Static Safety Guarantees for a Low-level Multithreaded Language with Regions

Prodromos Gerakios[a], Nikolaos Papaspyrou[a], Konstantinos Sagonas[a,b]

[a]*School of Electrical and Computer Engineering*
National Technical University of Athens, Greece
[b]*Department of Information Technology*
Uppsala University, Sweeden

Abstract

We present the design of a formal low-level multithreaded language with advanced region-based memory management and thread synchronization primitives, where well-typed programs are memory safe and race free. In our language, regions and locks are combined in a single hierarchy and are subject to uniform ownership constraints imposed by this hierarchical structure: deallocating a region causes its sub-regions to be deallocated. Similarly, when a region is read/write-protected, then its sub-regions inherit the same access rights. We discuss aspects of the integration and implementation of the formal language within Cyclone and evaluate the performance of code produced by the modified Cyclone compiler against highly optimized C programs using pthreads. Our results show that the performance overhead for guaranteed race freedom and memory safety is in most cases acceptable.

Keywords: Safe multithreading, type and effect systems, region-based memory management, Cyclone

1. Introduction

With the emergence of commodity multicore architectures, exploiting the performance benefits of multithreaded execution has become increasingly important

Email addresses: pgerakios@softlab.ntua.gr (Prodromos Gerakios), nickie@softlab.ntua.gr (Nikolaos Papaspyrou), kostis@softlab.ntua.gr (Konstantinos Sagonas)

to the extent that doing so is arguably a necessity these days. Programming languages that retain the transparency and control of memory, such as C, seem best-suited to exploit the benefits of multicore machines, except for the fact that programs written in these languages often compromise memory safety by allowing invalid memory accesses, buffer overruns, space leaks, etc., and are susceptible to data races. Thus, a challenge for programming language research is to design and implement multithreaded low-level languages that provide static guarantees for memory safety and data race freedom and, at the same time, allow for a relatively smooth conversion of legacy C code to its safe multithreaded counterpart.

Towards this challenge, we present the design of a formal low-level concurrent language that employs advanced region-based management and hierarchical lock-based synchronization primitives. Similar to other approaches, our memory regions are organized in a hierarchical manner where each region is physically allocated within a single parent region and may contain multiple child regions. Our language 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 reader/writer lock. Thus, locks also follow the hierarchical structure of regions and in this setting each region is read/write protected by its own lock as well as the reader/writer locks of all its ancestors. As opposed to the majority of type systems and analyses that guarantee race freedom for lexically-scoped locking constructs [1, 2, 3], our language employs non-lexically scoped locking primitives, which are more suitable for languages at the C level of abstraction. Furthermore, it allows regions and locks to be safely aliased and to escape their lexical scope when passed to a new thread. These features are invaluable for expressing numerous idioms of multithreaded programming such as *sharing*, *region ownership* or *lock ownership transfers*, and *region migration*.

More importantly, our formal language is not just a theoretical design with some nice properties. As we will see, we have integrated our language constructs into Cyclone [4], a strongly-typed dialect of C which preserves explicit control and representation of memory without sacrificing memory safety. We opt for Cyclone because it is publicly available and it is more than a memory safe variant of C. Cyclone is also a low-level language that 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 constructs and the additions that were required to Cyclone's implementation.

1.1. Contributions

This article combines ideas and material which we have presented in two workshop papers [5, 6], but at the same time it significantly extends these works. In particular, our work on the formalization of hierarchical region systems [5] has laid the foundation for this article, albeit it did so with a type and effect system that is quite complicated and has several drawbacks: it requires *explicit* effect annotations, restricts aliasing, and allows temporary leaks of regions. Some of these drawbacks were lifted in the simpler and at the same time more refined type and effect system we subsequently developed [6], but its effect annotation burden was still quite high and made programming in Cyclone cumbersome. In addition, in this later system [6] region aliasing requires the programmer to create new capabilities, which entails a run-time overhead and makes programming less intuitive, and to use explicit count annotations as well as information about the "parent-of" relation, which limit polymorphism and result in code duplication.

In this article, 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 article 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.

Design and implementation. We discuss implementation issues related to static analysis, code generation and additions to the run-time system that were required in order to make the integration of the type system into Cyclone possible.

Performance evaluation. We show the effectiveness of our approach by running benchmark programs.

1.2. Overview

The next section (Section 2) presents the design goals of our language and is followed by a brief section (Section 3) showing its main features by example. We then provide a description of the formal language, its operational semantics and static semantics (Section 4), followed by a section (Section 5) where the main theorems that guarantee the absence of memory violations and data races from well-typed programs are stated and proved. After briefly reviewing the Cyclone language (Section 6), we describe the integration of our language constructs into Cyclone (Section 7), followed by a presentation of implementation (Section 8) and performance (Section 9) aspects of this integration. The article ends by two sections discussing related work and containing some concluding remarks.

2. Language design

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). In some application domains, for example systems programming, embedded and time-critical applications, low-level concurrent languages are heavily used. We believe that they will continue to be used, both by programmers and by automatic optimization tools such as compiler back-ends for higher-level languages.

Shared memory. The language should use shared memory as the means for intra-process communication. This is useful 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 will 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 [7] 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* [8] 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 subtree of a region without having to manually 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 the lock 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 *avoided* by acquiring a single lock for a group of regions rather than acquiring multiple locks for each region separately or by more involved mechanisms.[1] Additionally, our language provides 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, our language should support functions that 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

[1] Including a deadlock avoidance mechanism is beyond the scope of this article. But we remark that in a language similar to the one we describe here (i.e., in the presence of unstructured locking) it is certainly possible to design an effective type and effect system that avoids deadlocks [9].

exclusion and early region deallocation, aliasing is dangerous. We want our language to allow for 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*.

3. Language features through examples

In our language, 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. A *type and effect system* guarantees that regions and their contents are properly used. The details will be made clear in Section 4. In this section, 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. The language that we use for the examples in this section is essentially our extended version of Cyclone that we describe in Section 7, with some stylistic deviations to simplify the presentation.

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 1 (Simple region usage) Our first example shows a typical use of regions. New regions are allocated via the `region` construct. This construct requires a handle to an existing region, the heap 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 @ 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*. Effects are ordered lists 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 a vector of three counters, representing its reference count, write lock count and read lock count. The `rfree` operator appends $\texttt{Cap}\,\{\rho \mapsto (-1,0,0))\}$ to the effect, which states that the vector $(-1,0,0)$ is added to the current vector of region ρ, i.e., the region count of ρ is decremented by one.

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

- The first action validates each constraint of the effect starting with the initial count $(1,1,0)$: when ρ is created, it is *live* and the thread that created it has exclusive/direct access to it, that is, ρ is *thread-local*. In our example, starting with 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 satisfied and the resulting count of ρ is $(0,1,0)$.

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

  Regions must definitely be released before the end of their lexical scope, either by releasing them explicitly or by releasing one of their ancestors. If

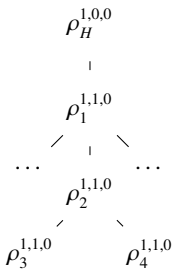
the resulting count for ρ were not of the form $(0, n_1, n_2)$, then it should be the case that one of its ancestors has been released; therefore, the constraint $\neg\texttt{Live}\,\rho_H$ would be appended during the translation (and, in this case, effect validation would fail, as the heap region is immortal).

In our example, 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.

Example 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.

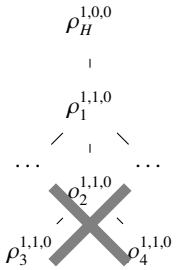
```
region<ρ1> h1 @ H;
    ...
    region<ρ2> h2 @ h1;
        ...
        region<ρ3> h3 @ h2;
            region<ρ4> h4 @ h2;
                ...
```



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 3 (Bulk region deallocation) In Example 1, 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;
    ...
    region<ρ2> h2 @ h1;
        ...
        region<ρ3> h3 @ h2;
            region<ρ4> h4 @ h2;
                ...
                rfree(h2);
                ...
                // ρ2, ρ3 and ρ4 are no longer alive
```



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

```
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!
    }
}
```

In each iteration of the loop, the server thread allocates a new region ρ in the heap and passes its handle h to function `wait_data`, which fills 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 only 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 `spawn` operator is also passed the tuple $(h, 1, 1, 0)$, which denotes 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 live nor accessible. In the output thread, ρ is directly accessible and no further lock operations are required. Furthermore, the output thread is now obliged to explicitly release ρ before it terminates.

Example 5 (Region sharing) 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 and display the data 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^{1,1,0}
            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}
                ...
    }
}
```

Operator `share` increases the region count. Consequently, the original count for ρ, namely $(1, 1, 0)$ is transformed to $(2, 1, 0)$. Thread `output` steals $(1, 1, 0)$, thus the counts remaining for ρ in the server thread are $(1, 0, 0)$. Region ρ is now shared between the two threads; however, the server thread does not have access to ρ. The `wr_lock` and `wr_unlock` operators have to be used to explicitly lock and unlock ρ, before accessing its contents. Notice, that we acquire exclusive (read and write) access to region ρ.

Example 6 (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 read-only 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, and then the next 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 7 (Hierarchical locking) In the previous example, locking and unlocking was performed on a leaf region. In general, locking some 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 the following code, 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 one of their common ancestors.

```
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 8 (Reentrant locks) An expressive language with regions will have to support region polymorphism. This invariably leads to *region aliasing*, which introduces the need for reentrant locks. To see this, let us define a swapping function that accepts two references in unlocked regions. For swapping their contents, the

function 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) {
    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
}
```

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

```
swap<ρ,ρ>(h, h, a, b);
```

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

Example 9 (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 10 (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. Then, region ρ_1 migrates to a newly spawned thread executing function `f`.

```
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 and therefore ρ_1 and ρ_2 are aliases to the same region ρ, a data race may occur since both threads (executing functions `bar` and `f`) will have write access to z.

```
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 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 11 (Negative constraints) As explained in Example 9, programs that grant region access to more than one 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(z);     // ρ1^{1,0,0}, ρ2^{1,1,0}
            *z = 17;                                // possible data race!
```

Expression	e	::=	$x \mid f \mid () \mid \texttt{true} \mid \texttt{false} \mid e\ e \mid e\,[r] \mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e$
		|	$\texttt{newrgn}\,\rho, x\ @\ e\ \texttt{in}\ e \mid \texttt{new}\ e\ @\ e \mid e := e \mid \texttt{deref}\ e \mid \texttt{cap}_{\eta}\ e$
		|	$\texttt{spawn}_{\xi}\ e \mid \texttt{loc}_{\ell} \mid \texttt{rgn}_{\iota}$
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 1: Syntax.

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 not satisfied and the program is rejected.

4. Formal language

In this section we formalize the main aspects of the language presented in the previous section using an extension of lambda calculus (see Figure 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 ($\texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2$). Functions can be monomorphic ($\lambda x.\, e$), polymorphic ($\Lambda\rho.\, f$) where ρ is a region variable, and recursive ($\texttt{fix}\ x.\, f$). The application of region polymorphic functions is explicit ($e\,[r]$) where r is a metavariable ranging over region variables ρ and region constants ι. 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 $\texttt{newrgn}\ \rho, x\ @\ e_1\ \texttt{in}\ e_2$ 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. Each region is associated with a count vector η, consisting of three non-negative numbers (n_1, n_2, n_3) that we call capability counts:

- 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.

Locks are represented by counters instead of boolean values to support re-entrant locks and region aliasing, as explained in Section 3. 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 read or written 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 `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 (`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 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, if a count is negative then 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

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	::=	$\Box\ e \mid v\ \Box \mid \Box[r] \mid \texttt{if}\ \Box\ \texttt{then}\ e\ \texttt{else}\ e \mid \texttt{newrgn}\ \rho, x\ @\ \Box\ \texttt{in}\ e$
		$\mid$	$\texttt{new}\ \Box\ @\ e \mid \texttt{new}\ v\ @\ \Box \mid \Box := e \mid v := \Box \mid \texttt{deref}\ \Box \mid \texttt{cap}_{\eta}\ \Box$

Figure 2: Auxiliary syntax for the operational semantics.

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. The construct $\texttt{cap}_{\eta}\ e$ is the formal counterpart of the constructs with the more descriptive names that were used in Section 3; for instance, `share` is syntactic sugar for $\texttt{cap}_{(1,0,0)}$, `rfree` for $\texttt{cap}_{(-1,0,0)}$, and `wr_lock` for $\texttt{cap}_{(0,1,0)}$.

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 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 $\texttt{rgn}_{\iota}$ and $\texttt{loc}_{\ell}$ correspond to explicit region and location handles (the metavariables ι 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 4.1.

4.1. Operational semantics

We define a *small-step* operational semantics for our language in Figures 2 and 3. The thread evaluation relation $C \rightsquigarrow C'$ transforms configurations. A con-

figuration C consists of an abstract store S and a thread list T.[2] A store S maps region identifiers (ι) 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 thread-local count vector η 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 [10]. 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 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 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 $\perp$. The rule adds a fresh and empty region ι (i.e ι does not belong in the regions of S)

[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.

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

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

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

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

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

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

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

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

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

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

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

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

Figure 3: Evaluation relation $C \rightsquigarrow C'$.

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

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

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

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

Figure 4: Auxiliary functions and predicates.

to the store and associates it with the pair, $((1,1,0), j)$ 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 ι is live in θ and updates the heap of ι with a fresh location ℓ (i.e ℓ does not belong in the locations of S) mapping to value v. Notice, that ι need not be protected.

Rule *E-AS* requires that location ℓ exists in some region ι of the global store S and requires that ι is exclusively owned by the current thread. This is established by $\iota \in \mathsf{wlocked}(\theta)$ and $\iota \notin \mathsf{rwlocked}(T)$. 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 ι or any other thread has read or write access to ι, then the evaluation will get stuck.

In contrast to the rule for assignment, rule *E-D* is more permissive as it admits simultaneous *read-only* access to region ι by more than one threads. It requires that the current thread has (possibly) non-exclusive access to region ι 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 approach differs from related work, e.g., the work of Grossman [2], 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. Our type safety results (see Section 5) guarantee that no thread can get stuck by violating mutual exclusion.

Rule *E-CP* requires that region ι is live in θ and adds the relative count vector η' to the current vector η for ι. 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 ι 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'.

4.2. 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 5. Basic types consist of the unit type and the type of boolean values. Monomor-

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	Γ	::=	$\emptyset \mid \Gamma, x : \tau$
Region context	Δ	::=	$\emptyset \mid \Delta, \rho$
Heap context	M	::=	$\emptyset \mid M, \ell \mapsto (\tau, \iota)$
Store context	R	::=	$\emptyset \mid R, \iota$

Figure 5: Syntax for types, effects and contexts.

phic 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 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. Preserving the order of events during type-checking is crucial for the correctness of our type system. Although in Figure 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 (¬RW), not writable (¬W), live (Live), and not live (¬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.[3]

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 6 and uses four typing contexts (defined in Figure 5 on page 21): 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, respectively) 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 $\texttt{Join}\,\gamma_1\,\gamma_2$ 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 can be 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

[3]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.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Figure 6: Typing rules.

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 4.3.

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 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

$$
\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 7: Effect validation and transformation.

may be erroneous because it tries to dereference z after it has been unlocked:

```
region<ρ> h @ hr;
    let z = rnew(h) 42;  ∗z = 17;  wr_unlock(h);  print(∗z);  rfree(h)
```

Starting with a count vector of $\eta = (1, 1, 0)$, the validation phase for γ first checks that the constraint $\mathtt{Live}\,\rho$ is satisfied and then checks that the constraint $\mathtt{W}\,\rho$ 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 $\mathtt{R}\,\rho$ is satisfied and finds that it is not, as ρ is now not protected for reading. It therefore generates a constraint $\mathtt{R}\,r$ 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 effect validation succeeded, either because we fixed the program (e.g., by removing the unlock operation or by moving the read operation before it), or because the parent region was indeed locked for reading. 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 phase translates all events that mention ρ to $\mathtt{Live}\,r$. 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:

```
region<ρ> h @ hr;
    let z = rnew(h) 42;  ∗z = 17;  rfree(hr);  print(∗z);  rfree(h)
```

where the effect for the second line is $\gamma = \mathtt{Live}\,\rho :: \mathtt{W}\,\rho :: \mathtt{Cap}\,\{r \mapsto (0, -1, 0)\} :: \mathtt{R}\,\rho :: \mathtt{Cap}\,\{\rho \mapsto (-1, 0, 0)\}$. In this case, with respect to ρ, validation succeeds and the transformed effect is $\gamma' = \mathtt{Live}\,r :: \mathtt{Live}\,r :: \mathtt{Cap}\,\{r \mapsto (0, -1, 0)\} :: \mathtt{Live}\,r :: \mathtt{Live}\,r$. 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 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 ¬`RW` and ¬`W` always propagate to the parent. The same happens with ¬`Live`, unless the region's reference count is zero.

The partial function translate(γ, r, η, 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; this rule enforces the invariant that r must have been released by the end of its lexical scope. 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, as well as 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 the function p-constraint, which guarantee mutual exclusion between the new thread 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 p-constraint(r', η) is to determine the constraints on r' that must be satisfied by the first thread to guarantee mutual exclusion. If 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 are imposed on r'. Notice that p-constraint is used twice in rule *TR-ST*, symmetrically.

4.3. 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 8, which imposes some restrictions on recursive functions and calculates the *summarized*

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

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

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

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

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

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

Figure 8: Summarized effects of recursive functions.

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 the 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 the 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 the rule *SUM*.

After all this, we can now return to the explanation of rule *T-FX* on page 23. 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 summarization.

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.

5. Type safety

In this section we discuss the fundamental theorems that prove type safety of our language.[4] 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 4.1, a thread may also 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 1 (Constraint validity) Predicate $\mathsf{cvalid}(\delta; \iota; \theta)$ is true when the constraint δ on ι is consistent with the run-time hierarchy θ. It is defined as follows:

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

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

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

Definition 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'.

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

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

Definition 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, $\mathsf{valid}(\gamma;\theta)$ is true when the effect γ is valid in hierarchy θ and the result is a hierarchy with no live regions.

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

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

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

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

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

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 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:

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

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 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:

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

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 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 4 on page 19).

Definition 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.

Definition 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:

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

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

Definition 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 article.

Lemma 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 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 translate entails effect validity, and *E-FX*, where it must be shown that function summary entails effect validity. These two are established by Lemmata 3 and 4, respectively.

Lemma 3 (Translate implies valid) If $\mathsf{valid}(\mathsf{translate}(\gamma, \iota, \eta, \jmath);\theta)$ for some region ι such that $\iota \notin \mathsf{dom}(\theta)$, then $\mathsf{valid}(\gamma;\theta,\iota \mapsto (\eta, \jmath))$.

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

Lemma 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 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. 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 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 1 implies that $S_n; T_n$ is not stuck.

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

6. Cyclone: a memory-safe dialect of C

To make this article self-contained, in this section we provide a brief overview of Cyclone before our additions. In particular, we first discuss memory management aspects of Cyclone: we describe how regions are used in Cyclone (in Section 7 we will refer to Cyclone's existing regions as "traditional") and identify some shortcomings that are alleviated by adopting the kind of regions that we proposed in this article (we will refer to these regions as "extended"). We also show, through several examples, that Cyclone's memory safety guarantees only hold for sequential programs. (All code excerpts are henceforth shown using Cyclone syntax.) In Section 7 we will present how the language additions we propose can be smoothly integrated into Cyclone and provide safety in the presence of multithreading.

6.1. 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> r;                          // live regions: {‘r}
  int *‘r z = rnew(r) 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 r.) 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* (r), which is given the *singleton type* `region<‘r>`. The second statement uses r 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>r;
  int * ‘r z = rnew(r) 42;
  int * ‘r y = rnew(r) 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. It should be noted that a dynamic region cannot be deallocated as long as key has been *borrowed*.

```
void access_and_deallocate (NewDynRgn pr ) {
    let NewDynRgn{<'r> key} =  pr;          // open existential
      { region r = open(key);               // borrow key for this scope
        let x = rnew(r) 42;
            ...                             // region 'r cannot be deallocated here
      }
      ...                                   // do some work
      free_ukey(key);                       // deallocate region 'r
}
```

Additionally, Cyclone allows tracked references to leak and thus allows dynamic regions to leak as well. This would be the case, for instance, in the program above if the programmer had not called `free_ukey`. To tackle this issue, an intra-procedural analysis can be used to report tracked reference leaks as warnings. In practice, this analysis produces a large number of false positives [11]. 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.* [11].

6.2. Concurrency in Cyclone

Besides our work, the only attempt to add safe multithreading to Cyclone has been the work of Grossman [2]. However, Grossman's proposal was never implemented and consequently the Cyclone implementation does not come with built-in 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:

```
int pthread_create (pthread_t @, const pthread_attr_t *,
                    'a(@'H)('b), 'b arg : regions('b) ≤ 'H)
```

The most interesting part of this 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 multithreading. Here we will only mention a few such cases.

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

```
void foo (int * `r * `r x) {
  if (x != NULL && *x != 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 deallocations.

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 non-blocking and thread-safe.

7. Interaction with Cyclone

In this section, we provide an in-depth description of the interaction between our system and Cyclone.

7.1. Extended regions and kind system

In contrast with Cyclone's traditional lexically scoped regions, which are allocated in a LIFO manner, our *extended* regions can be allocated at any extended

region ancestor or under the main heap. We consider the main heap ($`H$) as the root of our region hierarchy (it coincides with $\perp$ in the hierarchy of extended regions). We have already shown a number of examples using extended regions in Section 3. This form of allocation generalizes the stack-based region organization to a tree-based organization and enables finer-grained control of region lifetimes.

Traditional lexically scoped regions cannot be shared safely. For instance, *stack frames* are treated as traditional regions and sharing the stack in a safe manner would have a severe impact on concurrency between threads. Nonetheless, traditional regions must be preserved for backwards compatibility. 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. 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*. Notice that extended regions start as thread-local, i.e., the thread that creates them can immediately access their contents without locking.

In contrast with traditional dynamic regions that may leak, as we saw in Section 6.1, our extended regions can never leak as each well-typed thread must explicitly release its extended regions. Our effect analysis employs negative liveness constraints ($\neg$`Live`) to verify definite release of regions. Of course, it is possible to safely release extended regions in bulk by releasing their common ancestor. Finally, less effort is required to write programs that explicitly manage the lifetime of extended regions as there is no distinction between linear and aliased regions.

7.2. Operating on capabilities

The `cap` operator of the formal semantics is available as is in our extension of Cyclone and can be used in programs. However, since this operator is quite low-level and not particularly user-friendly, our implementation also comes with various built-in defines, using the macro preprocessor of Cyclone, that provide mnemonic aliases as those used in the examples of Section 3 for most common operations. Two of these defines are shown below.

`#define rfree(`h`)`	`cap` $(h,-1,0,0)$
`#define wr_unlock(`h`)`	`cap` $(h,0,-1,0)$

In the code excerpts shown in the rest of the article, we will assume that these defines are available.

7.3. Exceptions

Having static guarantees about the control flow of a program plays a crucial role in manual memory management. As mentioned in Section 6, 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), 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.[5] 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. `@throws`(...) enumerates all exceptions that may be thrown from a function body;
2. `@nothrow` 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.)

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.

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

7.4. Reentrant and extended region functions

Global data is implicitly shared by all threads and this may cause data races. To preserve race freedom, we have constrained our language so that only extended regions can be shared between threads. (However, 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 is declared as `@re_entrant`. A function annotated as `@re_entrant` cannot access 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.

To speed up the compilation process, we require that functions that modify or access in any way extended regions must be explicitly declared as `@xrgn`. In summary, the use of `@re_entrant` implies that a function does not use existing traditional regions (i.e., the heap, global data and tracked objects), whereas the absence of `@xrgn` implies that a function does not create or use extended regions. The two annotations can be used independently.

7.5. Thread creation

Threads can be explicitly created by 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.

7.6. Type polymorphism

Cyclone effects are not polymorphic. To allow the invocation of functions, which have polymorphic arguments (e.g., ‘a), Cyclone programmers use the operator `regions`(‘a). 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.

7.7. 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
    rfree(child);
```

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 had more than one lock capability) during the scope of `xopen`. On the downside, region ‘*child must* remain locked for the scope of `xopen`.

7.8. Memory consistency

Our formal language semantics assumes a *sequentially consistent* memory model [10], 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 [12, 13] 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, 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* [14]. 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*" [15, 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.

8. Implementation

8.1. 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 the insertion of dynamic checks (e.g., array bound checks).

The compiler may eliminate some candidate locations, by utilizing checks inserted by the programmer (e.g, `if` $(i < len)\ a[i] = 42;$). As illustrated in Section 6, 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 `@throws` specification.

Finally, a control-flow sensitive effect analysis is performed. The analysis propagates effects through the control-flow graph and appends new effects when appropriate. As in the type-checking rules, effect *translation* is performed when the entire effect of a region has been gathered.

It is worth noting that iteration and exception handling are treated *conservatively*. In particular, iterative statements are treated similarly to recursive functions (i.e., their effects are computed using the rules of Figure 8). 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 as our effect system maintains the effects of *both* branches. 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., `__attribute((noreturn))__`).

The formal analysis does permit higher-order functions whose parameters have non-empty effects. In the actual implementation, function pointers can be used as parameters but their effect must be empty. However, it would be possible to perform a points-to analysis for function pointers, which would infer for each pointer the set of functions that it may refer to and their corresponding effects $\{f_1 : \gamma_1, ..., f_n : \gamma_n\}$. The effect of a indirect function call could then be formulated as a joint effect $\texttt{Join}\,\gamma_1\,(\ldots(\texttt{Join}\,\gamma_{n-1}\,\gamma_n)\ldots)$.

8.2. Error reporting

During region checking, it is possible that effect validation fails. In this case, an error message is reported and our implementation tries to associate it with the offending statement. This is not always an easy task, as the offending statement in general contributes some constraint to the effect and this constraint is found not to be satisfied at a later time, possibly in the context of a different function. The example in Figure 9 illustrates this. This program is rejected, as function `f` assigns to a reference in region h that is not write-locked. The error message issued by the compiler points to line 4 of the source code, where the actual assignment takes place, but also reports the path of function calls that led to this error (line 11).

```
COMPILER OUTPUT:
t0.cyc:4: Could not solve: Read('h) for region 'h. XCap: (1, 0, 0).
          Path: t0.cyc:11
COMPILATION FAILED!
```

```
1:  #include <core.h>
2:
3:  void f(region_t<‘r :: X> h, int @ ‘r0 :: X ref) @xrgn {
4:      let tmp = *ref;
5:  }
6:
7:  int main() {
8:      region h @Core :: heap_region;
9:      wr_unlock(h);                          // defined as cap(h, 0, −1, 0)
10:     let ref = rnew(h) 30;
11:     f(h, ref);
12:     rfree(h);                              // defined as cap(h, −1, 0, 0)
13:     return 0;
14: }
```

Figure 9: A program with an error illustrating how errors are reported in our implementation.

8.3. Code generation

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

- Translate `spawn` statements to low-level primitives, packing and unpacking function arguments and placing the actual call in a wrapper function, which acts as a glue between the call and the thread that will execute it.
- Generate specialized code for allocating extended regions and references.

8.4. Run-time system

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

- It registers new allocated regions to the appropriate local thread hierarchy.
- 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 the `spawn` operation to construct the subtree passed to the 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).

9. Performance evaluation

We evaluated our implementation on concurrent benchmark programs taken from "The Computer Language Benchmarks Game."[6] 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:[7]

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

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

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

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

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 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 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 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 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.

In terms of memory consumption, our implementation uses more or less the same amount of memory as the original programs. The only benchmark with a noticeable difference in memory consumption 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.

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

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 Section 7. 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 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.

10. Related work

Regions. The first statically checked stack-based region system was developed by Tofte and Talpin [7]. Since then, several memory-safe systems that enable early region deallocation for a sequential language have been proposed [16, 17, 18, 19]. RC [8] and Cyclone [4] were the first imperative languages to allow safe region-based memory management with explicit constructs. Both allowed early region deallocation and RC also introduced the notion of multi-level region hierarchies. RC programs may throw region-related exceptions, whereas our approach is purely static. Both Cyclone and RC make no claims of memory safety or race freedom in the presence of multithreading. To overcome such limitations, Gross-

man proposed a type system for safe multithreading in Cyclone [2]. Race freedom is guaranteed by statically tracking locksets within lexically-scoped synchronization constructs. Grossman's proposal, which was never implemented, allows for fine-grained locking, but does not enable early release of regions and locks and provides no support for data migration or lock transfers. In contrast, we provide such support and also support bulk region deallocation and hierarchical locking via read/write locks, as opposed to just primitive locking. Moreover, we also offer a complete implementation of our ideas.

Type systems for objects. Statically checked region systems have also been proposed for real-time Java to eliminate the dynamic checks imposed by its specification. Boyapati *et al.* have introduced hierarchical regions in ownership types [20], but the approach suffers from similar disadvantages as Grossman's work. Additionally, their type system only allows sub-regions for *shared* regions, whereas we do not have this limitation. In previous work, Boyapati *et al.* also proposed an ownership-based type system that prevents deadlocks and data races [3]. In contrast to that system, we support locking of arbitrary nodes in the region hierarchy. Static region hierarchies (depth-wise) have been used by Zhao *et al.* [21]. Their main advantage is that programs require fewer annotations as compared to programs with explicit region constructs. In the same track, Zhao *et al.* proposed implicit ownership annotations for regions [22]. Thus, classes that have no explicit owner can be allocated in any static region. This is a form of *existential ownership*. None of the above approaches allow full ownership abstraction for region subtrees.

Cunningham *et al.* proposed a universe type system to guarantee race freedom in a calculus of objects [23]. Similarly to our system, object hierarchies can be atomically locked at any level. Unlike our system, they do not support early lock releases and lock ownership transfers between threads.

More recently, Bocchino *et al.* proposed a type and effect system for DPJ (Deterministic Parallel Java [24]) that partitions the heap into hierarchical regions and uses those regions to disambiguate accesses to distinct objects. Their 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 region disjointness invariant places significant constraints on region 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 [25]. 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* [26]. 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.

Safer variants of C. Many systems aim to make C code safer. Among them we mention Safe-C [27], CCured [28] and Deputy [29]. Some of these systems do not guarantee soundness, drop the explicit memory representation of C programs which make them inappropriate for certain kinds of low-level programming, or provide no guarantees when concurrency enters the picture. There are also numerous languages at the C level of abstraction with explicit concurrency features (notably Cilk [30], nesC [31] and Pillar [32]) but to the best of our knowledge none of them provides both memory safety and race freedom guarantees like our language.

11. Concluding remarks

In this article we presented the design and implementation of a formal low-level language, employing region-based memory management and locking primitives, that provides memory safety and race freedom guarantees for well-typed programs. We discussed the integration of our formal language within Cyclone, argued about the decisions that we have made in order to guarantee memory safety and race freedom at the implementation level, and evaluated the performance of programs written in our language against highly optimized C programs, showing

performance overheads that are acceptable given the static safety guarantees in the presence of multithreading.

Our language is the first variant of Cyclone that has been implemented with these properties, and one of the very few programming languages at this level of abstraction that has been designed and implemented with this goal in mind.

Acknowledgement

This research is 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."

References

[1] C. Flanagan, M. Abadi, Object types against races, in: J. C. M. Baeten, S. Mauw (Eds.), International Conference on Concurrency Theory, volume 1664 of *LNCS*, Springer, 1999, pp. 288–303.

[2] D. Grossman, Type-safe multithreading in Cyclone, in: Proceedings of the ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, ACM Press, New York, NY, USA, 2003, pp. 13–25.

[3] C. Boyapati, R. Lee, M. 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, ACM Press, New York, NY, USA, 2002, pp. 211–230.

[4] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, J. Cheney, Region-based memory management in Cyclone, in: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, ACM Press, New York, NY, USA, 2002, pp. 282–293.

[5] P. Gerakios, N. Papaspyrou, K. Sagonas, A concurrent language with a uniform treatment of regions and locks, in: A. R. Beresford, S. Gay (Eds.), Proceedings of the Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software, volume 17 of *EPTCS*, pp. 79–93.

[6] P. Gerakios, N. Papaspyrou, K. Sagonas, Race-free and memory-safe multithreading: Design and implementation in Cyclone, in: Proceedings of the ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, ACM Press, New York, NY, USA, 2010, pp. 15–26.

[7] M. Tofte, J.-P. 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, ACM Press, New York, NY, USA, 1994, pp. 188–201.

[8] D. Gay, A. Aiken, Language support for regions, in: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, ACM Press, New York, NY, USA, 2001, pp. 70–80.

[9] P. Gerakios, N. Papaspyrou, K. Sagonas, A type and effect system for deadlock avoidance in low-level languages, in: Proceedings of the ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, ACM Press, New York, NY, USA, 2011, pp. 15–28.

[10] L. Lamport, A new approach to proving the correctness of multiprocess programs, ACM Trans. Progr. Lang. Syst. 1 (1979) 84–97.

[11] N. Swamy, M. Hicks, G. Morrisett, D. Grossman, T. Jim, Safe manual memory management in Cyclone, Science of Computer Programming 62 (2006) 122–144.

[12] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, J. Hennessy, Memory consistency and event ordering in scalable shared-memory multiprocessors, in: Proceedings of the Annual International Symposium on Computer Architecture, ACM Press, New York, NY, USA, 1990, pp. 15–26.

[13] S. V. Adve, M. D. Hill, Weak ordering – a new definition, in: Proceedings of the Annual International Symposium on Computer Architecture, ACM, New York, NY, USA, 1990, pp. 2–14.

[14] H.-J. Boehm, Threads cannot be implemented as a library, in: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, ACM Press, New York, NY, USA, 2005, pp. 261–268.

[15] R. M. Stallman, the GCC Developer Community, Using the GNU Compiler Collection, 2011. http://gcc.gnu.org/onlinedocs/ (version 4.6.0).

[16] A. Aiken, M. Fähndrich, R. Levien, Better static memory management: Improving region-based analysis of higher-order languages., in: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, ACM Press, New York, NY, USA, 1995, pp. 174–185.

[17] F. Henglein, H. Makholm, H. 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, ACM, New York, NY, USA, 2001, pp. 175–186.

[18] D. Walker, K. Watkins, On regions and linear types, in: Proceedings of the ACM SIGPLAN International Conference on Functional Programming, ACM Press, New York, NY, USA, 2001, pp. 181–192.

[19] M. Fluet, G. Morrisett, A. Ahmed, Linear regions are all you need, in: P. Sestoft (Ed.), Programming Language and Systems: Proceedings of the European Symposium on Programming, volume 3924 of *LNCS*, Springer, 2006, pp. 7–21.

[20] C. Boyapati, A. Salcianu, W. S. Beebee, M. 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, ACM Press, New York, NY, USA, 2003, pp. 324–337.

[21] T. Zhao, J. Noble, J. Vitek, Scoped types for real-time Java, in: Proceedings of the 25th IEEE International Real-Time Systems Symposium, IEEE Computer Society, 2004, pp. 241–251.

[22] T. Zhao, J. Baker, J. Hunt, J. Noble, J. Vitek, Implicit ownership types for memory management, Science of Computer Programming 71 (2008) 213–241.

[23] D. Cunningham, S. Drossopoulou, S. Eisenbach, Universes for race safety, in: Proceedings of the Workshop on Verification and Analysis of Multi-threaded Java-like Programs, pp. 20–51.

[24] R. L. Bocchino, Jr., V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, M. 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, ACM, New York, NY, USA, 2009, pp. 97–116.

[25] R. L. Bocchino, Jr., S. Heumann, N. Honarmand, S. V. Adve, V. S. Adve, A. Welc, T. Shpeisman, Safe nondeterminism in a deterministic-by-default parallel language, in: Conference Record of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ACM, New York, NY, USA, 2011, pp. 535–548.

[26] N. D. Matsakis, T. 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, ACM, New York, NY, USA, 2010, pp. 634–651.

[27] T. M. Austin, S. E. Breach, G. S. Sohi, Efficient detection of all pointer and array access errors, in: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, ACM, New York, NY, USA, 1994, pp. 290–301.

[28] G. C. Necula, J. Condit, M. Harren, S. McPeak, W. Weimer, CCured: Type-safe retrofitting of legacy software, ACM Trans. Progr. Lang. Syst. 27 (2005) 477–526.

[29] J. Condit, M. Harren, Z. R. Anderson, D. Gay, G. C. Necula, Dependent types for low-level programming, in: R. De Nicola (Ed.), Programming Language and Systems: Proceedings of the European Symposium on Programming, volume 4421 of *LNCS*, Springer, 2007, pp. 520–535.

[30] M. Frigo, C. E. Leiserson, K. H. Randall, The implementation of the Cilk-5 multithreaded language, in: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, ACM Press, New York, NY, USA, 1998, pp. 212–223.

[31] D. Gay, P. Levis, J. R. von Behren, M. Welsh, E. A. Brewer, D. 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, ACM, New York, NY, USA, 2003, pp. 1–11.

[32] T. Anderson, N. Glew, P. Guo, B. T. Lewis, W. Liu, Z. Liu, L. Petersen, M. Rajagopalan, J. M. Stichnoth, G. Wu, D. Zhang, Pillar: A parallel implementation language, in: Proceedings of the International Workshop on Languages and Compilers for Parallel Computing, volume 5234 of *LNCS*, Springer, 2008, pp. 141–155.

Appendix: Formalization and proof

In this appendix, we provide the complete formalization of our type and effect system, as well as the proof of type safety. Although a lot of material (e.g., the language syntax, the operational semantics, the typing rules) is duplicated in the paper, we opted to make this appendix self contained, to facilitate the task of the reviewers. It is up to the editor to decide if this appendix (or a version with duplicated elements removed) will be part of the article or not, should the article finally be accepted for publication.

Appendix A. Language syntax

Expression	e	::=	$x \mid f \mid () \mid \texttt{true} \mid \texttt{false} \mid e\ e \mid e\,[r] \mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e$
		|	$\texttt{newrgn}\,\rho, x\ @\ e\ \texttt{in}\ e \mid \texttt{new}\ e\ @\ e \mid e := e \mid \texttt{deref}\ e \mid \texttt{cap}_\eta\ e$
		|	$\texttt{spawn}_\xi\ e \mid \texttt{loc}_\ell \mid \texttt{rgn}_\iota$
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	::=	$\emptyset \mid \xi, r \mapsto \eta$

Appendix B. 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	::=	$\square \mid E[F]$
Frame	F	::=	$\square\ e \mid v\ \square \mid \square\,[r] \mid \texttt{if}\ \square\ \texttt{then}\ e\ \texttt{else}\ e \mid \texttt{newrgn}\,\rho, x\ @\ \square\ \texttt{in}\ e$
		|	$\texttt{new}\ \square\ @\ e \mid \texttt{new}\ v\ @\ \square \mid \square := e \mid v := \square \mid \texttt{deref}\ \square \mid \texttt{cap}_\eta\ \square$
Redex	u	::=	$(\lambda x.\, e)\ v \mid \texttt{cap}_\eta\ \texttt{rgn}_\iota \mid \texttt{deref}\ \texttt{loc}_\ell \mid \texttt{loc}_\ell := v \mid \texttt{new}\ v\ @\ \texttt{rgn}_\iota$
		|	$\texttt{newrgn}\,\rho, x\ @\ \texttt{rgn}_\iota\ \texttt{in}\ e_2 \mid (\Lambda\rho.\, f)\,[r] \mid \texttt{spawn}_\xi\ e_1 \mid (\texttt{fix}\ x.\, f)\ v$
		|	$\texttt{if true then}\ e_1\ \texttt{else}\ e_2 \mid \texttt{if false then}\ e_1\ \texttt{else}\ e_2$

Evaluation relation $C \rightsquigarrow C'$

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

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

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

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

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

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

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

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

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

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

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

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

Auxiliary functions and predicates

$$\begin{array}{lcll} \mathsf{merge}(\emptyset) & = & \emptyset & \\ \mathsf{merge}(\xi, r \mapsto \eta) & = & \mathsf{merge}(\xi), r \mapsto \eta & \text{if } r \notin \{r' \mid r' \mapsto \eta' \in \xi\} \\ \mathsf{merge}(\xi, r \mapsto \eta, r \mapsto \eta') & = & \mathsf{merge}(\xi, r \mapsto (\eta_1 + \eta_2)) & \end{array}$$

$$
\begin{array}{lcll}
\mathsf{ok}(n_1, n_2, n_3) & = & n_1 \geq 0 \wedge n_2 \geq 0 \wedge n_3 \geq 0 & \\
(c_1, w_1, z_1) \oplus (c_2, w_2, z_2) & = & (c_1 + c_2, w_1 + w_2, z_1 + z_2) & \text{if } \mathsf{ok}(c_1, w_1, z_1) \wedge \mathsf{ok}(c_2, w_2, z_2) \wedge \\
 & & & (w_1 = 0 \vee w_2 = 0) \wedge (c_2 > 0) \wedge \\
 & & & (c_1 = 0 \implies w_1 = z_1 = 0) \wedge \\
 & & & (w_1 > 0 \implies z_2 = 0) \wedge \\
 & & & (w_2 > 0 \implies z_1 = 0)
\end{array}
$$

$$
\begin{array}{lcll}
\mathsf{ancestors}(\theta, \bot) & = & \emptyset & \\
\mathsf{ancestors}(\theta, \iota) & = & \{\iota\} \cup \mathsf{ancestors}(\theta', \jmath) & \text{if } \theta = \theta', \iota \mapsto (\eta, \jmath) \\
\mathsf{live}(\theta) & = & \{\iota \mid \forall \jmath \in \mathsf{ancestors}(\theta, \iota).\ \exists \jmath \mapsto (\eta, \jmath') \in \theta.\ \mathsf{ok}(\eta - (1,0,0))\} & \\
\mathsf{wlocked}(\theta) & = & \{\iota \mid \iota \in \mathsf{live}(\theta) \wedge \exists \jmath \mapsto (\eta, \jmath') \in \theta.\ \jmath \in \mathsf{ancestors}(\theta, \iota) \wedge \mathsf{ok}(\eta - (0,1,0))\} & \\
\mathsf{rlocked}(\theta) & = & \{\iota \mid \iota \in \mathsf{live}(\theta) \wedge \exists \jmath \mapsto (\eta, \jmath') \in \theta.\ \jmath \in \mathsf{ancestors}(\theta, \iota) \wedge \mathsf{ok}(\eta - (0,0,1))\} & \\
\mathsf{rwlocked}(\theta) & = & \mathsf{rlocked}(\theta) \cup \mathsf{wlocked}(\theta) & \\
\mathsf{wlocked}(T) & = & \{\iota \mid \exists \langle \theta, e \rangle \in T.\ \iota \in \mathsf{wlocked}(\theta)\} & \\
\mathsf{rwlocked}(T) & = & \{\iota \mid \exists \langle \theta, e \rangle \in T.\ \iota \in \mathsf{rwlocked}(\theta)\} & \\
\mathsf{mutex}(\{\theta_1, \ldots, \theta_n\}) & = & \forall \iota \neq \jmath.\ \mathsf{rwlocked}(\theta_\iota) \cap \mathsf{wlocked}(\theta_\jmath) = \mathsf{wlocked}(\theta_\iota) \cap \mathsf{rwlocked}(\theta_\jmath) = \emptyset &
\end{array}
$$

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

Appendix C. 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	Γ	::=	$\emptyset \mid \Gamma, x : \tau$
Region context	Δ	::=	$\emptyset \mid \Delta, \rho$
Heap context	M	::=	$\emptyset \mid M, \ell \mapsto (\tau, \iota)$
Store context	R	::=	$\emptyset \mid R, \iota$
Variable list substitution	$\Gamma[r/\rho]$	::=	$\emptyset \mid \Gamma_1[r/\rho], x : \tau[r/\rho]$

Typing rules

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Auxiliary functions and predicates

$$\begin{array}{lcl}
\xi_1 - \xi_2 & = & \{r \mapsto \xi_1(r) - \xi_2(r)\} \qquad \text{where } \xi(r) = \eta \text{ if } r \mapsto \eta \in \xi \text{ and } \xi(r) = (0,0,0) \text{ otherwise} \\
\mathsf{ok}(\xi) & = & \forall \iota \mapsto \eta \in \xi.\ \mathsf{ok}(\eta) \\
\mathsf{ok}(\theta) & = & \forall \iota \mapsto (\eta, \jmath) \in \theta.\ \mathsf{ok}(\eta) \wedge \mathsf{ancestors}(\theta, \iota) \text{ defined} \\
\mathsf{hierarchy_ok}(\theta_1; \theta_2) & = & \forall \iota \mapsto (\eta, \jmath) \in \theta_1.\ \exists \jmath'.\ \iota \mapsto (\eta', \jmath') \in \theta_2 \wedge (\jmath = \jmath' \vee (\jmath = \bot \wedge \jmath' \notin \mathsf{dom}(\theta_1)))
\end{array}$$

Summary

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

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

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

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

$$\frac{\mathsf{recursive}(\xi_1; \gamma) = \xi_1 \qquad \xi_1 = \{r \mapsto (1,0,0) \mid r \in \mathsf{dom}(\gamma)\} \qquad \xi_2 = \{r \mapsto (-1,0,0) \mid r \in \mathsf{dom}(\gamma)\}}{\mathsf{summary}(\gamma) = \texttt{Cap}\,\xi_1 :: \texttt{Spawn}\,\xi_1\,(\gamma :: \texttt{Cap}\,\xi_2)} \;\textit{(SUM)}$$

Effect validation and transformation

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

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

$$\frac{\mathsf{solve}(\neg\texttt{Live}, r', \eta) = \gamma}{\mathsf{translate}(\emptyset, r, \eta, r') = \gamma} \quad \textit{(TR-E)}$$

$$\frac{r \notin \mathsf{dom}(\xi) \qquad \mathsf{translate}(\gamma, r, \eta, r') = \gamma'}{\mathsf{translate}(\texttt{Cap}\,\xi :: \gamma, r, \eta, r') = \texttt{Cap}\,\xi :: \gamma'} \quad \textit{(TR-CN)}$$

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

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

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

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

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

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

Type well-formedness

Region well-formedness

$$\frac{r \in \Delta \cup R \cup \{\bot\}}{R; \Delta \vdash r}$$

Constraint well-formedness

$$\frac{}{R; \Delta \vdash \emptyset} \qquad \frac{R; \Delta \vdash r \qquad R; \Delta \vdash \gamma}{R; \Delta \vdash \delta\, r :: \gamma} \qquad \frac{\begin{array}{c}\forall (r, \eta) \in \xi.\mathsf{ok}(\eta) \;\wedge\; R; \Delta \vdash r \\ R; \Delta \vdash \gamma_1 \qquad R; \Delta \vdash \gamma_2\end{array}}{R; \Delta \vdash \texttt{Spawn}\,\xi\,\gamma_1 :: \gamma_2}$$

$$\frac{R;\Delta \vdash \gamma_1 \quad R;\Delta \vdash \gamma_2 \quad R;\Delta \vdash \gamma_3}{R;\Delta \vdash \texttt{Join}\,\gamma_1\,\gamma_2 :: \gamma_3} \qquad \frac{\forall r \in \mathsf{dom}(\xi).\, R;\Delta \vdash r \quad R;\Delta \vdash \gamma_1}{R;\Delta \vdash \texttt{Cap}\,\xi :: \gamma_1}$$

Type well-formedness

$$\frac{R;\Delta \vdash r}{R;\Delta \vdash \texttt{Rgn}(r)} \qquad \frac{R;\Delta,\rho \vdash \tau}{R;\Delta \vdash \forall\rho.\,\tau} \qquad \frac{R;\Delta \vdash \tau \quad R;\Delta \vdash r}{R;\Delta \vdash \texttt{Ref}(\tau, r)}$$

$$\frac{R;\Delta \vdash \tau_1 \quad R;\Delta \vdash \gamma_1 \quad R;\Delta \vdash \tau_2}{R;\Delta \vdash \tau_1 \xrightarrow{\gamma_1} \tau_2} \qquad \frac{}{R;\Delta \vdash \texttt{unit}} \qquad \frac{}{R;\Delta \vdash \texttt{bool}}$$

Context well-formedness

$$\frac{R \vdash M \quad R;\Delta \vdash \Gamma \quad \bot \notin R}{\vdash R; M; \Delta; \Gamma}$$

Γ **well-formedness**

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

M **Well-formedness**

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

Appendix D. Type safety

Type Safety: Evaluation Context Typing

$$\frac{\vdash R; M; \Delta; \Gamma \quad R;\Delta \vdash \tau}{R; M; \Delta; \Gamma \vdash \Box : \tau \longrightarrow \tau \,\&\, \emptyset} \;(E0) \qquad \frac{R; M; \Delta; \Gamma \vdash E : \tau_2 \longrightarrow \tau_3 \,\&\, \gamma_2 \quad R; M; \Delta; \Gamma \vdash F : \tau_1 \longrightarrow \tau_2 \,\&\, \gamma_1}{R; M; \Delta; \Gamma \vdash E[F] : \tau_1 \longrightarrow \tau_3 \,\&\, \gamma_1 :: \gamma_2} \;(E1)$$

$$\frac{\tau \equiv \tau_1 \xrightarrow{\gamma_a} \tau_2 \quad R;\Delta \vdash \tau \quad R; M; \Delta; \Gamma \vdash e_2 : \tau_1 \,\&\, \gamma_1}{R; M; \Delta; \Gamma \vdash \Box\ e_2 : \tau \longrightarrow \tau_2 \,\&\, \gamma_1 :: \gamma_a} \;(F1) \qquad \frac{\tau \equiv \tau_1 \xrightarrow{\gamma_a} \tau_2 \quad R; M; \Delta; \Gamma \vdash v_1 : \tau \,\&\, \emptyset}{R; M; \Delta; \Gamma \vdash v_1\ \Box : \tau_1 \longrightarrow \tau_2 \,\&\, \gamma_a} \;(F2)$$

$$\frac{R; M; \Delta, \rho; \Gamma, x : \texttt{Rgn}(\rho) \vdash e_2 : \tau \,\&\, \gamma_1 \quad R;\Delta \vdash r \quad R;\Delta \vdash \tau \quad \texttt{Live}\, r :: \mathsf{translate}(\gamma_1, \rho, (1,1,0), r) = \gamma_2}{R; M; \Delta; \Gamma \vdash \texttt{newrgn}\,\rho, x\ @\ \Box\ \texttt{in}\ e_2 : \texttt{Rgn}(r) \longrightarrow \tau \,\&\, \gamma_2} \;(F3)$$

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

$$\frac{R; M; \Delta; \Gamma \vdash e_2 : \texttt{Rgn}(r) \,\&\, \gamma_2 \quad R;\Delta \vdash \tau \quad r \neq \bot}{R; M; \Delta; \Gamma \vdash \texttt{new}\ \Box\ @\ e_2 : \tau \longrightarrow \texttt{Ref}(\tau, r) \,\&\, \gamma_2 :: \texttt{Live}\, r} \;(F5)$$

$$\frac{R;\Delta \vdash r \qquad R;M;\Delta;\Gamma \vdash v : \tau \,\&\, \emptyset \qquad r \neq \bot}{R;M;\Delta;\Gamma \vdash \texttt{new}\ v\ @\ \square : \texttt{Rgn}(r) \longrightarrow \texttt{Ref}(\tau, r) \,\&\, \texttt{Live}\, r} \quad \textit{(F6)}$$

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

$$\frac{R;M;\Delta;\Gamma \vdash \texttt{loc}_\ell : \texttt{Ref}(\tau, \iota) \,\&\, \emptyset \qquad \iota \neq \bot}{R;M;\Delta;\Gamma \vdash \texttt{loc}_\ell := \square : \tau \longrightarrow \texttt{unit} \,\&\, \texttt{W}\, \iota} \quad \textit{(F8)}$$

$$\frac{\vdash R;M;\Delta;\Gamma \qquad R;\Delta \vdash \texttt{Ref}(\tau, r) \qquad r \neq \bot}{R;M;\Delta;\Gamma \vdash \texttt{deref}\ \square : \texttt{Ref}(\tau, r) \longrightarrow \tau \,\&\, \texttt{R}\, r} \quad \textit{(F9)}$$

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

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

Predicate “cvalid”

$$\frac{}{\mathsf{cvalid}(\texttt{Live}; \bot; \theta)} \quad \textit{(C-T)}$$

$$\frac{\theta = \theta', \iota \mapsto (\eta, \jmath) \qquad \mathsf{solve}(\delta, \jmath, \eta) = \emptyset}{\mathsf{cvalid}(\delta; \iota; \theta)} \quad \textit{(C-B)}$$

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

Predicate “valid”

$$\frac{}{\mathsf{xvalid}(\emptyset; \theta) = \theta} \quad \textit{(X-E)} \qquad \frac{\mathsf{ok}(\eta + \eta') \qquad \mathsf{cvalid}(\texttt{Live}\,; \iota; \theta, \iota \mapsto (\eta, \jmath)) \qquad \mathsf{xvalid}(\xi; \theta, \iota \mapsto (\eta + \eta', \jmath)) = \theta'}{\mathsf{xvalid}(\xi, \iota \mapsto \eta'; \theta, \iota \mapsto (\eta, \jmath)) = \theta'} \quad \textit{(X-S)}$$

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

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

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

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

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

Configuration typing

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

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

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

Predicate "Not stuck"

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

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

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

Multi-step evaluation

$$\frac{}{S\,; T \rightsquigarrow^0 S\,; T}\ \textit{(EM-1)} \qquad \frac{S\,; T \rightsquigarrow^n S_n; T_n \qquad S_n; T_n \rightsquigarrow S_{n+1}; T_{n+1}}{S\,; T \rightsquigarrow^{n+1} S_{n+1}; T_{n+1}}\ \textit{(EM-2)}$$

Other predicates

$$\frac{}{\emptyset - \emptyset = \emptyset}\ \textit{(DF-0)} \qquad \frac{\eta_1 \geq \eta_2 \qquad \theta_1 - \theta_2 = \theta'}{\theta_1, r \mapsto \eta_1 - \theta_2 \mapsto \eta_2 = \theta', r \mapsto \eta_1 - \eta_2}\ \textit{(DF-1)}$$

Appendix E. Proof

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 2 (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. Given $\emptyset; \emptyset; \emptyset; \emptyset \vdash e : \texttt{unit} \,\&\, \emptyset$, $S_0 = T_0 = \emptyset$ 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 5 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 31 to $R_n; M_n \vdash S_n; T_n$ implies $S_n; T_n$ is not stuck.

Lemma 5 (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 6 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 6 (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. By induction on the thread evaluation relation:

Case *E-T*: Rule *E-T* implies that $\theta; E[e] = \theta; \square[()]$, $S' = S$ and $T' = T$, $\mathsf{live}(\theta) = \emptyset$. By inversion of the configuration typing assumption we have that:

- $R; M \vdash T, \langle \theta; \square[()] \rangle$: by inversion of this derivation we have $R; M \vdash T$.
- $R; M \vdash S$
- $\mathsf{mutex}(\{\theta' \mid \langle \theta'; e' \rangle \in T, \langle \theta; \square[()] \}\rangle)$: 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 $\forall \iota \mapsto (\eta, \jmath) \in \theta.\ \iota \in R \wedge \jmath \in R \cup \{\bot\}$.
  - $R; M; \emptyset; \emptyset \vdash E[u] : \mathtt{unit} \,\&\, \gamma$: The application of Lemma 27 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 16 implies that $R; M; \emptyset; \emptyset \vdash e_1[v/x] : \tau_2 \,\&\, \gamma_a$ holds. The application of Lemma 26 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 $\forall \iota \mapsto (\eta, \jmath) \in \theta.\ \iota \in R \wedge \jmath \in R \cup \{\bot\}$.

– $R; M; \emptyset; \emptyset \vdash E[u] : \texttt{unit} \,\&\, \gamma$: The application of Lemma 27 to the typing derivation of $E[u]$ yields $R; M; \emptyset; \emptyset \vdash E : \tau_2 \longrightarrow \texttt{unit} \,\&\, \gamma_c$ 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 $R; M; \emptyset; \emptyset \vdash \texttt{fix}\ x.\ f : \tau \,\&\, \emptyset$, where τ equals $\tau_1 \xrightarrow{\gamma_a} \tau_2$. By inversion of the typing derivation of $\texttt{fix}\ x.\ f$ 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 16 implies that $R; M; \emptyset; \emptyset \vdash f[\texttt{fix}\ x.\ f/x] : \tau' \,\&\, \emptyset$ holds. The typing derivation of $f[\texttt{fix}\ x.\ f/x]$ and v and rule *T-A* imply that $R; M; \emptyset; \emptyset \vdash (f[\texttt{fix}\ x.\ f/x])\ v : \tau_2 \,\&\, \gamma_b$. The application of Lemma 26 yields $R; M; \emptyset; \emptyset \vdash E[(f[\texttt{fix}\ x.\ f/x])\ v] : \texttt{unit} \,\&\, \gamma_b :: \gamma_c$ 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 7 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 $u = \texttt{spawn}_\xi\ e_1$ 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 $\forall \iota \mapsto (\eta, \jmath) \in \theta.\ \iota \in R \wedge \jmath \in R \cup \{\bot\}$.
  – $R; M; \emptyset; \emptyset \vdash E[u] : \texttt{unit} \,\&\, \gamma$. The application of Lemma 27 to the typing derivation of $E[u]$ yields $R; M; \emptyset; \emptyset \vdash E : \texttt{unit} \longrightarrow \texttt{unit} \,\&\, \gamma_b$, $R; M; \emptyset; \emptyset \vdash \texttt{spawn}_\xi\ e_1 : \texttt{unit} \,\&\, \gamma_a$, where γ_1 is the effect of expression e_1, $\gamma_a = \texttt{Spawn}\,\xi\,\gamma_1$ 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 15 to the typing of derivation of u). The application of Lemma 26 yields $R; M; \emptyset; \emptyset \vdash E[()] : \texttt{unit} \,\&\, \gamma_b$.
  – $R; M; \emptyset; \emptyset \vdash e_1 : \texttt{unit} \,\&\, \gamma_1$. Then, by inversion of $R; M; \emptyset; \emptyset \vdash \texttt{spawn}_{\gamma_1}\ e_1 : \texttt{unit} \,\&\, \gamma_a$ we obtain that $R; M; \emptyset; \emptyset \vdash e_1 : \texttt{unit} \,\&\, \gamma_1$. The application of rules *E0* and *D0* implies that $R; M; \emptyset; \emptyset \vdash \Box[e_1] : \texttt{unit} \,\&\, \gamma_1$.

– $\mathsf{mutex}(\{\theta_0 \mid \langle \theta_0; e_0 \rangle \in T, \langle \theta; E[u] \rangle\})$: 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 ι is fresh (i.e., it does not belong in R), $\theta' = \theta, \iota \mapsto ((1,1,0), \jmath)$ $S' = S, \iota \mapsto \emptyset$, $\jmath \in \mathsf{live}(\theta) \cup \{\bot\}$ holds, $T' = S'; T, \langle \theta'; E[e_2[\iota/\rho][\mathtt{rgn}_\iota/x]] \rangle$, where u equals $\mathtt{newrgn}\, \rho, x \,@\, \mathtt{rgn}_\jmath\, \mathtt{in}\, e_2$.

By inversion of the configuration typing assumption we have that:

- $R, \iota; M \vdash S'$ is immediate from $R; M \vdash S$, the fact that ι is fresh and its heap is empty.

- $\mathsf{mutex}(\{\theta_0 \mid \langle \theta_0; e_0 \rangle \in T, \langle \theta; E[u] \rangle\})$. Hence $\mathsf{mutex}(\{\theta_0 \mid \langle \theta_0; e_0 \rangle \in T'\})$ holds, as no other thread in T has ι in its local hierarchy (i.e., even if one of ι's ancestors is locked by a thread in T, ι 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 $\forall \iota \mapsto (\eta, \jmath) \in \theta.\ \iota \in R \wedge \jmath \in R \cup \{\bot\}$.

  – $R; M; \emptyset; \emptyset \vdash E[u] : \mathtt{unit} \,\&\, \gamma$: The application of Lemma 27 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 $R; \emptyset \vdash \tau_1$, $R; M; \emptyset; \emptyset \vdash \mathtt{rgn}_\jmath : \mathtt{Rgn}(\jmath) \,\&\, \emptyset$ and $R; M; \emptyset, \rho; \emptyset, x : \mathtt{Rgn}(\rho) \vdash e_2 : \tau_1 \,\&\, \gamma_2$. Lemma 20 implies that $R, \iota; M; \emptyset, \rho; \emptyset, x : \mathtt{Rgn}(\rho) \vdash e_2 : \tau_1 \,\&\, \gamma_2$. Lemmata 19 and 16 imply that $R, \iota; M; \emptyset; \emptyset \vdash e_2[\iota/\rho][\mathtt{rgn}_\iota/x] : \tau_1 \,\&\, \gamma_2[\iota/\rho]$ (notice that $R; \emptyset \vdash \tau_1$ implies that $\tau_1[\iota/\rho] = \tau_1$). The application of Lemma 21 to the typing derivation of E implies that $R, \iota; M; \emptyset; \emptyset \vdash E : \tau_1 \longrightarrow \mathtt{unit} \,\&\, \gamma_b$. The application of Lemma 26 yields $R, \iota; M; \emptyset; \emptyset \vdash E[e_2[\iota/\rho][\mathtt{rgn}_\iota/x]] : \mathtt{unit} \,\&\, \gamma_2[\iota/\rho] :: \gamma_b$ holds.

  – by inversion of $\mathsf{valid}(\mathtt{Live}\, \jmath :: \mathsf{translate}(\gamma_2, \rho, (1,1,0), \jmath) :: \gamma_b; \theta)$ there exists a θ'' such that $\mathsf{live}(\theta'') = \emptyset$ and $\mathsf{gvalid}(\mathsf{translate}(\gamma_2; \rho; (1,1,0); \jmath) :: \gamma_b; \theta) = \theta''$. Thus, $\mathsf{valid}\ (\mathsf{translate}(\gamma_2, \rho, (1,1,0), \jmath) ::$

$\gamma_b; \theta$) holds. Hence valid (translate ($\gamma_2[\iota/\rho], \iota, (1,1,0), \jmath$) :: $\gamma_b; \theta$) holds. The translation function transforms effects containing ι. Such effects only exist in $\gamma_2[\iota/\rho]$ (this can be shown by lemma 18) and not in γ_b. Thus, valid(translate($\gamma_2[\iota/\rho]$ :: $\gamma_b, \iota, (1,1,0), \jmath); \theta$) also holds. Lemma 30 implies that 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 27 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 15 to the typing of derivation of u). The application of Lemma 26 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:

- $\mathsf{mutex}(\{\theta_0 \mid \langle \theta_0; e_0 \rangle \in T, \langle \theta; E[u] \}\})$: as no new locks are acquired, $\mathsf{mutex}(\{\theta_0 \mid \langle \theta_0; e_0 \rangle \in T, \langle \theta; E[()] \}\})$ trivially 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 $\forall \iota \mapsto (\eta, \jmath) \in \theta.\ \iota \in R \wedge \jmath \in R \cup \{\bot\}$.

– $R; M; \emptyset; \emptyset \vdash E[u] : \texttt{unit} \,\&\, \gamma$: The application of Lemma 27 to the typing derivation of $E[u]$ yields $R; M; \emptyset; \emptyset \vdash E : \texttt{unit} \longrightarrow \texttt{unit} \,\&\, \gamma_b$, $R; M; \emptyset; \emptyset \vdash \texttt{loc}_\iota := v : \texttt{unit} \,\&\, \gamma_a$, where $\gamma_a = \texttt{W}\,\iota$ 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 15 to the typing of derivation of u). The application of Lemma 26 yields $R; M; \emptyset; \emptyset \vdash E[()] : \texttt{unit} \,\&\, \gamma_b$.

– 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 $R; M; \emptyset; \emptyset \vdash \texttt{loc}_\iota := v : \texttt{unit} \,\&\, \gamma_a$ we obtain that $R; M; \emptyset; \emptyset \vdash v : \tau \,\&\, \emptyset$ and $R; M; \emptyset; \emptyset \vdash \texttt{loc}_\ell : \texttt{Ref}(\tau, \iota) \,\&\, \emptyset$ (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:

- $R; M \vdash S$
- $\mathsf{mutex}(\{\theta_0 \mid \langle \theta_0; e_0 \rangle \in T, \langle \theta; E[u] \rangle\})$: as no new locks are acquired, $\mathsf{mutex}(\{\theta_0 \mid \langle \theta_0; e_0 \rangle \in T, \langle \theta; E[f[\iota/\rho]] \rangle\})$ trivially 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 $\forall \iota \mapsto (\eta, \jmath) \in \theta.\, \iota \in R \wedge \jmath \in R \cup \{\bot\}$.
  – $R; M; \emptyset; \emptyset \vdash E[u] : \texttt{unit} \,\&\, \gamma$: The application of Lemma 27 to the typing derivation of $E[u]$ yields $R; M; \emptyset; \emptyset \vdash E : \tau[\iota/\rho] \longrightarrow \texttt{unit} \,\&\, \gamma$, $R; M; \emptyset; \emptyset \vdash (\Lambda\rho.\, f)\,[\iota] : \tau[\iota/\rho] \,\&\, \emptyset$. By inversion of the latter derivation we obtain that $R; M; \emptyset, \rho; \emptyset \vdash f : \tau \,\&\, \emptyset$ and $R; \emptyset \vdash \iota$. Lemma 19 implies that $R; M; \emptyset; \emptyset \vdash f[\iota/\rho] : \tau[\iota/\rho] \,\&\, \emptyset$. The application of Lemma 26 yields $R; M; \emptyset; \emptyset \vdash E[f[\iota/\rho]] : \texttt{unit} \,\&\, \gamma$.

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 $\texttt{if true then}\ e_1\ \texttt{else}\ e_2$.

By inversion of the configuration typing assumption we have that:

- $R; M \vdash S$
- $\mathsf{mutex}(\{\theta_0 \mid \langle \theta_0; e_0 \rangle \in T, \langle \theta; E[u] \rangle\})$: as no new locks are acquired, $\mathsf{mutex}(\{\theta_0 \mid \langle \theta_0; e_0 \rangle \in T, \langle \theta; E[e_1] \rangle\})$ trivially 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 $\forall \iota \mapsto (\eta, \jmath) \in \theta.\ \iota \in R \wedge \jmath \in R \cup \{\bot\}$.
  - $R; M; \emptyset; \emptyset \vdash E[u] : \mathtt{unit} \,\&\, \gamma$: The application of Lemma 27 to the typing derivation of $E[u]$ yields $R; M; \emptyset; \emptyset \vdash E : \tau \longrightarrow \mathtt{unit} \,\&\, \gamma_b$ and $R; M; \emptyset; \emptyset \vdash \mathtt{if\ true\ then}\ e_1\ \mathtt{else}\ e_2 : \tau \,\&\, \gamma_a$, 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 26 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 7 (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 9 implies that there exists a θ'' such that $\mathsf{gvalid}(\gamma_s; \theta) = \theta''$ and $\mathsf{gvalid}(\gamma; \theta'') = \theta'$. Lemma 10 implies that $\theta'' = \theta$ and therefore $\mathsf{gvalid}(\gamma_s; \theta) = \theta$. Lemma 11 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 $\forall \iota \mapsto (\eta, \jmath) \in \theta_{live}.\ \eta = (1, 0, 0)$. 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 12 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 13 then easily yields $\mathsf{gvalid}(\phi(\gamma_s); \theta) = \theta$, and Lemma 8 yields $\mathsf{gvalid}(\phi(\gamma_s) :: \gamma; \theta) = \theta'$. The proof is completed by rule V-V.

Lemma 8 (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 9 (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 10 (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 $\mathsf{summary}$, 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 11 (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 $\forall \iota \mapsto (\eta, \jmath) \in \theta'.\, \eta = (1, 0, 0)$.

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 $\forall \iota \mapsto (\eta, \jmath) \in \theta'.\, \eta = (1, 0, 0)$. Also, by inversion of $\mathsf{valid}(\phi(\gamma_L) :: \gamma_n :: \mathtt{Cap}\, \xi_2; \theta')$ and Lemma 9, 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 12 (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 ι such that $\mathsf{cvalid}(\mathtt{Live}\,; \iota; \theta_1)$, $\iota \mapsto (\eta, j)$ in θ_1 and $\iota \mapsto \eta'$ in $\mathsf{merge}(\xi_c)$, we know that $\mathsf{ok}(\eta{+}\eta')$ and ι is live in θ_1. Assuming that $\iota \mapsto (\eta'', j)$ exists in θ_3, then $\theta = \theta_3 - \theta_1$ implies that $\eta'' \geq \eta$ therefore $\mathsf{ok}(\eta'' + \eta')$. $\mathsf{cvalid}(\mathtt{Live}\,; \iota; \theta_1)$ and $\mathsf{live}(\theta_1) \subseteq \mathsf{live}(\theta_3)$ imply that $\mathsf{cvalid}(\mathtt{Live}\,; \iota; \theta_3)$. 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 $\forall \iota \mapsto \eta \in \xi_s.\, \mathsf{rd}(\eta) = \mathsf{wr}(\eta) = 0$. 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 $\delta\,\iota$: By inversion of $\mathsf{evalid}(\zeta; \theta_1) = \theta_5$ we have $\theta_1 = \theta_5$ and $\mathsf{cvalid}(\delta; \iota; \theta_1)$. 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 $\mathsf{cvalid}(\delta; \iota; \theta_3)$. 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 13 (Recursion preserves gvalid)
If $\gamma_L = \{\texttt{Live}\, r \mid r \in \mathsf{dom}(\phi(\emptyset))\}$, $\gamma_s = \mathsf{summary}(\phi(\gamma_L))$, $\mathsf{gvalid}(\gamma_s;\theta_0) = \theta_0$, $\mathsf{hierarchy\_ok}(\theta_1;\theta_0)$, 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 9 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 14 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 8 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 perfoming a case analysis on the structure of ψ:

Case $\psi(\gamma) = \texttt{Cap}\,\xi$ or $\psi(\gamma) = \delta\,\iota$: The proof is immediate, as $\psi(\gamma_s) = \psi(\gamma_L)$.

Case $\psi(\gamma) = \texttt{Spawn}\,\xi_1\,\phi'''(\gamma)$: 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 $\psi(\gamma) = \texttt{Join}\,\phi_1(\gamma)\,\phi_2(\gamma)$: 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 14 (Preservation of gvalid for a smaller θ)
If $\gamma_L = \{\texttt{Live}\,r \mid r \in \mathsf{dom}(\phi(\emptyset))\}$, $\gamma_s = \mathsf{summary}(\phi(\gamma_L))$, $\mathsf{gvalid}(\gamma_s; \theta) = \theta$, $\mathsf{hierarchy\_ok}(\theta'; \theta)$, $\mathsf{ok}(\theta')$, and $\mathsf{dom}(\gamma_s) \subseteq \mathsf{live}(\theta')$, then $\mathsf{gvalid}(\gamma_s; \theta') = \theta'$.

Proof. Immediate by the definition of function $\mathsf{summary}$ 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 15 (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 16 (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 17 (Well-typed expressions have well-formed types)
If $R; M; \Delta; \Gamma \vdash e : \tau \,\&\, \gamma$ then $R; \Delta \vdash \tau$.

Proof. Straightforward induction on the typing rules.

Lemma 18 (Well-typed expressions have well-formed effects)
If $R; M; \Delta; \Gamma \vdash e : \tau \,\&\, \gamma$ then $R; \Delta \vdash \gamma$.

Proof. Straightforward induction on the typing rules.

Lemma 19 (Region substitution preserves typing)
If $R, \iota; M; \Delta, \rho; \Gamma \vdash e : \tau \,\&\, \gamma$, then $R, \iota; M; \Delta; \Gamma[\iota/\rho] \vdash e[\iota/\rho] : \tau[\iota/\rho] \,\&\, \gamma[\iota/\rho]$.

Proof. Proof by induction on the typing derivation of e.

Lemma 20 (Region context expansion preserves expression typing)
If $R; M; \Delta; \Gamma \vdash e : \tau \,\&\, \gamma$ and $\iota \notin R \cup \{\bot\}$, then $R, \iota; M; \Delta; \Gamma \vdash e : \tau \,\&\, \gamma$.

Proof. Proof by induction on the typing derivation of e.

Lemma 21 (Region context expansion preserves evaluation context typing)
If $R; M; \Delta; \Gamma \vdash E : \tau \longrightarrow \tau' \,\&\, \gamma$ and $\iota \notin R \cup \{\bot\}$, then $R, \iota; M; \Delta; \Gamma \vdash E : \tau \longrightarrow \tau' \,\&\, \gamma$.

Proof. Proof by induction on the derivation of E. In the case of rule *E1*, where $E = E'[F]$, Lemma 22 is used.

Lemma 22 (Region context expansion preserves frame typing)
If $R; M; \Delta; \Gamma \vdash F : \tau \longrightarrow \tau' \,\&\, \gamma$ and $\iota \notin R \cup \{\bot\}$, then $R, \iota; M; \Delta; \Gamma \vdash F : \tau \longrightarrow \tau' \,\&\, \gamma$.

Proof. Proof by induction on the derivation of F.

Lemma 23 (Memory context expansion preserves expression typing)
If $R; M; \Delta; \Gamma \vdash e : \tau \,\&\, \gamma$, $\ell \notin \mathsf{dom}(M)$, $\iota \in R$ and $R; \Delta \vdash \tau$, then $R; M, \ell \mapsto (\tau, \iota); \Delta; \Gamma \vdash e : \tau \,\&\, \gamma$.

Proof. Proof by induction on the typing derivation of e.

Lemma 24 (Memory context expansion preserves evaluation context typing)
If $R; M; \Delta; \Gamma \vdash E : \tau \longrightarrow \tau' \,\&\, \gamma$, $\ell \notin \mathsf{dom}(M)$, $\iota \in R$ and $R; \Delta \vdash \tau$, then $R; M, \ell \mapsto (\tau, \iota); \Delta; \Gamma \vdash E : \tau \longrightarrow \tau' \,\&\, \gamma$.

Proof. Proof by induction on the derivation of E. In the case of rule *E1*, where $E = E'[F]$, Lemma 25 is used.

Lemma 25 (Memory context expansion preserves frame typing)
If $R; M; \Delta; \Gamma \vdash F : \tau \longrightarrow \tau' \,\&\, \gamma$, $\ell \notin \mathsf{dom}(M)$, $\iota \in R$ and $R; \Delta \vdash \tau$, then $R; M, \ell \mapsto (\tau, \iota); \Delta; \Gamma \vdash F : \tau \longrightarrow \tau' \,\&\, \gamma$.

Proof. Proof by induction on the derivation of F.

Lemma 26 (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 28.

Lemma 27 (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 15 and 17) and the application rule *E0*. The inductive case, where $E[e] = E'[F][e]$ is immediate by Lemma 29 and rule *E1*.

Lemma 28 (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 29 (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 *F1F11*) are given by the premises of the typing derivation of $F[e]$.

Lemma 30 (Translate implies valid)
If $\mathsf{valid}(\mathsf{translate}(\gamma, \iota, \eta), \jmath); \theta)$ and $\iota \notin \mathsf{dom}(\theta)$, then $\mathsf{valid}(\gamma; \theta, \iota \mapsto (\eta, \jmath))$.

Proof. Let θ' be equal to $\theta, \iota \mapsto (\eta, \jmath)$. We proceed by induction on the structure of γ.

Case $\gamma = \emptyset$: the assumption and the definition of function $\mathsf{translate}$ imply that $\mathsf{valid}(\mathsf{translate}(\emptyset, \iota, \eta, \jmath); \theta)$ holds if and only if $\mathsf{valid}(\gamma_s; \theta)$ holds, where $\gamma_s = \mathsf{solve}(\neg\texttt{Live}\,; \jmath; \eta)$. 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 $\gamma_a = \mathsf{translate}(\gamma_1 :: \gamma_3, \iota, \eta, \jmath)$ and $\gamma_a = \mathsf{translate}(\gamma_2 :: \gamma_3, \iota, \eta, \jmath)$. 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 $\theta, \iota \mapsto (\eta + \eta_0, \jmath)$. The assumption and the definition of translate imply that $\mathsf{merge}(\xi) = \xi', \iota \mapsto \eta_0$, $\mathsf{valid}(\gamma_s :: \gamma_b; \theta)$, where $\gamma_s = \mathsf{solve}(\texttt{Live}, \jmath, \eta) :: \texttt{Cap}\,\xi'$ and $\gamma_b = \mathsf{translate}(\gamma_a, \iota, \eta + \eta_0, \jmath)$. 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 $\iota \notin \mathsf{dom}(\theta)$. $\mathsf{cvalid}(\texttt{Live}\,; \iota; \theta'')$ 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 $\gamma = \delta\,\iota :: \gamma_a$: 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 $\gamma_b = \mathsf{translate}(\gamma_a, \iota, \eta, \jmath)$. To complete the proof it suffices to show that $\mathsf{cvalid}(\delta\,; \iota; \theta')$ and $\mathsf{valid}(\gamma_a; \theta')$ hold. We proceed by a case analysis on γ_s. If γ_s is empty, then $\mathsf{cvalid}(\delta; \iota; \theta')$ is immediate by using rule *C-B* and the definition of γ_s. Otherwise, γ_s is non-empty and by the definition 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 $\mathsf{cvalid}(\delta; \iota; \theta')$. $\mathsf{valid}(\gamma_a; \theta')$ is immediate by applying the induction hypothesis to $\mathsf{valid}(\gamma_b; \theta)$, which can be derived from $\mathsf{valid}(\gamma; \theta)$.

Case $\gamma = \texttt{Spawn}\,\xi\,\gamma_s :: \gamma_a$: the assumption and the definition of translate imply that $\mathsf{valid}(\gamma_r''' :: \texttt{Spawn}\,\xi'\,(\gamma_s' :: \gamma_s'')\,(\gamma_r' :: \gamma_r''))$, where $\gamma_r''' = \mathsf{bot}(\texttt{Live}, \jmath)$,

$\mathsf{merge}(\xi) = \xi', \iota \mapsto \eta_s$, $\eta = \eta_r \oplus \eta_s$, $\jmath' = \text{if } \jmath \in \mathsf{dom}(\xi) \text{ then } \jmath \text{ else } \bot$, $\gamma'_s = \mathsf{p\text{-}constraint}(\jmath', \eta_r)$, $\gamma''_s = \mathsf{translate}(\gamma_s, \iota, \eta_s, \jmath')$, $\gamma'_r = \mathsf{p\text{-}constraint}(\jmath, \eta_s)$ and $\gamma''_r = \mathsf{translate}(\gamma_a, \iota, \eta_r, \jmath)$.

By inversion of $\mathsf{valid}(\gamma'''_r :: \mathtt{Spawn}\,\xi'\,(\gamma'_s :: \gamma''_s) :: (\gamma'_r :: \gamma''_r))$ we have that $\mathsf{cvalid}(\mathtt{Live}\,; \jmath; \theta)$ (if $\jmath \neq \bot$), $\forall \iota' \in \mathsf{dom}(\theta_s).\ \mathsf{cvalid}(\mathtt{Live}\,; \iota'; \theta)$, $\mathsf{merge}(\xi') \vdash \theta = \theta_r \oplus \theta_s$, $\mathsf{mutex}(\{\theta_r, \theta_s\})$, $\mathsf{valid}(\gamma'_s :: \gamma''_s; \theta_s)$ 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:

- $\forall \iota' \in \mathsf{dom}(\theta_s, \iota \mapsto (\eta_s, \jmath')).\ \mathsf{cvalid}(\mathtt{Live}\,; \iota'; \theta, \iota \mapsto (\eta, \jmath))$: immediate by the assumptions $\mathsf{cvalid}(\mathtt{Live}\,; \jmath; \theta)$ (if $\jmath \neq \bot$), $\eta = \eta_r \oplus \eta_s$ and $\forall \iota' \in \mathsf{dom}(\theta_s).\ \mathsf{cvalid}(\mathtt{Live}\,; \iota'; \theta)$.
- $\mathsf{merge}(\xi) \vdash \theta, \iota \mapsto (\eta, \jmath) = \theta_r, \iota \mapsto (\eta_r, \jmath) \oplus \theta_s, \iota \mapsto (\eta_s, \jmath')$: $\mathsf{merge}(\xi) = \xi', \iota \mapsto \eta_s$ 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.
- $\mathsf{mutex}(\{(\theta_r, \iota \mapsto (\eta_r, \jmath)), (\theta_s, \iota \mapsto (\eta_s, \jmath'))\})$: if at least one of the threads has read or write capabilities on ι, then the mutex invariant may be violated once ι is added to the hierarchy of each thread. Assuming that the new thread has read or write access to ι, 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 ι and its ancestors ancestors respectively. Similarly, if the main thread has read or write access to ι, 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 ι and its ancestors respectively. We also have from the assumptions that $\mathsf{mutex}(\{\theta_r, \theta_s\})$. Therefore, $\mathsf{mutex}(\{(\theta_r, \iota \mapsto (\eta_r, \jmath)), (\theta_s, \iota \mapsto (\eta_s, \jmath'))\})$ holds.
- $\mathsf{valid}(\gamma_a; \theta_r, \iota \mapsto (\eta_r, \jmath))$: 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.

- $\mathsf{valid}(\gamma_s; \theta_s, \iota \mapsto (\eta_s, \jmath'))$: similar to the previous case; here we use $\mathsf{valid}(\gamma_s' :: \gamma_s''; \theta_s)$.

Lemma 31 (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. 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 36, there exists a redex u and an evaluation context E such that $e = E[u]$. The application of Lemma 27 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 perfoming a case analysis on u:

Case $(\lambda x.\, e')\ v$: a step can be taken by rule *E-A*.

Case $(\Lambda \rho.\, f)\,[\iota]$: a step can be taken by rule *E-RP*.

Case $(\mathtt{fix}\ x.\, f)\ v$: a step can be taken by rule *E-FX*.

Case $\mathtt{if\ true\ then}\ e_1\ \mathtt{else}\ e_2$: a step can be taken by rule *E-IT*.

Case $\mathtt{if\ false\ then}\ e_1\ \mathtt{else}\ e_2$: a step can be taken by rule *E-IF*.

Case $\mathtt{newrgn}\ \rho, x\ @\ \mathtt{rgn}_\jmath\ \mathtt{in}\ e_2$: it suffices to prove $\jmath \in \mathsf{live}(\theta) \cup \{\bot\}$. The typing derivation of u implies $\gamma_a = \mathtt{Live}\ \jmath :: \mathsf{translate}(\gamma_2, \rho, (1, 1, 0), \jmath)$, where γ_2 is the effect of e_2. The application of Lemma 33 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 $\mathtt{new}\ v\ @\ \mathtt{rgn}_\iota$: identical to the previous case. Rule *E-NL* can be applied to perform a step.

Case $\mathtt{deref}\ \mathtt{loc}_\ell$: it suffices to prove $\iota \in \mathsf{rwlocked}(\theta)$, $\ell \mapsto v \in S(\iota)$ and $\iota \notin \mathsf{wlocked}(T)$. $\ell \mapsto v \in S(\iota)$ is immediate by $R; M \vdash S$ and the fact that $\ell \mapsto (\tau, \iota)$ belongs in M by the typing derivation of u. The typing derivation of u also implies $\gamma_a = \mathtt{R}\,\iota$. Lemma 34 and $\mathsf{valid}(\gamma_a :: \gamma'; \theta)$ imply that

$\iota \in \mathsf{rwlocked}(\theta)$. We also have the assumption that $\mathsf{mutex}(\{\theta_0 \mid \langle \theta_0; e_0 \rangle \in T\})$, which implies $\iota \notin \mathsf{wlocked}(T) = \emptyset$. Rule *E-D* can be applied to perform a step.

Case $\mathtt{loc}_\iota := v$: it suffices to prove $\ell \mapsto v' \in S(\iota)$, $\iota \in \mathsf{wlocked}(\theta)$ and $\iota \notin \mathsf{rwlocked}(T)$. $\ell \mapsto v' \in S(\iota)$ is immediate by $R; M \vdash S$ and the fact that $\ell \mapsto (\tau, \iota)$ belongs in M by the typing derivation of u. The typing derivation of u also implies $\gamma_a = \mathtt{W}\,\iota$. Lemma 35 and $\mathsf{valid}(\gamma_a :: \gamma'; \theta)$ imply that $\iota \in \mathsf{wlocked}(\theta)$. We also have the assumption that $\mathsf{mutex}(\{\theta_0 \mid \langle \theta_0; e_0 \rangle \in T\})$, which implies $\iota \notin \mathsf{rwlocked}(T)$. Rule *E-AS* can be applied to perform a step.

Case $\mathtt{cap}_{\eta'}\ \mathtt{rgn}_\iota$: given that $\theta = \theta_1, \iota \mapsto (\eta', r')$ it suffices to prove $\theta' = \theta_1, \iota \mapsto (\eta + \eta', r')$, $\iota \in \mathsf{live}(\theta)$ and $\mathsf{mutex}(\{\theta'\} \cup \{\theta'' \mid \langle \theta''; e' \rangle \in T\})$ or $\mathsf{blocked}(T; \theta; E[\mathtt{cap}_{\eta'}\ \mathtt{rgn}_\iota])$. The typing derivation of u implies $\gamma_a = \mathtt{Cap}\,\{\iota \mapsto \eta\}$. Then, Lemma 33 and $\mathsf{valid}(\gamma_a :: \gamma'; \theta)$ imply that $\iota \in \mathsf{live}(\theta)$. 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, $\mathsf{blocked}(T; \theta; E[\mathtt{cap}_{\eta'}\ \mathtt{rgn}_\iota])$ holds using the assumption $\mathsf{mutex}(\{\theta_0 \mid \langle \theta_0; e_0 \rangle \in T\})$.

Case $\mathtt{spawn}_{\gamma_1}\ e_1$: 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 $\gamma_a = \mathtt{Spawn}\,\xi\,\gamma_1$. By inversion of $\mathsf{valid}(\gamma_a :: \gamma'; \theta)$ we have that $\mathsf{merge}(\xi) \vdash \theta = \theta' \oplus \theta''$ and $\forall \iota \in \mathsf{dom}(\theta'').\ \mathsf{cvalid}(\mathtt{Live}; \iota; \theta)$ hold. The latter fact and Lemma 32 imply that $\mathsf{dom}(\theta'') \subseteq \mathsf{live}(\theta)$. Rule *E-SP* can be applied to perform a single step.

Lemma 32 (Cvalid implies live, wlocked and rwlocked)
If $\mathsf{cvalid}(\delta, \iota, \theta)$ and $\delta \neq \neg\mathtt{Live}$ then $\iota \in \mathsf{live}(\theta) \cup \{\bot\}$, $\delta = \mathtt{W} \Rightarrow \iota \in \mathsf{wlocked}(\theta)$ and $\delta = \mathtt{R} \Rightarrow \iota \in \mathsf{rwlocked}(\theta)$.

Proof. We perform case analysis on cvalid derivation:

Case *C-T*: the proof is immediate.

Case *C-B* then the following hold $\theta = \theta', \iota \mapsto (\eta, \jmath)$ and $\mathsf{solve}(\delta, \jmath, \eta) = \emptyset$. We have assumed that $\delta \neq \neg\mathtt{Live}$, therefore $\mathsf{solve}(\delta, \jmath, \eta) = \emptyset$ implies that $\jmath = \bot$ and $\mathsf{ok}(\eta - (1, 0, 0))$. Thus $\iota \in \mathsf{live}(\theta)$ holds. If $\delta = \mathtt{W}$, then solve also implies that $\mathsf{rw}(\eta) > 0$ hence $\iota \in \mathsf{wlocked}(\theta)$. If $\delta = \mathtt{R}$, then solve also implies that $\mathsf{rw}(\eta) \geq 0$, $\mathsf{rw}(\eta) \geq 0$ and $\mathsf{rw}(\eta) + \mathsf{rw}(\eta) > 0$ hence $\iota \in \mathsf{rwlocked}(\theta)$.

Case *C-R* then $\theta = \theta', \iota \mapsto (\eta, \jmath)$, $\mathsf{solve}(\delta, \jmath, \eta) = \delta'\ \jmath$ 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 `Live` if δ equals `Live`. If δ is equal to `R`, then δ' can be either `R` and $\mathsf{rd}(\eta) = 0$ or `Live` when $\mathsf{rd}(\eta) > 0$. If δ is equal to `W`, then δ' can be either `W` and $\mathsf{rd}(\eta) = \mathsf{wr}(\eta) = 0$ or `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 $\jmath \in \mathsf{live}(\theta) \cup \{\bot\}$, $\delta' = \texttt{W} \Rightarrow \jmath \in \mathsf{wlocked}(\theta)$ and $\delta' = \texttt{R} \Rightarrow \jmath \in \mathsf{rwlocked}(\theta)$.

$\mathsf{ok}(\eta - (1,0,0))$ and $\jmath \in \mathsf{live}(\theta) \cup \{\bot\}$ imply that $\iota \in \mathsf{live}(\theta)$. If δ is `Live` the proof is completed. Otherwise, if $\delta' = \delta = \texttt{W}$, then $\jmath \in \mathsf{wlocked}(\theta)$ implies $\iota \in \mathsf{wlocked}(\theta)$ and the proof is completed. Otherwise, if $\delta' = \delta = \texttt{R}$, then $\jmath \in \mathsf{rwlocked}(\theta)$ implies $\iota \in \mathsf{rwlocked}(\theta)$ and the proof is completed. The last case is $\delta' = \texttt{Live}$. 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 33 (Valid implies live)
If $\mathsf{valid}(\delta\, \iota :: \gamma; \theta)$ and $\delta \neq \neg\texttt{Live}$ then $\iota \in \mathsf{live}(\theta) \cup \{\bot\}$.

Proof. By inversion of valid we obtain $\mathsf{cvalid}(\delta, \iota, \theta)$. The proof is immediate by using Lemma 32.

Lemma 34 (Valid implies rwlocked)
If $\mathsf{valid}(\texttt{R}\, r :: \gamma; \theta)$, then $r \in \mathsf{rwlocked}(\theta)$.

Proof. By inversion of valid we obtain $\mathsf{cvalid}(\texttt{R}, \iota, \theta)$. The proof is immediate by using Lemma 32.

Lemma 35 (Valid implies wlocked)
If $\mathsf{valid}(\texttt{W}\, r :: \gamma; \theta)$, then $r \in \mathsf{wlocked}(\theta)$.

Proof. By inversion of valid we obtain $\mathsf{cvalid}(\texttt{W}, \iota, \theta)$. The proof is immediate by using Lemma 32.

Lemma 36 (Well-typed expressions contain a well-typed redexes)
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'.