gamma, x : \tau_1 \vdash e' : \tau_2 \,\&\, (\gamma_1;\gamma_2)$ holds.

We then apply rule *T-F* to the above facts to derive $R'; M; \Delta; \Gamma \vdash \lambda x.\, e' \texttt{ as } \tau : \tau' \,\&\, (\gamma;\gamma)$.

Case *T-AP*, *T-CP*, *T-RP*, *T-NG*, *T-NR*, *T-D*, *T-RF, T-E, T-A*: similar reasoning is performed to prove the remaining cases. Lemmas A.15 and A.17 can be used for premises of the form $R; \Delta \vdash r$ and $R; \Delta \vdash \tau$ respectively.

Lemma A.22 (Memory Context Weakening) If expression e is well-typed in the typing context $R; M; \Delta; \Gamma$, $R \vdash M'$ holds, and M' is a superset of M, then e is well-typed in the context $R; M'; \Delta; \Gamma$ withthe same type and effect.

Proof. By applying lemma A.12 to the typing derivation of v we have that $\vdash R; M; \Delta; \Gamma; \gamma_1; \gamma_2$. Thus, we can substitute premise $R \vdash M$ with $R \vdash M'$ to obtain $\vdash R; M'; \Delta; \Gamma; \gamma_1; \gamma_2$.

- *T-I*: Immediate by applying rule *T-I* to $\vdash R; M'; \Delta; \Gamma; \gamma_1; \gamma_2$.
- *T-U*: Immediate by applying rule *T-U* to $\vdash R; M'; \Delta; \Gamma; \gamma_1; \gamma_2$.
- *T-R*: By applying lemma A.10 to this derivation we have that $R; \Delta \vdash \imath$. The proof is completed by applying rule *T-R* to $R; \Delta \vdash \imath$. and $\vdash R; M'; \Delta; \Gamma; \gamma_1; \gamma_2$.
- *T-L*: By applying lemma A.10 to this derivation we have that $(\ell \mapsto (\tau', \imath)) \in M$ and $\texttt{ref}\, M(\ell) \simeq \tau$. Thus, $(\ell \mapsto (\tau', \imath)) \in M'$ and $\texttt{ref}\, M'(\ell) \simeq \tau$ also hold as $M \subseteq M'$. We can apply rule *T-L* to $\vdash R; M'; \Delta; \Gamma; \gamma_1; \gamma_2$, $(\ell \mapsto (\tau', \imath)) \in M'$ and $\texttt{ref}\, M'(\ell) \simeq \tau$ to complete the proof.
- *T-V*: By applying lemma A.10 to this derivation we have that $(x : \tau') \in \Gamma$ and $\tau' \simeq \tau$. Thus, we can apply rule *T-V* to $\vdash R; M'; \Delta; \Gamma; \gamma_1; \gamma_2$, $(x : \tau') \in \Gamma$ and $\tau' \simeq \tau$ to complete the proof.
- *T-F*: By applying lemma A.10 to this derivation we have that
  – $\vdash R; M; \Delta; \Gamma; \gamma; \gamma$: We have shown that $\vdash R; M'; \Delta; \Gamma; \gamma_1; \gamma_2$ holds.
  – $\mathit{set}(\gamma_1; \gamma_2)$
  – $R; \Delta \vdash \tau$
  – $\tau' \simeq \tau$
  – $\tau \equiv \tau_1 \xrightarrow{\gamma_1 \to \gamma_2} \tau_2$
  – $\mathit{ok}(\gamma_1; \gamma_2) \Rightarrow R; M; \Delta; \Gamma, x : \tau_1 \vdash e' : \tau_2 \,\&\, (\gamma_1;\gamma_2)$: assume that $\mathit{ok}(\gamma_1; \gamma_2)$ holds. By applying the induction hypothesis to this derivation we have that $R; M'; \Delta; \Gamma, x : \tau_1 \vdash e' : \tau_2 \,\&\, (\gamma_1;\gamma_2)$ holds.

  We can apply rule *T-F* to the above facts to derive $R; M'; \Delta; \Gamma \vdash \lambda x.\, e' \texttt{ as } \tau : \tau' \,\&\, (\gamma;\gamma)$.

Case *T-AP*, *T-CP*, *T-RP*, *T-NG*, *T-NR*, *T-D*, *T-RF, T-E, T-A*: We can perform similar reasoning to prove the remaining cases.

Lemma A.23 (Replacement) If expressions $E[e_1]$, e_1 and e_2 are well-typed in the typing context $R; M; \Delta; \Gamma$, with effects $(\gamma_1; \gamma_2)$,$(\gamma_1; \gamma_3)$ and $(\gamma_4; \gamma_3)$ respectively, then expression $E[e_2]$ is also well-typed in the same typing context with effect $(\gamma_4; \gamma_2)$.

Proof. By straightforward induction on the shape of the evaluation context. The intuition behind this proof is that the substitution of e_2 for e_1 in the evaluation context E will not surpise its environment as both e_1 and e_2 yield the same output effect. In regards to the input effect, we know that the environment will not be surprised as the expressions preceding e_1 will definitely be values and can be given the input effect of e_2 (by lemma A.14).

Case $\Box[e]$ then proof is immediate.

Case $(\texttt{new}\ v\ @\ E')[e]$: Lemma A.10 implies that $R; M; \Delta; \Gamma \vdash v : \tau_1 \& (\gamma_1; \gamma_1)$. The application of lemma A.14 to the latter judgement and the fact e_2 is well-typed with effect $(\gamma_4; \gamma_3)$ yields $R; M; \Delta; \Gamma \vdash v : \tau_1 \& (\gamma_4; \gamma_4)$. By applying lemma A.10 to the memory allocation construct typing derivation yields *is_live*(γ_3, r) and $R; M; \Delta; \Gamma \vdash E'[e] : \texttt{rgn}(r) \& (\gamma_1; \gamma_2)$. The application of the induction hypothesis on the derivation of $E'[e_2]$ and the derivation of e_2 (assumption) yields $R; M; \Delta; \Gamma \vdash E'[e_2] : \tau_1$ & $(\gamma_4;\gamma_2)$. Now, *T-NR* can be applied to the latter judgment, the new derivation of v, and the fact that *is_live*(γ_3, r) to obtain $R; M; \Delta; \Gamma \vdash \texttt{new}\, v\ @\ E'[e_2] : \texttt{ref}(\tau_1, r)$ & $(\gamma_4;\gamma_2)$ or equivalently $R; M; \Delta; \Gamma \vdash (\texttt{new}\, v\ @\ E')[e_2] : \texttt{ref}(\tau_1, r)$ & $(\gamma_4;\gamma_2)$.

Case $((E'\ e_2)^{\xi})[e]$, $((v\ E')^{\xi})[e]$, $(\texttt{cap}^r_\eta\ E')[e]$,$(\texttt{deref}\ E')[e]$,$(E' := e_2)[e]$, $(\texttt{loc}_\ell := E')[e]$, $(\texttt{new}\, E'\ @\ e_2))[e]$, $(\texttt{pop}_\gamma\ E'))[e]$,$(E'\ [r])[e]$, $(\texttt{newrgn}^r\ \rho, x\ @\ E'\ \texttt{in}\ e_2)[e]$: Similar to the above proof structure.

Lemma A.24 (Parallel-Sequential typing implication) If a parallel application term is well-typed $(R; M; \Delta; \Gamma \vdash (v_1\ v_2)^{\mathsf{par}} : \langle\rangle \& (\gamma; \gamma'))$, where $v_1 \equiv \lambda x.\, e\ \texttt{as}\ \tau_1 \xrightarrow{\gamma_1 \to \emptyset} \langle\rangle$, then the corresponding sequential application term $((v_1\ v_2)^{\mathsf{seq}})$ is also well-typed in the same typing context, with effect $(\gamma_1; \emptyset)$.

Proof. Lemma A.10 implies that v_1 and v_2 are well typed in the same typing context $R; M; \Delta; \Gamma$, with effects $(\gamma; \gamma)$ and $(\gamma; \gamma)$ respectively. It also implies that $R; M; \Delta; \Gamma, x : \tau_1 \vdash e : \langle\rangle \& (\gamma_1; \emptyset)$. By applying lemma A.14 to the typing derivations of v_1,v_2, and the fact that e is well-typed with effect $(\gamma_1; \emptyset)$, we obtain that v_1 and v_2 are well-typed in the same typing context with effect $(\gamma_1; \gamma_1)$. We can derive $\mathsf{seq} \vdash \emptyset = \emptyset \oplus (\gamma_1 \ominus \gamma_1)$. By applying *T-AP* to the latter facts, we have that $R; M; \Delta; \Gamma \vdash (v_1\ v_2)^{\mathsf{seq}} : \langle\rangle \& (\gamma_1; \emptyset)$ holds.

Lemma A.25 (Store Typing Preservation for $\vdash \delta$ — Helper 1) If $\sigma; \gamma \vdash \delta''$, *ok*$(\gamma; \gamma')$ and $\mathsf{par} \vdash \gamma' = \emptyset \oplus (\gamma \ominus \gamma_1)$ hold, $\delta_y \subseteq \delta''$ then $\sigma; \gamma' \vdash \delta_y$ also holds.

Proof. Proof by induction on the structure of δ_y:

- $\emptyset$: given that $\sigma; \gamma' \simeq \sigma_1; \gamma_x, \imath^{\kappa} \triangleright \pi + \sigma_2$, *rg*$(\kappa) > 0$ and *is_pure*(κ) hold, it suffices to prove that *zero_pure*$(\sigma_1, \imath)$ and $\imath \notin$ *dom*$(\sigma_2; \gamma_x)$ for all $\imath$ in the domain of $\sigma; \gamma'$. The assumption that $\sigma; \gamma \vdash \delta''$ holds implies that $\sigma; \gamma \simeq \sigma_3; \gamma_y, \imath^{\kappa'} \triangleright \pi' + \sigma_4$, $\imath \notin$ *dom*$(\sigma_4; \gamma_y)$ and *zero_pure*$(\sigma_3, \imath)$. We proceed by performing a case analysis:

  – $\imath$ belongs in the domain of $\emptyset; \gamma$: the following constraints hold from the above facts: $\sigma_2 = \sigma_4 = \emptyset$ and $\sigma_1 \simeq \sigma_3 \simeq \sigma$, $\gamma \simeq \gamma_y$, $\imath^{\kappa'} \triangleright \pi'$ and $\gamma' \simeq \gamma_x, \imath^{\kappa} \triangleright \pi$. Thus, *zero_pure*$(\sigma_1, \imath)$ holds and the assumption *ok*$(\gamma; \gamma')$ implies that $\imath \notin$ *dom*$(\sigma_2; \gamma_x)$.

– $\imath$ does not belong in the domain of $\emptyset;\gamma$: the following constraints hold from the above facts: $\sigma_1;\gamma_x,\imath^{\kappa}\triangleright\pi \simeq \sigma_3;\gamma_y,\imath^{\kappa'}\triangleright\pi'$, $\sigma_2 \simeq \sigma_a;\gamma'$, $\sigma_4 \simeq \sigma_b;\gamma$ and $\sigma_a \simeq \sigma_b$. Thus, *zero_pure*$(\sigma_1,\imath)$ is immediate. The equalties and $\imath \notin$ *dom*$(\sigma_4;\gamma_y)$ imply that $\imath \notin$ *dom*(σ_b) and thus $\imath \notin$ *dom*(σ_a). It suffices to show that $\imath \notin \gamma_x$. This is immediate by the fact that $\imath \notin$ *dom*$(\sigma_2;\gamma_y)$ and *dom*$(\emptyset;\gamma_y) \subseteq$ *dom*$(\emptyset;\gamma_x)$ (by the capability addition assumption).

Case $\delta_1, n_1 \mapsto \sigma_1$: by applying the induction hypothesis we have that $\sigma;\gamma' \vdash \delta_1$ holds. Given that *is_accessible*$(\sigma;\gamma',\imath)$ holds for all $\imath$ that belong in the domain of $\sigma;\gamma'$, it suffices to prove that $\neg$*is_accessible*$(\sigma_1,\imath)$ holds. The assumption that $\sigma;\gamma \vdash \delta''$ holds implies that *is_accessible*$(\sigma;\gamma,\imath) \Rightarrow \neg$*is_accessible*$(\sigma_1,\imath)$. The capability addition assumption implies that if *is_accessible*$(\sigma;\gamma',\imath)$, then *is_accessible*$(\sigma;\gamma,\imath)$. Thus, the latter two facts imply that $\neg$*is_accessible*$(\sigma_1,\imath)$.

Lemma A.26 (Store Typing Preservation for $\vdash \delta$ — Helper 2) If $\vdash \delta'', n \mapsto \sigma;\gamma$, $\mathsf{par} \vdash \gamma' = \emptyset \oplus (\gamma \ominus \gamma_1)$ hold and $\delta_y \subseteq \delta'', n \mapsto \sigma;\gamma'$, then $\emptyset;\gamma_1 \vdash \delta_y$ holds.

Proof. Proof by induction on the shape of δ_y.

- $\emptyset$: given that $\emptyset;\gamma_1 \simeq \sigma_1;\gamma_x,\imath^{\kappa}\triangleright\pi+\sigma_2$, *rg*$(\kappa) > 0$ and *is_pure*(κ) hold, then it suffices to prove that *zero_pure*(σ_1) and $\imath \notin$ *dom*$(\sigma_2;\gamma)$. for all $\imath$ in the domain of $\emptyset;\gamma_1$. This is immediate by the fact that σ_1 and σ_2 are empty and *ok*$(\gamma_1;\gamma_2)$ (obtained by inversion of the effect addition assumption).

- $\delta_1, n_1 \mapsto \sigma_1$: $\emptyset;\gamma_1 \vdash \delta_1$ is immediate by applying the induction hypothesis. It suffices to prove $\neg$*is_accessible*$(\sigma_1,\imath)$ holds for all $\imath$ that belong in the domain of γ_1 given that *is_accessible*$(0;\gamma_1,\imath)$ holds. If $\imath$ exists in the domain of γ_1, then the effect addition assumption tells us that $\imath$ exists in γ and *is_accessible*$(\gamma,\imath)$ holds. Thus, by inversion of $\vdash \delta'', n \mapsto \sigma;\gamma$ we have that $\neg$*is_accessible*$(\sigma_1,\imath)$ holds, when $n_1 \neq n$.

  To complete the proof it must be proved that $\neg$*is_accessible*$(\sigma;\gamma',\imath)$ holds. The capability addition assumption implies that $\imath$ or at least one of its ancestors has a positive *pure* capability in both γ and γ_1. It also tells us that there exists no positive *impure* capability in γ_1. Assume $\jmath$ is a region protecting $\imath$ (may be equal to $\imath$) with a positive and pure capability. By inversion of $\vdash \delta'', n \mapsto \sigma;\gamma$ we have that $\sigma;\gamma \vdash \emptyset$ holds. By inversion of the latter derivation we have that *zero_pure*$(\sigma_1,\jmath)$ and $\jmath \notin$ *dom*$(\sigma_2;\gamma_3)$. Region $\jmath$ is positive in γ thus, $\sigma_1 = \sigma$, $\sigma_2 = \emptyset$ and $\gamma = \gamma_3,\jmath^{\kappa'}\triangleright\pi'$. Consequently, $\neg$*is_accessible*$(\sigma;\gamma',\jmath)$ holds by the latter fact and the effect addition assumption.

Lemma A.27 (Store Typing Preservation for $\vdash \delta$ — Spawn) If $\delta = \delta'', n \mapsto \sigma;\gamma$, $\delta = \delta'', n \mapsto \sigma;\gamma', n' \mapsto \emptyset;\gamma_1$, *ok*$(\gamma;\gamma')$, $\vdash \delta$ and $\mathsf{par} \vdash \gamma' = \emptyset \oplus (\gamma \ominus \gamma_1)$ hold, then $\vdash \delta'$ holds.

Proof. It suffices to show that:

- $\vdash \delta''$: immediate by inversion of $\vdash \delta$.

- $\sigma;\gamma' \vdash \delta''$: by inversion of $\vdash \delta$ we obtain that $\sigma;\gamma \vdash \delta''$. The proof for this case is completed by the application of lemma A.25.

- $\emptyset;\gamma_1 \vdash \delta'', n \mapsto \sigma;\gamma'$: the proof for this case is immediate by lemma A.26.

Lemma A.28 (Store Typing Preservation — Spawn) If $\delta; S$ is a well-typed store in respect to $R; M$ where δ equals $\delta'', n \mapsto \sigma;\gamma$, *ok*$(\gamma;\gamma')$ holds, $\mathsf{par} \vdash \gamma' = \emptyset \oplus (\gamma \ominus \gamma_1)$ holds, *live*$(\gamma_1) = \gamma_1$, and δ' equals $\delta'', n \mapsto \sigma;\gamma', n' \mapsto \emptyset;\gamma_1$ (fresh n'), then $\delta'; S$ is well-typed in respect to $R; M$.

Proof. By inversion of the store typing assumption, we have that:

- $R \vdash \delta$
- $R; M \vdash S$
- $\vdash \delta$

The capability addition assumption implies that the regions of γ_1 and γ' are subsets of the regions of γ. Therefore $R \vdash \delta$ implies that $R \vdash \delta'$ holds. We have that $R; M \vdash S$, thus it suffices to show that $\vdash \delta$ and $\mathsf{par} \vdash \gamma' = \emptyset \oplus (\gamma \ominus \gamma_1)$ imply that $\vdash \delta'$ holds. This is immediate by lemma A.27.

Lemma A.29 (Preservation — Expressions) Let e be a well-typed expression with, $\vdash \delta[n \mapsto \sigma'; \gamma']$, $\delta(n) = \sigma; \gamma$, $R; M; \emptyset; \emptyset \vdash e : \tau \,\&\, (\gamma;\gamma'')$ and $R; M \vdash \delta; S$. If the operational semantics takes a step $\delta(n); S; e \to (\sigma'; \gamma'); S'; e'$, then there exist $R' \supseteq R$ and $M' \supseteq M$, such that the resulting expression and the resulting store are well-typed with $R'; M'; \emptyset; \emptyset \vdash e' : \tau \,\&\, (\gamma';\gamma'')$, $R; M \vdash \delta[n \mapsto \sigma'; \gamma']; S'$

Proof. By induction on the typing derivation. It is worth noting that e is a redex, which is immediate by the definition of evaluation relation. Henceforth, we use u where e should be used to stress that u is a redex.

Case *T-I*, *T-U*, *T-F*, *T-L*, *T-R*, *T-V*, *T-RF*: the proof is immediate as u is a value and the assumption that we perform a single operational step does not hold.

Case *T-E*: The shape of u is $\mathsf{pop}_{\gamma_r}\ v$ for some value v. By applying lemma A.10 to rule *T-E*, we have that $ok(\gamma_r; \emptyset)$, $\mathsf{seq} \vdash \gamma' = \gamma_2 \oplus (\gamma_r \ominus \emptyset)$ and $R; M; \emptyset; \emptyset \vdash v : \tau' \& (\gamma_1; \gamma_2)$, where γ_1 and γ' is the input and output effect of $\mathsf{pop}_{\gamma_r}\ v$ respectively, and $\tau' \simeq \tau$. By applying lemma A.10 to the latter fact we have that $\gamma_1 = \gamma_2$. Thus, the earlier facts can be rewritten as $\mathsf{seq} \vdash \gamma' = \gamma_1 \oplus (\gamma_r \ominus \emptyset)$ and $R; M; \emptyset; \emptyset \vdash v : \tau' \& (\gamma_1; \gamma_1)$. The application of lemma A.34 to the latter derivation and $\tau \simeq \tau'$ implies that $R; M; \emptyset; \emptyset \vdash v : \tau \& (\gamma_1; \gamma_1)$ holds.

The operational rule that matches the shape of u is *E-E* and gives us that $\delta; S; \mathsf{pop}_{\gamma_r}\ v$ evaluates to $\delta'; S; v$. The premises of rule *E-E* are $\mathsf{seq} \vdash \gamma'' = \gamma_1 \oplus (\gamma_r \ominus \emptyset)$, where δ and δ' equal $\delta'', n \mapsto \sigma; \gamma_r; \gamma_1$ and $\delta'', n \mapsto \sigma; \gamma''$ respectively. The capability addition rule is deterministic, thus γ'' equals γ'. The application of lemma A.13 to $\vdash R; M; \emptyset; \emptyset; \gamma'; \gamma'$ and $R; M; \emptyset; \emptyset \vdash v : \tau \& (\gamma_1; \gamma_2)$, yields $R; M; \emptyset; \emptyset \vdash v : \tau \& (\gamma'; \gamma')$. To complete the proof, we need to show that $R; M \vdash \delta'; S$. This is immediate by the application of lemma A.52 to $R; M \vdash S$, $ok(\gamma_r; \gamma_1)$ (obtained by $ok(\gamma_r; \emptyset)$ and $ok(\gamma_1; \gamma_2)$; $ok(\gamma_1; \gamma_2)$ is immediate by applying lemma A.12 to the typing derivation of v), $\mathsf{seq} \vdash \gamma'' = \gamma_1 \oplus (\gamma_r \ominus \emptyset)$, $\delta = \delta'', n \mapsto \sigma; \gamma_r; \gamma_1$ and $\delta' = \delta'', n \mapsto \sigma; \gamma''$.

Case *T-RP*: The typing derivation of *T-RP* gives us that u is of the form $(e)\,[r]$. The operational rule that matches the shape of u is *E-RP*. Thus, u is of the form $(\Lambda\rho.\ f)\,[r]$. We can apply lemma A.10 to the latter derivation to obtain that $R; M; \emptyset, \rho; \emptyset \vdash f : \tau \& (\gamma; \gamma)$, where γ equals $\delta(n)$. The application of lemma A.35 to the latter fact, $\overline{\bar{r}@n'} \in R$ (premise $R; \emptyset \vdash r$ of rule *T-RP*), the fact that $\bar{r}@n'$ is *fresh* (premise of rule *E-RP*), $R; \emptyset \vdash \gamma$ (premise of $\vdash R; M; \emptyset; \emptyset; \gamma; \gamma$; the well-formedness fact is immediate by the application of lemma A.12 to the typing derivation of type application), $\overline{\bar{r}@n'} \simeq r$ (by the premise of rule *E-RP* and the definition of relation $\simeq$), gives us that $R; M; \emptyset; \emptyset \vdash f[\bar{r}@n'/\rho] : \tau[r/\rho] \& (\gamma; \gamma)$.

Therefore, typing is preserved. The resulting store is identical to the input store, thus it is also well-typed by the assumption of this lemma.

Case *T-CP*: **Expression typing:** The application of lemma A.12 to the typing derivation of the assumption gives us that $\vdash R; M; \emptyset; \emptyset; \gamma; \gamma''$, where γ is the equal to $\delta(n)$. Thus, $\vdash R; M; \emptyset; \emptyset; \gamma''; \gamma''$ also holds. The application of rule *T-U* to the latter fact gives us $R; M; \emptyset; \emptyset \vdash () : \langle\rangle \& (\gamma''; \gamma'')$.

Store typing: The operational rule *E-C* matches the shape of u. Thus, we need to prove that $R; M \vdash \delta'; S$ holds, where $\delta' = \delta[n \mapsto \sigma'; \gamma']$. It suffices to show that $\vdash \delta'$ holds. This is immediate by the assumptions of this lemma.

Case *T-NG*: Rule *E-NG* matches the shape of u. This rule implies that

$\sigma; S; \texttt{newrgn}^r\ \rho, x\ @\ \texttt{rgn}_{\bar{r}}\ \texttt{in}\ e_1 \rightarrow (\sigma; \gamma, \imath^{1,1} \triangleright r); S'; e_2[\imath/\rho][\texttt{rgn}_\imath/x], \sigma = \sigma_0; \gamma$, *is_live*$(\gamma, r)$, $S' = S, \imath \mapsto \emptyset$, fresh $\imath$ and $\delta' = \delta, n \mapsto \sigma; \gamma, \imath^{1,1} \triangleright r$ hold.

Store typing: We must prove that $R, \imath; M \vdash \delta'; S'$ hold given that $R; M \vdash \delta; S$ holds. The latter derivation gives us that $R; M \vdash S$, $R \vdash S$ and $\vdash \delta$.

- $R, \imath; M \vdash S'$:
  - $M \vdash S'$: Trivially holds as $M \vdash S$ holds and $S' = S, \imath \mapsto \emptyset$.
  - $R, \imath \vdash S'$: Trivially holds as $R \vdash S$ holds and $S' = S, \imath \mapsto \emptyset$.
- $R, \imath \vdash \delta'$: $R \vdash \delta$ holds and the only new region introduced in δ' is $\imath$.
- $\vdash \delta[n \mapsto \sigma_0; \gamma, \imath^{1,1} \triangleright r]$: immediate by the assumption of this lemma.

Expression typing: The store typing derivation of $\delta'; S'$ implies that $\imath \notin R$. Lemma A.10 implies that $R; M; \emptyset, \rho; \emptyset, x : \texttt{rgn}(\rho) \vdash e_2 : \tau \& (\gamma, \rho^{1,1} \triangleright r; \gamma'')$, such that $\rho \notin dom(\gamma'')$. The application of lemma A.21 to the typing derivation of e_2 and the fact that $\imath \notin R$ yields $R, \imath; M; \emptyset, \rho; \emptyset, x : \texttt{rgn}(\rho) \vdash e_2 : \tau \& (\gamma, \rho^{1,1} \triangleright r; \gamma'')$. By applying lemma A.12 to the original typing derivation of $newrgn$ construct we obtain the well-formedness derivation. By inversion of the latter derivation we have that *ok*$(\gamma; \gamma'')$, thus *ok*$(\gamma; \emptyset)$ holds. ρ is a fresh type variable so it does not exist in the domain of γ. Thus, *ok*$(\gamma, \rho^{1,1} \triangleright r)$ also holds. We then apply lemma A.46 on the latter fact, the derivation of e_2 and the fact that $\imath$ is *fresh* to obtain $R, \imath; M; \emptyset; \emptyset, x : \texttt{rgn}([\imath/\rho]) \vdash e_2[\imath/\rho] : \tau[\imath/\rho] \& (\gamma[\imath/\rho], \imath^{1,1} \triangleright r; \gamma''[\imath/\rho])$. By applying lemma A.12 to the original typing derivation of $newrgn$ construct we have that the typing the context (including γ and γ'') is not defined in terms of ρ (i.e. ρ is *fresh*). Further, the premise of $newrgn$ derivation suggests that τ is also independent of ρ (i.e. $R; \emptyset \vdash \tau$). Hence, the above facts and the definition of the substitution relation imply that the typing derivation of e_2 becomes $R, \imath; M; \emptyset; \emptyset, x : \texttt{rgn}(\imath) \vdash e_2[\imath/\rho] : \tau \& (\gamma, \imath^{1,1} \triangleright r; \gamma'')$. By the application of lemma A.12 to the fact that e_2 is well-typed, we have that $\vdash R, \imath; M; \emptyset; \emptyset; \gamma, \rho^{1,1} \triangleright r; \gamma''$ is well formed. By the definition of well-formedness, $\vdash R, \imath; M; \emptyset; \emptyset; \emptyset; \emptyset$ also holds. The definition of the typing rule *T-R*, the latter fact and the fact that $R, \imath \vdash \imath$ holds imply that $\texttt{rgn}_\imath$ is well-typed (with type $\texttt{rgn}(\imath)$) in the context $R, \imath; M; \emptyset; \emptyset$ with effect $(\emptyset; \emptyset)$. By applying lemma A.36 to the latter derivation and the fact that $R, \imath; M; \emptyset; \emptyset, x : \texttt{rgn}(\imath) \vdash e_2[k/\rho] : \tau \& (\gamma, \imath^{1,1} \triangleright r; \gamma'')$ we obtain $R, \imath; M; \emptyset; \emptyset \vdash e_2[\imath/\rho][\texttt{rgn}_\imath/x] : \tau \& (\gamma, \imath^{1,1} \triangleright r; \gamma'')$.

Case *T-D*: Rule *E-D* matches the shape of u. Its premises also imply that the value read from the store is equal to $S(\bar{r})(\ell)$, for some r such that $\bar{r} = \imath$. The store typing assumption yields that $R; M; \emptyset; \emptyset \vdash v : \tau' \& (\emptyset; \emptyset)$, where $v = S(\imath)(\ell)$ and $M(\ell) = (\tau', \imath)$.

The application of lemma A.12 to the typing derivation of $deref$ gives us $\vdash R; M; \emptyset; \emptyset; \gamma; \gamma$. By applying lemma A.13 to the latter derivation and $R; M; \emptyset; \emptyset \vdash v : \tau' \& (\emptyset; \emptyset)$ gives us that $R; M; \emptyset; \emptyset \vdash v : \tau' \& (\gamma; \gamma)$. By applying lemma A.10 to the typing derivation of $deref$ we have that if $\texttt{ref}(\tau'', r)$ is the type assigned to $\texttt{loc}_\ell$, then $\texttt{ref}(\tau'', r) \simeq \texttt{ref}\ M(\ell)$ holds. Thus $\tau'' \simeq \tau'$ also holds. We can use lemma A.34, the latter fact and $R; M; \emptyset; \emptyset \vdash v : \tau' \& (\gamma; \gamma)$ (lemma A.10) to derive $R; M; \emptyset; \emptyset \vdash v : \tau'' \& (\gamma; \gamma)$. The output store is identical to the input store hence it is also well-typed.

Case *T-A*:

Expression typing: The application of lemma A.12 to the typing derivation of e yields that $R; M; \emptyset; \emptyset; \gamma; \gamma'$ holds. Thus, $R; M; \emptyset; \emptyset; \gamma'; \gamma'$ holds. The application of rule *T-U* to the latter fact yields that $R; M; \emptyset; \emptyset \vdash () : \langle\rangle \& (\gamma'; \gamma')$.

Store typing: The store preservation proof is as follows: Lemma A.10 implies that the following hold: $R; M; \emptyset; \emptyset \vdash \texttt{loc}_\ell : \texttt{ref}(\tau, r)\&(\gamma; \gamma)$, where γ is equal to $\delta(n)$, $R; M; \emptyset; \emptyset \vdash v : \tau\&(\gamma; \gamma)$ and $\texttt{ref}(\tau, r) \simeq \texttt{ref}\ M(\ell)$ (this also implies that $\texttt{ref}(\tau, r)$ contains no type variables). Let $M(\ell)$ be equal to (τ', r') for some τ' and r', then we also have that $\tau \simeq \tau'$.

By applying lemma A.34 to $\tau \simeq \tau'$ and $R; M; \emptyset; \emptyset \vdash v : \tau\&(\gamma; \gamma)$, we have that $R; M; \emptyset; \emptyset \vdash v : \tau'\&(\gamma; \gamma)$. The application of lemma A.12 to the latter derivation implies $\vdash R; M; \emptyset; \emptyset; \gamma; \gamma$. Thus, $\vdash R; M; \emptyset; \emptyset; \emptyset; \emptyset$ also holds. The application of lemma A.13 to the latter fact and $R; M; \emptyset; \emptyset \vdash v : \tau'\&(\gamma; \gamma)$ gives us $R; M; \emptyset; \emptyset \vdash v : \tau'\&(\emptyset; \emptyset)$.

The premise of the operational rule *E-AS* implies that if the input store is $\delta; S$, then the output store is $\delta; S[\bar{r} \mapsto S(\bar{r}), \ell \mapsto v]$. We have from the original store typing assumption that:

- $\vdash \delta$
- $R \vdash \delta$
- $R; M \vdash S$: $R \vdash S$ and $M \vdash S$

Thus, it suffices to show that $R; M \vdash S'$ holds. $R \vdash S'$ holds as $R \vdash S$ holds as no regions are added to S'. $M \vdash S'$ holds as $M \vdash S$ holds for all other locations than ℓ, and ℓ itself contains now the updated value v with typing derivation $R; M; \emptyset; \emptyset \vdash v : \tau'\&(\emptyset; \emptyset)$. Thus, $M \vdash S'$ holds.

Case *T-NR*: The rule that matches this case is rule *E-NR*. This rules implies that the new store $S' = S[n \mapsto S(\bar{r}), \ell \mapsto v]$, where v is the new value that is stored in S', δ is constant and ℓ is a *fresh* location (i.e. ℓ does not exist in S). Therefore, store typing assumption ($R; M \vdash S$) implies that ℓ does not belong in the domain of M.

By applying lemma A.10 to the typing derivation of construct new we have that:

- $R; M; \emptyset; \emptyset \vdash v : \tau\&(\gamma; \gamma)$
- $R; M; \emptyset; \emptyset \vdash \texttt{rgn}_{\bar{r}} : \texttt{rgn}(r')\&(\gamma; \gamma)$
- $r' \simeq \bar{r}$

Let τ' be such that $\tau' \simeq \tau$ and τ' contains no region names of the form $\imath@n$. By applying lemma A.34 to the latter fact we have that $R; M; \emptyset; \emptyset \vdash v : \tau'\&(\gamma; \gamma)$.

The application of lemma A.30 to the latter typing derivation of v tells us that $R; \emptyset \vdash \tau'$ holds. The application of lemma A.10 to the typing derivation of $\texttt{rgn}_{\bar{r}}$ gives us that $R; \emptyset \vdash \imath$. Thefore, $R; \emptyset \vdash \texttt{ref}(\tau', \imath)$ holds. By applying lemma A.12 to the typing derivation of v we have that $\vdash R; M; \emptyset; \emptyset; \gamma; \gamma$. By inversion of the latter derivation $R \vdash M$ holds. Location ℓ is *fresh* so it does not belong to the domain of M. Consequently, we can combine the latter facts to derive that $R \vdash M, \ell \mapsto (\tau', \bar{r})$. **Expression typing:** The latter derivation is substituted for $R \vdash M$ in the premises of $\vdash R; M; \emptyset; \emptyset; \gamma; \gamma$ to derive that $\vdash R; M, \ell \mapsto (\tau', \bar{r}); \emptyset; \emptyset; \gamma; \gamma$ holds. By applying rule *T-L* to the latter fact, $M, \ell \mapsto (\tau', \bar{r})$ and $\texttt{ref}(\tau, r) \simeq \texttt{ref}\ (\tau', \bar{r})$ we obtain that $R; M, \ell \mapsto (\tau', \bar{r}); \emptyset; \emptyset \vdash \texttt{loc}_\ell : \texttt{ref}(\tau, r)\&(\gamma; \gamma)$.

Store typing: By applying lemma A.12 to the typing derivation of construct new we have that $\vdash R; M; \emptyset; \emptyset; \gamma; \gamma''$, where γ'' equals γ. Thus, $\vdash R; M; \emptyset; \emptyset; \emptyset; \emptyset$ also holds. By applying lemma A.13 to the latter fact and $R; M; \emptyset; \emptyset \vdash v : \tau'\&(\gamma; \gamma)$ we have that $R; M; \emptyset; \emptyset \vdash v : \tau'\&(\emptyset; \emptyset)$ holds. By applying lemma A.22 to the latter derivation $R \vdash M, \ell \mapsto (\tau', \bar{r})$ and $M \subseteq M, \ell \mapsto (\tau', \bar{r})$ we have that $R; M, \ell \mapsto (\tau', \bar{r}); \emptyset; \emptyset \vdash v : \tau'\&(\emptyset; \emptyset)$.

By inversion of the store typing assumption we have that $M \vdash S$ and $\forall(\ell' \mapsto (\tau'', \jmath)) \in M.R; M; \emptyset; \emptyset \vdash S(\jmath)(\ell') : \tau''\ \&\ (\emptyset;\emptyset)$. We must show that $R; M, \ell \mapsto (\tau', \bar{r}) \vdash \delta; S'$. It suffices to show that the following hold:

- $M, \ell \mapsto (\tau', \bar{r}) \vdash S'$: The locations contained in store S' are equal to the location contained in S except for an additional location ℓ. Thus, the latter fact and $M \vdash S$ imply that $M, \ell \mapsto (\tau', \bar{r}) \vdash S'$ holds.
- $\forall(\ell' \mapsto (\tau'', \jmath)) \in M, \ell \mapsto (\tau', \bar{r}).R; M; \emptyset; \emptyset \vdash S'(\jmath)(\ell') : \tau''$ & $(\emptyset;\emptyset)$: immediate by $\forall(\ell' \mapsto (\tau'', \jmath)) \in M.R; M; \emptyset; \emptyset \vdash S(\jmath)(\ell') : \tau''$ & $(\emptyset;\emptyset)$ and $R; M, \ell \mapsto (\tau', \bar{r}); \emptyset; \emptyset \vdash v : \tau'$&$(\emptyset; \emptyset)$.

Case *T-AP*: The only operational rule that matches the shape of the application term is rule *E-A*:

- $\xi = \mathsf{seq}$
- $\mathsf{seq} \vdash \gamma = \gamma_1 \oplus \gamma_r$
- $\delta = \delta'', n \mapsto \sigma; \gamma$
- $\delta' = \delta'', n \mapsto \sigma; \gamma_r; \gamma_1$
- $\delta(n); S; ((\lambda x.\, e \text{ as } \tau)\ v)^{\mathsf{seq}} \to \delta(n)'; S; \mathsf{pop}_{\gamma_r}\ e[v/x]$

Expression typing: The proof for the typing preservation is similar to the previous proofs. By applying lemma A.10 to the derivation of the application term we obtain the following premises:

- $R; M; \emptyset; \emptyset \vdash \lambda x.\, e_1 \text{ as } \tau_x : \tau_1' \xrightarrow{\gamma_1' \to \gamma_2'} \tau_2'$&$(\gamma; \gamma)$: γ is equal to $\delta(n)$. by applying lemma A.10 to this derivation we obtain that:
  - $ok(\gamma_1; \gamma_2) \Rightarrow R; M; \emptyset; \emptyset, x : \tau_1 \vdash e_1 : \tau_2$&$(\gamma_1; \gamma_2)$
  - $\tau_x \equiv \tau_1 \xrightarrow{\gamma_1 \to \gamma_2} \tau_2$.
  - $\tau' \simeq \tau$: by inversion of this fact we obtain that $\gamma_1 \simeq \gamma_1'$, $\gamma_2 \simeq \gamma_2'$, $\tau_1' \simeq \tau_1$ and $\tau_2' \simeq \tau_2$.
- $\mathsf{seq} \vdash \gamma'' = \gamma_2' \oplus (\gamma \ominus \gamma_1')$: by applying lemma A.33 to $\mathsf{seq} \vdash \gamma'' = \gamma_2' \oplus (\gamma \ominus \gamma_1')$, $\gamma_1 \simeq \gamma_1'$ and $\gamma_2 \simeq \gamma_2'$, we have that $\mathsf{seq} \vdash \gamma'' = \gamma_2 \oplus (\gamma \ominus \gamma_1)$. By inversion of the latter derivation we obtain that $ok(\gamma_1; \gamma_2)$ holds. By applying lemma A.31 to $\mathsf{seq} \vdash \gamma = \gamma_1 \oplus \gamma_r$ and the latter fact we have that $\mathsf{seq} \vdash \gamma'' = \gamma_2 \oplus (\gamma_r \ominus \emptyset)$. $R; \emptyset \vdash \gamma_r$ holds as $R; \emptyset \vdash \gamma$ holds (premise of $\vdash R; M; \emptyset; \emptyset; \gamma; \gamma''$, which is immediate by lemma A.12) and the regions of γ_r is a subset of the of regions γ (by $\mathsf{seq} \vdash \gamma = \gamma_1 \oplus \gamma_r$). $\vdash R; M; \emptyset; \emptyset; \gamma; \gamma''$ also tells us that $ok(\gamma; \gamma'')$ holds. As mentioned earlier, γ_r is a subset of γ, thus $set(\gamma_r; \emptyset)$ holds.
- $R; M; \emptyset; \emptyset \vdash v : \tau_1'$&$(\gamma; \gamma)$: the application of lemma A.12 to the typing derivation of the application term gives us that $\vdash R; M; \emptyset; \emptyset; \gamma; \gamma''$. Thus, $\vdash R; M; \emptyset; \emptyset; \emptyset; \emptyset$ also holds. We can use the latter fact and the derivation of value v along with lemma A.13 to obtain $R; M; \emptyset; \emptyset \vdash v : \tau_1'$&$(\emptyset; \emptyset)$. The application of lemma A.34 to the latter derivation and $\tau_1' \simeq \tau_1$ gives us $R; M; \emptyset; \emptyset \vdash v : \tau_1$&$(\emptyset; \emptyset)$.

We have shown that $ok(\gamma_1; \gamma_2) \Rightarrow R; M; \emptyset; \emptyset, x : \tau_1 \vdash e_1 : \tau_2$&$(\gamma_1; \gamma_2)$ and $ok(\gamma_1; \gamma_2)$ holds, thus $R; M; \emptyset; \emptyset, x : \tau_1 \vdash e_1 : \tau_2$&$(\gamma_1; \gamma_2)$ holds. Lemma A.36 is applied to the typing derivation of v and e yields: $R; M; \emptyset; \emptyset \vdash e[v/x] : \tau_2$&$(\gamma_1; \gamma_2)$.

The application of rule *T-E* to $\tau_2' \simeq \tau_2$, $set(\gamma_r; \emptyset)$, $\mathsf{seq} \vdash \gamma'' = \gamma_2 \oplus (\gamma_r \ominus \emptyset)$, $R; \emptyset \vdash \gamma_r$ and $R; M; \emptyset; \emptyset \vdash e[v/x] : \tau_2$&$(\gamma_1; \gamma_2)$. gives us $R; M; \emptyset; \emptyset \vdash \mathsf{pop}_{\gamma_r}\ e[v/x] : \tau_2'$&$(\gamma_1; \gamma'')$.

Store typing: The application of lemma A.49 to the store typing assumption ($R; M \vdash \delta; S$), $\mathsf{seq} \vdash \gamma = \gamma_1 \oplus \gamma_r$ and $ok(\gamma; \gamma_1)$ (obtained by $ok(\gamma_1; \gamma_2)$ and $ok(\gamma; \gamma'')$ holds as shown earlier) implies that $R; M \vdash \delta'; S$ holds.

Lemma A.30 (Type Well-formedness) $R; M; \Delta; \Gamma \vdash e : \tau \& (\gamma; \gamma') \Rightarrow R; \Delta \vdash \tau$

Proof. Straightforward induction on the typing rules.

Lemma A.31 (Capability Addition Implication) $\xi \vdash \gamma = \gamma_1 \oplus \gamma_r \wedge \xi \vdash \gamma' = \gamma_2 \oplus (\gamma \ominus \gamma_1) \Rightarrow \xi \vdash \gamma' = \gamma_2 \oplus (\gamma_r \ominus \emptyset)$

Proof. By inversion of $\xi \vdash \gamma' = \gamma_2 \oplus (\gamma \ominus \gamma_1)$ we have that:

- $\xi \vdash \gamma = \gamma_1 \oplus \gamma_{r'}$: the capability addition rule *ES-C* is determistic, thus $\gamma_{r'} = \gamma_r$.
- $\xi \vdash \gamma' = \gamma_2 \oplus \gamma_{r'}$: similarly $\xi \vdash \gamma' = \gamma_2 \oplus \gamma_r$ holds. rule *ES-N* implies that $\xi \vdash \gamma_r = \emptyset \oplus \gamma_r$ holds.
- $\gamma'' = \mathit{live}(\gamma')$
- $\mathit{ok}(\gamma_1; \gamma_2)$: $\mathit{ok}(\gamma_2; \emptyset)$ trivially holds.
- $\xi = \mathsf{par} \Rightarrow \gamma_2 = \emptyset$

Rule *ESJ* is applied to the above facts to derive $\xi \vdash \gamma' = \gamma_2 \oplus (\gamma_r \ominus \emptyset)$.

Lemma A.32 (Effect Addition Implication) $\xi \vdash \gamma' = \gamma'_1 \oplus \gamma_r \wedge \gamma_1 \simeq \gamma'_1 \Rightarrow \xi \vdash \gamma = \gamma_1 \oplus \gamma_r$

Proof. If γ_1 is empty then the conclusion holds by rule *ES-N*. Otherwise,rule *ES-C* applies and gives us the following facts:

- $\gamma = \gamma', r^{\kappa} \triangleright \pi$
- $\gamma_1 = \gamma'', r''^{\kappa_1} \triangleright \pi''$: $\gamma_1 \simeq \gamma'_1$ implies that $\gamma'_1 = \gamma''_1, r'''^{\kappa_1} \triangleright \pi'''$, $r''' \simeq r''$, $\pi''' \simeq \pi''$ and $\gamma'' \simeq \gamma''_1$.
- $\xi \vdash \gamma', r^{\kappa_2} \triangleright \pi = \gamma'' \oplus \gamma_r$: we can apply the induction hypothesis to this fact and $\gamma'' \simeq \gamma''_1$ to obtain $\xi \vdash \gamma', r^{\kappa_2} \triangleright \pi = \gamma''_1 \oplus \gamma_r$.
- $\xi \vdash \kappa = \kappa_1 + \kappa_2$
- $\pi \simeq \pi''$: we use the fact that $\pi''' \simeq \pi''$ to obtain $\pi \simeq \pi'''$.
- $r \simeq r''$: we use the fact $r''' \simeq r''$, and that to obtain $r''' \simeq r$.

Thus, $\xi \vdash \gamma = \gamma'_1 \oplus \gamma_r$ holds by applying rule *E-SC* to the above facts.

Lemma A.33 (Effect Addition/Subtraction Implication) $\xi \vdash \gamma'' = \gamma'_2 \oplus (\gamma \ominus \gamma'_1) \wedge \gamma_1 \simeq \gamma'_1 \wedge \gamma_2 \simeq \gamma'_2 \Rightarrow \xi \vdash \gamma'' = \gamma_2 \oplus (\gamma \ominus \gamma_1)$

Proof. By inversion of the effect addition/subtraction assumption we have that

- $\xi \vdash \gamma = \gamma'_1 \oplus \gamma_r$: the application of lemma A.32 to $\xi \vdash \gamma = \gamma'_1 \oplus \gamma_r$ and $\gamma_1 \simeq \gamma'_1$ implies that $\xi \vdash \gamma = \gamma_1 \oplus \gamma_r$.
- $\xi \vdash \gamma' = \gamma'_2 \oplus \gamma_r$: the application of lemma A.32 to $\xi \vdash \gamma' = \gamma'_2 \oplus \gamma_r$ and $\gamma_2 \simeq \gamma'_2$ implies $\xi \vdash \gamma' = \gamma_2 \oplus \gamma_r$.
- $\gamma'' = \mathit{live}(\gamma')$
- $\xi = \mathsf{par} \Rightarrow \gamma_2 = \emptyset$
- $\mathit{ok}(\gamma'_1; \gamma'_2)$: $\mathit{ok}(\gamma'_1; \gamma'_2)$ trivially holds.

Lemma A.34 (Value Type Implication) $R; M; \Delta; \Gamma \vdash v : \tau\&(\gamma;\gamma) \wedge \tau \simeq \tau' \Rightarrow R; M; \Delta; \Gamma \vdash v : \tau'\&(\gamma;\gamma)$

Proof. Trivial proof by case analysis on the shape of value v.

Lemma A.35 (Polymorphic value substitution) $R, \bar{r}; \emptyset \vdash \gamma \wedge R, \bar{r}; M; \Delta, \rho; \emptyset \vdash f : \tau\&(\gamma;\gamma) \wedge$ fresh $r \wedge r \simeq r' \Rightarrow R, \bar{r}; M; \Delta; \emptyset \vdash f[r/\rho] : \tau[r'/\rho]\&(\gamma;\gamma)$

Proof. We proceed by performing a case analysis on the shape of f:

Case $f \equiv \lambda x.\, e$ as τ': By inversion (lemma A.10) of the assumption typing derivation we have that $ok(\gamma_1;\gamma_2) \Rightarrow R, \bar{r}; M; \Delta, \rho; \emptyset \vdash \lambda x.\, e$ as $\tau' : \tau\&(\gamma;\gamma)$ holds. If $ok(\gamma_1[r/\rho];\emptyset)$ does not hold then the proof is immediate. Otherwise, the application of lemma A.46 to the latter derivation, the fact that r is *fresh*, and $ok(\gamma_1[r/\rho];\emptyset)$ gives us $R, \bar{r}; M; \Delta; \emptyset \vdash (\lambda x.\, e$ as $\tau')[r/\rho] : \tau[r/\rho]\&(\gamma[r/\rho];\gamma[r/\rho])$. The assumption implies that γ is defined independently of ρ ($R, \bar{r}; \emptyset \vdash \gamma$). Thus, $R, \bar{r}; M; \Delta; \emptyset \vdash (\lambda x.\, e$ as $\tau')[r/\rho] : \tau[r/\rho]\&(\gamma;\gamma)$ also holds. By lemma A.10 we obtain the premises of the latter derivation. We can use rule *T-F*, the premises and the fact that $\tau[r/\rho] \simeq \tau[r'/\rho]$ ($r \simeq r'$) to derive $R, \bar{r}; M; \Delta; \emptyset \vdash (\lambda x.\, e$ as $\tau')[r/\rho] : \tau[r'/\rho]\&(\gamma;\gamma)$.

Case $f \equiv \Lambda\rho'.\, f'$: By inversion (lemma A.10) of the typing derivation of the assumption we have that $R, \bar{r}; M; \Delta, \rho, \rho'; \emptyset \vdash f' : \tau\&(\gamma;\gamma)$. We can use the induction hypothesis to derive that $R, \bar{r}; M; \Delta, \rho'; \emptyset \vdash f'[r/\rho] : \tau[r'/\rho]\&(\gamma;\gamma)$. The application of rule *T-RF* to the latter derivation yields $R, \bar{r}; M; \Delta; \emptyset \vdash \Lambda\rho'.\, f'[r/\rho] : \forall\rho'.\, \tau[r'/\rho]\&(\gamma;\gamma)$.

Lemma A.36 (Variable substitution) $R; M; \Delta; \Gamma, x : \tau_1 \vdash e : \tau_2 \,\&\, (\gamma_1;\gamma_2) \wedge R; M; \emptyset; \emptyset \vdash v : \tau_1 \,\&\, (\emptyset;\emptyset) \Rightarrow R; M; \Delta; \Gamma \vdash e[v/x] : \tau_2 \,\&\, (\gamma_1;\gamma_2)$

Proof. Straightforward induction on the expression typing derivation.

Lemma A.37 (Region substitution preserves ok) $\xi \vdash \gamma_3 = \gamma_2 \oplus \gamma_1 \wedge ok(\gamma_3;\emptyset) \wedge ok(\gamma_2;\emptyset) \wedge ok(\gamma_3[r_x/\rho];\emptyset) \Rightarrow ok(\gamma_2[r_x/\rho];\emptyset)$

Proof. If ρ does not exist in γ_2 then the proof is immediate by the assumption that $ok(\gamma_2;\emptyset)$. Otherwise we assume that ρ must exist in both γ_2 and γ_3 (by $\xi \vdash \gamma_3 = \gamma_2 \oplus \gamma_1$ we have that the domain of γ_2 is a subset of the domain of γ_3). Assume that r_x belongs in the domain of γ_3. This is a contradiction as the assumption $ok(\gamma_3[r_x/\rho];\emptyset)$ does not hold. We have mentioned that the regions of γ_2 are a subset of the regions of γ_3. Therefore, r_x does not belong in the domain of γ_2 either. By the assumption that $ok(\gamma_2;\emptyset)$ holds, the definition of predicate ok and the fact that r_x does not occur in the domain of γ_2 implies that $ok(\gamma_2[r_x/\rho];\emptyset)$ holds.

Lemma A.38 (Region substitution preserves $\oplus$) $\xi \vdash \gamma_3 = \gamma_2 \oplus \gamma_1 \Rightarrow \xi[r/\rho] \vdash \gamma_3[r/\rho] = \gamma_2[r/\rho] \oplus \gamma_1[r/\rho]$

Proof. If γ_1 is empty then rule *ES-N* implies that γ_3 equals γ_1. Thefore, $\xi[r/\rho] \vdash \gamma_1[r/\rho] = \emptyset \oplus \gamma_1[r/\rho]$ holds. It can be trivially shown that if $\xi \vdash \kappa = \kappa_1 + \kappa_2$, then for *any* r, ρ, $\xi[r/\rho] \vdash \kappa = \kappa_1 + \kappa_2$ also holds. If γ_1 is not empty then rule *ES-C* applies. By inversion of this rule we have that the following hold:

- $\pi \simeq \pi'$: $\pi[r/\rho] \simeq \pi'[r/\rho]$ is immediate.
- $r' \simeq r$: $r'[r/\rho] \simeq r[r/\rho]$ is immediate.
- $\xi \vdash \gamma_{31}, r'^{\kappa_2} \triangleright \pi = \gamma_{12} \oplus \gamma_1$: $\xi[r/\rho] \vdash (\gamma_{31}, r'^{\kappa_2} \triangleright \pi)[r/\rho] = \gamma_{12}[r/\rho] \oplus \gamma_1[r/\rho]$ holds by the induction hypothesis.

- $\gamma_3 = \gamma_{31}, r'^{\kappa_2} \triangleright \pi'$: $\gamma_3[r/\rho] = (\gamma_{31}, r'^{\kappa_2} \triangleright \pi')[r/\rho]$ is immediate.
- $\gamma_1 = \gamma_{12}, r^{\kappa_1} \triangleright \pi$: $\gamma_1[r/\rho] = (\gamma_{12}, r^{\kappa_1} \triangleright \pi)[r/\rho]$ is immediate.

By using rule *ES-C* we obtain that: $\xi[r/\rho] \vdash \gamma_3[r/\rho] = \gamma_2[r/\rho] \oplus \gamma_1[r/\rho]$

Lemma A.39 (Valid implication — $\oplus$) $\xi \vdash \gamma = \gamma_1 \oplus \gamma_r \wedge \mathit{valid}(\gamma_a; \gamma_b) \wedge \gamma_1 \subseteq \gamma_a \wedge \gamma_2 \subseteq \gamma_b \wedge \mathit{dom}(\gamma_2) \subseteq \mathit{dom}(\gamma_1) \Rightarrow \xi \vdash \gamma = \gamma_2 \oplus \gamma'_r$

Proof. Proof by induction on the structure of γ_2:

- $\emptyset$: immediate by rule *ES-N*.
- $\gamma_2 = \gamma_{21}, r^{\kappa_3} \triangleright \pi$: $\mathit{valid}(\gamma_a; \gamma_b)$, $\gamma_1 \subseteq \gamma_a$, $\gamma_2 \subseteq \gamma_b$ and $\mathit{dom}(\gamma_2) \subseteq \mathit{dom}(\gamma_1)$ imply that $\gamma_1 = \gamma_{12}, r^{\kappa_1} \triangleright \pi$ and $\mathit{is\_pure}(\kappa_1) \Leftrightarrow \mathit{is\_pure}(\kappa_3)$. By inversion of the assumption $\xi \vdash \gamma = \gamma_1 \oplus \gamma_r$ we have that:
  - $\gamma_3 = \gamma_{31}, r'^{\kappa_2} \triangleright \pi'$.
  - $\pi \simeq \pi'$ and $r' \simeq r$.
  - $\xi \vdash \kappa = \kappa_1 + \kappa_2$: $\xi \vdash \kappa = \kappa_3 + \kappa'_2$ also holds for some κ'_2 as a result of $\mathit{is\_pure}(\kappa_1) \Leftrightarrow \mathit{is\_pure}(\kappa_3)$.
  - $\xi \vdash \gamma_{31}, r'^{\kappa_2} \triangleright \pi = \gamma_{12} \oplus \gamma'_r$: $\xi \vdash \gamma_{31}, r'^{\kappa_2} \triangleright \pi = \gamma_{21} \oplus \gamma''_r$ holds by induction hypothesis.

  By applying rule *ES-C* to the latter facts we obtain that $\xi \vdash \gamma = \gamma_2 \oplus \gamma''_r$.

Lemma A.40 (Region substitution preserves $\oplus/\ominus$) $\xi \vdash \gamma_3 = \gamma_2 \oplus (\gamma \ominus \gamma_1) \wedge \mathit{ok}(\gamma; \gamma_3) \wedge \mathit{ok}(\gamma[r/\rho]; \emptyset) \wedge \mathit{valid}(\gamma_1; \gamma_2) \wedge \text{fresh } r \Rightarrow \xi[r/\rho] \vdash \gamma_3[r/\rho] = \gamma_2[r/\rho] \oplus (\gamma[r/\rho] \ominus \gamma_1[r/\rho])$

Proof. The assumption that $\mathit{ok}(\gamma; \gamma_3)$ holds implies that $\mathit{ok}(\gamma; \emptyset)$ and $\mathit{ok}(\gamma_3; \emptyset)$ hold. By inversion of the first assumption we obtain the following facts:

- $\mathit{ok}(\gamma_1; \gamma_2)$: this fact implies that $\mathit{ok}(\gamma_1; \emptyset)$ and $\mathit{ok}(\gamma_2; \emptyset)$ hold. The application of lemma A.37 to $\xi \vdash \gamma = \gamma_1 \oplus \gamma_r$, $\mathit{ok}(\gamma_1; \emptyset)$, $\mathit{ok}(\gamma; \emptyset)$ and $\mathit{ok}(\gamma[r/\rho]; \emptyset)$ implies that $\mathit{ok}(\gamma_1[r/\rho]; \emptyset)$ holds. The application of lemma A.39 to $\mathit{valid}(\gamma_1; \gamma_2)$, $\gamma_1 \subseteq \gamma_1$, $\gamma_2 \subseteq \gamma_2$ and $\mathit{dom}(\gamma_2) \subseteq \mathit{dom}(\gamma_1)$ (by inversion of $\mathit{valid}(\gamma_1; \gamma_2)$), we have that $\xi \vdash \gamma = \gamma_2 \oplus \gamma'_r$, for some γ'_r. The application of lemma A.37 to $\xi \vdash \gamma = \gamma_2 \oplus \gamma'_r$, $\mathit{ok}(\gamma_2; \emptyset)$, $\mathit{ok}(\gamma; \emptyset)$ and $\mathit{ok}(\gamma[r/\rho]; \emptyset)$ implies that $\mathit{ok}(\gamma_2[r/\rho]; \emptyset)$ holds. Thus, $\mathit{ok}(\gamma_1[r/\rho]; \gamma_2[r/\rho])$ holds.
- $\xi \vdash \gamma = \gamma_1 \oplus \gamma_r$: $\xi[r/\rho] \vdash \gamma[r/\rho] = \gamma_1[r/\rho] \oplus \gamma_r[r/\rho]$ immediate by the application of lemma A.38 to $\xi \vdash \gamma = \gamma_1 \oplus \gamma_r$.
- $\xi \vdash \gamma_a = \gamma_2 \oplus \gamma_r$: $\xi[r/\rho] \vdash \gamma_a[r/\rho] = \gamma_2[r/\rho] \oplus \gamma_r[r/\rho]$ immediate by the application of lemma A.38 to $\vdash \gamma_a = \gamma_2 \oplus \gamma_r$.
- $\gamma_3 = \mathit{live}(\gamma_a)$: it suffices to prove that $(\mathit{live}(\gamma_a))[r/\rho] = \mathit{live}(\gamma_a[r/\rho])$. This is trivial to show given that r is fresh (i.e. it does not belong in the domain of γ_a).
- $\xi = \mathsf{par} \Rightarrow \gamma_2 = \emptyset$: $\xi[r/\rho] = \mathsf{par} \Rightarrow \gamma_2[r/\rho] = \emptyset$ trivially holds.

Lemma A.41 (R Well-Formedness substitution) $R, \bar{r'}; \Delta, \rho \vdash r \Rightarrow R, \bar{r'}; \Delta \vdash r[r'/\rho]$

Proof. We proceed by performing a case analysis on r:

- $\imath@n$: By inversion of this derivation we have that $R, \bar{r'}; \Delta, \rho \vdash \imath$. The proof is completed by applying the induction hypothesis.

- $r \neq \imath@n$: By inversion of this derivation we have that $\bar{r} \in R, \bar{r'} \uplus \Delta, \rho$. Thus, $\bar{r}[r'/\rho] \in R, \bar{r'} \uplus \Delta$ also holds as $\bar{r}[r'/\rho]$ cannot be contained in Δ. Therefore, $R, \bar{r'}; \Delta \vdash r[r'/\rho]$ holds.

Lemma A.42 (Effect Well-formedness substitution) $R, \bar{r''}; \Delta, \rho \vdash \gamma \Rightarrow R, \bar{r''}; \Delta \vdash \gamma[r''/\rho]$

Proof. We proceed by performing a case analysis on γ:

- $\emptyset$: $R, \bar{r''}; \Delta \vdash \emptyset$ trivially holds.

  $R, \bar{r''}; \Delta, \rho \vdash \gamma', r^{\kappa} \triangleright \pi$: $R, \bar{r''}; \Delta \vdash \gamma'[r''/\rho]$ holds by the induction hypothesis. $R, \bar{r''}; \Delta \vdash r[r''/\rho]$ holds by lemma A.41. If $\pi = r'$, then $R, \bar{r''}; \Delta \vdash r'[r''/\rho]$ holds by lemma A.41.

Lemma A.43 (Type Context Well-formedness substitution) $R, \bar{r'}; \Delta, \rho \vdash \tau \wedge \text{fresh } r' \Rightarrow R, \bar{r'}; \Delta \vdash \tau[r'/\rho]$

Proof. We proceed by performing a case analysis on τ:

- b: $R, \bar{r'}; \Delta \vdash b$ trivially holds.

- $\langle\rangle$: $R, \bar{r'}; \Delta \vdash \langle\rangle$ trivially holds.

- $\texttt{rgn}(r)$: $R, \bar{r'}; \Delta \vdash r[r'/\rho]$ holds by lemma A.41.

- $\texttt{ref}(\tau', r)$: $R, \bar{r'}; \Delta \vdash r[r'/\rho]$ holds by lemma A.41. $R, \bar{r'}; \Delta \vdash \tau'[r'/\rho]$ holds by the induction hypothesis.

- $\forall \rho'.\, \tau'$: $R, \bar{r'}; \Delta, \rho' \vdash \tau'[r'/\rho]$ holds by the induction hypothesis.

- $\tau' \xrightarrow{\gamma_1 \to \gamma_2} \tau''$: $R, \bar{r'}; \Delta \vdash \tau'[r'/\rho]$ holds by the induction hypothesis. $R, \bar{r'}; \Delta \vdash \tau''[r'/\rho]$ holds by the induction hypothesis. $R, \bar{r'}; \Delta \vdash \gamma_1[r'/\rho]$ holds by lemma A.42. $R, \bar{r'}; \Delta \vdash \gamma_2[r'/\rho]$ holds by lemma A.42. We have that $\mathit{valid}(\gamma_1; \gamma_2)$ and we must prove that $\mathit{valid}(\gamma_1[r'/\rho]; \gamma_2[r'/\rho])$ holds. It suffices to show that:

  - if $(r[r'/\rho]^{\kappa} \triangleright \pi[r'/\rho]) \in \gamma_1[r'/\rho]$ and $(r[r'/\rho]^{\kappa'} \triangleright \pi'[r'/\rho]) \in \gamma_2[r'/\rho]$ for some r, then $\pi = \pi' \wedge (\mathit{is\_pure}(\kappa) \Leftrightarrow \mathit{is\_pure}(\kappa'))$: this is immediate by $(r^{\kappa} \triangleright \pi) \in \gamma_1$ and $(r^{\kappa'} \triangleright \pi') \in \gamma_2$, then $\pi = \pi' \wedge (\mathit{is\_pure}(\kappa) \Leftrightarrow \mathit{is\_pure}(\kappa'))$, which can be obtained by inversion of $\mathit{valid}(\gamma_1; \gamma_2)$.
  - $\mathit{live}(\gamma_1[r'/\rho]) = \gamma_1[r'/\rho]$ and $\mathit{live}(\gamma_2[r'/\rho]) = \gamma_2[r'/\rho]$: immediate by inversion of $\mathit{valid}(\gamma_1; \gamma_2)$, the definition of substitution and the fact that r' is *fresh*.
  - $\mathit{dom}(\gamma_2) \subseteq \mathit{dom}(\gamma_1)$: $\mathit{dom}(\gamma_2[r'/\rho]) \subseteq \mathit{dom}(\gamma_1[r'/\rho])$ is immediate.

Lemma A.44 (Variable Context Well-formedness substitution) $R, \bar{r}; \Delta, \rho \vdash \Gamma \wedge \text{fresh } r \Rightarrow R, \bar{r}; \Delta \vdash \Gamma[r/\rho]$

Proof. We proceed by performing a case analysis on Γ:

- $\emptyset$: $R'; \Delta \vdash \emptyset$ trivially holds.

  $R; \Delta \vdash \Gamma', x : \tau$: $R'; \Delta \vdash \Gamma'[r/\rho]$ holds by the induction hypothesis. $R'; \Delta \vdash \tau[r/\rho]$ holds by lemma A.43 and the fact that r is *fresh*.

Lemma A.45 (Well-formedness substitution) $R, \bar{r}; M; \Delta, \rho; \Gamma; \gamma_1; \gamma_2 \wedge$ fresh $r \wedge ok(\gamma_1[r/\rho]; \gamma_2[r/\rho]) \Rightarrow R, \bar{r}; M; \Delta; \Gamma[r/\rho]; \gamma_1[r/\rho]; \gamma_2[r/\rho]$

Proof. By inversion of the first typing context and effect well-formedness assumption we have that

- $R, \bar{r} \vdash M$
- $R, \bar{r}; \Delta, \rho \vdash \Gamma$: $R, \bar{r}; \Delta \vdash \Gamma[r/\rho]$ immediate by lemma A.44 and the fact that r is *fresh*.
- $R, \bar{r}; \Delta, \rho \vdash \gamma_1$: $R, \bar{r}; \Delta \vdash \gamma_1[r/\rho]$ immediate by lemma A.42.
- $R, \bar{r}; \Delta, \rho \vdash \gamma_2$: $R, \bar{r}; \Delta \vdash \gamma_2[r/\rho]$ immediate by lemma A.42.
- $ok(\gamma_1; \gamma_2)$: $ok(\gamma_1[r/\rho]; \gamma_2[r/\rho])$ immediate from the assumption.

Lemma A.46 (Region substitution) $R, \bar{r}; M; \Delta, \rho; \Gamma \vdash e : \tau$ & $(\gamma_1;\gamma_2) \wedge$ fresh $r \wedge ok(\gamma_1[r/\rho]; \emptyset) \Rightarrow R, \bar{r}; M; \Delta; \Gamma[r/\rho] \vdash e[r/\rho] : \tau[r/\rho]$ & $(\gamma_1[r/\rho];\gamma_2[r/\rho])$

Proof. Proof by induction on the expression typing derivation.

Case *T-I*: by applying lemma A.10 to the derivation of e we have that $\gamma_1 = \gamma_2$. Thus, $ok(\gamma_2[r/\rho]; \emptyset)$ is immediate from $\gamma_1 = \gamma_2$ and the assumption that $ok(\gamma_1[r/\rho]; \emptyset)$ holds. The application of lemma A.45 to the latter derivation, the fact that r is *fresh* and $ok(\gamma_1[r/\rho]; \gamma_2[r/\rho])$ implies that $\vdash R, \bar{r}; M; \Delta; \Gamma[r/\rho]; \gamma_1[r/\rho]; \gamma_2[r/\rho]$ holds. The proof for this case is completed by applying rule *T-I*.

Case *T-U*, by applying lemma A.10 to the derivation of e we have that $\gamma_1 = \gamma_2$. Thus, $ok(\gamma_2[r/\rho]; \emptyset)$ is immediate from $\gamma_1 = \gamma_2$ and the assumption that $ok(\gamma_1[r/\rho]; \emptyset)$ holds. The application of lemma A.45 to the latter derivation, the fact that r is *fresh* and $ok(\gamma_1[r/\rho]; \gamma_2[r/\rho])$ implies that $\vdash R, \bar{r}; M; \Delta; \Gamma[r/\rho]; \gamma_1[r/\rho]; \gamma_2[r/\rho]$ holds. The proof for this case is completed by applying rule *T-U*.

Case *T-R*: the application of lemma A.10 to the derivation of e yields:

- $\gamma_1 = \gamma_2$: $ok(\gamma_2[r/\rho]; \emptyset)$ is immediate from $\gamma_1 = \gamma_2$ and the assumption that $ok(\gamma_1[r/\rho]; \emptyset)$ holds. the application of lemma A.45 to the latter derivation, the fact that r is *fresh* and $ok(\gamma_1[r/\rho]; \gamma_2[r/\rho])$ implies that $\vdash R, \bar{r}; M; \Delta; \Gamma[r/\rho]; \gamma_1[r/\rho]; \gamma_2[r/\rho]$ holds.
- $\vdash R, \bar{r}; M; \Delta, \rho, \Gamma; \gamma; \gamma$: we have already shown that $\vdash R, \bar{r}; M; \Delta; \Gamma[r/\rho]; \gamma[r/\rho]; \gamma[r/\rho]$ holds.
- $R; \Delta \vdash \imath$: $R, \bar{r}; \Delta \vdash \imath[r/\rho]$ holds by lemma A.41.
- $r' \simeq \imath$: $r'[r/\rho] \simeq \imath[r/\rho]$ trivially holds.

The proof for this case is completed by applying rule *T-R* to the derived facts.

Case *T-L*: the application of lemma A.10 to the derivation of e yields:

- $\gamma_1 = \gamma_2$: $ok(\gamma_2[r/\rho]; \emptyset)$ is immediate from $\gamma_1 = \gamma_2$ and the assumption that $ok(\gamma_1[r/\rho]; \emptyset)$ holds. The application of lemma A.45 to the latter derivation, the fact that r is *fresh* and $ok(\gamma_1[r/\rho]; \gamma_2[r/\rho])$ implies that $\vdash R, \bar{r}; M; \Delta; \Gamma[r/\rho]; \gamma_1[r/\rho]; \gamma_2[r/\rho]$ holds.

– $\vdash R, \bar{r}; M; \Delta, \rho, \Gamma; \gamma; \gamma$: we have already shown that $\vdash R, \bar{r}; M; \Delta; \Gamma[r/\rho]; \gamma[r/\rho]; \gamma[r/\rho]$ holds.

– $(\ell \mapsto (\tau, \imath)) \in M$

– $\tau' \simeq \mathtt{ref}(\tau, \imath)$: $\tau'[r/\rho] \simeq \mathtt{ref}(\tau, \imath)[r/\rho]$ trivially holds.

The proof for this case is completed by applying rule *T-L* to the derived facts.

Case *T-V*: the application of lemma A.10 to the derivation of e yields:

– $\gamma_1 = \gamma_2$: $ok(\gamma_2[r/\rho]; \emptyset)$ is immediate from $\gamma_1 = \gamma_2$ and the assumption that $ok(\gamma_1[r/\rho]; \emptyset)$ holds. The application of lemma A.45 to the latter derivation, the fact that r is *fresh* and $ok(\gamma_1[r/\rho]; \gamma_2[r/\rho])$ implies that $\vdash R, \bar{r}; M; \Delta; \Gamma[r/\rho]; \gamma_1[r/\rho]; \gamma_2[r/\rho]$ holds.

– $\vdash R, \bar{r}; M; \Delta, \rho, \Gamma; \gamma; \gamma$: we have already shown that $\vdash R, \bar{r}; M; \Delta; \Gamma[r/\rho]; \gamma[r/\rho]; \gamma[r/\rho]$ holds.

– $(x : \tau) \in \Gamma$: $(x : \tau[r/\rho]) \in \Gamma[r/\rho]$ trivially holds.

– $\tau \simeq \tau'$: $\tau[r/\rho] \simeq \tau'[r/\rho]$ trivially holds.

The proof for this case is completed by applying rule *T-V* to the derived facts.

Case *T-F*: the application of lemma A.10 to the derivation of e yields:

– $\gamma_1 = \gamma_2$: $ok(\gamma_2[r/\rho]; \emptyset)$ is immediate from $\gamma_1 = \gamma_2$ and the assumption that $ok(\gamma_1[r/\rho]; \emptyset)$ holds. The application of lemma A.45 to the latter derivation, the fact that r is *fresh* and $ok(\gamma_1[r/\rho]; \gamma_2[r/\rho])$ implies that $\vdash R, \bar{r}; M; \Delta; \Gamma[r/\rho]; \gamma_1[r/\rho]; \gamma_2[r/\rho]$ holds.

– $\vdash R, \bar{r}; M; \Delta, \rho, \Gamma; \gamma; \gamma$: we have already shown that $\vdash R, \bar{r}; M; \Delta; \Gamma[r/\rho]; \gamma[r/\rho]; \gamma[r/\rho]$ holds.

– $R, \bar{r}; \Delta, \rho \vdash \tau$: lemma A.43 and the fact that r is *fresh* imply that $R, \bar{r}; \Delta \vdash \tau[r/\rho]$ holds.

– $\tau' \simeq \tau$: $\tau'[r/\rho] \simeq \tau[r/\rho]$ trivially holds.

– $\tau \equiv \tau_1 \xrightarrow{\gamma_a \to \gamma_b} \tau_2$: the function type after substitution is
$\tau[r/\rho] \equiv \tau_1[r/\rho] \xrightarrow{\gamma_a[r/\rho] \to \gamma_b[r/\rho]} \tau_2[r/\rho]$.

– $set(\gamma_a; \gamma_b)$: $set(\gamma_a[r/\rho], \gamma_b[r/\rho])$ trivially holds as r is *fresh*.

– $ok(\gamma_a; \gamma_b) \Rightarrow R, \bar{r}; M; \Delta, \rho; \Gamma, x : \tau_1 \vdash e : \tau_2 \,\&\, (\gamma_a; \gamma_b)$: Let us assume that $ok(\gamma_a[r/\rho]; \gamma_b[r/\rho])$ holds, then,
$R, \bar{r}; M; \Delta; (\Gamma, x : \tau_1)[r/\rho] \vdash e[r/\rho] : \tau_2[r/\rho] \,\&\, (\gamma_a[r/\rho]; \gamma_b[r/\rho])$ holds by the induction hypothesis.

The proof for this case is completed by applying rule *T-F* to the derived facts.

Case *T-AP*: the application of lemma A.10 to the derivation of e yields:

– $R, \bar{r}; M; \Delta, \rho; \Gamma \vdash e_1 : \tau_1 \xrightarrow{\gamma_a \to \gamma_b} \tau_2 \,\&\, (\gamma_1; \gamma_3)$: By applying lemma A.30 to the derivation of e_1 we obtain that $R, \bar{r}; \Delta, \rho \vdash \tau_1 \xrightarrow{\gamma_a \to \gamma_b} \tau_2$. By inversion of the latter fact $valid(\gamma_a; \gamma_b)$ holds.
$R, \bar{r}; M; \Delta; \Gamma[r/\rho] \vdash e_1[r/\rho] : (\tau_1 \xrightarrow{\gamma_a \to \gamma_b} \tau_2)[r/\rho] \,\&\, (\gamma_1[r/\rho]; \gamma_3[r/\rho])$ holds by the induction hypothesis, the assumption that r is *fresh* and $ok(\gamma_1[r/\rho]; \emptyset)$. By applying lemma A.12 to the latter fact and performing inversion to the resulting well-formedness derivation we have that $ok(\gamma_3[r/\rho]; \emptyset)$.

– $\mathsf{par} \Rightarrow \tau_2 = \langle\rangle$: $\mathsf{par} \Rightarrow \tau_2[r/\rho] = \langle\rangle$ trivially holds.

– $R, \bar{r}; M; \Delta, \rho; \Gamma \vdash e_2 : \tau_1 \,\&\, (\gamma_3;\gamma_4)$:
$R, \bar{r}; M; \Delta; \Gamma[r/\rho] \vdash e_2[r/\rho] : \tau_1[r/\rho] \,\&\, (\gamma_3[r/\rho];\gamma_4[r/\rho])$ holds by the induction hypothesis and $ok(\gamma_3[r/\rho]; \emptyset)$. By applying lemma A.12 to the latter fact and performing inversion to the resulting well-formedness derivation we have that $ok(\gamma_4[r/\rho]; \emptyset)$.

– $\xi \vdash \gamma_2 = \gamma_b \oplus (\gamma_4 \ominus \gamma_a)$: we have shown that $ok(\gamma_4[r/\rho]; \emptyset)$ and $valid(\gamma_a; \gamma_b)$. $\xi \vdash \gamma_2[r/\rho] = \gamma_b[r/\rho] \oplus (\gamma_4[r/\rho] \ominus \gamma_a[r/\rho])$ is immediate by lemma A.40, $ok(\gamma_4[r/\rho]; \emptyset)$, $valid(\gamma_a; \gamma_b)$ and $ok(\gamma_4; \gamma_2)$, which can be obtained by applying lemma A.12 to the typing derivation of e, and the fact that r is *fresh*.

Case *T-CP*, *T-RP*, *T-NG*, *T-NR*, *T-D*, *T-RF, T-E, T-A*: We can perform similar reasoning to prove the remaining cases. The key point is to prove in the remaining cases that $(live(\gamma_x))[r/\rho] = live(\gamma_x[r/\rho])$, where γ_x is the effect of interest. The proof can be summarized as follows:

– ρ is a leaf element in γ_x: liveness for this regions is unaffected as its parents are unaffected.

– ρ is an intermediate node in γ_x: assuming that there exist an immediate and *live* descendant r', then its parent annotation is ρ. Thus after substitution r' will still be *live*.

Lemma A.47 (Store typing preservation — Push Helper 2) $\sigma; \gamma \vdash \delta \wedge ok(\gamma; \gamma_1) \wedge \mathsf{seq} \vdash \gamma = \gamma_1 \oplus \gamma_r \Rightarrow \sigma; \gamma_r; \gamma_1 \vdash \delta$

Proof. Proof by induction on the shape of δ:

- $\emptyset$: it must be shown that for all $\imath$ that belong in the domain of $dom(\sigma; \gamma_r; \gamma_1)$, an given that $\sigma; \gamma_r; \gamma_1 \simeq \sigma_1; \gamma_x, \imath^{\kappa} \triangleright \pi + \sigma_2$, $rg(\kappa) > 0$ and $is\_pure(\kappa)$ hold, then both $zero\_pure(\sigma_1)$ and $\imath \notin dom(\sigma_2; \gamma_x)$ hold. We proceed by performing a case analysis as follows:

  – $\imath$ does belong in the domain of $\emptyset; \gamma$: stack $\sigma_1 \simeq \sigma$ and $\sigma_2 = \emptyset$. By inversion of $\sigma; \gamma \vdash \emptyset$ (obtained by $\sigma; \gamma \vdash \delta$) we have that $zero\_pure(\sigma)$ (or $zero\_pure(\sigma_1)$) and $\imath \notin dom(\emptyset; \gamma_x)$. The assumption that $ok(\gamma; \gamma_1)$ holds implies that $\imath$, which is associated with a pure capability, belongs in the domain of either $\emptyset; \gamma_r$ or $\emptyset; \gamma_1$ with a pure and positive capability. If it does belong in the domain of $\emptyset; \gamma_r$, then $\gamma_r \simeq \gamma_x, \imath^{\kappa} \triangleright \pi$ and $\imath \notin dom(\emptyset; \gamma_1; \gamma_x)$ trivially hold from the above fact, the assumption that $ok(\gamma; \gamma_1)$ and $\mathsf{seq} \vdash \gamma = \gamma_1 \oplus \gamma_r$. Otherwise, $\gamma_1 \simeq \gamma_x, \imath^{\kappa} \triangleright \pi$, $zero\_pure(\sigma; \gamma_r)$ and $\imath \notin dom(\emptyset; \gamma_x)$ trivially hold from the above facts and the assumption that $ok(\gamma; \gamma_1)$.

  – $\imath$ does not belong in the domain of $\emptyset; \gamma$: stack $\sigma_2 \simeq \sigma_2'; \gamma_r; \gamma_1$ for some stack σ_2' (the assumption that $\mathsf{seq} \vdash \gamma = \gamma_1 \oplus \gamma_r$ holds implies that $dom(\emptyset; \gamma_r) \subseteq dom(\emptyset; \gamma)$ and $dom(\emptyset; \gamma_1) \subseteq dom(\emptyset; \gamma)$). By inversion of $\sigma; \gamma \vdash \emptyset$ (obtained by $\sigma; \gamma \vdash \delta$) we have that $zero\_pure(\sigma_1)$ and $\imath \notin dom(\sigma_2)$. (or $\imath \notin dom(\sigma_2'; \gamma; \gamma_x)$). The latter fact implies that $\imath \notin dom(\sigma_2'; \gamma_r; \gamma_1; \gamma_x)$.

Case $\delta_1, n_1 \mapsto \sigma_1$: it suffices to show that

  – $\sigma; \gamma_r; \gamma_1 \vdash \delta_1$: by inversion of $\sigma; \gamma \vdash \delta$ we obtain $\sigma; \gamma \vdash \delta_1$. The proof for this case is completed by applying the induction hypothesis.

  – Given that $is\_accessible(\sigma; \gamma_r; \gamma_1, \imath)$, then prove that $\neg is\_accessible(\sigma_1, \imath)$ for all regions $\imath$ that belong in the domain of $\sigma; \gamma_r; \gamma_1$: the capability addition assumption implies that $is\_accessible(\sigma; \gamma_r; \gamma_1, \imath)$ implies $is\_accessible(\sigma; \gamma, \imath)$. By inversion of $\sigma; \gamma \vdash \delta$ and the latter fact, and the fact that $\imath$ belongs in the domain of $\emptyset; \gamma$ (capability addition assumption), we obtain that $is\_accessible(\sigma; \gamma, \imath)$ and thus $\neg is\_accessible(\sigma_1, \imath)$.

Lemma A.48 (Store typing preservation — Push Helper 1) $\vdash \delta, n \mapsto \sigma; \gamma \wedge ok(\gamma; \gamma_1) \wedge \mathsf{seq} \vdash \gamma = \gamma_1 \oplus \gamma_r \Rightarrow \vdash \delta, n \mapsto \sigma; \gamma_r; \gamma_1$

Proof. It suffices to prove that:

- $\vdash \delta$: immediate by inversion inversion of the assumption $\vdash \delta, n \mapsto \sigma; \gamma$.
- $\sigma; \gamma_r; \gamma_1 \vdash \delta$: by inversion of the assumption $\vdash \delta, n \mapsto \sigma; \gamma$ we have that $\sigma; \gamma \vdash \delta$. The application of lemma A.47 completes the proof for this case.

Lemma A.49 (Store typing preservation — Push) $R; M \vdash \delta; S \wedge ok(\gamma; \gamma_1) \wedge \mathsf{seq} \vdash \gamma = \gamma_1 \oplus \gamma_r \wedge \delta = \delta'', n \mapsto \sigma; \gamma \Rightarrow \delta' = \delta'', n \mapsto \sigma; \gamma_r; \gamma_1 \wedge R; M \vdash \delta'; S$

Proof. The store typing assumption implies that the following hold:

- $R; M \vdash S$
- $R \vdash \delta$: immediate by the fact that the regions of γ_1 and γ_r are a subset of γ (by the effect addition assumption).
- $\vdash \delta$: by inversion of $R; M \vdash \delta; S$ we have that $\vdash \delta'', n \mapsto \sigma; \gamma$. The proof is immediate by the application of lemma A.48 to the latter fact, $ok(\gamma; \gamma_1)$ and $\mathsf{seq} \vdash \gamma = \gamma_1 \oplus \gamma_r$.

Lemma A.50 (Store typing preservation — Pop Helper 2) $\sigma; \gamma_r; \gamma_1 \vdash \delta \wedge ok(\gamma_r; \gamma_1) \wedge \mathsf{seq} \vdash \gamma = \gamma_1 \oplus (\gamma_r \ominus \emptyset) \Rightarrow \sigma; \gamma \vdash \delta$

Proof. Proof by induction on the shape of δ:

- $\emptyset$: it must be shown that for all $\imath$ that belong in the domain of $dom(\sigma; \gamma)$, an given that $\sigma; \gamma \simeq \sigma_1; \gamma_x, \imath^{\kappa} \triangleright \pi + \sigma_2, rg(\kappa) > 0$ and $is\_pure(\kappa)$ hold, then both $zero\_pure(\sigma_1)$ and $\imath \notin dom(\sigma_2; \gamma_x)$ hold. By inversion of $\sigma; \gamma_r; \gamma_1 \vdash \emptyset$ (obtained by the assumption $\sigma; \gamma_r; \gamma_1 \vdash \delta$) we have that $\sigma; \gamma_r; \gamma_1 \simeq \sigma_3; \gamma_y, \imath^{\kappa'} \triangleright \pi + \sigma_4, zero\_pure(\sigma_3)$ and $\imath \notin dom(\sigma_2; \gamma_y)$.

  Region $\imath$ cannot be contained as a positive and pure effect in both γ_r and γ_1 as the effect addition assumption would not hold. We proceed by performing a case analysis as follows:

  - $\imath$ does belong in the domain of γ_1: $\sigma_1 \simeq \sigma_3 \simeq \sigma; \gamma_r$, $\sigma_2 = \sigma_4 = \emptyset$, and $\gamma_1 \simeq \gamma_y, \imath^{\kappa'} \triangleright \pi$. Thus, we have that $zero\_pure(\sigma_3, \imath)$ (or $zero\_pure(\sigma, \imath)$) and $\imath \notin dom(\emptyset; \gamma_y)$. The effect addition assumption and the assumption that $ok(\gamma_r; \gamma_1)$ imply that $ok(\gamma)$ holds. Thus, $\imath \notin dom(\emptyset; \gamma_x)$.
  - $\imath$ does belong in the domain of γ_r: $\sigma_1 \simeq \sigma_3 \simeq \sigma$, $\sigma_2 \simeq \emptyset$, $\sigma_4 \simeq \emptyset; \gamma_1$, and $\gamma_r \simeq \gamma_y, \imath^{\kappa'} \triangleright \pi$. Thus, we have that $zero\_pure(\sigma_3, \imath)$ (or $zero\_pure(\sigma, \imath)$) and $\imath \notin dom(\emptyset; \gamma_1; \gamma_y)$. The effect addition assumption and the assumption that $ok(\gamma_r; \gamma_1)$ imply that $ok(\gamma)$ holds. Thus, $\imath \notin dom(\emptyset; \gamma_x)$.
  - $\imath$ does not belong in the domain of $\emptyset; \gamma_r; \gamma_1$: $\sigma_2 \simeq \sigma_2'; \gamma_r; \gamma_1$, $\sigma_4 \simeq \sigma_2'; \gamma$, for some stack $\sigma_2', \sigma_3; \gamma_y, \imath^{\kappa'} \triangleright \pi \simeq \sigma_1; \gamma_x, \imath^{\kappa} \triangleright \pi$. Thus, we have that $zero\_pure(\sigma_3, \imath)$ (or $zero\_pure(\sigma_1, \imath)$) and $\imath \notin dom(\sigma_2'; \gamma_1; \gamma_r; \gamma_y)$. The effect addition assumption implies that $dom(\emptyset; \gamma) \subseteq dom(\emptyset; \gamma_r)$. Thus, $\imath \notin dom(\sigma_2'; \gamma; \gamma_x)$.

Case $\delta_1, n_1 \mapsto \sigma_1$: it suffices to show that

- $\sigma; \gamma \vdash \delta_1$: by inversion of $\sigma; \gamma_r; \gamma_1 \vdash \delta$ we obtain that $\sigma; \gamma_r; \gamma_1 \vdash \delta_1$. The proof for this case is completed by applying the induction hypothesis.

– Given that *is_accessible*$(\sigma;\gamma,\imath)$, then prove that $\neg$*is_accessible*$(\sigma_1,\imath)$ for all regions $\imath$ that belong in the domain of $\sigma;\gamma_r;\gamma_1$: the capability addition assumption implies that *is_accessible*$(\sigma;\gamma,\imath)$ implies *is_accessible*$(\sigma;\gamma_r;\gamma_1,\imath)$. By inversion of $\sigma;\gamma_r;\gamma_1 \vdash \delta$ and the latter fact, and the fact that $\imath$ belongs in the domain of $\emptyset;\gamma_r;\gamma_1$ (capability addition assumption), we obtain that *is_accessible*$(\sigma;\gamma_r;\gamma_1,\imath)$ and thus, $\neg$*is_accessible*$(\sigma_1,\imath)$.

Lemma A.51 (Store typing preservation — Pop Helper 1) $\vdash \delta, n \mapsto \sigma;\gamma_r;\gamma_1 \wedge ok(\gamma_r;\gamma_1) \wedge \mathsf{seq} \vdash \gamma = \gamma_1 \oplus (\gamma_r \ominus \emptyset) \Rightarrow \vdash \delta, n \mapsto \sigma;\gamma$

Proof. It suffices to prove that:

- $\vdash \delta$: immediate by inversion inversion of the assumption $\vdash \delta, n \mapsto \sigma;\gamma_r;\gamma_1$.

- $\sigma;\gamma_r;\gamma_1 \vdash \delta$: by inversion of the assumption $\vdash \delta, n \mapsto \sigma;\gamma_r;\gamma_1$ we have that $\sigma;\gamma_r;\gamma_1 \vdash \delta$. The application of lemma A.50 completes the proof for this case.

Lemma A.52 (Store typing preservation — Pop) $R;M \vdash \delta;S \wedge ok(\gamma;\gamma') \wedge \mathsf{seq} \vdash \gamma'' = \gamma' \oplus (\gamma \ominus \emptyset) \wedge \delta = \delta'', n \mapsto \sigma;\gamma;\gamma' \wedge \delta' = \delta'', n \mapsto \sigma;\gamma'' \Rightarrow R;M \vdash \delta';S$

Proof. The store typing assumption implies that the following hold:

- $R;M \vdash S$

- $R \vdash \delta$: immediate as the regions of γ'' are a subset of γ' and γ (by the effect addition assumption).

- $\vdash \delta'$: by inversion of $R;M \vdash \delta;S$ we have that $\vdash \delta'', n \mapsto \sigma;\gamma;\gamma'$. The proof is immediate by the application of lemma A.51 to the latter fact, $ok(\gamma;\gamma')$ and $\mathsf{seq} \vdash \gamma'' = \gamma' \oplus (\gamma \ominus \emptyset)$.

Lemma A.53 (Progress — Program) Let $S;T$ be a closed well-typed configuration with $R;M \vdash \delta;S;T$, then $S;T$ is not stuck ($\vdash S;T$).

Proof. In order to prove that the configuration is not stuck, we need to prove that each of the executing threads can either perform a step or $nolock$ predicate holds for it. Without loss of generality, we choose a random thread from the thread list, namely $n:e$ and show that it is not stuck. Thus, $T = T_1, n:e$ for some T_1. We use lemma A.3 to obtain $R;M \vdash \delta;S;T_1, n:e$. By inversion of the configuration typing derivation we have that $R;M;\delta \vdash T_1, n:e$ and $R;M \vdash \delta;S$. By inversion of the former derivation we obtain that $R;M;\emptyset;\emptyset \vdash e : \langle\rangle\&(\gamma;\emptyset)$, *pops*$(\sigma;\gamma:e)$, $\delta = \delta_1, n \mapsto \sigma;\gamma$ and $R;M;\delta_1 \vdash T_1$.

If e is a value then lemma A.56 tells us that e is $()$. *pops*$(\sigma;\gamma : ())$ implies that $\sigma;\gamma \equiv \emptyset;\emptyset$. We already have that $T = T_1, n:e$ for some T_1. Thus, a single step can be performed via rule *E-T*. Otherwise, e is not a value. The application of lemma A.55 to the latter fact and the typing derivation of e implies that $\exists e_1, E.\ E[e_1] = e$ and *redex*(e_1). Thus, $R;M;\emptyset;\emptyset \vdash E[e_1] : \langle\rangle\&(\gamma;\emptyset)$ is also well-typed. The application of lemma A.54 to *redex*(e_1), $\delta = \delta_1, n \mapsto \sigma;\gamma$, the typing derivation of $E[e_1]$, *pops*$(\sigma;\gamma : E[e_1])$ and $R;M \vdash \delta;S$ implies that one of the following holds:

- *nolock*(δ, n, e_1): the proof is trivially completed as *nolock*$(\delta, n, E[e_1])$ also holds.

- $\vdash \delta[n \mapsto \sigma'] \wedge \exists\, \delta', S', e'.\ \sigma;S;e_1 \rightarrow \sigma';S';e'$: A single step can be performed via rule *E-S*.

- $\exists e_2, v, \tau, \gamma_1.\ e_1 \equiv (\lambda x.\, e_2\ \mathsf{as}\ \tau\ v)^{\mathsf{par}}$: A step can only be performed via rule *E-SN*. Thus, it suffices to show that the premises of that rule are satisfied:

- $T_1, n : e = T$: we have shown that this property holds.
- fresh n': it is possible to find a thread identifier n' that has never been used previously.
- $v_1 \equiv \lambda x.\, e_2$ as τ: immediate from the assumption.
- $e_1 \equiv (v_1\ v)^{\mathsf{par}}$: immediate from the assumption.
- $e' \equiv (v_1\ v)^{\mathsf{seq}}$: the language syntax allows us to formulate this term.
- $\delta = \delta_1, n \mapsto \sigma; \gamma$: we have shown that this property holds.
- $\mathsf{par} \vdash \gamma' = \emptyset \oplus (\gamma \ominus \gamma_1)$: The application of lemma A.11 to the typing derivation of $E[e_1]$ implies that $R; M; \emptyset; \emptyset \vdash e_1 : \tau \& (\gamma; \gamma'')$ for some τ and γ''. By applying lemma A.10 to the latter derivation we obtain that v_1 is a well-typed abstraction and $\mathsf{par} \vdash \gamma'' = \emptyset \oplus (\gamma \ominus \gamma_1')$, where γ_1' is the effect embedded in the type assigned to v_1. By applying lemma A.10 to the typing derivation of v_1 we have that if γ_1 the type ascribed on v_1, then $\gamma_1 \simeq \gamma_1'$. The application of lemma A.33 to the latter facts give us that $\mathsf{par} \vdash \gamma'' = \emptyset \oplus (\gamma \ominus \gamma_1)$. The effect addition derivation is deterministic thus $\gamma'' = \gamma'$.
- $\delta' = \delta_1, n \mapsto \sigma; \gamma', n' \mapsto \emptyset; \gamma_1$: the syntax of δ allows us to formulate this context.

Lemma A.54 (Progress — Expressions) *redex*$(e) \wedge \delta = \delta_1, n \mapsto \sigma; \gamma \wedge R; M; \emptyset; \emptyset \vdash E[e] : \langle\rangle$ & $(\gamma;\emptyset) \wedge$ *pops*$(\sigma; \gamma : E[e]) \wedge R; M \vdash \delta; S \Rightarrow$ *nolock*$(\delta, n, e) \vee (\vdash \delta[n \mapsto \sigma'] \wedge \exists\, \sigma', S', e'.\, (\sigma; \gamma); S; e \rightarrow \sigma'; S'; e') \vee (\exists e_1, \tau, v.\, e \equiv (\lambda x.\, e_1$ as $\tau\ v)^{\mathsf{par}})$

Proof. The application of lemma A.11 to the typing derivation of $E[e]$ gives us that $R; M; \emptyset; \emptyset \vdash e : \tau$ & $(\gamma;\gamma')$ for some γ' and τ. We proceed by perfoming induction on the typing derivation of e.

Case *T-I*, *T-U*, *T-F*, *T-L*, *T-R*, *T-RF*,*T-V*: this is a contradiction as e should be a value, but we have assumed that e is a *redex*.

Case *T-E*: The conclusion of rule *T-E* implies that shape of e is of the form $\mathsf{pop}_{\gamma_r}\ e'$. We have assumed that e is a *redex*, thus e is of the form $\mathsf{pop}_{\gamma_r}\ v$. The assumption that *pops*$(\sigma; \gamma : \mathsf{pop}_{\gamma_r}\ v)$ implies that $\sigma = \sigma'; \gamma_r$. By applying lemma A.10 to the derivation of e we obtain that $\mathsf{seq} \vdash \gamma' = \gamma \oplus (\gamma_r \ominus \emptyset)$ holds. We can apply rule *E-E* to the latter fact and the fact that $\exists\, \delta_1.\, \delta = \delta_1, n \mapsto \sigma'; \gamma_r; \gamma$ to perform a single step. Lemma A.51, *ok*$(\gamma_r; \gamma_1)$[1] and $\mathsf{seq} \vdash \gamma' = \gamma \oplus (\gamma_r \ominus \emptyset)$ we have that $\vdash \delta[n \mapsto \sigma']$ holds, where $\sigma' = \sigma; \gamma'$.

Case *T-AP*: The conclusion of rule *T-AP* implies that shape of e is of the form $(e_1\ e_2)^{\xi}$. We have assumed that e is a *redex*, thus e is of the form $(v_1\ v_2)^{\xi}$. Thus, $R; M; \emptyset; \emptyset \vdash (v_1\ v_2)^{\xi} : \tau \& (\gamma; \gamma')$ holds. If ξ equals par then the proof is trivially completed. Otherwise, $\xi = \mathsf{seq}$ and we can use lemma A.10 to derive that $\xi \vdash \gamma' = \gamma_2' \oplus (\gamma \ominus \gamma_1')$, if $\tau_1' \xrightarrow{\gamma_1' \rightarrow \gamma_2'} \tau$ is the type assigned to v_1. The application of lemma A.10 to v_1 typing derivation, implies that $\gamma_1 \simeq \gamma_1'$ and $\gamma_2 \simeq \gamma_2'$. Thus, by lemma A.33 we have that $\xi \vdash \gamma' = \gamma_2 \oplus (\gamma \ominus \gamma_1)$, where γ_1 and γ_2 are the types ascribed on v_1. The latter derivation implies that $\xi \vdash \gamma = \gamma_1 \oplus \gamma_r$, for some γ_r. Consequently, rule *E-A* can be used to perform a single step. Lemma A.48, $\vdash \delta$ (immediate by $R; M \vdash \delta; S$), *ok*$(\gamma; \gamma_1)$ (obtained by *ok*$(\gamma_1; \gamma_2)$ and *ok*$(\gamma; \gamma'')$ holds[2]) and $\xi \vdash \gamma = \gamma_1 \oplus \gamma_r$ imply that $\vdash \delta[n \mapsto \sigma']$ holds, where $\sigma' = \sigma; \gamma_r; \gamma_1$.

Case *T-RP*: rule *E-RP* can be used to perform a single step. $\vdash \delta$ holds by the assumption $R; M \vdash \delta; S$.

[1] *ok*$(\gamma_r; \gamma_1)$ can be obtained in the same way as in lemma A.29 case *T-E*.
[2] we can obtain *ok*$(\gamma_1; \gamma_2)$ and *ok*$(\gamma; \gamma'')$ in the same way as in lemma A.29 case *T-AP*.

Case *T-NG*: Lemma A.11 implies that the region allocation construct is well-typed. Lemma A.10 implies that *is_live*(γ, r) holds, where r is the parent region. Therefore, we can perform a single step by rule *E-NG*. $\vdash \delta[n \mapsto \sigma''; \gamma, \imath^{1,1} \triangleright r]$, where $\sigma = \sigma''; \gamma$, trivially holds as $\imath$ is a *fresh* region.

Case *T-NR*: Similar to the previous case. The store and redex typing gives us that region $\bar{r}$ exists in S. Rule *E-NR* can be used to perform a single step. $\vdash \delta$ holds by the assumption $R; M \vdash \delta; S$.

Case *T-D*: Similar to the previous case. The store and redex typing gives us that region $\bar{r}$ and location ℓ exist in S. The application of lemma A.10 to the typing derivation of e implies that *is_accessible*(γ, r) holds. Rule *E-D* can be used to perform a single step. $\vdash \delta$ holds by the assumption $R; M \vdash \delta; S$.

Case *T-A*: Similar to the previous case. Rule *E-AS* can be used to perform a single step. $\vdash \delta$ holds by the assumption $R; M \vdash \delta; S$.

Case *T-CP*: Lemma A.11 implies that the *cap* construct is well-typed. Lemma A.10 implies that region *is_live*(γ, r). If *nolock*(δ, n, e) holds and the proof is immediate. Otherwise, $\neg$*nolock*(δ, n, e) holds and we have that at least one of the following holds:

- $\delta \neq \delta'', n \mapsto \sigma; \gamma, r^{\kappa} \triangleright \pi$: this case does not hold as it contradicts an assumption of this lemma and *is_live*(γ, r).
- $\eta \neq \mathsf{lk}+$: the proof is completed by applying lemma A.57 to $\delta = \delta'', n \mapsto \sigma; \gamma, r^{\kappa} \triangleright \pi$, $\delta' = \delta'', n \mapsto \sigma'$, where $\sigma' = \sigma;$ *live*$(\gamma, r^{\kappa'} \triangleright \pi)$, $\vdash \delta$ (store typing assumption), $\kappa' = [\![\eta]\!](\kappa)$ (premise of rule *E-C*) and $\eta \neq \mathsf{lk}+$.
- $\vdash \delta[n \mapsto \sigma']$: if this case holds the proof is trivially completed.

Lemma A.55 (Expression — Redex) $R; M; \Delta; \Gamma \vdash e : \tau_1 \,\&\, (\gamma_1;\gamma_2) \wedge e \not\equiv v_1 \Rightarrow \exists\, e', E.E[u] \equiv e \wedge$ *redex*(e)

Proof. Straightforward proof by induction on the typing derivation.

Case *T-I*, *T-U*, *T-F*, *T-L*, *T-R*, *T-RF* then the proof is immediate as e is a value.

Case *T-V*: Immediate as it holds for $E \equiv \Box$ and $u \equiv x \not\equiv v$.

Case *T-NR*: By observing the shape of the expression of *T-NR* typing derivation, $e \equiv \mathsf{new}\ e_1\ @\ e_2$. If e_1 and e_2 are both values then the proof is immediate ($E \equiv \Box$ and $u \equiv \mathsf{new}\ e_1\ @\ e_2$). Otherwise, if e_1 is not a value the application of the induction hypothesis on the typing derivation of e_1 (obtained from *T-NR* inversion) yields that $\exists\, E[u].E[u] \equiv e_1 \wedge u \not\equiv v_2$. Consequently, $\exists\, E.\mathsf{new}\ E[u]\ @\ e_2 \equiv e \wedge u \not\equiv v_2$ or equivalently, $\exists\, E.(\mathsf{new}\ E\ @\ e_2)[u] \equiv e \wedge u \not\equiv v_2$. The last case is that e_1 is a value and e_2 is not. By applying similar reasoning we can prove that $\exists\, E.(\mathsf{new}\ e_1\ @\ E)[u] \equiv e \wedge u \not\equiv v_2$.

Case *T-AP*, *T-RP*, *T-NG*, *T-CP*, *T-D*, *T-A*,*T-E*: We can perform similar reasoning to prove the remaining cases.

Lemma A.56 (Cannonical Forms) $R; M; \Delta; \Gamma \vdash v : \tau \,\&\, (\gamma_1;\gamma_2) \Rightarrow$
$\tau \equiv \langle\rangle \Rightarrow v \equiv () \wedge$
$\tau \equiv \mathsf{rgn}(\bar{r}) \Rightarrow (v \equiv \mathsf{rgn}_{\bar{r}} \wedge \bar{r} \in R) \wedge$
$\tau \equiv \mathsf{ref}(\tau, r) \Rightarrow (v \equiv \mathsf{loc}_{\ell} \wedge \ell \mapsto (\tau, \bar{r}) \in M) \wedge$
$\tau \equiv b \Rightarrow v \equiv n \wedge$
$\tau \equiv \tau_1 \xrightarrow{\gamma_1 \to \gamma_2} \tau_2 \Rightarrow v \equiv \lambda x.\, e\ \mathsf{as}\ \tau_1 \xrightarrow{\gamma_1 \to \gamma_2} \tau_2 \wedge$
$\tau \equiv \forall \rho.\, \tau \Rightarrow v \equiv \Lambda \rho.\, f$

Proof. Straightforward proof by observation of the value typing derivations.

Lemma A.57 (Store Progress —$\vdash \delta$) $\delta = \delta'', n \mapsto \sigma; \gamma, r^{\kappa} \triangleright \pi \wedge \delta' = \delta'', n \mapsto \sigma; \mathit{live}(\gamma, r^{\kappa'} \triangleright \pi) \wedge \vdash \delta \wedge \kappa' = [\![\eta]\!](\kappa) \wedge \eta \neq \mathsf{lk}+ \Rightarrow \vdash \delta'$.

Proof. To prove that $\vdash \delta'$ holds it suffices to show that its premises hold. By inversion of $\vdash \delta'$ we have that:

- $\vdash \delta''$: The assumption tells us that $\vdash \delta'', n \mapsto \sigma; \gamma, r^{\kappa} \triangleright \pi$ holds. By inversion of the latter fact we have that $\vdash \delta''$ holds.
- $\sigma; \mathit{live}(\gamma, r^{\kappa'} \triangleright \pi) \vdash \delta''$: Let γ_0 be equal to $\gamma, r^{\kappa} \triangleright \pi$ and γ_0' be equal to $\mathit{live}(\gamma, r^{\kappa'} \triangleright \pi)$. The proof is completed by the application of lemma A.58 to $\kappa' = [\![\eta]\!](\kappa)$, $\delta = \delta'', n \mapsto \sigma; \gamma_0$, $\delta' = \delta'', n \mapsto \sigma; \gamma_0', \sigma; \gamma_0 \vdash \delta''$ (by inversion of $\vdash \delta$), $\delta'' \subseteq \delta''$ and $\eta \neq \mathsf{lk}+$.

Lemma A.58 (Store Progress — Helper lemma 1) $\kappa' = [\![\eta]\!](\kappa) \wedge \delta = \delta'', n \mapsto \sigma; \gamma \wedge \delta' = \delta'', n \mapsto \sigma; \gamma' \wedge \gamma = \gamma'', r^{\kappa} \triangleright \pi \wedge \gamma' = \mathit{live}(\gamma'', r^{\kappa'} \triangleright \pi) \wedge \sigma; \gamma \vdash \delta'' \wedge \delta_0 \subseteq \delta'' \wedge \eta \neq \mathsf{lk}+ \Rightarrow \sigma; \gamma' \vdash \delta_0$.

Proof. We proceed by induction on the derivation of δ_0.

- $\emptyset$: given that $\sigma; \gamma' \simeq \sigma_1; \gamma_x, \imath^{\kappa_1} \triangleright \pi + \sigma_2$ and $\mathit{rg}(\kappa_1) > 0$ and $\mathit{is\_pure}(\kappa_1)$, it suffices to prove that $\mathit{zero\_pure}(\sigma_1, \imath)$ and $\imath \notin \mathit{dom}(\sigma_2; \gamma_x)$ for all regions $\imath$ that belong in the domain of $\sigma; \gamma'$. By inversion of $\sigma; \gamma \vdash \emptyset$ (derived by the assumption $\sigma; \gamma \vdash \delta''$), we have that $\sigma; \gamma \simeq \sigma_3; \gamma_y, \imath^{\kappa_2} \triangleright \pi + \sigma_4$, $\mathit{rg}(\kappa_2) > 0$ and $\mathit{is\_pure}(\kappa_2)$, $\mathit{zero\_pure}(\sigma_3, \imath)$, and $\imath \notin \mathit{dom}(\sigma_4; \gamma_y)$. We proceed by performing a case analysis:
  - If $\imath$ does not belong in the domain of $\emptyset; \gamma$, then $\sigma_3 \simeq \sigma_1$, $\gamma_y \simeq \gamma_x$ and there exists a stack σ_4' such that $\sigma_2 \simeq \sigma_4'; \gamma'$ and $\sigma_4 \simeq \sigma_4'; \gamma$. Hence, $\mathit{zero\_pure}(\sigma_1, \imath)$ holds. The assumption that $\gamma' = \mathit{live}(\gamma'', r^{\kappa'} \triangleright \pi)$ and $\gamma = \gamma'', r^{\kappa} \triangleright \pi$ holds implies that $\mathit{dom}(\emptyset; \gamma') \subseteq \mathit{dom}(\emptyset; \gamma)$. Consequently, $\imath \notin \mathit{dom}(\sigma_4'; \gamma'; \gamma_x)$ holds.
  - If $\imath$ does belong in the domain of $\emptyset; \gamma$, then $\sigma_4 = \sigma_2 = \emptyset$ and $\sigma_1 \simeq \sigma_3 \simeq s$. Hence, $\mathit{zero\_pure}(\sigma_1, \imath)$ trivially holds. The assumption that $\gamma' = \mathit{live}(\gamma'', r^{\kappa'} \triangleright \pi)$ and $\gamma = \gamma'', r^{\kappa} \triangleright \pi$ holds implies that $\mathit{dom}(\emptyset; \gamma') \subseteq \mathit{dom}(\emptyset; \gamma)$ and thus, $\mathit{dom}(\emptyset; \gamma_x) \subseteq \mathit{dom}(\emptyset; \gamma_y)$. Consequently, $\imath \notin \mathit{dom}(\emptyset; \gamma_x)$ holds.

Case $\delta_1, n_1 \mapsto \sigma_1$: it suffices to prove that

- $\sigma; \gamma' \vdash \delta_1$: is immediate by applying the induction hypothesis.
- Given that $\mathit{is\_accessible}(\sigma; \gamma', \imath)$ holds for all $\imath$ that belong in the domain of $\sigma; \gamma'$, it suffices to prove that $\neg \mathit{is\_accessible}(\sigma_1, \imath)$ holds. The assumption that $\sigma; \gamma \vdash \delta''$ holds, the fact that $\imath$ is accessible in γ' and $\eta \neq \mathsf{lk}+$ imply that $\mathit{is\_accessible}(\sigma; \gamma, \imath)$ and thus $\neg \mathit{is\_accessible}(\sigma_1, \imath)$.

Appendix B

Formal semantics and proof of soundness for Chapter 3

B.1 Language syntax

Expression	e	$::=$	$x \mid f \mid () \mid \texttt{true} \mid \texttt{false} \mid e\ e \mid e\,[r] \mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e$
		$\mid$	$\texttt{newrgn}\ \rho, x\ @\ e\ \texttt{in}\ e \mid \texttt{new}\ e\ @\ e \mid e := e \mid \texttt{deref}\ e \mid \texttt{cap}_\eta\ e$
		$\mid$	$\texttt{spawn}_\xi\ e \mid \texttt{loc}_\ell \mid \texttt{rgn}_\iota$
Function	f	$::=$	$\lambda x.\, e \mid \Lambda\rho.\, f \mid \texttt{fix}\ x.\, f$
Value	v	$::=$	$f \mid () \mid \texttt{true} \mid \texttt{false} \mid \texttt{loc}_\ell \mid \texttt{rgn}_\iota$
Region	r	$::=$	$\rho \mid \iota$
Count vector	η	$::=$	(n, n, n)
Spawn effect	ξ	$::=$	$\emptyset \mid \xi, r \mapsto \eta$

B.2 Operational semantics

Auxiliary syntax for operational semantics

Hierarchy	θ	$::=$	$\emptyset \mid \theta, \iota \mapsto (\eta, \iota)$
Heap	H	$::=$	$\emptyset \mid H, \ell \mapsto v$
Store	S	$::=$	$\emptyset \mid S, \iota \mapsto H$
Threads	T	$::=$	$\emptyset \mid T, \langle \theta; e \rangle$
Configuration	C	$::=$	$S; T$
Stack	E	$::=$	$\Box \mid E[F]$
Frame	F	$::=$	$\Box\ e \mid v\ \Box \mid \Box\,[r] \mid \texttt{if}\ \Box\ \texttt{then}\ e\ \texttt{else}\ e \mid \texttt{newrgn}\ \rho, x\ @\ \Box\ \texttt{in}\ e$
		$\mid$	$\texttt{new}\ \Box\ @\ e \mid \texttt{new}\ v\ @\ \Box \mid \Box := e \mid v := \Box \mid \texttt{deref}\ \Box \mid \texttt{cap}_\eta\ \Box$
Redex	u	$::=$	$(\lambda x.\, e)\ v \mid \texttt{cap}_\eta\ \texttt{rgn}_\iota \mid \texttt{deref}\ \texttt{loc}_\ell \mid \texttt{loc}_\ell := v \mid \texttt{new}\ v\ @\ \texttt{rgn}_\iota$
		$\mid$	$\texttt{newrgn}\ \rho, x\ @\ \texttt{rgn}_\iota\ \texttt{in}\ e_2 \mid (\Lambda\rho.\, f)\,[r] \mid \texttt{spawn}_\xi\ e_1 \mid (\texttt{fix}\ x.\, f)\ v$
		$\mid$	$\texttt{if true then}\ e_1\ \texttt{else}\ e_2 \mid \texttt{if false then}\ e_1\ \texttt{else}\ e_2$

Evaluation relation $C \rightsquigarrow C'$

$$\frac{\mathsf{live}(\theta) = \emptyset}{S; T, \langle \theta; () \rangle \rightsquigarrow S; T} \quad \textit{(E-T)}$$

$$\frac{\mathsf{merge}(\xi) \vdash \theta = \theta' \oplus \theta'' \qquad \mathsf{dom}(\theta'') \subseteq \mathsf{live}(\theta)}{S; T, \langle \theta; E[\texttt{spawn}_\xi\ e] \rangle \rightsquigarrow S; T, \langle \theta'; E[()] \rangle, \langle \theta''; \Box[e] \rangle} \quad \textit{(E-SP)}$$

$$\frac{f \equiv \lambda x.\, e}{S; T, \langle \theta; E[f\ \ v]\rangle \rightsquigarrow S; T, \langle \theta; E[e[v/x]]\rangle} \textit{(E-A)}$$

$$\frac{f \equiv \Lambda \rho.\, f'}{S; T, \langle \theta; E[f\ [\imath]]\rangle \rightsquigarrow S; T, \langle \theta; E[f'[\imath/\rho]]\rangle} \textit{(E-RP)}$$

$$\frac{f \equiv \texttt{fix}\ x.\, f'}{S; T, \langle \theta; E[f\ \ v]\rangle \rightsquigarrow S; T, \langle \theta; E[f'[f/x]\ \ v]\rangle} \textit{(E-FX)}$$

$$\frac{}{S; T, \langle \theta; E[\texttt{if true then}\ e_1\ \texttt{else}\ e_2]\rangle \rightsquigarrow S; T, \langle \theta; E[e_1]\rangle} \textit{(E-IT)}$$

$$\frac{}{S; T, \langle \theta; E[\texttt{if false then}\ e_1\ \texttt{else}\ e_2]\rangle \rightsquigarrow S; T, \langle \theta; E[e_2]\rangle} \textit{(E-IF)}$$

$$\frac{\jmath \in \mathsf{live}(\theta) \cup \{\bot\} \qquad \text{fresh}\ \imath \qquad \theta' = \theta, \imath \mapsto ((1,1,0), \jmath)}{S; T, \langle \theta; E[\texttt{newrgn}\ \rho, x\ @\ \texttt{rgn}_\jmath\ \texttt{in}\ e]\rangle \rightsquigarrow S, \imath \mapsto \emptyset; T, \langle \theta'; E[e[\imath/\rho][\texttt{rgn}_\imath/x]]\rangle} \textit{(E-NR)}$$

$$\frac{\imath \in \mathsf{live}(\theta) \qquad \text{fresh}\ \ell}{S; T, \langle \theta; E[\texttt{new}\ v\ @\ \texttt{rgn}_\imath]\rangle \rightsquigarrow S[\imath \mapsto S(\imath), \ell \mapsto v]; T, \langle \theta; E[\texttt{loc}_\ell]\rangle} \textit{(E-NL)}$$

$$\frac{\ell \mapsto v' \in S(\imath) \qquad \imath \in \mathsf{wlocked}(\theta) \qquad \imath \notin \mathsf{rwlocked}(T)}{S; T, \langle \theta; E[\texttt{loc}_\ell := v]\rangle \rightsquigarrow S[\imath \mapsto S(\imath)[\ell \mapsto v]]; T, \langle \theta; E[()]\rangle} \textit{(E-AS)}$$

$$\frac{\ell \mapsto v \in S(\imath) \qquad \imath \in \mathsf{rwlocked}(\theta) \qquad \imath \notin \mathsf{wlocked}(T)}{S; T, \langle \theta; E[\texttt{deref loc}_\ell]\rangle \rightsquigarrow S; T, \langle \theta; E[v]\rangle} \textit{(E-D)}$$

$$\frac{\imath \in \mathsf{live}(\theta) \qquad \theta' = \theta, \imath \mapsto (\eta + \eta', \jmath) \qquad \mathsf{mutex}(\{\theta'\} \cup \{\theta'' \mid \langle \theta''; e'\rangle \in T\})}{S; T, \langle \theta, \imath \mapsto (\eta, \jmath); E[\texttt{cap}_{\eta'}\ \texttt{rgn}_\imath]\rangle \rightsquigarrow S; T, \langle \theta'; E[()]\rangle} \textit{(E-CP)}$$

Auxiliary functions and predicates

$$\begin{array}{lcll}
\mathsf{merge}(\emptyset) & = & \emptyset & \\
\mathsf{merge}(\xi, r \mapsto \eta) & = & \mathsf{merge}(\xi), r \mapsto \eta & \text{if } r \notin \{r' \mid r' \mapsto \eta' \in \xi\} \\
\mathsf{merge}(\xi, r \mapsto \eta, r \mapsto \eta') & = & \mathsf{merge}(\xi, r \mapsto (\eta_1 + \eta_2)) & \\
\mathsf{ok}(n_1, n_2, n_3) & = & n_1 \geq 0 \wedge n_2 \geq 0 \wedge n_3 \geq 0 & \\
(c_1, w_1, z_1) \oplus (c_2, w_2, z_2) & = & (c_1 + c_2, w_1 + w_2, z_1 + z_2) & \text{if } \mathsf{ok}(c_1, w_1, z_1) \wedge \mathsf{ok}(c_2, w_2, z_2) \wedge \\
 & & & (w_1 = 0 \vee w_2 = 0) \wedge (c_2 > 0) \wedge \\
 & & & (c_1 = 0 \implies w_1 = z_1 = 0) \wedge \\
 & & & (w_1 > 0 \implies z_2 = 0) \wedge \\
 & & & (w_2 > 0 \implies z_1 = 0)
\end{array}$$

$$\begin{array}{lcll}
\mathsf{ancestors}(\theta, \bot) & = & \emptyset & \\
\mathsf{ancestors}(\theta, \imath) & = & \{\imath\} \cup \mathsf{ancestors}(\theta', \jmath) & \text{if } \theta = \theta', \imath \mapsto (\eta, \jmath) \\
\mathsf{live}(\theta) & = & \{\imath \mid \forall \jmath \in \mathsf{ancestors}(\theta, \imath).\ \exists \jmath \mapsto (\eta, \jmath') \in \theta.\ \mathsf{ok}(\eta - (1,0,0))\} & \\
\mathsf{wlocked}(\theta) & = & \{\imath \mid \imath \in \mathsf{live}(\theta) \wedge \exists \jmath \mapsto (\eta, \jmath') \in \theta.\ \jmath \in \mathsf{ancestors}(\theta, \imath) \wedge \mathsf{ok}(\eta - (0,1,0))\} & \\
\mathsf{rlocked}(\theta) & = & \{\imath \mid \imath \in \mathsf{live}(\theta) \wedge \exists \jmath \mapsto (\eta, \jmath') \in \theta.\ \jmath \in \mathsf{ancestors}(\theta, \imath) \wedge \mathsf{ok}(\eta - (0,0,1))\} & \\
\mathsf{rwlocked}(\theta) & = & \mathsf{rlocked}(\theta) \cup \mathsf{wlocked}(\theta) & \\
\mathsf{wlocked}(T) & = & \{\imath \mid \exists \langle \theta, e\rangle \in T.\ \imath \in \mathsf{wlocked}(\theta)\} & \\
\mathsf{rwlocked}(T) & = & \{\imath \mid \exists \langle \theta, e\rangle \in T.\ \imath \in \mathsf{rwlocked}(\theta)\} & \\
\mathsf{mutex}(\{\theta_1, \ldots, \theta_n\}) & = & \forall \imath \neq \jmath.\ \mathsf{rwlocked}(\theta_\imath) \cap \mathsf{wlocked}(\theta_\jmath) = \mathsf{wlocked}(\theta_\imath) \cap \mathsf{rwlocked}(\theta_\jmath) = \emptyset &
\end{array}$$

$$\frac{}{\emptyset \vdash \theta = \theta \oplus \emptyset} \qquad \frac{\eta = \eta_1 \oplus \eta_2 \quad \forall \imath' \in \mathsf{dom}(\xi).\, \imath \notin \mathsf{ancestors}(\theta, \imath') \qquad \xi \vdash \theta = \theta_1 \oplus \theta_2 \quad \jmath' = \text{if } \jmath \in \mathsf{dom}(\xi) \text{ then } \jmath \text{ else } \bot}{\xi, \imath \mapsto \eta_2 \vdash \theta, \imath \mapsto (\eta, \jmath) = \theta_1, \imath \mapsto (\eta_1, \jmath) \oplus \theta_2, \imath \mapsto (\eta_2, \jmath')}$$

B.3 Static semantics

Syntax for types, effects and contexts

Type	τ	$::=$	$\mathtt{unit} \mid \mathtt{bool} \mid \tau \xrightarrow{\gamma} \tau \mid \forall \rho.\, \tau \mid \mathtt{Ref}(\tau, r) \mid \mathtt{Rgn}(r)$
Constraint	δ	$::=$	$\mathtt{R} \mid \mathtt{W} \mid \neg\mathtt{RW} \mid \neg\mathtt{W} \mid \mathtt{Live} \mid \neg\mathtt{Live}$
Event	ζ	$::=$	$\mathtt{Cap}\,\xi \mid \delta\, r \mid \mathtt{Spawn}\,\xi\,\gamma \mid \mathtt{Join}\,\gamma\,\gamma$
Effect	γ	$::=$	$\emptyset \mid \zeta :: \gamma$
Type context	$\square$	$::=$	$\emptyset \mid \Gamma, x : \tau$
Region context	$\square$	$::=$	$\emptyset \mid \Delta, \rho$
Heap context	M	$::=$	$\emptyset \mid M, \ell \mapsto (\tau, \imath)$
Store context	R	$::=$	$\emptyset \mid R, \imath$
Variable list substitution	$\square[r/\rho]$	$::=$	$\emptyset \mid \Gamma_1[r/\rho], x : \tau[r/\rho]$

Typing rules

$$\frac{x : \tau \in \Gamma \qquad \vdash R; M; \Delta; \Gamma}{R; M; \Delta; \Gamma \vdash x : \tau \,\&\, \emptyset} \textit{ (T-V)} \qquad \frac{\vdash R; M; \Delta; \Gamma}{R; M; \Delta; \Gamma \vdash () : \mathtt{unit} \,\&\, \emptyset} \textit{ (T-U)}$$

$$\frac{\vdash R; M; \Delta; \Gamma}{R; M; \Delta; \Gamma \vdash \mathtt{true} : \mathtt{bool} \,\&\, \emptyset} \textit{ (T-TR)} \qquad \frac{\vdash R; M; \Delta; \Gamma}{R; M; \Delta; \Gamma \vdash \mathtt{false} : \mathtt{bool} \,\&\, \emptyset} \textit{ (T-FL)}$$

$$\frac{\imath \in R \cup \{\bot\} \qquad \vdash R; M; \Delta; \Gamma}{R; M; \Delta; \Gamma \vdash \mathtt{rgn}_\imath : \mathtt{Rgn}(\imath) \,\&\, \emptyset} \textit{ (T-R)} \qquad \frac{\ell \mapsto (\tau, \imath) \in M \qquad \vdash R; M; \Delta; \Gamma}{R; M; \Delta; \Gamma \vdash \mathtt{loc}_\ell : \mathtt{Ref}(\tau, \imath) \,\&\, \emptyset} \textit{ (T-L)}$$

$$\frac{R; M; \Delta; \Gamma, x : \tau_1 \vdash e : \tau_2 \,\&\, \gamma}{R; M; \Delta; \Gamma \vdash \lambda x.\, e : \tau_1 \xrightarrow{\gamma} \tau_2 \,\&\, \emptyset} \textit{ (T-F)}$$

$$\frac{R; M; \Delta; \Gamma \vdash e_1 : \tau_1 \xrightarrow{\gamma} \tau_2 \,\&\, \gamma_1 \qquad R; M; \Delta; \Gamma \vdash e_2 : \tau_1 \,\&\, \gamma_2}{R; M; \Delta; \Gamma \vdash e_1\ e_2 : \tau_2 \,\&\, \gamma_1 :: \gamma_2 :: \gamma} \textit{ (T-A)}$$

$$\frac{R; \Delta \vdash \Gamma \qquad R; M; \Delta, \rho; \Gamma \vdash f : \tau \,\&\, \emptyset}{R; M; \Delta; \Gamma \vdash \Lambda\rho.\, f : \forall \rho.\, \tau \,\&\, \emptyset} \textit{ (T-RF)}$$

$$\frac{R; M; \Delta; \Gamma \vdash e : \forall \rho.\, \tau \,\&\, \gamma \qquad R; \Delta \vdash r \qquad r \neq \bot}{R; M; \Delta; \Gamma \vdash e\,[r] : \tau[r/\rho] \,\&\, \gamma} \textit{ (T-RP)}$$

$$\frac{R; M; \Delta; \Gamma \vdash e : \mathtt{bool} \,\&\, \gamma \qquad R; M; \Delta; \Gamma \vdash e_1 : \tau \,\&\, \gamma_1 \qquad R; M; \Delta; \Gamma \vdash e_2 : \tau \,\&\, \gamma_2}{R; M; \Delta; \Gamma \vdash \mathtt{if}\ e\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 : \tau \,\&\, \gamma :: \mathtt{Join}\,\gamma_1\,\gamma_2} \textit{ (T-IF)}$$

$$\frac{R; \Delta \vdash \xi \qquad R; M; \Delta; \Gamma \vdash e : \mathtt{unit} \,\&\, \gamma \qquad \mathsf{dom}(\xi) = \mathsf{dom}(\gamma)}{R; M; \Delta; \Gamma \vdash \mathtt{spawn}_\xi\ e : \mathtt{unit} \,\&\, \mathtt{Spawn}\,\xi\,\gamma} \textit{ (T-SP)}$$

$$\frac{\gamma_L = \{\mathtt{Live}\, r \mid r \in \mathsf{dom}(\phi(\emptyset))\} \qquad \gamma_s = \mathsf{summary}(\phi(\gamma_L)) \qquad R; M; \Delta; \Gamma, x : \tau_1 \xrightarrow{\gamma_s} \tau_2 \vdash f : \tau_1 \xrightarrow{\phi(\gamma_s)} \tau_2 \,\&\, \emptyset}{R; M; \Delta; \Gamma \vdash \mathtt{fix}\ x.\, f : \tau_1 \xrightarrow{\gamma_s} \tau_2 \,\&\, \emptyset} \textit{ (T-FX)}$$

$$\frac{R; M; \Delta; \Gamma \vdash e : \mathtt{Rgn}(r) \,\&\, \gamma \qquad r \neq \bot}{R; M; \Delta; \Gamma \vdash \mathtt{cap}_\eta\ e : \mathtt{unit} \,\&\, \gamma :: \mathtt{Cap}\,\{r \mapsto \eta\}}\ \textit{(T-CP)}$$

$$\frac{R; M; \Delta; \Gamma \vdash e_1 : \mathtt{Ref}(\tau, r) \,\&\, \gamma_1 \qquad R; M; \Delta; \Gamma \vdash e_2 : \tau \,\&\, \gamma_2 \qquad r \neq \bot}{R; M; \Delta; \Gamma \vdash e_1 := e_2 : \mathtt{unit} \,\&\, \gamma_1 :: \gamma_2 :: \mathtt{W}\, r}\ \textit{(T-AS)}$$

$$\frac{R; M; \Delta; \Gamma \vdash e : \mathtt{Ref}(\tau, r) \,\&\, \gamma \qquad r \neq \bot}{R; M; \Delta; \Gamma \vdash \mathtt{deref}\ e : \tau \,\&\, \gamma :: \mathtt{R}\, r}\ \textit{(T-D)}$$

$$\frac{R; M; \Delta; \Gamma \vdash e_1 : \tau \,\&\, \gamma_1 \qquad R; M; \Delta; \Gamma \vdash e_2 : \mathtt{Rgn}(r) \,\&\, \gamma_2 \qquad r \neq \bot}{R; M; \Delta; \Gamma \vdash \mathtt{new}\ e_1\ @\ e_2 : \mathtt{Ref}(\tau, r) \,\&\, \gamma_1 :: \gamma_2 :: \mathtt{Live}\, r}\ \textit{(T-NL)}$$

$$\frac{\begin{array}{cc} R; M; \Delta; \Gamma \vdash e_1 : \mathtt{Rgn}(r) \,\&\, \gamma_1 & R; M; \Delta, \rho; \Gamma, x : \mathtt{Rgn}(\rho) \vdash e_2 : \tau \,\&\, \gamma_2 \\ R; \Delta \vdash \tau & \mathsf{translate}(\gamma_2, \rho, (1,1,0), r) = \gamma_2' \end{array}}{R; M; \Delta; \Gamma \vdash \mathtt{newrgn}\ \rho, x\ @\ e_1\ \mathtt{in}\ e_2 : \tau \,\&\, \gamma_1 :: \mathtt{Live}\, r :: \gamma_2'}\ \textit{(T-NR)}$$

Auxiliary functions and predicates

$$\xi(r) = \begin{cases} \eta & \text{if } r \mapsto \eta \in \xi \\ (0,0,0) & \text{otherwise} \end{cases}$$

$\xi_1 - \xi_2 = \{r \mapsto \xi_1(r) - \xi_2(r)\}$

$\mathsf{ok}(\xi) = \forall \imath \mapsto \eta \in \xi.\ \mathsf{ok}(\eta)$

$\mathsf{ok}(\theta) = \forall \imath \mapsto (\eta, \jmath) \in \theta.\ \mathsf{ok}(\eta) \wedge \mathsf{ancestors}(\theta, \imath)$ defined

$\mathsf{hierarchy\_ok}(\theta_1; \theta_2) = \forall \imath \mapsto (\eta, \jmath) \in \theta_1.\ \exists \jmath'.\ \imath \mapsto (\eta', \jmath') \in \theta_2 \wedge (\jmath = \jmath' \vee (\jmath = \bot \wedge \jmath' \notin \mathsf{dom}(\theta_1)))$

Summary

$$\frac{\mathsf{ok}(\xi)}{\mathsf{recursive}(\xi; \emptyset) = \xi} \qquad \frac{\delta \notin \{\neg\mathtt{RW}, \neg\mathtt{W}, \neg\mathtt{Live}\} \qquad \mathsf{recursive}(\xi; \gamma) = \xi'}{\mathsf{recursive}(\xi; \delta\, r :: \gamma) = \xi'}$$

$$\frac{\mathsf{ok}(\xi) \qquad \mathsf{recursive}(\xi - \xi'; \gamma) = \xi''}{\mathsf{recursive}(\xi; \mathsf{Cap}\, \xi' :: \gamma) = \xi''}$$

$$\frac{\begin{array}{cc} \mathsf{ok}(\xi) & \xi_r = \xi - \xi_s \\ \forall r \mapsto \eta \in \xi_s.\ \mathsf{rd}(\eta) = \mathsf{wr}(\eta) = 0 & \mathsf{recursive}(\xi_r; \gamma) = \xi_r' \end{array}}{\mathsf{recursive}(\xi; \mathsf{Spawn}\, \xi_s\, \gamma_s :: \gamma) = \xi_r'}$$

$$\frac{\mathsf{recursive}(\xi; \gamma_1) = \xi' \qquad \mathsf{recursive}(\xi; \gamma_2) = \xi' \qquad \mathsf{recursive}(\xi'; \gamma) = \xi''}{\mathsf{recursive}(\xi; \mathtt{Join}\, \gamma_1\, \gamma_2 :: \gamma) = \xi''}$$

$$\frac{\begin{array}{c} \mathsf{recursive}(\xi_1; \gamma) = \xi_1 \\ \xi_1 = \{r \mapsto (1,0,0) \mid r \in \mathsf{dom}(\gamma)\} \qquad \xi_2 = \{r \mapsto (-1,0,0) \mid r \in \mathsf{dom}(\gamma)\} \end{array}}{\mathsf{summary}(\gamma) = \mathsf{Cap}\, \xi_1 :: \mathsf{Spawn}\, \xi_1\, (\gamma :: \mathsf{Cap}\, \xi_2)}$$

$$
\begin{array}{lcll}
\mathsf{wr}(n_1, n_2, n_3) & = & n_2 & \\
\mathsf{rd}(n_1, n_2, n_3) & = & n_3 & \\
\mathsf{rg}(n_1, n_2, n_3) & = & n_1 & \\
\mathsf{bot}(\delta, \bot) & = & \emptyset & \text{if } \delta \notin \{\mathtt{R}, \mathtt{W}\} \\
\mathsf{bot}(\delta, r) & = & \delta\, r & \text{if } r \neq \bot \\
\\
\mathsf{solve}(\mathtt{R}, r, \eta) & = & \mathsf{bot}(\mathtt{Live}, r) & \text{if } \mathsf{ok}(\eta - (1,0,0)) \wedge \mathsf{wr}(\eta) + \mathsf{rd}(\eta) > 0 \\
\mathsf{solve}(\mathtt{R}, r, \eta) & = & \mathsf{bot}(\mathtt{R}, r) & \text{if } \mathsf{ok}(\eta - (1,0,0)) \wedge \mathsf{wr}(\eta) + \mathsf{rd}(\eta) = 0 \\
\mathsf{solve}(\mathtt{W}, r, \eta) & = & \mathsf{bot}(\mathtt{Live}, r) & \text{if } \mathsf{ok}(\eta - (1,0,0)) \wedge \mathsf{wr}(\eta) > 0 \\
\mathsf{solve}(\mathtt{W}, r, \eta) & = & \mathsf{bot}(\mathtt{W}, r) & \text{if } \mathsf{ok}(\eta - (1,0,0)) \wedge \mathsf{wr}(\eta) = 0 \\
\mathsf{solve}(\neg\mathtt{RW}, r, \eta) & = & \mathsf{bot}(\neg\mathtt{RW}, r) & \text{if } \mathsf{ok}(\eta - (1,0,0)) \wedge \mathsf{wr}(\eta) = \mathsf{rd}(\eta) = 0 \\
\mathsf{solve}(\neg\mathtt{W}, r, \eta) & = & \mathsf{bot}(\neg\mathtt{W}, r) & \text{if } \mathsf{ok}(\eta - (1,0,0)) \wedge \mathsf{wr}(\eta) = 0 \\
\mathsf{solve}(\mathtt{Live}, r, \eta) & = & \mathsf{bot}(\mathtt{Live}, r) & \text{if } \mathsf{ok}(\eta - (1,0,0)) \\
\mathsf{solve}(\neg\mathtt{Live}, r, \eta) & = & \emptyset & \text{if } \mathsf{ok}(\eta) \wedge \mathsf{rg}(\eta) = 0 \\
\mathsf{solve}(\neg\mathtt{Live}, r, \eta) & = & \neg\mathtt{Live}\, r & \text{if } \mathsf{ok}(\eta - (1,0,0)) \wedge r \neq \bot \\
\\
\mathsf{p\text{-}constraint}(r, \eta) & = & \mathsf{bot}(\neg\mathtt{RW}, r) & \text{if } \mathsf{wr}(\eta) > 0 \\
\mathsf{p\text{-}constraint}(r, \eta) & = & \mathsf{bot}(\neg\mathtt{W}, r) & \text{if } \mathsf{wr}(\eta) = 0 \wedge \mathsf{rd}(\eta) > 0 \\
\mathsf{p\text{-}constraint}(r, \eta) & = & \emptyset & \text{if } \mathsf{wr}(\eta) = \mathsf{rd}(\eta) = 0
\end{array}
$$

$$\frac{\mathsf{solve}(\neg\mathtt{Live}, r', \eta) = \gamma}{\mathsf{translate}(\emptyset, r, \eta, r') = \gamma} \quad \textit{(TR-E)}$$

$$\frac{r \notin \mathsf{dom}(\xi) \qquad \mathsf{translate}(\gamma, r, \eta, r') = \gamma'}{\mathsf{translate}(\mathtt{Cap}\,\xi :: \gamma, r, \eta, r') = \mathtt{Cap}\,\xi :: \gamma'} \quad \textit{(TR-CN)}$$

$$\frac{\begin{array}{c}\mathsf{merge}(\xi) = \xi', r \mapsto \eta' \qquad \gamma_s = \mathsf{solve}(\mathtt{Live}, r', \eta) :: \mathtt{Cap}\,\xi' \\ \mathsf{translate}(\gamma, r, \eta + \eta', r') = \gamma' \qquad \mathsf{ok}(\eta + \eta')\end{array}}{\mathsf{translate}(\mathtt{Cap}\,\xi :: \gamma, r, \eta, r') = \gamma_s :: \gamma'} \quad \textit{(TR-CT)}$$

$$\frac{r_1 \neq r_2 \qquad \mathsf{translate}(\gamma, r_2, \eta, r') = \gamma'}{\mathsf{translate}(\delta\, r_1 :: \gamma, r_2, \eta, r') = \delta\, r_1 :: \gamma'} \quad \textit{(TR-DN)}$$

$$\frac{\mathsf{solve}(\delta, r', \eta) = \gamma_s \qquad \mathsf{translate}(\gamma, r, \eta, r') = \gamma'}{\mathsf{translate}(\delta\, r :: \gamma, r, \eta, r') = \gamma_s :: \gamma'} \quad \textit{(TR-DT)}$$

$$\frac{\mathsf{translate}(\gamma_1 :: \gamma, r, \eta, r') = \gamma_1' \qquad \mathsf{translate}(\gamma_2 :: \gamma, r, \eta, r') = \gamma_2'}{\mathsf{translate}(\mathtt{Join}\,\gamma_1\,\gamma_2 :: \gamma, r, \eta, r') = \mathtt{Join}\,\gamma_1'\,\gamma_2'} \quad \textit{(TR-J)}$$

$$\frac{r \notin \mathsf{dom}(\xi) \qquad \mathsf{translate}(\gamma, r, \eta, r') = \gamma'}{\mathsf{translate}(\mathtt{Spawn}\,\xi\,\gamma_s :: \gamma, r, \eta, r') = \mathtt{Spawn}\,\xi\,\gamma_s :: \gamma'} \quad \textit{(TR-SN)}$$

$$\frac{\begin{array}{c}\mathsf{merge}(\xi) = \xi', r \mapsto \eta_s \qquad \eta = \eta_r \oplus \eta_s \qquad r_s = \text{if } r' \in \mathsf{dom}(\xi) \text{ then } r' \text{ else } \bot \\ \mathsf{p\text{-}constraint}(r_s, \eta_r) = \gamma_s' \qquad \mathsf{translate}(\gamma_s, r, \eta_s, r_s) = \gamma_s'' \\ \mathsf{p\text{-}constraint}(r', \eta_s) = \gamma_r' \qquad \mathsf{translate}(\gamma, r, \eta_r, r') = \gamma_r'' \qquad \gamma_r''' = \mathsf{bot}(\mathtt{Live}, r')\end{array}}{\mathsf{translate}(\mathtt{Spawn}\,\xi\,\gamma_s :: \gamma, r, \eta, r') = \gamma_r''' :: \mathtt{Spawn}\,\xi'\,(\gamma_s' :: \gamma_s'') :: \gamma_r' :: \gamma_r''} \quad \textit{(TR-ST)}$$

Region well-formedness

$$\frac{r \in \Delta \cup R \cup \{\bot\}}{R; \Delta \vdash r}$$

Constraint well-formedness

$$\frac{}{R; \Delta \vdash \emptyset} \qquad \frac{R; \Delta \vdash r \qquad R; \Delta \vdash \gamma}{R; \Delta \vdash \delta\, r :: \gamma} \qquad \frac{\begin{array}{c}\forall (r, \eta) \in \xi.\mathsf{ok}(\eta) \ \wedge\ R; \Delta \vdash r \\ R; \Delta \vdash \gamma_1 \qquad R; \Delta \vdash \gamma_2\end{array}}{R; \Delta \vdash \mathtt{Spawn}\, \xi\, \gamma_1 :: \gamma_2}$$

$$\frac{R; \Delta \vdash \gamma_1 \qquad R; \Delta \vdash \gamma_2 \qquad R; \Delta \vdash \gamma_3}{R; \Delta \vdash \mathtt{Join}\, \gamma_1\, \gamma_2 :: \gamma_3} \qquad \frac{\forall r \in \mathsf{dom}(\xi).\, R; \Delta \vdash r \qquad R; \Delta \vdash \gamma_1}{R; \Delta \vdash \mathtt{Cap}\, \xi :: \gamma_1}$$

Type well-formedness

$$\frac{R; \Delta \vdash r}{R; \Delta \vdash \mathtt{Rgn}(r)} \qquad \frac{R; \Delta, \rho \vdash \tau}{R; \Delta \vdash \forall \rho.\, \tau} \qquad \frac{R; \Delta \vdash \tau \qquad R; \Delta \vdash r}{R; \Delta \vdash \mathtt{Ref}(\tau, r)}$$

$$\frac{R; \Delta \vdash \tau_1 \qquad R; \Delta \vdash \gamma_1 \qquad R; \Delta \vdash \tau_2}{R; \Delta \vdash \tau_1 \xrightarrow{\gamma_1} \tau_2} \qquad \frac{}{R; \Delta \vdash \mathtt{unit}} \qquad \frac{}{R; \Delta \vdash \mathtt{bool}}$$

Context well-formedness

$$\frac{R \vdash M \qquad R; \Delta \vdash \Gamma \qquad \bot \notin R}{\vdash R; M; \Delta; \Gamma}$$

Γ well-formedness

$$\frac{}{R; \Delta \vdash \emptyset} \qquad \frac{R; \Delta \vdash \tau \qquad x \notin \mathsf{dom}(\Gamma) \qquad R; \Delta \vdash \Gamma}{R; \Delta \vdash \Gamma, x : \tau}$$

M Well-formedness

$$\frac{}{R \vdash \emptyset} \qquad \frac{R \vdash M \qquad \ell \notin \mathsf{dom}(M) \qquad \imath \in R \qquad R; \emptyset \vdash \tau}{R \vdash M, \ell \mapsto (\tau, \imath)}$$

B.4 Type safety

Type Safety: Evaluation Context Typing

$$\frac{\vdash R; M; \Delta; \Gamma \qquad R; \Delta \vdash \tau}{R; M; \Delta; \Gamma \vdash \Box : \tau \longrightarrow \tau \,\&\, \emptyset} \textit{ (E0)} \qquad \frac{R; M; \Delta; \Gamma \vdash E : \tau_2 \longrightarrow \tau_3 \,\&\, \gamma_2 \quad R; M; \Delta; \Gamma \vdash F : \tau_1 \longrightarrow \tau_2 \,\&\, \gamma_1}{R; M; \Delta; \Gamma \vdash E[F] : \tau_1 \longrightarrow \tau_3 \,\&\, \gamma_1 :: \gamma_2} \textit{ (E1)}$$

$$\frac{\tau \equiv \tau_1 \xrightarrow{\gamma_a} \tau_2 \qquad R; \Delta \vdash \tau \quad R; M; \Delta; \Gamma \vdash e_2 : \tau_1 \,\&\, \gamma_1}{R; M; \Delta; \Gamma \vdash \Box\ e_2 : \tau \longrightarrow \tau_2 \,\&\, \gamma_1 :: \gamma_a} \textit{ (F1)} \qquad \frac{\tau \equiv \tau_1 \xrightarrow{\gamma_a} \tau_2 \quad R; M; \Delta; \Gamma \vdash v_1 : \tau \,\&\, \emptyset}{R; M; \Delta; \Gamma \vdash v_1\ \Box : \tau_1 \longrightarrow \tau_2 \,\&\, \gamma_a} \textit{ (F2)}$$

$$\frac{R; M; \Delta, \rho; \Gamma, x : \mathtt{Rgn}(\rho) \vdash e_2 : \tau \,\&\, \gamma_1 \qquad R; \Delta \vdash r \quad R; \Delta \vdash \tau \qquad \mathtt{Live}\, r :: \mathsf{translate}(\gamma_1, \rho, (1,1,0), r) = \gamma_2}{R; M; \Delta; \Gamma \vdash \mathtt{newrgn}\ \rho, x\ @\ \Box\ \mathtt{in}\ e_2 : \mathtt{Rgn}(r) \longrightarrow \tau \,\&\, \gamma_2} \textit{ (F3)}$$

$$\frac{\vdash R; M; \Delta; \Gamma \qquad R; \Delta \vdash \forall \rho. \tau \qquad R; \Delta \vdash r \qquad r \neq \bot}{R; M; \Delta; \Gamma \vdash \Box\,[r] : \forall \rho. \tau \longrightarrow \tau[r/\rho] \,\&\, \emptyset} \textit{ (F4)}$$

$$\frac{R; M; \Delta; \Gamma \vdash e_2 : \mathtt{Rgn}(r) \,\&\, \gamma_2 \qquad R; \Delta \vdash \tau \qquad r \neq \bot}{R; M; \Delta; \Gamma \vdash \mathtt{new}\ \Box\ @\ e_2 : \tau \longrightarrow \mathtt{Ref}(\tau, r) \,\&\, \gamma_2 :: \mathtt{Live}\, r} \textit{ (F5)}$$

$$\frac{R; \Delta \vdash r \qquad R; M; \Delta; \Gamma \vdash v : \tau \,\&\, \emptyset \qquad r \neq \bot}{R; M; \Delta; \Gamma \vdash \mathtt{new}\ v\ @\ \Box : \mathtt{Rgn}(r) \longrightarrow \mathtt{Ref}(\tau, r) \,\&\, \mathtt{Live}\, r} \textit{ (F6)}$$

$$\frac{R; \Delta \vdash r \qquad R; M; \Delta; \Gamma \vdash e_2 : \tau \,\&\, \gamma_1 \qquad r \neq \bot}{R; M; \Delta; \Gamma \vdash \Box := e_2 : \mathtt{Ref}(\tau, r) \longrightarrow \mathtt{unit} \,\&\, \gamma_1 :: \mathtt{W}\, r} \textit{ (F7)}$$

$$\frac{R; M; \Delta; \Gamma \vdash \mathtt{loc}_\ell : \mathtt{Ref}(\tau, \imath) \,\&\, \emptyset \qquad \imath \neq \bot}{R; M; \Delta; \Gamma \vdash \mathtt{loc}_\ell := \Box : \tau \longrightarrow \mathtt{unit} \,\&\, \mathtt{W}\, \imath} \textit{ (F8)}$$

$$\frac{\vdash R; M; \Delta; \Gamma \qquad R; \Delta \vdash \mathtt{Ref}(\tau, r) \qquad r \neq \bot}{R; M; \Delta; \Gamma \vdash \mathtt{deref}\ \Box : \mathtt{Ref}(\tau, r) \longrightarrow \tau \,\&\, \mathtt{R}\, r} \textit{ (F9)}$$

$$\frac{\vdash R; M; \Delta; \Gamma \qquad R; \Delta \vdash \mathtt{Rgn}(r) \qquad r \neq \bot}{R; M; \Delta; \Gamma \vdash \mathtt{cap}_\eta\ \Box : \mathtt{Rgn}(r) \longrightarrow \mathtt{unit} \,\&\, \mathtt{Cap}\,\{r \mapsto \eta\}} \textit{ (F10)}$$

$$\frac{R; M; \Delta; \Gamma \vdash e_1 : \tau \,\&\, \gamma_2 \qquad R; M; \Delta; \Gamma \vdash e_2 : \tau \,\&\, \gamma_3}{R; M; \Delta; \Gamma \vdash \mathtt{if}\ \Box\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 : \mathtt{bool} \longrightarrow \tau \,\&\, \mathtt{Join}\, \gamma_2\, \gamma_3} \textit{ (F11)}$$

Predicate “cvalid”

$$\frac{}{\mathsf{cvalid}(\mathtt{Live}; \bot; \theta)} \textit{ (C-T)}$$

$$\frac{\theta = \theta', \imath \mapsto (\eta, \jmath) \qquad \mathsf{solve}(\delta, \jmath, \eta) = \emptyset}{\mathsf{cvalid}(\delta; \imath; \theta)} \textit{ (C-B)}$$

$$\frac{\theta = \theta', \imath \mapsto (\eta, \jmath) \qquad \mathsf{solve}(\delta, \jmath, \eta) = \delta'\, \jmath \qquad \mathsf{cvalid}(\delta'; \jmath; \theta')}{\mathsf{cvalid}(\delta; \imath; \theta)} \textit{ (C-R)}$$

Predicate “valid”

$$\frac{}{\mathsf{xvalid}(\emptyset;\theta)=\theta}\ \textit{(X-E)} \qquad \frac{\mathsf{ok}(\eta+\eta') \quad \mathsf{cvalid}(\texttt{Live}\,;\imath;\theta,\imath\mapsto(\eta,\jmath)) \quad \mathsf{xvalid}(\xi;\theta,\imath\mapsto(\eta+\eta',\jmath))=\theta'}{\mathsf{xvalid}(\xi,\imath\mapsto\eta';\theta,\imath\mapsto(\eta,\jmath))=\theta'}\ \textit{(X-S)}$$

$$\frac{\mathsf{ok}(\theta)}{\mathsf{gvalid}(\emptyset;\theta)=\theta}\ \textit{(V-E)} \qquad \frac{\mathsf{ok}(\theta_1) \quad \mathsf{evalid}(\zeta;\theta_1)=\theta_2 \quad \mathsf{gvalid}(\gamma;\theta_2)=\theta_3}{\mathsf{gvalid}(\zeta::\gamma;\theta_1)=\theta_3}\ \textit{(V-K)}$$

$$\frac{\mathsf{cvalid}(\delta;\imath;\theta)}{\mathsf{evalid}(\delta\,\imath;\theta)=\theta}\ \textit{(V-D)} \qquad \frac{\mathsf{xvalid}(\mathsf{merge}(\xi);\theta)=\theta'}{\mathsf{evalid}(\texttt{Cap}\,\xi;\theta)=\theta'}\ \textit{(V-C)}$$

$$\frac{\mathsf{gvalid}(\gamma_1;\theta)=\theta' \quad \mathsf{gvalid}(\gamma_2;\theta)=\theta'}{\mathsf{evalid}(\texttt{Join}\,\gamma_1\,\gamma_2;\theta)=\theta'}\ \textit{(V-J)}$$

$$\frac{\forall\imath\in\mathsf{dom}(\theta_s).\ \mathsf{cvalid}(\texttt{Live}\,;\imath;\theta) \quad \mathsf{merge}(\xi)\vdash\theta=\theta_r\oplus\theta_s \quad \mathsf{valid}(\gamma_s;\theta_s) \quad \mathsf{mutex}(\{\theta_s,\theta_r\})}{\mathsf{evalid}(\texttt{Spawn}\,\xi\,\gamma_s;\theta)=\theta_r}\ \textit{(V-S)}$$

$$\frac{\mathsf{gvalid}(\gamma;\theta)=\theta' \quad \mathsf{live}(\theta')=\emptyset}{\mathsf{valid}(\gamma;\theta)}\ \textit{(V-V)}$$

Configuration typing

$$\frac{}{R;M\vdash\emptyset} \qquad \frac{R;M\vdash T \quad R;M;\emptyset;\emptyset\vdash e:\texttt{unit}\,\&\,\gamma \quad \mathsf{valid}(\gamma;\theta) \quad \forall\imath\mapsto(\eta,\jmath)\in\theta.\ \imath\in R\wedge\jmath\in R\cup\{\bot\}}{R;M\vdash T,\langle\theta;e\rangle}$$

$$\frac{R=\{\imath\mid\imath\mapsto H\in S\} \quad \{(\ell,\imath)\mid\ell\mapsto(\tau,\imath)\in M\}=\{(\ell,\imath)\mid\ell\mapsto v\in H\wedge\imath\mapsto H\in S\} \quad \forall\ell\mapsto(\tau,\imath)\in M.\ R;M;\emptyset;\emptyset\vdash S(\imath)(\ell):\tau\,\&\,\emptyset}{R;M\vdash S}$$

$$\frac{R;M\vdash T \quad R;M\vdash S \quad \mathsf{mutex}(\{\theta\mid\langle\theta;e\rangle\in T\})}{R;M\vdash S;T}$$

Predicate “Not stuck”

$$\frac{S;T,\langle\theta;e\rangle\rightsquigarrow S';T' \quad T\subseteq T'}{\mathsf{running}(S;T,\langle\theta;e\rangle;\langle\theta;e\rangle)}$$

$$\frac{\iota \in \mathsf{live}(\theta, \iota \mapsto (\eta, \jmath)) \qquad \mathsf{mutex}(\{\theta, \iota \mapsto (\eta, \jmath)\} \cup \{\theta' \mid \langle \theta'; e' \rangle \in T\}) \quad \neg\mathsf{mutex}(\{\theta, \iota \mapsto (\eta + \eta', \jmath)\} \cup \{\theta' \mid \langle \theta'; e' \rangle \in T\})}{\mathsf{blocked}(T; \langle \theta, \iota \mapsto (\eta, \jmath); E[\mathtt{cap}_{\eta'}\ \mathtt{rgn}_\iota] \rangle)}$$

$$\frac{\forall \langle \theta; e \rangle \in T.\ \mathsf{running}(S; T; \langle \theta; e \rangle) \vee \mathsf{blocked}(T; \langle \theta; e \rangle)}{\vdash S; T}$$

Multi-step evaluation

$$\frac{n > 0 \qquad S; T \rightsquigarrow^{n-1} S_{n-1}; T_{n-1} \qquad S_{n-1}; T_{n-1} \rightsquigarrow S_n; T_n}{S; T \rightsquigarrow^{n} S_n; T_n}\ \textit{(E-M1)}$$

$$\frac{}{S; T \rightsquigarrow^{0} S; T}\ \textit{(E-M2)}$$

Other predicates

$$\frac{}{\emptyset - \emptyset = \emptyset}\ \textit{(DF-0)} \qquad \frac{\eta_1 \geq \eta_2 \qquad \theta_1 - \theta_2 = \theta'}{\theta_1, r \mapsto \eta_1 - \theta_2 \mapsto \eta_2 = \theta', r \mapsto \eta_1 - \eta_2}\ \textit{(DF-1)}$$

B.5 Proof of soundness

Assume that e is the expression that represents the initial program. Let $S_0 = \emptyset$ be the initial empty store and $T_0 = \emptyset, \langle \emptyset; e \rangle$ be the initial set of threads, consisting of just e with an empty region hierarchy. We are interested only in programs that are closed, well typed and whose effect is consistent with the initial empty region hierarchy.

Theorem B.1 (Type Safety) Let e be such that $\emptyset; \emptyset; \emptyset; \emptyset \vdash e : \mathtt{unit}\ \&\ \emptyset$. If the operational semantics takes any number of steps $S_0; T_0 \rightsquigarrow^n S_n; T_n$, then the resulting configuration $S_n; T_n$ is not stuck.

Proof. Given $\emptyset; \emptyset; \emptyset; \emptyset \vdash e : \mathtt{unit}\ \&\ \emptyset$, $S_0 = T_0 = \emptyset$ and the definitions of store typing and thread typing it is immediate that $\emptyset; \emptyset \vdash \emptyset; \emptyset, \langle \emptyset; e \rangle$ holds (i.e, the initial configuration is well-typed). The application of Lemma B.1 to the assumption implies that $R_n; M_n \vdash S_n; T_n$. Therefore, $S_n; T_n$ is well-typed for some $R_n; M_n$. The application of Lemma B.27 to $R_n; M_n \vdash S_n; T_n$ implies $S_n; T_n$ is not stuck.

Lemma B.1 (Multi-step Program Preservation) Let $S_0; T_0$ be a *well-typed configuration* for some $R_0; M_0$ and assume that $S_0; T_0$ evaluates to $S_n; T_n$ in n steps. Then $R_n; M_n \vdash S_n; T_n$ holds.

Proof. Proof by induction on the number of steps n. When no steps are performed (i.e., $n = 0$) the proof is immediate from the assumption. When some steps are performed (i.e., $n > 0$), we have that $S_0; T_0 \rightsquigarrow^n S_n; T_n$ or $S_0; T_0 \rightsquigarrow^{n-1} S_{n-1}; T_{n-1}$ and $S_{n-1}; T_{n-1} \rightsquigarrow S_n; T_n$. By applying the induction hypothesis on the fact that $S_0; T_0$ is well-typed and that $n-1$ steps are performed we obtain that $R_{n-1}; M_{n-1} \vdash S_{n-1}; T_{n-1}$. The application of Lemma B.2 to $R_{n-1}; M_{n-1} \vdash S_{n-1}; T_{n-1}$ and $R_{n-1}; S_{n-1}; T_{n-1} \rightsquigarrow S_n; T_n$. implies that $R_n; M_n \vdash S_n; T_n$. Therefore, $R_n; M_n \vdash S_n; T_n$.

Lemma B.2 (Preservation) Let $S;T$ be a well-typed configuration with $R;M \vdash S;T$. If the operational semantics takes a step $S;T \rightsquigarrow S';T'$, then there exist $R' \supseteq R$ and $M' \supseteq M$ such that the resulting configuration is well-typed with $R';M' \vdash S';T'$.

Proof. By induction on the thread evaluation relation:

Case *E-T*: Rule *E-T* implies that $\theta; E[e] = \theta; \Box[()]$, $S' = S$ and $T' = T$, $\mathsf{live}(\theta) = \emptyset$. By inversion of the configuration typing assumption we have that:

- $R;M \vdash T, \langle \theta; \Box[()]\rangle$: by inversion of this derivation we have $R;M \vdash T$.
- $R;M \vdash S$
- $\mathsf{mutex}(\{\theta' \mid \langle \theta'; e'\rangle \in T, \langle \theta; \Box[()]\}\rangle)$: implies that $\mathsf{mutex}(\{\theta' \mid \langle \theta'; e'\rangle \in T\})$.

Given the above facts, $R;M \vdash S;T$ holds.

Case *E-A*: Rule *E-A* implies that $\theta' = \theta$, $S' = S$, $T' = T, \langle \theta; E[e_1[v/x]]\rangle$ and $u = (\lambda x.\, e_1)\ v$.

By inversion of the configuration typing assumption we have that:

- $R;M \vdash S$
- $\mathsf{mutex}(\{\theta' \mid \langle \theta'; e'\rangle \in T, \langle \theta; E[u]\}\rangle)$: no new locks are acquired ($\theta' = \theta$). Thus, $\mathsf{mutex}(\{\theta' \mid \langle \theta'; e'\rangle \in T, \langle \theta; E[e_1[v/x]]\}\rangle)$ holds.
- $R;M \vdash T, \langle \theta; E[u]\rangle$: by inversion of this derivation we have that:
  - $R;M \vdash T$, $\mathsf{valid}(\gamma;\theta)$ and $\forall \imath \mapsto (\eta, \jmath) \in \theta.\ \imath \in R \wedge \jmath \in R \cup \{\bot\}$.
  - $R;M;\emptyset;\emptyset \vdash E[u] : \mathtt{unit}\ \&\ \gamma$: The application of Lemma B.23 to the typing derivation of $E[u]$ yields $R;M;\emptyset;\emptyset \vdash E : \tau_2 \longrightarrow \mathtt{unit}\ \&\ \gamma_b$, $R;M;\emptyset;\emptyset \vdash u : \tau_2\ \&\ \gamma_a$, where $\gamma = \gamma_a :: \gamma_b$. By inversion of the latter derivation we have that $R;M;\emptyset;\emptyset \vdash v : \tau_1\ \&\ \emptyset$, $R;M;\emptyset;\emptyset \vdash \lambda x.\, e_1 : \tau_1 \xrightarrow{\gamma_a} \tau_2\ \&\ \emptyset$. By inversion of the function typing derivation we obtain that $R;M;\emptyset;\emptyset, x : \tau_1 \vdash e_1 : \tau_2\ \&\ \gamma_a$. Lemma B.12 implies that $R;M;\emptyset;\emptyset \vdash e_1[v/x] : \tau_2\ \&\ \gamma_a$ holds. The application of Lemma B.22 yields $R;M;\emptyset;\emptyset \vdash E[e_1[v/x]] : \mathtt{unit}\ \&\ \gamma$.

Case *E-FX*: Rule *E-FX* implies $S;T, \langle \theta; E[u]\rangle \rightsquigarrow S;T, \langle \theta; E[u']\rangle$ holds, where $u = (\mathtt{fix}\ x.\, f)\ v$ and $u' = f[\mathtt{fix}\ x.\, f/x]\ v$. By inversion of the configuration typing assumption we have that:

- $R;M \vdash S$
- $\mathsf{mutex}(\{\theta' \mid \langle \theta'; e'\rangle \in T, \langle \theta; E[u]\}\rangle)$: no new locks are acquired. Thus, $\mathsf{mutex}(\{\theta' \mid \langle \theta'; e'\rangle \in T, \langle \theta; E[u']\}\rangle)$ holds.
- $R;M \vdash T, \langle \theta; E[u]\rangle$: by inversion of this derivation we obtain
  - $R;M \vdash T$ and $\forall \imath \mapsto (\eta, \jmath) \in \theta.\ \imath \in R \wedge \jmath \in R \cup \{\bot\}$.
  - $R;M;\emptyset;\emptyset \vdash E[u] : \mathtt{unit}\ \&\ \gamma$: The application of Lemma B.23 to the typing derivation of $E[u]$ yields $R;M;\emptyset;\emptyset \vdash E : \tau_2 \longrightarrow \mathtt{unit}\ \&\ \gamma_c$ and $R;M;\emptyset;\emptyset \vdash u : \tau_2\ \&\ \gamma_a$, where $\gamma = \gamma_a :: \gamma_c$. By inversion of the latter derivation we have that $R;M;\emptyset;\emptyset \vdash v : \tau_1\ \&\ \emptyset$, and $R;M;\emptyset;\emptyset \vdash \mathtt{fix}\ x.\, f : \tau\ \&\ \emptyset$, where τ equals $\tau_1 \xrightarrow{\gamma_a} \tau_2$. By inversion of the typing derivation of $\mathtt{fix}\ x.\, f$ we obtain that $R;M;\emptyset;\emptyset, x : \tau \vdash f : \tau'\ \&\ \emptyset$, where $\gamma_a = \mathsf{summary}(\gamma_b)$ and $\tau' = \tau_1 \xrightarrow{\gamma_b} \tau_2$. Lemma B.12 implies that $R;M;\emptyset;\emptyset \vdash f[\mathtt{fix}\ x.\, f/x] : \tau'\ \&\ \emptyset$ holds. The typing derivation of $f[\mathtt{fix}\ x.\, f/x]$ and v and rule *T-A* imply that $R;M;\emptyset;\emptyset \vdash (f[\mathtt{fix}\ x.\, f/x])\ v : \tau_2\ \&\ \gamma_b$. The application of Lemma B.22 yields $R;M;\emptyset;\emptyset \vdash E[(f[\mathtt{fix}\ x.\, f/x])\ v] : \mathtt{unit}\ \&\ \gamma_b :: \gamma_c$ holds.
  - $\mathsf{valid}(\gamma;\theta)$: it suffices to prove that $\mathsf{valid}(\phi(\gamma_s) :: \gamma_c;\theta)$ holds. The assumptions imply that $\gamma = \mathsf{summary}(\phi(\gamma_L)) :: \gamma_c$ and therefore $\mathsf{valid}(\mathsf{summary}(\phi(\gamma_L)) :: \gamma_c;\theta)$ holds. The application of Lemma B.3 completes the proof.

Case *E-SP*: Rule *E-SP* implies that $S' = S,\ \ T' = T, \langle \theta'; E[()]\rangle, \theta''; \Box[e_1]$, where $\mathsf{merge}(\xi) \vdash \theta = \theta' \oplus \theta''$ and $u = \mathtt{spawn}_{\xi}\ e_1$ hold.

By inversion of the configuration typing assumption we have that:

- $R; M \vdash S$
- $R; M \vdash T, \langle \theta; E[u]\rangle$: by inversion of this derivation we have that:
  - $R; M \vdash T$, $\mathsf{valid}(\gamma; \theta)$ and $\forall \imath \mapsto (\eta, \jmath) \in \theta.\ \imath \in R \wedge \jmath \in R \cup \{\bot\}$.
  - $R; M; \emptyset; \emptyset \vdash E[u] : \mathtt{unit}\ \&\ \gamma$: The application of Lemma B.23 to the typing derivation of $E[u]$ yields $R; M; \emptyset; \emptyset \vdash E : \mathtt{unit} \longrightarrow \mathtt{unit}\ \&\ \gamma_b$, $R; M; \emptyset; \emptyset \vdash \mathtt{spawn}_{\xi}\ e_1 : \mathtt{unit}\ \&\ \gamma_a$, where γ_1 is the effect of expression e_1, $\gamma_a = \mathtt{Spawn}\ \xi\ \gamma_1$ and $\gamma = \gamma_a :: \gamma_b$. The typing derivation for the unit value can be obtained by establishing that the typing context is well-formed (i.e., by the application of Lemma B.11 to the typing of derivation of u). The application of Lemma B.22 yields $R; M; \emptyset; \emptyset \vdash E[()] : \mathtt{unit}\ \&\ \gamma_b$.
  - $R; M; \emptyset; \emptyset \vdash e_1 : \mathtt{unit}\ \&\ \gamma_1$: by inversion of $R; M; \emptyset; \emptyset \vdash \mathtt{spawn}_{\gamma_1}\ e_1 : \mathtt{unit}\ \&\ \gamma_a$ we obtain that $R; M; \emptyset; \emptyset \vdash e_1 : \mathtt{unit}\ \&\ \gamma_1$. The application of rules *E0,D0* implies that $R; M; \emptyset; \emptyset \vdash \Box[e_1] : \mathtt{unit}\ \&\ \gamma_1$.
  - $\mathsf{mutex}(\{\theta_0 \mid \langle \theta_0; e_0\rangle \in T, \langle \theta; E[u]\rangle\})$: by inversion of $\mathsf{valid}(\gamma; \theta)$ we have that $\mathsf{mutex}(\{\theta', \theta''\})$ and $\mathsf{merge}(\xi) \vdash \theta = \theta' \oplus \theta''$. Notice that the above imply that $\mathsf{rwlocked}(\theta') \cup \mathsf{rwlocked}(\theta'') \subseteq \mathsf{rwlocked}(\theta)$. Therefore, $\mathsf{mutex}(\{\theta_0 \mid \langle \theta_0; e_0\rangle \in T'\})$ holds by the above facts and the assumption $\mathsf{mutex}(\{\theta_0 \mid \langle \theta_0; e_0\rangle \in T, \langle \theta; E[u]\rangle\})$.
  - $\mathsf{valid}(\gamma_1; \theta'')$: immediate by inversion of $\mathsf{valid}(\gamma; \theta)$.
  - $\mathsf{valid}(\gamma_b; \theta')$: immediate by inversion of $\mathsf{valid}(\gamma; \theta)$.

Case *E-NR*: Rule *E-NR* implies that $\imath$ is fresh (i.e., it does not belong in R), $\theta' = \theta, \imath \mapsto ((1,1,0), \jmath)$ $S' = S, \imath \mapsto \emptyset$, $\jmath \in \mathsf{live}(\theta) \cup \{\bot\}$ holds, $T' = S'; T, \langle \theta'; E[e_2[\imath/\rho][\mathtt{rgn}_{\imath}/x]]\rangle$, where u equals $\mathtt{newrgn}\ \rho, x\ @\ \mathtt{rgn}_{\jmath}\ \mathtt{in}\ e_2$.

By inversion of the configuration typing assumption we have that:

- $R, \imath; M \vdash S'$ is immediate from $R; M \vdash S$, the fact that $\imath$ is fresh and its heap is empty.
- $\mathsf{mutex}(\{\theta_0 \mid \langle \theta_0; e_0\rangle \in T, \langle \theta; E[u]\rangle\})$:
  $\mathsf{mutex}(\{\theta_0 \mid \langle \theta_0; e_0\rangle \in T'\})$ holds from $\mathsf{mutex}(\{\theta_0 \mid \langle \theta_0; e_0\rangle \in T, \langle \theta; E[u]\rangle\})$ and the fact that no other thread in T has $\imath$ in its local hierarchy (i.e., even if one of $\imath$'s ancestors is locked by a thread in T, $\imath$ is not locked by that thread as it does not exist in its local hierarchy).
- $R; M \vdash T, \langle \theta; E[u]\rangle$: by inversion of this derivation we have that:
  - $R; M \vdash T$, $\mathsf{valid}(\gamma; \theta)$ and $\forall \imath \mapsto (\eta, \jmath) \in \theta.\ \imath \in R \wedge \jmath \in R \cup \{\bot\}$.
  - $R; M; \emptyset; \emptyset \vdash E[u] : \mathtt{unit}\ \&\ \gamma$: The application of Lemma B.23 to the typing derivation of $E[u]$ yields $R; M; \emptyset; \emptyset \vdash E : \tau_1 \longrightarrow \mathtt{unit}\ \&\ \gamma_b$, $R; M; \emptyset; \emptyset \vdash u : \tau_1\ \&\ \gamma_a$, where γ_a equals $\mathtt{Live}\ \jmath :: \mathsf{translate}(\gamma_2, \rho, 1, 1, 0, \jmath)$ and $\gamma = \gamma_a :: \gamma_b$. By inversion of the derivation of u we have that $R; \emptyset \vdash \tau_1$, $R; M; \emptyset; \emptyset \vdash \mathtt{rgn}_{\jmath} : \mathtt{Rgn}(\jmath)\ \&\ \emptyset$ and $R; M; \emptyset, \rho; \emptyset, x : \mathtt{Rgn}(\rho) \vdash e_2 : \tau_1\ \&\ \gamma_2$. Lemma B.16 implies that $R, \imath; M; \emptyset, \rho; \emptyset, x : \mathtt{Rgn}(\rho) \vdash e_2 : \tau_1\ \&\ \gamma_2$. Lemmata B.15 and B.12 imply that $R, \imath; M; \emptyset; \emptyset \vdash e_2[\imath/\rho][\mathtt{rgn}_{\imath}/x] : \tau_1\ \&\ \gamma_2[\imath/\rho]$ (notice that $R; \emptyset \vdash \tau_1$ implies that $\tau_1[\imath/\rho] = \tau_1$). The application of Lemma B.17 to the typing derivation of E implies that $R, \imath; M; \emptyset; \emptyset \vdash E : \tau_1 \longrightarrow \mathtt{unit}\ \&\ \gamma_b$. The application of Lemma B.22 yields $R, \imath; M; \emptyset; \emptyset \vdash E\,[e_2[\imath/\rho][\mathtt{rgn}_{\imath}/x]] : \mathtt{unit}\ \&\ \gamma_2[\imath/\rho] :: \gamma_b$ holds.
  - by inversion of $\mathsf{valid}(\mathtt{Live}\ \jmath :: \mathsf{translate}(\gamma_2, \rho, (1,1,0), \jmath) :: \gamma_b; \theta)$ we have that there exists a θ'' such that $\mathsf{gvalid}\ (\mathsf{translate}(\gamma_2, \rho, (1,1,0), \jmath) :: \gamma_b; \theta) = \theta''$ and $\mathsf{live}(\theta'') = \emptyset$. Therefore, $\mathsf{valid}\ (\mathsf{translate}\ (\gamma_2, \rho, (1,1,0)\ , \jmath) :: \gamma_b; \theta)$ holds. Hence $\mathsf{valid}\ (\mathsf{translate}\ (\gamma_2[\imath/\rho], \imath, (1,1,0)\ , \jmath) :: \gamma_b; \theta)$ holds. The translation function transforms effects containing $\imath$. Such effects only exist in $\gamma_2[\imath/\rho]$ (this can be shown by lemma B.14) and not

in γ_b. Therefore, $\mathsf{valid}(\mathsf{translate}(\gamma_2[\iota/\rho] :: \gamma_b, \iota, (1,1,0), \jmath); \theta)$ also holds. Lemma B.26 implies that $\mathsf{valid}(\gamma_2[\iota/\rho] :: \gamma_b; \theta, \iota \mapsto ((1,1,0), \jmath))$.

Case *E-CP*: Rule *E-CP* implies that $\iota \in \mathsf{live}(\theta)$, $\theta = \theta'', \iota \mapsto (\eta, \jmath)$, $\theta' = \theta'', \iota \mapsto (\eta + \eta_0, \jmath)$, $\mathsf{mutex}(\{\theta'\} \cup \{\theta'' \mid \langle \theta''; e' \rangle \in T\})$, $S' = S$, $T' = T, \langle \theta; E[()] \rangle$, and $u = \mathtt{cap}_\eta\ \mathtt{rgn}_\iota$.

By inversion of the configuration typing assumption we have that:

- $R; M \vdash S$
- $\mathsf{mutex}(\{\theta'\} \cup \{\theta'' \mid \langle \theta''; e' \rangle \in T\})$: immediate by the premises of rule *E-CP*.
- $R; M \vdash T, \langle \theta; E[u] \rangle$: by inversion of this derivation we have that:
  - $R; M \vdash T$, $\mathsf{valid}(\gamma; \theta)$ and $\forall \iota \mapsto (\eta, \jmath) \in \theta.\ \iota \in R \wedge \jmath \in R \cup \{\bot\}$.
  - $R; M; \emptyset; \emptyset \vdash E[u] : \mathtt{unit}\ \&\ \gamma$: The application of Lemma B.23 to the typing derivation of $E[u]$ yields $R; M; \emptyset; \emptyset \vdash E : \mathtt{unit} \longrightarrow \mathtt{unit}\ \&\ \gamma_b$, $R; M; \emptyset; \emptyset \vdash \mathtt{cap}_{\eta_0}\ \mathtt{rgn}_\iota : \mathtt{unit}\ \&\ \gamma_a$, where $\gamma_a = \mathtt{Cap}\ \{\iota \mapsto \eta_0\}$ and $\gamma = \gamma_a :: \gamma_b$. The typing derivation for the unit value can be obtained by establishing that the typing context is well-formed (i.e., by the application of Lemma B.11 to the typing of derivation of u). The application of Lemma B.22 yields $R; M; \emptyset; \emptyset \vdash E[()] : \mathtt{unit}\ \&\ \gamma_b$.
  - by inversion of $\mathsf{valid}(\mathtt{Cap}\ \{\iota \mapsto \eta_0\} :: \gamma_b; \theta)$ we have that $\mathsf{xvalid}\ (\mathtt{Cap}\ \{\iota \mapsto \eta_0\}; \theta) = \theta'$, $\mathsf{gvalid}(\gamma_b; \theta') = \theta''$ and $\mathsf{live}(\theta'') = \emptyset$ for some θ''. Therefore, $\mathsf{valid}(\gamma_b; \theta')$ holds.

Case *E-AS*: Rule *E-AS* implies that $\theta' = \theta$, $S' = S[\iota : S(\iota)[\ell \mapsto v]]$, $\ell \mapsto v' \in S(\iota)$, $T' = T, \langle \theta; E[()] \rangle$, and $u = \mathtt{loc}_\ell := v$.

By inversion of the configuration typing assumption we have that:

- $\mathsf{mutex}(\{\theta_0 \mid \langle \theta_0; e_0 \rangle \in T, \langle \theta; E[u] \rangle\})$: $\mathsf{mutex}(\{\theta_0 \mid \langle \theta_0; e_0 \rangle \in T, \langle \theta; E[()] \rangle\})$ trivially holds as no new locks are acquired.
- $R; M \vdash T, \langle \theta; E[u] \rangle$: by inversion of this derivation we have that:
  - $R; M \vdash T$, $\mathsf{valid}(\gamma; \theta)$ and $\forall \iota \mapsto (\eta, \jmath) \in \theta.\ \iota \in R \wedge \jmath \in R \cup \{\bot\}$.
  - $R; M; \emptyset; \emptyset \vdash E[u] : \mathtt{unit}\ \&\ \gamma$: The application of Lemma B.23 to the typing derivation of $E[u]$ yields $R; M; \emptyset; \emptyset \vdash E : \mathtt{unit} \longrightarrow \mathtt{unit}\ \&\ \gamma_b$, $R; M; \emptyset; \emptyset \vdash \mathtt{loc}_\iota := v : \mathtt{unit}\ \&\ \gamma_a$, where $\gamma_a = \mathtt{W}\,\iota$ and $\gamma = \gamma_a :: \gamma_b$. The typing derivation for the unit value can be obtained by establishing that the typing context is well-formed (i.e., by the application of Lemma B.11 to the typing of derivation of u). The application of Lemma B.22 yields $R; M; \emptyset; \emptyset \vdash E[()] : \mathtt{unit}\ \&\ \gamma_b$.
  - by inversion of $\mathsf{valid}(\gamma; \theta)$ and the fact that $\theta' = \theta$ it is immediate that $\mathsf{valid}(\gamma_b; \theta')$ holds.
- $R; M \vdash S$: By inversion of $R; M; \emptyset; \emptyset \vdash \mathtt{loc}_\iota := v : \mathtt{unit}\ \&\ \gamma_a$ we obtain that $R; M; \emptyset; \emptyset \vdash v : \tau\ \&\ \emptyset$ and $R; M; \emptyset; \emptyset \vdash \mathtt{loc}_\ell : \mathtt{Ref}(\tau, \iota)\ \&\ \emptyset$ (i.e., $\ell \mapsto (\tau, \iota) \in M$). Given the above facts, the definition of S' and $R; M \vdash S$ we can conclude that $R; M \vdash S'$ holds.

Case *E-D*: similar to the previous case.

Case *E-NL*: similar to the previous case.

Case *E-RP*: Rule *E-RP* implies that $\theta' = \theta$, $S' = S$, $T' = T, \langle \theta; E[f[\iota/\rho]] \rangle$, where u is equal to $(\Lambda\rho.\ f)\ [\iota]$.

By inversion of the configuration typing assumption we have that:

- $R; M \vdash S$
- $\mathsf{mutex}(\{\theta_0 \mid \langle \theta_0; e_0 \rangle \in T, \langle \theta; E[u] \rangle\})$: $\mathsf{mutex}(\{\theta_0 \mid \langle \theta_0; e_0 \rangle \in T, \langle \theta; E[f[\iota/\rho]] \rangle\})$ trivially holds as no new locks are acquired.

- $R; M \vdash T, \langle \theta; E[u] \rangle$: by inversion of this derivation we have that:
  - $R; M \vdash T$, $\mathsf{valid}(\gamma; \theta)$ and $\forall \imath \mapsto (\eta, \jmath) \in \theta.\ \imath \in R \wedge \jmath \in R \cup \{\bot\}$.
  - $R; M; \emptyset; \emptyset \vdash E[u] : \mathtt{unit} \,\&\, \gamma$: The application of Lemma B.23 to the typing derivation of $E[u]$ yields $R; M; \emptyset; \emptyset \vdash E : \tau[\imath/\rho] \longrightarrow \mathtt{unit} \,\&\, \gamma$, $R; M; \emptyset; \emptyset \vdash (\Lambda\rho.\ f)\ [\imath] : \tau[\imath/\rho] \,\&\, \emptyset$. By inversion of the latter derivation we obtain that $R; M; \emptyset, \rho; \emptyset \vdash f : \tau \,\&\, \emptyset$ and $R; \emptyset \vdash \imath$. Lemma B.15 implies that $R; M; \emptyset; \emptyset \vdash f[\imath/\rho] : \tau[\imath/\rho] \,\&\, \emptyset$. The application of Lemma B.22 yields $R; M; \emptyset; \emptyset \vdash E[f[\imath/\rho]] : \mathtt{unit} \,\&\, \gamma$.

Case *E-IT*: Rule *E-IT* implies that $\theta' = \theta, S' = S, T' = T, \langle \theta; E[e_1] \rangle$, where u is equal to `if true then` e_1 `else` e_2.

By inversion of the configuration typing assumption we have that:

- $R; M \vdash S$
- $\mathsf{mutex}(\{\theta_0 \mid \langle \theta_0; e_0 \rangle \in T, \langle \theta; E[u] \rangle\})$: $\mathsf{mutex}(\{\theta_0 \mid \langle \theta_0; e_0 \rangle \in T, \langle \theta; E[e_1] \rangle\})$ trivially holds as no new locks are acquired.
- $R; M \vdash T, \langle \theta; E[u] \rangle$: by inversion of this derivation we have that:
  - $R; M \vdash T$, $\mathsf{valid}(\gamma; \theta)$ and $\forall \imath \mapsto (\eta, \jmath) \in \theta.\ \imath \in R \wedge \jmath \in R \cup \{\bot\}$.
  - $R; M; \emptyset; \emptyset \vdash E[u] : \mathtt{unit} \,\&\, \gamma$: The application of Lemma B.23 to the typing derivation of $E[u]$ yields $R; M; \emptyset; \emptyset \vdash E : \tau \longrightarrow \mathtt{unit} \,\&\, \gamma_b$, $R; M; \emptyset; \emptyset \vdash$ `if true then` e_1 `else` $e_2 : \tau \,\&\, \gamma_a$, where $\gamma_a = \mathtt{Join}\, \gamma_1\, \gamma_2$, and $\gamma = \gamma_a :: \gamma_b$. By inversion of the latter derivation we obtain the typing derivation for e_1. The application of Lemma B.22 yields $R; M; \emptyset; \emptyset \vdash E[e_1] : \mathtt{unit} \,\&\, \gamma_1 :: \gamma_b$.
  - $\mathsf{valid}(\gamma_1 :: \gamma_b; \theta)$: immediate by inversion of $\mathsf{valid}(\gamma; \theta)$.

Case *E-IF*: similar to the previous case.

Lemma B.3 (Recursion preserves valid) If $\gamma_L = \{\mathtt{Live}\, r \mid r \in \mathsf{dom}(\phi(\emptyset))\}$, $\gamma_s = \mathsf{summary}(\phi(\gamma_L))$, and $\mathsf{valid}(\gamma_s :: \gamma; \theta)$, then $\mathsf{valid}(\phi(\gamma_s) :: \gamma; \theta)$.

Proof. By inversion of the second assumption, we have $\mathsf{recursive}(\xi_1; \phi(\gamma_L)) = \xi_1$, where $\xi_1 = \{r \mapsto (1, 0, 0) \mid r \in \mathsf{dom}(\phi(\gamma_L))\}$. Also, by inversion of the third assumption, we have that $\mathsf{gvalid}(\gamma_s :: \gamma; \theta) = \theta'$ for some θ' such that $\mathsf{live}(\theta') = \emptyset$, and the former easily implies $\mathsf{ok}(\theta)$. Lemma B.5 implies that there exists a θ'' such that $\mathsf{gvalid}(\gamma_s; \theta) = \theta''$ and $\mathsf{gvalid}(\gamma; \theta'') = \theta'$. Lemma B.6 implies that $\theta'' = \theta$ and therefore $\mathsf{gvalid}(\gamma_s; \theta) = \theta$. Lemma B.7 then implies that there exists a θ_{live} such that $\mathsf{live}(\theta_{live}) \subseteq \mathsf{live}(\theta)$, $\mathsf{gvalid}(\phi(\gamma_L); \theta_{live}) = \theta_{live}$, $\mathsf{hierarchy\_ok}(\theta_{live}; \theta)$, and $\forall \imath \mapsto (\eta, \jmath) \in \theta_{live}.\ \eta = (1, 0, 0)$. By taking θ_r to be the same as θ but with all region counts of the regions in the domain of θ_{live} reduced by one, it is easy to verify that $\theta_r = \theta - \theta_{live}$. We can now apply Lemma B.8 and deduce that there exists a θ_x such that $\mathsf{gvalid}(\phi(\gamma_L); \theta) = \theta_x$ and $\theta_r = \theta_x - \theta_{live}$. By the definition of hierarchy subtraction, it is easy to prove that $\theta_x = \theta$, and therefore $\mathsf{gvalid}(\phi(\gamma_L); \theta) = \theta$. Lemma B.9 then easily yields $\mathsf{gvalid}(\phi(\gamma_s); \theta) = \theta$, and Lemma B.4 yields $\mathsf{gvalid}(\phi(\gamma_s) :: \gamma; \theta) = \theta'$. The proof is completed by rule V-V.

Lemma B.4 (Composition of gvalid) If $\mathsf{gvalid}(\gamma_1; \theta_1) = \theta_2$ and $\mathsf{gvalid}(\gamma_2; \theta_2) = \theta_3$, then $\mathsf{gvalid}(\gamma_1 :: \gamma_2; \theta_1) = \theta_3$.

Proof. Simple proof by induction on γ_1.

Lemma B.5 (Decomposition of gvalid) If $\mathsf{gvalid}(\gamma_1 :: \gamma_2; \theta) = \theta'$, then there exists a θ'' such that $\mathsf{gvalid}(\gamma_1; \theta) = \theta''$ and $\mathsf{gvalid}(\gamma_2; \theta'') = \theta'$.

Proof. Simple proof by induction on γ_1.

Lemma B.6 (Summary preserves hierarchy) If $\gamma_s = \mathsf{summary}(\gamma)$ and $\mathsf{gvalid}(\gamma_s; \theta_1) = \theta_2$ then $\theta_1 = \theta_2$.

Proof. Using the definition of summary, it suffices to prove that if $\mathsf{xvalid}(\xi;\theta_1) = \theta'$ and $\xi \vdash \theta' = \theta_2 \oplus \theta_s$ then $\theta_1 = \theta_2$. This is easy by induction on the derivation of $\xi \vdash \theta' = \theta_2 \oplus \theta_s$.

Lemma B.7 (Summary preserves gvalid backwards) If $\gamma_L = \{\mathtt{Live}\, r \mid r \in \mathsf{dom}(\phi(\emptyset))\}$, $\gamma_s = \mathsf{summary}(\phi(\gamma_L))$, and $\mathsf{gvalid}(\gamma_s;\theta) = \theta$, then there exists θ' such that $\mathsf{live}(\theta') \subseteq \mathsf{live}(\theta)$, $\mathsf{gvalid}(\phi(\gamma_L);\theta') = \theta'$, $\mathsf{hierarchy\_ok}(\theta';\theta)$, and $\forall \imath \mapsto (\eta, \jmath) \in \theta'.\, \eta = (1,0,0)$.

Proof. By inversion of the second assumption, we take that $\gamma_s = \mathtt{Cap}\,\xi_1 :: \mathtt{Spawn}\,\xi_1\,(\phi(\gamma_L) :: \gamma_n :: \mathtt{Cap}\,\xi_2)$ and $\mathsf{recursive}(\xi_1;\phi(\gamma_L)) = \xi_1$, where $\xi_1 = \{r \mapsto (1,0,0) \mid r \in \mathsf{dom}(\phi(\gamma_L))\}$, $\xi_2 = \{r \mapsto (-1,0,0) \mid r \in \mathsf{dom}(\phi(\gamma_L))\}$, and $\gamma_n = \{\neg\mathtt{RW}\, r \mid r \in \mathsf{dom}(\phi(\gamma_L))\}$. By a series of inversions on the third assumption, there exists some θ'' such that $\mathsf{evalid}(\mathtt{Cap}\,\xi_1;\theta) = \theta''$ and $\mathsf{evalid}(\mathtt{Spawn}\,\xi_1\,\phi(\gamma_L) :: \gamma_n :: \mathtt{Cap}\,\xi_2;\theta'') = \theta$. By inversion of the latter, there exists a θ_s such that $\mathsf{merge}(\xi_1) \vdash \theta'' = \theta \oplus \theta_s$ and $\mathsf{valid}(\phi(\gamma_L) :: \gamma_n :: \mathtt{Cap}\,\xi_2;\theta_s)$. We take $\theta' = \theta_s$, which is basically equal to the hierarchy θ restricted to $\mathsf{dom}(\phi(\gamma_L))$ and with all counts equal to $(1,0,0)$. It is easy to verify $\mathsf{live}(\theta') \subseteq \mathsf{live}(\theta)$, $\mathsf{hierarchy\_ok}(\theta';\theta)$, and $\forall \imath \mapsto (\eta,\jmath) \in \theta'.\, \eta = (1,0,0)$. Also, by inversion of $\mathsf{valid}(\phi(\gamma_L) :: \gamma_n :: \mathtt{Cap}\,\xi_2;\theta')$ and Lemma B.5, there exists a θ_x such that $\mathsf{gvalid}(\phi(\gamma_L);\theta') = \theta_x$. We can deduce $\theta_x = \theta'$ and thus conclude our proof by showing that, in general, if $\mathsf{recursive}(\gamma;\xi) = \xi$ and $\mathsf{gvalid}(\gamma;\theta) = \theta'$ then $\theta = \theta'$. This can easily be proved by induction on the derivation of the first assumption.

Lemma B.8 (Preservation of gvalid for a greater θ) If $\mathsf{gvalid}(\gamma;\theta_1) = \theta_2$, $\mathsf{recursive}(\xi;\gamma) = \xi'$, $\theta = \theta_3 - \theta_1$, $\mathsf{ok}(\theta_3)$, and $\mathsf{live}(\theta_1) \subseteq \mathsf{live}(\theta_3)$, then there exists a θ_4 such that $\mathsf{gvalid}(\gamma;\theta_3) = \theta_4$, $\theta = \theta_4 - \theta_2$, $\mathsf{ok}(\theta_4)$ and $\mathsf{live}(\theta_2) \subseteq \mathsf{live}(\theta_4)$.

Proof. By induction on the length of γ. If γ is empty, then $\theta_1 = \theta_2$; we take $\theta_4 = \theta_3$ and the proof is immediate. Otherwise, if γ is of the form $\zeta :: \gamma'$, we know from the first assumption that $\mathsf{ok}(\theta_1)$, $\mathsf{evalid}(\zeta;\theta_1) = \theta_5$, and $\mathsf{gvalid}(\gamma';\theta_5) = \theta_2$, for some θ_5. By a case analysis on ζ, we will show that there exists a θ_6 such that $\mathsf{evalid}(\zeta;\theta_3) = \theta_6$, $\theta = \theta_6 - \theta_5$, $\mathsf{ok}(\theta_6)$ and $\mathsf{live}(\theta_5) \subseteq \mathsf{live}(\theta_6)$.

Case $\mathtt{Cap}\,\xi_c$: By inversion of $\mathsf{evalid}(\zeta;\theta_1) = \theta_5$ we have $\mathsf{xvalid}(\mathsf{merge}(\xi_c);\theta_1) = \theta_5$, and therefore for each $\imath$ such that $\mathsf{cvalid}(\mathtt{Live}\,;\imath;\theta_1)$, $\imath \mapsto (\eta,\jmath)$ in θ_1 and $\imath \mapsto \eta'$ in $\mathsf{merge}(\xi_c)$, we know that $\mathsf{ok}(\eta + \eta')$ and $\imath$ is live in θ_1. Assuming that $\imath \mapsto (\eta'',\jmath)$ exists in θ_3, then $\theta = \theta_3 - \theta_1$ implies that $\eta'' \geq \eta$ therefore $\mathsf{ok}(\eta'' + \eta')$. $\mathsf{cvalid}(\mathtt{Live}\,;\imath;\theta_1)$ and $\mathsf{live}(\theta_1) \subseteq \mathsf{live}(\theta_3)$ imply that $\mathsf{cvalid}(\mathtt{Live}\,;\imath;\theta_3)$. We take θ_6 to be identical to θ_3, except for the counts which are taken equal to $\eta'' + \eta'$. It follows easily that $\theta = \theta_6 - \theta_5$, $\mathsf{evalid}(\zeta;\theta_3) = \theta_6$, $\mathsf{ok}(\theta_6)$ and $\mathsf{live}(\theta_5) \subseteq \mathsf{live}(\theta_6)$.

Case $\mathtt{Spawn}\,\xi_s\,\gamma_s$: By inversion of $\mathsf{evalid}(\zeta;\theta_1) = \theta_5$ we obtain θ_s and θ'_s, such that $\mathsf{gvalid}(\gamma_s;\theta_s) = \theta'_s$, $\mathsf{live}(\theta'_s) = \emptyset$, $\mathsf{merge}(\xi_s) \vdash \theta_1 = \theta_5 \oplus \theta_s$ (this implies that $\theta_5 = \theta_1 - \theta_s$), and $\mathsf{mutex}(\{\theta_s,\theta_5\})$. From $\mathsf{recursive}(\xi;\gamma) = \xi'$, as the spawn event was an element of this γ, we know that $\forall \imath \mapsto \eta \in \xi_s.\, \mathsf{rd}(\eta) = \mathsf{wr}(\eta) = 0$. We take θ_6 to be identical to θ_3, except for the counts which are incremented by the respective η in ξ_s. It is easy to deduce that $\mathsf{merge}(\xi_s) \vdash \theta_3 = \theta_6 \oplus \theta_s$, (equivalently $\theta_6 = \theta_3 - \theta_s$) and $\mathsf{mutex}(\{\theta_s,\theta_6\})$. Therefore, $\theta = \theta_6 - \theta_5$ and $\mathsf{evalid}(\zeta;\theta_3) = \theta_6$, $\mathsf{ok}(\theta_6)$ and $\mathsf{live}(\theta_5) \subseteq \mathsf{live}(\theta_6)$.

Case $\delta\,\imath$: By inversion of $\mathsf{evalid}(\zeta;\theta_1) = \theta_5$ we have $\theta_1 = \theta_5$ and $\mathsf{cvalid}(\delta;\imath;\theta_1)$. Therefore, we take $\theta_6 = \theta_3$ and it easily follows that $\theta = \theta_6 - \theta_5$, because of $\theta = \theta_3 - \theta_1$. It suffices to show $\mathsf{cvalid}(\delta;\imath;\theta_3)$. We proceed by performing case analysis on δ. From $\mathsf{recursive}(\xi;\gamma) = \xi'$, as the constraint event was an element of this γ, we know that δ cannot be one of $\neg\mathtt{Live}$, $\neg\mathtt{RW}$, or $\neg\mathtt{W}$. The remaining cases for δ are $\mathtt{R}$, $\mathtt{W}$ and $\mathtt{Live}$, which are all satisfied with θ_3, as all counts of θ_3 are greater than or equal to the counts of θ_1 and $\mathsf{live}(\theta_1) \subseteq \mathsf{live}(\theta_3)$.

Case $\mathtt{Join}\,\gamma_1\,\gamma_2$: By inversion of $\mathsf{evalid}(\zeta;\theta_1) = \theta_5$ we have that $\mathsf{gvalid}(\gamma_1;\theta_1) = \theta_5$ and $\mathsf{gvalid}(\gamma_2;\theta_1) = \theta_5$. The application of the induction hypothesis on these two yields $\mathsf{gvalid}(\gamma_1;\theta_3) = \theta_{x1}$ and $\mathsf{gvalid}(\gamma_2;\theta_3) = \theta_{x2}$, for some θ_{x1} and θ_{x2} such that $\theta = \theta_{x1} - \theta_5$, $\theta = \theta_{x2} - \theta_5$, $\mathsf{ok}(\theta_{x1})$,

$\mathsf{live}(\theta_5) \subseteq \mathsf{live}(\theta_{x1})$, $\mathsf{ok}(\theta_{x2})$ and $\mathsf{live}(\theta_5) \subseteq \mathsf{live}(\theta_{x2})$. It easily follows that $\theta_{x1} = \theta_{x2}$ and we take θ_6 to be equal to these two. Therefore, $\mathsf{evalid}(\zeta; \theta_3) = \theta_6$, $\mathsf{ok}(\theta_6)$ and $\mathsf{live}(\theta_5) \subseteq \mathsf{live}(\theta_6)$.

We have now shown that there exists a θ_6 such that $\mathsf{evalid}(\zeta; \theta_3) = \theta_6$, $\theta = \theta_6 - \theta_5$, $\mathsf{ok}(\theta_6)$ and $\mathsf{live}(\theta_5) \subseteq \mathsf{live}(\theta_6)$. We also know that $\mathsf{gvalid}(\gamma'; \theta_5) = \theta_2$. By application of the induction hypothesis, there exists a θ_4 such that $\mathsf{gvalid}(\gamma'; \theta_6) = \theta_4$ and $\theta = \theta_4 - \theta_2$. The proof is completed by rule *V-K*.

Lemma B.9 (Recursion preserves gvalid) If $\gamma_L = \{\mathtt{Live}\, r \mid r \in \mathsf{dom}(\phi(\emptyset))\}$, $\gamma_s = \mathsf{summary}(\phi(\gamma_L))$, $\mathsf{gvalid}(\gamma_s; \theta_0) = \theta_0$, $\mathsf{hierarchy\_ok}(\theta_1; \theta_0)$, and $\mathsf{gvalid}(\phi'(\gamma_L); \theta_1) = \theta_2$, then $\mathsf{gvalid}(\phi'(\gamma_s); \theta_1) = \theta_2$.

Proof. We suppose that ϕ' is a "compositional" function on effects, which can use its parameter in a number of places to synthesize its result. We proceed by induction on the structure of ϕ'.

Case $\phi'(\gamma) = \emptyset$: Then $\phi'(\gamma_s) = \phi'(\gamma_L)$ and the proof is immediate.

Case $\phi'(\gamma) = \gamma :: \phi''(\gamma)$: Lemma B.5 implies that $\mathsf{gvalid}(\gamma_L; \theta_1) = \theta_1$ (the resulting hierarchy is necessarily equal to θ_1, as γ_L contains only liveness constraints) and $\mathsf{gvalid}(\phi''(\gamma_L); \theta_1) = \theta_2$. $\mathsf{gvalid}(\gamma_L; \theta_1) = \theta_1$ implies $\mathsf{dom}(\gamma_s) \subseteq \mathsf{live}(\theta_1)$. Lemma B.10 implies that $\mathsf{gvalid}(\gamma_s; \theta_1) = \theta_1$. The induction hypothesis yields $\mathsf{gvalid}(\phi''(\gamma_s); \theta_1) = \theta_2$. The application of Lemma B.4 completes the proof.

Case $\phi'(\gamma) = \psi(\gamma) :: \phi''(\gamma)$, for some compositional function ψ producing events: By inversion of $\mathsf{gvalid}(\phi'(\gamma_L); \theta_1) = \theta_2$, we know that there exists a θ_3 such that $\mathsf{ok}(\theta_1)$ holds, $\mathsf{evalid}(\psi(\gamma_L); \theta_1) = \theta_3$, and $\mathsf{gvalid}(\phi''(\gamma_L); \theta_3) = \theta_2$. From $\mathsf{evalid}(\psi(\gamma_L); \theta_1) = \theta_3$ and $\mathsf{hierarchy\_ok}(\theta_1; \theta_0)$ it is easy to deduce that $\mathsf{hierarchy\_ok}(\theta_3; \theta_0)$. By applying the induction hypothesis, we have that $\mathsf{gvalid}(\phi''(\gamma_s); \theta_3) = \theta_2$ holds. To complete the proof it suffices to show that $\mathsf{evalid}(\psi(\gamma_s); \theta_1) = \theta_3$. We proceed by performing a case analysis on the structure of ψ:

Case $\psi(\gamma) = \mathtt{Cap}\, \xi$ or $\psi(\gamma) = \delta\, \imath$: The proof is immediate, as $\psi(\gamma_s) = \psi(\gamma_L)$.

Case $\psi(\gamma) = \mathtt{Spawn}\, \xi_1\, \phi'''(\gamma)$: By inversion of $\mathsf{evalid}(\psi(\gamma_L); \theta_1) = \theta_3$ we obtain θ_s and θ'_s such that $\mathsf{merge}(\xi_1) \vdash \theta_1 = \theta_3 \oplus \theta_s$, $\mathsf{gvalid}(\phi'''(\gamma_L); \theta_s) = \theta'_s$, $\mathsf{live}(\theta'_s) = \emptyset$, and $\mathsf{mutex}(\{\theta_s, \theta_3\})$. From $\mathsf{merge}(\xi_1) \vdash \theta_1 = \theta_3 \oplus \theta_s$ and $\mathsf{hierarchy\_ok}(\theta_1; \theta_0)$ we can easily deduce that $\mathsf{hierarchy\_ok}(\theta_s; \theta_0)$. The application of the induction hypothesis on $\mathsf{gvalid}(\phi'''(\gamma_L); \theta_s) = \theta'_s$ yields $\mathsf{gvalid}(\phi'''(\gamma_s); \theta_s) = \theta'_s$. It follows that $\mathsf{evalid}(\psi(\gamma_s); \theta_1) = \theta_3$.

Case $\psi(\gamma) = \mathtt{Join}\, \phi_1(\gamma)\, \phi_2(\gamma)$: By inversion of $\mathsf{evalid}(\psi(\gamma_L); \theta_1) = \theta_3$ we have that $\mathsf{gvalid}(\phi_1(\gamma_L); \theta_1) = \theta_3$ and $\mathsf{gvalid}(\phi_2(\gamma_L); \theta_1) = \theta_3$. Applying the induction hypothesis on these two, we have $\mathsf{gvalid}(\phi_1(\gamma_s); \theta_1) = \theta_3$ and $\mathsf{gvalid}(\phi_2(\gamma_s); \theta_1) = \theta_3$. It follows that $\mathsf{evalid}(\psi(\gamma_s); \theta_1) = \theta_3$.

Lemma B.10 (Preservation of gvalid for a smaller θ) If $\gamma_L = \{\mathtt{Live}\, r \mid r \in \mathsf{dom}(\phi(\emptyset))\}$, $\gamma_s = \mathsf{summary}(\phi(\gamma_L))$, $\mathsf{gvalid}(\gamma_s; \theta) = \theta$, $\mathsf{hierarchy\_ok}(\theta'; \theta)$, $\mathsf{ok}(\theta')$, and $\mathsf{dom}(\gamma_s) \subseteq \mathsf{live}(\theta')$, then $\mathsf{gvalid}(\gamma_s; \theta') = \theta'$.

Proof. Immediate by the definition of function $summary$ and the validity of γ_s, which only require all regions in γ_s are live in θ' and that the ancestors of each region in θ' are identical to the ancestors of this region in θ. These requirements are satisfied by the assumptions that $\mathsf{dom}(\gamma_s) \subseteq \mathsf{live}(\theta')$ and $\mathsf{hierarchy\_ok}(\theta'; \theta)$.

Lemma B.11 (Well-typed expressions have well-formed contexts) If an expression e is well-typed in the typing context $R; M; \Delta; \Gamma$ then $\vdash R; M; \Delta; \Gamma$ holds.

Proof. Straightforward proof by induction on the expression typing derivation.

Lemma B.12 (Value substitution preserves typing) If $R; M; \Delta; \Gamma, x : \tau_1 \vdash e : \tau_2 \,\&\, \gamma_1$ and $R; M; \emptyset; \emptyset \vdash v : \tau_1 \,\&\, \emptyset$, then $R; M; \Delta; \Gamma \vdash e[v/x] : \tau_2 \,\&\, \gamma_1$

Proof. Straightforward induction on the expression typing derivation.

Lemma B.13 (Well-typed expressions have well-formed types) $R; M; \Delta; \Gamma \vdash e : \tau \,\&\, \gamma \Rightarrow R; \Delta \vdash \tau$

Proof. Straightforward induction on the typing rules.

Lemma B.14 (Well-typed expressions have well-formed effects) $R; M; \Delta; \Gamma \vdash e : \tau \,\&\, \gamma \Rightarrow R; \Delta \vdash \gamma$

Proof. Straightforward induction on the typing rules.

Lemma B.15 (Region substitution preserves typing) If $R, \imath; M; \Delta, \rho; \Gamma \vdash e : \tau \,\&\, \gamma$, then $R, \imath; M; \Delta; \Gamma[\imath/\rho] \vdash e[\imath/\rho] : \tau[\imath/\rho] \,\&\, \gamma[\imath/\rho]$.

Proof. Proof by induction on the typing derivation of e.

Lemma B.16 (Region context expansion preserves expression typing) $R; M; \Delta; \Gamma \vdash e : \tau \,\&\, \gamma$ and $\imath \notin R \cup \{\bot\}$, then $R, \imath; M; \Delta; \Gamma \vdash e : \tau \,\&\, \gamma$.

Proof. Proof by induction on the typing derivation of e.

Lemma B.17 (Region context expansion preserves evaluation context typing) $R; M; \Delta; \Gamma \vdash E : \tau \longrightarrow \tau' \,\&\, \gamma$ and $\imath \notin R \cup \{\bot\}$, then $R, \imath; M; \Delta; \Gamma \vdash E : \tau \longrightarrow \tau' \,\&\, \gamma$.

Proof. Proof by induction on the derivation of E. In the case of rule *E1*, where $E = E'[F]$, Lemma B.18 is used.

Lemma B.18 (Region context expansion preserves frame typing) $R; M; \Delta; \Gamma \vdash F : \tau \longrightarrow \tau' \,\&\, \gamma$ and $\imath \notin R \cup \{\bot\}$, then $R, \imath; M; \Delta; \Gamma \vdash F : \tau \longrightarrow \tau' \,\&\, \gamma$.

Proof. Proof by induction on the derivation of F.

Lemma B.19 (Memory context expansion preserves expression typing) $R; M; \Delta; \Gamma \vdash e : \tau \,\&\, \gamma$, $\ell \notin \mathsf{dom}(M)$, $\imath \in R$ and $R; \Delta \vdash \tau$, then $R; M, \ell \mapsto (\tau, \imath); \Delta; \Gamma \vdash e : \tau \,\&\, \gamma$.

Proof. Proof by induction on the typing derivation of e.

Lemma B.20 (Memory context expansion preserves evaluation context typing) $R; M; \Delta; \Gamma \vdash E : \tau \longrightarrow \tau' \,\&\, \gamma$, $\ell \notin \mathsf{dom}(M)$, $\imath \in R$ and $R; \Delta \vdash \tau$, then $R; M, \ell \mapsto (\tau, \imath); \Delta; \Gamma \vdash E : \tau \longrightarrow \tau' \,\&\, \gamma$.

Proof. Proof by induction on the derivation of E. In the case of rule *E1*, where $E = E'[F]$, Lemma B.21 is used.

Lemma B.21 (Memory context expansion preserves frame typing) $R; M; \Delta; \Gamma \vdash F : \tau \longrightarrow \tau' \,\&\, \gamma$, $\ell \notin \mathsf{dom}(M)$, $\imath \in R$ and $R; \Delta \vdash \tau$, then $R; M, \ell \mapsto (\tau, \imath); \Delta; \Gamma \vdash F : \tau \longrightarrow \tau' \,\&\, \gamma$.

Proof. Proof by induction on the derivation of F.

Lemma B.22 (Evaluation Context Composition — E) If $R; M; \Delta; \Gamma \vdash e : \tau \,\&\, \gamma_a$ and $R; M; \Delta; \Gamma \vdash E : \tau \longrightarrow \tau' \,\&\, \gamma_b$, then $R; M; \Delta; \Gamma \vdash E[e] : \tau' \,\&\, \gamma_a :: \gamma_b$

Proof. Proof by induction on typing derivation of E. The base case is immediate as $\Box[e] = e$. The inductive case where $E = E'[F]$, the proof is immediate by inversion of the derivation of E (rule *E1*) and the application of Lemma B.24.

Lemma B.23 (Evaluation Context Decomposition — E) If $R; M; \Delta; \Gamma \vdash E[e] : \tau' \,\&\, \gamma$, then there exists a γ_a,γ_b and τ such that $R; M; \Delta; \Gamma \vdash e : \tau \,\&\, \gamma_a$ and $R; M; \Delta; \Gamma \vdash E : \tau \longrightarrow \tau' \,\&\, \gamma_b$ and $\gamma = \gamma_a :: \gamma_b$.

Proof. Proof by induction on the structure of E. The base case is immediate by using the well-formedness derivation for the type and typing context of e (i.e., Lemmata B.11 and B.13) and the application rule *E0*. The inductive case, where $E[e] = E'[F][e]$ is immediate by Lemma B.25 and rule *E1*.

Lemma B.24 (Evaluation Context Composition — F) If $R; M; \Delta; \Gamma \vdash e : \tau \,\&\, \gamma_a$ and $R; M; \Delta; \Gamma \vdash F : \tau \longrightarrow \tau' \,\&\, \gamma_b$, then $R; M; \Delta; \Gamma \vdash F[e] : \tau' \,\&\, \gamma_a :: \gamma_b$.

Proof. Proof by case analysis on typing derivation of F. The premises required to construct the typing derivation of $F[e]$ are given as premises of the typing derivation of F.

Lemma B.25 (Evaluation Context Decomposition — F) If $R; M; \Delta; \Gamma \vdash F[e] : \tau' \,\&\, \gamma$, then there exists a γ_a,γ_b and τ such that $R; M; \Delta; \Gamma \vdash e : \tau \,\&\, \gamma_a$ and $R; M; \Delta; \Gamma \vdash F : \tau \longrightarrow \tau' \,\&\, \gamma_1$ and $\gamma = \gamma_a :: \gamma_b$.

Proof. Proof by case analysis on the structure of F. The premises required for each case (i.e., rules *F1-F11*) are given by the premises of the typing derivation of $F[e]$.

Lemma B.26 (Translate implies valid) If $\mathsf{valid}(\mathsf{translate}(\gamma, \imath, \eta), \jmath); \theta)$ and $\imath \notin \mathsf{dom}(\theta)$, then $\mathsf{valid}(\gamma ; \theta, \imath \mapsto (\eta, \jmath))$.

Proof. Let θ' be equal to $\theta, \imath \mapsto (\eta, \jmath)$. We proceed by induction on the structure of γ.

Case $\gamma = \emptyset$: the assumption and the definition of function $translate$ imply that $\mathsf{valid}(\mathsf{translate}(\emptyset, \imath, \eta, \jmath); \theta)$ holds, where $\gamma_s = \mathsf{solve}(\neg\texttt{Live}\, \jmath; \eta) = \mathsf{valid}(\gamma_s; \theta)$. If γ_s is empty, then $\mathsf{valid}(\emptyset; \theta)$, $\mathsf{ok}(\eta)$ and $\mathsf{rg}(\eta) = 0$ hold. The former fact implies that $\mathsf{live}(\theta) = \emptyset$ and $\mathsf{ok}(\theta)$. Therefore, $\mathsf{live}(\theta') = \emptyset$ and $\mathsf{ok}(\theta')$ hold. Hence $\mathsf{gvalid}(\emptyset; \theta') = \theta'$ and $\mathsf{valid}(\emptyset; \theta')$ hold.

If γ_s is non-empty, then by the definition of solve, $\gamma_s = \neg\texttt{Live}\, \jmath$ and thus, $\mathsf{valid}(\gamma_s; \theta)$ holds. By inversion of the latter derivation we have that $\mathsf{ok}(\theta)$, $\mathsf{live}(\theta) = \emptyset$ and $\mathsf{cvalid}(\neg\texttt{Live}\, \jmath; \theta)$. Therefore, $\mathsf{live}(\theta') = \emptyset$ and $\mathsf{ok}(\theta')$ hold. Thus, $\mathsf{valid}(\gamma_s; \theta')$ holds.

Case $\gamma = \texttt{Join}\, \gamma_1\, \gamma_2 :: \gamma_3$: the assumption and the definition of translate imply that $\mathsf{valid}(\texttt{Join}\, \gamma_a\, \gamma_b; \theta)$, where $\gamma_a = \mathsf{translate}(\gamma_1 :: \gamma_3, \imath, \eta, \jmath)$ and $\gamma_a = \mathsf{translate}(\gamma_2 :: \gamma_3, \imath, \eta, \jmath)$. By inversion of the derivation of $\mathsf{valid}(\texttt{Join}\, \gamma_a\, \gamma_b; \theta)$ we have $\mathsf{gvalid}(\gamma_a; \theta) = \theta''$, $\mathsf{gvalid}(\gamma_b; \theta) = \theta''$ and $\mathsf{live}(\theta'') = \emptyset$. Therefore, $\mathsf{valid}(\gamma_a; \theta)$ and $\mathsf{valid}(\gamma_b; \theta)$ hold. Using the induction hypothesis we obtain that $\mathsf{valid}(\gamma_1 :: \gamma_3; \theta')$ and $\mathsf{valid}(\gamma_2 :: \gamma_3; \theta')$. Hence, $\mathsf{valid}(\texttt{Join}\ (\gamma_1 :: \gamma_3)\ (\gamma_2 :: \gamma_3); \theta')$.

Case $\gamma = \texttt{Cap}\, \xi :: \gamma_a$: let us assume that θ'' equals $\theta, \imath \mapsto (\eta{+}\eta_0, \jmath)$. The assumption and the definition of translate imply that $\mathsf{merge}(\xi) = \xi', \imath \mapsto \eta_0$, $\mathsf{valid}(\gamma_s :: \gamma_b; \theta)$, where $\gamma_s = \mathsf{solve}(\texttt{Live}, \jmath, \eta) :: \texttt{Cap}\, \xi'$ and $\gamma_b = \mathsf{translate}(\gamma_a, \imath, \eta + \eta_0, \jmath)$. By inversion of the initial validity assumption we have that $\mathsf{ok}(\theta)$, $\mathsf{xvalid}(\xi'; \theta)$, if $\jmath$ is not $\bot$ then $\mathsf{evalid}(\texttt{Live}\, \jmath; \theta) = \theta$, $\mathsf{gvalid}(\gamma_a; \theta) = \theta_0$ and $\mathsf{live}(\theta_0) = \emptyset$. The latter two facts yield $\mathsf{valid}(\gamma_a; \theta)$ and the application of the induction hypothesis implies $\mathsf{valid}(\gamma_a; \theta'')$. The definition of function $\mathsf{translate}$ implies that $\mathsf{ok}(\eta{+}\eta_0)$ and the definition of γ_s yields $\mathsf{ok}(\eta - (1,0,0))$. Hence, $\mathsf{ok}(\theta'')$ holds by $\mathsf{ok}(\eta - (1,0,0))$, $\mathsf{ok}(\theta'')$ and $\imath \notin \mathsf{dom}(\theta)$. $\mathsf{cvalid}(\texttt{Live}\, \jmath; \imath; \theta'')$ holds by using the facts $\mathsf{ok}(\eta - (1,0,0))$ (and $\mathsf{evalid}(\texttt{Live}\, \jmath; \theta) = \theta$ if $\jmath \neq \bot$). Therefore $\mathsf{valid}(\gamma; \theta'')$ holds by the above facts.

Case $\gamma = \delta\, \imath :: \gamma_a$: the assumption and the definition of translate imply that $\mathsf{valid}(\gamma_s :: \gamma_b; \theta)$, where $\gamma_s = \mathsf{solve}(\delta\, \jmath, \eta)$ and $\gamma_b = \mathsf{translate}(\gamma_a, \imath, \eta, \jmath)$. To complete the proof it suffices to show that $\mathsf{cvalid}(\delta\, \jmath; \imath; \theta')$ and $\mathsf{valid}(\gamma_a; \theta')$ hold. We proceed by a case analysis on γ_s. If γ_s is empty, then

$\mathsf{cvalid}(\delta; \imath; \theta')$ is immediate by using rule *C-B* and the definition of γ_s. Otherwise, γ_s is non-empty and by the definition of $solve$ we have that, $\gamma_s = \delta'\ \jmath$. By inversion of $\mathsf{valid}(\gamma_s :: \gamma_b; \theta)$ we have that $\mathsf{cvalid}(\delta'; \jmath; \theta)$. The application of rule *C-R* to the latter fact and the definition of γ_s implies $\mathsf{cvalid}(\delta; \imath; \theta')$. $\mathsf{valid}(\gamma_a; \theta')$ is immediate by applying the induction hypothesis to $\mathsf{valid}(\gamma_b; \theta)$, which can be derived from $\mathsf{valid}(\gamma; \theta)$.

Case $\gamma = \mathsf{Spawn}\,\xi\,\gamma_s :: \gamma_a$: the assumption and the definition of translate imply that $\mathsf{valid}(\gamma_r''' :: \mathsf{Spawn}\,\xi'\,(\gamma_s' :: \gamma_s'')\ ::\ (\gamma_r' :: \gamma_r''))$, where $\gamma_r''' = \mathsf{bot}(\mathtt{Live}, \jmath)$, $\mathsf{merge}(\xi) = \xi', \imath \mapsto \eta_s$, $\eta = \eta_r \oplus \eta_s$, $\jmath' = \text{if } \jmath \in \mathsf{dom}(\xi) \text{ then } \jmath \text{ else } \bot$, $\gamma_s' = \mathsf{p\text{-}constraint}(\jmath', \eta_r)$, $\gamma_s'' = \mathsf{translate}(\gamma_s, \imath, \eta_s, \jmath')$, $\gamma_r' = \mathsf{p\text{-}constraint}(\jmath, \eta_s)$ and $\gamma_r'' = \mathsf{translate}(\gamma_a, \imath, \eta_r, \jmath)$.

By inversion of $\mathsf{valid}(\gamma_r''' :: \mathsf{Spawn}\,\xi'\,(\gamma_s' :: \gamma_s'')\ ::\ (\gamma_r' :: \gamma_r''))$ we have that $\mathsf{cvalid}(\mathtt{Live}\,; \jmath; \theta)$ (if $\jmath \neq \bot$), $\forall \imath' \in \mathsf{dom}(\theta_s).\ \mathsf{cvalid}(\mathtt{Live}\,; \imath'; \theta)$, $\mathsf{merge}(\xi') \vdash \theta = \theta_r \oplus \theta_s$, $\mathsf{mutex}(\{\theta_r, \theta_s\})$, $\mathsf{valid}(\gamma_s' :: \gamma_s''; \theta_s)$ and $\mathsf{valid}(\gamma_r' :: \gamma_r''; \theta_r)$. By inversion of the latter two derivations, we have that $\mathsf{gvalid}(\gamma_s'; \theta_s) = \theta_s$, $\mathsf{gvalid}(\gamma_s''; \theta_s) = \theta_s'$, $\mathsf{live}(\theta_s') = \emptyset$, $\mathsf{gvalid}(\gamma_r'; \theta_r) = \theta_r$, $\mathsf{gvalid}(\gamma_r''; \theta_r) = \theta_r'$ and $\mathsf{live}(\theta_r') = \emptyset$.

It suffices to prove the following obligations:

- Case $\forall \imath' \in \mathsf{dom}(\theta_s, \imath \mapsto (\eta_s, \jmath'))$. $\mathsf{cvalid}(\mathtt{Live}\,; \imath'; \theta, \imath \mapsto (\eta, \jmath))$: immediate by the assumptions $\mathsf{cvalid}(\mathtt{Live}\,; \jmath; \theta)$ (if $\jmath \neq \bot$), $\eta = \eta_r \oplus \eta_s$ and $\forall \imath' \in \mathsf{dom}(\theta_s).\ \mathsf{cvalid}(\mathtt{Live}\,; \imath'; \theta)$.
- Case $\mathsf{merge}(\xi) \vdash \theta, \imath \mapsto (\eta, \jmath) = \theta_r, \imath \mapsto (\eta_r, \jmath) \oplus \theta_s, \imath \mapsto (\eta_s, \jmath')$: $\mathsf{merge}(\xi) = \xi', \imath \mapsto \eta_s$ and the definition of function $merge$ imply that $\mathsf{merge}(\xi') = \xi'$. Therefore, $\mathsf{merge}(\xi') \vdash \theta = \theta_r \oplus \theta_s$ can be rewritten as $\xi' \vdash \theta = \theta_r \oplus \theta_s$. The latter derivation, the definition of $\jmath'$ and $\eta = \eta_r \oplus \eta_s$ complete the proof for this case.
- Case $\mathsf{mutex}(\{(\theta_r, \imath \mapsto (\eta_r, \jmath)), (\theta_s, \imath \mapsto (\eta_s, \jmath'))\})$: if at least one of the threads has read or write capabilities on $\imath$, then the mutex invariant may be violated once $\imath$ is added to the hierarchy of each thread. Assuming that the new thread has read or write access to $\imath$, then $\mathsf{gvalid}(\gamma_r'; \theta_r) = \theta_r$ and $\eta = \eta_r \oplus \eta_s$ imply that the main thread has no write or no read/write access to $\imath$ and its ancestors ancestors respectively. Similarly, if the main thread has read or write access to $\imath$, then $\mathsf{gvalid}(\gamma_s'; \theta_s) = \theta_s$ and $\eta = \eta_r \oplus \eta_s$ imply that the new thread has no write or no read/write access to $\imath$ and its ancestors respectively. We also have from the assumptions that $\mathsf{mutex}(\{\theta_r, \theta_s\})$. Therefore, $\mathsf{mutex}(\{(\theta_r, \imath \mapsto (\eta_r, \jmath)), (\theta_s, \imath \mapsto (\eta_s, \jmath'))\})$ holds.
- Case $\mathsf{valid}(\gamma_a; \theta_r, \imath \mapsto (\eta_r, \jmath))$: we have shown that $\mathsf{valid}(\gamma_r' :: \gamma_r''; \theta_r)$ holds. If γ_r' is empty, then the proof is immediate by the application of the induction hypothesis. If γ_r' is non-empty, then the definition of $p - constraint$ implies that $\gamma_r' = \delta\ \jmath$. By inversion of the derivation of valid (using rule *V-K*) $\mathsf{valid}(\gamma_r''; \theta_r)$ holds. The proof is completed by the application of the induction hypothesis to the latter fact.
- Case $\mathsf{valid}(\gamma_s; \theta_s, \imath \mapsto (\eta_s, \jmath'))$: similar to the previous case; here we use $\mathsf{valid}(\gamma_s' :: \gamma_s''; \theta_s)$.

Lemma B.27 (Progress) Let $S; T$ be a well-typed configuration with $R; M \vdash S; T$ then $S; T$ is not stuck ($\vdash S; T$).

Proof. It suffices to show that for any thread in T, a step can be performed or $block$ predicate holds for it. Let e be an arbitrary thread in T such that $T = T_1, \langle \theta; e \rangle$ for some T_1. By inversion of the typing derivation of $S; T$ we have that $R; M; \emptyset; \emptyset \vdash e : \mathtt{unit}\ \&\ \gamma$, $\mathsf{valid}(\gamma; \theta)$, $\mathsf{mutex}(\{\theta_0 \mid \langle \theta_0; e_0 \rangle \in T\})$, and $R; M \vdash S$.

If e is a *value* then $e = ()$ and $\gamma = \emptyset$ and $\mathsf{valid}(\emptyset; \theta)$, which implies $\mathsf{live}(\theta) = \emptyset$. Thus, rule *E-T* can be applied.

If e is not a value then according to Lemma B.32, there exists a redex u and an evaluation context E such that $e = E[u]$. The application of Lemma B.23 to the typing derivation of $E[u]$ yields $R; M; \emptyset; \emptyset \vdash u : \tau\ \&\ \gamma_a$, $R; M; \emptyset; \emptyset \vdash E : \tau \longrightarrow \mathtt{unit}\ \&\ \gamma'$, where $\gamma = \gamma_a :: \gamma'$. Then, we proceed by performing a case analysis on u:

Case $(\lambda x.\, e')\ v$: a step can be taken by rule *E-A*.

Case $(\Lambda\rho.\, f)\ [\imath]$: a step can be taken by rule *E-RP*.

Case $(\texttt{fix}\ x.\, f)\ v$: a step can be taken by rule *E-FX*.

Case $\texttt{if true then}\ e_1\ \texttt{else}\ e_2$: a step can be taken by rule *E-IT*.

Case $\texttt{if false then}\ e_1\ \texttt{else}\ e_2$: a step can be taken by rule *E-IF*.

Case $\texttt{newrgn}\ \rho, x\ @\ \texttt{rgn}_\jmath\ \texttt{in}\ e_2$: it suffices to prove $\jmath \in \mathsf{live}(\theta) \cup \{\bot\}$. The typing derivation of u implies $\gamma_a = \texttt{Live}\ \jmath :: \mathsf{translate}(\gamma_2, \rho, (1,1,0), \jmath)$, where γ_2 is the effect of e_2. The application of Lemma B.29 to $\mathsf{valid}(\gamma_a :: \gamma'; \theta)$ implies that $\jmath \in \mathsf{live}(\theta) \cup \{\bot\}$. Rule *E-NR* can be applied to perform a step.

Case $\texttt{new}\ v\ @\ \texttt{rgn}_\imath$: identical to the previous case. Rule *E-NL* can be applied to perform a step.

Case $\texttt{deref}\ \texttt{loc}_\ell$: it suffices to prove $\imath \in \mathsf{rwlocked}(\theta)$, $\ell \mapsto v \in S(\imath)$ and $\imath \notin \mathsf{wlocked}(T)$. $\ell \mapsto v \in S(\imath)$ is immediate by $R; M \vdash S$ and the fact that $\ell \mapsto (\tau, \imath)$ belongs in M by the typing derivation of u. The typing derivation of u also implies $\gamma_a = \texttt{R}\,\imath$. Lemma B.30 and $\mathsf{valid}(\gamma_a :: \gamma'; \theta)$ imply that $\imath \in \mathsf{rwlocked}(\theta)$. We also have the assumption that $\mathsf{mutex}(\{\theta_0 \mid \langle \theta_0; e_0\rangle \in T\})$, which implies $\imath \notin \mathsf{wlocked}(T) = \emptyset$. Rule *E-D* can be applied to perform a step.

Case $\texttt{loc}_\imath := v$: it suffices to prove $\ell \mapsto v' \in S(\imath), \imath \in \mathsf{wlocked}(\theta)$ and $\imath \notin \mathsf{rwlocked}(T)$. $\ell \mapsto v' \in S(\imath)$ is immediate by $R; M \vdash S$ and the fact that $\ell \mapsto (\tau, \imath)$ belongs in M by the typing derivation of u. The typing derivation of u also implies $\gamma_a = \texttt{W}\,\imath$. Lemma B.31 and $\mathsf{valid}(\gamma_a :: \gamma'; \theta)$ imply that $\imath \in \mathsf{wlocked}(\theta)$. We also have the assumption that $\mathsf{mutex}(\{\theta_0 \mid \langle \theta_0; e_0\rangle \in T\})$, which implies $\imath \notin \mathsf{rwlocked}(T)$. Rule *E-AS* can be applied to perform a step.

Case $\texttt{cap}_{\eta'}\ \texttt{rgn}_\imath$: given that $\theta = \theta_1, \imath \mapsto (\eta', r')$ it suffices to prove $\theta' = \theta_1, \imath \mapsto (\eta + \eta', r'), \imath \in \mathsf{live}(\theta)$ and $\mathsf{mutex}(\{\theta'\} \cup \{\theta'' \mid \langle \theta''; e'\rangle \in T\})$ or $\mathsf{blocked}(T; \theta; E[\texttt{cap}_{\eta'}\ \texttt{rgn}_\imath])$ holds. The typing derivation of u implies $\gamma_a = \texttt{Cap}\ \{\imath \mapsto \eta\}$. Lemma B.29 and $\mathsf{valid}(\gamma_a :: \gamma'; \theta)$ imply that $\imath \in \mathsf{live}(\theta)$. If $\mathsf{mutex}(\{\theta'\} \cup \{\theta'' \mid \langle \theta''; e'\rangle \in T_1\})$ holds, then rule *E-CP* can be used to perform a single step. Otherwise, $\mathsf{blocked}(T; \theta; E[\texttt{cap}_{\eta'}\ \texttt{rgn}_\imath])$ holds using the assumption $\mathsf{mutex}(\{\theta_0 \mid \langle \theta_0; e_0\rangle \in T\})$.

Case $\texttt{spawn}_{\gamma_1}\ e_1$: it suffices to prove $\mathsf{merge}(\xi) \vdash \theta = \theta' \oplus \theta''$ and $\mathsf{dom}(\theta'') \subseteq \mathsf{live}(\theta)$ hold. The typing derivation of u implies $\gamma_a = \texttt{Spawn}\,\xi\,\gamma_1$. By inversion of $\mathsf{valid}(\gamma_a :: \gamma'; \theta)$ we have that $\mathsf{merge}(\xi) \vdash \theta = \theta' \oplus \theta''$ and $\forall \imath \in \mathsf{dom}(\theta'').\ \mathsf{cvalid}(\texttt{Live}\,; \imath; \theta)$ hold. The latter fact and Lemma B.28 imply that $\mathsf{dom}(\theta'') \subseteq \mathsf{live}(\theta)$. Rule *E-SP* can be applied to perform a single step.

Lemma B.28 (Cvalid implies live, wlocked and rwlocked) If $\mathsf{cvalid}(\delta, \imath, \theta)$ and $\delta \neq \neg\texttt{Live}$ then $\imath \in \mathsf{live}(\theta) \cup \{\bot\}$, $\delta = \texttt{W} \Rightarrow \imath \in \mathsf{wlocked}(\theta)$ and $\delta = \texttt{R} \Rightarrow \imath \in \mathsf{rwlocked}(\theta)$.

Proof. We perform case analysis on $cvalid$ derivation:

Case *C-T*: the proof is immediate.

Case *C-B* then the following hold $\theta = \theta', \imath \mapsto (\eta, \jmath)$ and $\mathsf{solve}(\delta, \jmath, \eta) = \emptyset$. We have assumed that $\delta \neq \neg\texttt{Live}$, therefore $\mathsf{solve}(\delta, \jmath, \eta) = \emptyset$ implies that $\jmath = \bot$ and $\mathsf{ok}(\eta - (1,0,0))$. Thus $\imath \in \mathsf{live}(\theta)$ holds. If $\delta = \texttt{W}$, then $solve$ also implies that $\mathsf{rw}(\eta) > 0$ hence $\imath \in \mathsf{wlocked}(\theta)$. If $\delta = \texttt{R}$, then $solve$ also implies that $\mathsf{rw}(\eta) \geq 0$, $\mathsf{rw}(\eta) \geq 0$ and $\mathsf{rw}(\eta) + \mathsf{rw}(\eta) > 0$ hence $\imath \in \mathsf{rwlocked}(\theta)$.

Case *C-R* then $\theta = \theta', \imath \mapsto (\eta, \jmath)$, $\mathsf{solve}(\delta, \jmath, \eta) = \delta'\ \jmath$ and $\mathsf{cvalid}(\delta'; \jmath; \theta')$ hold. Function $solve$ implies that $\delta' \neq \neg\texttt{Live}$ and $\mathsf{ok}(\eta - (1,0,0))$. In fact δ' is equal to $\texttt{Live}$ if δ equals $\texttt{Live}$. If δ is equal to $\texttt{R}$, then δ' can be either $\texttt{R}$ and $\mathsf{rd}(\eta) = 0$ or $\texttt{Live}$ when $\mathsf{rd}(\eta) > 0$. If δ is equal to $\texttt{W}$, then δ' can be either $\texttt{W}$ and $\mathsf{rd}(\eta) = \mathsf{wr}(\eta) = 0$ or $\texttt{Live}$ and $\mathsf{rw}(\eta) \geq 0$, $\mathsf{rw}(\eta) \geq 0$ and $\mathsf{rw}(\eta) + \mathsf{rw}(\eta) > 0$. The

application of the induction hypothesis to $\mathsf{cvalid}(\delta'; \jmath; \theta')$ yields $\jmath \in \mathsf{live}(\theta) \cup \{\bot\}$, $\delta' = \mathtt{W} \Rightarrow \jmath \in \mathsf{wlocked}(\theta)$ and $\delta' = \mathtt{R} \Rightarrow \jmath \in \mathsf{rwlocked}(\theta)$.

$\mathsf{ok}(\eta - (1,0,0))$ and $\jmath \in \mathsf{live}(\theta) \cup \{\bot\}$ imply that $\imath \in \mathsf{live}(\theta)$. If δ is `Live` the proof is completed. Otherwise, if $\delta' = \delta = \mathtt{W}$, then $\jmath \in \mathsf{wlocked}(\theta)$ implies $\imath \in \mathsf{wlocked}(\theta)$ and the proof is completed. Otherwise, if $\delta' = \delta = \mathtt{R}$, then $\jmath \in \mathsf{rwlocked}(\theta)$ implies $\imath \in \mathsf{rwlocked}(\theta)$ and the proof is completed. The last case is $\delta' = \mathtt{Live}$. This can only be the case when `R` or `W` are satisfied in $\mathsf{solve}(\delta, \jmath, \eta)$ and therefore the proof is immediate.

Lemma B.29 (Valid implies live) If $\mathsf{valid}(\delta\, \imath :: \gamma; \theta)$ and $\delta \neq \neg\mathtt{Live}$ then $\imath \in \mathsf{live}(\theta) \cup \{\bot\}$.

Proof. By inversion of $valid$ we obtain $\mathsf{cvalid}(\delta, \imath, \theta)$. The proof is immediate by using Lemma B.28.

Lemma B.30 (Valid implies rwlocked) If $\mathsf{valid}(\mathtt{R}\, r :: \gamma; \theta)$, then $r \in \mathsf{rwlocked}(\theta)$.

Proof. By inversion of $valid$ we obtain $\mathsf{cvalid}(\mathtt{R}, \imath, \theta)$. The proof is immediate by using Lemma B.28.

Lemma B.31 (Valid implies wlocked) If $\mathsf{valid}(\mathtt{W}\, r :: \gamma; \theta)$, then $r \in \mathsf{wlocked}(\theta)$.

Proof. By inversion of $valid$ we obtain $\mathsf{cvalid}(\mathtt{W}, \imath, \theta)$. The proof is immediate by using Lemma B.28.

Lemma B.32 (Well-typed expressions contain a well-typed redeces) If $R; M; \Delta; \Gamma \vdash e : \tau \,\&\, \gamma_1$ and e is *not* a value then $R; M; \Delta; \Gamma \vdash E'[u] : \tau \,\&\, \gamma_1$ such that $E'[u] = e$.

Proof. By induction on the shape of e. The key idea is to convert typing derivations of e, when e is not a redex, to typing derivations of the form $E'[e']$ and apply induction for e'.

Appendix C

Formal semantics and proof of soundness for Chapter 6

C.1 Summary of additional functions and relations

$\mathit{run}(\gamma, \iota, n)$	computes the future lockset for a $\texttt{lock}\,\iota$ operation, by traversing the continuation effect γ until n matching unlock operations for ι are found.
$\mathit{stack}(E)$	computes the continuation effect that corresponds to the evaluation context E, by concatenating the annotations found in stacked `pop` expressions.
$\mathit{available}(S, n)$	computes the set of locks that are available to thread n in the lock store S; the locks that are not currently owned by some thread other than n.
$\gamma_1 :: \gamma_2$	appends the two effects γ_1 and γ_2, so that all operations in γ_2 chronologically follow those in γ_1.
$\gamma \setminus r$	removes all occurrences of r from effect γ.
$r; n \vdash_{ok} \gamma$	checks that γ contains exactly n unmatched unlock operations for r, i.e. that the effect $\gamma_r :: \gamma$ is well balanced, where γ_r contains exactly n lock operations for r.
$M; \Delta \vdash \Gamma$	well formedness relation for typing environments.
$M; \Delta \vdash \gamma$	well formedness relation for effects.
$M; \Delta \vdash \tau$	well formedness relation for types.
$\mathit{summary}(\gamma)$	the effect summarization function.
$M; \Delta; \Gamma \vdash E : \tau \xrightarrow{\gamma_a;\gamma_b} \tau' \,\&\, (\gamma_1;\gamma_2)$	typing relation for evaluation contexts; τ is the expected type for the innermost hole in E, γ_a and γ_b are the hole's input and output effects, τ' is the type of the expression computed by E, and γ_1 and γ_2 are the input and output effects.

C.2 Language syntax

Language syntax

Value $v ::= f \mid () \mid \texttt{loc}_\iota \mid \texttt{true} \mid \texttt{false}$

Expression $e ::= x \mid f \mid (e\ e)^\xi \mid (e)\,[r] \mid e := e$
$\mid \texttt{deref}\ e \mid \texttt{let}\ \rho, x = \texttt{ref}\ e\ \texttt{in}\ e$
$\mid \texttt{share}\ e \mid \texttt{release}\ e \mid \texttt{lock}_\gamma\ e$
$\mid \texttt{unlock}\ e \mid () \mid \texttt{pop}_\gamma\ e \mid \texttt{loc}_\iota$
$\mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \mid \texttt{true} \mid \texttt{false}$

Function $f ::= \lambda x.\, e\ \texttt{as}\ \tau \xrightarrow{\gamma} \tau \mid \Lambda\rho.\, f \mid \texttt{fix}\ x : \tau.\, f$

Type $\tau ::= \langle\rangle \mid \tau \xrightarrow{\gamma} \tau \mid \forall\rho.\,\tau$
$\mid \mathtt{ref}(\tau, r) \mid \mathtt{bool}$

Location $r ::= \rho \mid \imath@n \mid \rho@n$

Calling mode $\xi ::= \mathsf{seq}(\gamma) \mid \mathsf{par}$

Capability $\kappa ::= n, n \mid \overline{n, n}$

Effect $\gamma ::= \emptyset \mid \gamma, r^\kappa \mid \gamma, \gamma\,?\,\gamma$

C.3 Operational semantics

Auxiliary syntax for operational semantics

Access Lists $\theta ::= \emptyset \mid \theta, \imath \mapsto n; n; \epsilon; \epsilon$

Store $S ::= \emptyset \mid S, \imath \mapsto v$

Threads $T ::= \emptyset \mid T, n : \theta; e$

Configuration $C ::= S; T$

Locations $\epsilon ::= \emptyset \mid \epsilon, \imath$

Evaluation context

Stack $E ::= \square \mid E[F]$

Frame $F ::= (\square\ e)^\xi \mid (v\ \square)^\xi \mid (\square)\,[r] \mid \mathtt{let}\ \rho, x = \mathtt{ref}\ \square\ \mathtt{in}\ e$
$\mid \mathtt{deref}\ \square \mid \square := e \mid v := \square \mid \mathtt{share}\ \square \mid \mathtt{release}\ \square$
$\mid \mathtt{lock}_\gamma\ \square \mid \mathtt{unlock}\ \square \mid \mathtt{pop}_\gamma\ \square$
$\mid \mathtt{if}\ \square\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2$

Redex $u ::= (\lambda x.\, e\ \mathtt{as}\ \tau\ v)^\xi \mid \mathtt{lock}_{\gamma_1}\ \mathtt{loc}_\imath \mid \mathtt{unlock}\ \mathtt{loc}_\imath \mid \mathtt{share}\ \mathtt{loc}_\imath$
$\mid \mathtt{release}\ \mathtt{loc}_\imath \mid \mathtt{deref}\ \mathtt{loc}_\imath \mid \mathtt{loc}_\imath := v \mid \mathtt{let}\ \rho, x = \mathtt{ref}\ v\ \mathtt{in}\ e_2$
$\mid \mathtt{pop}_\gamma\ v \mid \mathtt{if}\ v\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 \mid (\Lambda\rho.\, f)\,[r] \mid (\mathtt{fix}\ x : \tau.\, f\ \ v)^{\mathsf{seq}(\gamma_b)}$

Generic predicates and functions

$$\mathsf{set}(\gamma) \quad = \quad \forall\alpha,\, \gamma_1,\, \gamma_2.\, \gamma = (\gamma_1, \alpha) :: \gamma_2 \Rightarrow \alpha = r^\kappa \wedge r \notin \mathsf{dom}(\gamma_1) \cup \mathsf{dom}(\gamma_2)$$

$$\frac{}{\gamma :: \emptyset = \gamma} \qquad \frac{\gamma :: \gamma' = \gamma''}{\gamma :: \gamma', r^\kappa = \gamma'', r^\kappa} \qquad \frac{\gamma :: \gamma' = \gamma''}{\gamma :: \gamma', (\gamma_1\,?\,\gamma_2) = \gamma'', (\gamma_1\,?\,\gamma_2)}$$

$$\frac{\mathsf{dom}(\gamma) = \epsilon}{\mathsf{dom}(\gamma, r^\kappa) = \epsilon \cup \{r\}} \qquad \frac{\mathsf{dom}(\gamma) = \epsilon \qquad \mathsf{dom}(\gamma') = \epsilon' \qquad \mathsf{dom}(\gamma'') = \epsilon''}{\mathsf{dom}(\gamma, \gamma'\,?\,\gamma'') = \epsilon \cup \epsilon' \cup \epsilon''} \qquad \frac{}{\mathsf{dom}(\emptyset) = \emptyset}$$

$$\frac{}{\mathsf{max}(\emptyset) = \emptyset} \qquad \frac{\gamma_2 = \{r'^\kappa \in \mathsf{max}(\gamma_1) \mid r' \neq r\}}{\mathsf{max}(\gamma_1, r^\kappa) = \gamma_2, r^\kappa} \qquad \frac{\gamma_4 = \mathsf{max}(\gamma_1 :: \gamma_2) = \mathsf{max}(\gamma_1 :: \gamma_3)}{\mathsf{max}(\gamma_1, \gamma_2\,?\,\gamma_3) = \gamma_4}$$

$$\frac{\mathsf{set}(\gamma_1) \qquad \gamma_2 = \gamma_1 :: \gamma_3 \qquad \mathsf{dom}(\gamma_2) = \mathsf{dom}(\gamma_1)}{\mathsf{min}(\gamma_2) = \gamma_1} \qquad \frac{\gamma_1 = \gamma_2 :: \gamma_3}{\gamma_2 \lhd \gamma_1} \qquad \frac{\mathsf{max}(\gamma) = (\gamma_1, r^\kappa) :: \gamma_2}{\gamma(r) = \kappa}$$

Helper rules & predicates

$$
\begin{array}{lcl}
\mathsf{locked}(T) & = & \{\, \imath \mid (n : \theta; e) \in T \ \wedge\ \theta(\imath) \geq (1,1)\} \\
\mathsf{lk}(\kappa) & = & n_2 \qquad \text{if } \kappa = (n_1, n_2) \\
\theta\ +_\imath\ (n_1, n_2) & = & \theta[\imath \mapsto n_1 + n_3; n_2 + n_4; \epsilon_1; \epsilon_2] \qquad \text{if } (\imath \mapsto n_3; n_4; \epsilon_1; \epsilon_2) \in \theta \\
\theta(\imath) & = & (n_1, n_2) \qquad \text{if } (\imath \mapsto n_1; n_2; \epsilon_1; \epsilon_2) \in \theta
\end{array}
$$

$$
\frac{\kappa = (n_1, n_2) \qquad (\theta_a, \theta_b) = \mathsf{split}(\theta\,,\ \gamma) \qquad (\theta_c, \theta_d) = (\theta_a\ +_\imath\ (-n_1, -n_2)\,,\ \theta_b\ +_\imath\ (n_1, n_2))}{(\theta_c, \theta_d) = \mathsf{split}(\theta\,,\ \gamma, (\imath @ n_0)^\kappa)}\ \textit{(A1)}
\qquad
\frac{}{(\theta, \emptyset) = \mathsf{split}(\theta\,,\ \emptyset)}\ \textit{(A2)}
$$

$$
\frac{}{(\emptyset, n) = \mathsf{frame\_lockset}(\imath, n,\ \emptyset)}\ \textit{(W0)}
\qquad
\frac{}{(\emptyset, 0) = \mathsf{frame\_lockset}(\imath, 0,\ \gamma)}\ \textit{(W1)}
$$

$$
\frac{n_1 > 0 \qquad n_2 = \mathsf{lk}(\kappa) - \mathsf{lk}(\gamma(\imath @ n_0)) \qquad (\epsilon, n_3) = \mathsf{frame\_lockset}(\imath, n_1 + n_2,\ \gamma)}{(\epsilon, n_3) = \mathsf{frame\_lockset}(\imath, n_1,\ \gamma, (\imath @ n_0)^\kappa)}\ \textit{(W2)}
$$

$$
\frac{n_1 > 0 \qquad (\epsilon, n_2) = \mathsf{frame\_lockset}(\imath, n_1,\ \gamma_1 :: \gamma_2) \qquad (\epsilon', n_2) = \mathsf{frame\_lockset}(\imath, n_1,\ \gamma_1 :: \gamma_3)}{(\epsilon \cup \epsilon', n_2) = \mathsf{frame\_lockset}(\imath, n_1,\ \gamma_1, \gamma_2\,?\,\gamma_3)}\ \textit{(W3)}
$$

$$
\frac{n_1 > 0 \qquad (\epsilon, n_2) = \mathsf{frame\_lockset}(\imath, n_1,\ \gamma) \qquad \mathtt{j} \neq \imath \qquad \epsilon' = \{\mathtt{j} \mid \mathsf{lk}(\kappa) - \mathsf{lk}(\gamma(\mathtt{j} @ n_0)) < 0\}}{(\epsilon\ \cup\ \epsilon', n_2) = \mathsf{frame\_lockset}(\imath, n_1,\ \gamma, (\mathtt{j} @ n_0)^\kappa)}\ \textit{(W4)}
$$

$$
\frac{}{\{\imath\} = \mathsf{lockset}(\imath, n, \square)}\ \textit{(L1)}
$$

$$
\frac{n_1 > 0 \qquad \epsilon' = \mathsf{lockset}(\imath, n_2, E) \qquad (\epsilon, n_2) = \mathsf{frame\_lockset}(\imath, n_1,\ \gamma)}{\epsilon \cup \epsilon' = \mathsf{lockset}(\imath, n_1, E[\mathsf{pop}_\gamma\ \square])}\ \textit{(L2)}
$$

$$
\frac{E \neq \square}{\{\imath\} = \mathsf{lockset}(\imath, 0, E)}\ \textit{(L3)}
\qquad
\frac{F \neq \mathsf{pop}_\gamma\ \square \qquad \epsilon = \mathsf{lockset}(\imath, n_1, E) \qquad n_1 > 0}{\epsilon = \mathsf{lockset}(\imath, n_1, E[F])}\ \textit{(L4)}
$$

Reduction relation $C \rightsquigarrow C'$

$$\frac{v' \equiv \lambda x.\, e_1 \;\mathsf{as}\; \tau_1 \xrightarrow{\gamma_a} \tau_2 \qquad \text{fresh } n' \qquad (\theta_1, \theta_2) = \mathsf{split}(\theta\,,\ \mathsf{max}(\gamma_a))}{S; T, n:\theta; E[(v'\ v)^{\mathsf{par}}] \rightsquigarrow S; T, n:\theta_1; E[()], n':\theta_2; \square[(v'\ v)^{\mathsf{seq}(\mathsf{min}(\gamma_a))}]} \quad \textit{(E-SN)}$$

$$\frac{\forall \imath.\, \theta(\imath) = (0,0)}{S; T, n:\theta; ()\ \rightsquigarrow S; T} \quad \textit{(E-T)}$$

$$\frac{v' \equiv \lambda x.\, e_1 \;\mathsf{as}\; \tau'}{S; T, n:\theta; E[(v'\ v)^{\mathsf{seq}(\gamma_b)}] \rightsquigarrow S; T, n:\theta; E[\mathsf{pop}_{\gamma_b}\ e_1[v/x]]} \quad \textit{(E-A)}$$

$$\frac{\theta(\imath) \geq (1,1) \qquad \imath \notin \mathsf{locked}(T)}{S; T, n:\theta; E[\mathtt{loc}_\imath := v] \rightsquigarrow S[\imath \mapsto v]; T, n:\theta; E[()]} \quad \textit{(E-AS)}$$

$$\frac{\theta(\imath) \geq (1,1) \qquad \imath \notin \mathsf{locked}(T)}{S; T, n:\theta; E[\mathtt{deref}\ \mathtt{loc}_\imath] \rightsquigarrow S; T, n:\theta; E[S(\imath)]} \quad \textit{(E-D)}$$

$$\frac{\text{fresh } \imath@n_1 \qquad S' = S, \imath \mapsto v \qquad \theta' = \theta, \imath \mapsto 1; 1; \emptyset; \emptyset}{S; T, n:\theta; E[\mathtt{let}\ \rho, x = \mathtt{ref}\ v\ \mathtt{in}\ e_2] \rightsquigarrow S'; T, n:\theta'; E[e_2[\imath@n_1/\rho][\mathtt{loc}_\imath/x]]} \quad \textit{(E-NG)}$$

$$\frac{}{S; T, n:\theta; E[\mathtt{if\ true\ then}\ e_1\ \mathtt{else}\ e_2] \rightsquigarrow S; T, n:\theta; E[e_1]} \quad \textit{(E-IT)}$$

$$\frac{\theta(\imath) \geq (1,1) \qquad \theta' = \theta\ +_\imath\ (0,-1)}{S; T, n:\theta; E[\mathtt{unlock}\ \mathtt{loc}_\imath] \rightsquigarrow S; T, n:\theta'; E[()]} \quad \textit{(E-UL)}$$

$$\frac{}{S; T, n:\theta; E[\mathtt{if\ false\ then}\ e_1\ \mathtt{else}\ e_2] \rightsquigarrow S; T, n:\theta; E[e_2]} \quad \textit{(E-IF)}$$

$$\frac{\begin{array}{c}\epsilon = \mathsf{lockset}(\imath, 1, E[\mathsf{pop}_{\gamma_1}\ \square]) \qquad \theta = \theta'', \imath \mapsto n_1; 0; \epsilon_1; \epsilon_2 \\ \theta' = \theta'', \imath \mapsto n_1; 1; \mathsf{dom}(S); \epsilon \qquad n_1 \geq 1 \qquad \mathsf{locked}(T) \cap \epsilon = \emptyset\end{array}}{S; T, n:\theta; E[\mathtt{lock}_{\gamma_1}\ \mathtt{loc}_\imath] \rightsquigarrow S; T, n:\theta'; E[()]} \quad \textit{(E-LK0)}$$

$$\frac{\theta(\imath) \geq (1,1) \qquad \theta' = \theta\ +_\imath\ (0,1)}{S; T, n:\theta; E[\mathtt{lock}_{\gamma_1}\ \mathtt{loc}_\imath] \rightsquigarrow S; T, n:\theta'; E[()]} \quad \textit{(E-LK1)}$$

$$\frac{}{S; T, n:\theta; E[(\mathtt{fix}\ x:\tau.\, f\ \ v)^{\mathsf{seq}(\gamma_a)}] \rightsquigarrow S; T, n:\theta; E[(f[\mathtt{fix}\ x:\tau.\, f/x]\ \ v)^{\mathsf{seq}(\gamma_a)}]} \quad \textit{(E-FX)}$$

$$\frac{\theta(\imath) \geq (1,0) \qquad \theta' = \theta\ +_\imath\ (1,0)}{S; T, n:\theta; E[\mathtt{share}\ \mathtt{loc}_\imath] \rightsquigarrow S; T, n:\theta'; E[()]} \quad \textit{(E-SH)}$$

$$\frac{\begin{array}{c}\theta(\imath) \geq (1,0) \qquad \theta(\imath) = (n_1, n_2) \\ n_1 = 1 \Rightarrow n_2 = 0 \qquad \theta' = \theta\ +_\imath\ (-1,0)\end{array}}{S; T, n:\theta; E[\mathtt{release}\ \mathtt{loc}_\imath] \rightsquigarrow S; T, n:\theta'; E[()]} \quad \textit{(E-RL)}$$

$$\frac{\text{fresh } n_2}{S; T, n:\theta; E[(\Lambda\rho.\, f)[\imath@n_1]] \rightsquigarrow S; T, n:\theta; E[f[\imath@n_2/\rho]]} \quad \textit{(E-RP)}$$

$$\frac{}{S; T, n:\theta; E[\mathsf{pop}_\gamma\ v] \rightsquigarrow S; T, n:\theta; E[v]} \quad \textit{(E-PP)}$$

C.4 Static semantics

Static semantics syntax

Type variable list $\square ::= \emptyset \mid \Delta, \rho$
Memory List $M ::= \emptyset \mid M, \iota \mapsto \tau$
Variable list $\square ::= \emptyset \mid \Gamma, x : \tau$

Type equivalence

$$\frac{}{r \simeq r}\ \textit{(S0)} \qquad \frac{r' \simeq r}{r \simeq r'}\ \textit{(S1)} \qquad \frac{r \simeq r'}{r \simeq r'@n_2}\ \textit{(S2)} \qquad \frac{}{\emptyset \simeq \emptyset}\ \textit{(S3)}$$

$$\frac{r_1 \simeq r_2 \quad \gamma_1 \simeq \gamma_2}{\gamma_1, {r_1}^{\kappa} \simeq \gamma_2, {r_2}^{\kappa}}\ \textit{(S4)} \qquad \frac{\gamma_1 \simeq \gamma_4 \quad \gamma_2 \simeq \gamma_5 \quad \gamma_3 \simeq \gamma_6}{\gamma_1, \gamma_2 \,?\, \gamma_3 \simeq \gamma_4, \gamma_5 \,?\, \gamma_6}\ \textit{(S5)} \qquad \frac{}{\tau \simeq \tau}\ \textit{(S6)}$$

$$\frac{\begin{array}{c}\tau_3 \simeq \tau_4 \\ r_1 \simeq r_2\end{array}}{\mathtt{ref}(\tau_3, r_1) \simeq \mathtt{ref}(\tau_4, r_2)}\ \textit{(S7)} \qquad \frac{\begin{array}{c}\text{fresh } \rho_1@n \\ \tau_1[\rho_1@n/\rho] \simeq \tau_2[\rho_1@n/\rho']\end{array}}{\forall \rho.\, \tau_1 \simeq \forall \rho'.\, \tau_2}\ \textit{(S8)}$$

$$\frac{\begin{array}{cc}\tau_1 \simeq \tau_3 & \tau_2 \simeq \tau_4 \\ \gamma_1 \simeq \gamma_3 & \gamma_2 \simeq \gamma_4\end{array}}{\tau_1 \xrightarrow{\gamma_1} \tau_2 \simeq \tau_3 \xrightarrow{\gamma_3} \tau_4}\ \textit{(S9)}$$

Capability-related rules

$$\frac{\begin{array}{c}\mathsf{is\_pure}(\kappa) = \exists n_1.\, \exists n_2.\, \kappa = n_1, n_2 \\ \forall r^{\kappa} \in \gamma.\, \mathsf{is\_pure}(\kappa) \Rightarrow \forall r'^{\kappa'} \in \gamma.\, r' \neq r \Rightarrow \neg(r \simeq r') \\ \xi = \mathsf{par} \Rightarrow \forall r^{\kappa}.(r^{\kappa} \in \mathsf{min}(\gamma) \Rightarrow \kappa = (0,0)) \wedge (r^{\kappa} \in \mathsf{max}(\gamma) \wedge \neg \mathsf{is\_pure}(\kappa) \Rightarrow \mathsf{lk}(\kappa) = 0)\end{array}}{\xi \vdash \gamma}\ \textit{(OK)}$$

$$\frac{r' \simeq r \quad \gamma' = \gamma \setminus r'}{\gamma' = \gamma, r^{\kappa} \setminus r'}\ \textit{(M0)} \qquad \frac{\neg(r' \simeq r) \quad \gamma' = \gamma \setminus r'}{\gamma', r^{\kappa} = \gamma, r^{\kappa} \setminus r'}\ \textit{(M1)} \qquad \frac{}{\emptyset = \emptyset \setminus r}\ \textit{(M2)}$$

$$\frac{\begin{array}{ccc}\mathsf{is\_pure}(\kappa_3) \Rightarrow \kappa_2 = (0,0) \wedge \mathsf{is\_pure}(\kappa_1) & & \mathsf{is\_pure}(\kappa_1) \Leftrightarrow \mathsf{is\_pure}(\kappa_2) \\ \kappa_1 = (n_3 + n_5, n_4 + n_6) & \kappa_3 = (n_5, n_6) & \kappa_2 = (n_3, n_4)\end{array}}{\kappa_1 = \kappa_2 + \kappa_3}\ \textit{(K1)}$$

$$\frac{}{\gamma = \mathsf{subtract}(\gamma\,,\,\emptyset)}\ \textit{(ES1)} \qquad \frac{\gamma_2 = \mathsf{subtract}(\gamma, r^{\kappa_2}\,,\,\gamma_1) \quad \kappa = \kappa_2 + \kappa_1}{\gamma_2 = \mathsf{subtract}((\gamma, r^{\kappa})\,,\,(\gamma_1, r^{\kappa_1}))}\ \textit{(ES2)}$$

$$\frac{\kappa = \gamma(r) + \kappa_2 \qquad \gamma'' = \mathsf{add}(\gamma\,,\,\gamma')}{\gamma'', r^{\kappa} = \mathsf{add}(\gamma\,,\,(\gamma', r^{\kappa_2}))}\ \textit{(AD1)} \qquad \frac{}{\emptyset = \mathsf{add}(\gamma\,,\,\emptyset)}\ \textit{(AD2)}$$

$$\frac{\begin{array}{c}\gamma_2 = \mathsf{add}(\mathsf{subtract}(\mathsf{max}(\gamma)\,,\,\mathsf{min}(\gamma_1))\,,\,\gamma_1) \\ \gamma_2 = \mathsf{min}(\gamma_2) :: \gamma_3 \qquad\qquad \mathsf{seq}(\emptyset) \vdash \gamma\end{array}}{\mathsf{seq}(\gamma) \vdash \gamma :: \gamma_3 = \gamma \oplus \gamma_1}\ \textit{(D0)}$$

$$\frac{\begin{array}{c}\mathsf{par} \vdash \gamma_1 \\ \gamma_2 = \mathsf{add}(\mathsf{subtract}(\mathsf{max}(\gamma)\,,\,\mathsf{min}(\gamma_1))\,,\,\mathsf{max}(\gamma_1))\end{array}}{\mathsf{par} \vdash \gamma :: \gamma_2 = \gamma \oplus \gamma_1}\ \textit{(D1)}$$

$$\frac{\mathsf{locked}(\gamma, 1, r) \qquad \mathsf{lk}(\kappa) = 0}{\mathsf{locked}((\gamma, r^{\kappa}), 0, r)}\ \textit{(X1)} \qquad \frac{\mathsf{lk}(\kappa) > 0}{\mathsf{locked}((\gamma, r^{\kappa}), 1, r)}\ \textit{(X2)}$$

$$\frac{r \neq r' \qquad \mathsf{locked}(\gamma, n, r)}{\mathsf{locked}((\gamma, r'^{\kappa}), n, r)}\ \textit{(X3)}$$

$$\frac{\mathsf{locked}((\gamma :: \gamma_1), n, r) \vee \mathsf{locked}((\gamma :: \gamma_2), n, r)}{\mathsf{locked}((\gamma, \gamma_1 \,?\, \gamma_2), n, r)}\ \textit{(X4)}$$

$$\frac{\gamma_b = \mathsf{min}(\gamma_a) \qquad \gamma_c = \mathsf{max}(\gamma_a) \qquad \gamma_d = \{r^{\gamma_c(r)+(0,1)} \mid \mathsf{locked}(\gamma_a, 0, r)\}}{\gamma_b :: \gamma_d :: \gamma_c = \mathsf{summary}(\gamma_a)}\ \textit{(L0)}$$

Constraint Well-formedness

$$\frac{}{M;\Delta \vdash \emptyset} \qquad \frac{M;\Delta \vdash r \quad M;\Delta \vdash \gamma_1}{M;\Delta \vdash \gamma_1, r^{\kappa}} \qquad \frac{M;\Delta \vdash \gamma_1 \quad M;\Delta \vdash \gamma_2 \quad M;\Delta \vdash \gamma_3}{M;\Delta \vdash \gamma_1, \gamma_2 \,?\, \gamma_3}$$

Region Well-formedness

$$\frac{r \in \Delta \cup \mathsf{dom}(M)}{M;\Delta \vdash r} \qquad \frac{M;\Delta \vdash \iota}{M;\Delta \vdash \iota@n} \qquad \frac{M;\Delta \vdash \rho}{M;\Delta \vdash \rho@n}$$

Program Typing Context Well-formedness

$$\frac{\vdash M \quad M;\Delta \vdash \Gamma \quad M;\Delta \vdash \gamma_1 \quad M;\Delta \vdash \gamma_2 \quad \gamma_1 \lhd \gamma_2 \quad \mathsf{seq}(\emptyset) \vdash \gamma_2}{\vdash M;\Delta;\Gamma;\gamma_1;\gamma_2}$$

Type Well-formedness

$$\frac{}{M;\Delta \vdash \texttt{bool}} \qquad \frac{M;\Delta,\rho \vdash \tau}{M;\Delta \vdash \forall\rho.\,\tau} \qquad \frac{M;\Delta \vdash \tau \quad M;\Delta \vdash r}{M;\Delta \vdash \texttt{ref}(\tau, r)}$$

$$\frac{\mathsf{min}(\gamma_1)\text{ defined} \qquad M;\Delta \vdash \tau_1 \quad M;\Delta \vdash \gamma_1 \quad M;\Delta \vdash \tau_2}{M;\Delta \vdash \tau_1 \xrightarrow{\gamma_1} \tau_2} \qquad \frac{}{M;\Delta \vdash \langle\rangle}$$

Γ Well-formedness

$$\frac{}{M;\Delta \vdash \emptyset} \qquad \frac{M;\Delta \vdash \tau \quad x \notin \mathsf{dom}(\Gamma) \quad M;\Delta \vdash \Gamma}{M;\Delta \vdash \Gamma, x : \tau}$$

M Well-formedness

$$\frac{}{\vdash \emptyset} \qquad \frac{\vdash M \quad \iota \notin \mathsf{dom}(M) \quad M;\emptyset \vdash \tau}{\vdash M, \iota \mapsto \tau}$$

$$\frac{\vdash M;\Delta;\Gamma;\gamma;\gamma \quad (x:\tau') \in \Gamma \quad \tau \simeq \tau'}{M;\Delta;\Gamma \vdash x : \tau \,\&\, (\gamma;\gamma)} \textit{(T-V)} \qquad \frac{\vdash M;\Delta;\Gamma;\gamma;\gamma}{M;\Delta;\Gamma \vdash \mathtt{true} : \mathtt{bool} \,\&\, (\gamma;\gamma)} \textit{(T-TR)}$$

$$\frac{M;\Delta,\rho;\Gamma \vdash f : \tau \,\&\, (\gamma;\gamma)}{M;\Delta;\Gamma \vdash \Lambda\rho.\, f : \forall\rho.\, \tau \,\&\, (\gamma;\gamma)} \textit{(T-RF)}$$

$$\frac{\vdash M;\Delta;\Gamma;\gamma;\gamma \quad \tau' \equiv \tau_1 \xrightarrow{\gamma_b} \tau_2 \quad M;\Delta \vdash \tau' \quad \tau \simeq \tau' \quad \mathsf{seq}(\emptyset) \vdash \gamma_b \Rightarrow M;\Delta;\Gamma, x:\tau_1 \vdash e_1 : \tau_2 \,\&\, (\mathsf{min}(\gamma_b);\gamma_b)}{M;\Delta;\Gamma \vdash \lambda x.\, e_1 \;\mathtt{as}\; \tau' : \tau \,\&\, (\gamma;\gamma)} \textit{(T-F)}$$

$$\frac{M;\Delta \vdash r \quad M;\Delta \vdash \tau[r/\rho] \quad M;\Delta;\Gamma \vdash e_1 : \forall\rho.\, \tau \,\&\, (\gamma;\gamma')}{M;\Delta;\Gamma \vdash (e_1)\,[r] : \tau[r/\rho] \,\&\, (\gamma;\gamma')} \textit{(T-RP)}$$

$$\frac{\vdash M;\Delta;\Gamma;\gamma;\gamma}{M;\Delta;\Gamma \vdash () : \langle\rangle \,\&\, (\gamma;\gamma)} \textit{(T-U)}$$

$$\frac{\vdash M;\Delta;\Gamma;\gamma;\gamma \quad (\imath \mapsto \tau') \in M \quad \tau \simeq \mathtt{ref}(\tau', \imath)}{M;\Delta;\Gamma \vdash \mathtt{loc}_\imath : \tau \,\&\, (\gamma;\gamma)} \textit{(T-L)}$$

$$\frac{\vdash M;\Delta;\Gamma;\gamma;\gamma}{M;\Delta;\Gamma \vdash \mathtt{false} : \mathtt{bool} \,\&\, (\gamma;\gamma)} \textit{(T-FL)}$$

$$\frac{M;\Delta;\Gamma \vdash e : \tau' \,\&\, (\mathsf{min}(\gamma_b);\gamma_b) \quad \gamma_b \simeq \gamma_b' \quad \mathsf{seq}(\gamma) \vdash \gamma' = \gamma \oplus \gamma_b' \quad \tau' \simeq \tau \quad \vdash M;\Delta;\Gamma;\gamma;\gamma'}{M;\Delta;\Gamma \vdash \mathtt{pop}_\gamma\; e : \tau \,\&\, (\gamma;\gamma')} \textit{(T-PP)}$$

$$\frac{M;\Delta;\Gamma \vdash e_1 : \mathtt{ref}(\tau, r) \,\&\, (\gamma_1;\gamma') \quad M;\Delta;\Gamma \vdash e_2 : \tau \,\&\, (\gamma;\gamma_1) \quad \gamma(r) \geq (1,1)}{M;\Delta;\Gamma \vdash e_1 := e_2 : \langle\rangle \,\&\, (\gamma;\gamma')} \textit{(T-AS)}$$

$$\frac{M;\Delta;\Gamma \vdash e : \mathtt{ref}(\tau, r) \,\&\, (\gamma, r^{\kappa-(1,0)};\gamma') \quad \kappa \geq (2,0) \quad \gamma(r) = \kappa}{M;\Delta;\Gamma \vdash \mathtt{share}\; e : \langle\rangle \,\&\, (\gamma;\gamma')} \textit{(T-SH)}$$

$$\frac{M;\Delta;\Gamma \vdash e : \mathtt{ref}(\tau, r) \,\&\, (\gamma, r^{\kappa+(1,0)};\gamma') \quad \kappa = (n_1, n_2) \quad n_1 = 0 \Rightarrow n_2 = 0 \quad \gamma(r) = \kappa}{M;\Delta;\Gamma \vdash \mathtt{release}\; e : \langle\rangle \,\&\, (\gamma;\gamma')} \textit{(T-RL)}$$

$$\frac{M;\Delta;\Gamma \vdash e : \mathtt{ref}(\tau, r) \,\&\, (\gamma, r^{\kappa-(0,1)};\gamma') \quad \kappa \geq (1,1) \quad \gamma(r) = \kappa}{M;\Delta;\Gamma \vdash \mathtt{lock}_\gamma\; e : \langle\rangle \,\&\, (\gamma;\gamma')} \textit{(T-LK)}$$

$$\frac{M;\Delta;\Gamma \vdash e_1 : \tau_1 \xrightarrow{\gamma_a} \tau_2 \,\&\, (\gamma_3;\gamma') \quad \xi \vdash \gamma_2 = \gamma \oplus \gamma_a \quad M;\Delta;\Gamma \vdash e_2 : \tau_1 \,\&\, (\gamma_2;\gamma_3) \quad \xi = \mathsf{par} \Rightarrow \tau_2 = \langle\rangle}{M;\Delta;\Gamma \vdash (e_1\ e_2)^\xi : \tau_2 \,\&\, (\gamma;\gamma')} \textit{(T-A)}$$

$$\frac{M;\Delta;\Gamma \vdash e : \mathtt{ref}(\tau, r)\,\&\,(\gamma, r^{\kappa+(0,1)};\gamma') \qquad \kappa \geq (1,0) \qquad \gamma(r) = \kappa}{M;\Delta;\Gamma \vdash \mathtt{unlock}\ e : \langle\rangle\,\&\,(\gamma;\gamma')} \textit{(T-UL)}$$

$$\frac{\begin{array}{c} M;\Delta;\Gamma \vdash e_1 : \tau_1\,\&\,(\gamma_1 \setminus \rho;\gamma') \qquad \gamma_1(\rho) = (1,1) \\ M;\Delta \vdash \tau \qquad M;\Delta,\rho;\Gamma,x : \mathtt{ref}(\tau_1,\rho) \vdash e_2 : \tau\,\&\,(\gamma,\rho^{0,0};\gamma_1) \end{array}}{M;\Delta;\Gamma \vdash \mathtt{let}\ \rho, x = \mathtt{ref}\ e_1\ \mathtt{in}\ e_2 : \tau\,\&\,(\gamma;\gamma')} \textit{(T-NG)}$$

$$\frac{\begin{array}{c} \tau \equiv \tau_1 \xrightarrow{\gamma_b} \tau_2 \qquad \tau' \equiv \tau_1' \xrightarrow{\gamma_a'} \tau_2' \qquad \tau \simeq \tau' \qquad \gamma_a \simeq \gamma_a' \\ M;\Delta;\Gamma, x : \tau \vdash f : \tau'\,\&\,(\gamma;\gamma) \qquad \gamma_b = \mathsf{summary}(\gamma_a) \end{array}}{M;\Delta;\Gamma \vdash \mathtt{fix}\ x : \tau.\, f : \tau\,\&\,(\gamma;\gamma)} \textit{(T-FX)}$$

$$\frac{\begin{array}{c} M;\Delta;\Gamma \vdash e_1 : \mathtt{bool}\,\&\,(\gamma, \gamma_2\,?\,\gamma_3;\gamma') \qquad \mathsf{max}(\gamma :: \gamma_2) = \mathsf{max}(\gamma :: \gamma_3) \\ M;\Delta;\Gamma \vdash e_2 : \tau\,\&\,(\gamma;\gamma :: \gamma_2) \qquad M;\Delta;\Gamma \vdash e_3 : \tau\,\&\,(\gamma;\gamma :: \gamma_3) \end{array}}{M;\Delta;\Gamma \vdash \mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3 : \tau\,\&\,(\gamma;\gamma')} \textit{(T-IF)}$$

$$\frac{M;\Delta;\Gamma \vdash e_1 : \mathtt{ref}(\tau, r)\,\&\,(\gamma;\gamma') \qquad \gamma(r) \geq (1,1)}{M;\Delta;\Gamma \vdash \mathtt{deref}\ e_1 : \tau\,\&\,(\gamma;\gamma')} \textit{(T-D)}$$

C.5 Type safety

Evaluation context typing

$$\frac{\vdash M;\Delta;\Gamma;\gamma_1;\gamma_2 \qquad M;\Delta \vdash \tau}{M;\Delta;\Gamma \vdash \Box : \tau \xrightarrow{\gamma_1;\gamma_2} \tau\,\&\,(\gamma_1;\gamma_2)} \textit{(E0)} \qquad \frac{\begin{array}{c} M;\Delta;\Gamma \vdash E : \tau_2 \xrightarrow{\gamma_5;\gamma_6} \tau_3\,\&\,(\gamma_1;\gamma_2) \\ M;\Delta;\Gamma \vdash F : \tau_1 \xrightarrow{\gamma_3;\gamma_4} \tau_2\,\&\,(\gamma_5;\gamma_6) \end{array}}{M;\Delta;\Gamma \vdash E[F] : \tau_1 \xrightarrow{\gamma_3;\gamma_4} \tau_3\,\&\,(\gamma_1;\gamma_2)} \textit{(E1)}$$

$$\frac{\begin{array}{c} \gamma_3 \lhd \gamma_4 \qquad \gamma_2 = \gamma_1 \oplus \gamma_a \\ M;\Delta;\Gamma \vdash e_2 : \tau_1\,\&\,(\gamma_2;\gamma_3) \qquad M;\Delta \vdash \tau_1 \xrightarrow{\gamma_a} \tau_2 \\ \xi \vdash \gamma_a \qquad \xi = \mathsf{seq}(\gamma_1) \vee (\xi = \mathsf{par} \wedge \tau_2 = \langle\rangle) \end{array}}{M;\Delta;\Gamma \vdash (\Box\ e_2)^{\xi} : (\tau_1 \xrightarrow{\gamma_a} \tau_2) \xrightarrow{\gamma_3;\gamma_4} \tau_2\,\&\,(\gamma_1;\gamma_4)} \textit{(F1)}$$

$$\frac{\begin{array}{c} M;\Delta;\Gamma \vdash v_1 : \tau_1 \xrightarrow{\gamma_a} \tau_2\,\&\,(\gamma_3;\gamma_3) \\ \gamma_2 = \gamma_1 \oplus \gamma_a \qquad \gamma_2 \lhd \gamma_3 \qquad M;\Delta \vdash \gamma_3 \\ \xi \vdash \gamma_a \qquad \xi = \mathsf{seq}(\gamma_1) \vee (\xi = \mathsf{par} \wedge \tau_2 = \langle\rangle) \end{array}}{M;\Delta;\Gamma \vdash (v_1\ \Box)^{\xi} : \tau_1 \xrightarrow{\gamma_2;\gamma_3} \tau_2\,\&\,(\gamma_1;\gamma_3)} \textit{(F2)}$$

$$\frac{\begin{array}{c} \vdash M;\Delta;\Gamma;\gamma;\gamma' \qquad \vdash M;\Delta;\Gamma;\gamma_1;\gamma_2 \qquad M;\Delta \vdash \tau \\ \gamma' = \gamma \oplus \gamma_2 \qquad e = v \Rightarrow \gamma_1 = \mathsf{min}(\gamma_1) \end{array}}{M;\Delta;\Gamma \vdash \mathtt{pop}_{\gamma}\ \Box : \tau \xrightarrow{\gamma_1;\gamma_2} \tau\,\&\,(\gamma;\gamma')} \textit{(F3)}$$

$$\frac{\begin{array}{c} \gamma_3 = \gamma_2 \setminus \rho \qquad \gamma_3 \lhd \gamma' \qquad \gamma_1(\rho) = (1,1) \qquad M;\Delta \vdash \tau_1 \qquad M;\Delta \vdash \tau \\ \vdash M;\Delta;\Gamma;\gamma;\gamma' \qquad M;\Delta,\rho;\Gamma,x : \mathtt{ref}(\tau_1,\rho) \vdash e_2 : \tau\,\&\,(\gamma,\rho^{0,0};\gamma_1) \end{array}}{M;\Delta;\Gamma \vdash \mathtt{let}\ \rho, x = \mathtt{ref}\ \Box\ \mathtt{in}\ e_2 : \tau_1 \xrightarrow{\gamma_3;\gamma'} \tau\,\&\,(\gamma;\gamma')} \textit{(F4)}$$

$$\frac{\vdash M;\Delta;\Gamma;\gamma;\gamma' \qquad M;\Delta \vdash \mathtt{ref}(\tau,r) \qquad M;\Delta;\Gamma \vdash e_2 : \tau \,\&\, (\gamma;\gamma_1) \qquad \gamma(r) \geq (1,1)}{M;\Delta;\Gamma \vdash \Box := e_2 : \mathtt{ref}(\tau,r) \xrightarrow{\gamma_1;\gamma'} \langle\rangle \,\&\, (\gamma;\gamma')} \textit{(F5)}$$

$$\frac{\gamma \lhd \gamma' \qquad \gamma(r) \geq (1,1) \qquad M;\Delta;\Gamma \vdash \mathtt{loc}_\iota : \mathtt{ref}(\tau,r) \,\&\, (\gamma';\gamma')}{M;\Delta;\Gamma \vdash \mathtt{loc}_\iota := \Box : \tau \xrightarrow{\gamma;\gamma'} \langle\rangle \,\&\, (\gamma;\gamma')} \textit{(F6)}$$

$$\frac{\vdash M;\Delta;\Gamma;\gamma;\gamma' \qquad \gamma(r) \geq (1,1) \qquad M;\Delta \vdash \mathtt{ref}(\tau,r)}{M;\Delta;\Gamma \vdash \mathtt{deref}\ \Box : \mathtt{ref}(\tau,r) \xrightarrow{\gamma;\gamma'} \tau \,\&\, (\gamma;\gamma')} \textit{(F7)}$$

$$\frac{\vdash M;\Delta;\Gamma;\gamma_1;\gamma' \qquad M;\Delta \vdash \mathtt{ref}(\tau,r) \qquad \kappa \geq (2,0) \qquad \gamma(r) = \kappa \qquad \gamma_1 = \gamma, r^{\kappa-(1,0)}}{M;\Delta;\Gamma \vdash \mathtt{share}\ \Box : \mathtt{ref}(\tau,r) \xrightarrow{\gamma_1;\gamma'} \langle\rangle \,\&\, (\gamma;\gamma')} \textit{(F8)}$$

$$\frac{n_1 = 0 \Rightarrow n_2 = 0 \qquad \gamma_1 = \gamma, r^{\kappa+(1,0)} \qquad \kappa = (n_1,n_2) \qquad \vdash M;\Delta;\Gamma;\gamma_1;\gamma' \qquad M;\Delta \vdash \mathtt{ref}(\tau,r) \qquad \gamma(r) = \kappa}{M;\Delta;\Gamma \vdash \mathtt{release}\ \Box : \mathtt{ref}(\tau,r) \xrightarrow{\gamma_1;\gamma'} \langle\rangle \,\&\, (\gamma;\gamma')} \textit{(F9)}$$

$$\frac{\vdash M;\Delta;\Gamma;\gamma_1;\gamma' \qquad M;\Delta \vdash \mathtt{ref}(\tau,r) \qquad \kappa \geq (1,0) \qquad \gamma(r) = \kappa \qquad \gamma_1 = \gamma, r^{\kappa+(0,1)}}{M;\Delta;\Gamma \vdash \mathtt{unlock}\ \Box : \mathtt{ref}(\tau,r) \xrightarrow{\gamma_1;\gamma'} \langle\rangle \,\&\, (\gamma;\gamma')} \textit{(F10)}$$

$$\frac{\vdash M;\Delta;\Gamma;\gamma_1;\gamma' \qquad M;\Delta \vdash \mathtt{ref}(\tau,r) \qquad \kappa \geq (1,1) \qquad \gamma(r) = \kappa \qquad \gamma_1 = \gamma, r^{\kappa-(0,1)}}{M;\Delta;\Gamma \vdash \mathtt{lock}_\gamma\ \Box : \mathtt{ref}(\tau,r) \xrightarrow{\gamma_1;\gamma'} \langle\rangle \,\&\, (\gamma;\gamma')} \textit{(F11)}$$

$$\frac{\gamma_3 = \gamma, \gamma_1\,?\,\gamma_2 \qquad M;\Delta \vdash \gamma' \qquad \gamma_3 \lhd \gamma' \qquad \mathsf{max}(\gamma :: \gamma_1) = \mathsf{max}(\gamma :: \gamma_2) \qquad M;\Delta;\Gamma \vdash e_2 : \tau \,\&\, (\gamma;\gamma :: \gamma_1) \qquad M;\Delta;\Gamma \vdash e_3 : \tau \,\&\, (\gamma;\gamma :: \gamma_2)}{M;\Delta;\Gamma \vdash \mathtt{if}\ \Box\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3 : \mathtt{bool} \xrightarrow{\gamma_3;\gamma'} \tau \,\&\, (\gamma;\gamma')} \textit{(F12)}$$

$$\frac{M;\Delta;\Gamma \vdash e : \tau \,\&\, (\gamma_a;\gamma_b) \qquad M;\Delta;\Gamma \vdash E : \tau \xrightarrow{\gamma_a;\gamma_b} \langle\rangle \,\&\, (\gamma_1;\gamma_2) \qquad \forall r^\kappa \in \gamma_1.\, \kappa = (0,0) \qquad \mathsf{counts\_ok}(E[\mathtt{pop}_{\gamma_b}\,\Box], \theta) \qquad \mathsf{lockset\_ok}(E[\mathtt{pop}_{\gamma_b}\,\Box], \theta)}{M;\Delta;\Gamma \vdash_t \theta; E[e] : \langle\rangle \,\&\, (\gamma_1;\gamma_2)} \textit{(EA)}$$

$$\begin{aligned} \mathsf{pure}(\gamma) &= \{\iota \mid \iota \simeq r \wedge r^{n_1,n_2} \in \gamma \wedge n_1 + n_2 > 0\} \\ \mathsf{counts\_ok}(E,\theta) &= \mathsf{counts\_rok}(E,\theta,\emptyset,\emptyset) \end{aligned}$$

$$\frac{\theta', \iota \mapsto n_1; n_2; \epsilon_1; \epsilon_2 = \theta - \gamma \qquad \kappa = (n_3,n_4) \qquad \iota \simeq r \qquad (n_1,n_2) \geq (n_3,n_4) \qquad \mathsf{is\_pure}(\kappa) \Rightarrow n_1 = n_3 \wedge n_2 = n_4}{\theta', \iota \mapsto n_1 - n_3; n_2 - n_4; \epsilon_1; \epsilon_2 = \theta - \gamma, r^\kappa} \textit{(B0)} \qquad \frac{}{\theta = \theta - \emptyset} \textit{(B1)}$$

$$\frac{\mathsf{counts\_rok}(E,\theta',\gamma,\epsilon \cup \epsilon') \qquad \epsilon' = \mathsf{pure}(\gamma'') \qquad \theta' = \theta - \gamma'' \qquad \epsilon \cap \epsilon' = \emptyset \qquad \gamma'' = \mathsf{subtract}(\mathsf{max}(\gamma)\,,\, \mathsf{max}(\gamma_1))}{\mathsf{counts\_rok}(E[\mathtt{pop}_\gamma\,\Box], \theta, \gamma_1, \epsilon)} \textit{(C1)}$$

$$\frac{\forall \imath.\theta(\imath) = (0,0)}{\mathsf{counts\_rok}(\Box, \theta, \gamma, \epsilon)}\ \textit{(C0)} \qquad \frac{F \neq \mathtt{pop}_{\gamma'}\ \Box \qquad \mathsf{counts\_rok}(E, \theta, \gamma, \epsilon)}{\mathsf{counts\_rok}(E[F], \theta, \gamma, \epsilon)}\ \textit{(C2)}$$

$$\frac{\mathsf{lockset\_ok}(E, \theta) \qquad \mathsf{lockset}(\imath, n_2, E) \cap \epsilon_1 \subseteq \epsilon_2}{\mathsf{lockset\_ok}(E\ ,\ \theta, \imath \mapsto n_1; n_2; \epsilon_1; \epsilon_2)}\ \textit{(DL0)} \qquad \frac{}{\mathsf{lockset\_ok}(E, \emptyset)}\ \textit{(DL1)}$$

Program typing

$$\begin{array}{lcl}
\mathsf{mutex}(T) & \equiv & \forall T_1, n : \theta; E[e].T = T_1, n : \theta; E[e] \Rightarrow \forall \imath.\theta(\imath) \geq (1,1) \Rightarrow \imath \notin \mathsf{locked}(T_1) \\
\mathsf{deadlocked}(T) & \equiv & T \supseteq T_1, n_0 : \theta_0; E[\mathtt{lock}_{\gamma_0}\ \mathtt{loc}_{\imath_0}], \ldots n_k : \theta_k; E_k[\mathtt{lock}_{\gamma_k}\ \mathtt{loc}_{\imath_k}] \wedge k > 0 \Rightarrow \\
 & & \forall m_1 \in [0,k].m_2 = (m_1 + 1)\ \mathit{mod}(k+1) \wedge \theta_{m_1}(\imath_{m_2}) \geq (1,1) \\
\mathsf{blocked}(T, n) & \equiv & T = T_1, n : \theta; E[\mathtt{lock}_{\gamma_2}\ \mathtt{loc}_{\imath}] \wedge \theta(\imath) = (n_1, n_2)\ \wedge\ n_1 > 0\ \wedge\ n_2 = 0 \wedge \\
 & & \mathsf{locked}(T_1) \cap \mathsf{lockset}(\imath, 1, E[\mathtt{pop}_{\gamma_2}\ \Box]) \neq \emptyset
\end{array}$$

Store Typing

$$\frac{\mathsf{dom}(M) = \mathsf{dom}(S) \qquad \forall (\imath \mapsto \tau) \in M.M; \emptyset; \emptyset \vdash S(\imath) : \tau\ \&\ (\emptyset;\emptyset)}{M \vdash S}$$

Configuration Typing

$$\frac{M \vdash T \qquad M \vdash S \qquad \mathsf{mutex}(T)}{M \vdash S; T}$$

Thread Typing

$$\frac{}{M \vdash \emptyset} \qquad \frac{M; \emptyset; \emptyset \vdash_t \theta; e : \langle\rangle\ \&\ (\gamma;\gamma') \qquad M \vdash T \qquad n \notin \mathsf{dom}(T)}{M \vdash T, n : \theta; e}$$

Not Stuck

$$\frac{\forall T', n : \theta; e.T = T', n : \theta; e \Rightarrow (T' \subseteq T''\ \wedge\ S; T\ \rightsquigarrow S'; T'') \vee \mathsf{blocked}(T, n)}{\vdash S; T}$$

Multi-step evaluation

$$\frac{n > 0 \quad S;T \rightsquigarrow^{n-1} S_{n-1};T_{n-1} \quad S_{n-1};T_{n-1} \rightsquigarrow S_n;T_n}{S;T \rightsquigarrow^{n} S_n;T_n} \textit{ (E-M1)} \qquad \frac{}{S;T \rightsquigarrow^{0} S;T} \textit{ (E-M2)}$$

Main theorems

Safety

$$\emptyset \vdash 0:\emptyset;e \wedge 0:\emptyset;e \rightsquigarrow^{n} S';T' \Rightarrow \vdash S';T' \wedge \neg \mathsf{deadlocked}(T)$$

Preservation

$$M \vdash S;T \wedge \ S;T \rightsquigarrow S';T' \Rightarrow \exists M' \supseteq M.\ M' \vdash S';T'$$

Progress

$$M \vdash S;T \Rightarrow \vdash S;T$$

Deadlock Freedom

$$\emptyset;0:e \rightsquigarrow^{n} S_n;T_n \wedge \forall \imath \in [0,n].\exists M_\imath.\, M_\imath \vdash S_\imath;T_\imath \Rightarrow \neg \mathsf{deadlocked}(T_n).$$

C.6 Proof of soundness

Theorem C.1 (Type Safety) Let expression e be the initial program and let the initial typing context M_0 and the initial program configuration $S_0;T_0$ be defined as follows: $M_0 = \emptyset$, $S_0 = \emptyset$, and $T_0 = \{0 : \emptyset; e\}$. If $S_0;T_0$ is well-typed in M_0 and the operational semantics takes any number of steps $S_0;T_0 \rightsquigarrow^{n} S_n;T_n$, then the resulting configuration $S_n;T_n$ is not stuck and T_n has not reached a deadlocked state.

Proof. The application of Lemma C.1 to the assumption implies that $\forall \imath \in [0,n].\exists M_\imath.M_\imath \vdash S_\imath;T_\imath$. Therefore, $S_n;T_n$ is well-typed for some M_n. The application of Lemma C.2 to $\forall \imath \in [0,n].\exists M_\imath.M_\imath \vdash S_\imath;T_\imath$ and $\emptyset;0:\emptyset;e \rightsquigarrow^{n} S_n;T_n$ implies that $\neg\mathsf{deadlocked}(T_n)$. The application of Lemma C.18 to $M_n \vdash S_n;T_n$ implies $S_n;T_n$ is not stuck.

Lemma C.1 (Multi-step Program Preservation) Let $S_0;T_0$ be a *closed well-typed configuration* for some M_0 and assume that $S_0;T_0$ evaluates to $S_n;T_n$ in n steps. Then for all $\imath \in [0,n]$ $M_\imath \vdash S_\imath;T_\imath$ holds. Let $S_0;T_0$ be a *closed well-typed configuration* such that $M_0 \vdash S_0;T_0$ for some M_0.

Proof. Proof by induction on the number of steps n. When no steps are performed (i.e., $n = 0$) the proof is immediate from the assumption. When some steps are performed (i.e., $n > 0$), we have that $S_0;T_0 \rightsquigarrow^{n} S_n;T_n$ or $S_0;T_0 \rightsquigarrow^{n-1} S_{n-1};T_{n-1}$ and $S_{n-1};T_{n-1} \rightsquigarrow S_n;T_n$. By applying the induction hypothesis on the fact that $S_0;T_0$ is well-typed and that $n-1$ steps are performed we obtain that $\forall \imath \in [0,n-1].\exists M_\imath.M_\imath \vdash S_\imath;T_\imath$. Thus, $M_{n-1} \vdash S_{n-1};T_{n-1}$ holds. The application of Lemma C.3 to $M_{n-1} \vdash S_{n-1};T_{n-1}$ and $S_{n-1};T_{n-1} \rightsquigarrow S_n;T_n$. implies that $M_n \vdash S_n;T_n$. Therefore, $\forall \imath \in [0,n].\exists M_\imath.M_\imath \vdash S_\imath;T_\imath$.

Lemma C.2 (Deadlock Freedom) If the initial configuration takes n steps, where each step is well typed, then the resulting configuration has not reached a deadlocked state.

Proof. Let us assume that z threads have reached a deadlocked state and let $m \in [0, z-1]$, $k = (m+1)$ *mod* z and $o = (k+1)$ *mod* z. According to definition of *deadlocked state*, thread m acquires lock $\imath_k$ and waits for lock $\imath_m$, whereas thread k acquires lock $\imath_o$ and waits for lock $\imath_k$. Assume that m is the first of the z threads that acquires a lock so it acquires lock $\imath_k$, before thread k acquires lock $\imath_o$.

Let us assume that $S_y; T_y$ is the configuration once $\imath_o$ is acquired by thread k for the first time, ϵ_{1y} is the corresponding lockset of $\imath_o$ ($\epsilon_{1y} = \mathsf{lockset}(\imath_o, 1, E[\mathsf{pop}_{\gamma_y}\ \Box])$) and ϵ_{2y} is the set of all heap locations ($\epsilon_{2y} = \mathsf{dom}(S_y)$) at the time $\imath_o$ is acquired. Then, $\imath_k$ does not belong to ϵ_{1y}, otherwise thread k would have been blocked at the lock request of $\imath_o$ as $\imath_k$ is already owned by thread m.

Let us assume that when thread k attempts to acquire $\imath_k$, the configuration is of the form $S_x; T_x$. According to the assumption of this lemma that all configurations are well typed so $S_x; T_x$ is well-typed as well. By inversion of the typing derivation of $S_x; T_x$, we obtain the typing derivation of thread $n_k : \theta_k; E_k[\mathtt{lock}_{\gamma'_k}\ \mathtt{loc}_{\imath_k}]$: $\mathtt{lock}_{\gamma'_k}\ \mathtt{loc}_{\imath_k}$ is well-typed with input-output effect $(\gamma'_k; \gamma''_k)$, where $\kappa = \gamma'_k(\imath_k @ n')$, $\kappa \geq (1,1)$, $\gamma'' = \gamma'_k, (\imath_k @ n')^{\kappa - (1,0)}$, and $\mathsf{lockset\_ok}(E_k[\mathsf{pop}_{\gamma''_k}\ \Box], \theta_k)$ holds, where θ_k is the access list of thread k.

$\mathsf{lockset\_ok}(E_k[\mathsf{pop}_{\gamma''_k}\ \Box], \theta_k)$ implies $\mathsf{lockset}(\imath_o, n_2, E_k[\mathsf{pop}_{\gamma''_k}\ \Box]) \cap \epsilon_1 \subseteq \epsilon_2$, where $\theta_k = \theta'_k, \imath_o \mapsto n_1; n_2; \epsilon_1; \epsilon_2$ (notice that n_2 is positive, $\epsilon_2 = \epsilon_{1y}$ and $\epsilon_1 = \epsilon_{2y}$ — this is immediate by the operational steps from $S_y; T_y$ to $S_x; T_x$ and rule *E-LK0*).

We have assumed that m is the first thread to lock $\imath_k$ at some step before $S_y; T_y$, thus $\imath_k \in \mathsf{dom}(S_y)$ (the store can only grow — this is immediate by observing the operational semantics rules). By the definition of $lockset$ function and the definition of γ''_k we have that $\imath_k \in \mathsf{lockset}(\imath_o, n_2, E_k[\mathsf{pop}_{\gamma''_k}\ \Box])$. Therefore, $\imath_k \in \mathsf{lockset}(\imath_o, n_2, E_k[\mathsf{pop}_{\gamma''_k}\ \Box]) \cap \mathsf{dom}(S_y) \subseteq \epsilon_{1y}$, which is a contradiction.

Lemma C.3 (Preservation) Let $S; T$ be a well-typed configuration with $M \vdash S; T$. If the operational semantics takes a step $S; T \rightsquigarrow S'; T'$, then there exist an $M' \supseteq M$ such that the resulting configuration is well-typed with $M' \vdash S'; T'$.

Proof. By case analysis on the thread evaluation relation:

Case *E-T*: Rule *E-T* implies that $\theta; E[e] = \theta; \Box[()]$, $S' = S$ and $T' = T$, $\forall \imath.\, \theta(\imath) = (0,0)$. By inversion of the configuration typing assumption we have that:

- $M \vdash T, n : \theta; \Box[()]$: by inversion of this derivation we have $M \vdash T$.
- $M \vdash S$
- $\mathsf{mutex}(T, n : \theta; \Box[()])$: implies that $\mathsf{mutex}(T)$.

Given the above facts, $M \vdash S; T$ holds.

Case *E-A*: Rule *E-A* implies that $S' = S, e = (\lambda x.\, e_1\ \mathtt{as}\ \tau_1 \xrightarrow{\gamma_c} \tau_2\ \ v)^{\mathsf{seq}(\gamma_a)}$ and $T' = T, n : \theta; E[\mathsf{pop}_{\gamma_a}\ e_1[v/x]]$.

By inversion of the configuration typing assumption we have that:

- $M \vdash S$
- $\mathsf{mutex}(T, n : \theta; E[e])$: no new locks are acquired.
  Thus, $\mathsf{mutex}(T, n : \theta; E[\mathsf{pop}_{\gamma_a}\ e_1[v/x]])$ holds.
- $M \vdash T, n : \theta; E[e]$: by inversion of this derivation we have that:
  - $M \vdash T$
  - $n \notin \mathsf{dom}(T)$
  - $M; \emptyset; \emptyset \vdash_t \theta; E[e] : \langle\rangle\ \&\ (\gamma;\gamma')$: by inversion we have that $\mathsf{counts\_ok}(E[\mathsf{pop}_{\gamma_b}\ \Box], \theta)$, $\mathsf{lockset\_ok}(E[\mathsf{pop}_{\gamma_b}\ \Box], \theta), \forall r^\kappa \in \gamma.\, \kappa = (0,0)$, and $M; \emptyset; \emptyset \vdash E : \tau'_2 \xrightarrow{\gamma_a;\gamma_b} \langle\rangle\ \&\ (\gamma;\gamma')$, $M; \emptyset; \emptyset \vdash e : \tau'_2\ \&\ (\gamma_a;\gamma_b)$. By inversion of the latter derivation we have that $M; \emptyset; \emptyset \vdash$

$v : \tau_1' \,\&\, (\gamma_b;\gamma_b)$, $\mathsf{seq}(\gamma_a) \vdash \gamma_b = \gamma_a \oplus \gamma_c'$ and $M;\emptyset;\emptyset \vdash \lambda x.\, e_1$ as $\tau_1 \xrightarrow{\gamma_c} \tau_2 : \tau_1' \xrightarrow{\gamma_c'} \tau_2' \,\&\, (\gamma_b;\gamma_b)$, where $\tau_1' \xrightarrow{\gamma_c'} \tau_2' \simeq \tau_1 \xrightarrow{\gamma_c} \tau_2$.
By inversion of the function typing derivation we obtain that $\mathsf{seq}(\emptyset) \vdash \gamma_c \Rightarrow M;\emptyset;\emptyset, x : \tau_1 \vdash e_1 : \tau_2 \,\&\, (\mathsf{min}(\gamma_c);\gamma_c)$. $\mathsf{seq}(\emptyset) \vdash \gamma_c'$ (by inversion of $\mathsf{seq}(\gamma_a) \vdash \gamma_b = \gamma_a \oplus \gamma_c'$) and $\gamma_c \simeq \gamma_c'$ imply that $\mathsf{seq}(\emptyset) \vdash \gamma_c$ hence $M;\emptyset;\emptyset, x : \tau_1 \vdash e_1 : \tau_2 \,\&\, (\mathsf{min}(\gamma_c);\gamma_c)$ holds. Lemma C.8 implies that $M;\emptyset;\emptyset \vdash v : \tau_1 \,\&\, (\gamma_b;\gamma_b)$. Lemma C.9 implies that $M;\emptyset;\emptyset \vdash e_1[v/x] : \tau_2 \,\&\, (\mathsf{min}(\gamma_c);\gamma_c)$ holds. The application of rule *T-PP* implies that $M;\emptyset;\emptyset \vdash \mathsf{pop}_{\gamma_a}\ e_1[v/x] : \tau_2' \,\&\, (\gamma_a;\gamma_b)$ holds ($\vdash M;\emptyset;\emptyset;\gamma_a;\gamma_b$ can be derived by the application of lemma C.4 to the typing derivation of e). Thus, $M;\emptyset;\emptyset \vdash E[\mathsf{pop}_{\gamma_a}\ e_1[v/x]] : \langle\rangle \,\&\, (\gamma;\gamma')$, by the application of rule *EA* to the typing derivation of $\mathsf{pop}_{\gamma_a}\ e_1[v/x]$, $\forall r^\kappa \in \gamma.\, \kappa = (0,0)$, $\mathsf{counts\_ok}(E[\mathsf{pop}_{\gamma_b}\ \square], \theta)$ and $\mathsf{lockset\_ok}(E[\mathsf{pop}_{\gamma_b}\ \square], \theta)$.

Case *E-FX*: Rule *E-FX* implies $S;T,n:\theta;E[(\mathtt{fix}\ x:\tau.\ f\ \ v)^{\mathsf{seq}(\gamma_a)}] \rightsquigarrow S;T,n:\theta;E[(f[\mathtt{fix}\ x:\tau.\ f/x]\ \ v)^{\mathsf{seq}(\gamma_a)}]$ holds. By inversion of the configuration typing assumption we have that:

- $M \vdash S$
- $\mathsf{mutex}(T, n:\theta; E[e])$: no new locks are acquired. Thus, $\mathsf{mutex}(T, n:\theta; E[e'])$ holds, where $e' = \mathsf{pop}_{\gamma_a}\ e_1[v/x]$.
- $M \vdash T, n:\theta; E[e]$, where e is equal to $(\mathtt{fix}\ x:\tau.\ f\ \ v)^{\mathsf{seq}(\gamma_a)}$: by inversion of this derivation we have that:
  - $M \vdash T$
  - $n \notin \mathsf{dom}(T)$
  - $M;\emptyset;\emptyset \vdash_t E[e] : \langle\rangle \,\&\, (\gamma_0;\gamma)$: lemma C.15 implies that $M;\emptyset;\emptyset \vdash E : \tau_2' \xrightarrow{\gamma_a;\gamma_b} \langle\rangle \,\&\, (\gamma_0;\gamma)$ and $M;\emptyset;\emptyset \vdash e : \tau_2' \,\&\, (\gamma_a;\gamma_b)$. By inversion of the latter derivation we have that $M;\emptyset;\emptyset \vdash v : \tau_1 \,\&\, (\gamma_b;\gamma_b)$, and $M;\emptyset;\emptyset \vdash \mathtt{fix}\ x : \tau.f : \tau \,\&\, (\gamma_b;\gamma_b)$, where $\mathsf{seq}(\gamma_a) \vdash \gamma_b = \gamma_a \oplus \gamma_c$ and τ equals $\tau_1 \xrightarrow{\gamma_c} \tau_2$. By inversion of the typing derivation of $\mathtt{fix}\ x : \tau.f$ we obtain that $M;\emptyset;\emptyset, x : \tau \vdash f : \tau' \,\&\, (\gamma_b;\gamma_b)$, where $\tau' = \tau_1' \xrightarrow{\gamma_d'} \tau_2'$, $\gamma_d' \simeq \gamma_d$, $\tau \simeq \tau'$ and $\gamma_c = \mathsf{summary}(\gamma_d)$. Lemma C.9 implies that $M;\emptyset;\emptyset \vdash f[\mathtt{fix}\ x : \tau.f/x] : \tau' \,\&\, (\gamma_b;\gamma_b)$ holds.
    $\mathsf{seq}(\gamma_a) \vdash \gamma_a :: \gamma_b' = \gamma_a \oplus \gamma_d'$ is implied by $\gamma_d' \simeq \gamma_d$, $\mathsf{seq}(\gamma_a) \vdash \gamma_b = \gamma_a \oplus \gamma_c$, $\mathsf{max}(\gamma_c) = \mathsf{max}(\gamma_d)$ and $\mathsf{min}(\gamma_c) = \mathsf{min}(\gamma_d)$.
    Lemma C.6 implies that $M;\emptyset;\emptyset \vdash E : \tau_2' \xrightarrow{\gamma_a;\gamma_a} \langle\rangle \,\&\, (\gamma_0;\gamma')$ and $M;\emptyset \vdash \gamma$, where $\gamma = \gamma' :: \gamma_c$. Thus, $M;\emptyset \vdash \gamma' :: \gamma_b'$ holds (the domain of γ_b' is a subset of the domain of γ_b). The application of lemma C.6 to the latter fact, $M;\emptyset;\emptyset \vdash E : \tau_2' \xrightarrow{\gamma_a;\gamma_a} \langle\rangle \,\&\, (\gamma_0;\gamma')$ implies that $M;\emptyset;\emptyset \vdash E : \tau_2' \xrightarrow{\gamma_a;\gamma_a::\gamma_b'} \langle\rangle \,\&\, (\gamma_0;\gamma' :: \gamma_b')$.
    Lemma C.5 implies that $M;\emptyset;\emptyset \vdash f[\mathtt{fix}\ x : \tau.\, f/x] : \tau_1' \xrightarrow{\gamma_d'} \tau_2' \,\&\, (\gamma_a :: \gamma_b';\gamma_a :: \gamma_b')$ and $M;\emptyset;\emptyset \vdash v : \tau_1' \,\&\, (\gamma_a :: \gamma_b';\gamma_a :: \gamma_b')$. Therefore, $M;\emptyset;\emptyset \vdash (f[\mathtt{fix}\ x : \tau.\, f/x]\ \ v)^{\mathsf{seq}(\gamma_a)} : \tau_2' \,\&\, (\gamma_a;\gamma_a :: \gamma_b')$.
  - $\mathsf{lockset\_ok}(E[\mathsf{pop}_{\gamma_b}\ \square], \theta)$ and $\mathsf{counts\_ok}(E[\mathsf{pop}_{\gamma_b}\ \square], \theta)$: $\mathsf{lockset\_ok}(E[\mathsf{pop}_{\gamma_a::\gamma_b'}\ \square], \theta)$ and $\mathsf{counts\_ok}(E[\mathsf{pop}_{\gamma_a::\gamma_b'}\ \square], \theta)$ are immediate from $\mathsf{lockset\_ok}(E[\mathsf{pop}_{\gamma_b}\ \square], \theta)$, $\gamma_c = \mathsf{summary}(\gamma_d)$, $\mathsf{max}(\gamma_c) = \mathsf{max}(\gamma_d)$, $\mathsf{min}(\gamma_c) = \mathsf{min}(\gamma_d)$, $\mathsf{counts\_ok}(E[\mathsf{pop}_{\gamma_b}\ \square], \theta)$ and the fact that each locked location in γ_c is also locked in γ_d and vice versa.

Case *E-LK0*, *E-LK1*, *E-UL*, *E-SH* and *E-RL*: these rules generate side-effects as they modify the reference/lock count of location ι. We provide a single proof for all cases. Hence, we are assuming here that u (i.e. in $E[u]$) has one of the following forms: $\mathtt{lock}_{\gamma_1}\ \mathtt{loc}_\iota$, $\mathtt{unlock}\ \mathtt{loc}_\iota$ $\mathtt{share}\ \mathtt{loc}_\iota$ or $\mathtt{release}\ \mathtt{loc}_\iota$. Rules *E-LK0*, *E-LK1*, *E-UL*, *E-SH* and *E-RL* imply that $S' = S$, $T' = T, n:\theta'; E[()]$, where $()$ replaces u in context E and θ differs with respect to θ' only in the

one of the counts of $\imath$ (i.e., $\theta' = \theta[\imath \mapsto \theta(\imath) + (n_1, n_2)]$ and $\gamma_a(r) - \kappa = (n_1, n_2)$ — γ_a is the input effect of $E[u]$ and κ is the count of the last element of the output effect of u).

By inversion of the configuration typing assumption we have that:

- $\mathsf{mutex}(T, n:\theta; E[u])$: In the case of *E-UL*, *E-SH*, *E-LK1* and *E-RL* no new locks are acquired. Thus, $\mathsf{mutex}(T, n:\theta'; E[()])$ holds. In the case of rule *E-LK0*, a new lock $\imath$ is acquired (i.e., when the lock count of $\imath$ is zero) the precondition of *E-LK0* suggests that no other thread holds $\imath$:

  $\mathsf{locked}(T) \cap \mathsf{lockset}(\imath, 1, E[\mathsf{pop}_{\gamma_a}\ \square]) = \emptyset$. Thus, $\mathsf{mutex}(T, n:\theta'; E[()])$ holds.

- $M; \emptyset; \emptyset \vdash_t \theta; E[u] : \langle\rangle \,\&\, (\gamma;\gamma')$: By inversion we have that $M; \emptyset; \emptyset \vdash E : \langle\rangle \xrightarrow{\gamma_a;\gamma_b} \langle\rangle \,\&\, (\gamma;\gamma')$ and $M; \emptyset; \emptyset \vdash u : \langle\rangle \,\&\, (\gamma_a;\gamma_b)$, where $\gamma_b = \gamma_a, (\imath@n')^\kappa$ for some n'. It can be trivially shown from the latter derivation that $M; \emptyset; \emptyset \vdash () : \langle\rangle \,\&\, (\gamma_a;\gamma_a)$. Lemma C.6 implies that $M; \emptyset; \emptyset \vdash E : \langle\rangle \xrightarrow{\gamma_a;\gamma_a} \langle\rangle \,\&\, (\gamma;\gamma'')$, where $\gamma' = \gamma'', (\imath@n')^\kappa$.

- $\mathsf{lockset\_ok}(E[\mathsf{pop}_{\gamma_b}\ \square], \theta)$ and $\mathsf{counts\_ok}(E[\mathsf{pop}_{\gamma_b}\ \square], \theta)$: By the definition of $lockset$ function it can be shown that:

  $\mathsf{lockset}(\mathtt{j}, n_b, E[\mathsf{pop}_{\gamma_a}\ \square]) \subseteq \mathsf{lockset}(\mathtt{j}, n_b, E[\mathsf{pop}_{\gamma_b}\ \square])$ for all $\mathtt{j} \neq \imath$ in the domain of θ' (n_b is the lock count of $\mathtt{j}$ in θ).

  The same applies for $\mathtt{j} = \imath$ in the case of rules *E-SH*, *E-RL* as the lock count of $\imath$ is not affected. In the case of rules *E-LK0*, *E-LK1*, *E-UL* we have $\mathsf{lockset}(\imath, n_b \pm 1, E[\mathsf{pop}_{\gamma_a}\ \square])$, but this is identical to $\mathsf{lockset}(\imath, n_b, E[\mathsf{pop}_{\gamma_b}\ \square])$ by the definition of $lockset$. Therefore $\mathsf{lockset\_ok}(E[\mathsf{pop}_{\gamma_a}\ \square], \theta')$ holds. The predicate $counts\_ok\ (E[\mathsf{pop}_{\gamma_b}\ \square], \theta)$ enforces the invariant that the static counts are identical to the dynamic counts (θ) of $\imath$. The lock count of θ is modified by ± 1 and γ_a differs with respect to γ_b by $(\imath@n')^\kappa$. We can use this fact to show that $\mathsf{counts\_ok}(E[\mathsf{pop}_{\gamma_a}\ \square], \theta')$.

Case *E-D*,*E-AS*, *E-RP* and *E-PP*: these rules are side-effect free and therefore we provide a single proof for all cases. Hence, we are assuming here that u (i.e. in $E[u]$) has one of the following forms: $(\Lambda\rho.\, f)\ [\imath@n_1]$, $\mathtt{deref}\ \mathtt{loc}_\imath$ $\mathtt{loc}_\imath := v$ or $\mathsf{pop}_\gamma\ v$. Rules *E-AS*,*E-D*,*E-RP*, and *E-PP* imply that $S' = S,\ \ T' = T, n:\theta; E[v]$, where v is the value that replaces u in context E. By inversion of the configuration typing assumption we have that:

- $M \vdash S$
- $\mathsf{mutex}(T, n:\theta; E[u])$: no new locks are acquired.

  Thus, $\mathsf{mutex}(T, n:\theta; E[v])$ holds.
- $M \vdash T, n:\theta; E[u]$: by inversion of this derivation we have that:
  - $M \vdash T$
  - $n \notin \mathsf{dom}(T)$
  - $M; \emptyset; \emptyset \vdash_t \theta; E[u] : \langle\rangle \,\&\, (\gamma;\gamma')$: In the case of rule *E-D* and *E-AS* the value v that substitutes u is $()$ and it can be trivially shown that it is well-typed. In the case rule *E-PP* applies, v is well-typed by inversion of the typing derivation of u. In the case of rule *E-RP*, v is obtained by substituting $\imath@n_2$ in the body of function f (i.e. the initial term is $(\Lambda\rho.\, f)\ [\imath@n_1]$). We can obtain the typing derivation of v by applying lemma C.11.Notice, that we are dealing with side-effect free rules in this case thus u has the same input and output effect.

Case *E-IT*: rule *E-IT* implies that $S; T, n:\theta; E[u] \rightsquigarrow S; T, n:\theta; E[e_1]$, where u is equal to $\mathtt{if\ true\ then}\ e_1\ \mathtt{else}\ e_2$. By inversion of the configuration typing assumption we have that:

- $M \vdash S$
- $\mathsf{mutex}(T, n:\theta; E[u])$: no new locks are acquired. Thus, $\mathsf{mutex}(T, n:\theta; E[v])$ holds.

- $M \vdash T, n:\theta; E[u]$, where u is equal to `if true then` e_1 `else` e_2: by inversion of this derivation we have that:
  - $M \vdash T$
  - $n \notin \mathsf{dom}(T)$
  - $M;\emptyset;\emptyset \vdash_t \theta; E[u] : \langle\rangle \,\&\, (\gamma;\gamma')$: by inversion of the typing derivation of $\theta; E[e]$ we obtain that $M;\emptyset;\emptyset \vdash E : \tau_2' \xrightarrow{\gamma_a;\gamma_b} \langle\rangle \,\&\, (\gamma;\gamma')$ and $M;\emptyset;\emptyset \vdash u : \tau_2' \,\&\, (\gamma_a;\gamma_b)$. By inversion of the latter derivation we have that $M;\emptyset;\emptyset \vdash e_1 : \tau_2' \,\&\, (\gamma_a;\gamma_a :: \gamma_{b_1})$, $M;\emptyset;\emptyset \vdash e_2 : \tau_2' \,\&\, (\gamma_a;\gamma_a :: \gamma_{b_2})$ and $\gamma_b = \gamma_a, \gamma_{b_1} \,?\, \gamma_{b_2}$. Lemma C.6 implies that $M;\emptyset;\emptyset \vdash E : \tau_2' \xrightarrow{\gamma_a;\gamma_a} \langle\rangle \,\&\, (\gamma;\gamma'')$ and $M;\emptyset \vdash \gamma'$, where $\gamma' = \gamma'', \gamma_{b_1} \,?\, \gamma_{b_2}$. Thus, $M;\emptyset \vdash \gamma'' :: \gamma_{b_1}$ holds. The application of lemma C.6 to the latter fact, $M;\emptyset;\emptyset \vdash E : \tau_2' \xrightarrow{\gamma_a;\gamma_a} \langle\rangle \,\&\, (\gamma;\gamma'')$ implies that $M;\emptyset;\emptyset \vdash E : \tau_2' \xrightarrow{\gamma_a;\gamma_a :: \gamma_{b_1}} \langle\rangle \,\&\, (\gamma;\gamma'' :: \gamma_{b_1})$.
  - $\mathsf{lockset\_ok}(E[\mathsf{pop}_{\gamma_b}\,\Box], \theta)$ and $\mathsf{counts\_ok}(E[\mathsf{pop}_{\gamma_b}\,\Box], \theta)$: $\mathsf{lockset\_ok}(E[\mathsf{pop}_{\gamma_a :: \gamma_{b_1}}\,\Box], \theta)$ is immediate from $\mathsf{lockset}(\mathtt{j}, n_b, E[\mathsf{pop}_{\gamma_a :: \gamma_{b_1}}\,\Box])$ is a subset of $\mathsf{lockset}(\mathtt{j}, n_b, E[\mathsf{pop}_{\gamma_b}\,\Box])$ that holds for all `j` in the domain of θ (n_b is the lock count of `j` in θ) and rule *W4*. Predicate $\mathsf{counts\_ok}(E[\mathsf{pop}_{\gamma_a :: \gamma_{b_1}}\,\Box], \theta)$ holds from $\mathsf{counts\_ok}(E[\mathsf{pop}_{\gamma_b}\,\Box], \theta)$ and the fact that $\mathsf{max}(\gamma_a :: \gamma_{b_1}) = \mathsf{max}(\gamma_b)$.

Case *E-IF*: similar to the previous case.

Case *E-NG* and *E-SN*: similar to case *E-AP*. In the case of rule *E-NG* the use of lemma C.9 will also be required in addition to lemmas C.12 and C.13. In the case of *E-SN*, θ is divided into θ_1 and θ_2 for threads n and n' respectively. Thus, it is shown that the effects of the remaining computation of thread n match θ_1, whereas the effect of the new thread n' matches θ_2.

Lemma C.4 (Well-Formedness) If an expression e is well-typed in the typing context $M;\Delta;\Gamma$, with effect $\gamma;\gamma'$, then $\vdash M;\Delta;\Gamma;\gamma;\gamma'$ holds.

Proof. Straightforward proof by induction on the expression typing derivation. The most interesting case is rule *T-AP*, where it needs to be shown that if $\vdash M;\Delta;\Gamma;\gamma_1;\gamma_2$ and $\vdash M;\Delta;\Gamma;\gamma_2;\gamma_3$ are the well-formedness derivations of expressions e_2 and e_1 respectively and γ_0 is the input effect to the application term, then $\vdash M;\Delta;\Gamma;\gamma_0;\gamma_3$ holds.

The premise that $\gamma_1 = \gamma_0 \oplus \gamma_a$, where γ_a is the annotation of the abstraction type (i.e. the type of e_1) implies that $\gamma_0 \lhd \gamma_1$. $\vdash M;\Delta;\Gamma;\gamma_1;\gamma_2$ and $\vdash M;\Delta;\Gamma;\gamma_2;\gamma_3$ imply that $\gamma_1 \lhd \gamma_2$, $\gamma_2 \lhd \gamma_3$. Thus, $\gamma_0 \lhd \gamma_3$. They also imply that $\mathsf{seq}(\emptyset) \vdash \gamma_3$, $\vdash M$, $M;\Delta \vdash \Gamma$ and $M;\Delta \vdash \gamma_3$. The latter fact and the fact that $\gamma_0 \lhd \gamma_3$ imply that $M;\Delta \vdash \gamma_0$. Thus, $\vdash M;\Delta;\Gamma;\gamma_0;\gamma_3$ holds.

Lemma C.5 (Value-Effect — Using Well-Formedness) If value v is well-typed in the typing context $M;\Delta;\Gamma$, with effect $(\gamma;\gamma)$ and $\vdash M;\Delta;\Gamma;\gamma_1;\gamma_2$, then v is well-typed in the same typing context with effect $(\gamma_1;\gamma_1)$ and $(\gamma_2;\gamma_2)$.

Proof. The proof is trivial, but we provide the key steps behind the proof. The assumption implies that $\vdash M;\Delta;\Gamma;\gamma_1;\gamma_1$ and also $\vdash M;\Delta;\Gamma;\gamma_2;\gamma_2$ hold (trivial). By inversion of the typing derivation of v (for any v) we obtain the well-formedness derivation as well as some other premises (in the case of rules *T-L,T-V,T-F,T-RF,T-Tf,T-Tr,T-FX,T-U*). We may use the latter premises of value typing, which *still hold* (same typing context), along with the latter two well-formedness derivations to formulate the new value typing derivations with effect $(\gamma_1;\gamma_1)$ and $(\gamma_2;\gamma_2)$ respectively. The case for rule *T-RF* or rule *T-FX* can be trivially shown by induction (the base case is the same as for rule *T-F*).

Lemma C.6 (Evaluation Context Sub-typing)

- $M;\Delta;\Gamma \vdash E : \tau \xrightarrow{\gamma_1;\gamma_2} \tau' \,\&\, (\gamma_3;\gamma_4)$
- $\gamma_2 = \gamma_{21} :: \gamma_{22}$ and $\gamma_1 \lhd \gamma_{21}$

if and only if

- $M; \Delta; \Gamma \vdash E : \tau \xrightarrow{\gamma_1;\gamma_{21}} \tau' \,\&\, (\gamma_3;\gamma_5)$
- $\gamma_4 = \gamma_5 :: \gamma_{22}$ and $M; \Delta \vdash \gamma_4$

Proof. Straightforward induction on the evaluation context typing relation. The base case is trivial. The inductive hypothesis is trivial by lemma C.7.

Lemma C.7 (Frame Sub-typing) If the following conditions hold

- $M; \Delta; \Gamma \vdash F : \tau \xrightarrow{\gamma_1;\gamma_2} \tau' \,\&\, (\gamma_3;\gamma_4)$
- $\gamma_2 = \gamma_{21} :: \gamma_{22}$ and $\gamma_1 \lhd \gamma_{21}$

if and only if

- $M; \Delta; \Gamma \vdash F : \tau \xrightarrow{\gamma_1;\gamma_{21}} \tau' \,\&\, (\gamma_3;\gamma_5)$
- $\gamma_4 = \gamma_5 :: \gamma_{22}$ and $M; \Delta \vdash \gamma_4$

Proof. Straightforward case analysis on the frame typing relation.

Lemma C.8 (Typing Equivalence) If $\tau' \simeq \tau$ and $M; \Delta; \Gamma \vdash e : \tau \,\&\, (\gamma;\gamma)$ then $M; \Delta; \Gamma \vdash e : \tau' \,\&\, (\gamma;\gamma)$

Proof. Straightforward induction on the shape of e'.

Lemma C.9 (Variable Substitution) $M; \Delta; \Gamma, x : \tau_1 \vdash e : \tau_2 \,\&\, (\gamma_1;\gamma_2) \wedge M; \emptyset; \emptyset \vdash v : \tau_1 \,\&\, (\gamma;\gamma) \Rightarrow M; \Delta; \Gamma \vdash e[v/x] : \tau_2 \,\&\, (\gamma_1;\gamma_2)$

Proof. Straightforward induction on the expression typing derivation.

Lemma C.10 (Type Well-formedness) $M; \Delta; \Gamma \vdash e : \tau \,\&\, (\gamma;\gamma') \Rightarrow M; \Delta \vdash \tau$

Proof. Straightforward induction on the typing rules.

Lemma C.11 (Location Substitution) If the following hold:

- $M, \imath \mapsto \tau'; \Delta, \rho; \Gamma \vdash e : \tau \,\&\, (\gamma;\gamma')$
- *fresh* n

then $M, \imath \mapsto \tau'; \Delta; \Gamma[\imath@n/\rho] \vdash e[\imath@n/\rho] : \tau[\imath@n/\rho] \,\&\, (\gamma[\imath@n/\rho];\gamma'[\imath@n/\rho])$.

Proof. Proof by induction on the typing derivation of e.

Lemma C.12 (Evaluation Typing Weakening) $M; \Delta; \Gamma \vdash e : \tau \,\&\, (\gamma;\gamma')$, $M; \emptyset \vdash \tau'$ and $\imath \notin \mathsf{dom}(M)$ then $M, \imath \mapsto \tau'; \Delta; \Gamma \vdash e : \tau \,\&\, (\gamma;\gamma')$.

Proof. Proof by induction on the typing derivation of e.

Lemma C.13 (Evaluation Context Typing Weakening) $M; \Delta; \Gamma \vdash E : \tau \xrightarrow{\gamma_1;\gamma_2} \tau' \,\&\, (\gamma;\gamma'), M; \emptyset \vdash \tau'$ and $\imath \notin \mathsf{dom}(M)$ then $M, \imath \mapsto \tau'; \Delta; \Gamma \vdash E : \tau \xrightarrow{\gamma_1;\gamma_2} \tau' \,\&\, (\gamma;\gamma')$.

Proof. Proof by induction on the derivation of E.

Lemma C.14 (Evaluation Context Composition — E) If $M; \Delta; \Gamma \vdash e : \tau \,\&\, (\gamma_a;\gamma_b)$ and $M; \Delta; \Gamma \vdash E : \tau \xrightarrow{\gamma_a;\gamma_b} \tau' \,\&\, (\gamma_1;\gamma_2)$, then $M; \Delta; \Gamma \vdash E[e] : \tau' \,\&\, (\gamma_1;\gamma_2)$.

Proof. Proof by induction on typing derivation of E. The base case is immediate as $\square[e] = e$. The inductive case where $E = E'[F]$, the proof is immediate by inversion of the derivation of E and the application of lemma C.16.

Lemma C.15 (Evaluation Context Decomposition — E) If $M; \Delta; \Gamma \vdash E[e] : \tau' \,\&\, (\gamma_1;\gamma_2)$, then there exists a γ_a, γ_b and τ such that $M; \Delta; \Gamma \vdash e : \tau \,\&\, (\gamma_a;\gamma_b)$ and $M; \Delta; \Gamma \vdash E : \tau \xrightarrow{\gamma_a;\gamma_b} \tau' \,\&\, (\gamma_1;\gamma_2)$.

Proof. Proof by induction on the structure of E. The base case is immediate by using the well-formedness derivation for the type and typing context of e (i.e., lemmas C.4 and C.10) and the application rule *E0*. The inductive case, where $E[e] = E'[F][e]$ is immediate by lemma C.17 and rule *E1*.

Lemma C.16 (Evaluation Context Composition — F) If $M; \Delta; \Gamma \vdash e : \tau \,\&\, (\gamma_a;\gamma_b)$ and $M; \Delta; \Gamma \vdash F : \tau \xrightarrow{\gamma_a;\gamma_b} \tau' \,\&\, (\gamma_1;\gamma_2)$, then $M; \Delta; \Gamma \vdash F[e] : \tau' \,\&\, (\gamma_1;\gamma_2)$.

Proof. Proof by case analysis on typing derivation of F. The premises required to construct the typing derivation of $F[e]$ are given as premises of the typing derivation of F.

Lemma C.17 (Evaluation Context Decomposition — F) If $M; \Delta; \Gamma \vdash F[e] : \tau' \,\&\, (\gamma_1;\gamma_2)$, then there exists a γ_a, γ_b and τ such that $M; \Delta; \Gamma \vdash e : \tau \,\&\, (\gamma_a;\gamma_b)$ and $M; \Delta; \Gamma \vdash F : \tau \xrightarrow{\gamma_a;\gamma_b} \tau' \,\&\, (\gamma_1;\gamma_2)$.

Proof. Proof by case analysis on the structure of F. The premises required for each case (i.e., rules *F1-F10*) are given by the premises of the typing derivation of $F[e]$.

Lemma C.18 (Progress) Let $S; T$ be a closed well-typed configuration with $M \vdash S; T$ then $S; T$ is not stuck ($\vdash S; T$).

Proof. It suffices to show that for any thread in T, a step can be performed or $block$ predicate holds for it. Let n be an arbitrary thread in T such that $T = T_1, n : \theta; e$ for some T_1. By inversion of the typing derivation of $S; T$ we have that $M; \emptyset; \emptyset \vdash_t \theta; e : \langle\rangle \,\&\, (\gamma;\gamma')$, $\mathsf{mutex}(T)$, and $M \vdash S$.

If e is a *value* then by inversion of $M; \emptyset; \emptyset \vdash_t \theta; e : \langle\rangle \,\&\, (\gamma;\gamma')$, we obtain that $\gamma = \gamma'$, $E[e] = \square[()]$ and $\forall \imath.\theta(\imath) = (0,0)$, as a consequence of the following:

$\forall r^\kappa \in \gamma.\kappa = (0,0)$ and $\mathsf{counts\_ok}(\square[\mathsf{pop}_\gamma\ \square], \theta)$. Thus, rule *E-T* can be applied.

If e is not a value then according to lemma C.19, there exists a redex u and an evaluation context E such that $e = E[u]$. By inversion of the thread typing derivation for e we obtain that $M; \emptyset; \emptyset \vdash u : \tau \,\&\, (\gamma_a;\gamma_b)$, $M; \emptyset; \emptyset \vdash E : \tau \xrightarrow{\gamma_a;\gamma_b} \langle\rangle \,\&\, (\gamma;\gamma')$, $\mathsf{counts\_ok}(E[\mathsf{pop}_{\gamma_b}\ \square], \theta)$ hold.

Then, we proceed by performing a case analysis on u:

Case $(\lambda x.\, e'\ \mathtt{as}\ \tau\ v)^{\mathsf{par}}$: it suffices to show that $(\theta_1, \theta_2) = \mathsf{split}(\theta\,,\ \mathsf{max}(\gamma_c))$ is defined, where γ_c is the annotation of type τ. If $\mathsf{max}(\gamma_c)$ is empty, then the proof is immediate from the base case of split function. Otherwise, we must show that for all $\imath$, the count $\theta(\imath)$ is greater than or equal to the sum of all $(\imath@n)^\kappa$ in $\mathsf{max}(\gamma_c)$. This can be shown by considering $\mathsf{par} \vdash \gamma_b = \gamma_a \oplus \gamma_c$ (i.e., the max counts in γ_c are less than or equal to the max counts in γ_b), which can be obtained by inversion of the typing derivation of $(\lambda x.\, e'\ \mathtt{as}\ \tau\ v)^{\mathsf{par}}$, and the exact correspondence between static (γ_b) and dynamic counts (i.e, $\mathsf{counts\_ok}(E[\mathsf{pop}_{\gamma_b}\ \square], \theta)$). Thus, rule *E-SN* can be applied to perform a single step.

Case $\mathtt{share}\ \mathtt{loc}_\imath$: $\mathsf{counts\_ok}(E[\mathsf{pop}_{\gamma_b}\ \square], \theta)$ establishes an exact correspondence between dynamic and static counts. The typing derivation implies that $\gamma_a(\imath@n_1) \geq (2,0)$, for some n_1 existentially bound in the premise of the derivation. Therefore, $\theta(\imath) \geq (1,0)$. It is possible to perform a single step using rule *E-SH*. The cases for $\mathtt{release}\ \mathtt{loc}_\imath$ and $\mathtt{unlock}\ \mathtt{loc}_\imath$ can be shown in a similar manner.

Case $\texttt{lock}_{\gamma_a}\ \texttt{loc}_\iota$: similarly to the case we can show that $\theta(\iota) = (n_1, n_2)$ and n_1 is positive. If n_2 is positive, rule *E-LK1* can be applied. Otherwise, n_2 is zero. Let ϵ be equal to $\mathsf{locked}(T_1) \cap \mathsf{lockset}(\iota, 1, E[\texttt{pop}_{\gamma_a}\ \square])$. If ϵ is empty then rule *E-LK0* can be applied in order to perform a single step. Otherwise, $\mathsf{blocked}(T, n)$ predicate holds and the configuration is not stuck.

Case $\texttt{deref}\ \texttt{loc}_\iota$: it can be trivially shown (as in the previous case of $share$ that we proved $\theta(\iota) \geq (1, 0)$), that $\theta(\iota) \geq (1, 1)$ and since $\mathsf{mutex}(T_1, n : \theta; E[\texttt{deref}\ \texttt{loc}_\iota])$ holds, then $\iota \notin \mathsf{locked}(T_1)$ and thus rule *E-D* can be used to perform a step. The case of $\texttt{loc}_\iota := v$ can be shown in a similar manner.

Case $(\lambda x.\, e'\ \texttt{as}\ \tau'\ v)^{\mathsf{seq}(\gamma_a)}$: a step can be taken by rule *E-A*.

Case $\texttt{pop}_{\gamma_b}\ e'[v/x]$: a step can be taken by rule *E-PP*.

Case $(\Lambda\rho.\, f)\ [\iota@n_1]$: a step can be taken by rule *E-RP*.

Case $(\texttt{fix}\ x : \tau.\, f\ \ v)^{\mathsf{seq}(\gamma_a)}$: a step can be taken by rule *E-FX*.

Case $\texttt{if true then}\ e_1\ \texttt{else}\ e_2$: a step can be taken by rule *E-IT*.

Case $\texttt{if false then}\ e_1\ \texttt{else}\ e_2$: a step can be taken by rule *E-IF*.

Lemma C.19 (Redex) If $M; \Delta; \Gamma \vdash e : \tau\ \&\ (\gamma_1;\gamma_2)$ and e is *not* a value then $M; \Delta; \Gamma \vdash E'[u] : \tau\ \&\ (\gamma_1;\gamma_2)$ such that $E'[u] = e$.

Proof. By induction on the shape of e. The key idea is to convert typing derivations of e, when e is not a redex, to typing derivations of the form $E'[e']$ and apply induction for e'.

Appendix D

Formal semantics and proof of soundness for Chapter 7

D.1 Language syntax

Language syntax

Value	v	$::=$	$() \mid \texttt{true} \mid \texttt{false} \mid f \mid \texttt{lk}_\iota$
Expression	e	$::=$	$x \mid v \mid (e\ e)^\xi \mid (e)\,[r] \mid \texttt{pop}_\gamma\ e$
		$\mid$	$\texttt{newlock}\ \rho, x\ \texttt{in}\ e$
		$\mid$	$\texttt{lock}_\gamma\ e \mid \texttt{unlock}\ e$
		$\mid$	$\texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e$
Function	f	$::=$	$\lambda x.\, e \mid \Lambda\rho.\, f \mid \texttt{fix}\ x.\, f$
Type	τ	$::=$	$\langle\rangle \mid \texttt{Bool} \mid \texttt{Lk}(r) \mid \tau \xrightarrow{\gamma} \tau \mid \forall\rho.\, \tau$
Lock	r	$::=$	$\rho \mid \iota$
Calling mode	ξ	$::=$	$\mathsf{seq}(\gamma) \mid \mathsf{par}$
Operation	κ	$::=$	$+ \mid -$
Effect	γ	$::=$	$\emptyset \mid r^\kappa, \gamma \mid \gamma\,?\,\gamma, \gamma$

D.2 Operational semantics

Auxiliary syntax for operational semantics

Lock Store	S	$::=$	$\emptyset \mid S, \iota \mapsto n; n; \epsilon; \epsilon$
Threads	T	$::=$	$\emptyset \mid T, n : e$
Configuration	C	$::=$	$S; T$
Lockset	ϵ	$::=$	$\emptyset \mid \epsilon, \iota$

Evaluation context

$$
\begin{array}{lrl}
\textbf{Context} & E & ::= \Box \mid E[F] \\
\textbf{Frame} & F & ::= (\Box\ e)^{\xi} \mid (v\ \Box)^{\xi} \mid (\Box)\,[r] \mid \mathtt{pop}_{\gamma}\ \Box \\
 & & \mid\ \mathtt{lock}_{\gamma_1}\ \Box \mid \mathtt{unlock}\ \Box \mid \mathtt{if}\ \Box\ \mathtt{then}\ e\ \mathtt{else}\ e \\
\textbf{Redex} & u & ::= (v'\ v)^{\xi} \mid (f)\,[r] \mid \mathtt{lock}_{\gamma_1}\ v \mid \mathtt{unlock}\ v \\
 & & \mid\ \mathtt{newlock}\ \rho, x\ \mathtt{in}\ e_1 \mid \mathtt{if}\ v\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 \mid \mathtt{pop}_{\gamma}\ v
\end{array}
$$

Helper relations

$$
\mathsf{run}(\gamma, \imath, n) = \begin{cases}
\emptyset & \text{if } n = 0 \\
\mathsf{run}(\gamma', \imath, n+1) & \text{if } \gamma = \imath^{+}, \gamma' \text{ and } n > 0 \\
\mathsf{run}(\gamma', \imath, n-1) & \text{if } \gamma = \imath^{-}, \gamma' \text{ and } n > 0 \\
\mathsf{run}(\gamma', \imath, n) \cup \{\mathtt{j}\} & \text{if } \gamma = \mathtt{j}^{+}, \gamma' \text{ and } n > 0 \\
\mathsf{run}(\gamma', \imath, n) & \text{if } \gamma = \mathtt{j}^{-}, \gamma' \text{ and } n > 0 \\
\mathsf{run}((\gamma_1 :: \gamma'), \imath, n) \cup \mathsf{run}((\gamma_2 :: \gamma'), \imath, n) & \text{if } \gamma = \gamma_1 \,?\, \gamma_2, \gamma' \text{ and } n > 0
\end{cases}
$$

$$
\mathsf{stack}(E) = \begin{cases}
\emptyset & \text{if } E = \Box \\
\mathsf{stack}(E') & \text{if } E = E'[F] \text{ and } F \neq \mathtt{pop}_{\gamma'}\ \Box \\
\gamma' :: \mathsf{stack}(E') & \text{if } E = E'[\mathtt{pop}_{\gamma'}\ \Box]
\end{cases}
$$

$$
\mathsf{available}(S, n) = \begin{cases}
\emptyset & \text{if } S = \emptyset \\
\mathsf{available}(S', n) \cup \{\imath\} & \text{if } S = S', \imath \mapsto n_1; n_2; \epsilon_1; \epsilon_2 \text{ and } n_1 = n \text{ or } n_2 = 0 \\
\mathsf{available}(S', n) & \text{if } S = S', \imath \mapsto n_1, n_2; \epsilon_1; \epsilon_2 \text{ and } n_1 \neq n \text{ and } n_2 > 0
\end{cases}
$$

$$
\mathsf{dom}(S) = \{\imath \mid \imath \mapsto n_1; n_2; \epsilon_a; \epsilon_b \in S\}
$$

Reduction relation

$$
\frac{}{S; T, n : E[\mathtt{if\ true\ then}\ e_1\ \mathtt{else}\ e_2] \rightsquigarrow S; T, n : E[e_1]}\ \textit{(E-IT)}
$$

$$
\frac{}{S; T, n : E[\mathtt{if\ false\ then}\ e_1\ \mathtt{else}\ e_2] \rightsquigarrow S; T, n : E[e_2]}\ \textit{(E-IF)}
$$

$$
\frac{v' = \mathtt{fix}\ x.\ f}{S; T, n : E[(v'\ v)^{\mathsf{seq}(\gamma)}] \rightsquigarrow S; T, n : E[(f[v'/x]\ v)^{\mathsf{seq}(\gamma)}]}\ \textit{(E-FX)}
$$

$$
\frac{\text{fresh } n'}{S; T, n : E[(v'\ v)^{\mathsf{par}}] \rightsquigarrow S; T, n : E[()], n' : \Box[(v'\ v)^{\mathsf{seq}(\emptyset)}]}\ \textit{(E-SN)}
$$

$$
\frac{}{S; T, n : \Box[()] \rightsquigarrow S; T}\ \textit{(E-T)} \qquad \frac{}{S; T, n : E[((\lambda x.\, e_1)\ v)^{\mathsf{seq}(\gamma)}] \rightsquigarrow S; T, n : E[\mathtt{pop}_{\gamma}\ e_1[v/x]]}\ \textit{(E-A)}
$$

$$
\frac{}{S; T, n : E[(\Lambda \rho.\, f)[\imath]] \rightsquigarrow S; T, n : E[f[\imath/\rho]]}\ \textit{(E-RP)} \qquad \frac{}{S; T, n : E[\mathtt{pop}_{\gamma}\ v] \rightsquigarrow S; T, n : E[v]}\ \textit{(E-PP)}
$$

$$
\frac{\text{fresh } \imath \qquad S' = S, \imath \mapsto n; 0; \emptyset; \emptyset}{S; T, n : E[\mathtt{newlock}\ \rho, x\ \mathtt{in}\ e_1] \rightsquigarrow S'; T, n : E[e_1[\imath/\rho][\mathtt{lk}_{\imath}/x]]}\ \textit{(E-NG)}
$$

$$\frac{\begin{array}{c} S(\imath) = n_1; 0; \epsilon_1; \epsilon_2 \qquad S' = S[\imath \mapsto n; 1; \mathsf{dom}(S); \epsilon] \\ \epsilon = \mathsf{run}(\mathsf{stack}(E[\mathsf{pop}_{\gamma_1}\ \square]), \imath, 1) \qquad \epsilon \cup \{\imath\} \subseteq \mathsf{available}(S, n) \end{array}}{S; T, n : E[\mathtt{lock}_{\gamma_1}\ \mathtt{lk}_\imath] \rightsquigarrow S'; T, n : E[()]} \textit{ (E-LK0)}$$

$$\frac{S(\imath) = n; n_2; \epsilon_1; \epsilon_2 \qquad n_2 > 0 \qquad S' = S[\imath \mapsto n; n_2 + 1; \epsilon_1; \epsilon_2]}{S; T, n : E[\mathtt{lock}_{\gamma_1}\ \mathtt{lk}_\imath] \rightsquigarrow S'; T, n : E[()]} \textit{ (E-LK1)}$$

$$\frac{S(\imath) = n; n_2; \epsilon_1; \epsilon_2 \qquad n_2 > 0 \qquad S' = S[\imath \mapsto n; n_2 - 1; \epsilon_1; \epsilon_2]}{S; T, n : E[\mathtt{unlock}\ \mathtt{lk}_\imath] \rightsquigarrow S'; T, n : E[()]} \textit{ (E-UL)}$$

D.3 Static semantics

Static semantics syntax

Type variable list $\square ::= \emptyset \mid \Delta, \rho$
Memory List $M ::= \emptyset \mid M, \imath$
Variable list $\square ::= \emptyset \mid \Gamma, x : \tau$

Well-formedness relation

Constraint Well-formedness

$$\frac{}{M; \Delta \vdash \emptyset} \qquad \frac{r \in M \cup \Delta \qquad M; \Delta \vdash \gamma_1}{M; \Delta \vdash \gamma_1, r^\kappa} \qquad \frac{M; \Delta \vdash \gamma_1 \qquad M; \Delta \vdash \gamma_2 \qquad M; \Delta \vdash \gamma_3}{M; \Delta \vdash \gamma_1, \gamma_2\, ?\, \gamma_3}$$

Type Well-formedness

$$\frac{}{M; \Delta \vdash \mathtt{Bool}} \qquad \frac{M; \Delta, \rho \vdash \tau}{M; \Delta \vdash \forall \rho.\, \tau} \qquad \frac{r \in M \cup \Delta}{M; \Delta \vdash \mathtt{Lk}(r)}$$

$$\frac{M; \Delta \vdash \tau_1 \qquad M; \Delta \vdash \gamma_1 \qquad M; \Delta \vdash \tau_2}{M; \Delta \vdash \tau_1 \xrightarrow{\gamma_1} \tau_2} \qquad \frac{}{M; \Delta \vdash \langle\rangle}$$

Γ Well-formedness

$$\frac{}{M; \Delta \vdash \emptyset} \qquad \frac{M; \Delta \vdash \tau_1 \qquad x \notin \mathsf{dom}(\Gamma_1) \qquad M; \Delta \vdash \Gamma_1}{M; \Delta \vdash \Gamma_1, x : \tau_1}$$

Typing rules

$$\frac{x : \tau \in \Gamma \quad M;\Delta \vdash \Gamma \quad M;\Delta \vdash \gamma}{M;\Delta;\Gamma \vdash x : \tau \,\&\, (\gamma;\gamma)} \textit{(T-V)} \qquad \frac{M;\Delta \vdash \Gamma \quad M;\Delta \vdash \gamma}{M;\Delta;\Gamma \vdash \texttt{true} : \texttt{Bool} \,\&\, (\gamma;\gamma)} \textit{(T-T)}$$

$$\frac{M;\Delta \vdash \Gamma \quad M;\Delta \vdash \gamma}{M;\Delta;\Gamma \vdash \texttt{false} : \texttt{Bool} \,\&\, (\gamma;\gamma)} \textit{(T-F)} \qquad \frac{M;\Delta \vdash \Gamma \quad M;\Delta \vdash \gamma}{M;\Delta;\Gamma \vdash () : \langle\rangle \,\&\, (\gamma;\gamma)} \textit{(T-U)}$$

$$\frac{M;\Delta,\rho;\Gamma \vdash f : \tau \,\&\, (\gamma;\gamma)}{M;\Delta;\Gamma \vdash \Lambda\rho.\, f : \forall\rho.\, \tau \,\&\, (\gamma;\gamma)} \textit{(T-RF)} \qquad \frac{M;\Delta \vdash \Gamma \quad M;\Delta \vdash \gamma \quad \iota \in M}{M;\Delta;\Gamma \vdash \texttt{lk}_\iota : \texttt{Lk}(\iota) \,\&\, (\gamma;\gamma)} \textit{(T-L)}$$

$$\frac{\begin{array}{c} M;\Delta \vdash \Gamma \quad M;\Delta \vdash \gamma \quad \tau \equiv \tau_1 \xrightarrow{\gamma_b} \tau_2 \\ M;\Delta \vdash \tau \quad M;\Delta;\Gamma, x : \tau_1 \vdash e_1 : \tau_2 \,\&\, (\emptyset;\gamma_b) \end{array}}{M;\Delta;\Gamma \vdash \lambda x.\, e_1 : \tau \,\&\, (\gamma;\gamma)} \textit{(T-FN)}$$

$$\frac{\begin{array}{c} R;M;\Delta;\Gamma, x : \tau_a \vdash f : \tau_b \,\&\, (\gamma;\gamma) \\ \tau_a \equiv \tau_1 \xrightarrow{\gamma_a} \tau_2 \quad \tau_b \equiv \tau_1 \xrightarrow{\gamma_b} \tau_2 \quad \gamma_a = \mathsf{summary}(\gamma_b) \end{array}}{R;M;\Delta;\Gamma \vdash \texttt{fix}\, x.\, f : \tau_a \,\&\, (\gamma;\gamma)} \textit{(T-FX)}$$

$$\frac{M;\Delta;\Gamma \vdash e : \texttt{Lk}(r) \,\&\, (r^+, \gamma;\gamma')}{M;\Delta;\Gamma \vdash \texttt{lock}_\gamma\, e : \langle\rangle \,\&\, (\gamma;\gamma')} \textit{(T-LK)} \qquad \frac{M;\Delta;\Gamma \vdash e : \texttt{Lk}(r) \,\&\, (r^-, \gamma;\gamma')}{M;\Delta;\Gamma \vdash \texttt{unlock}\, e : \langle\rangle \,\&\, (\gamma;\gamma')} \textit{(T-UL)}$$

$$\frac{\begin{array}{c} M;\Delta;\Gamma \vdash e_1 : \tau_1 \xrightarrow{\gamma_a} \tau_2 \,\&\, (\gamma_1;\gamma') \\ M;\Delta;\Gamma \vdash e_2 : \tau_1 \,\&\, (\gamma_a :: \gamma;\gamma_1) \end{array}}{M;\Delta;\Gamma \vdash (e_1\ e_2)^{\mathsf{seq}(\gamma)} : \tau_2 \,\&\, (\gamma;\gamma')} \textit{(T-SA)}$$

$$\frac{\begin{array}{c} \forall r \in \mathsf{dom}(\gamma_a).\, r;0 \vdash_{ok} \gamma_a \\ M;\Delta;\Gamma \vdash e_1 : \tau_1 \xrightarrow{\gamma_a} \langle\rangle \,\&\, (\gamma_1;\gamma') \quad M;\Delta;\Gamma \vdash e_2 : \tau_1 \,\&\, (\gamma;\gamma_1) \end{array}}{M;\Delta;\Gamma \vdash (e_1\ e_2)^{\mathsf{par}} : \langle\rangle \,\&\, (\gamma;\gamma')} \textit{(T-PA)}$$

$$\frac{r \in M \cup \Delta \quad M;\Delta;\Gamma \vdash e_1 : \forall\rho.\, \tau \,\&\, (\gamma;\gamma')}{M;\Delta;\Gamma \vdash (e_1)\,[r] : \tau[r/\rho] \,\&\, (\gamma;\gamma')} \textit{(T-RP)}$$

$$\frac{\begin{array}{c} M;\Delta \vdash \tau \quad \rho \notin \mathsf{dom}(\gamma) \quad \rho;0 \vdash_{ok} \gamma' \\ M;\Delta,\rho;\Gamma, x : \texttt{Lk}(\rho) \vdash e_1 : \tau \,\&\, (\gamma;\gamma') \end{array}}{M;\Delta;\Gamma \vdash \texttt{newlock}\ \rho, x\ \texttt{in}\ e_1 : \tau \,\&\, (\gamma;\gamma' \setminus \rho)} \textit{(T-NG)}$$

$$\frac{M;\Delta \vdash \gamma \quad M;\Delta;\Gamma \vdash e : \tau \,\&\, (\emptyset;\gamma')}{M;\Delta;\Gamma \vdash \texttt{pop}_\gamma\ e : \tau \,\&\, (\gamma;\gamma' :: \gamma)} \textit{(T-PP)}$$

$$\frac{\begin{array}{c} M;\Delta;\Gamma \vdash e_1 : \texttt{Bool} \,\&\, (\gamma_1\,?\,\gamma_2, \gamma;\gamma') \\ M;\Delta;\Gamma \vdash e_2 : \tau \,\&\, (\gamma;\gamma_1 :: \gamma) \quad M;\Delta;\Gamma \vdash e_3 : \tau \,\&\, (\gamma;\gamma_2 :: \gamma) \end{array}}{M;\Delta;\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \tau \,\&\, (\gamma;\gamma')} \textit{(T-IF)}$$

where $\mathsf{summary}(\gamma_a) = \gamma_1 :: \gamma_2 :: \gamma_3$ if $\mathsf{rsummary}(\gamma_a) = \gamma_1;\gamma_2;\gamma_3$

Lock removal

$$\frac{\gamma' = \gamma \setminus r}{\gamma' = r^{\kappa}, \gamma \setminus r}\ \textit{(M0)} \qquad \frac{r' \neq r \quad \gamma' = \gamma \setminus r'}{r^{\kappa}, \gamma' = r^{\kappa}, \gamma \setminus r'}\ \textit{(M1)} \qquad \frac{}{\emptyset = \emptyset \setminus r}\ \textit{(M3)}$$

$$\frac{\gamma_1' = \gamma_1 \setminus r' \quad \gamma_2' = \gamma_2 \setminus r' \quad \gamma_3' = \gamma_3 \setminus r'}{\gamma_2' \,?\, \gamma_3', \gamma_1' = \gamma_2 \,?\, \gamma_3, \gamma_1 \setminus r}\ \textit{(M4)}$$

Effect validation

$$\frac{0 \leq n \quad r; n+1 \vdash_{ok} \gamma}{r; n \vdash_{ok} r^{+}, \gamma}\ \textit{(OK1)} \qquad \frac{r; n-1 \vdash_{ok} \gamma \quad n > 0}{r; n \vdash_{ok} r^{-}, \gamma}\ \textit{(OK2)}$$

$$\frac{0 \leq n \quad r; n \vdash_{ok} \gamma \quad r \neq r'}{r; n \vdash_{ok} r'^{\kappa}, \gamma}\ \textit{(OK3)}$$

$$\frac{}{r; 0 \vdash_{ok} \emptyset}\ \textit{(OK4)} \qquad \frac{0 \leq n \quad r; n \vdash_{ok} \gamma_1 :: \gamma \quad r; n \vdash_{ok} \gamma_2 :: \gamma}{r; n \vdash_{ok} \gamma_1 \,?\, \gamma_2, \gamma}\ \textit{(OK5)}$$

Summary of recursive effect

$$\frac{\begin{array}{cc} r; n_a \vdash_{ok} \gamma_a :: (r^{-})^{n_b} \quad r \in \mathsf{dom}(\gamma_a) & \forall n_c.\, \neg\, (r; n_a - 1 \vdash_{ok} \gamma_a :: (r^{-})^{n_c}) \\ \gamma_3 = \{r^{+}, r^{-} \mid r^{+} \in \gamma_a\} & \mathsf{rsummary}(\gamma_a \setminus r) = \gamma_1; \gamma_2; \gamma_0 \end{array}}{\mathsf{rsummary}(\gamma_a) = \gamma_3 :: \gamma_1; (r^{+})^{n_b} :: \gamma_2; (r^{-})^{n_a} :: \gamma_0}\ \textit{(PX0)}$$

$$\frac{}{\mathsf{rsummary}(\emptyset) = \emptyset; \emptyset; \emptyset}\ \textit{(PX1)}$$

D.4 Type safety

Evaluation context typing

Sub-effect $\gamma \lhd \gamma' \equiv \exists \gamma''.\gamma' = \gamma'' :: \gamma$

$$\frac{M; \Delta \vdash \Gamma \quad M; \Delta \vdash \gamma_1 \quad M; \Delta \vdash \gamma_2 \quad M; \Delta \vdash \tau}{M; \Delta; \Gamma \vdash \Box : \tau \xrightarrow{\gamma_1;\gamma_2} \tau \,\&\, (\gamma_1;\gamma_2)}\ \textit{(E0)}$$

$$\frac{\begin{array}{c} M; \Delta; \Gamma \vdash E : \tau_2 \xrightarrow{\gamma_5;\gamma_6} \tau_3 \,\&\, (\gamma_1;\gamma_2) \\ M; \Delta; \Gamma \vdash F : \tau_1 \xrightarrow{\gamma_3;\gamma_4} \tau_2 \,\&\, (\gamma_5;\gamma_6) \end{array}}{M; \Delta; \Gamma \vdash E[F] : \tau_1 \xrightarrow{\gamma_3;\gamma_4} \tau_3 \,\&\, (\gamma_1;\gamma_2)}\ \textit{(E1)}$$

$$\frac{M;\Delta \vdash \gamma_2 \quad \gamma_1 \lhd \gamma_2 \quad M;\Delta \vdash \tau_2 \quad M;\Delta;\Gamma \vdash e_2 : \tau_1 \,\&\, (\gamma_a :: \gamma;\gamma_1)}{M;\Delta;\Gamma \vdash (\square\ e_2)^{\mathsf{seq}(\gamma)} : (\tau_1 \xrightarrow{\gamma_a} \tau_2) \xrightarrow{\gamma_1;\gamma_2} \tau_2 \,\&\, (\gamma;\gamma_2)} \quad \textit{(F1)}$$

$$\frac{\gamma_2 \lhd \gamma_3 \quad M;\Delta \vdash \tau_1 \xrightarrow{\gamma_a} \langle\rangle \quad M;\Delta;\Gamma \vdash e_2 : \tau_1 \,\&\, (\gamma_1;\gamma_2)}{M;\Delta;\Gamma \vdash (\square\ e_2)^{\mathsf{par}} : (\tau_1 \xrightarrow{\gamma_a} \langle\rangle) \xrightarrow{\gamma_2;\gamma_3} \langle\rangle \,\&\, (\gamma_1;\gamma_3)} \quad \textit{(F2)}$$

$$\frac{\gamma_2 = \gamma_1 :: \gamma_a \quad M;\Delta \vdash \gamma_3 \quad \gamma_2 \lhd \gamma_3 \quad M;\Delta;\Gamma \vdash v_1 : \tau_1 \xrightarrow{\gamma_a} \tau_2 \,\&\, (\gamma_3;\gamma_3)}{M;\Delta;\Gamma \vdash (v_1\ \square)^{\mathsf{seq}(\gamma_1)} : \tau_1 \xrightarrow{\gamma_2;\gamma_3} \tau_2 \,\&\, (\gamma_1;\gamma_3)} \quad \textit{(F3)}$$

$$\frac{M;\Delta \vdash \gamma_2 \quad \gamma_1 \lhd \gamma_2 \quad M;\Delta;\Gamma \vdash v_1 : \tau_1 \xrightarrow{\gamma_a} \langle\rangle \,\&\, (\gamma_2;\gamma_2)}{M;\Delta;\Gamma \vdash (v_1\ \square)^{\mathsf{par}} : \tau_1 \xrightarrow{\gamma_1;\gamma_2} \langle\rangle \,\&\, (\gamma_1;\gamma_2)} \quad \textit{(F4)}$$

$$\frac{M;\Delta \vdash \tau \quad M;\Delta \vdash \gamma' \quad \gamma' = \gamma_2 :: \gamma}{M;\Delta;\Gamma \vdash \mathtt{pop}_\gamma\ \square : \tau \xrightarrow{\emptyset;\gamma_2} \tau \,\&\, (\gamma;\gamma')} \quad \textit{(F5)}$$

$$\frac{M;\Delta \vdash \Gamma \quad M;\Delta \vdash \gamma' \quad M;\Delta \vdash \mathtt{Lk}(r) \quad \gamma_1 = r^-, \gamma \quad \gamma_1 \lhd \gamma'}{M;\Delta;\Gamma \vdash \mathtt{unlock}\,\square : \mathtt{Lk}(r) \xrightarrow{\gamma_1;\gamma'} \langle\rangle \,\&\, (\gamma;\gamma')} \quad \textit{(F6)}$$

$$\frac{M;\Delta \vdash \Gamma \quad M;\Delta \vdash \gamma' \quad M;\Delta \vdash \mathtt{Lk}(r) \quad \gamma_1 = r^+, \gamma \quad \gamma_1 \lhd \gamma'}{M;\Delta;\Gamma \vdash \mathtt{lock}_\gamma\,\square : \mathtt{Lk}(r) \xrightarrow{\gamma_1;\gamma'} \langle\rangle \,\&\, (\gamma;\gamma')} \quad \textit{(F7)}$$

$$\frac{\gamma_3 = \gamma_1 \,?\, \gamma_2, \gamma \quad M;\Delta \vdash \gamma' \quad \gamma_3 \lhd \gamma' \quad M;\Delta;\Gamma \vdash e_2 : \tau \,\&\, (\gamma;\gamma_1 :: \gamma) \quad M;\Delta;\Gamma \vdash e_3 : \tau \,\&\, (\gamma;\gamma_2 :: \gamma)}{M;\Delta;\Gamma \vdash \mathtt{if}\ \square\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3 : \mathtt{Bool} \xrightarrow{\gamma_3;\gamma'} \tau \,\&\, (\gamma;\gamma')} \quad \textit{(F8)}$$

$$\frac{M;\Delta \vdash \gamma' \quad M;\Delta \vdash \forall\rho.\,\tau \quad r \in M \cup \Delta \quad \gamma \lhd \gamma'}{M;\Delta;\Gamma \vdash (\square)\,[r] : \forall\rho.\,\tau \xrightarrow{\gamma;\gamma'} \tau[r/\rho] \,\&\, (\gamma;\gamma')} \quad \textit{(F9)}$$

Program typing

$$\mathsf{locks}(S,\iota,n) = \begin{cases} n_2 & \text{if } S(\iota) = (n; n_2; \epsilon_1; \epsilon_2) \\ 0 & \text{if } S(\iota) = (n_1; n_2; \epsilon_1; \epsilon_2) \wedge n_1 \neq n \end{cases}$$

$$\mathsf{deadlocked}(T) \equiv T \supseteq T_1, n_0 : E[\mathtt{lock}_{\gamma_0}\,\mathtt{lk}_{\iota_0}], \ldots, n_k : E_k[\mathtt{lock}_{\gamma_k}\,\mathtt{lk}_{\iota_k}] \wedge k > 0 \wedge \forall m_1 \in [0,k].\, m_2 = (m_1+1)\, mod (k+1) \wedge \mathsf{locks}(S, \iota_{m_2}, m_1) > 0$$

$$\mathsf{dom}(\gamma) = \begin{cases} \emptyset & \text{if } \gamma = \emptyset \\ \mathsf{dom}(\gamma') \cup \mathsf{dom}(\gamma_1) \cup \mathsf{dom}(\gamma_2) & \text{if } \gamma = \gamma_1 \,?\, \gamma_2, \gamma' \\ \mathsf{dom}(\gamma') \cup \{r\} & \text{if } \gamma = r^\kappa, \gamma' \end{cases}$$

$$\mathsf{dom}(T) = \{n \mid n : e \in T\}$$

Configuration Typing

$$\frac{S; M \vdash T \qquad M = \mathsf{dom}(S)}{M \vdash S; T}$$

Thread Effect Consistency

$$\frac{\begin{array}{c}\iota; n_1 \vdash_{ok} \gamma \qquad \epsilon_3 = \mathsf{run}(\gamma, \iota, n_1) \\ n; \gamma \vdash S \qquad \epsilon_1 \cap \epsilon_3 \subseteq \epsilon_2\end{array}}{n; \gamma \vdash S, \iota \mapsto n; n_1; \epsilon_1; \epsilon_2} \qquad \frac{\iota; 0 \vdash_{ok} \gamma \qquad n \neq n_1 \qquad n; \gamma \vdash S}{n; \gamma \vdash S, \iota \mapsto n_1; n_2; \epsilon_1; \epsilon_2} \qquad \frac{}{n; \gamma \vdash \emptyset}$$

Thread Typing

$$\frac{}{S; M \vdash \emptyset} \qquad \frac{\begin{array}{c}M; \emptyset; \emptyset \vdash e : \langle\rangle \,\&\, (\emptyset;\gamma) \qquad S; M \vdash T \\ n \notin \mathsf{dom}(T) \qquad n; \gamma \vdash S\end{array}}{S; M \vdash T, n : e}$$

Not Stuck

$$\frac{}{\vdash S; \emptyset} \qquad \frac{\vdash S; T \qquad S; T, n : e \rightsquigarrow S'; T' \qquad T \subseteq T'}{\vdash S; T, n : e}$$

$$\frac{\begin{array}{c}\vdash S; T \qquad \mathsf{locks}(S, \iota, n) = 0 \\ \mathsf{run}(\mathsf{stack}(E[\mathsf{pop}_\gamma\ \square), \iota; 1) = \epsilon \qquad \epsilon \cup \{\iota\} \supset \mathsf{available}(S, n)\end{array}}{\vdash S; T, n : E[\mathtt{lock}_\gamma\ \mathtt{lk}_\iota]}$$

Multi-step evaluation

$$\frac{n > 0 \qquad S; T \rightsquigarrow^{n-1} S_{n-1}; T_{n-1} \qquad S_{n-1}; T_{n-1} \rightsquigarrow S_n; T_n}{S; T \rightsquigarrow^{n} S_n; T_n}\ \textit{(E-M1)} \qquad \frac{}{S; T \rightsquigarrow^{0} S; T}\ \textit{(E-M2)}$$

Main theorems

Safety

$$S_0; T_0 \equiv \emptyset; 0 : e \wedge \emptyset \vdash S_0; T_0 \ \wedge\ S_0; T_0 \rightsquigarrow^{n} S'; T' \Rightarrow \vdash S'; T' \ \wedge\ \neg\mathsf{deadlocked}(T')$$

Preservation

$$M \vdash S; T \ \wedge\ \ S; T \rightsquigarrow S'; T' \Rightarrow \exists M' \supseteq M.\ M' \vdash S'; T'$$

Progress

$$M \vdash S; T \Rightarrow \vdash S; T$$

Deadlock Freedom

$$\emptyset; 0 : e \rightsquigarrow^n S_n; T_n \wedge \forall \imath \in [0,n]. \exists M_\imath.\, M_\imath \vdash S_\imath; T_\imath \Rightarrow \neg \mathsf{deadlocked}(T_n).$$

D.5 Proof of soundness

Theorem D.1 (Type Safety) If the initial configuration $S_0; T_0$ is well-typed (cf. page 54) with $\emptyset \vdash S_0; T_0$ and the operational semantics takes any number of steps $S_0; T_0 \rightsquigarrow^n S_n; T_n$, then the resulting configuration $S_n; T_n$ is not stuck and T_n has not reached a deadlocked state.

Proof. The application of lemma D.1 to the assumption implies that $\forall \imath \in [0,n]. \exists M_\imath. M_\imath \vdash S_\imath; T_\imath$. Therefore, $S_n; T_n$ is well-typed for some M_n. The application of lemma D.19 to $M_n \vdash S_n; T_n$ implies $S_n; T_n$ is not stuck. The application of lemma D.2 to $\forall \imath \in [0,n]. \exists M_\imath. M_\imath \vdash S_\imath; T_\imath$ and $\emptyset; 0 : \emptyset; e \rightsquigarrow^n S_n; T_n$ implies that T_n has not reached a deadlocked state.

Lemma D.1 (Multi-step Program Preservation) Let $S_0; T_0$ be a *closed well-typed configuration* such that $M_0 \vdash S_0; T_0$ for some M_0. If the operational semantics evaluates $S_0; T_0$ to $S_n; T_n$ in n steps, then $\forall \imath \in [0,n]. \exists M_\imath.\, M_\imath \vdash S_\imath; T_\imath$

Proof. Proof by induction on the number of steps n. When no steps are performed (i.e., $n = 0$) the proof is immediate from the assumption. When some steps are performed (i.e., $n > 0$), we have that $S_0; T_0 \rightsquigarrow^n S_n; T_n$ or $S_0; T_0 \rightsquigarrow^{n-1} S_{n-1}; T_{n-1}$ and $S_{n-1}; T_{n-1} \rightsquigarrow S_n; T_n$. The application of the induction hypothesis to the fact that $S_0; T_0$ is well-typed implies $\forall \imath \in [0, n-1]. \exists M_\imath. M_\imath \vdash S_\imath; T_\imath$. Thus, $M_{n-1} \vdash S_{n-1}; T_{n-1}$ holds. The application of lemma D.3 to $M_{n-1} \vdash S_{n-1}; T_{n-1}$ and $S_{n-1}; T_{n-1} \rightsquigarrow S_n; T_n$. implies that $M_n \vdash S_n; T_n$. Therefore, $\forall \imath \in [0,n]. \exists M_\imath. M_\imath \vdash S_\imath; T_\imath$.

Lemma D.2 (Deadlock Freedom) Let the initial configuration take n steps, where each step is well-typed for some M, then the resulting configuration has not reached a deadlocked state.

Proof. The assumptions imply that $\emptyset; 0 : e \rightsquigarrow^n S_n; T_n$ and $\forall \imath \in [0,n]. \exists M_\imath. M_\imath \vdash S_\imath; T_\imath$. Assume that $\mathsf{deadlocked}(T_x)$ holds for some $x \in [0,n]$ and the first deadlock occurring in the program is in T_x (i.e. $\forall \imath. \imath < x \Rightarrow \neg \mathsf{deadlocked}(T_\imath)$). Then, the following hold:

- $T_x = T, n_0 : E_0[\mathtt{lock}_{\gamma_0}\ \mathtt{lk}_{\imath_0}], \ldots n_z : E_z[\mathtt{lock}_{\gamma_z}\ \mathtt{lk}_{\imath_z}]$, where threads 0 to z are in a deadlocked state.
- $z > 0$ and $\forall m_1 \in [0,z].\ \mathsf{locks}(S, \imath_{\mathsf{succ}(m_1)}, m_1) > 0$, where $succ(n) = (n+1)\ mod (z+1)$.

Let m be the thread that acquires the *first* of the $z+1$ locks that cause the deadlock, namely $\imath_{\mathsf{succ}(m)}$ (given the definition of T_x). Then thread m acquired lock $\imath_k$, where k equals $\mathsf{succ}(m)$, before thread k acquired lock $\imath_{\mathsf{succ}(\kappa)}$. Let us assume that $\epsilon_{1y} = \mathsf{run}(\mathsf{stack}(E[\mathsf{pop}_{\gamma_y}\ \square]), \imath_{\mathsf{succ}(k)}\ , 1)$ and $\epsilon_{2y} = \mathsf{dom}(S_y)$, where $y < x$ such that $\imath_{\mathsf{succ}(k)}$ is acquired for the *first* time by thread k. Then, $\imath_k$ does not belong to ϵ_{1y}, otherwise thread k would have been blocked at the lock request of $\imath_{\mathsf{succ}(k)}$ as $\imath_k$ is already owned by thread m.

According to the assumption, each step is well-typed so $S_x; T_x$ is well-typed. By inversion of the typing configuration of thread $n_k : E_k[\mathtt{lock}_{\gamma'_k}\ \mathtt{lk}_{\imath_k}]$ we obtain that $n_k; \gamma_k \vdash S_k$, where γ_k is the effect assigned to expression $E_k[\mathtt{lock}_{\gamma'_k}\ \mathtt{lk}_{\imath_k}]$ by thread typing. We have that $\imath_{\mathsf{succ}(k)}$ is locked by thread k so by inversion of $n_k; \gamma_k \vdash S_k$ we have that:

- $S_k(\imath_{\mathsf{succ}(k)}) = n_k; n_1; \epsilon_1; \epsilon_2$, where n_1 is positive.
- $\imath_{\mathsf{succ}(k)}; n_1 \vdash_{ok} \gamma_k$
- $\epsilon_3 = \mathsf{run}(\gamma_k, \imath_{\mathsf{succ}(k)}, n_1)$

- $\epsilon_1 \cap \epsilon_3 \subseteq \epsilon_2$, where $\epsilon_2 = \epsilon_{1y}$ and $\epsilon_1 = \epsilon_{2y}$ (this is immediate by the operational steps from step y to x).

Thus, it suffices to prove that $\imath_k \in \epsilon_1$ and $\imath_k \in \epsilon_3$. For all evaluation steps f and g such that f less or than equal to g, $\mathsf{dom}(S_f) \subseteq \mathsf{dom}(S_g)$ holds (trivial to show by observation of the evaluation relation). We have assumed that m is the first thread to lock $\imath_k$ at some step y' (so $\imath_k \in \mathsf{dom}(S_{y'})$) prior to y so $\imath_k \in \epsilon_1$ (so $\imath_k \in \mathsf{dom}(S_{y'}) \subseteq (\mathsf{dom}(S_y) = \epsilon_{2y} = \epsilon_1)$.

The application of lemma D.16 to the typing derivation of $E_k[\mathtt{lock}_{\gamma'_k}\ \mathtt{lk}_{\imath_k}]$ implies that $\mathtt{lock}_{\gamma'_k}\ \mathtt{lk}_{\imath_k}$ is well-typed with effect $(\gamma'_k; {\imath_k}^+, \gamma'_k)$ and that E_k is well-typed. Thus, $M_k; \emptyset; \emptyset \vdash E_k : \langle\rangle \xrightarrow{\gamma'_k;{\imath_k}^+,\gamma'_k} \langle\rangle$ & $(\emptyset; \gamma_k)$. Lemma D.9 implies that $\gamma_k = {\imath_k}^+, \gamma''_k$ for some γ''_k. Thus, $\epsilon_3 = \mathsf{run}({\imath_k}^+, \gamma''_k, \imath_{\mathsf{succ}(k)}, n_1) = \mathsf{run}(\gamma''_k, \imath_{\mathsf{succ}(k)}, n_1) \cup \{\imath_k\}$ (by the definition of function run). Therefore $\imath_k \in \epsilon_{1y}$, which is a contradiction.

Lemma D.3 (Preservation) Let $S;T$ be a well-typed configuration with $M \vdash S;T$. If the operational semantics takes a step $S;T \rightsquigarrow S';T'$, then there exist an $M' \supseteq M$ such that the resulting configuration is well-typed with $M' \vdash S';T'$.

Proof. By case analysis on the thread evaluation relation:

Case *E-T*: rule *E-T* implies that $S;T,n:\Box[()] \rightsquigarrow S;T$. By inversion of the configuration typing assumption we have that:

- $S;M \vdash T, n:\Box[()]$: by inversion of this derivation we have that:
  - $M;\emptyset;\emptyset \vdash \Box[()] : \langle\rangle$ & $(\emptyset;\emptyset)$
  - $n \notin \mathsf{dom}(T)$
  - $S;M \vdash T$
  - $n;\emptyset \vdash S$
- $M = \mathsf{dom}(S)$

Given the above facts, $M \vdash S;T$ holds.

Case *E-A*: rule *E-A* implies that $S;T,n:E[(v'\ v)^{\mathsf{seq}(\gamma_a)}] \rightsquigarrow S;T,n:E[\mathsf{pop}_{\gamma_a}\ e_1[v/x]]$, where v' is equal to $\lambda x.\, e_1$. By inversion of the configuration typing assumption we have that:

- $M = \mathsf{dom}(S)$
- $S;M \vdash T, n:E[e]$, where e is equal to $(v'\ v)^{\mathsf{seq}(\gamma_a)}$: by inversion of this derivation we have that:
  - $S;M \vdash T$
  - $n \notin \mathsf{dom}(T)$
  - $n;\gamma \vdash S$
  - $M;\emptyset;\emptyset \vdash E[e] : \langle\rangle$ & $(\emptyset;\gamma)$: lemma D.16 implies that $M;\emptyset;\emptyset \vdash E : \tau'_2 \xrightarrow{\gamma_a;\gamma_b} \langle\rangle$ & $(\emptyset;\gamma)$ and $M;\emptyset;\emptyset \vdash e : \tau'_2$ & $(\gamma_a;\gamma_b)$. By inversion of the latter derivation we have that $M;\emptyset;\emptyset \vdash v : \tau'_1$ & $(\gamma_b;\gamma_b)$, and $M;\emptyset;\emptyset \vdash \lambda x.\, e_1 : \tau'_1 \xrightarrow{\gamma'_c} \tau'_2$ & $(\gamma_b;\gamma_b)$, where $\gamma_b = \gamma'_c :: \gamma_a$. By inversion of the typing derivation of v' we obtain that $M;\emptyset;\emptyset, x:\tau'_1 \vdash e_1 : \tau'_2$ & $(\emptyset;\gamma'_c)$. Lemma D.11 implies that $M;\emptyset;\emptyset \vdash e_1[v/x] : \tau'_2$ & $(\emptyset;\gamma'_c)$ holds. The application of rule *T-PP* implies that $M;\emptyset;\emptyset \vdash \mathsf{pop}_{\gamma_a}\ e_1[v/x] : \tau'_2$ & $(\gamma_a;\gamma_b)$ holds. The application of lemma D.15 implies that $M;\emptyset;\emptyset \vdash E[\mathsf{pop}_{\gamma_a}\ e_1[v/x]] : \langle\rangle$ & $(\emptyset;\gamma)$.

Case *E-SN*: Rule *E-SN* implies that $S;T,n:E[(v'\ v)^{\mathsf{par}}] \rightsquigarrow S;T,n:E[()], n':\Box[(v'\ v)^{\mathsf{seq}(\emptyset)}]$ By inversion of the configuration typing assumption we have that:

- $M = \mathsf{dom}(S)$

- $S; M \vdash T, n : E[e]$: by inversion of this derivation we have that:
  - $S; M \vdash T$
  - $n; \gamma \vdash S$
  - $n \notin \mathsf{dom}(T)$
  - $M; \emptyset; \emptyset \vdash E[(v'\ v)^{\mathsf{par}}] : \langle\rangle \,\&\, (\emptyset;\gamma)$: lemma D.16 implies that $M; \emptyset; \emptyset \vdash E : \langle\rangle \xrightarrow{\gamma_a;\gamma_a} \langle\rangle \,\&\, (\emptyset;\gamma)$ and $M; \emptyset; \emptyset \vdash (v'\ v)^{\mathsf{par}} : \langle\rangle \,\&\, (\gamma_a;\gamma_a)$. By inversion of the latter derivation we have that $M; \emptyset; \emptyset \vdash v : \tau_1' \,\&\, (\gamma_a;\gamma_a)$, $M; \emptyset; \emptyset \vdash v' : \tau_1' \xrightarrow{\gamma_c'} \langle\rangle \,\&\, (\gamma_a;\gamma_a)$ and $\forall r \in \mathsf{dom}(\gamma_c').\ r; 0 \vdash_{ok} \gamma_c'$. The application of lemma D.7 to the typing derivation of v' implies that $M; \emptyset \vdash \gamma_c'$. Lemma D.8 implies that $M; \emptyset; \emptyset \vdash v : \tau_1' \,\&\, (\gamma_c';\gamma_c')$ and $M; \emptyset; \emptyset \vdash v' : \tau_1' \xrightarrow{\gamma_c'} \langle\rangle \,\&\, (\gamma_c';\gamma_c')$. Rule *T-AP* implies $M; \emptyset; \emptyset \vdash (v'\ v)^{\mathsf{seq}(\emptyset)} : \langle\rangle \,\&\, (\emptyset;\gamma_c')$. Therefore, lemma D.15 implies that $M; \emptyset; \emptyset \vdash \Box(v'\ v)^{\mathsf{seq}(\emptyset)} : \langle\rangle \,\&\, (\emptyset;\gamma_c')$. The application of lemma D.6 to the typing derivation of v' implies $M; \emptyset \vdash \gamma_a$. Rule *T-U* yields that $M; \emptyset; \emptyset \vdash () : \langle\rangle \,\&\, (\gamma_a;\gamma_a)$. The application of lemma D.15 implies that $M; \emptyset; \emptyset \vdash E[()] : \langle\rangle \,\&\, (\emptyset;\gamma)$.
  - $n'; \gamma_c' \vdash S$: this is immediate from $\forall r \in \mathsf{dom}(\gamma_c').\ r; 0 \vdash_{ok} \gamma_c'$ and the fact that $\mathsf{locks}(S, \imath, n') = 0$ for all $\imath$.

Case *E-IT*: rule *E-IT* implies that $S; T, n : E[\texttt{if true then } e_1 \texttt{ else } e_2] \rightsquigarrow S; T, n : E[e_1]$. By inversion of the configuration typing assumption we have that:

- $M = \mathsf{dom}(S)$
- $S; M \vdash T, n : E[e]$, where e is equal to $\texttt{if true then } e_1 \texttt{ else } e_2$: by inversion of this derivation we have that:
  - $S; M \vdash T$
  - $n \notin \mathsf{dom}(T)$
  - $M; \emptyset; \emptyset \vdash E[e] : \langle\rangle \,\&\, (\emptyset;\gamma)$: lemma D.16 implies that $M; \emptyset; \emptyset \vdash E : \tau_2' \xrightarrow{\gamma_a;\gamma_b} \langle\rangle \,\&\, (\emptyset;\gamma)$ and $M; \emptyset; \emptyset \vdash e : \tau_2' \,\&\, (\gamma_a;\gamma_b)$. By inversion of the latter derivation we have that $M; \emptyset; \emptyset \vdash e_1 : \tau_2' \,\&\, (\gamma_a;\gamma_{b_1} :: \gamma_a)$, $M; \emptyset; \emptyset \vdash e_2 : \tau_2' \,\&\, (\gamma_a;\gamma_{b_2} :: \gamma_a)$ and $\gamma_b = \gamma_{b_1} \,?\, \gamma_{b_2}, \gamma_a$. Lemma D.9 implies that $M; \emptyset; \emptyset \vdash E : \tau_2' \xrightarrow{\gamma_a;\gamma_a} \langle\rangle \,\&\, (\emptyset;\gamma')$ and $M; \emptyset \vdash \gamma$, where $\gamma = \gamma_{b_1} \,?\, \gamma_{b_2}, \gamma'$. Thus, $M; \emptyset \vdash \gamma_{b_1} :: \gamma'$ holds. The application of lemma D.9 to the latter fact, $M; \emptyset; \emptyset \vdash E : \tau_2' \xrightarrow{\gamma_a;\gamma_a} \langle\rangle \,\&\, (\emptyset;\gamma')$ implies that $M; \emptyset; \emptyset \vdash E : \tau_2' \xrightarrow{\gamma_a;\gamma_{b_1} :: \gamma_a} \langle\rangle \,\&\, (\emptyset, \gamma_{b_1} :: \gamma')$. Lemma D.15 implies that $M; \emptyset; \emptyset \vdash E[e_1] : \langle\rangle \,\&\, (\emptyset;\gamma_{b_1} :: \gamma')$.
  - $n; \gamma \vdash S$: Let us assume that $S = S', \imath_1 \mapsto n_a; n_b; \epsilon_a; \epsilon_b$, for any $\imath_1$ in $\mathsf{dom}(S)$. Given $n; \gamma \vdash \imath_1 \mapsto n_a; n_b; \epsilon_a; \epsilon_b$, where $\gamma = \gamma_{b_1} \,?\, \gamma_{b_2}, \gamma'$, it suffices to prove $n; \gamma_{b_1} :: \gamma' \vdash \imath_1 \mapsto n_a; n_b; \epsilon_a; \epsilon_b$. If $n_a \neq n$, then it suffices to prove $\imath_1; 0 \vdash_{ok} \gamma_{b_1} :: \gamma'$, which immediate by $\imath_1; 0 \vdash_{ok} \gamma_{b_1} \,?\, \gamma_{b_2}, \gamma'$ (by inversion of $n; \gamma \vdash \imath_1 \mapsto n_a; n_b; \epsilon_a; \epsilon_b$). If $n_a \neq n$, then it suffices to prove $\imath_1; n_a \vdash_{ok} \gamma_{b_1} :: \gamma'$ and $\mathsf{run}((\gamma_{b_1} :: \gamma'), \imath_1, n_a) \cap \epsilon_a \subseteq \epsilon_b$. The proof for the former is identical to the proof where $n_a \neq n$. By inversion of $n; \gamma \vdash \imath_1 \mapsto n_a; n_b; \epsilon_a; \epsilon_b$ we obtain that $\mathsf{run}(\gamma, \imath_1, n_a) \cap \epsilon_a \subseteq \epsilon_b$. Thus, it suffices to show that $\mathsf{run}(\gamma, \imath_1, n_a) \supseteq \mathsf{run}((\gamma_{b_1} :: \gamma'), \imath_1, n_a)$, which is immediate by the definition of run.

Case *E-IF*: similar to the previous case.

Case *E-FX*: rule *E-FX* implies $S; T, n : E[(\texttt{fix } x.\ f\ v)^{\mathsf{seq}(\gamma_a)}] \rightsquigarrow S; T, n : E[(f'\ v)^{\mathsf{seq}(\gamma_a)}]$ holds, where $f' = f[\texttt{fix } x.\ f/x]$. By inversion of the configuration typing assumption we have that:

- $M = \mathsf{dom}(S)$

- $S; M \vdash T, n : E[e]$, where e is equal to $(\texttt{fix}\ x.\ f\ \ v)^{\mathsf{seq}(\gamma_a)}$: by inversion of this derivation we have that:
  - $S; M \vdash T$
  - $n \notin \mathsf{dom}(T)$
  - $M; \emptyset; \emptyset \vdash E[e] : \langle\rangle \,\&\, (\emptyset;\gamma)$: lemma D.16 implies that $M; \emptyset; \emptyset \vdash E : \tau_2' \xrightarrow{\gamma_a;\gamma_b} \langle\rangle \,\&\, (\emptyset;\gamma)$ and $M; \emptyset; \emptyset \vdash e : \tau_2' \,\&\, (\gamma_a;\gamma_b)$. By inversion of the latter derivation we have that $M; \emptyset; \emptyset \vdash v : \tau_1' \,\&\, (\gamma_b;\gamma_b)$, and $M; \emptyset; \emptyset \vdash \texttt{fix}\ x.\ f : \tau_1' \xrightarrow{\gamma_c'} \tau_2' \,\&\, (\gamma_b;\gamma_b)$, where $\gamma_b = \gamma_c' :: \gamma_a$. By inversion of the typing derivation of $\texttt{fix}\ x.\ f$ we obtain that $M; \emptyset; \emptyset, x : \tau_1' \xrightarrow{\gamma_c'} \tau_2' \vdash f : \tau_1' \xrightarrow{\gamma_c''} \tau_2' \,\&\, (\gamma_b;\gamma_b)$, $\mathsf{summary}(\gamma_c'') = \gamma_c' = \gamma_{x1} :: \gamma_y :: \gamma_{x2}$ such that $\mathsf{rsummary}(\gamma_c'') = \gamma_{x1}; \gamma_y; \gamma_{x2}$. Lemma D.11 implies that $M; \emptyset; \emptyset \vdash f[\texttt{fix}\ x.\ f/x] : \tau_1' \xrightarrow{\gamma_c''} \tau_2' \,\&\, (\gamma_b;\gamma_b)$ holds. Lemma D.9 implies that $M; \emptyset; \emptyset \vdash E : \tau_2' \xrightarrow{\gamma_a;\gamma_a} \langle\rangle \,\&\, (\emptyset;\gamma')$ and $M; \emptyset \vdash \gamma$, where $\gamma = \gamma_c' :: \gamma'$. Thus, $M; \emptyset \vdash \gamma_c'' :: \gamma'$ holds ($M; \emptyset \vdash \gamma_c''$ holds by the application of lemma D.7 to the typing derivation $f[\texttt{fix}\ x.\ f/x]$). The application of lemma D.9 to the latter fact, $M; \emptyset; \emptyset \vdash E : \tau_2' \xrightarrow{\gamma_a;\gamma_a} \langle\rangle \,\&\, (\emptyset;\gamma')$ implies that $M; \emptyset; \emptyset \vdash E : \tau_2' \xrightarrow{\gamma_a;\gamma_c'' :: \gamma_a} \langle\rangle \,\&\, (\emptyset;\gamma_c'' :: \gamma')$. Lemma D.8 implies that $M; \emptyset; \emptyset \vdash f[\texttt{fix}\ x.\ f/x] : \tau_1' \xrightarrow{\gamma_c''} \tau_2' \,\&\, (\gamma_c'' :: \gamma_a;\gamma_c'' :: \gamma_a)$ and $M; \emptyset; \emptyset \vdash v : \tau_1' \,\&\, (\gamma_c'' :: \gamma_a;\gamma_c'' :: \gamma_a)$. Therefore, $M; \emptyset; \emptyset \vdash (f[\texttt{fix}\ x.\ f/x]\ \ v)^{\mathsf{seq}(\gamma_a)} : \tau_2' \,\&\, (\gamma_a;\gamma_c'' :: \gamma_a)$. The application of lemma D.15 implies that $M; \emptyset; \emptyset \vdash E[(f[\texttt{fix}\ x.\ f/x]\ \ v)^{\mathsf{seq}(\gamma_a)}] : \langle\rangle \,\&\, (\emptyset;\gamma_c'' :: \gamma')$.
  - $n; \gamma \vdash S$: $n; \gamma_c'' :: \gamma' \vdash S$ is immediate by lemma D.4.

Case *E-RP* and *E-PP*: these rules are side-effect free and therefore we provide a single proof for all cases. Hence, we are assuming here that u (i.e. in $E[u]$) has one of the following forms: $(\Lambda\rho.\ f)\ [\imath]$ or $\mathsf{pop}_\gamma\ v$. Rules E-RP and *E-PP* imply that $S' = S$, $T' = T, n : E[v]$, where v is the value that replaces u in context E. By inversion of the configuration typing assumption we have that:

- $M = \mathsf{dom}(S)$
- $S; M \vdash T, n : E[u]$: by inversion of this derivation we have that:
  - $S; M \vdash T$
  - $n \notin \mathsf{dom}(T)$
  - $M; \emptyset; \emptyset \vdash E[u] : \langle\rangle \,\&\, (\emptyset;\gamma)$: in the case of rule *E-PP*, v is well-typed by inversion of the typing derivation of u. We need to apply lemma D.8 so as to change the effect of v from $\emptyset$ to γ. In the case of rule *E-RP*, v is obtained by substituting $\imath$ in the body of function f (i.e. the initial term is $(\Lambda\rho.\ f)\ [\imath]$). This is immediate by lemma D.12.
  - $n; \gamma \vdash S$

Case *E-NG*: rule *E-NG* implies that $S; T, n : E[\texttt{newlock}\ \rho, x\ \texttt{in}\ e_1] \rightsquigarrow S, \imath \mapsto n; 0; \emptyset; \emptyset; T, n : E[e_1[\imath/\rho]\ [\texttt{lk}_\imath/x]]$. By inversion of the configuration typing assumption we have that:

- $M = \mathsf{dom}(S)$: $M, \imath = \mathsf{dom}(S, \imath \mapsto n; 0; \emptyset; \emptyset)$ is immediate.
- $S; M \vdash T, n : E[\texttt{newlock}\ \rho, x\ \texttt{in}\ e_1]$: by inversion of this derivation we have that:
  - $S; M \vdash T$: $S, \imath \mapsto n; 0; \emptyset; \emptyset; M, \imath \vdash T$ trivially holds by using lemma D.13 to obtain that threads of T are well-typed in the extended context $M, \imath$; $\imath$ is *fresh* (it does not exist in the effects or stack of other threads) so the invariant $n; \gamma_{n'} \vdash S, \imath \mapsto n; 0; \emptyset; \emptyset$ trivially holds from $n; \gamma_{n'} \vdash S$, where $\gamma_{n'}$ is the effect of thread n' (other than n).
  - $n \notin \mathsf{dom}(T)$

– $M;\emptyset;\emptyset \vdash E[\texttt{newlock}\ \rho, x\ \texttt{in}\ e_1] : \langle\rangle \,\&\, (\emptyset;\gamma)$: lemma D.16 implies that $M;\emptyset;\emptyset \vdash E : \tau \xrightarrow{\gamma_a;\gamma_b} \langle\rangle \,\&\, (\emptyset;\gamma)$ and $M;\emptyset;\emptyset \vdash \texttt{newlock}\ \rho, x\ \texttt{in}\ e_1 : \tau \,\&\, (\gamma_a;\gamma_b)$. By inversion of the latter derivation we obtain that $M;\Delta \vdash \tau,\ \rho;0 \vdash_{ok}\ \gamma_c,\ M;\emptyset,\rho;\emptyset,x:\texttt{Lk}(\rho) \vdash e_1 : \tau \,\&\, (\gamma_a;\gamma_c)$ and $\gamma_b = \gamma_c \setminus \rho = (\gamma_e \setminus \rho) :: \gamma_a$ for some γ_e. Lemma D.13 implies that $M,\imath;\emptyset,\rho;\emptyset,x:\texttt{Lk}(\rho) \vdash e_1 : \tau \,\&\, (\gamma_a;\gamma_c)$ holds. Lemma D.12 implies that $M,\imath;\emptyset;\emptyset,x:\texttt{Lk}(\imath) \vdash e_1[\imath/\rho] : \tau[\imath/\rho] \,\&\, (\gamma_a[\imath/\rho];\gamma_c[\imath/\rho])$ holds. γ_a and τ do not any contain occurrences of ρ so the above derivation can be further simplified to $M,\imath;\emptyset;\emptyset,x:\texttt{Lk}(\imath) \vdash e_1[\imath/\rho] : \tau \,\&\, (\gamma_a;\gamma_c[\imath/\rho])$ holds. Lemma D.11 and $M,\imath;\emptyset;\emptyset \vdash \texttt{lk}_\imath : \texttt{Lk}(\imath) \,\&\, (\emptyset;\emptyset)$ imply that $M,\imath;\emptyset;\emptyset \vdash e_1[\imath/\rho][\texttt{lk}_\imath/x] : \tau \,\&\, (\gamma_a;\gamma_c[\imath/\rho])$ holds. Lemma D.9 implies that $M;\emptyset;\emptyset \vdash E : \tau \xrightarrow{\gamma_a;\gamma_a} \langle\rangle \,\&\, (\emptyset;\gamma')$ and $M;\emptyset \vdash \gamma$, where $\gamma = (\gamma_e \setminus \rho) :: \gamma'$. The application of lemma D.6 to the typing derivation of $e_1[\imath/\rho][\texttt{lk}_\imath/x]$ implies that $M;\emptyset \vdash \gamma_c[\imath/\rho]$. Lemma D.14 implies that $M,\imath;\emptyset;\emptyset \vdash E : \tau \xrightarrow{\gamma_a;\gamma_a} \langle\rangle \,\&\, (\emptyset;\gamma')$. The application of lemma D.9 to the latter facts imply that $M,\imath;\emptyset;\emptyset \vdash E : \tau \xrightarrow{\gamma_a;\gamma_c[\imath/\rho]} \langle\rangle \,\&\, (\emptyset,\gamma'')$, where $\gamma'' = \gamma_e[\imath/\rho] :: \gamma'$. Lemma D.15 implies that $M,\imath;\emptyset;\emptyset \vdash E[e_1[\imath/\rho][\texttt{lk}_\imath/\rho\,]] : \langle\rangle \,\&\, (\emptyset;\gamma_e[\imath/\rho] :: \gamma')$.

– $n;\gamma \vdash S$: we need to prove that $n;\gamma_e[\imath/\rho] :: \gamma' \vdash S, \imath \mapsto n;0;\emptyset;\emptyset$. It suffices to show that $\imath;0 \vdash_{ok}\ \gamma_e[\imath/\rho] :: \gamma'$, $\mathsf{run}(\gamma_e[\imath/\rho] :: \gamma', \imath, 0)$ is defined, $\mathsf{run}(\gamma_e[\imath/\rho] :: \gamma', \imath, 0) \cap \emptyset \subseteq \emptyset$ (immediate), and $n;\gamma_e[\imath/\rho] :: \gamma' \vdash S$. $\imath;0 \vdash_{ok}\ \gamma_e[\imath/\rho] :: \gamma'$ holds as a consequence of the following facts:

  * $\rho;0 \vdash_{ok}\ \gamma_e :: \gamma_a$ holds by the typing rule *T-NG*
  * ρ does not occur in γ_a nor γ'
  * $\imath$ does not occur anywhere
  * thus $\imath;0 \vdash_{ok}\ \gamma_e[\imath/\rho] :: \gamma'$ holds.

  Now, $\mathsf{run}(\gamma_e[\imath/\rho] :: \gamma', \imath, 0)$ is defined as $\imath;0 \vdash_{ok}\ \gamma_e[\imath/\rho] :: \gamma'$ holds (by simply observing that ok is defined then so is run). Finally, $n;\gamma_e[\imath/\rho] :: \gamma' \vdash S$ holds as a consequence of the following facts:

  * for all $\texttt{j} \neq \imath\ \ \gamma_e[\imath/\rho]$ contains that same order of $+$ and $-$ operations as $\gamma_e \setminus \rho$,
  * $n;(\gamma_e \setminus \rho) :: \gamma' \vdash S$ holds (by inversion of $n;\gamma \vdash S$) and
  * for all $\texttt{j}$ such that $S(\texttt{j}) = (n_x; n_y; \epsilon_x; \epsilon_y)$, $\imath \notin \epsilon_x$ as $\imath$ is *fresh*.

Case *E-UL*: this rule creates side-effects as it modifies the count of lock $\imath$. rule *E-UL* implies that $T' = T, n : E[()]$, where $()$ replaces u ($u = \texttt{unlock}\ \texttt{lk}_\imath$) in context E. The rule also implies that $S(\imath) = (n; n_2; \epsilon_1; \epsilon_2)$, $n_2 > 0$ and $S' = S[\imath \mapsto n; n_2 - 1; \epsilon_1; \epsilon_2]$. By inversion of the configuration typing assumption we have that:

- $M = \mathsf{dom}(S)$: $\imath$ is already contained in S so $M = \mathsf{dom}(S')$ trivially holds.
- $S;M \vdash T, n : E[u]$: by inversion of this derivation we have that:
  – $S;M \vdash T$: we must prove that $S';M \vdash T$. It suffices to prove $n';\gamma_{n'} \vdash S'$ given that $n';\gamma_{n'} \vdash S$ holds, where $\gamma_{n'}$ is the effect of thread n'. This is immediate for all locks $\texttt{j}$ other than $\imath$ as they remain unchanged. The invariant holds for the updated $\imath$ as *only* the reference count of lock $\imath$ is modified and therefore $\mathsf{locks}(S', \imath, n') = 0$ for all $n' \neq n$.
  – $n \notin \mathsf{dom}(T)$
  – $M;\emptyset;\emptyset \vdash E[u] : \langle\rangle \,\&\, (\emptyset;\gamma)$: lemma D.16 implies that $M;\emptyset;\emptyset \vdash E : \langle\rangle \xrightarrow{\gamma_a;\imath^-;\gamma_a} \langle\rangle \,\&\, (\emptyset;\gamma)$ and $M;\emptyset;\emptyset \vdash u : \langle\rangle \,\&\, (\gamma_a;\imath,^-\gamma_a)$. Lemma D.9 implies that $M;\emptyset;\emptyset \vdash E : \langle\rangle \xrightarrow{\gamma_a;\gamma_a} \langle\rangle \,\&\, (\emptyset;\gamma')$ and $\gamma = \imath^-, \gamma'$. The application of lemma D.7 to the typing derivation of u implies that $M;\emptyset \vdash \gamma_a$. Thus, rule *T-U* implies $M;\emptyset;\emptyset \vdash () : \langle\rangle \,\&\, (\gamma_a;\gamma_a)$. $M;\emptyset;\emptyset \vdash E[()] : \langle\rangle \,\&\, (\emptyset;\gamma')$.

– $n; \imath^-, \gamma' \vdash S$: $S = S'', \imath \mapsto n; n_2; \epsilon_1; \epsilon_2$, where n_2 is positive, and $S' = S'', \imath \mapsto n; n_2 - 1; \epsilon_1; \epsilon_2$. The thread identifier of $\imath$ is unchanged in S' so it suffices to prove the following:

  * $\imath; n_2 - 1 \vdash_{ok} \gamma'$: by inversion of $n; \imath^-, \gamma' \vdash S$ we obtain $\imath; n_2 \vdash_{ok} \imath^-, \gamma'$. By inversion (rule *OK2*) of the latter fact we have that $\imath; n_2 - 1 \vdash_{ok} \gamma'$.
  * $\epsilon_3 = \mathsf{run}(\gamma', \imath, n_2 - 1)$ is defined: by inversion of $n; \imath^-, \gamma' \vdash S$ we obtain $\mathsf{run}((\imath^-, \gamma'), \imath, n_2)$. By unfolding the definition of run $\mathsf{run}((\imath^-, \gamma'), \imath, n_2)$ becomes $\mathsf{run}(\imath, \gamma', n_2 - 1)$.
  * $\epsilon_1 \cap \epsilon_3 \subseteq \epsilon_2$: trivially holds from the above.
  * $n; \gamma' \vdash S''$: we have that $n; \imath^-, \gamma' \vdash S''$ by inversion of $n; \imath^-, \gamma' \vdash S$. $n; \imath^-, \gamma' \vdash S''$ implies that for all $\mathtt{j}$ in $\mathsf{dom}(S'')$ such that $S''(\mathtt{j}) = (n_{1\mathtt{j}}; n_{2\mathtt{j}}; \epsilon_{1\mathtt{j}}; \epsilon_{2\mathtt{j}})$ the following hold: $\mathtt{j}; n_{2\mathtt{j}} \vdash_{ok} \imath^-, \gamma'$ and $\mathsf{run}((\imath^-, \gamma'), \mathtt{j}, , n_{2\mathtt{j}}) \cap \epsilon_{1\mathtt{j}} \subseteq \epsilon_{2\mathtt{j}}$. By inversion of $\mathtt{j}; n_{2\mathtt{j}} \vdash_{ok} \imath^-, \gamma'$ (rule *OK3*) we have that $\mathtt{j}; n_{2\mathtt{j}} \vdash_{ok} \gamma'$. If we unfold $\mathsf{run}((\imath^-, \gamma'), \mathtt{j}, n_{2\mathtt{j}})$ once then we obtain $\mathsf{run}((\imath^-, \gamma'), \mathtt{j}, n_{2\mathtt{j}})$ is equal to $\mathsf{run}((\gamma'), \mathtt{j}, n_{2\mathtt{j}})$.

Case *E-LK1*: the proof is identical to the previous case. In the case of proving $n; \gamma' \vdash S''$ is more interesting: $\mathsf{run}((\imath^+, \gamma'), \mathtt{j}, n_{2\mathtt{j}})$ is equal to $\mathsf{run}((\gamma'), \mathtt{j}, n_{2\mathtt{j}}) \cup \{\imath\}$. Thus $\mathsf{run}((\gamma'), \mathtt{j}, n_{2\mathtt{j}})$ is a subset of $\mathsf{run}((\imath^+, \gamma'), \mathtt{j}, n_{2\mathtt{j}})$ and therefore, $\mathsf{run}((\gamma'), \mathtt{j}, n_{2\mathtt{j}}) \cap \epsilon_{1\mathtt{j}}$ is a subset of $\epsilon_{2\mathtt{j}}$ holds.

Case *E-LK0*: rule *E-LK0* implies that $T' = T, n : E[()]$, where $()$ replaces u ($u = \mathtt{lock}_{\gamma_a}\ \mathtt{lk}_\imath$) in context E. It also implies that $\epsilon = \mathsf{run}(\mathsf{stack}(E[\mathsf{pop}_{\gamma_a}\ \square]), \imath, 1)$, $\epsilon \cup \{\imath\} \subseteq \mathsf{available}(S, n)$, $S(\imath) = (n_a; 0; \epsilon_a; \epsilon_b)$ and $S = S[\imath \mapsto n; 1; \mathsf{dom}(S); \epsilon]$. By inversion of the configuration typing assumption we have that:

- $M = \mathsf{dom}(S)$: in both cases $\imath$ is already contained in S so $M = \mathsf{dom}(S')$ trivially holds.
- $S; M \vdash T, n : E[u]$: by inversion of this derivation we have that:
  – $S; M \vdash T$: we must prove that $S'; M \vdash T$. It suffices to prove $n'; \gamma_{n'} \vdash S'$ given that $n'; \gamma_{n'} \vdash S$ holds, where $\gamma_{n'}$ is the effect of thread n'. This is immediate for all locks $\mathtt{j}$ other than $\imath$ as S' differs from S in respect to lock $\imath$. It also holds for $\imath$ as $\mathsf{locks}(S', \imath, n') = 0$ for all $n' \neq n$.
  – $n \notin \mathsf{dom}(T)$
  – $M; \emptyset; \emptyset \vdash E[u] : \langle\rangle$ & $(\emptyset;\gamma)$: lemma D.16 implies that $M; \emptyset; \emptyset \vdash E : \langle\rangle \xrightarrow{\gamma_a;\imath^+;\gamma_a} \langle\rangle$ & $(\emptyset;\gamma)$ and $M; \emptyset; \emptyset \vdash u : \langle\rangle$ & $(\gamma_a;\imath^+, \gamma_a)$. Lemma D.9 implies that $M; \emptyset; \emptyset \vdash E : \langle\rangle \xrightarrow{\gamma_a;\gamma_a} \langle\rangle$ & $(\emptyset;\gamma')$ and $\gamma = \imath^+, \gamma'$. The application of lemma D.7 to the typing derivation of u implies that $M; \emptyset \vdash \gamma_a$. Thus, rule *T-U* implies $M; \emptyset; \emptyset \vdash () : \langle\rangle$ & $(\gamma_a;\gamma_a)$. $M; \emptyset; \emptyset \vdash E[()] : \langle\rangle$ & $(\emptyset;\gamma')$.
  – $n; \gamma \vdash S$: $S = S'', \imath \mapsto n_1; 0; \epsilon_a; \epsilon_b$, and $S' = S'', \imath \mapsto n; 1; \mathsf{dom}(S); \epsilon$. It suffices to prove the following:
    * $\imath; 1 \vdash_{ok} \gamma'$: by inversion of $n; \imath^+, \gamma' \vdash S$ we obtain $\imath; 0 \vdash_{ok} \imath^+, \gamma'$. By inversion (rule *OK1*) of the latter fact we have that $\imath; 1 \vdash_{ok} \gamma'$.
    * $\epsilon_3 = \mathsf{run}(\gamma', \imath, 1)$ is defined: by inversion of $n; \imath^+, \gamma' \vdash S$ we obtain $\mathsf{run}((\imath^+, \gamma'), \imath, 0)$. By unfolding the definition of run $\mathsf{run}((\imath^+, \gamma'), \imath, 0)$ becomes $\mathsf{run}(\gamma', \imath, 1)$.
    * $\epsilon \cap \mathsf{dom}(S) \subseteq \epsilon$: trivially holds. The typing implies that $\mathsf{dom}(S) = M$ and ϵ is derived from γ' which is well-typed in the context of M.
    * $n; \gamma' \vdash S''$: we have that $n; \imath^+, \gamma' \vdash S''$ by inversion of $n; \imath^+, \gamma' \vdash S$. $n; \imath^+, \gamma' \vdash S''$ implies that for all $\mathtt{j}$ in $\mathsf{dom}(S'')$ such that $S''(\mathtt{j}) = (n_{1\mathtt{j}}; n_{2\mathtt{j}}; \epsilon_{1\mathtt{j}}; \epsilon_{2\mathtt{j}})$ the following hold:
      $\mathtt{j}; n_{2\mathtt{j}} \vdash_{ok} \imath^+, \gamma'$ and $\mathsf{run}((\imath^+, \gamma'), \mathtt{j}, n_{2\mathtt{j}}) \cap \epsilon_{1\mathtt{j}} \subseteq \epsilon_{2\mathtt{j}}$. By inversion of $\mathtt{j}; n_{2\mathtt{j}} \vdash_{ok} \imath^+, \gamma'$ (rule *OK3*) we have that $\mathtt{j}; n_{2\mathtt{j}} \vdash_{ok} \gamma'$. If we unfold $\mathsf{run}((\imath^+, \gamma'), \mathtt{j}, n_{2\mathtt{j}})$

once then we obtain $\mathsf{run}((\imath^+, \gamma'), \mathtt{j}, n_{2\mathtt{j}})$ is equal to $\mathsf{run}((\gamma'), \mathtt{j}, n_{2\mathtt{j}}) \cup \{\imath\}$. Thus $\mathsf{run}((\gamma'), \mathtt{j}, n_{2\mathtt{j}}) \subseteq \mathsf{run}((\imath^+, \gamma'), \mathtt{j}, n_{2\mathtt{j}})$ and therefore, $\mathsf{run}((\gamma'), \mathtt{j}, n_{2\mathtt{j}}) \cap \epsilon_{1\mathtt{j}} \subseteq \epsilon_{2\mathtt{j}}$ holds.

Lemma D.4 (Thread Lock Typing Preservation — Recursion) If $n; \gamma \vdash S,\ \gamma = \gamma_{x1} :: \gamma_{x2} :: \gamma_{x3} :: \gamma'$ and $\mathsf{rsummary}(\gamma_x) = \gamma_{x1}; \gamma_{x2}; \gamma_{x3}$ then $n; \gamma_x :: \gamma' \vdash S$.

Proof. Proof by induction. If S is empty the conclusion trivially holds. Otherwise S is of the form $S', \imath \mapsto n_1; n_2; \epsilon_a; \epsilon_b$ for some S'. There are two cases:

- $n_1 \neq n$: we need to prove that $\imath; 0 \vdash_{ok} \gamma_x :: \gamma'$ given that $\imath; 0 \vdash_{ok} \gamma_{x1} :: \gamma_{x2} :: \gamma_{x3} :: \gamma'$ holds. This is immediate by Lemma D.5.

- $n_1 = n$: as in the previous case, lemma D.5 suggests that $\imath; n_2 \vdash_{ok} \gamma_x :: \gamma'$ holds. The remaining proof obligation is $\mathsf{run}((\gamma_x :: \gamma'), \imath, n_2) \subseteq \mathsf{run}((\gamma_{x1} :: \gamma_{x2} :: \gamma_{x3} :: \gamma'), \imath, n_2)$. By observation of function run it suffices to prove that the lockset of $\gamma_{x1} :: \gamma_{x2} :: \gamma_{x3}$ is a superset of the lockset of γ_x. This is immediate by the definition of γ_{x1} that only contains r^+, r^- pairs for all all r in the domain of γ_x.

$n; \gamma \vdash S'$ holds by the induction hypothesis.

Lemma D.5 (Implication of ok)
If

- $\mathsf{rsummary}(\gamma_x) = \gamma_{x1}; \gamma_{x2}; \gamma_{x3}$

- $r; n \vdash_{ok} \gamma_{x1} :: \gamma_{x2} :: \gamma_{x3} :: \gamma$

then $r; n \vdash_{ok} \gamma_x :: \gamma$.

Proof. The second assumption implies that r belongs in the domain of γ_x and thus by inversion of the first assumption we have that $r; n_a \vdash_{ok} \gamma_x :: (r^-)^{n_b}$ such that n_a and n_b are the number of unmatched *unlock* and *lock* operations respectively for r in γ_x (notice that $n_a \leq n$ by the second assumption). The definition of $rsumarry$ also tells us that there exist exactly n_a and n_b unmatched *unlock* and *lock* operations for r in in $\gamma_{x1} :: \gamma_{x2} :: \gamma_{x3}$. Therefore, γ_x can safely replace $\gamma_{x1} :: \gamma_{x2} :: \gamma_{x3}$ and $r; n \vdash_{ok} \gamma_x :: \gamma$ holds.

Lemma D.6 (Well-Formedness) If an expression e is well-typed in the typing context $M; \Delta; \Gamma$, with effect $\gamma; \gamma'$, then $M; \Delta \vdash \Gamma$, $M; \Delta \vdash \gamma$ and $M; \Delta \vdash \gamma'$ hold.

Proof. Straightforward proof by induction on the expression typing derivation.

Lemma D.7 (Type Well-formedness) $M; \Delta; \Gamma \vdash e : \tau \,\&\, (\gamma;\gamma') \Rightarrow M; \Delta \vdash \tau$

Proof. Straightforward induction on the typing rules.

Lemma D.8 (Value-Effect) If value v is well-typed in the typing context $M; \Delta; \Gamma$, with effect $(\gamma; \gamma)$ and $M; \Delta \vdash \gamma_1$ then v is well-typed in the same typing context with effect $(\gamma_1; \gamma_1)$.

Proof. The proof is trivial, but we provide the key steps behind the proof. By inversion of the typing derivation of v (for any v) we obtain the well-formedness derivation as well as some other premises (in the case of rules *T-L*, *T-V*, *T-F*, *T-RF*, *T-T*, *T-FN*, *T-U*, and *T-FX*). We may use the latter premises of value typing, which *still hold* (same typing context), along with $M; \Delta \vdash \gamma_1$ to formulate the new value typing derivations with effect $(\gamma_1; \gamma_1)$. The cases for rules *T-RF* and *T-FX* can be shown trivially by induction (the base case is the same as for rule *T-F*).

Lemma D.9 (Evaluation Context Sub-typing)

- $M; \Delta; \Gamma \vdash E : \tau \xrightarrow{\gamma_1;\gamma_2} \tau' \,\&\, (\gamma_3;\gamma_4)$
- $\gamma_2 = \gamma_{22} :: \gamma_{21}$ and $\gamma_1 \lhd \gamma_{21}$

if and only if

- $M; \Delta; \Gamma \vdash E : \tau \xrightarrow{\gamma_1;\gamma_{21}} \tau' \,\&\, (\gamma_3;\gamma_5)$
- $\gamma_4 = \gamma_{22} :: \gamma_5$ and $M; \Delta \vdash \gamma_4$

Proof. Straightforward induction on the evaluation context typing relation. The base case is trivial. The inductive hypothesis is trivial by lemma D.10.

Lemma D.10 (Frame Sub-typing) If the following conditions hold

- $M; \Delta; \Gamma \vdash F : \tau \xrightarrow{\gamma_1;\gamma_2} \tau' \,\&\, (\gamma_3;\gamma_4)$
- $\gamma_2 = \gamma_{22} :: \gamma_{21}$ and $\gamma_1 \lhd \gamma_{21}$

if and only if

- $M; \Delta; \Gamma \vdash F : \tau \xrightarrow{\gamma_1;\gamma_{21}} \tau' \,\&\, (\gamma_3;\gamma_5)$
- $\gamma_4 = \gamma_{22} :: \gamma_5$ and $M; \Delta \vdash \gamma_4$

Proof. Straightforward case analysis on the frame typing relation.

Lemma D.11 (Variable Substitution) $M; \Delta; \Gamma, x : \tau_1 \vdash e : \tau_2 \,\&\, (\gamma_1;\gamma_2) \wedge M; \emptyset; \emptyset \vdash v : \tau_1 \,\&\, (\gamma;\gamma) \Rightarrow M; \Delta; \Gamma \vdash e[v/x] : \tau_2 \,\&\, (\gamma_1;\gamma_2)$

Proof. Straightforward induction on the expression typing derivation.

Lemma D.12 (Lock Substitution) If $M, \iota; \Delta, \rho; \Gamma \vdash e : \tau \,\&\, (\gamma;\gamma')$ then $M, \iota; \Delta; \Gamma[\iota/\rho] \vdash e[\iota/\rho] : \tau[\iota/\rho] \,\&\, (\gamma[\iota/\rho];\gamma'[\iota/\rho])$.

Proof. Proof by induction on the typing derivation of e.

Lemma D.13 (Evaluation Typing Weakening) $M; \Delta; \Gamma \vdash e : \tau \,\&\, (\gamma;\gamma')$, $M; \emptyset \vdash \tau'$ and $\iota \notin \mathsf{dom}(M)$ then $M, \iota; \Delta; \Gamma \vdash e : \tau \,\&\, (\gamma;\gamma')$.

Proof. Proof by induction on the typing derivation of e.

Lemma D.14 (Evaluation Context Typing Weakening) $M; \Delta; \Gamma \vdash E : \tau \xrightarrow{\gamma_1;\gamma_2} \tau' \,\&\, (\gamma;\gamma')$ and $\iota \notin \mathsf{dom}(M)$ then $M, \iota; \Delta; \Gamma \vdash E : \tau \xrightarrow{\gamma_1;\gamma_2} \tau' \,\&\, (\gamma;\gamma')$.

Proof. Proof by induction on the derivation of E.

Lemma D.15 (Evaluation Context Composition — E) If $M; \Delta; \Gamma \vdash e : \tau \,\&\, (\gamma_a;\gamma_b)$ and $M; \Delta; \Gamma \vdash E : \tau \xrightarrow{\gamma_a;\gamma_b} \tau' \,\&\, (\gamma_1;\gamma_2)$, then $M; \Delta; \Gamma \vdash E[e] : \tau' \,\&\, (\gamma_1;\gamma_2)$.

Proof. Proof by induction on typing derivation of E. The base case is immediate as $\square[e] = e$. The inductive case where $E = E'[F]$, the proof is immediate by inversion of the derivation of E and the application of lemma D.17.

Lemma D.16 (Evaluation Context Decomposition — E) If $M; \Delta; \Gamma \vdash E[e] : \tau' \,\&\, (\gamma_1;\gamma_2)$, then there exists a γ_a, γ_b and τ such that $M; \Delta; \Gamma \vdash e : \tau \,\&\, (\gamma_a;\gamma_b)$ and $M; \Delta; \Gamma \vdash E : \tau \xrightarrow{\gamma_a;\gamma_b} \tau' \,\&\, (\gamma_1;\gamma_2)$.

Proof. Proof by induction on the structure of E. The base case is immediate by using the well-formedness derivation for the type and typing context of e (i.e., lemmas D.6 and D.7) and the application rule *E0*. The inductive case, where $E[e] = E'[F][e]$ is immediate by lemma D.18 and rule *E1*.

Lemma D.17 (Frame Composition — F) If $M; \Delta; \Gamma \vdash e : \tau \,\&\, (\gamma_a;\gamma_b)$ and $M; \Delta; \Gamma \vdash F : \tau \xrightarrow{\gamma_a;\gamma_b} \tau' \,\&\, (\gamma_1, \gamma_2)$, then $M; \Delta; \Gamma \vdash F[e] : \tau' \,\&\, (\gamma_1;\gamma_2)$.

Proof. Proof by case analysis on typing derivation of F. The premises required to construct the typing derivation of $F[e]$ are given as premises of the typing derivation of F.

Lemma D.18 (Frame Decomposition — F) If $M; \Delta; \Gamma \vdash F[e] : \tau' \,\&\, (\gamma_1;\gamma_2)$, then there exists a γ_a, γ_b and τ such that $M; \Delta; \Gamma \vdash e : \tau \,\&\, (\gamma_a;\gamma_b)$ and $M; \Delta; \Gamma \vdash F : \tau \xrightarrow{\gamma_a;\gamma_b} \tau' \,\&\, (\gamma_1;\gamma_2)$.

Proof. Proof by case analysis on the structure of F. The premises required for each case (i.e., rules *F1-F9*) are given by the premises of the typing derivation of $F[e]$.

Lemma D.19 (Progress) Let $S; T$ be a closed well-typed configuration with $M \vdash S; T$ then $S; T$ is not stuck ($\vdash S; T$).

Proof. Without loss of generality, we choose a random thread from the thread list such that $T = T_1, n : e$ for some T_1 and show that it is either blocked or it can perform a step. By inversion of the configuration typing derivation we have that $S; M \vdash T_1, n : e$, and $M = \mathsf{dom}(S)$. By inversion of the former derivation we obtain that

- $n \notin \mathsf{dom}(T_1)$
- $n; \gamma \vdash S$
- $M; \emptyset; \emptyset \vdash e : \langle\rangle \,\&\, (\emptyset;\gamma)$: If e is a value then it can only be the unit value and a step can be performed using rule *E-T*. If e is not value then according to lemma D.20 there exists a $E[u]$ such that $e = E[u]$. Lemma D.16 implies that $M; \emptyset; \emptyset \vdash u : \tau \,\&\, (\gamma_a;\gamma_b)$, $M; \emptyset; \emptyset \vdash E' : \tau \xrightarrow{\gamma_a;\gamma_b} \langle\rangle \;\&\; \&\, (\emptyset;\gamma)$. We proceed by a case analysis on u:

  Case $\mathtt{pop}_{\gamma_a}\ v$: rule *E-PP* can be applied to perform a single step.

  Case $(v'\ v)^{\mathsf{seq}(\gamma_a)}$: the typing derivation of v' implies that v' is of the form $\lambda x.\, e'$ or $\mathtt{fix}\ x.\, e'$. In the first case rule *E-A* can be applied, whereas in the second case rule *E-FX* can be applied.

  Case $(v'\ v)^{\mathsf{par}}$: rule *E-SN* can be applied to perform a single step.

  Case $(f)\,[r]$: the typing derivation of u implies that f is of form $\Lambda\rho.\, f'$. Rule *E-RP* can be applied to perform a single step.

  Case $\mathtt{newlock}\ \rho, x\ \mathtt{in}\ e_2$: rule *E-NG* can be applied to perform a single step.

  Case $\mathtt{if}\ v\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2$: the typing derivation of u implies that v is of type $\mathtt{Bool}$. Therefore v can be either $\mathtt{true}$ or $\mathtt{false}$. In the first case rule *E-IT* can be applied, whereas in the second case rule *E-IF* can be applied.

  Case $\mathtt{unlock}\ v$: the typing derivation of u implies that v is a lock handle (i.e., $v = \mathtt{lk}_\iota$). As in lemma D.3, case *E-UL* we can use the typing derivation for thread n to derive $\gamma = \iota^-, \gamma'$, where γ is the effect assigned to the entire thread. By inversion of the store typing premise ($n; \gamma \vdash S$) of the derivation for thread n we have that $\iota; n_2 \vdash_{ok} \iota^-, \gamma'$, where n_2 is the reference count of lock ι. By inversion of the latter derivation (rule *OK2*) n_2 is positive. The latter fact and the store typing derivation also tell us that the thread identifier of ι is n. Therefore, a single step can be performed via rule *E-UL*.

Case $\texttt{lock}_{\gamma_a}\ v$: the typing derivation of u implies that v is a lock handle (i.e., $v = \texttt{lk}_\imath$). If the reference count (n_2) of lock $\imath$ is positive then the proof is similar to the case of $\texttt{unlock}\ v$ and a step can be performed via rule *E-LK1*. Otherwise, $n_2 = 0$. As in lemma D.3, case *E-LK0* we can use the typing derivation for thread n to derive $\gamma = (\imath^+, \gamma_a) :: \gamma'$, where γ is the effect assigned to the entire thread. By inversion of the store typing premise ($n; \gamma \vdash S$) of the derivation for thread n we have that $\imath; 0 \vdash_{ok} (\imath^+, \gamma_a) :: \gamma'$ and that the thread identifier of $\imath$ is n. Therefore $\imath; 0 \vdash_{ok} (\imath^+, \gamma_a) :: \gamma'$ implies $\epsilon = \mathsf{run}(\mathsf{stack}(E[\mathsf{pop}_{\gamma_a}\ \Box]), \imath, 1)$ is defined (here we are using the fact that the typing derivation implies that $\gamma_a :: \gamma' = \mathsf{stack}(E[\mathsf{pop}_{\gamma_a}\ \Box])$ and also the fact than when ok is defined so is run — this can be trivially shown).

Now, if $\epsilon \cup \{\imath\} \subseteq \mathsf{available}(S, n)$, then rule *E-LK0* can be applied. Otherwise, the thread is considered to be blocked *but not stuck* (see the third rule of judgement $stuck$).

Lemma D.20 (Redex) If $M; \Delta; \Gamma \vdash E[e] : \tau \,\&\, (\gamma_1;\gamma_2)$ and $E[e]$ is *not* a value then $M; \Delta; \Gamma \vdash E'[u] : \tau \,\&\, (\gamma_1;\gamma_2)$ such that $E'[u] = E[e]$.

Proof. By induction on the shape of e. The key idea is to convert typing derivations of e, when e is not a redex, to typing derivations of the form $E'[e']$ and apply induction for e'.