A Type System for Race-free and Memory-safe Multithreading Using Region Hierarchies

Prodromos Gerakios Nikolaos Papaspyrou Konstantinos Sagonas

School of Electrical and Computer Engineering
National Technical University of Athens, Greece

{pgerakios,nickie,kostis}@softlab.ntua.gr

Abstract

A current challenge for programming language research is to design and implement multithreaded low-level languages providing static guarantees for memory safety and freedom from data races. Towards this goal, in this paper we present such a language and its type system. Our language guarantees memory safety by employing region-based memory management. Unlike other similar languages employing regions, our regions are organized in a hierarchical manner so that each region is *owned* by a single parent region and may contain multiple children regions. This structure imposes constraints over region capability manipulation. In this setting, we allow early region deallocation and thus overcome the restrictions of LIFO region lifetimes. Our type system combines *fractional, aliasable* capabilities and *hierarchy abstraction* in a multithreaded setting. Regions may remain thread-local, migrate to another thread or become shared between threads. Implicit locks are used to protect each region from data races. Both lock and region capabilities are treated uniformly. Locking also follows a hierarchical scheme as a parent lock subsumes children locks.

Keywords: Type systems, regions, memory management, safety, data race prevention, multi-threaded programming, types and effects

1 Introduction

Writing safe and robust code is a hard task; writing safe and robust multithreaded low-level code is even harder. In contrast to high-level languages, low-level languages such as C allow for explicit memory management and precise control over data representations in memory. These features are often invaluable for so-called *systems programming*, but at the same time open the door for memory safety violations such as dangling pointer dereferences. Memory safety in such languages can be obtained via *region-based memory management* [Tofte and Talpin, 1994, 1997]. A region can be thought of as a segment of memory. The key idea of region-based memory management is that each data object is allocated in some region. When a region is deallocated, all its objects are deallocated simultaneously. Region-based memory management has numerous benefits. The programmer has explicit control of the location and lifetime of memory objects as it is possible to allocate an object in any live region. In some languages it is also possible to perform early region deallocation. Moreover, it is more efficient to allocate objects in an existing memory segment or deallocate objects from it, rather than requesting a new memory area for each object individually. Lexically-scoped regions must be deallocated at the end of the block in which they were introduced and therefore such regions also protect against memory leaks.

So far, regions have been adopted by numerous languages. In the area of low-level memory-safe languages, regions have been a key feature of the programming language Cyclone [Grossman et al., 2002]. Cyclone offers multiple kinds of regions which can be roughly categorized as lexically-scoped and reference-counted. It is worth noting that reference-counted regions have the same behaviour when *opened* within a block scope as lexically-scoped regions, but when they are *closed* it is possible to manipulate an implicit region reference count. Cyclone's type system does not statically track reference counts, so a dynamic check is performed when opening or manipulating the reference count of such regions. Grossman [2003] has presented a type system that prevents data races in a multithreaded extension of Cyclone, but his proposal was never fully integrated in Cyclone's implementation. As a result, Cyclone does not guarantee safety for multithreaded programs.

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

Up to now, extensive research has taken place on how type systems employing locking primitives can be used to provide both *data race freedom* [Grossman, 2003, Flanagan and Abadi, 1999] and *atomicity* [McCloskey et al., 2006, Hicks et al., 2006, Flanagan et al., 2008]. Lock-based type systems for low-level languages have focused in embedding type-level lock names into reference types as well as using singleton lock types parameterized by lock names to track *run-time locks* [Hicks et al., 2006]; a type and effect system statically tracks the acquired linear set of lock names at each program point. Dangling pointer dereferences in concurrent programs can be prevented by using *implicit reference counts* for each memory block [Grossman, 2003].

In this paper, we present a first-order region-polymorphic capability-based calculus, that allows for safe early region deallocation, offers primitives that enable unscoped locking, and guarantees the absence of data races in the presence of multiple threads. In this calculus, there exist no explicit lock handles. Instead, each region is protected by an implicit lock. The main novelty of our type system is that it allows a tree-like hierarchy of regions and locks. Furthermore, it allows safe capability aliasing in the presence of threads using explicit capability consumption primitives. Our type system supports capability *fork and join operations* [Boyland, 2003]. It also provides additional abstraction mechanisms such as *capability subtyping*, *hierarchy abstraction* [Fahndrich and DeLine, 2002] and *bounded existential types* [Smith et al., 2000]. In contrast to other type systems, we allow regions to switch state from *thread-local* to *sharable*. Thread-local regions are allowed to *migrate* to another thread and preserve the right to access data without requiring locking.

The key idea of a region hierarchy is that a region can only be deallocated when its children have been deallocated. Locking is dual to deallocation: a region can only be locked when its children have been locked. Therefore, we apply the idea of *ownership* of Boyapati et al. [2003] to regions and locks. At compile time, a *capability-based* type system [Walker et al., 2000] verifies that a program respects the above invariant. Our type system provides the means to safely alias a capability an unlimited number of times within a certain scope, in the presence of threads. A region can also be temporarily unadopted from the entire region hierarchy, thus allowing to write generic functions.

Outline. The rest of this paper is structured as follows: The next section provides an overview of our language and Section 3 describes its operational semantics. Section 4 builds a type system on top of the operational semantics which guarantees memory safety and the absence of data races. The type system also enforces the ownership invariants for locks and regions as described in this section. Section 5 presents progress and preservation lemmata and the main type safety theorem; proof sketches can be found in the appendix. Section 6 presents related work. Finally, Section 7 ends the paper with some concluding remarks.

2 Language

Before proceeding to a detailed description of our language and its semantics, we provide a number of examples illustrating its main features. It should be noted that our language is low-level and is not meant to be used directly by programmers. Instead, we expect it to be used as a compilation target for a syntactically much more user-friendly language (which we do not present here).

It should also be noted that, for presentation purposes, in the examples that follow we often simplify the type system, e.g. by omitting some input/output effects. We use also more intuitive syntax, e.g. `spawn`, which can be trivially translated to our language constructs. In a few cases we use language extensions, e.g. List.

2.1 The Main Language Features through Examples

The first example shows the hierarchical organization of regions and how regions, which start their life as local to a thread, can migrate or become shared between threads. It also shows how capabilities are being transfered or consumed during execution.

Example 1 (Region hierarchy and capability flow) Assume that we have functions whose types are as shown below:

$$\begin{array}{ll} \mathsf{create\_list} & : \forall \rho.\, \mathtt{rgn}(\rho) \to \mathsf{List}\,[\rho] \\ \mathsf{thread\_fun}_1 & : \forall \rho.\, \langle \mathtt{rgn}(\rho) \times \mathsf{List}\,[\rho] \rangle \to \langle\rangle \\ \mathsf{thread\_fun}_2 & : \forall \rho.\, \langle \mathtt{rgn}(\rho) \times \mathsf{List}\,[\rho] \rangle \to \langle\rangle \end{array}$$

Notice that the functions are *region polymorphic*.

Function create_list takes a *region handle* and returns a list allocated in this region. Similarly, thread_fun$_1$ and thread_fun$_2$ take a pair, whose first element is a region handle and its second element is a list located in this region. We can then define the following function:

```
def par_comp = $\Lambda\rho.\lambda x.$
    newrgn $\rho_1, x_1$ at $x$ in
    newrgn $\rho_2, x_2$ at $x$ in
        let $l_1$ = call create_list $[\rho_1]$ $x_1$ in
        spawn thread_fun$_1$ $[\rho_1]$ $(x_1, l_1)$;
        let $l_2$ = call create_list $[\rho_2]$ $x_2$ in
        ldec $x_2$; rinc $x_2$;
        spawn thread_fun$_2$ $[\rho_2]$ $(x_2, l_2)$;
        call thread_fun$_2$ $[\rho_2]$ $(x_2, l_2)$
```

Function par_comp is a region polymorphic function, which takes a region handle x and returns the unit value. Using the `newrgn` construct, two regions ρ_1 and ρ_2 are allocated within region ρ. The handles of these two regions are x_1 and x_2. By invoking function create_list, a list is then allocated in region ρ_1.

Each region name is associated with two distinct type-level reference counts, namely *lock and region capabilities*. A region capability is a type-level count indicating whether a region is live. A lock capability indicates whether a region is *thread-local* or *sharable*. When *sharable*, the lock capability indicates whether the associated region has been locked (i.e., its reference count is greater than zero) or unlocked.

Region ρ_1 starts its life with a thread-local lock capability and a single region capability. Subsequently, passing the handle of region ρ_1 with a thread-local lock capability to the `spawn` operation has two implications. First, the obligation for deallocating ρ_1 is delegated to the spawned thread. That is, *all* region capabilities are consumed by the spawned thread. Second, the thread-local lock capability is also transfered to the spawned thread, which can now access ρ_1 without requiring locking. Therefore, ρ_1 *migrates* from the spawning to the spawned thread.

In the following `let`, a new list is allocated at region ρ_2 and bound to a variable l_2 for the scope of the remaining expressions. We wish to share list l_2 between this thread and another thread. Sharing a region implies that access to that region is maintained once the spawn operation has completed. Therefore, we certainly must avoid migration and also keep ρ_2 *alive* for this thread.

It is worth noting that migration of a region to a spawned thread implies that it will become *dead* (i.e., inaccessible) for the spawning thread. However, it is possible for a region to become dead after a spawn operation without migrating. This is because we distinguish between thread-local lock capability and region capability delegation at once (*migration*) and strict region capability delegation (i.e., non-zero reference-counted lock capabilities cannot be passed to another thread).

The expression `ldec` x_2 allows the spawning thread to yield the *local access* right by transforming the lock capability of ρ_2 from *thread-local* to *sharable-unlocked*. Thus, we have established that ρ_2 will not migrate.

The second `spawn` will definitely consume *at least one* capability.[1] Therefore, we need to establish that ρ_2 will remain *live* after the `spawn` operation. Expression `rinc` x_2 provides the spawning thread with an additional *region capability* for region ρ_2 so that ρ_2 will remain live after the `spawn` operation.

The last expression is an ordinary function application, which applies $\mathsf{thread\_fun}_2$ to a pair consisting of the region handle to ρ_2 as well as the variable l_2. There exists no explicit region decrement command after invoking $\mathsf{thread\_fun}_2$ for deallocating ρ_2. This is because $\mathsf{thread\_fun}_2$ will definitely deallocate ρ_2. This fact can be deduced by the preceding spawn operation: for a function to typecheck, it must definitely deallocate all regions passed to it. Function $\mathsf{thread\_fun}_2$ is such an example.

The following example illustrates how reference-counted capabilities can be used to resolve the unsoundness caused by the interaction between region polymorphism and explicit capability consumption. Furthermore, we discuss how this approach allowed us to unify `spawn` and the ordinary `call` operation. Finally, this approach exposes and controls information, often encoded implicitly in the run-time system, in a type-safe manner.

Example 2 (Linearity, polymorphism and capability forking) The key invariant that must be enforced in a capability-based calculus is to preserve *capability linearity*. In systems without support for early region deallocation, region capabilities may be freely aliased within function calls as the capability will remain linear upon return to the calling context. In the presence of early region deallocation, region polymorphism is unsound.

Consider the $\mathsf{deallocate\_both}$ function, which deallocates both regions ρ_1 and ρ_2, whose type is shown below.

$$\mathsf{deallocate\_both} : \forall\rho_1.\forall\rho_2.\ \langle \mathtt{rgn}(\rho_1) \times \mathtt{rgn}(\rho_2)\rangle \to \langle\rangle$$

[1]The number of capabilities that will be consumed are explicitly specified in the thread function type as an *effect*. However, we are not dealing with effects in this example, so we shall assume that the thread consumes exactly one capability.

It is unsound to instantiate ρ_1 and ρ_2 with the same region ρ and then invoke the function with the same region handle, as it will attempt to deallocate the same region twice:

```
newrgn $\rho, x_2$ at $x_1$ in
    call deallocate_both $[\rho]\ [\rho]\ (x_2, x_2)$
```

The above example will be rejected by our type system as ρ is created with a *single region capability*, whereas function deallocate_both $[\rho]\ [\rho]$ attempts to consume two region capabilities from the same region ρ.

To allow safe region polymorphism in the presence of early region deallocation, we use *reference-counted* capabilities, which can be passed to a polymorphic function, behaving as distinct capabilities, and *joined* upon return to the calling context.

Henceforth, we use the term *fractional* to describe a capability which has been derived by a *pure* (i.e., non-fractional) capability. A fractional capability is generated when a pure capability is aliased within a function call. When a new region is allocated, both its lock and region capabilities are pure.

The following example typechecks as ρ starts (`newrgn` construct) with a single region capability, and `rinc` provides ρ with an additional capability. Therefore, its pure region capability is two. The function call aliases the pure region capability of ρ so the type system splits ρ into two fractional region capabilities. Upon return to the calling context, the remaining fractional capabilities are joined to form a pure capability.

```
newrgn $\rho, x_2$ at $x_1$ in
    rinc $x_2$;
    call deallocate_both $[\rho]\ [\rho]\ (x_2, x_2)$
```

Reference-counted capabilities allow us to treat spawn operations as ordinary function calls: A spawn operation takes a function, which accepts fractional or pure capabilities and is obliged to consume those capabilities by the end of its execution. Therefore, the type system does not distinguish between thread and ordinary functions: thread functions can be invoked by a `call` operation or a `spawn` operation. This is also reflected upon our run-time system, which in contrast to other approaches, does not need to perform implicit accounting operations before spawning a new thread.

The bad interaction between region polymorphism and capability consumption also applies to lock capabilities. In our calculus,we allow primitives that enable *unscoped locking*: When a lock ℓ_1 is acquired before another lock ℓ_2, then ℓ_1 may be released before ℓ_2 is released.

In the presence of region polymorphism, it is possible to write a function unlock_fst_access_snd that accepts two region handles and assumes (in an effect annotation, which has been elided in the following example) that the two regions ρ_1 and ρ_2 are locked (i.e., have a single lock capability).

$$\text{unlock\_fst\_access\_snd} : \forall\rho_1.\forall\rho_2.\langle \texttt{rgn}(\rho_1) \times \texttt{rgn}(\rho_2)\rangle \rightarrow \langle\rangle$$

As the name of the above function suggests, its body releases the first lock handle and accesses data allocated in the second handle ρ_2, as ρ_2 is already locked. This is unsound with respect to data race freedom, when this function is instantiated with the same region name and passed the same region handle:

```
                                                  {* $x_2$ is a handle to a locked region *}
call unlock_fst_access_snd $[\rho]\ [\rho]\ (x_2, x_2)$
```

As in the region deallocation example, this example will also be rejected by our type system as the function call requires two capabilities and ρ has a single lock capability. To preserve soundness, we have adapted the counter-based capability approach for lock capabilities. The following example will typecheck as we produce an additional lock capability for ρ before invoking function unlock_fst_access_snd.

{* x_2 is a handle to a locked region *}

$\texttt{linc}\ x_2;$
$\texttt{call}\ \mathsf{unlock\_fst\_access\_snd}\ [\rho]\ [\rho]\ (x_2, x_2)$

Example 3 (Capability subtyping) On the one hand, reference-counted capabilities resolve the bad interaction between polymorphism and early deallocation or unscoped locking. On the other hand, reference-counted capabilities limit polymorphism as reference counts are not polymorphic themselves.

Let region and lock capabilities be denoted as κ_1 and κ_2 respectively, then the *region effect* of region ρ_1 with parent region ρ_2 is denoted as $\rho_1^{\kappa_1,\kappa_2} \rhd \rho_2$.

$$\mathsf{new\_int} : \forall\rho_1.\forall\rho_2.\ \langle \texttt{rgn}(\rho_1) \times \texttt{int}\rangle \xrightarrow{\gamma\to\gamma} \texttt{ref}(\texttt{int}, \rho_1)$$
$$\{\text{* where } \gamma \equiv \rho_1^{1,1} \rhd \rho_2, \rho_2^{1,1} \rhd \emptyset \text{ *}\}$$

Function $\mathsf{new\_int}$ is a integer constructor which, given a region handle and an integer value, returns a fresh reference to a location at region ρ_1 that is initialized to the integer value of the second parameter. The input and output *effect lists* (γ) of this function are denoted on top of the function type. In this example, the input and output effect lists are identical. This implies that the body of function $\mathsf{new\_int}$ invokes no operations that consume capabilities or, if it does, then it restores the original capabilities of ρ_1 and ρ_2.

The input effect list consists of two elements: $\rho_1^{1,1} \rhd \rho_2$ and $\rho_2^{1,1} \rhd \emptyset$. Both elements have a single region and lock capability $(1, 1)$. Region ρ_2 could be the root region as it has no parents.

To enable capability polymorphism, we have used a subtyping relation. As discussed earlier, pure capabilities can be split into several fractional capabilities. Capability subtyping allows to temporarily treat a pure or fractional capability as a (smaller) fractional capability.

$$\mathsf{new\_int} : \forall\rho_1.\forall\rho_2.\ \langle \texttt{rgn}(\rho_1) \times \texttt{int}\rangle \xrightarrow{\gamma\to\gamma} \texttt{ref}(\texttt{int}, \rho_1)$$
$$\{\text{* where } \gamma \equiv \rho_1^{\bar{1},\bar{1}} \rhd \rho_2, \rho_2^{\bar{1},\bar{1}} \rhd \emptyset \text{ *}\}$$

The type of $\mathsf{new\_int}$ can now be expressed in terms of fractional capabilities. The function type assumes that the input effect γ contains two region names ρ_1 and ρ_2, which hold *at least* a single region and lock capability. The calling context tracks the fractional capabilities not passed to the function and joins them along with the fractional capabilities returned from the function call.

The *top* symbol in the subtyping hierarchy for non-zero capabilities is denoted by $*$. It should be noted that *thread-local* capabilities can be converted to $*$.

$$\mathsf{new\_int} : \forall\rho_1.\forall\rho_2.\ \langle \texttt{rgn}(\rho_1) \times \texttt{int}\rangle \xrightarrow{\gamma\to\gamma} \texttt{ref}(\texttt{int}, \rho_1)$$
$$\{\text{* where } \gamma \equiv \rho_1^{*,*} \rhd \rho_2, \rho_2^{*,*} \rhd \emptyset \text{ *}\}$$

This definition of $\mathsf{new\_int}$ is more generic than the previous one, as the *lock capability* can be either *thread-local* or *locked.*

The $*$ capability *reserves* a single fractional capability for the scope of a lexical block. Therefore, it can be safely aliased multiple times. The main disadvantage of $*$ is that it cannot be passed to capability modification operations, such as *increment* or *decrement*, or to a new thread (`spawn`).

Example 4 (Hierarchy abstraction) Region hierarchies provide a higher degree of explicit control over region lifetimes. On the downside, this requires explicit annotations of the form $\rho_1 \rhd \rho_2$, which means that ρ_1 is a *subregion* of ρ_2 (here we have omitted capability superscripts, for simplifying the presentation). A subregion is *live* when its region count is greater than zero and its parent region is *live* or there exists no parent region. This fact implies that a region is inaccessible when its parent is not present in the type annotation of a function. This feature is limiting for region-polymorphic functions as all region names up to the *root region* need to be declared in a function type annotation to typecheck accesses to a particular subregion. The following function illustrates a polymorphic function that accepts immediate children of the root region:

$$\mathsf{access} : \forall\rho_1.\forall\rho_2.\ \langle \mathtt{rgn}(\rho_1) \times \mathtt{rgn}(\rho_2)\rangle \xrightarrow{\gamma\to\gamma} \langle\rangle \qquad \{\text{* where } \gamma \equiv \rho_1 \triangleright \rho_2, \rho_2 \triangleright \emptyset \text{ *}\}$$

To tackle this problem, we allow parent regions to be *temporarily abstracted* in the typing annotations of functions. Of course, this may lead to unsoundness, as a parent region may be deallocated if it is abstracted by all its children regions, before deallocating its subregions. We prevent such issues by enforcing *hierarchy invariants* before and after a function call. With the use of *hierarchy abstraction*, the access function can be written as a truly generic function:

$$\mathsf{access} : \forall\rho_1.\ \mathtt{rgn}(\rho_1) \xrightarrow{\gamma\to\gamma} \langle\rangle \qquad \{\text{* where } \gamma \equiv \rho_1 \triangleright \emptyset \text{ *}\}$$

Example 5 (Bounded existential quantification) Bounded region quantification has been used before (for example in Cyclone) as a means of allowing region abstraction without sacrificing memory safety. The key idea of that work was to use existential quantification so as to abstract regions. For instance a region handle can be *packaged* within an existential type as follows:

$$\mathtt{pack}\ \rho, x\ \mathtt{as}\ \exists\rho.\mathtt{rgn}(\rho)$$

Existential values may *escape* the lexical scope of a region, in this case the scope of region ρ, and be opened when that region is deallocated. To overcome the unsoundness, Grossman et al. [2001] added *region bound annotations* to existential types in Cyclone. For instance, if region ρ has a greater lexical scope than (*outlives*) ρ' then the value in the above example can be given the following type:

$$\mathtt{pack}\ \rho, x\ \mathtt{as}\ \exists\rho[\rho'].\mathtt{rgn}(\rho)$$

In our calculus, we have adapted bounded existential types so that they can be safely used in the presence of early region deallocation and region sharing between multiple threads. In particular, it is unsound to rely on region subtyping (i.e., the outlives relation) in the presence of early deallocation and threads. First, early region deallocation allows a region of certain lexical scope to be deallocated *before* a region of a smaller scope. Second, it is unsafe to assume that when a region of smaller lexical scope is *locked* then a region of greater scope is also locked.

Therefore, we allow bounded existential types, but instead of using a subtyping relation to determine whether an existentially quantified region is *accessible*, we use a *sub-effecting* relation. That is, an existentially quantified region must belong to a *list of region names* (ϵ). To access that region all regions that belong in ϵ must be *accessible*.

$$\mathtt{pack}\ \rho, x\ \mathtt{as}\ \exists\rho[\rho_1, \rho_2].\mathtt{rgn}(\rho)$$

2.2 Syntax Description

We have divided the language syntax into a core syntax and an extended syntax as illustrated in Figures 1 and 2 respectively. The core syntax comprises values and expressions. Values can be integers, tuples, region handles ($\mathtt{rgn}_\iota$) and memory locations ($\mathtt{loc}_\ell$). Expression syntax is similar to value syntax and has additional elements such as constructs for manipulating references: memory allocation

Value	v	$::=$	$n \mid (v, \ldots, v) \mid \texttt{rgn}_\imath \mid \texttt{loc}_l$
Expressions	e	$::=$	$x \mid n \mid (e, \ldots, e) \mid \texttt{prj}_n\ e$
		$\mid$	$\texttt{newrgn}\ \rho, x\ \texttt{at}\ e\ \texttt{in}\ e \mid \texttt{rgn}_\imath \mid \texttt{cap}^{\psi}_{\eta}\ e$
		$\mid$	$\texttt{new}\ e\ \texttt{at}\ e \mid \texttt{deref}\ e \mid e := e \mid \texttt{loc}_l$
Capability kind	ψ	$::=$	$\mathsf{rg} \mid \mathsf{lk}$
Capability op	η	$::=$	$+ \mid -$

Figure 1: Core syntax

Region	r	$::=$	$\rho \mid \imath$
Region list	ϵ	$::=$	$\emptyset \mid \epsilon, r$
Capability	κ	$::=$	$n \mid \overline{n} \mid \bot \mid *$
Effect list	γ	$::=$	$\emptyset \mid \gamma, r^{\kappa,\kappa} \rhd \epsilon \mid \gamma, r^{\top} \rhd \epsilon$
Type	τ	$::=$	$\texttt{int} \mid \texttt{rgn}(r) \mid \langle \tau \times \ldots \times \tau \rangle \mid \texttt{ref}(\tau, r)$
		$\mid$	$\tau \xrightarrow{\gamma \to \gamma} \tau \mid \forall \rho.\, \tau \mid \exists \rho[\epsilon].\, \tau$
Function	f	$::=$	$\lambda\, x.\, e\ \mathsf{as}\ \tau \xrightarrow{\gamma \to \gamma} \tau \mid \Lambda \rho.\, f$
Operation	ξ	$::=$	$\mathsf{seq} \mid \mathsf{par}$
Value	v	$::=$	$\ldots \mid f \mid \texttt{pack}\ r, v\ \texttt{as}\ \tau$
Expressions	e	$::=$	$\ldots \mid f \mid (e\ e)^{\xi} \mid e[r]$
		$\mid$	$\texttt{pack}\ r, e\ \texttt{as}\ \tau \mid \texttt{open}\ e\ \texttt{as}\ \rho, x\ \texttt{in}\ e$
Program	P	$::=$	$\texttt{def}\ x = f; \mid \texttt{def}\ x = f; P$

Figure 2: Full syntax

($\texttt{new}\ e_1\ \texttt{at}\ e_2$), location dereferencing ($\texttt{deref}\ e$), assignment ($e_1\ :=\ e_2$), constructs for manipulating regions ($\texttt{newrgn}\ \rho, x\ \texttt{at}\ e_1\ \texttt{in}\ e_2$) and ($\texttt{cap}^{\psi}_{\eta}\ e$), and tuple projection ($\texttt{prj}_n\ e$).

Most constructs are standard, except for the region manipulation constructs. The $\texttt{newrgn}\ \rho, x\ \texttt{at}\ e_1\ \texttt{in}\ e_2$ construct allocates a *new region* ρ with *region handle* x for the scope of e_2. The new region is allocated in a *parent region* specified by expression e_1, which must evaluate to a *region handle*. Expression e_2 is *obliged* to deallocate region ρ by the end of its scope. As mentioned earlier, each region has two reference counts: one for itself and one for the implicit lock protecting it. The $\texttt{cap}^{\psi}_{\eta}\ e$ construct enables region reference count and lock count manipulation. This construct is parameterized by ψ, which can be either a region (rg) or a lock (lk), and η, which can increment ($+$) or decrement ($-$) the appropriate reference count. Expression e must evaluate to a region handle.

Our calculus is region polymorphic. Therefore, the extended syntax (Figure 2) defines a new meta-variable r, which can be a region *variable* (ρ) or a concrete *name* ($\imath$). The run-time region state is statically tracked via *effects*. Elements of an effect list γ are of the form $r^{state} \rhd \epsilon$, which means that region r is at *state* and *depends* on regions defined in ϵ. Region state can either be *unknown* ($\top$) or some state κ_1, κ_2 for the region and lock reference count correspondingly.

A capability κ can be a *pure* capability n, a *fractional* capability $\overline{n}$ or an *aliasable* capability ($*$). This applies to both regions and locks. Additionally, a lock capability may also be equal to $\bot$ when a region is *thread-local*. *Non-aliasable* capabilities are treated linearly whereas *aliasable* capabilities may be copied an unbounded number of times. A pure capability n implies that a thread can decrement n times the same capability. Pure lock capabilities differ with respect to pure region capabilities in that they can be incremented once they reach the value 0. Fractional capabilities are derived by splitting *pure* or other *fractional* capabilities into several pieces. Therefore, fractional capabilities have a similar behaviour as pure capabilities. Of course, there exist some restrictions when using fractional capabilities; for instance one thread is not allowed to pass a fractional lock capability to another thread, as it is unsound to assume that two threads can simultaneously acquire the same lock. Section 4 explains how and where these invariants are enforced.

Contents	H	$::=$	$\emptyset \mid H, \ell \mapsto v$
Run-time region	I	$::=$	$(\imath, n_1, n_2, n_3)$
Region list	S	$::=$	$\emptyset \mid S, I : (H, S)$
Threads	T	$::=$	$\emptyset \mid T, n : e$
Configuration	C	$::=$	$S; T$

Figure 3: Store, threads and declarations

$$\begin{array}{rcl} E & ::= & \Box \mid (v, \ldots, E, \ldots, e) \mid \mathtt{prj}_\imath\ E \mid (E\ e)^\xi \mid (v\ E)^\xi \mid E\,[r] \\ & \mid & \mathtt{newrgn}\ \rho, x\ \mathtt{at}\ E\ \mathtt{in}\ e \mid \mathtt{cap}^\psi_\eta\ E \\ & \mid & \mathtt{new}\ E\ \mathtt{at}\ e \mid \mathtt{new}\ v\ \mathtt{at}\ E \mid \mathtt{deref}\ E \mid E := e \mid v := E \\ & \mid & \mathtt{pack}\ r, E\ \mathtt{as}\ \tau \mid \mathtt{open}\ E\ \mathtt{as}\ \rho, x\ \mathtt{in}\ e \end{array}$$

Figure 4: Evaluation contexts

Types consist of integers ($\mathtt{int}$), region handles ($\mathtt{rgn}(r)$), tuples ($\langle \tau_1 \times \cdots \times \tau_n \rangle$), references ($\mathtt{ref}(\tau, r)$), functions ($\tau_1 \xrightarrow{\gamma_1 \rightarrow \gamma_2} \tau_2$), polymorphic functions ($\forall \rho.\, \tau$) and bounded existential ($\exists \rho[\epsilon].\, \tau$) types. Region handles and reference types are associated by a type level region name. In combination with the effects, this formulation allows us to determine the state of each region handle and memory reference at each program point. Bounded existential types differ from ordinary existential types in that the abstracted region must belong in one of the regions of ϵ.

At the term level, functions are explicitly annotated with their types. Function types are annotated with the input effects that the calling environment must supply as well as the output effects (i.e., transformed input effects) returned to that environment. The syntax of values and expressions is extended with standard primitives for handling existential values and functions. It is worth noting that there exists no explicit construct for spawning threads, but rather function application is annotated with “par” for parallel and “seq” for sequential (i.e., thread-local) application. Finally, we define the syntax of the entire program as a series of function definitions.

3 Operational Semantics

The *small-step* operational semantics of our language is defined as two evaluation relations, at the level of *threads* and *expressions* (Figures 5 and 6 on the next page). The thread evaluation relation transforms *configurations*. A configuration C (see Figure 3) consists of an abstract *store* S and a list of active threads T. Each thread in T is of the form $n : e$, where n is a thread identifier and e is an expression. The store is a list of regions of the form $I : (H, S)$. Regions are hierarchically organized. Each region can be decomposed into a *region header* I, a memory heap H and a list of subregions S. The region memory heap is a function that maps locations to values. A region header is defined as a tuple which consists of a run-time region name ($\imath$), a region reference count and a lock count (n_1 and n_2 respectively) and a thread owner identifier (n_3).

A *thread evaluation context* E (Figure 4) is defined as an expression with a *hole*, represented as $\Box$. The hole indicates the position where the next reduction step can take place. Our notion of evaluation context imposes a call-by-value evaluation strategy to our language. Subexpressions are evaluated in a left-to-right order.

We assume that concurrent reduction events can be totally ordered. At each step, a random thread is chosen from the thread list for evaluation (rule *E-S* in Figure 5). The thread evaluation relation is annotated with the entire program P, which is passed to the expression evaluation relation. The evaluation relation is also indexed by $\imath$, the identifier of the currently evaluated thread. It should be noted that rule *E-S* is the only *non-deterministic* rule in the operational semantics of our language; in the

$$\frac{S; e \rightarrow_{\imath}^{P} S'; e'}{S; T_1, \imath : E[e], T_2 \rightsquigarrow^{P} S'; T_1, \imath : E[e'], T_2} \textit{ (E-S)} \qquad \frac{}{S; T_1, \imath : (), T_2 \rightsquigarrow^{P} S; T_1, T_2} \textit{ (E-T)}$$

$$\frac{\jmath \text{ is a fresh thread identifier}}{S; T_1, \imath : E[((\lambda\, x.\, e \;\texttt{as}\; \tau)\; v)^{\mathsf{par}}], T_2 \rightsquigarrow^{P} S; T_1, \imath : E[()], T_2, \jmath : e[v/x]} \textit{ (E-SN)}$$

Figure 5: Thread evaluation relation $C \rightsquigarrow^{P} C'$

$$\frac{1 \leq i \leq n}{S; \texttt{prj}_i\; (v_1, \ldots, v_n) \rightarrow_{\imath}^{P} S; v_i} \textit{ (E-P)} \qquad \frac{(\texttt{def}\; x = f) \in P}{S; x \rightarrow_{\imath}^{P} S; f} \textit{ (E-F)}$$

$$\frac{}{S; ((\lambda\, x.\, e \;\texttt{as}\; \tau)\; v)^{\mathsf{seq}} \rightarrow_{\imath}^{P} S; e[v/x]} \textit{ (E-A)} \qquad \frac{}{S; (\Lambda \rho.\, f)[r] \rightarrow_{\imath}^{P} S; f[r/\rho]} \textit{ (E-RP)}$$

$$\frac{(\ell, S') = \mathrm{alloc}(\jmath, S, v)}{S; \texttt{new}\; v \;\texttt{at}\; \texttt{rgn}_{\jmath} \rightarrow_{\imath}^{P} S'; \texttt{loc}_{\ell}} \textit{ (E-NRF)} \qquad \frac{v = \mathrm{lookup}_{\imath}(S, \ell)}{S; \texttt{deref}\; \texttt{loc}_{\ell} \rightarrow_{\imath}^{P} S; v} \textit{ (E-D)} \qquad \frac{S' = \mathrm{update}_{\imath}(S, \ell, v)}{S; \texttt{loc}_{\ell} := v \rightarrow_{\imath}^{P} S'; ()} \textit{ (}$$

$$\frac{(k, S') = \mathrm{newrgn}(\jmath, S)}{S; \texttt{newrgn}\; \rho, x \;\texttt{at}\; \texttt{rgn}_{\jmath} \;\texttt{in}\; e \rightarrow_{\imath}^{P} S'; e[k/\rho][\texttt{rgn}_k/x]} \textit{ (E-NR)} \qquad \frac{S' = \mathrm{updcap}_{\imath}(\jmath, S, \psi, \eta)}{S; \texttt{cap}_{\eta}^{\psi}\; \texttt{rgn}_{\jmath} \rightarrow_{\imath}^{P} S'; ()} \textit{ (E-C)}$$

$$\frac{\mathrm{updcap}_{\imath}(\jmath, S, \psi, \eta) = \text{“block”}}{S; \texttt{cap}_{\eta}^{\psi}\; \texttt{rgn}_{\jmath} \rightarrow_{\imath}^{P} S; \texttt{cap}_{\eta}^{\psi}\; \texttt{rgn}_{\jmath}} \textit{ (E-CB)} \qquad \frac{}{S; \texttt{open}\; (\texttt{pack}\; \jmath, v \;\texttt{as}\; \tau) \;\texttt{as}\; \rho, x \;\texttt{in}\; e \rightarrow_{\imath}^{P} S; e[\jmath/\rho][v/x]} \textit{ (E-(}$$

Figure 6: Expression evaluation relation $S; e \rightarrow_{\imath}^{P} S'e'$

presence of more than one active threads, our semantics does not specify which one will be selected for evaluation.

Threads that have completed their evaluation and have been reduced to *unit* values, represented as $()$, are removed from the active thread list (rule *E-T*). When a parallel function application redex is detected within the evaluation context of a thread, a new thread is created (rule *E-SN*). The redex is replaced with a unit value in the currently executing thread and a new thread is added to the thread list, with a *fresh* thread identifier.

The expression evaluation relation is defined in Figure 6. The rules for reducing projections (*E-P*), opening existential values (*E-OR*), function application (*E-A*) and region application (*E-AR*) are standard. Rule *E-F* resolves function names to the appropriate function bodies. The remaining rules make use of five *partial* functions that manipulate the store. These functions are undefined when their constraints are not met. All of them require that some region be *live*. A region is live when its own region count is greater than zero, as well as the region counts of all its parents. In addition to liveness, some of these functions require that some region be *accessible* from the currently executing thread. A region is accessible from some thread $\jmath$ when it is live, its lock count is greater than zero and its thread owner is equal to $\jmath$ (in other words, $\jmath$ has successfully obtained a lock for this region).

- $\mathrm{alloc}(\jmath, S, v)$ is used in rule *E-NRF* for creating a new reference. It allocates a new object in S. The object is placed in region $\jmath$ and is set initially to the value v. Region $\jmath$ must be live. Upon success, the function returns a pair (ℓ, S') containing a fresh location of the new object and the new store.
- $\mathrm{lookup}_{\imath}(S, \ell)$ is used in rule *E-D* to look up the value of location ℓ in S. The region in which ℓ resides must be live and accessible from the currently executed thread $\imath$. Upon success, the function returns the value v held in ℓ.

- $\mathrm{update}_\imath(S, \ell, v)$ is used in rule *E-AS* to assign the value v to location ℓ in S. The region in which ℓ resides must be live and accessible from the currently executed thread $\imath$. Upon success, the function returns the new store S'.
- $\mathrm{newrgn}(\jmath, S)$ is used in rule *E-NR* to create a new region in S. The new region is made a subregion of $\jmath$, which must be live. Its region and lock counts are set to 1 and -1 respectively. Upon success, the function returns a pair (k, S') containing a fresh region name for the new region and the new store.
- $\mathrm{updcap}_\imath(\jmath, S, \psi, \eta)$ is used in rules *E-C* and *E-CB*. It requests to obtain or drop (depending on η) a capability of kind ψ (i.e., region or lock) for region $\jmath$ in S. Upon success, the function returns the new store S'. It may also return the special value "block" in case the currently executing thread ($\imath$) cannot immediately obtain or drop this capability.

Rules *E-C* and *E-CB* guarantee that our operational semantics does not get stuck as a result of a *deadlock*, when attempting to modify a region or lock count. The former rule is used in case the partial function updcap returns a new store, which means that the modification attempt was successful. The latter implements a naïve "busy-wait" semantics by taking an idle step. It is used in case updcap returns the special value "block," meaning e.g. that the lock for a region is already held by another thread and that the currently executed thread must wait for it to be released, before it can obtain it.

It should be noted at this point that our semantics will get stuck if a thread attempts to assign to a memory location without first acquiring the lock for the region where this location belongs. In this case, the result of update will be undefined and it will not be possible to use rule *E-AS*. The same is true in several other situations (e.g., reading from a non-existent location). Threads that may cause a data race will definitely get stuck.

Our semantics follows a different approach from related work (e.g., the work of Grossman [2003]), where a special kind of value $junk_v$ is used as an intermediate step when assigning a value v to a location, before the real assignment takes place. Then, type safety guarantees that no junk values will ever be used. As we described above, we use a more direct approach by incorporating the locking mechanism in our operational semantics. However, our progress lemma in Section 5 guarantees that, at any time, *all* threads can make progress. Thus, a possible implementation of our semantics does not have to check at run-time if a lock is held by the currently executed thread.

4 Static Semantics

In this section we discuss the typing rules of our language. To enforce our safety invariants we use a *type and effect system*. As mentioned earlier, effects are used to statically track the state of each region. A well-typed expression e has a type τ under an *input effect* list γ and results in an *output effect* list γ'. We denote this by $R; M; \Delta; \Gamma \vdash e : \tau \;\&\; (\gamma;\gamma')$. This typing judgement uses four standard typing contexts: R is a set of *concrete region names*, M is a mapping of *concrete location names* to types, Δ is a set of region variables, and Γ is a mapping of term variables to types. Region and location types are tagged with regions. This allows us to associate well-typed region handles and locations with the set of effects available at each program point.

The typing rules for variables, values, tuple creation and projection are standard. To improve presentation, we have omitted all well-formedness premises for contexts from our typing rules. In the typing of variables and values, the input effect list is the same as the output effect list. The typing rule for lambda abstraction terms requires that the type annotation be α-equivalent to the arrow type deduced by the premise of the rule. Furthermore, it is necessary to *summarize* the input and output effect lists before typechecking the body of a lambda abstraction; we denote the summarized effect lists by $\overline{\gamma_1}$ and $\overline{\gamma_2}$. The rationale behind the summarization process is that region substitution in an effect list may violate the

Variables and Values

$$\frac{(x \mapsto \tau) \in \Gamma, \Gamma_0}{R; M; \Delta; \Gamma \vdash\ x : \tau \mathbin{\&} (\gamma;\gamma)}\ \textit{(T-V)} \qquad \frac{}{R; M; \Delta; \Gamma \vdash\ n : \texttt{int} \mathbin{\&} (\gamma;\gamma)}\ \textit{(T-I)}$$

$$\frac{R; \Delta \vdash_R \imath}{R; M; \Delta; \Gamma \vdash\ \texttt{rgn}_\imath : \texttt{rgn}(\imath) \mathbin{\&} (\gamma;\gamma)}\ \textit{(T-R)} \qquad \frac{(\ell \mapsto (\tau, \imath)) \in M}{R; M; \Delta; \Gamma \vdash\ \texttt{loc}_l : \texttt{ref}(\tau, \imath) \mathbin{\&} (\gamma;\gamma)}\ \textit{(T-L)}$$

$$\frac{\tau \equiv \tau_1 \xrightarrow{\gamma_1 \rightarrow \gamma_2} \tau_2 \qquad R; M; \Delta; \Gamma, x : \tau_1 \vdash\ e : \tau_2 \mathbin{\&} (\overline{\gamma_1};\overline{\gamma_2})}{R; M; \Delta; \Gamma \vdash\ \lambda x.\, e\ \texttt{as}\ \tau : \tau \mathbin{\&} (\gamma;\gamma)}\ \textit{(T-F)} \qquad \frac{R; M; \Delta, \rho; \Gamma \vdash\ f : \tau \mathbin{\&} (\gamma;\gamma)}{R; M; \Delta; \Gamma \vdash\ \Lambda\rho.\, f : \forall\rho.\, \tau \mathbin{\&} (\gamma;\gamma)}\ \textit{(T-RI}$$

Tuples

$$\frac{R; M; \Delta; \Gamma \vdash\ e_i : \tau_i \mathbin{\&} (\gamma_i;\gamma_{i+1})\ \ \text{forall } 0 \leq i < n}{R; M; \Delta; \Gamma \vdash\ (e_0, \ldots, e_{n-1}) : \langle \tau_0 \times \ldots \times \tau_{n-1} \rangle \mathbin{\&} (\gamma_0;\gamma_n)}\ \textit{(T-Tu)} \qquad \frac{\begin{array}{c} 0 \leq i < n \\ R; M; \Delta; \Gamma \vdash\ e : \langle \tau_0 \times \ldots \times \tau_{n-1} \rangle \mathbin{\&} (\gamma;\gamma') \end{array}}{R; M; \Delta; \Gamma \vdash\ \texttt{prj}_i\ e : \tau_i \mathbin{\&} (\gamma;\gamma')}$$

Function and Region Application

$$\frac{\begin{array}{c} R; M; \Delta; \Gamma \vdash\ e_1 : \tau_1 \xrightarrow{\gamma_1 \rightarrow \gamma_2} \tau_2 \mathbin{\&} (\gamma;\gamma') \\ R; M; \Delta; \Gamma \vdash\ e_2 : \tau_1 \mathbin{\&} (\gamma';\gamma_{in}) \\ \text{if } \xi \equiv \mathsf{par} \text{ then } \tau_2 \equiv \langle\rangle \qquad \gamma_{out} = \text{output}(\gamma_{in}, \xi, \gamma_1, \gamma_2) \end{array}}{R; M; \Delta; \Gamma \vdash\ (e_1\ e_2)^\xi : \tau_2 \mathbin{\&} (\gamma;\gamma_{out})}\ \textit{(T-App)} \qquad \frac{\begin{array}{c} R; \Delta \vdash_R r \\ R; M; \Delta; \Gamma \vdash\ e : \forall\rho.\, \tau \mathbin{\&} (\gamma;\gamma') \end{array}}{R; M; \Delta; \Gamma \vdash\ e\ [r] : \tau[r/\rho] \mathbin{\&} (\gamma;\gamma')}\ \textit{(T-RA}$$

References to Mutable Objects

$$\frac{\begin{array}{c} R; M; \Delta; \Gamma \vdash\ e_1 : \tau \mathbin{\&} (\gamma;\gamma') \\ R; M; \Delta; \Gamma \vdash\ e_2 : \texttt{rgn}(r) \mathbin{\&} (\gamma';\gamma'') \qquad \text{live}(\gamma'', \{r\}) \end{array}}{R; M; \Delta; \Gamma \vdash \texttt{new}\ e_1\ \texttt{at}\ e_2 : \texttt{ref}(\tau, r) \mathbin{\&} (\gamma;\gamma'')}\ \textit{(T-NR)} \qquad \frac{R; M; \Delta; \Gamma \vdash\ e : \texttt{ref}(\tau, r) \mathbin{\&} (\gamma;\gamma') \qquad \text{accessible}(\gamma', \{}{R; M; \Delta; \Gamma \vdash\ \texttt{deref}\ e : \tau \mathbin{\&} (\gamma;\gamma')}$$

$$\frac{R; M; \Delta; \Gamma \vdash\ e_1 : \texttt{ref}(\tau, r) \mathbin{\&} (\gamma;\gamma') \qquad R; M; \Delta; \Gamma \vdash\ e_2 : \tau \mathbin{\&} (\gamma';\gamma'') \qquad \text{accessible}(\gamma'', \{r\})}{R; M; \Delta; \Gamma \vdash\ e_1 := e_2 : \langle\rangle \mathbin{\&} (\gamma;\gamma'')}\ \textit{(T-A)}$$

Region Primitives

$$\frac{\begin{array}{c} R; M; \Delta; \Gamma \vdash\ e_1 : \texttt{rgn}(r) \mathbin{\&} (\gamma;\gamma') \qquad \text{live}(\gamma', \{r\}) \qquad R; \Delta \vdash_T \tau \\ R; M; \Delta, \rho; \Gamma, x : \texttt{rgn}(\rho) \vdash\ e_2 : \tau \mathbin{\&} (\gamma', \rho^{1,\perp} \triangleright r;\gamma'') \qquad \rho \notin \text{dom}(\gamma'') \end{array}}{R; M; \Delta; \Gamma \vdash\ \texttt{newrgn}\ \rho, x\ \texttt{at}\ e_1\ \texttt{in}\ e_2 : \tau \mathbin{\&} (\gamma;\gamma'')}\ \textit{(T-NRG)} \qquad \frac{\begin{array}{c} R; M; \Delta; \Gamma \vdash\ e_1 : \texttt{rgn}(r) \mathbin{\&} (\gamma;\gamma') \\ \gamma'' = \text{modcap}(\gamma', \psi, \eta, r) \end{array}}{R; M; \Delta; \Gamma \vdash\ \texttt{cap}_\eta^\psi\ e_1 : \langle\rangle \mathbin{\&} (\gamma;\gamma''}$$

Bounded Existential Quantification

$$\frac{\begin{array}{c} R; \Delta \vdash_T \tau \qquad \tau \equiv \exists\rho[\epsilon].\, \tau' \qquad R; \Delta \vdash_R r \qquad r \in \epsilon \\ R; M; \Delta; \Gamma \vdash\ e : \tau'[r/\rho] \mathbin{\&} (\gamma;\gamma') \end{array}}{R; M; \Delta; \Gamma \vdash\ \texttt{pack}\ r, e\ \texttt{as}\ \tau : \tau \mathbin{\&} (\gamma;\gamma')}\ \textit{(T-EP)} \qquad \frac{\begin{array}{c} R; \Delta \vdash_T \tau' \qquad R; M; \Delta; \Gamma \vdash\ e : \exists\rho[\epsilon].\, \tau \mathbin{\&} (\gamma;\gamma') \\ R; M; \Delta, \rho'; \Gamma, x : \tau[\rho'/\rho] \vdash\ e' : \tau' \mathbin{\&} (\gamma', \rho'^{\top} \triangleright \epsilon;\gamma'', \rho'^{\top} \triangleright \end{array}}{R; M; \Delta; \Gamma \vdash\ \texttt{open}\ e\ \texttt{as}\ \rho', x\ \texttt{in}\ e' : \tau' \mathbin{\&} (\gamma;\gamma'')}$$

Figure 7: Static semantics

Table 1: Capability conversion

ψ	ξ	κ_1	κ_2	κ_{in}	$\mathrm{out}(\kappa_{in}, \psi, \xi, \kappa_1, \kappa_2)$	constraints
any	seq	n_1	n_2	n_1	n_2	
any	seq	$\overline{n_1}$	$\overline{n_2}$	n_3	$n_2 + n_3 - n_1$	$n_1 \leq n_3$
any	seq	$\overline{n_1}$	$\overline{n_2}$	$\overline{n_3}$	$\overline{n_2 + n_3 - n_1}$	$n_1 \leq n_3$
any	seq	$*$	$*$	n_3	n_3	$0 < n_3$
any	seq	$*$	$*$	$\overline{n_3}$	$\overline{n_3}$	$0 < n_3$
any	seq	$*$	$*$	$*$	$*$	
lk	seq	$\bot$	$\bot$	$\bot$	$\bot$	
lk	seq	$\bot$	0	$\bot$	0	
lk	seq	$*$	$*$	$\bot$	$\bot$	
rg	par	n_1	0	n_2	$n_1 - n_2$	$0 \leq n_1 \leq n_2$
rg	par	n_1	0	$\overline{n_2}$	$\overline{n_1 - n_2}$	$0 \leq n_1 \leq n_2$
lk	par	0	0	n_1	n_1	
lk	par	0	0	$\overline{n_1}$	$\overline{n_1}$	
lk	par	$\bot$	0	$\bot$	0	

invariant that each effect $r^{state} \triangleright \epsilon$ in the list refers to a unique region r. Summarization restores this invariant by appropriately merging different states.

The typing rule for function application computes the output effect list for the call γ_{out}, taking into account γ_{in} (the output effect list of expression e_2, which corresponds to the beginning of the call), the input and output effect lists γ_1 and γ_2 ascribed on the type of the function, and the application tag ξ which can be either seq or par. For any region r, the partial function output isolates the effects related to r in γ_{in}, γ_1 and γ_2. If such an effect only exists in γ_{in}, then it is copied to γ_{out} unchanged (the function does not affect region r at all). If, however, effects for r also exist in γ_1 and γ_2, then the region and lock capabilities of all three effects must be combined in the effect that will be copied to γ_{out}, in the way given in Table 1. In all other cases, e.g. if γ_1 contains an effect for region r and γ_2 does not contain such an effect, then the result of $output$ is undefined and this results in a type error.

Table 1 defines the output capability for any region r, given the capability kind ψ, the type of application ξ and the three capabilities κ_{in}, κ_1 and κ_2 for r that are found in γ_{in}, γ_1 and γ_2, respectively. Effects of *unknown state* ($r^{\top} \triangleright \epsilon$) must have previously been excluded from the transformation process. From Table 1 it is obvious that aliasable capabilities cannot be passed in a parallel function application. Furthermore, when a thread-local region (with a $\bot$ lock capability) is passed to another thread, it remains local (i.e., we have *region migration*). On the other hand, we can permanently convert the local lock capability $\bot$ to 0. This feature is particularly useful when sharing a region between threads. Moreover, we can convert a pure or fractional capability to another fractional capability.

Fractional capabilities of the two kinds are treated differently in the presence of parallel application. Region capabilities in γ_{in} can be either fractional or pure, but in the scope of the thread function both become pure as we delegate the obligation of region deallocation to the new thread. The output capability in γ_{out} in this case is defined as the difference between κ_{in} and κ_1. Fractional lock capabilities are disallowed in parallel application, as it is unsound to assume statically that two threads can acquire the same lock simultaneously. Finally, as we see in Table 1, the new thread is forced to release all capabilities by the end of its scope (or earlier).

In addition to allowing for *capability abstraction*, the partial function output allows for *abstraction over dependencies* of each effect (i.e. shrink ϵ in $r^{state} \triangleright \epsilon$). The idea is to temporarily ignore (some or all) dependencies for the function scope. This feature is particularly useful in the presence of threads: a region is *live* (at the type level) when its dependencies are live and its region capability is not zero. Therefore, if there exists no abstraction over dependencies, the entire region hierarchy up to the root region must be passed to the new thread. Of course, this implies that there exist *only sharable regions*, and thread functions must accept all possible regions.

To preserve soundness, certain restrictions are imposed, e.g. the abstracted dependencies *must* be live at the beginning and the end of the function scope (region invariant). As mentioned earlier, a region can be locked when its children are locked. When a region temporarily abstracts parent dependencies within a certain scope, it is able to break that invariant and unlock itself before its abstracted parent is unlocked. However, before returning to the original environment, function output checks that the regions “breaking” the invariant in γ_{out} lock themselves, so that the lock invariant is restored.

The typing rules for reference creation (*T-NR*), dereference (*T-D*) and assignment (*T-A*) are standard, except for the judgement $\mathrm{accessible}(\gamma, r)$ which appears as a premise. This judgement ensures that region r is *live* (i.e. all of its dependencies are live and its region capability is greater than zero) and *locked* or *thread-local*.

The rule for typing a new region construct (*T-NRG*) checks that expressions e_1 and e_2 are of type $\texttt{rgn}(r_1)$ and τ, respectively, and that τ does not contain any occurrences of the *fresh region variable* ρ. The body expression e_2 is typechecked in a context extended with the fresh region ρ and a new effect element is added to the input effect list of e_2 ($\rho^{1,\perp} \rhd r_1$), which states that there exists a new region ρ with region capability of 1, lock capability $\perp$ (thread-local) and its parent region is the same as the region name tagged in the type of expression e_1. From the output effect list of expression e_2, we notice that e_2 must have released both the region and lock capability, either by delegating the deallocation responsibility to another thread or by explicitly decrementing its capabilities. We also check that r_1 is live in the input effect of expression e_2 (we denote this by $\mathrm{live}(\gamma, r_1)$), as it is unsound to allocate a fresh region within a deallocated parent region.

Rule *T-CP* ensures that when we attempt to modify the state of an effect, the region of that effect is live and we are allowed to change its capability. This is achieved with the partial function $\mathrm{modcap}(\gamma, \psi, \eta, r)$, which modifies by η the capability of kind ψ for region r in the effect list γ and, upon success, returns the modified effect list. Several restrictions are imposed. For instance, a capability cannot be decremented below zero and a region capability which has reached the value zero cannot be incremented again. Function modcap again checks the resulting output effect list against the lock and region invariants described above, to ensure soundness.

The typing rule for packing bounded existential values (*T-EP*) requires that the abstracted region be an element of ϵ. The rule for opening an existential package in the scope of an expression e' adds the effect $r^{\top} \rhd \epsilon$ to the input effect list of e'. This kind of effect differs from effects with explicit capabilities in that r here is an abstracted region. Its state is always $\top$ and r is an element of the dependency list ϵ. This form of bounded quantification allows a finer control of region abstraction. Using such effects may be restrictive as *all regions* in ϵ must be *accessible* (i.e. live and locked), in order to access a memory location of region r.

5 Type Safety

In this section we discuss the fundamental theorems that prove type safety of our language. Our formulation of type safety is based on proving the *preservation* and *progress* theorems. Informally, our language is defined as safe when each evaluation step is *well-typed* and *not stuck*. A program configuration is *not stuck* when all threads are not stuck, that is, if all threads can make evaluation steps. As we discussed in Section 3, a thread can become stuck if it tries to access a region that is not *live*, if it tries to dereference or assign to a location residing in a region that is not *accessible* by the thread. (These are obviously the interesting cases in our concurrent setting. Of course a thread can become stuck if it tries to perform an operation that does not respect standard types.)

Definition 1 (Threads Typing) Let T be a collection of threads. Let $R; M; \Delta; \Gamma$ be a global typing context. Let δ be a mapping from thread identifiers to effect lists; for each thread $n : e$ in T we take $\delta(n)$ to be the input effect list that corresponds to the evaluation of expression e. Then, T is *well-typed* with respect to $R; M; \Delta; \Gamma; \delta$ when each thread $n : e$ in T is well-typed with respect to $R; M; \Delta; \Gamma$, the input effect list $\delta(n)$, and when there exist no *live* regions at the corresponding output effect list.

$$\frac{}{R; M; \Delta; \Gamma; \delta \vdash_T \emptyset}$$

$$\frac{R; M; \Delta; \Gamma \vdash e : \langle\rangle \; \& \; (\delta(n);\gamma) \qquad R; M; \Delta; \Gamma; \delta \vdash_T T \qquad \text{linear}(\gamma) \equiv \emptyset}{R; M; \Delta; \Gamma; \delta \vdash_T n : e, T}$$

Definition 2 (Store Consistency) A store S is *consistent* with respect to an effect mapping δ when the following conditions hold:

- The set of region names occurring in the co-domain of δ is a subset of the set of region names in S.
- Each region that appears in δ (and all of its parents) is live in S. Furthermore, the sum of all region capabilities (for all different threads) for a region in δ is equal to the region's reference count in S.
- All regions that appear in S but not in the co-domain of δ are deallocated (zero region and lock count).
- For each region that appears in δ, at most one thread in δ may have a lock capability that is non-zero. In this case, the lock capability of the region is either positive or $\bot$ and it is equal to the lock count of the region in S ($\bot$ is represented as -1).

Definition 3 (Store Typing) A store S is *well-typed* with respect to $R; M; \delta$ (we denote this by $R; M; \delta \vdash_{str} S$) when

- S is consistent with respect to δ,
- the set of region names in S is equal to R,
- the set of locations in M is equal to the set of locations in S, and
- each value v stored in a location ℓ of S is closed and has type $M(\ell)$ with empty effect lists ($R; M; \emptyset; \emptyset \vdash v : M(\ell) \; \& \; (\emptyset;\emptyset)$).

A configuration $S; T$ is *well-typed* with respect to $R; M; \Delta; \Gamma; \delta$ when the collection of threads T is well-typed with respect to $R; M; \Delta; \Gamma; \delta$ and the store S is well-typed with respect to $R; M; \delta$.

Definition 4 (Configuration Typing)

$$\frac{R; M; \Delta; \Gamma; \delta \vdash_T T \qquad R; M; \delta \vdash_{str} S}{R; M; \Delta; \Gamma; \delta \vdash_C S; T}$$

Definition 5 (Not stuck) A configuration $S; T$ is *not stuck* if *all* threads in T can take one of the evaluation steps in Figure 5 (*E-S*, *E-T* or *E-SN*).

Given these definitions, we can now present the main results of this paper. The *progress* and *preservation* lemmata are first formalized at the *program* level, i.e. for all concurrently executed threads. Proof sketches for all lemmata and the final type safety theorem are given in the Appendix.

Lemma 1 (Progress – Program) *Let $S; T$ be a closed well-typed configuration with $R; M; \emptyset; \emptyset; \delta \vdash_C S; T$. Then $S; T$ is not stuck.*

Lemma 2 (Preservation – Program) *Let $S; T$ be a well-typed configuration with $R; M; \Delta; \Gamma; \delta \vdash_C S; T$. If the operational semantics takes a step $S; T \leadsto^P S'; T'$, then there exist $R' \supseteq R$, $M' \supseteq M$ and δ' such that the resulting configuration is well-typed with $R'; M'; \Delta; \Gamma; \delta' \vdash_C S'; T'$.*

A thread-local version for each of these two lemmata is required, in order to prove the above. At the *expression* level, progress and preservation are defined as follows.

Lemma 3 (Progress – Expression) *Let S be a well-typed store with $R; M; \delta \vdash_{str} S$ and let e be a closed well-typed expression with $R; M; \emptyset; \emptyset \vdash e : \tau \,\&\, (\delta(n); \gamma)$. Then exactly one of the following is true:*

- *e is a value, or*
- *e is of the form $E[((\lambda\, x.\, e \texttt{ as } \tau)\ v)^{\mathsf{par}}]$, or*
- *e is of the form $E[e']$ and there exist S' and e'' such that $S; e' \to^P_n S'; e''$.*

Lemma 4 (Preservation – Expression) *Let e be a well-typed expression with $R; M; \Delta; \Gamma \vdash e : \tau \,\&\, (\delta(n); \gamma)$ and let S be a well-typed store with $R; M; \delta \vdash_{str} S$. If the operational semantics takes a step $S; e \to^P_n S'; e'$, then there exist $R' \supseteq R$, $M' \supseteq M$ and γ' such that the resulting expression and the resulting store are well-typed with $R'; M'; \Delta; \Gamma \vdash e' : \tau \,\&\, (\gamma'; \gamma)$ and $R; M; \delta[n \mapsto \gamma'] \vdash_{str} S'$.*

In the preservation lemma, we typecheck the expression e with the input effect list $\delta(n)$ that corresponds to thread n in the configuration. We require that the store S be well-typed with respect to the same R, M and δ. The input effect γ' of the resulting expression e' is placed back in the global mapping δ and we denote this by $\delta[n \mapsto \gamma']$: a mapping which is identical to δ, only it maps n to γ'. The resulting store S' must be well-typed in this updated mapping.

Finally, the *type safety* theorem is a direct consequence of Lemmata 1 and 2. It states that when the original program P contains a `main` function of the appropriate type and `main` is invoked with the program's heap region name and handle, every step of the thread evaluation relation results in a configuration which is *not stuck*. More specifically, let H be the region name that corresponds to the program heap and let $R_0 \equiv \emptyset, H$. Let δ_0 be such that it only maps thread 1 to the input effect list $H^{1,\perp} \triangleright \emptyset$, in other words, the heap region has reference count equal to 1 and is initially thread-local. Let $I_0 \equiv (H, 1, -1, 0)$ be the region header for the heap and let $S_0 \equiv \emptyset, I_0 : (\emptyset, \emptyset)$ be the initial store, in other words, the heap is initially empty. Also, let $T_0 \equiv \emptyset, 1 : e_0$ be the initial thread list, where $e_0 \equiv (\texttt{main}\,[H]\ \texttt{rgn}_H)^{\mathsf{seq}}$.

Theorem 1 (Type Safety) *If the initial configuration is well-typed with $R_0; \emptyset; \emptyset; \emptyset; \delta_0 \vdash_C S_0; T_0$ and the operational semantics takes any number of steps $S_0; T_0 \leadsto^{P*} S; T$, then the resulting configuration $S; T$ is not stuck.*

The empty (except for the heap H) contexts that are used when typechecking the initial configuration $S_0; T_0$ guarantee that all functions in the program are closed and that no explicit region values ($\texttt{rgn}_\imath$) or location values ($\texttt{loc}_\ell$) are used in the source of the original program.

6 Related Work

In this paper, we have presented a low-level language with lexically-scoped hierarchical regions and a type system which allows for safe early region deallocation in a multithreaded setting. We feel that our language is a promising point in the programming language design space but, quite naturally, it has many features in common with other languages having similar aims and adopts several ideas and concepts from the literature.

Languages employing region-based memory management In their seminal work, Tofte and Talpin [1994, 1997] proposed a new memory management scheme for higher-order, typed languages, as an alternative to manual (i.e., `malloc`/`free` in C) and automatic (i.e., garbage collection) memory management. The main idea is to introduce a block-structured construct (`letregion` ρ `in` e), which allocates a new region of memory for storing objects created during the evaluation of e, and automatically deallocates this region upon e's termination. Although the basic Tofte-Talpin framework imposes a strict LIFO order on region lifetimes, a number of extensions have emerged that relax this constraint. For example, Aiken et al. [1995] provide an analysis to free some regions early and Walker and Watkins [2001] propose systems for freeing regions based on linear types. In the same track, Fluet et al. [2006] have designed a very powerful substructural type system for the safe deallocation of non-LIFO regions, based on linear *region capabilities*. More recently, Boudol [2008] has presented a higher-order language which allows for safe region deallocation by using a type system with *deallocation effects*.

As reported by Tofte et al. [2004], region-based memory management allows the ML Kit compiler to produce code that executes quite efficiently, without the support of a garbage collector. Furthermore, memory management with statically scoped regions is provably safe. As a result, several language implementations have adopted region-based memory management to obtain memory safety while avoiding (totally or partly) the cost of garbage collection. For instance, Cyclone [Grossman et al., 2002] uses static regions to achieve these goals. In addition to static regions, other languages, such as RC [Gay and Aiken, 2001], also provide more flexible dynamic regions which employ *reference counting* and programmer-supplied annotations. This however is error-prone and sacrifices static safety guarantees. Other languages for low-level applications, such as Vault [Fahndrich and DeLine, 2002], employ a type system that restricts aliases and two mechanisms, termed *adoption* and *focus*, which allow to statically track stateful properties about data without knowing all aliases of this data.

Safe multithreaded low-level languages All the above works rely crucially on the absence of concurrent accesses and are not safe in the presence of multithreading. For example, in the actual implementation of Cyclone, the `let alias` construct allows temporary aliasing of capabilities within the scope of that construct. If an aliased capability is passed to another thread, then the linearity of that capability is violated once the `let alias` construct terminates. To extend Cyclone with threads and locks and overcome problems with data races breaking type safety, Grossman [2003] introduced a type system where each pointer and lock type is explicitly annotated with a lock name. This system has scoped locks and does not allow for early lock release and region deallocation. Moreover, it has several drawbacks as it has a non-uniform interaction with lock polymorphism and a complicated kind system for *reference sharabilities*. In our system, locks do not have to be scoped and we provide lock polymorphism through effect subtyping. Furthermore, there is no need for a complicated kind system, as the lock state of a region may switch from thread-local to shared. We also treat locks as an operating system resource, which must be released when a region is deallocated, and completely eliminate the use of hidden run-time reference counts so as to gain control over the run-time system in a type-safe manner. On the downside, our language offers less fine-grained locking than Grossman's extension of Cyclone. To allow safety over reference counts, we employed *fractional capabilities* inspired from Boyland's work on *fractional permissions*. However, unlike the work of Boyland [2003], we provide fractional capabilities in a multithreaded setting and also a type safety proof.

7 Concluding Remarks

In this paper, we have presented a first-order region-polymorphic calculus which allows for safe early deallocation of regions in a multithreaded setting and prevents data races. The main novelty of our type system is that it employs a tree-like hierarchy of regions and locks. We also allow safe capability aliasing in the presence of threads, explicit capability consumption and polymorphism. Our type system provides support for capability fork and join operations. We also provide additional abstraction mechanisms such as *capability subtyping*, *hierarchy abstraction* and *bounded existential types*. Our system supports region migration and sharing. In contrast to other type systems, we allow regions to switch state from *thread-local* to *sharable*.

References

A. Aiken, M. Fähndrich, and 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*, pages 174–185, New York, NY, USA, June 1995. ACM Press.

G. Boudol. Typing safe deallocation. In S. Drossopoulou, editor, *Programming Language and Systems: Proceedings of the European Symposium on Programming*, volume 4960 of *LNCS*, pages 116–130. Springer, Apr. 2008.

C. Boyapati, A. Salcianu, W. S. Beebee, and 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*, pages 324–337, New York, NY, USA, June 2003. ACM Press.

J. Boyland. Checking interference with fractional permissions. In R. Cousot, editor, *Static Analysis: Proceedings of the 10th International Symposium*, volume 2694 of *LNCS*, pages 55–72. Springer, June 2003.

M. Fahndrich and R. DeLine. Adoption and focus: practical linear types for imperative programming. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 13–24, New York, NY, USA, June 2002. ACM Press. doi: http://doi.acm.org/10.1145/543552.512532.

C. Flanagan and M. Abadi. Object types against races. In J. C. M. Baeten and S. Mauw, editors, *Concurrency Theory: Proceedings of the 10th International Conference*, volume 1664 of *LNCS*, pages 288–303. Springer, 1999.

C. Flanagan, S. N. Freund, M. Lifshin, and S. Qadeer. Types for atomicity: Static checking and inference for Java. *ACM Trans. Prog. Lang. Syst.*, 30(4), July 2008.

M. Fluet, G. Morrisett, and A. Ahmed. Linear regions are all you need. In P. Sestoft, editor, *Programming Language and Systems: Proceedings of the European Symposium on Programming*, volume 3924 of *LNCS*, pages 7–21. Springer, Mar. 2006.

D. Gay and A. Aiken. Language support for regions. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 70–80, New York, NY, USA, May 2001. ACM Press.

D. Grossman. Type-safe multithreading in Cyclone. In *Proceedings of the ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*, pages 13–25, New York, NY, USA, Jan. 2003. ACM Press.

D. Grossman, G. Morrisett, Y. Wang, T. Jim, M. Hicks, and J. Cheney. Formal type soundness for Cyclone's region system. Technical Report TR2001-1856, Cornell University, 2001.

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

M. Hicks, J. S. Foster, and P. Pratikakis. Lock inference for atomic sections. In *Proceedings of the First ACM SIGPLAN Workshop on Languages Compilers, and Hardware Support for Transactional Computing*, June 2006.

B. McCloskey, F. Zhou, D. Gay, and E. A. Brewer. Autolocker: Synchronization inference for atomic sections. In *Conference Record of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 346–358. ACM Press, Jan. 2006.

F. Smith, D. Walker, and G. Morrisett. Alias types. In G. Smolka, editor, *Programming Language and Systems: Proceedings of the European Symposium on Programming*, volume 1782 of *LNCS*, pages 366–381. Springer, Mar./Apr. 2000.

M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 132(2): 109–176, Feb. 1997.

M. Tofte and 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*, pages 188–201, New York, NY, USA, Jan. 1994. ACM Press.

M. Tofte, L. Birkedal, M. Elsman, and N. Hallenberg. A retrospective on region-based memory management. *Higher-Order and Symbolic Computation*, 17(3):245–265, 2004.

D. Walker and K. Watkins. On regions and linear types. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*, pages 181–192, New York, NY, USA, Oct. 2001. ACM Press.

D. Walker, K. Crary, and G. Morrisett. Typed memory management via static capabilities. *ACM Trans. Prog. Lang. Syst.*, 22(4):701–771, July 2000.

Appendix

Language Syntax

$$
\begin{array}{lcll}
\textbf{Value} & v & ::= & f \mid n \mid (v,\ldots,v) \mid \texttt{rgn}_\imath \mid \texttt{loc}_l \mid \texttt{pack}\ r, v\ \texttt{as}\ \tau \\
\textbf{Expressions} & e & ::= & x \mid n \mid (e,\ldots,e) \mid \texttt{prj}_n\ e \mid \texttt{newrgn}\ \rho, x\ \texttt{at}\ e\ \texttt{in}\ e \mid \texttt{rgn}_\imath \mid \texttt{cap}_\eta^\psi\ e \\
 & & \mid & \texttt{new}\ e\ \texttt{at}\ e \mid \texttt{deref}\ e \mid e := e \mid \texttt{loc}_l \mid f \mid (e\ e)^\xi \mid e[r] \mid \texttt{pack}\ r, e\ \texttt{as}\ \tau \\
 & & \mid & \texttt{open}\ e\ \texttt{as}\ \rho, x\ \texttt{in}\ e \\
\textbf{Capability kind} & \psi & ::= & \mathsf{rg} \mid \mathsf{lk} \\
\textbf{Capability op} & \eta & ::= & + \mid - \\
\textbf{Region} & r & ::= & \rho \mid \imath \\
\textbf{Region list} & \epsilon & ::= & \emptyset \mid \epsilon, r \\
\textbf{Capability} & \kappa & ::= & n \mid \overline{n} \mid \bot \mid * \\
\textbf{Effect list} & \gamma & ::= & \emptyset \mid \gamma, r^{\kappa,\kappa} \triangleright \epsilon \mid \gamma, r^{\top} \triangleright \epsilon \\
\textbf{Type} & \tau & ::= & \texttt{int} \mid \texttt{rgn}(r) \mid \langle \tau \times \ldots \times \tau \rangle \mid \texttt{ref}(\tau, r) \\
 & & \mid & \tau \xrightarrow{\gamma \rightarrow \gamma} \tau \mid \forall \rho.\, \tau \mid \exists \rho[\epsilon].\, \tau \\
\textbf{Function} & f & ::= & \lambda\, x.\, e\ \texttt{as}\ \tau \xrightarrow{\gamma \rightarrow \gamma} \tau \mid \Lambda \rho.\, f \\
\textbf{Operation} & \xi & ::= & \mathsf{seq} \mid \mathsf{par} \\
\textbf{Program} & P & ::= & \texttt{def}\ x = f; \mid \texttt{def}\ x = f; P
\end{array}
$$

Substitution Relation

Definition 1 (Variable Substitution)

$$
\begin{array}{lcll}
x_1[v/x] & = & v & x_1 \equiv x \\
 & \mid & x_1 & \textit{otherwise}
\end{array}
$$

Definition 2 (Region substitution)

$$
\begin{array}{lcll}
r_1[r/\rho] & = & r & r_1 \equiv \rho \\
 & \mid & r_1 & \textit{otherwise}
\end{array}
$$

Definition 3 (Expression Substitution - Term Variable)

$$
\begin{array}{lcll}
e[v/x] & = & x[v/x] \mid n & \\
 & \mid & (e_1[v/x],\ldots,e_n[v/x]) \mid \texttt{prj}_n\ e_1[v/x] \mid \texttt{rgn}_\imath \mid \texttt{cap}_\eta^\psi\ e_1[v/x] & \\
 & \mid & \texttt{new}\ e_1[v/x]\ \texttt{at}\ e_2[v/x] \mid \texttt{deref}\ e_1[v/x] \mid e_1[v/x] := e_2[v/x] \mid \texttt{loc}_l \mid f \mid (e_1[v/x]\ e_2[v/x])^\xi \mid (e_1[v/x])[& \\
 & \mid & \texttt{pack}\ r, e_1[v/x]\ \texttt{as}\ \tau \mid \texttt{open}\ e_1[v/x]\ \texttt{as}\ \rho, y\ \texttt{in}\ e_2[v/x] & y \not\equiv x \\
 & \mid & \texttt{newrgn}\ \rho, y\ \texttt{at}\ e_1[v/x]\ \texttt{in}\ e_2[v/x] & y \not\equiv x
\end{array}
$$

Definition 4 (Expression Substitution - Type Variable)

$$
\begin{array}{lcll}
f & = & \lambda\, x.\, e[r/\rho]\ \texttt{as}\ \tau_1[r/\rho] \xrightarrow{\gamma_1[r/\rho] \rightarrow \gamma_2[r/\rho]} \tau_2[r/\rho] \mid \Lambda \rho'.\, f[r/\rho] & \rho' \not\equiv \rho \\
e[r/\rho] & = & x \mid n \mid (e_1[r/\rho],\ldots,e_n[r/\rho]) \mid \texttt{prj}_n\ e_1[r/\rho] \mid \texttt{rgn}_\imath \mid \texttt{cap}_\eta^\psi\ e_1[r/\rho] & \\
 & \mid & \texttt{new}\ e_1[r/\rho]\ \texttt{at}\ e_2[r/\rho] \mid \texttt{deref}\ e_1[r/\rho] \mid e_1[r/\rho] := e_2[r/\rho] \mid \texttt{loc}_l \mid f[r/\rho] \mid (e_1[r/\rho]\ e_2[r/\rho])^\xi \mid (e_1[r/ & \\
 & \mid & \texttt{pack}\ r_1, e_1[r/\rho]\ \texttt{as}\ \tau[r/\rho] \mid \texttt{open}\ e_1[r/\rho]\ \texttt{as}\ \rho', x\ \texttt{in}\ e_1[r/\rho] & \rho' \not\equiv \rho \\
 & \mid & \texttt{newrgn}\ \rho', x\ \texttt{at}\ e_1[r/\rho]\ \texttt{in}\ e_1[r/\rho] & \rho' \not\equiv \rho
\end{array}
$$

Definition 5 (Type Substitution)

$$
\begin{array}{lcll}
\tau[r_1/\rho] & = & \texttt{int} \mid \texttt{rgn}(r[r_1/\rho]) \mid \langle \tau_1[r_1/\rho] \times \ldots \times \tau_n[r_1/\rho] \rangle \mid \texttt{ref}(\tau[r_1/\rho], r[r_1/\rho]) & \\
 & \mid & \tau_1[r_1/\rho] \xrightarrow{\gamma_1[r_1/\rho] \rightarrow \gamma_2[r_1/\rho]} \tau_2[r_1/\rho] & \\
 & \mid & \forall \rho'.\, \tau[r_1/\rho] & \rho' \not\equiv \rho \\
 & \mid & \exists \rho'[\epsilon[r_1/\rho]].\, \tau[r_1/\rho] & \rho' \not\equiv \rho
\end{array}
$$

Definition 6 (**Γ Substitution - Type Variable)**

$$\mathit{\Gamma}[r/\rho] \quad = \quad \emptyset \mid \Gamma_1[r/\rho], x : \tau[r/\rho]$$

Definition 7 (**γ Substitution)**

$$\begin{array}{rcl}\gamma[r_{\mathit{1}}/\rho] & = & \emptyset \mid \gamma_1[r_1/\rho], r[r_1/\rho]^{\kappa,\kappa} \triangleright \epsilon_2[r_1/\rho] \\ & \mid & \gamma_1[r_1/\rho], r[r_1/\rho]^{\top} \triangleright \epsilon_2[r_1/\rho]\end{array}$$

Definition 8 (**ϵ Substitution)**

$$\epsilon[r_{\mathit{1}}/\rho] \quad = \quad \emptyset \mid \epsilon_1[r_1/\rho], r[r_1/\rho]$$

Operational Semantics

Helper Judgements and Abbreviations

Definition 9 (Region header-related functions)

Obtain region count rc: $rc((\imath, n_1, n_2, n_3)) \equiv n_1$

Obtain lock count lc: $lc((\imath, n_1, n_2, n_3)) \equiv n_2$

Obtain region name rn: $rn((\imath, n_1, n_2, n_3)) \equiv \imath$

Obtain lock name ln: $ln(\imath, n_1, n_2, n_3)) \equiv n_3$

Modify region count mrc: $mrc(I, k) \equiv (rn(I), rc(I) + k, lc(I), ln(I))$ where $rc(I) + k \geq 0$ and $rc(I) > 0$

Modify lock count mrc: $mlc(I, k) \equiv (rn(I), rc(I), lc(I) + k, ln(I))$ where $lc(I) + k \geq 0$ and $lc(I) > 0$

Modify lock name mln: $mln(I, k) \equiv (rn(I), rc(I), lc(I), k)$ where $k \geq 0$

Acquire/Release lock ar: $ar((\imath, n_1, n_2, n_3), n_A, k) \equiv \begin{cases} (\imath, \mathtt{n_1}, \mathtt{n_2} + \mathtt{k}, \mathtt{n_A}) & \text{if } n_3 \equiv n_A \not\equiv 0, n_2 + k \geq 1 \\ (\imath, \mathtt{n_1}, \mathtt{0}, \mathtt{0}) & \text{if } n_3 \equiv n_A \not\equiv 0, n_2 + k = 0 \\ (\imath, \mathtt{n_1}, \mathtt{n_2} + \mathtt{k}, \mathtt{n_A}) & \text{if } n_3 \equiv 0, n_A \not\equiv 0, n_2 \equiv 0, k \geq 1 \\ (\imath, \mathtt{n_1}, \mathtt{n_2}, \mathtt{n_3}) & \text{if } n_3 \not\equiv n_A, n_2 * n_3 > 0 \\ (\imath, \mathtt{n_1}, \mathtt{0}, \mathtt{0}) & \text{if } n_3 \equiv 0, n_2 \equiv -1, k \equiv -1 \end{cases}$

Definition 10 (Store-related Functions and Invariants)

$$dom_I(S) \equiv \{\imath \mid (\imath, n_1, n_2, n_3) : (H, S_1) \in S \vee \imath \in dom_I(S_1)\}$$

$$dom_\ell(S) \equiv \{\ell \mid (I : (H, S_1) \in S \wedge \ell \mapsto v \in H) \vee \ell \in dom_\ell(S_1)\}$$

$$dom_H(S) \equiv \{\ell \mapsto v \mid (I : (H, S_1) \in S \wedge \ell \mapsto v \in H) \vee \ell \mapsto v \in dom_H(S_1)\}$$

$$children(S, \imath) \equiv \{I_1 \mid I : (H, S_1) \in S \wedge ((rn(I) \equiv \imath \wedge I_1 \in dom_I(S_1)) \vee I_1 \in children(S_1, \imath))\}$$

$$parents(S, \imath) \equiv \{I \mid I \in dom_I(S) \wedge \imath \in \{rn(I) \mid children(S, rn(I))\}\}$$

$$live(S, \imath) \equiv \forall I \in parents(S, \imath).rc(I) > 0 \wedge \exists I \in dom_I(S).rn(I) \equiv \imath \wedge rc(I) > 0$$

Definition 11 (Search and Update region in s)

$$\frac{rn(I) \equiv rn(I') \qquad \vdash_= S_1 = S', I' : (H', S_2') \qquad \vdash_= S_3 = S', I : (H, S_2)}{S_1; I; H; S_2 \vdash_{su} S_3; I'; H'; S_2'}$$

$$\frac{\begin{array}{c}\vdash_= S_1 = \emptyset, I_1 : (H_1, S_{a1}), \ldots, I_k : (H_k, S_{ak} \\ rn(I_1), \ldots, rn(I_k); rn(I) \vdash_\in false \\ S_{a1}, \ldots, S_z, \ldots, S_{ak}; I; H; S_2 \vdash_{su} S_{a1}, \ldots, S_z', \ldots, S_{ak} \\ \vdash_= S_3 = \emptyset, I_1 : (H_1, S_{a1}), \ldots, I_z : (H_z, S_z), \ldots, I_k :\end{array}}{S_1; I; H; S_2 \vdash_{su} S_3; I'; H'; S_2'}$$

Definition 12 (Capability cap)

$$\frac{\begin{array}{c} S; I'; H; S'' \vdash_{su} S'''; I; H; S' \qquad rn(I) \equiv \imath \qquad live(S, \imath) \\ \forall I \in children(S, \imath).rc(I) = 0 \quad mrc(I, -1) \equiv I' \quad rc(I) \equiv 1 \end{array}}{\text{updcap}_{n_a}(\imath, S, \mathsf{rg}, -1) \equiv S'''} \qquad \frac{\begin{array}{c} \exists I \in children(S, \imath).rc(I) \neq 0 \\ \exists I \in dom(S).rn(I) \equiv \imath \wedge rc(I) = 1 \\ live(S, \imath) \end{array}}{\text{updcap}_{n_a}(\imath, S, \mathsf{rg}, -1) \equiv S}$$

$$\frac{\begin{array}{c} S; I'; H; S'' \vdash_{su} S'''; I; H; S' \qquad rn(I) \equiv \imath \\ I' \equiv mrc(I, \eta) \quad rc(I') > 0 \quad rc(I) > 0 \quad live(S, \imath) \end{array}}{\text{updcap}_{n_a}(\imath, S, \mathsf{rg}, \eta) \equiv S'''} \qquad \frac{\begin{array}{c} S; I_2; H; S_1 \vdash_{su} S_2; I_1; H; S_1 \quad rn(I_2) \equiv \imath \quad ar(I_1, n_a, \eta) \equiv I_2 \\ live(S, \imath) \end{array}}{\text{updcap}_{n_a}(\imath, S, \mathsf{lk}, \eta) \equiv S_2}$$

Definition 13 (Abbreviations)

$$\text{newrgn}(\jmath, S) \equiv (k, S'') \qquad \text{if } k \notin \{rn(I) \mid dom_I(S)\} \wedge live(S, \jmath) \wedge S; I'; H'; S', (k, 1, -1, 0) : (\emptyset, \emptyset) \vdash_{su} S''; I'; H'; S' \wedge rn(I') \equiv \jmath$$

$$\text{updcap}_\imath(\jmath, S, \psi, \eta) \equiv S' \qquad \text{if updcap}_\imath(\jmath, S, \psi, \eta) \equiv S'$$

$$\text{alloc}(\jmath, S, v) \equiv (\ell, S') \qquad \text{if } \ell \notin dom_\ell(S) \wedge live(S, \jmath) \wedge S; I; H, \ell \mapsto v; S' \vdash_{su} S''; I; H; S' \wedge rn(I) \equiv \kappa$$

$$\text{xupdate}_\imath(S, \ell, v) \equiv (S'', v_1) \qquad \text{if } S; I; H', \ell \mapsto v; S' \vdash_{su} S''; I; H', \ell \mapsto v_1; S' \wedge live(S, rn(I)) \wedge lc(I) \not\equiv -1 \Rightarrow ln(I) \equiv \imath \wedge lc(I) > 0$$

$$\text{lookup}_\imath(S, \ell) \equiv v_1 \qquad \text{if } (S', v_1) \equiv xupdate(S, \ell, v)$$

$$\text{update}_\imath(S, \ell, v) \equiv S' \qquad \text{if } (S', v_1) \equiv xupdate(S, \ell, v)$$

$$\text{threadid(T)} \equiv \{\imath \mid \imath : e \in T\}$$

Operational Semantics

Definition 14 (Operational Semantics Syntax)

Contents	H	$::=$	$\emptyset \mid H, \ell \mapsto v$
Run-time region	I	$::=$	$(\imath, n_1, n_2, n_3)$
Region list	S	$::=$	$\emptyset \mid S, I : (H, S)$
Threads	T	$::=$	$\emptyset \mid T, n : e$
Configuration	C	$::=$	$S; T$

Definition 15 (Operational Semantics - Evaluation Context)

$$\begin{array}{lll} E & ::= & \square \mid (v, \ldots, E, \ldots, e) \mid \texttt{prj}_\imath\ E \mid (E\ e)^\xi \mid (v\ E)^\xi \mid E\,[r] \mid \texttt{newrgn}\ \rho, x\ \texttt{at}\ E\ \texttt{in}\ e \mid \texttt{cap}_\eta^\psi\ E \\ & \mid & \texttt{new}\ E\ \texttt{at}\ e \mid \texttt{new}\ v\ \texttt{at}\ E \mid \texttt{deref}\ E \mid E := e \mid v := E \mid \texttt{pack}\ r, E\ \texttt{as}\ \tau \mid \texttt{open}\ E\ \texttt{as}\ \rho, x\ \texttt{in}\ e \end{array}$$

Definition 16 (Operational Semantics - Thread and Expression Reduction relation)

$$\frac{S; e \rightarrow_\imath^P S'; e'}{S; T_1, \imath : E[e], T_2 \rightsquigarrow^P S'; T_1, \imath : E[e'], T_2}\ \textit{(E-S)} \qquad \frac{}{S; T_1, \imath : (), T_2 \rightsquigarrow^P S; T_1, T_2}\ \textit{(E-T)}$$

$$\frac{\jmath \text{ is a fresh thread identifier}}{S; T_1, \imath : E[((\lambda x.\, e\ \texttt{as}\ \tau)\ v)^{\mathsf{par}}], T_2 \rightsquigarrow^P S; T_1, \imath : E[()], T_2, \jmath : e[v/x]}\ \textit{(E-SN)} \quad \frac{1 \leq i \leq n}{S; \texttt{prj}_i\ (v_1, \ldots, v_n) \rightarrow_\imath^P S; v_i}\ \textit{(E-P)}$$

$$\frac{(\texttt{def}\ x = f) \in P}{S; x \rightarrow_\imath^P S; f}\ \textit{(E-F)} \qquad \frac{}{S; ((\lambda x.\, e\ \texttt{as}\ \tau)\ v)^{\mathsf{seq}} \rightarrow_\imath^P S; e[v/x]}\ \textit{(E-A)} \qquad \frac{}{S; (\Lambda\rho.\, f)[r] \rightarrow_\imath^P S; f[r/\rho]}\ \textit{(E-RP)}$$

$$\frac{(\ell, S') = \text{alloc}(\jmath, S, v)}{S; \texttt{new}\ v\ \texttt{at}\ \texttt{rgn}_\jmath \rightarrow_\imath^P S'; \texttt{loc}_\ell}\ \textit{(E-NRF)} \qquad \frac{v = \text{lookup}_\imath(S, \ell)}{S; \texttt{deref}\ \texttt{loc}_\ell \rightarrow_\imath^P S; v}\ \textit{(E-D)} \qquad \frac{S' = \text{update}_\imath(S, \ell, v)}{S; \texttt{loc}_\ell := v \rightarrow_\imath^P S'; ()}\ \textit{(E-AS)}$$

$$\frac{(k, S') = \text{newrgn}(\jmath, S)}{S; \texttt{newrgn}\ \rho, x\ \texttt{at}\ \texttt{rgn}_\jmath\ \texttt{in}\ e \rightarrow_\imath^P S'; e[k/\rho][\texttt{rgn}_k/x]}\ \textit{(E-NR)} \qquad \frac{S' = \text{updcap}_\imath(\jmath, S, \psi, \eta)}{S; \texttt{cap}_\eta^\psi\ \texttt{rgn}_\jmath \rightarrow_\imath^P S'; ()}\ \textit{(E-C)}$$

$$\frac{\text{updcap}_\imath(\jmath, S, \psi, \eta) = S}{S; \texttt{cap}_\eta^\psi\ \texttt{rgn}_\jmath \rightarrow_\imath^P S; \texttt{cap}_\eta^\psi\ \texttt{rgn}_\jmath}\ \textit{(E-CB)} \qquad \frac{}{S; \texttt{open}\ (\texttt{pack}\ \jmath, v\ \texttt{as}\ \tau)\ \texttt{as}\ \rho, x\ \texttt{in}\ e \rightarrow_\imath^P S; e[\jmath/\rho][v/x]}\ \textit{(E-OR)}$$

Table 1: Capability conversion – Function $\mathrm{out}(\psi, \xi, \kappa_1, \kappa_2, \kappa_3)$

ψ	ξ	κ_1	κ_2	κ_{in}	$\mathrm{out}(\kappa_{in}, \psi, \xi, \kappa_1, \kappa_2)$	constraints
any	seq	n_1	n_2	n_1	n_2	
any	seq	$\overline{n_1}$	$\overline{n_2}$	n_3	$n_2 + n_3 - n_1$	$n_1 \leq n_3$
any	seq	$\overline{n_1}$	$\overline{n_2}$	$\overline{n_3}$	$\overline{n_2 + n_3 - n_1}$	$n_1 \leq n_3$
any	seq	$*$	$*$	n_3	n_3	$0 < n_3$
any	seq	$*$	$*$	$\overline{n_3}$	$\overline{n_3}$	$0 < n_3$
any	seq	$*$	$*$	$*$	$*$	
lk	seq	$\perp$	$\perp$	$\perp$	$\perp$	
lk	seq	$\perp$	0	$\perp$	0	
lk	seq	$*$	$*$	$\perp$	$\perp$	
rg	par	n_1	0	n_2	$n_1 - n_2$	$0 \leq n_1 \leq n_2$
rg	par	n_1	0	$\overline{n_2}$	$\overline{n_1 - n_2}$	$0 \leq n_1 \leq n_2$
lk	par	0	0	n_1	n_1	
lk	par	0	0	$\overline{n_1}$	$\overline{n_1}$	
lk	par	$\perp$	0	$\perp$	0	

Static Semantics

Static Semantics Helper Judgements and Abbreviations

Definition 17 (Abbreviations)

$$\mathrm{dom}(\gamma) \equiv \{r \mid r^{\kappa_1,\kappa_2} \triangleright \epsilon_1 \in \gamma\} \cup \{r \mid r^{\top} \triangleright \epsilon_1 \in \gamma\}$$

$$\mathrm{select}(\gamma, \mathrm{r}_1) \equiv \{r^{\kappa_1,\kappa_2} \triangleright \epsilon_1 \mid r^{\kappa_1,\kappa_2} \triangleright \epsilon_1 \in \gamma \wedge r \equiv r_1\} \cup \{r^{\top} \triangleright \epsilon_1 \mid r^{\top} \triangleright \epsilon_1 \in \gamma \wedge r \equiv r_1\}$$

$$\mathrm{linear}(\gamma) \equiv \{r^{\kappa_1,\kappa_2} \triangleright \epsilon_1 \mid r^{\kappa_1,\kappa_2} \triangleright \epsilon_1 \in \gamma\}$$

$$\mathrm{dom}(\Gamma) \equiv \{x \mid x : \tau \in \Gamma\}$$

$$\mathrm{dom}(\mathrm{M}) \equiv \{\ell \mid \ell \mapsto (\tau, \imath) \in M\}$$

$$\mathrm{dom}(\mathrm{P}) \equiv \{x \mid \mathtt{def}\ x = f \in P\}$$

$$\mathrm{dom}(\delta) \equiv \{\imath \mid \jmath \mapsto \gamma \in \delta \wedge \gamma \not\equiv \emptyset \wedge \imath \in dom(\gamma)\}$$

$$\mathrm{dom_t}(\delta) \equiv \{\jmath \mid \jmath \mapsto \gamma \in \delta \wedge \gamma \not\equiv \emptyset\}$$

$$\mathrm{parents}(\gamma, r) \equiv \{r_1 \mid ((r^{\kappa_1,\kappa_2} \triangleright \epsilon \in \gamma \vee r^{\top} \triangleright \epsilon \in \gamma) \wedge r_1 \in \epsilon) \vee r_1 \in \bigcup_{r_2 \in \epsilon} parents(\gamma, r_2)\}$$

$$\mathrm{children}(\gamma, r) \equiv \{r_1 \mid r_1 \in dom(\gamma) \wedge r \in parents(\gamma, r_1)\}$$

$$\mathrm{live}(\gamma, \epsilon) \equiv \epsilon \not\equiv \emptyset \wedge \forall r \in \epsilon. \forall r_1 \in (\{r\} \cup parents(\gamma, r)).r_1^{\kappa_1,\kappa_2} \triangleright \epsilon \in linear(\gamma) \Rightarrow \kappa_1 \not\equiv 0 \wedge \kappa_1 \not\equiv \bar{0}$$

$$\mathrm{accessible}(\gamma, \epsilon) \equiv live(\gamma, \epsilon) \wedge \forall r \in \epsilon.(r^{\kappa_1,\kappa_2} \triangleright \epsilon_1 \in \gamma \wedge \kappa_2 \not\equiv 0 \wedge \kappa_2 \not\equiv \bar{0}) \vee (r^{\top} \triangleright \epsilon_1 \in \gamma \wedge accessible(\gamma, \epsilon_1))$$

Definition 18 (Effect summarization judgement sum)

$$add(\kappa_1, \kappa_2, n_3) \equiv \begin{cases} \overline{n_1 + n_2}; n_3 & \text{if } \kappa_1 \equiv \overline{n_1} \ \wedge \ \kappa_2 \equiv \overline{n_2} \\ \overline{n_1 + (1 - n_3)}; 1 & \text{if } \kappa_1 \equiv \overline{n_1} \ \wedge \ \kappa_2 \equiv * \\ \overline{n_1 + (1 - n_3)}; 1 & \text{if } \kappa_2 \equiv \overline{n_1} \ \wedge \ \kappa_1 \equiv * \\ \overline{(1 - n_3)}; 1 & \text{if } \kappa_1 \equiv * \ \wedge \ \kappa_2 \equiv * \end{cases}$$

$$\mathrm{sum}(\gamma, n_1, n_2) \equiv \begin{cases} \emptyset & \text{if } \gamma \equiv \emptyset \\ \emptyset, r^{\kappa_1,\kappa_2} \triangleright \epsilon & \text{if } \gamma \equiv \emptyset, r^{\kappa_1,\kappa_2} \triangleright \epsilon \\ \mathrm{sum}(\gamma' \cup r^{\kappa_5,\kappa_6} \triangleright \epsilon_1 \cap \epsilon_2, n_3, n_4) & \text{if } \gamma \equiv \gamma', r^{\kappa_1,\kappa_2} \triangleright \epsilon_1, r^{\kappa_3,\kappa_4} \triangleright \epsilon_2 \wedge add(\kappa_1, \kappa_3, n_1) \equiv (\kappa_5, n_3) \wedge add(\kappa_2, \kappa_4, n_2) \equiv \end{cases}$$

$$notzero(\rho^{\kappa_1,\kappa_2} \triangleright \epsilon) \equiv \begin{cases} \rho^{\kappa_1,\kappa_2} \triangleright \epsilon & \text{if } \kappa_1 \not\equiv 0 \vee \kappa_1 \not\equiv \bar{0} \\ \emptyset & \text{otherwise} \end{cases}$$

$$\mathrm{sumt}(\gamma) \equiv (\textstyle\bigcup_{r \in dom(linear(\gamma))} \mathrm{sum}(select(linear(\gamma), r), 0, 0)) \cup (\gamma \setminus (linear(\gamma) \cup \{\imath^{\top} \triangleright \epsilon \mid \imath^{\top} \triangleright \epsilon \in \gamma\}))$$

$$\overline{\gamma} \equiv \textstyle\bigcup_{x \in \mathrm{sumt}(\gamma)} notzero(x)$$

Definition 19 (Output constraint judgement out)

$$\text{out}(\xi,\psi,\kappa_1,\kappa_3,\kappa_5) \equiv \text{see Table 1}$$

$$\text{tout}(\xi,\gamma_1,\gamma_2,\gamma_3) \equiv \begin{cases} (\emptyset,\emptyset) & \text{if } \gamma_3 \equiv \gamma_2 \equiv \emptyset \\ (y \cup r^{\top} \triangleright \epsilon, z) & \text{if } (y,z) \equiv \text{tout}(\xi, \gamma_1 \setminus x, \gamma_2 \setminus x, \gamma_3 \setminus x) \wedge x \equiv r^{\top} \triangleright \epsilon \wedge x \in \gamma_1 \cap \gamma_2 \cap \gamma_3 \\ (y \cup a, z \cup (\epsilon \setminus \epsilon_1)) & \text{if } (y,z) \equiv \text{tout}(\xi, \gamma_1', \gamma_2', \gamma_3') \wedge \gamma_1 = \gamma_1', r^{\kappa_1,\kappa_2} \triangleright \epsilon_1 \wedge \gamma_2 = \gamma_2', r^{\kappa_3,\kappa_4} \triangleright \epsilon_1 \wedge \gamma_3 = \gamma_3', r^{\kappa_5,\kappa_6} \triangleright \\ & \qquad \epsilon_1 \subseteq \epsilon \wedge \kappa_7 \equiv \text{out}(\mathsf{rg},\psi,\kappa_1,\kappa_3,\kappa_5) \wedge \kappa_8 \equiv \text{out}(\mathsf{lk},\psi,\kappa_2,\kappa_4,\kappa_6) \wedge \\ & \qquad (\kappa_1 \not\equiv 0 \vee \kappa_1 \not\equiv \bar{0}) \wedge a \equiv \text{notzero}(r^{\kappa_7,\kappa_8} \triangleright \epsilon) \wedge \xi; \kappa_6 \equiv \mathsf{par}; \bot \Rightarrow \kappa_5 \equiv n \equiv \kappa_1 \end{cases}$$

Definition 20 (Invariant judgement)

$$\text{iszero}(\kappa_1,\kappa_2) \equiv (\kappa_1\kappa_2 \equiv 00 \vee \kappa_1\kappa_2 \equiv \bar{0}\bar{0})$$

$$\text{locked}(\kappa_1) \equiv \kappa_1 \not\equiv 0 \wedge \kappa_1 \not\equiv \bar{0}$$

$$invrg(\gamma_1) \equiv \forall r \in r^{\kappa_1,\kappa_2} \triangleright \epsilon \in linear(\gamma_1).(iszero(\kappa_1,\kappa_2) \Rightarrow \forall r_1^{\kappa_3,\kappa_4} \triangleright \epsilon_1 \in linear(children(\gamma_1,r)).iszero(\kappa_3,\kappa_4)) \wedge (\neg iszero(\kappa_1,\kappa_2) =$$

$$invlk(\gamma_1) \equiv \forall r \in r^{\kappa_1,\kappa_2} \triangleright \epsilon \in linear(\gamma_1).(locked(\kappa_2) \Rightarrow \forall r_1^{\kappa_3,\kappa_4} \triangleright \epsilon_1 \in linear(children(\gamma_1,r)).locked(\kappa_3,\kappa_4))$$

$$inv(\gamma_1) \equiv invrg(\gamma_1) \wedge invlk(\gamma_2)$$

Definition 21 (Environment output constraint judgement)

$$\text{output}(\gamma_{in},\xi,\gamma_1,\gamma_2) \equiv \gamma_{out} \text{ if } (\gamma_{out},\epsilon_1) \equiv \text{tout}(\xi, \text{sumt}(\gamma_1), \text{sumt}(\gamma_2), \gamma_{\text{in}}) \wedge \text{live}(\gamma_{in},\epsilon_1) \wedge \text{live}(\gamma_{out},\epsilon_1) \wedge \text{inv}(\gamma_{out})$$

Definition 22 (Capability mod)

$$mod_{\eta}(\kappa) \equiv \begin{cases} n_2 + \eta & \text{if } \kappa \equiv n \ \wedge\ n + \eta \geq 0 \\ \overline{n_2 + \eta} & \text{if } \kappa \equiv \bar{n} \ \wedge\ n + \eta \geq 0 \\ 0 & \text{if } \kappa\eta \equiv -1, \bot \end{cases}$$

$$\text{modcap}(\gamma',\psi,\eta,r) \equiv \begin{cases} \gamma_2 & \text{if } \psi \equiv \mathsf{rg} \wedge x \equiv r^{\kappa_1,\kappa_2} \triangleright \epsilon_1 \wedge x \in \gamma_1 \wedge \kappa_1' \equiv \text{mod}_{\eta}(\kappa_1) \wedge \gamma_2 \equiv \gamma_1[r^{\kappa_1',\kappa_2} \triangleright \epsilon_1/x] \wedge \text{inv}(\gamma_2) \wedge \text{live}(\gamma_1,\{r\}) \\ & \qquad \kappa_1' \equiv 0 \Rightarrow \kappa_2 \equiv 0 \wedge \kappa_1' \equiv \bar{0} \Rightarrow \kappa_2 \equiv \bar{0} \\ \gamma_2 & \text{if } \psi \equiv \mathsf{lk} \wedge x \equiv r^{\kappa_1,\kappa_2} \triangleright \epsilon_1 \wedge x \in \gamma_1 \wedge \kappa_2' \equiv \text{mod}_{\eta}(\kappa_2) \wedge \gamma_2 \equiv \gamma_1[r^{\kappa_1,\kappa_2'} \triangleright \epsilon_1/x] \wedge \text{inv}(\gamma_2) \wedge \text{live}(\gamma_1,\{r\}) \\ & \qquad \eta; \kappa_2 \equiv -1; \bot \Rightarrow \kappa_1 \equiv n \end{cases}$$

Static Semantics Syntax

Region List	R	$::=$	$\emptyset \mid R, \imath$
Type variable list	$\varDelta$	$::=$	$\emptyset \mid \Delta, \rho$
Memory List	M	$::=$	$\emptyset \mid M, \ell \mapsto (\tau, \imath)$
Variable list	$\varGamma$	$::=$	$\emptyset \mid \Gamma, x : \tau$

Typing Context Well-formedness Judgements

$$Well-formednessJudgement$$

Region Well-formedness

$$\frac{}{R;\Delta,\rho \vdash_R \rho} \qquad \frac{}{R,\imath;\Delta \vdash_R \imath}$$

Effect Well-formedness

$$\frac{}{R;\Delta \vdash_{\epsilon} \emptyset} \qquad \frac{R;\Delta \vdash_{\epsilon} \epsilon_1 \quad R;\Delta \vdash_R r}{R;\Delta \vdash_{\epsilon} \epsilon_1\,, r}$$

Constraint Well-formedness

$$\frac{}{R;\Delta \vdash_{\gamma} \emptyset} \qquad \frac{\begin{array}{cc} R;\Delta \vdash_{\gamma} \gamma_1 & R;\Delta \vdash_R r \\ R;\Delta \vdash_{\epsilon} \epsilon & \kappa_1 \not\equiv \bot \end{array}}{R;\Delta \vdash_{\gamma} \gamma_1, r^{\kappa_1,\kappa_2} \triangleright \epsilon} \qquad \frac{\begin{array}{cc} R;\Delta \vdash_{\gamma} \gamma_1 & R;\Delta \vdash_R r \\ \multicolumn{2}{c}{R;\Delta \vdash_{\epsilon} \epsilon} \end{array}}{R;\Delta \vdash_{\gamma} \gamma_1, r^{\top} \triangleright \epsilon}$$

Type Well-formedness

$$\frac{}{R;\Delta \vdash \texttt{int}} \qquad \frac{R;\Delta \vdash_R r}{R;\Delta \vdash \texttt{rgn}(r)} \qquad \frac{\forall \imath \in [0, n-1].R;\Delta \vdash \tau_{\imath}}{R;\Delta \vdash (\tau_0,\ldots,\tau_{n-1})} \qquad \frac{R;\Delta,\rho \vdash \tau}{R;\Delta \vdash \forall \rho.\,\tau}$$

$$\frac{R;\Delta \vdash \tau \qquad R;\Delta \vdash_R r}{R;\Delta \vdash \mathtt{ref}(\tau, r)} \qquad \frac{\begin{array}{c} R;\Delta \vdash \tau_1 \qquad R;\Delta \vdash_\gamma \gamma_1 \\ R;\Delta \vdash \tau_2 \qquad R;\Delta \vdash_\gamma \gamma_2 \end{array}}{R;\Delta \vdash \tau_1 \xrightarrow{\gamma_1 \rightarrow \gamma_2} \tau_2} \qquad \frac{R;\Delta,\rho \vdash \tau \qquad R;\Delta \vdash_\epsilon \epsilon_1}{R;\Delta \vdash \exists\rho[\epsilon_1].\,\tau} \qquad \frac{}{R;\Delta \vdash \langle\rangle}$$

Variable Context Well-formedness $$\frac{}{R;\Delta \vdash_\Gamma \emptyset} \qquad \frac{R;\Delta \vdash_\Gamma \Gamma_1 \qquad R;\Delta \vdash \tau_1 \qquad x \notin \mathit{dom}(\Gamma_1)}{R;\Delta \vdash_\Gamma \Gamma_1, x : \tau_1}$$

Memory Location Well-formedness $$\frac{}{R \vdash_M \emptyset} \qquad \frac{R \vdash_M M_1 \qquad R;\emptyset \vdash_T \mathtt{ref}(\tau_1, \imath) \qquad \ell \notin \mathit{dom}(M_1)}{R \vdash_M M_1, \ell \mapsto (\tau_1, \imath)}$$

Program Typing Context Well-formedness $$\frac{\begin{array}{c} R \vdash_M M \qquad R;\Delta \vdash_\Gamma \Gamma \qquad R;\Delta \vdash_\gamma \gamma \qquad \mathit{dom}(\gamma') \subseteq \mathit{dom}(\gamma) \\ \P \vdash_P \Gamma_0 \qquad\qquad\qquad \mathit{dom}(\Gamma_0) \cap \mathit{dom}(\Gamma) \equiv \emptyset \end{array}}{\vdash_D R; M; \Delta; \Gamma; \gamma; \gamma'; P}$$

Program Well-formedness $$\frac{\mathit{dom}(P) = \mathit{dom}(\Gamma_0) \qquad \forall \mathtt{def}\ x = f \in P. \emptyset;\emptyset;\emptyset;\emptyset \vdash f : \Gamma_0(x) \mathbin{\&} (\emptyset;\emptyset)}{P \vdash_P \Gamma_0}$$

Expression Typing Judgements

$$\frac{(x \mapsto \tau) \in \Gamma, \Gamma_0 \qquad \vdash_D R; M; \Delta; \Gamma; \gamma; \gamma; P \qquad P \vdash \Gamma_0}{R; M; \Delta; \Gamma \vdash x : \tau \mathbin{\&} (\gamma;\gamma)}\ \textit{(T-V)} \qquad \frac{\vdash_D R; M; \Delta; \Gamma; \gamma; \gamma; P}{R; M; \Delta; \Gamma \vdash n : \mathtt{int} \mathbin{\&} (\gamma;\gamma)}\ \textit{(T-I)}$$

$$\frac{\vdash_D R; M; \Delta; \Gamma; \gamma; \gamma; P}{R; M; \Delta; \Gamma \vdash () : \langle\rangle \mathbin{\&} (\gamma;\gamma)}\ \textit{(T-U)} \qquad \frac{\begin{array}{c} \vdash_D R; M; \Delta; \Gamma; \gamma; \gamma; P \\ R;\Delta \vdash_R \imath \end{array}}{R; M; \Delta; \Gamma \vdash \mathtt{rgn}_\imath : \mathtt{rgn}(\imath) \mathbin{\&} (\gamma;\gamma)}\ \textit{(T-R)} \qquad \frac{\begin{array}{c} \vdash_D R; M; \Delta; \Gamma; \gamma; \gamma; P \\ (\ell \mapsto (\tau, \imath)) \in M \end{array}}{R; M; \Delta; \Gamma \vdash \mathtt{loc}_l : \mathtt{ref}(\tau, \imath) \mathbin{\&} (\gamma;\gamma)}$$

$$\frac{\begin{array}{c} \vdash_D R; M; \Delta; \Gamma; \gamma; \gamma; P \qquad\qquad R;\Delta \vdash \tau \\ \tau \equiv \tau_1 \xrightarrow{\gamma_1 \rightarrow \gamma_2} \tau_2 \qquad R; M; \Delta; \Gamma, x : \tau_1 \vdash e : \tau_2 \mathbin{\&} (\overline{\gamma_1};\overline{\gamma_2}) \end{array}}{R; M; \Delta; \Gamma \vdash \lambda x.\, e\ \mathtt{as}\ \tau : \tau \mathbin{\&} (\gamma;\gamma)}\ \textit{(T-F)} \qquad \frac{R; M; \Delta, \rho; \Gamma \vdash f : \tau \mathbin{\&} (\gamma;\gamma)}{R; M; \Delta; \Gamma \vdash \Lambda\rho.\, f : \forall\rho.\,\tau \mathbin{\&} (\gamma;\gamma)}\ \textit{(T-RF)}$$

$$\frac{R; M; \Delta; \Gamma \vdash e_i : \tau_i \mathbin{\&} (\gamma_i;\gamma_{i+1}) \ \text{ forall } 0 \leq i < n}{R; M; \Delta; \Gamma \vdash (e_0, \ldots, e_{n-1}) : \langle \tau_0 \times \ldots \times \tau_{n-1} \rangle \mathbin{\&} (\gamma_0;\gamma_n)}\ \textit{(T-Tu)} \qquad \frac{\begin{array}{c} 0 \leq i < n \\ R; M; \Delta; \Gamma \vdash e : \langle \tau_0 \times \ldots \times \tau_{n-1} \rangle \mathbin{\&} (\gamma;\gamma') \end{array}}{R; M; \Delta; \Gamma \vdash \mathtt{prj}_i\ e : \tau_i \mathbin{\&} (\gamma;\gamma')}\ \textit{(T-Pro}$$

$$\frac{\begin{array}{c} R; M; \Delta; \Gamma \vdash e_1 : \tau_1 \xrightarrow{\gamma_1 \rightarrow \gamma_2} \tau_2 \mathbin{\&} (\gamma;\gamma') \\ R; M; \Delta; \Gamma \vdash e_2 : \tau_1 \mathbin{\&} (\gamma';\gamma_{in}) \\ \text{if } \xi \equiv \text{par then } \tau_2 \equiv \langle\rangle \qquad \gamma_{out} = \text{output}(\gamma_{in}, \xi, \gamma_1, \gamma_2) \end{array}}{R; M; \Delta; \Gamma \vdash (e_1\ e_2)^\xi : \tau_2 \mathbin{\&} (\gamma;\gamma_{out})}\ \textit{(T-App)} \qquad \frac{\begin{array}{c} R;\Delta \vdash_R r \\ R; M; \Delta; \Gamma \vdash e : \forall\rho.\,\tau \mathbin{\&} (\gamma;\gamma') \end{array}}{R; M; \Delta; \Gamma \vdash e\ [r] : \tau[r/\rho] \mathbin{\&} (\gamma;\gamma')}\ \textit{(T-RApp)}$$

$$\frac{\begin{array}{c} R; M; \Delta; \Gamma \vdash e_1 : \tau \mathbin{\&} (\gamma;\gamma') \\ R; M; \Delta; \Gamma \vdash e_2 : \mathtt{rgn}(r) \mathbin{\&} (\gamma';\gamma'') \qquad \text{accessible}(\gamma'', \{r\}) \end{array}}{R; M; \Delta; \Gamma \vdash \mathtt{new}\ e_1\ \mathtt{at}\ e_2 : \mathtt{ref}(\tau, r) \mathbin{\&} (\gamma;\gamma'')}\ \textit{(T-NR)} \qquad \frac{R; M; \Delta; \Gamma \vdash e : \mathtt{ref}(\tau, r) \mathbin{\&} (\gamma;\gamma') \qquad \text{accessible}(\gamma', \{r\})}{R; M; \Delta; \Gamma \vdash \mathtt{deref}\ e : \tau \mathbin{\&} (\gamma;\gamma')}$$

$$\frac{R; M; \Delta; \Gamma \vdash e_1 : \mathtt{ref}(\tau, r) \mathbin{\&} (\gamma;\gamma') \qquad R; M; \Delta; \Gamma \vdash e_2 : \tau \mathbin{\&} (\gamma';\gamma'') \qquad \text{accessible}(\gamma'', \{r\})}{R; M; \Delta; \Gamma \vdash e_1 := e_2 : \langle\rangle \mathbin{\&} (\gamma;\gamma'')}\ \textit{(T-A)}$$

$$\frac{\begin{array}{c} R; M; \Delta; \Gamma \vdash e_1 : \mathtt{rgn}(r) \mathbin{\&} (\gamma;\gamma') \qquad \text{live}(\gamma', \{r\}) \qquad R;\Delta \vdash \tau \\ R; M; \Delta, \rho; \Gamma, x : \mathtt{rgn}(\rho) \vdash e_2 : \tau \mathbin{\&} (\gamma', \rho^{1,\perp} \triangleright r;\gamma'') \qquad \rho \notin \text{dom}(\gamma'') \end{array}}{R; M; \Delta; \Gamma \vdash \mathtt{newrgn}\ \rho, x\ \mathtt{at}\ e_1\ \mathtt{in}\ e_2 : \tau \mathbin{\&} (\gamma;\gamma'')}\ \textit{(T-NRG)} \qquad \frac{\begin{array}{c} R; M; \Delta; \Gamma \vdash e_1 : \mathtt{rgn}(r) \mathbin{\&} (\gamma;\gamma') \\ \gamma'' = \text{modcap}(\gamma', \psi, \eta, r) \end{array}}{R; M; \Delta; \Gamma \vdash \mathtt{cap}_\eta^\psi\ e_1 : \langle\rangle \mathbin{\&} (\gamma;\gamma'')}\ \textit{(T-}$$

$$\frac{\begin{array}{c} R;\Delta \vdash \tau \qquad \tau \equiv \exists\rho[\epsilon].\,\tau' \qquad R;\Delta \vdash_R r \qquad r \in \epsilon \\ R; M; \Delta; \Gamma \vdash e : \tau'[r/\rho] \mathbin{\&} (\gamma;\gamma') \end{array}}{R; M; \Delta; \Gamma \vdash \mathtt{pack}\ r, e\ \mathtt{as}\ \tau : \tau \mathbin{\&} (\gamma;\gamma')}\ \textit{(T-EP)} \qquad \frac{\begin{array}{c} R;\Delta \vdash \tau' \qquad\qquad R; M; \Delta; \Gamma \vdash e : \exists\rho[\epsilon].\,\tau \mathbin{\&} (\gamma;\gamma') \\ R; M; \Delta, \rho'; \Gamma, x : \tau[\rho'/\rho] \vdash e' : \tau' \mathbin{\&} (\gamma', \rho'^{\top} \triangleright \epsilon;\gamma'', \rho'^{\top} \triangleright \epsilon) \end{array}}{R; M; \Delta; \Gamma \vdash \mathtt{open}\ e\ \mathtt{as}\ \rho', x\ \mathtt{in}\ e' : \tau' \mathbin{\&} (\gamma;\gamma'')}\ \textit{(T-}$$

Type Safety

Type Safety Syntax

Definition 23 (Program Effect)

Program Effect $\delta \ ::= \ \emptyset \mid \delta, n \mapsto \gamma$

Type Safety Helper Judgements and Abbreviations

Definition 24 (Abbreviations)

$$rgcap(\delta, m, \imath) \equiv \begin{cases} n_1 & \text{if } \imath^{n_1,\kappa_2} \triangleright \epsilon \in \delta(m) \\ 0 & \text{otherwise} \end{cases}$$

$$rsum(\gamma, \imath) \equiv \sum_{m \in dom_t(\delta)} rgcap(\delta, m, \imath)$$

$$q(\delta, \imath) = \bigcup_{m \in dom_t(\delta)} \{m \mapsto \emptyset, \imath^{\kappa_1,\kappa_2} \triangleright \epsilon \mid \imath^{\kappa_1,\kappa_2} \triangleright \epsilon \in \delta(m)\}$$

$$Q_0(\delta) \equiv \forall m \in dom_t(\delta). \forall \imath \in dom(\delta(m)). \imath^{n_1,\kappa_2} \triangleright \epsilon \in \delta(m) \wedge n_1 \in I\!N^* \wedge \exists n_2 \in I\!N. \kappa_2 \equiv n_2 \vee \kappa_2 \equiv \bot$$

$$Q_1(\delta) \equiv \forall m \in dom_t(\delta). \forall \imath \in dom(\delta(m)). \imath^{\kappa_1,0} \triangleright \epsilon \in \delta(m)$$

$$Q_2(k, \imath, m, S) \equiv (\imath, n_1, x, n_2) \in dom_I(S) \ \wedge \ (k, n_2, x) \in \{(\bot, 0, -1), (0, 0, 0)\} \cup \{(y, m, y) \mid y \in I\!N^*\} \wedge n_1 \in I\!N^*$$

$$Q_3(S, \delta) \equiv \forall \imath \in dom(\delta).\ live(S, \imath) \wedge \exists (\imath, n, x, n_1) \in dom_I(S). rsum(\delta, \imath) \equiv n \in I\!N^*$$

$$Q_4(S, \delta) \equiv \forall I \in dom_I(S).\ rn(I) \notin dom(\delta) \Rightarrow I \equiv (rn(I), 0, 0, 0)$$

$$Q_5(\delta, \imath, S) \equiv Q_1(q(\delta, \imath)) \wedge Q_2(0, \imath, 0, S)$$

$$Q_6(\delta, \imath) \equiv q(\delta, \imath) = \delta_1 \uplus m \mapsto \emptyset, \imath^{n_1,\kappa_2} \triangleright \jmath \ \wedge \ Q_1(\delta_1) \ \wedge \ Q_2(\kappa_2, \imath, m, S) \wedge (\kappa_2 \equiv \bot \Rightarrow \delta_1 \equiv \emptyset)$$

Type Safety Helper Judgements

Definition 25 (Store Consistency)

$$\frac{Q_0(\delta) \qquad Q_3(S, \delta) \qquad Q_4(S, \delta) \qquad \forall \imath \in dom(\delta). Q_5(\delta, \imath, S) \vee Q_6(\delta, \imath, S)}{\delta \vdash_{cns} S}$$

Definition 26 (Store Typing)

$$\frac{\delta \vdash_{cns} S \qquad R = \{rn(I) \mid I \in dom_I(S)\} \qquad dom_\ell(S) = dom(M) \qquad \forall \ell \in dom(M). R; M; \emptyset; \emptyset \ \vdash \ (dom_H(S))(\ell) : M(\ell) \ \& \ (\emptyset;\emptyset)}{R; M; \delta \vdash_{str} S}$$

Definition 27 (Threads Typing)

$$\frac{}{R; M; \Delta; \Gamma; \delta \vdash_T \emptyset} \qquad \frac{R; M; \Delta; \Gamma \ \vdash \ e : \langle\rangle \ \& \ (\delta(n);\gamma) \qquad R; M; \Delta; \Gamma; \delta \vdash_T T \qquad \text{linear}(\gamma) \equiv \emptyset}{R; M; \Delta; \Gamma; \delta \vdash_T n : e, T}$$

Definition 28 (Configuration Typing)

$$\frac{R; M; \Delta; \Gamma; \delta \vdash_T T \qquad R; M; \delta \vdash_{str} S}{R; M; \Delta; \Gamma; \delta \vdash_C S; T}$$

Definition 29 (Not stuck)

$$\frac{}{\vdash_{ns} S; \cdot} \qquad \frac{\forall\ S'; T'.\ S; T \rightsquigarrow^P S'; T'}{\vdash_{ns} S; T}$$

Proof

Lemma 1 (Progress - Program) *Let $S; T$ be a closed well-typed configuration with $R; M; \emptyset; \emptyset; \delta \vdash_C S; T$. Then $S; T$ is not stuck.*

Proof. By case analysis on $\imath : E[u]$

Case $\imath : \Box[()]$ then proof is immediate by rule *E-T*.

Case $\imath : E[(\lambda\, x.\, e \texttt{ as } \tau\ v)^{\mathsf{par}}])$: Immediate by rule *E-SN*.

Case $\imath : E[e]$ By applying inversion twice to the *configuration typing* judgement we have that $R; M; \emptyset; \emptyset \vdash E[e] : \tau\&(\delta(\imath), \gamma_B)$. The application of Lemma 3 to the latter fact and the store typing assumption ($M; R; \delta \vdash_{str} S$) yields $\exists E[e] \equiv E'[u] \wedge S_1, e_1.S; u \rightarrow^P_\imath S_1; e_1$ or $E[e] \equiv v$ or $E[e] \equiv E'[(\lambda\, x.\, e \texttt{ as } \tau\ v)^{\mathsf{par}}]$. The latter two cases cannot hold as they have been dealt with in earlier cases. Thus, the first case holds, and can be used to perform a single step by rule *E-S*.

Lemma 2 (Preservation - Program) *Let $S; T$ be a well-typed configuration with $R; M; \Delta; \Gamma; \delta \vdash_C S; T$. If the operational semantics takes a step $S; T \rightsquigarrow^P S'; T'$, then there exist $R' \supseteq R$, $M' \supseteq M$ and δ' such that the resulting configuration is well-typed with $R'; M'; \Delta; \Gamma; \delta' \vdash_C S'; T'$.*

Proof. By case analysis on the thread evaluation relation:

Case *E-T*: By inversion of configuration typing assumption we obtain the thread typing derivation. By inversion of R;M; $\Delta; \Gamma; \delta \vdash_T T_1, \imath : \Box[()], T_2$, we get that $R; M; \Delta; \Gamma; \delta \vdash_T T_1$ and $R; M; \Delta; \Gamma; \delta \vdash_T T_2$. By observation of the thread typing relation it is obvious that we require a δ such that there exists a mapping for each thread identifier. We use the latter facts in conjuction with the fact that $\imath \notin \mathit{threadid}(T_1, T_2)$ to deduce that $R; M; \Delta; \Gamma; \delta[\imath \mapsto \emptyset] \vdash_T T_1, T_2$. The output store is identical to the input store, thus it is well-typed by the store typing assumption.

Case *E-S* : By applying inversion to the *configuration typing* judgement twice we obtain that $R; M; \Delta; \Gamma \vdash E[e] : \langle\rangle, (\delta(\imath), \gamma_2)$. By applying Lemma 7 we obtain that $R; M; \Delta; \Gamma \vdash e : \tau, (\delta(\imath), \gamma_B)$. By applying Lemma 4 to the typing derivation of e, the assumption (obtained from *E-S* rule) $S; e \rightarrow^P_\imath S'; e'$, and $M; R; \delta \vdash_{str} S$ (assumption) we obtain that $\exists R \subseteq R_2, M \subseteq M_2.\ R_2; M_2; \Delta; \Gamma \vdash e' : \tau, (\gamma_C, \gamma_B)$ and $M_2; R_2; \delta[\imath \mapsto \gamma_C] \vdash_{str} S'$. By applying Lemma 15 and 14 to the typing derivations of e, $E[e]$, and the premises of the store typing assumption of the thread list T_1, T_2 in order to substitute R with R_2, we have $R_2; M_2; \Delta; \Gamma \vdash E[e] : \langle\rangle, (\delta(\imath), \gamma_2)$, $R_2; M_2; \Delta; \Gamma \vdash e : \tau, (\delta(\imath), \gamma_B)$, and $R_2; M_2; \Delta; \Gamma; \delta \vdash_T T_1, T_2$. By Lemma 8 we can replace e' for e in the evaluation context E (all well-typed in $R_2; M_2$) to obtain $R_2; M_2; \Delta; \Gamma \vdash E[e'] : \langle\rangle, (\gamma_C, \gamma_2)$. We can combine the latter fact with the typing of threads T_1, T_2 in context $R_2; M_2$, and the fact that $\mathrm{linear}(\gamma_2) \equiv \emptyset$ (obtained from the typing assumption of thread $\imath$) to derive $R_2; M_2; \Delta; \Gamma; \delta[\imath \mapsto \gamma_C] \vdash_T T_1, \imath : E[e'], T_2$. We have shown that $M_2; R_2; \delta[\imath \mapsto \gamma_C] \vdash_{str} S'$, thus the latter two facts imply that the configuration $S'; T_1, \imath : E[e'], T_2$ is well-typed.

Case *E-SN*: By performing inversion on the configuration typing judgment twice we obtain that (thread $\imath$) $E[(\lambda\, x.\, e \texttt{ as } \tau\ v)^{\mathsf{par}}]$ is well-typed. By lemma 7 we have that $(\lambda\, x.\, e \texttt{ as } \tau\ v)^{\mathsf{par}}$ is well-typed. A well-typed unit value is then constructed by using the $\vdash_D R; M; \Delta; \Gamma; \gamma_B; \gamma_B$, which is an immediate result of $\vdash_D R; M; \Delta; \Gamma; \delta(\imath); \gamma_B$ (obtained by the application of lemma 5 on the typing derivation of $(\lambda\, x.\, e \texttt{ as } \tau\ v)^{\mathsf{par}}$). By applying lemma 8 to the typing derivations of the above facts we have $E[()]$ is well-typed. We also have that $R; M; \Delta; \Gamma; \delta \vdash T_1, T_2$ from the thread typing derivation. Let $\delta' \equiv \delta[\imath \mapsto \gamma_B]$ then by $\imath \notin \mathrm{threadid}(T_1, T_2)$ and the definition of threads typing judgement we have that $R; M; \Delta; \Gamma; \delta' \vdash T_1, \imath : E[()], T_2$.

We have established that the application term is well-typed: $R; M; \Delta; \Gamma \vdash (\lambda\, x.\, e \texttt{ as } \tau\ v)^{\mathsf{par}} : ()\&(\delta(\imath), \gamma_B)$. By inverting the application term derivation twice we have that $\tau \equiv \tau_1 \xrightarrow{\gamma_1 \rightarrow \gamma_2} ()$, $R; M; \Delta; \Gamma, x : \tau_1 \vdash e : \tau_2\&(\bar{\gamma_1}; \bar{\gamma_2})$, $R; M; \Delta; \Gamma \vdash v : \tau_1\&(\delta(\imath), \delta(\imath))$, and $\gamma_B \equiv \mathrm{output}(\delta(\imath), \xi, \gamma_1, \gamma_2)$

To complete the proof for this case, it suffices that the entire thread list (i.e.,$T_1, \imath : E[()], T_2, \jmath : e[v/x]$) and the output store are well-typed. We have established that $R; M; \Delta; \Gamma; \delta' \vdash T_1, \imath : E[()], T_2$. Thus, to show that the thread list is well-typed it suffices to prove that $e[v/x]$ is well-typed and $\mathrm{linear}(\overline{\gamma_2}) \equiv \emptyset$ By applying lemma 20 to the established fact $\gamma_B \equiv \mathrm{output}(\delta(\imath), \xi, \gamma_1, \gamma_2)$ it can be deduced that $\mathrm{linear}(\overline{\gamma_2}) \equiv \emptyset$.

By the application of lemma 18 to the typing derivation of v (see above) we have that $R; M; \emptyset; \emptyset \vdash v : \tau_1\&(\emptyset, \emptyset)$. We substitute v in the body of e by using lemma 11 to obtain $R; M; \Delta; \Gamma \vdash e[v/x] : \tau_2, (\bar{\gamma_1}; \bar{\gamma_2})$. Thus, there exists a $\jmath \notin \mathrm{threadid}(T_1, T_2) \cup \{\imath\}$ such that $R; M; \Delta; \Gamma; \delta'[\jmath \mapsto \bar{\gamma_1}] \vdash T_1, \imath : E[()], T_2, \jmath : e[v/x]$.

By applying lemma 18 to the typing derivation of the function abstraction we have that the abstraction is well-typed under an empty Δ and Γ. Therefore, there exist no elements of the form ρ in the domain of $\gamma_1 \cup \gamma_2$. By inversion of the strengthened typing derivation we have that the function type is well-formed in an empty Δ. Thus, by the definition of $\bar{\gamma}$, which excludes elements of the form $\imath^{\top} \triangleright \epsilon$, we have that $\mathit{linear}(\bar{\gamma_1}) = \bar{\gamma_1}$. By applying lemma 25 to the latter fact, the store typing assumption, and $\gamma_B \equiv \mathrm{output}(\delta(\imath), \xi, \gamma_1, \gamma_2)$ we obtain that $M; R; \delta[\imath \mapsto \gamma_B, \jmath \mapsto \bar{\gamma_1}] \vdash_{str} S$ and $\jmath$ is a fresh thread identifier.

Lemma 3 (Progress - Thread-local) *Let S be a well-typed store with $R; M; \delta \vdash_{str} S$ and let e be a closed well-typed expression with $R; M; \emptyset; \emptyset \vdash e : \tau \; \& \; (\delta(n);\gamma)$. Then exactly one of the following is true:*

Case *e is a value, or*

Case *e is of the form $E[((\lambda\, x.\, e \; \texttt{as} \; \tau) \; v)^{\mathsf{par}}]$, or*

Case *e is of the form $E[e']$ and there exist S' and e'' such that $S; e' \; \rightarrow^{P}_{n} \; S'; e''$.*

Proof. By induction on the expression typing derivation. It is worth noting that by expression-redex lemma and the fact that e is well-typed there exists a redex u and an evaluation context E such that $E[u] = e$. Thus, this proves the first part of the third term of the conclusion. Therefore, we do not deal with inductive cases in the proof as all redexes are composed of values.

Case *T-I*, *T-U*, *T-F*, *T-L*, *T-R*, *T-EP*, *T-Tu*, *T-PRF*: the proof is immediate as u is a value

Case *T-Proj*, *T-App*, *T-RApp*, *T-EO*, *T-V*: the proof is a straightforward application of canonical forms lemma and the operational rules *E-P*, *E-A*, *E-RP*, *E-OR*, and *E-F* respectively. In the case of *T-App* we may have that u is equal to a parallel application term. In that case the second case of the conclusion holds.

Case *T-NRG*: The application of lemma 6 to the typing derivation of v_1, which is obtained by inverting *T-NRG*, yields that $v_1 \equiv \texttt{rgn}_k$. It suffices to show that the premise of *E-NR*, namely $(\jmath, S_1) = \text{newrgn}(k, S)$ is satisfied. This can be shown by applying lemma 27 to store typing derivation $M; R; \delta \vdash_{str} S$, which is obtained by the assumption of progress theorem, and $\text{live}(\delta(\imath), \{k\})$, which is obtained by inversion of *T-NRG*.

Case *T-NR*: The application of lemma 6 to the typing derivation of v_2, which is obtained by inverting *T-NR*, yields that $v_1 \equiv \texttt{rgn}_k$. To perform a single step, we need to prove that *E-NRF* applies. We already have that the term has the appropriate form, thus it suffices to prove the premise of *E-NRG*, which is $(\ell, S_1) = \text{alloc}(\jmath, S, v_1)$. By inversion of *T-NR* we have that $R; M; \emptyset; \emptyset \vdash v_1 : \tau\&(\delta(\imath); \delta(\imath))$, and $\text{accessible}(\delta(\imath), \{\jmath\})$. By applying lemma lemma 18 on the derivation of v_1 we obtain $R; M; \emptyset; \emptyset \vdash v_1 : \tau\&(\emptyset; \emptyset)$. By applying lemma 26 to the latter derivation as well as $\text{accessible}(\delta(\imath), \{\jmath\})$ and the store typing $M; R; \delta \vdash_{str} S$ we have that $(\ell, S_1) = \text{alloc}(\jmath, S, v_1)$.

Case *T-CP*: The application of lemma 6 to the typing derivation of v_1, which is obtained by inverting *T-NR*, yields that $v_1 \equiv \texttt{rgn}_k$. To perform a single step, we need to prove that *E-C* or *E-CB* can be applied. We have already established that the term has the appropriate form. Both operational rules share the same premise so it suffices to prove $\text{updcap}_\imath(k, S, \psi, \eta)$ is defined. If updcap returns an identical store then rule *E-CB* applies otherwise *E-C* can be used. By applying lemma 24 to $\text{modcap}(\gamma, \psi, \eta, \imath)$, which is obtained by inverting *T-CP*, and the store typing derivation, which is obtained from the progress theorem assumption, we have that $\text{updcap}_k(\imath, S, \psi, \eta)$ is defined.

Case *T-D*: The application of lemma 6 to the typing derivation of v, which is obtained by inverting *T-D*, yields $v \equiv \texttt{loc}_\ell$. Further, inversion of the typing derivation of $\texttt{loc}_\ell$ yields that $\ell \in dom(M)$ and $\vdash_D R; M; \emptyset; \emptyset; \delta(\imath); \delta(\imath)$. The latter derivation can be used to construct a typing derivation of a dummy value v of type $M(\ell)$: $R; M; \emptyset; \emptyset \vdash v : M(\ell)\&(\delta(\imath); \delta(\imath))$. By applying lemma 28 on the derivation for the dummy value v, $\text{live}(\delta(\imath), \{\jmath\})$, which is obtained by inversion of *T-D*, the fact that $\ell \in dom(M)$, and the store typing assumption, we have that $(S', v_1) \equiv \text{xupdate}_\imath(S, \ell, v)$. This implies $v_1 \equiv \text{lookup}_\imath(S, \ell)$. Therefore, rule *E-D* can be applied to perform a single step.

Case *T-A*: Similar to the proof of *T-D* (i.e., use lemma 28 to prove premise of *E-AS*).

Lemma 4 (Preservation - Thread-local) *Let e be a well-typed expression with $R; M; \Delta; \Gamma \vdash e : \tau \; \& \; (\delta(n);\gamma)$ and let S be a well-typed store with $R; M; \delta \vdash_{str} S$. If the operational semantics takes a step $S; e \rightarrow^{P}_{n} S'; e'$, then there exist $R' \supseteq R$, $M' \supseteq M$ and γ' such that the resulting expression and the resulting store are well-typed with $R'; M'; \Delta; \Gamma \vdash e' : \tau \; \& \; (\gamma';\gamma)$ and $R; M; \delta[n \mapsto \gamma'] \vdash_{str} S'$.*

Proof. By induction on the typing derivation. It is worth noting that e is a redex, which is immediate by the definition of evaluation relation. Henceforth, we use u where e should be used to stress that u is a redex.

Case *T-I*, *T-U*, *T-F*, *T-L*, *T-R*, *T-EP*, *T-Tu*, *T-PRF*: the proof is immediate as u is a value and the assumption that we perform a single operational step does not hold.

Case *T-Proj*: Immediate by observing that the tuple is well-typed, so its $\jmath^{th}$ premise will also be well-typed. Further, the resulting store is the same as the original, hence the store typing assumption gives us that it is also well-typed.

Case *T-RApp*: Immediate once the region substitution lemma (10) is applied to the premise of this typing derivation.

Case *T-V*: By inversion of *T-V*, we have that $x : \tau \in \Gamma_0, \Gamma$. Γ_0 is a list of program function names mapped to their types. The premise of *T-V* $P \vdash \Gamma_0$ in conjuction with the premise of rule *E-F*, namely $\mathtt{def}x{=}f \in P$, imply that $\emptyset; \emptyset; \emptyset; \emptyset \vdash f : \tau\&(\emptyset; \emptyset)$. By applying lemmas 14 ($\emptyset \subseteq R$), 15 ($\emptyset \subseteq M$), 16 ($\emptyset \subseteq \Delta$), 17 ($\emptyset \subseteq \Gamma$) and 13 ($R; \Delta \vdash \delta(n)$,by 5 lemma on typing derivation of *T-V*) on the typing derivation of function f we have that: $R; M; \Delta; \Gamma \vdash f : \tau\&(\delta(n); \delta(n))$. Therefore, typing is preserved. The output store S' is identical to the input store, thus it is well-typed by the store typing assumption.

Case *T-CP*: There exist two operational rules which apply for this case. If rule ***E-CB*** applies then the proof is immediate as the resulting configuration is identical to the initial configuration. If rule ***E-C*** applies then the typing preservation is immediate as this rule returns a *unit value*, which is typeable under any well-formed typing context (by lemma 5 and the definition of *T-U*). In particular, $R; M; \Delta; \Gamma \vdash () : \langle\rangle\&(\delta(n); \delta(n))$.

We have established that typing is preserved. To complete the proof it suffices to show that store typing is preserved for rule *E-C* as the output store S' differs in respect to the input store S. Inversion of the *T-CP* derivation yields $\gamma' \equiv \mathrm{modcap}(\delta(n), \psi, \eta, k)$. By applying lemma 24 to the store typing assumption and the modcap fact, we have that $S \not\equiv S' \Rightarrow M; R; \delta[n \mapsto \gamma'] \vdash_{str} S'$. We have established that $S' \not\equiv S$, thus the new store S' is well-typed: $M; R; \delta[n \mapsto \gamma'] \vdash_{str} S'$.

Case *T-NRG*: It suffices to prove that the body of new regionas well as the resulting store are well-typed :

Store: The premise of $S; newrgn\ \rho, x\ at\ rgn_\jmath\ in\ e_2 \rightarrow {}^P_\imath S_1; e_2[k/\rho][k/x]$ yields $(k, S_1) \equiv \mathrm{newrgn}(\jmath, S)$. Finally, the inversion of the typing derivation of $newrgn$ construct yields that $\mathrm{live}(\delta(n), \jmath)$ holds. The application of lemma 21 to the latter two facts and the store typing assumption yields $M; R, k; \delta[k \mapsto \delta(n), \imath^{1,\perp} \triangleright \jmath] \vdash_{str} S_1$.

Typing derivation: The typing derivation of store S_1 implies that $k \notin R$.

By inversion of the typing derivation of $newrgn$ we have that $R; M; \Delta, \rho; \Gamma, x : \mathtt{rgn}(\rho) \vdash e_2 : \tau\&(\gamma_1, \gamma_2)$, where $\gamma_1 \equiv \delta(n), \rho^{1,\perp} \triangleright \jmath$ and $\rho \notin dom(\gamma_2)$.

The application of lemma 14 to the typing derivation of e_2 and the fact that $k \notin R$ yields $R, k; M; \Delta, \rho; \Gamma, x : \mathtt{rgn}(\rho) \vdash e_2 : \tau\&(\gamma_1, \gamma_2)$. We then apply lemma 10 on the latter fact and $R, k \vdash k$ (immediate) to obtain $R, k; M; \Delta; \Gamma[k/\rho], x : \mathtt{rgn}([k/\rho]) \vdash e_2[k/\rho] : \tau[k/\rho]\&(\gamma_1[k/\rho], \gamma_2[k/\rho])$. By lemma 5 of the original typing derivation of $newrgn$ construct we have that the typing the context is not defined in terms of ρ (i.e. ρ is *fresh*). Further, the premise of $newrgn$ derivation suggests that τ is also independent of ρ (i.e. $R; \Delta \vdash \tau$). Hence, by the above facts and the definition of the substitution relation, the typing derivation of e_2 becomes $R, k; M; \Delta; \Gamma, x : \mathtt{rgn}(k) \vdash e_2[k/\rho] : \tau\&(\gamma_3, \gamma_2)$, where $\gamma_3 \equiv \delta(\imath), k^{1,\perp} \triangleright \jmath$.

The inversion of the typing derivation of $newrgn$ construct yields $R; M; \Delta; \Gamma \vdash \mathtt{rgn}_k : \mathtt{rgn}(k)\&(\delta(n); \delta(n))$. By lemma 14 and the fact that $k \notin R$ we have that $R, k; M; \Delta; \Gamma \vdash \mathtt{rgn}_k : \mathtt{rgn}(k)\&(\delta(n); \delta(n))$. The application of lemma 18 to the latter derivation yields $R, k; M; \emptyset; \emptyset \vdash \mathtt{rgn}_k : \mathtt{rgn}(k)\&(\emptyset; \emptyset)$. By applying lemma 11 to the last derivation and the fact that $R, k; M; \Delta; \Gamma, x : \mathtt{rgn}(k) \vdash e_2[k/\rho] : \tau\&(\gamma_3, \gamma_2)$ we obtain $R, k; M; \Delta; \Gamma \vdash e_2[k/\rho][\mathtt{rgn}_k/x] : \tau\&(\gamma_3, \gamma_2)$.

Case *T-D*: The store typing assumption yields that $R; M; \emptyset; \emptyset \vdash v : M(\ell)\&(\emptyset; \emptyset)$. By performing inversion twice on the typing derivation of $\mathtt{deref}\ v$ construct, we have that $M(\ell) \equiv \tau$. Hence, the type is preserved: $R; M; \emptyset; \emptyset \vdash v : \tau\&(\emptyset; \emptyset)$. Well-formedness lemma 5 of the typing derivation of $\mathtt{deref}\ v$ is $\vdash_D R; M; \Delta; \Gamma; \delta(n); \delta(n)$. We can use well-formedness to construct a new derivation for v: $R; M; \Delta; \Gamma \vdash v : \tau\&(\delta(n); \delta(n))$. The output store is identical to the input store hence it is also well-typed.

Case *T-A*: The proof for the typing preservation is similar to the proof for rule *T-D*. The store preservation proof is as follows:

The inversion of the typing derivation of the assignment construct yields $R; M; \Delta; \Gamma \vdash v : M(\ell)\&(\delta(n); \delta(n))$ and $\mathrm{accessible}(\delta(n), \jmath)$, and $\mathtt{ref}(M(\ell), \jmath)$ is the type of the location $\mathtt{loc}_\ell$ at the left hand side of the assignment construct. The premise of the operational rule *E-AS* implies that $(S', v_1) \equiv \mathrm{xupdate}_n(S, \ell, v)$. We can apply lemma 22 to the above facts and the store typing assumption to obtain that the new store S_1 is well-typed ($M; R; \delta \vdash_{str} S_1$).

Case *T-NR*: The proof for the typing preservation is similar to the proof for rule *T-D*. It suffices to prove that S' is well-typed. The inversion of the typing derivation of the memory allocation construct yields $\mathrm{live}(\delta(n), \{\jmath\})$ holds. The premise of the operational rule *E-NRF* gives $(\ell, S') \equiv \mathrm{alloc}(\jmath, S, v)$. We can apply lemma 23 to the above facts to obtain that the new store S' is well-typed ($M, \ell \mapsto (\tau, \jmath); R; \delta \vdash_{str} S'$). It should be noted that τ is the type of the initialization value v of the allocation construct.

Case *T-EO*: By performing a similar proof to *T-NRG* case, we obtain $R; M; \Delta; \Gamma \vdash e[\imath/\rho][v_1/x] : \tau\&(\gamma_3, \gamma_4)$, where $\gamma_3 \equiv \delta(n), \imath^\top \triangleright \epsilon[\imath/\rho]$, $\gamma_4 \equiv \gamma_5, \imath^\top \triangleright \epsilon[\imath/\rho]$ and γ_5 is the output effect of the entire construct, and v_1 is the value hidden in the existential package. By applying lemma 19 to the above typing derivation we have $R; M; \Delta; \Gamma \vdash e[\imath/\rho][v_1/x] : \tau\&(\delta(n), \gamma_5)$. The resulting store is identical to the input store, hence it is well-typed by the preservation assumption.

Case *T-App*: The store preservation proof is immediate as the output store is identical to the input store. The proof for the typing preservation is similar to the previous proofs. Briefly, the function application typing derivation is inverted twice, so as to obtain $R; M; \Delta; \Gamma, x : \tau_1 \vdash e : \tau_2(\overline{\gamma_1}; \overline{\gamma_2})$. By inversion of the application derivation we have that $R; M; \Delta; \Gamma \vdash v : \tau_1\&(\delta(n); \delta(n))$ and $\gamma_3 \equiv \mathrm{output}(\delta(n), \mathsf{seq}, \gamma_1, \gamma_2)$ The application lemma 18 to the typing derivation of v yields: $R; M; \emptyset; \emptyset \vdash v : \tau_1\&(\emptyset; \emptyset)$. Now lemma 11 is applied to the typing derivation of v and e to obtain: $R; M; \Delta; \Gamma \vdash e[v/x] : \tau_2\&(\overline{\gamma_1}; \overline{\gamma_2})$. Finally, lemma 12 is applied to $R; M; \Delta; \Gamma \vdash e[v/x] : \tau_2\&(\overline{\gamma_1}; \overline{\gamma_2})$ and $\gamma_3 \equiv \mathrm{output}(\delta(n), \mathsf{seq}, \gamma_1, \gamma_2)$ to obtain $R; M; \Delta; \Gamma \vdash e[v/x] : \tau_2((\delta(n); \gamma_3)$.

Theorem 1 (Type safety) *Let R_0, δ_0, S_0 and T_0 be defined as in page 9. If the initial configuration $S_0; T_0$ is well-typed with $R_0; \emptyset; \emptyset; \emptyset; \delta_0 \vdash_C S_0; T_0$ and the operational semantics takes any number of steps $S_0; T_0 \leadsto^{P^*} S_n; T_n$, then the resulting configuration $S_n; T_n$ is not stuck.*

Proof. The proof is trivial. Assume that we perform n steps, where $n \in I\!N$. We obtain that $S_n; T_n$ is well typed, in with respect to some R_n, M_n and δ_n, by applying n times lemma 2, starting from $S_0; T_0$. Then, we apply lemma 1 to prove that $S_n; T_n$ is not stuck.

Lemma 5 (Well-Formedness) $R; M; \Delta; \Gamma \vdash e : \tau \;\&\; (\gamma_1;\gamma_2) \Rightarrow \vdash_{WF} R; M; \Delta; \Gamma; \gamma_1; \gamma_2$

Proof. Straightforward proof by induction on the expression typing derivation.

Lemma 6 (Cannonical Forms) $R; M; \Delta; \Gamma \vdash v : \tau \;\&\; (\gamma_1;\gamma_2) \Rightarrow$
$\tau \equiv \langle\rangle \Rightarrow v \equiv () \;\wedge$
$\tau \equiv (\tau_1, \ldots, \tau_n) \Rightarrow v \equiv (v_1, \ldots, v_n) \;\wedge$
$\tau \equiv \mathtt{rgn}(\imath) \Rightarrow (v \equiv \mathtt{rgn}_\imath \;\wedge\; \imath \in R) \;\wedge$
$\tau \equiv \mathtt{ref}(\tau, \imath) \Rightarrow (v \equiv \mathtt{loc}_\ell \;\wedge\; \ell \mapsto (\tau, \imath) \in M) \;\wedge$
$\tau \equiv \mathtt{int} \Rightarrow v \equiv n \;\wedge$
$\tau \equiv \tau_1 \xrightarrow{\gamma_1 \to \gamma_2} \tau_2 \Rightarrow v \equiv \lambda\, x.\, e \;\mathtt{as}\; \tau_1 \xrightarrow{\gamma_1 \to \gamma_2} \tau_2 \;\wedge$
$\tau \equiv \forall \rho.\, \tau \Rightarrow v \equiv \Lambda \rho.\, f \;\wedge$
$\tau \equiv \exists \rho[\epsilon].\, \tau \Rightarrow v \equiv \mathtt{pack}\; r, v \;\mathtt{as}\; \tau$

Proof. Straightforward proof by observation of the value typing derivations.

Lemma 7 (Inversion) $R; M; \Delta; \Gamma \vdash E[e] : \tau_1 \;\&\; (\gamma_1;\gamma_2) \Rightarrow \exists \gamma_3, \tau, R; M; \Delta; \Gamma \vdash e : \tau \;\&\; (\gamma_1;\gamma_3)$

Proof. By straightforward induction on the shape of the evaluation context.

Case $\Box[e]$ then proof is immediate.

Case $(\mathtt{prj}_\jmath\; E)[e]$: The assumption implies that $R; M; \Delta; \Gamma \vdash (\mathtt{prj}_\jmath\; E)[e] : \tau_1 \;\&\; (\gamma_1;\gamma_2)$ or equivalently $R; M; \Delta; \Gamma \vdash \mathtt{prj}_\jmath\; E[e] : \tau_1 \;\&\; (\gamma_1;\gamma_2)$. By inverting the latter derivation we obtain that $R; M; \Delta; \Gamma \vdash E[e] : (\tau_1 \times, \ldots, \times \tau_n) \;\&\; (\gamma_1;\gamma_2)$. By induction hypothesis $\exists \gamma_3, \tau. R; M; \Delta; \Gamma \vdash e : \tau \;\&\; (\gamma_1;\gamma_3)$.

Case $((v_1, \ldots, E, \ldots e_n))[e]$: Similar to the above proof structure.

Case $((E\; e_2)^\xi)[e]$: Similar to the above proof structure.

Case $(((v_1)\; E)^\xi)[e]$: Similar to the above proof structure.

Case $(cap^\psi_\eta E)[e]$: Similar to the above proof structure.

Case $(E[r])[e]$: Similar to the above proof structure.

Case $(\mathtt{newrgn}\; \rho, x \;\mathtt{at}\; E \;\mathtt{in}\; e_2)[e]$: Similar to the above proof structure.

Case $(deref E)[e]$: Similar to the above proof structure.

Case $(E := e_2)[e]$: Similar to the above proof structure.

Case $(\mathtt{loc}_\ell := E)[e]$: Similar to the above proof structure.

Case $(\mathtt{new}\; E \;\mathtt{at}\; e_2)[e]$: Similar to the above proof structure.

Case $(\mathtt{new}\; v \;\mathtt{at}\; E)[e]$: Similar to the above proof structure.

Case $(\mathtt{pack}\; r, E \;\mathtt{as}\; \tau)[e]$: Similar to the above proof structure.

Lemma 8 (Replacement) $R; M; \Delta; \Gamma \vdash E[e_1] : \tau_1 \;\&\; (\gamma_1;\gamma_2) \wedge R; M; \Delta; \Gamma \vdash e_1 : \tau_2 \;\&\; (\gamma_1;\gamma_3) \wedge R; M; \Delta; \Gamma \vdash e_2 : \tau_2 \;\&\; (\gamma_4;\gamma_3) \Rightarrow R; M; \Delta; \Gamma \vdash E[e_2] : \tau_1 \;\&\; (\gamma_4;\gamma_2)$

Proof. By straightforward induction on the shape of the evaluation context. The intuition behind this proof is that the substitution of e_2 for e_1 in the evaluation context E will not surpise its environment as both e_1 and e_2 yield the same output effect. In regards to the input effect, we know that the environment will not be surprised as the expressions preceding e_1 will definately be values and can be given any effect provided that their input and output effects are the same (lemma 13).

Case $\square[e]$ then proof is immediate.

Case $(\texttt{prj}_j\ E)[e_1]$: The assumption implies that $R; M; \Delta; \Gamma \vdash (\texttt{prj}_j\ E)[e_1] : \tau_1 \ \&\ (\gamma_1;\gamma_2)$ or equivalently $R; M; \Delta; \Gamma \vdash \texttt{prj}_j\ E[e_1] : \tau_1 \ \&\ (\gamma_1;\gamma_2)$. By inverting the latter derivation we obtain that $R; M; \Delta; \Gamma \vdash E[e_1] : (\tau_1 \times, \dots, \times \tau_n) \ \&\ (\gamma_1;\gamma_2)$. The application of the induction hypothesis on the latter derivation and the assumption on the typing derivation of e_2 yields $R; M; \Delta; \Gamma \vdash E[e_2] : \tau \ \&\ (\gamma_4;\gamma_2)$. Now, *T-Proj* can be applied to obtain $R; M; \Delta; \Gamma \vdash \texttt{prj}_j\ E[e_2] : \tau \ \&\ (\gamma_4;\gamma_2)$ or equivalently $R; M; \Delta; \Gamma \vdash (\texttt{prj}_j\ E)[e_2] : \tau \ \&\ (\gamma_4;\gamma_2)$.

Case $(\texttt{new}\ v\ \texttt{at}\ E)[e]$: By inversion of the typing derivation of the memory allocation construct we have that $R; M; \Delta; \Gamma \vdash v : \tau_1 \& (\gamma_1, \gamma_1)$. The application of lemma 13 on the latter judgement and the fact that $R; \Delta \vdash \gamma_4$ (obtained by applying lemma 5 to the derivation of e_2) yields $R; M; \Delta; \Gamma \vdash v : \tau_1 \& (\gamma_4, \gamma_4)$. The inversion of the memory allocation construct also yields $\gamma_3 \vdash_{abl} r$ and $R; M; \Delta; \Gamma \vdash E[e] : \texttt{rgn}_r \& (\gamma_1, \gamma_2)$. The application of the induction hypothesis on the derivation of $E[e]$ as well as the derivation of e_2 (assumption) yields $R; M; \Delta; \Gamma \vdash E[e_2] : \tau \ \&\ (\gamma_4;\gamma_2)$. Now, *T-NR* can be applied to the latter judgment, the new derivation of v, and the accessibility judgement of r (*abl*) to obtain $R; M; \Delta; \Gamma \vdash \texttt{new}\ v\ \texttt{at}\ E[e_2] : \texttt{ref}(\tau_1, r) \ \&\ (\gamma_4;\gamma_2)$ or equivalently $R; M; \Delta; \Gamma \vdash (\texttt{new}\ v\ \texttt{at}\ E)[e_2] : \texttt{ref}(\tau_1, r) \ \&\ (\gamma_4;\gamma_2)$.

Case $((v_1, \dots, E, \dots e_n))[e]$: Similar to the above proof structure.

Case $((E\ e_2)^\xi)[e]$: Similar to the above proof structure.

Case $(((v_1)\ E)^\xi)[e]$: Similar to the above proof structure.

Case $(cap^\psi_\eta E)[e]$: Similar to the above proof structure.

Case $(E[r])[e]$: Similar to the above proof structure.

Case $(\texttt{newrgn}\ \rho, x\ \texttt{at}\ E\ \texttt{in}\ e_2)[e]$: Similar to the above proof structure.

Case $(deref E)[e]$: Similar to the above proof structure.

Case $(E := e_2)[e]$: Similar to the above proof structure.

Case $(\texttt{loc}_\ell := E)[e]$: Similar to the above proof structure.

Case $(\texttt{new}\ E\ \texttt{at}\ e_2)[e]$: Similar to the above proof structure.

Case $(\texttt{pack}\ r, E\ \texttt{as}\ \tau)[e]$: Similar to the above proof structure.

Lemma 9 (Expression-Redex) $R; M; \Delta; \Gamma \vdash e : \tau_1 \ \&\ (\gamma_1;\gamma_2) \land e \not\equiv v_1 \Rightarrow \exists\, E[u].E[u] \equiv e \land u \not\equiv v_2$

Proof. Straightforward proof by induction on the typing derivation.

Case *T-I*, *T-U*, *T-F*, *T-L*, *T-R*, *T-PRF* then the proof is immediate as e is not a value.

Case *T-V*: Immediate as it holds for $E \equiv \square$ and $u \equiv x \not\equiv v$.

Case *T-Proj*: By observing the shape of the expression of *T-Proj* typing derivation, $e \equiv \texttt{prj}_j\ e_1$. If e_1 is a value then the proof is immediate ($E \equiv \square$ and $u \equiv \texttt{prj}_j\ e_1$). Otherwise, the application of the induction hypothesis on the typing derivation of e_1 (obtained from *T-Proj* inversion) yields that $\exists\, E[u].E[u] \equiv e_1 \land u \not\equiv v_2$. Consequently, $\exists\, E.\texttt{prj}_j\ E[u] \equiv e \land u \not\equiv v_2$ or equivalently, $\exists\, E.(\texttt{prj}_j\ E)[u] \equiv e \land u \not\equiv v_2$.

Case *T-NR*: By observing the shape of the expression of *T-NR* typing derivation, $e \equiv \texttt{new}\ e_1\ \texttt{at}\ e_2$. If e_1 and e_2 are both a values then the proof is immediate ($E \equiv \square$ and $u \equiv \texttt{new}\ e_1\ \texttt{at}\ e_2$). Otherwise, if e_1 is not a value the application of the induction hypothesis on the typing derivation of e_1 (obtained from *T-NR* inversion) yields that $\exists\, E[u].E[u] \equiv e_1 \land u \not\equiv v_2$. Consequently, $\exists\, E.\texttt{new}\ E[u]\ \texttt{at}\ e_2 \equiv e \land u \not\equiv v_2$ or equivalently, $\exists\, E.(\texttt{new}\ E\ \texttt{at}\ e_2)[u] \equiv e \land u \not\equiv v_2$. The last case is that e_1 is a value and e_2 is not. By applying similar reasoning we can prove that $\exists\, E.(\texttt{new}\ e_1\ \texttt{at}\ E)[u] \equiv e \land u \not\equiv v_2$.

Case *T-App*, *T-RApp*, *T-V*,*T-EO*, *T-NRG*, *T-CP*, *T-D*, *T-A*, *T-EP*, *T-Tu*: We can perform similar reasoning to prove the remaining cases.

Lemma 10 (Region Substitution) $R; M; \Delta, \rho; \Gamma \vdash e : \tau_1 \ \&\ (\gamma_1;\gamma_2) \land R; \Delta \vdash r \Rightarrow R; M; \Delta; \Gamma[r/\rho] \vdash e[r/\rho] : \tau_1[r/\rho] \ \&\ (\gamma_1[r/\rho];\gamma_2[r/\rho])$

Proof. By induction on the typing derivation for expressions.

Case *T-I*, *T-U*, *T-L*, *T-R*: These derivations have a single premise: $\vdash_D R; M; \Delta, \rho; \Gamma; \gamma_1; \gamma_1$. We shall only discuss about the assumptions of the well-formedness derivation that will be affected by the substitution of r for ρ, namely $R; \Delta, \rho \vdash \Gamma$ and $R; \Delta, \rho \vdash \gamma_1$. The substitution of each occurence of ρ in Γ and γ_1 allow us to conclude that $R; \Delta \vdash \Gamma[r/\rho]$ and $R; \Delta \vdash \gamma_1[r/\rho]$ (this can be shown by an easy induction on the structure of the above derivations). Therefore, by combining the above derivations with the premises of well-formedness that do not get affected by the substitution we have that: $\vdash_D R; M; \Delta; \Gamma[r/\rho]; \gamma_1[r/\rho]; \gamma_1[r/\rho]$. Hence it is possible to construct a derivation of the form $R; M; \Delta; \Gamma[r/\rho] \vdash v : \tau \&(\gamma_1[r/\rho]; \gamma_1[r/\rho])$, where v is an integer, a unit value, a region handle or a run-time location. It should be noted that the type τ does not get affected by the region substitution as in the case of *T-R* the type is of the form $\mathtt{rgn}(\imath)$ and $\imath \in R$. The same applies to *T-L* as well-formedness suggests that $R \vdash M$ (i.e. M is independent of Δ). The types of *T-I* and *T-U* are constant so they also do not get affected.

Case *T-F*: By weakening the well-formedness assumption as we did in the previous case we have that: $\vdash_D R; M; \Delta; \Gamma[r/\rho]; \gamma_1[r/\rho]; \gamma_1[r/\rho]$. By applying the induction hypothesis on the derivation of the function body e we have that $R; M; \Delta; \Gamma[r/\rho], x : \tau[r/\rho] \vdash e[r/\rho] : \tau_2[r/\rho]\&(\gamma_3[r/\rho], \gamma_4; [r/\rho]\gamma_5[r/\rho]\gamma_4)[r/\rho]$. The proof that $\gamma_1 \vdash_{out} \gamma_3; \gamma_4$ implies $\gamma_1[r/\rho] \vdash_{out} \gamma_3[r/\rho]; \gamma_4[r/\rho]$ is trivial. Therefore, we can safely deduce that $R; M; \Delta; \Gamma[r/\rho] \vdash (\lambda\, x.\, e \;\mathtt{as}\; \tau)[r/\rho] : \tau[r/\rho]\&(\gamma_1[r/\rho]; \gamma_1[r/\rho]$.

Case *T-V*: By weakening the well-formedness assumption as we did in the previous case we have that: $\vdash_D R; M; \Delta; \Gamma[r/\rho]; \gamma_1[r/\rho]; \gamma_1[r/\rho]$. After substitution of ρ, the premise $(x \mapsto \tau[r/\rho] \in (\Gamma_0, \Gamma)[r/\rho]$ still holds as the domain of Γ_0, Γ is that same before and after substitution. Hence the proof we can derive that $R; M; \Delta; \Gamma[r/\rho] \vdash x : \tau[r/\rho]\&(\gamma_1[r/\rho]; \gamma_1[r/\rho])$

Case *T-App*: We apply the induction hypothesis on both subexpressions e_1 and e_2 to obtain that $R; M; \Delta; \Gamma[r/\rho] \vdash e_1 : (\tau_1 \xrightarrow{\gamma_a \to \gamma_b} \tau_2)[r/\rho]\&(\gamma_1[r/\rho]; \gamma_2[r/\rho])$ and $R; M; \Delta; \Gamma[r/\rho] \vdash e_2 : \tau_1[r/\rho]\&(\gamma_3[r/\rho]; \gamma_4[r/\rho])$. The proof that $\gamma_5 \equiv \mathrm{output}(\gamma_4, \xi, \gamma_a, \gamma_b)$ implies $\gamma_5[r/\rho] \equiv \mathrm{output}(\gamma_4[r/\rho], \xi, \gamma_a[r/\rho], \gamma_b[r/\rho])$ is trivial and has not been included here. Therefore, we may conclude that $R; M; \Delta; \Gamma[r/\rho] \vdash (e_1\ e_2)^{\xi} : \tau_2[r/\rho]\&(\gamma_1[r/\rho]; \gamma_5[r/\rho])$.

Case *T-CP*, *T-EP*, *T-Tu*, *T-PRF*, *T-Proj*, *T-RApp*, *T-NRG*, *T-NR*, *T-D*,*T-EO* *T-A*: We can perform similar reasoning to prove the remaining cases.

Lemma 11 (Variable Substitution) $R; M; \Delta; \Gamma, x : \tau_1 \vdash e : \tau_2 \;\&\; (\gamma_1;\gamma_2) \wedge R; M; \emptyset; \emptyset \vdash v : \tau_1 \;\&\; (\emptyset;\emptyset) \Rightarrow R; M; \Delta; \Gamma \vdash e[v/x] : \tau_2 \;\&\; (\gamma_1;\gamma_2)$

Proof. Straightforward induction on the expression typing derivation.

Lemma 12 (Context Weakening) $R; M; \Delta; \Gamma \vdash e : \tau_1 \;\&\; (\overline{\gamma_5};\overline{\gamma_6}) \wedge \gamma_8 \equiv \mathrm{output}(\gamma_5, \mathsf{seq}, \gamma_6, \gamma_7) \Rightarrow R; M; \Delta; \Gamma \vdash e : \tau_1 \;\&\; (\gamma_7;\gamma_8)$

Proof. This lemma can be directly derived by applying lemma 29 to the assumptions of this lemma. The application yields $\exists\gamma_9.R; \Delta \vdash \gamma_9 \wedge \gamma_9 \equiv \mathrm{output}(\gamma_5, \mathsf{seq}, \gamma_6, \gamma_7) \wedge \gamma_8 \equiv \mathrm{output}(\gamma_6, \mathsf{seq}, \gamma_6, \gamma_9) \wedge R; M; \Delta; \Gamma \vdash e : \tau_1\&(\gamma_7; \gamma_9)$.

By observing $\gamma_9 \equiv \mathrm{output}(\gamma_6, \mathsf{seq}, \gamma_6, \gamma_8)$ it is immediate that $\gamma_9 \equiv \gamma_8$. Thus, $R; M; \Delta; \Gamma \vdash e : \tau_1\&(\gamma_7; \gamma_8)$

Lemma 13 (Value-Effect) $R; M; \Delta; \Gamma \vdash v : \tau_1 \;\&\; (\gamma_1;\gamma_1) \wedge R; \Delta \vdash \gamma_3 \Rightarrow R; M; \Delta; \Gamma \vdash v : \tau_1 \;\&\; (\gamma_3;\gamma_3)$

Proof. By case analysis on value typing derivations. The proof is trivial but we provide the key steps behind the proof. By applying inversion in any of the value typing derivations we obtain the well-formedness derivation : $\vdash_D R; M; \Delta; \Gamma; \gamma_1; \gamma_1$. By observing the premises of this derivation we have $R; \Delta \vdash \gamma_1$. It is possible to replace $R; \Delta \vdash \gamma_1$ with the the assumption derivation, namely $R; \Delta \vdash \gamma_3$ to obtain $\vdash_D R; M; \Delta; \Gamma; \gamma_3; \gamma_3$. Now we may use the remaining premises of value typing, which remain intact, along with the modified well-formedness derivation to formulate the new value typing derivation: $R; M; \Delta; \Gamma \vdash v : \tau_1 \;\&\; (\gamma_3;\gamma_3)$.

Lemma 14 (Region Context Weakening) $R \subseteq R_1 \wedge R; M; \Delta; \Gamma \vdash e : \tau \;\&\; (\gamma_1;\gamma_1) \Rightarrow R_1; M; \Delta; \Gamma \vdash e : \tau \;\&\; (\gamma_1;\gamma_1)$

Proof. Straightforward induction on the expression typing derivation.

The base case is to *weaken* the well-formedness judgement of value typing derivations so that $\vdash_D R; M; \Delta; \Gamma; \gamma_1; \gamma_1$ becomes $\vdash_D R_1; M; \Delta; \Gamma; \gamma_1; \gamma_1$. The well-formdness weakening can be shown by performing a trivial induction on its premises. As in the previous lemma lemma 13, we can combine the weakened well-formedness derivation with the remaning premises of value typing derivations to obtain $R_1; M; \Delta; \Gamma \vdash v : \tau \;\&\; (\gamma_1;\gamma_1)$. In the case of *T-R* we need to show that $\imath \in R_1$ but this is immediate as $\imath \in R$ and $R \subseteq R_1$.

The inductive step is to prove that for each non-value typing derivation it is possible to perform R weakening. This can be shown by applying the induction hypothesis on its subexpression typing derivations and re-writing the derivation using the the weakened typing premises. In some derivations we also need to prove that if $R; \Delta \vdash_R r$ or $R; \Delta \vdash_T \tau$ then $R_1; \Delta \vdash_R r$ and $R_1; \Delta \vdash_T \tau$ respectively. Again, both proofs are immediate by observing their structure and using facts from the well-formedness weakening (i.e. $\forall r \in R.R_1; \Delta \vdash r$ suffices to prove both).

Lemma 15 (Memory Context Weakening) $M \subseteq M_1 \wedge R; M; \Delta; \Gamma \vdash e : \tau \,\&\, (\gamma_1;\gamma_1) \Rightarrow R; M_1; \Delta; \Gamma \vdash e : \tau \,\&\, (\gamma_1;\gamma_1)$

Proof. Similar to the previous proof.

Lemma 16 (Type Variable Context Weakening) $\Delta \subseteq \Delta_1 \wedge R; M; \Delta; \Gamma \vdash e : \tau \,\&\, (\gamma_1;\gamma_1) \Rightarrow R; M; \Delta_1; \vdash e : \tau \,\&\, (\gamma_1;\gamma_1)$

Proof. Similar to the previous proof.

Lemma 17 (Variable Context Weakening) $\Gamma \subseteq \Gamma_1 \wedge R; \Delta \vdash \Gamma_1 \wedge R; M; \Delta; \Gamma \vdash e : \tau \,\&\, (\gamma_1;\gamma_1) \Rightarrow R; M; \Delta; \Gamma_1 \vdash e : \tau \,\&\, (\gamma_1;\gamma_1)$

Proof. Similar to the previous proof. It is worth mentioning that in the base case the well-formedness weakening is immediate by the second assumption, namely $R; \Delta \vdash \Gamma_1$.

Lemma 18 (Value Strengthening) $R; M; \Delta; \Gamma \vdash v : \tau \,\&\, (\gamma_1;\gamma_1) \Rightarrow R; M; \emptyset; \emptyset \vdash v : \tau \,\&\, (\emptyset;\emptyset)$

Proof. By case analysis on value typing derivations. The proof is trivial as it suffices to prove well-formedness effect strenghening, which is immediate from the definition of well-formedness: $\vdash_D R; M; \Delta; \Gamma; \gamma_1; \gamma_1$ can be immediately be strengthened to $\vdash_D R; M; \Delta; \Gamma; \emptyset; \emptyset$. We can then combine the strengthened well-formedness derivation with the remaining premises of each value derivation, which remain intact, to obtain $R; M; \emptyset; \emptyset \vdash v : \tau \,\&\, (\emptyset;\emptyset)$.

Lemma 19 (Alias Strengthening) $R; M; \Delta; \Gamma \vdash e : \tau \,\&\, (\gamma_1, \imath^\top \rhd \epsilon, \imath;\gamma_2, \imath^\top \rhd \epsilon, \imath) \Rightarrow R; M; \Delta; \Gamma \vdash e : \tau \,\&\, (\gamma_1;\gamma_2)$

Proof. By induction on the expression typing derivations.

Case *T-I*, *T-U*, *T-F*, *T-L*, *T-R*, , *T-PRF*, *T-V*: The proof for values is immediate as we only need to strengthen the well-formedness derivation. Assume that the well-formedness premise, which is common for all values, is of the form $\vdash_D R; M; \Delta; \Gamma; \gamma_1, \imath^\top \rhd \epsilon, \imath; \gamma_1, \imath^\top \rhd \epsilon, \imath$ then by its definition it is possible to remove *effect elements* provided that the input and output effects are the same (values do not modify effects): $\vdash_D R; M; \Delta; \Gamma; \gamma_1; \gamma_1$. The strengthened well-formedness derivation along with the unaffected premises of the value derivations yield: $R; M; \Delta; \Gamma \vdash v : \tau \,\&\, (\gamma_1;\gamma_1)$.

Case *T-Proj*, *T-RApp*, *T-Tu*, *T-EP*, *T-EO*: These derivations have no effect checking or modification judgements at their premises hence it suffices to apply the induction hypothesis on the subexpression typing premises and re-write the typing derivation by using the strengthened typing premises.

Case *T-App*: As above, we can apply the induction hypothesis on the subexpression derivations, which are the premises of *T-App*, to obtain $R; M; \Delta; \Gamma \vdash e_1 : \tau_1 \xrightarrow{\gamma_a \to \gamma_b} \tau_2 \& (\gamma_1, \gamma_3)$, $R; M; \Delta; \Gamma \vdash e_2 : \tau_1 \& (\gamma_3, \gamma_4)$, and $\gamma_4; \xi \vdash \gamma_a; \gamma_b; \gamma_2$. The first two derivations hold by the induction hypothesis. The latter derivation holds even though $\imath^\top \rhd \epsilon, \imath$ has been removed from the environment input $(\gamma_4, \imath^\top \rhd \epsilon, \imath)$ and output $(\gamma_2, \imath^\top \rhd \epsilon, \imath)$ effect. This is immediate by observing that this judgement ignores elements of the form $\imath^\top \rhd \epsilon_1$.

Case *T-CP*: In this case, the proof can be performed in two steps. Firstly, the induction hypothesis is applied to the typing derivation of the sub-expression e_1. This yields $R; M; \Delta; \Gamma \vdash e_1 : \texttt{rgn}(r) \& (\gamma_1, \gamma_2)$. The second step requires to prove that $\gamma_2, \imath^\top \rhd \epsilon, \imath; \psi; \eta; r \vdash_{tcap} \gamma_3, \imath^\top \rhd \epsilon, \imath$ implies $\gamma_2; \psi; \eta; r \vdash_{tcap} \gamma_3$. This is immediate by observing that $tcap$ judgement make use of *linear* capabilities only. Therefore, we can safely ignore non-linear capabilities.

Case *T-NRG*: Similarly, we apply the induction hypothesis on the sub-expression derivations e_1 and e_2. Now, we have to prove that $\gamma_2, \imath^\top \rhd \epsilon, \imath; r_1 \vdash_{liv} \gamma_z$ implies $\gamma_2; r_1 \vdash_{liv} \gamma_z$. This can be proven by a straightforward observation of liv relation: elements of the form $\jmath^\top \rhd \epsilon'$ are ignored when encountered in γ. Therefore, it is possible to remove $\imath^\top \rhd \epsilon$ from $\gamma_2, \imath^\top \rhd \epsilon, \imath$ and still obtain the same outcome $\gamma_2; r_1 \vdash_{liv} \gamma_z$.

Case *T-NR*: Similar to the proof for *T-NRG*.

Case *T-D*, *T-A*: Similarly, we apply the induction hypothesis on the sub-expression derivations to remove $\imath^\top \rhd \epsilon, \imath$ from the input and output effects. We also have to prove that $\gamma_2, \imath^\top \rhd \epsilon, \imath \vdash_{abl} r_1$ implies $\gamma_2 \vdash_{abl} r_1$. The proof is the same as for liv (see $T - NRG$).

Lemma 20 (Spawn-Deallocate) $\mathrm{output}(\gamma_1, \mathsf{par}, \gamma_2, \gamma_3) \equiv \gamma_4 \Rightarrow \mathrm{linear}(\overline{\gamma_2}) \equiv \emptyset$

Proof. $\text{output}(\gamma_1, \mathsf{par}, \gamma_2, \gamma_3) \equiv \gamma_4$ implies $\exists\epsilon.(\gamma_4, \epsilon) \equiv \text{tout}(\mathsf{par}, \text{sumt}(\gamma_2), \text{sumt}(\gamma_3), \gamma_1)$. By the definition of $\overline{\gamma_2}$, $\overline{\gamma_2}$ consists of all effects of $\text{sumt}(\gamma_2)$ except the ones with a zero region capability. Thus, it suffices to show that all linear effects in $\text{sumt}(\gamma_2)$ have a zero region capability.

By observing the premise of tout for effects, whose exponent is of the form κ_1, κ_2 (as mentioned earlier other kinds of effects are not transformed), and given the fact that $\xi \equiv \mathsf{par}$, the capability conversion table 1 yields that the region capability κ_1' of each transformed effect will be zero.

Lemma 21 (Preservation-Add Region) $M; R; \delta \vdash_{str} S \wedge \text{live}(\delta(k), \{\jmath\}) \wedge (\imath, S_1) \equiv \text{newrgn}(\jmath, S) \wedge \imath \notin \{rn(I) \mid I \in dom_I(S)\} \Rightarrow M; R, k; \delta[k \mapsto \delta(k), \imath^{1,\perp} \triangleright \jmath] \vdash_{str} S_1$

Proof. To prove $M; R, k; \delta[k \mapsto \delta(k), \imath^{1,\perp} \triangleright \jmath] \vdash_{str} S_1$ it suffices to prove only two of is premises, namely $\delta[k \mapsto \delta(k), \imath^{1,\perp} \triangleright \jmath] \vdash_{cns} S_1$ and $R, \imath = \{rn(I) \mid I \in dom(S_1)\}$, as the other premises are related to the consistency between the store and the location typing context M and we are not adding new locations to the store S. Thefore, those premises, which hold for S also hold for S_1.

By inversion of the store typing derivation of S we have that $R = \{rn(I) \mid I \in dom(S)\}$. Thus, it is immediate that $R, \imath = \{rn(I) \mid I \in dom(S_1)\}$ as $dom_I(S_1) = dom(S), (\imath, 1-1, 0)$.

To prove $\delta[k \mapsto \delta(k), \imath^{1,\perp} \triangleright \jmath] \vdash_{cns} S_1$ then $Q_0(\delta, k \mapsto \gamma, \imath^{1,\perp} \triangleright \jmath)$, $Q_3(S_1, \delta, k \mapsto \gamma, \imath^{1,\perp} \triangleright \jmath)$, $Q_4(S_1, \delta, k \mapsto \gamma, \imath^{1,\perp} \triangleright \jmath)$,and $\forall \imath \in dom(\delta, k \mapsto \gamma, \imath^{1,\perp} \triangleright \jmath)$. $Q_5(\delta, k \mapsto \gamma, \imath^{1,\perp} \triangleright \jmath, \imath, S_1) \vee Q_6(\delta, k \mapsto \gamma, \imath^{1,\perp} \triangleright \jmath, \imath, S_1)$ must hold. From inversion of *cns* for store S we have that $Q_0(\delta)$, $Q_4(S, \delta)$, $Q_3(S, \delta)$, and $\forall \imath \in dom(\delta)$. $Q_5(\delta, \imath, S) \vee Q_6(\delta, \imath, S)$ hold.

- $Q_0(\delta, k \mapsto \gamma, \imath^{1,\perp} \triangleright \jmath)$ holds as $Q_0(\delta)$ and the exponent of $k \mapsto \gamma, \imath^{1,\perp} \triangleright \jmath$ has the appropriate form.
- $Q_3(S_1, \delta, k \mapsto \gamma, \imath^{1,\perp} \triangleright \jmath)$ holds as $Q_3(S, \delta)$ holds and $rsum(\delta, k \mapsto \gamma, \imath^{1,\perp} \triangleright \jmath, \imath) = 1$ (as $\imath \notin dom(\delta)$) and $(\imath, 1, -1, 0) \in dom_I(S_1)$.
- $Q_4(S_1, \delta, k \mapsto \gamma, \imath^{1,\perp} \triangleright \jmath)$ holds as $Q_4(S, \delta)$, $(\imath, 1, -1, 0) \in S_1$, and $\imath \in dom(\delta, k \mapsto \gamma, \imath^{1,\perp} \triangleright \jmath)$ hold.
- $\imath \notin dom(\delta)$ implies that $q(\delta, \imath) = \emptyset$ hence $Q_1(\delta)$ holds.
- $Q_2(\perp, \imath, k)$ holds as $(\imath, 1, -1, 0) \in dom_I(S)$ and $(\perp, 0, -1)$ belongs to the set of valid configurations (see defn of Q_2).

Therefore, $\delta, k \mapsto \gamma, \imath^{1,\perp} \triangleright \jmath \vdash_{cns} S_1$ holds.

Lemma 22 (Preservation-Update Value) $M; R; \delta \vdash_{str} S \wedge \text{accessible}(\delta(\jmath), \{\imath\}) \wedge (S, v') \equiv \text{xupdate}_\jmath(S, \ell, v) \wedge R; M; \Delta; \Gamma \vdash v : M(\ell)\ \&\ (\gamma_1;\gamma_1) \Rightarrow M; R; \delta \vdash_{str} S_1$

Proof. By observation of the third assumption we deduce that $dom_I(S) = dom_I(S_1)$. Therefore, *cns* that holds for S also holds for S_1. Further, the above observation also implies that $R = \{rn(I) \mid I \in dom(S_1)\}$ holds as it does hold for S.

The third assumption also tells us that $\ell \mapsto v_1 \in dom_\ell(S)$. The first assumption implies that $R; M; \emptyset; \emptyset \vdash v_1 : M(\ell)\&(\emptyset;\emptyset)$. It suffices to show $R; M; \emptyset; \emptyset \vdash v : M(\ell)\&(\emptyset;\emptyset)$ as all other heap locations and their corresponding values of S_1 are identical to the ones of S. This is immediate by applying lemma 18 to the fourth assumption.

Lemma 23 (Preservation-Add Location) $M; R; \delta \vdash_{str} S \wedge R; M; \Delta; \Gamma \vdash v : \tau\ \&\ (\gamma_1;\gamma_1) \wedge \text{live}(\delta(k), \{\imath\}) \wedge (\ell, S_1) \equiv \text{alloc}(\imath, S, v) \Rightarrow M, \ell \mapsto \tau; R; \delta \vdash_{str} S_1$

Proof. By observation of the third assumption we deduce that $dom_I(S) = dom_I(S_1)$. Therefore, *cns* that holds for S also holds for S_1. Further, the above observation also implies that $R = \{rn(I) \mid I \in dom(S_1)\}$ holds as it does hold for S.

The fifth assumption also tells us that $\ell \notin \{\ell_1 \mid \ell_1 \mapsto v_1 \in dom_\ell(S)\}$. It suffices to show $R; M, \ell \mapsto \tau; \emptyset; \emptyset \vdash v : \tau\&(\emptyset;\emptyset)$ as for all other heap locations and their corresponding values, which are identical to the ones of S, we have from *str* assumption that $\forall \ell_2 \in \{\ell_1 \mid \ell_1 \mapsto v_1 \in dom_\ell(S)\}.R; M; \emptyset; \emptyset \vdash v_2 : M(\ell_2)\&(\emptyset;\emptyset)$. Thus by applying lemma 15 to each one of them we have that: $\forall \ell_2 \in \{\ell_1 \mid \ell_1 \mapsto v_1 \in dom_\ell(S)\}.R; M, \ell \mapsto \tau; \emptyset; \emptyset \vdash v_2 : M(\ell_2)\&(\emptyset;\emptyset)$. As mentioned earlier, these locations are identical for S_1 hence, $\forall \ell_2 \in (\{\ell_1 \mid \ell_1 \mapsto v_1 \in dom_\ell(S_1)\} \setminus \{\ell\}).R; M, \ell \mapsto \tau; \emptyset; \emptyset \vdash v_2 : M(\ell_2)\&(\emptyset;\emptyset)$. The proof is completed by applying lemma 15 and lemma 18 to the second assumption.

Lemma 24 (Preservation/Progress-Capability Modification) $M; R; \delta \vdash_{str} S \wedge \gamma' \equiv \text{modcap}(\gamma, \psi, \eta, \imath) \Rightarrow \text{updcap}_k(\imath, S, \psi, \eta) \equiv S_1 \wedge (S_1 \not\equiv S \Rightarrow M; R; \delta[k \mapsto \gamma'] \vdash_{str} S_1)$

Proof. Prood by case analysis on ψ.

Case $\psi \equiv R$: Assume that $\delta(k) = \gamma_1, \imath^{\kappa_1,\kappa_2} \triangleright \epsilon$, $\gamma = \gamma_1, \imath^{\kappa_3,\kappa_2} \triangleright \epsilon$. Then by inversion of the second assumption we have that:

- $\kappa_3 \equiv \mathrm{mod}_\eta(\kappa_2)$
- $\kappa_3 > 0 \vee \kappa_3 > \bar{0}$.

The store typing (first) assumption implies $\delta \vdash_{cns} S$, which in turn implies $Q_0(\delta)$. Q_0 states that κ_3 is of the form n_1 (i.e., not of the form $\bar{n_1}$). Hence it holds that $\kappa_3 > 0$.

case $(\eta \equiv -1 \wedge \kappa_2 > 1) \vee \eta \equiv 1$: By inversion of cns we also have $Q_3(S, \delta)$, which states the fact that $\imath$ belongs to $dom(\delta)$ implies $live(S, \{\imath\})$ and there exists some $I \in dom_I(S)$ such that $rn(I) = \imath$ and $rc(I) = n \in I\!N^*$. From the last two facts we can derive $\mathrm{updcap}_k(\imath, S, \psi, \eta) \equiv S_1$. $S_1 \not\equiv S$ hence the last step is to prove $M; R; \delta[k \mapsto \gamma] \vdash_{str} S_1$. It suffices to prove $\delta[k \mapsto \gamma] \vdash_{cns} S_1$ as the remaining premises of str hold for S_1. This is a direct implication of the fact that the remaining premises hold for S and the heap locations and their corresponding values and region names of S are identical to the ones of S_1. The following hold by inversion of $\delta \vdash_{cns} S$:

$Q_0(\delta), Q_4(S, \delta), \forall \jmath \in dom(\delta).\ Q_5(\delta, \jmath, S) \vee Q_6(\delta, \jmath, S)$

$Q_3(S, \delta)$. The first group of invariants, namely Q_0, Q_4, Q_5, Q_6 also hold for S_1 and $\delta[k \mapsto \gamma]$. This is because $\delta[k \mapsto \gamma]$ and δ differ only in κ_3. Q_0 holds as $\kappa_3 \in I\!N^*$. Q_4 holds as $dom(\delta) = dom(\delta[k \mapsto \gamma])$. Q_5 and Q_6 are related to lock capabilities of each region in $dom(\delta[k \mapsto \gamma])$, which are identical to the ones of δ. Similarly, the run-time lock counts of region headers of S are identical to the ones of S_1. Therefore, Q_5, Q_6 hold for S_1 and $\delta[k \mapsto \gamma]$. It suffices to prove that $Q_3(S_1, \delta[k \mapsto \gamma])$ to complete the proof. Assume that $\delta' = \delta \setminus q(\delta, \imath)$ and $\delta'' = (\delta[k \mapsto \gamma]) \setminus q((\delta[k \mapsto \gamma]), \imath)$ then $\delta'' = \delta$ as other region effects than $\imath$ are identical. Therefore, we can deduce that $Q_3(\delta', S_1)$ by using the fact that $Q_3(\delta', S)$ (directly derived from $Q_3(\delta, S)$) as well as the fact that regions of S_1 othen than $\imath$ are identical as the ones of S. Therefore, we only need to prove $Q_3(S_1, q((\delta[k \mapsto \gamma]), \imath))$.

- case $\eta \equiv 1$: Assume that $I \in dom_I(S)$ and $rn(I) = \imath$ then from $Q_3(S, q(\delta, \imath))$ we have that $rsum(q(\delta, \imath)) = rc(I) \in I\!N^*$ or $rsum(\delta[\kappa \mapsto \emptyset], \imath) = rc(I) - \kappa_2$. Therefore, $rsum(\delta[\kappa \mapsto \gamma], \imath) = rsum(\delta[\kappa \mapsto \emptyset], \imath) + \kappa_3$ or $rsum(\delta[\kappa \mapsto \gamma], \imath) = rc(I) - \kappa_2 + \kappa_3$. By the fact that $\kappa_3 \equiv mod_1(\kappa_2)$ or $\kappa_3 = \kappa_2 + 1$ we can deduce that $rsum(\delta[\kappa \mapsto \gamma], \imath) = rc(I) + 1$. This holds as the new region header I' of store S_1 has the following properties $rn(I') = \imath$ and $rc(I') = rc(I) + 1$. Therefore, $Q_3(S_1, q(\delta[k \mapsto \gamma], \imath)$

- case $\eta \equiv -1$ and $\kappa_2 > 1$: Similar proof to the proof of the previous case.

- case $\eta \equiv -1 \wedge \kappa_2 \equiv 1$: As in the previous case we have that $live(S, \{\imath\})$ by inversion of cns and the fact that $\imath \in dom(\delta)$. If subregions of $\imath$ have a non-zero region count then cap will return an S_1 such that $S_1 \equiv S$. In this case the proof is immediate as $S_1 \not\equiv S$ does not hold. If $S_1 \not\equiv S$ then it suffices to prove $\delta[k \mapsto \gamma] \vdash_{cns} S_1$ in order to prove str, as the region names of region names as well as the locations and their corresponding values are identical to the ones of S. Therefore, the premises of str for S also hold for S_1 (except for cns that will be proved shortly). We have that $\imath \notin dom(\gamma)$ by observing the definition of $tcap$ and using the fact that $\kappa_3 \equiv 0$. Further, $\imath \notin dom(\delta \setminus \{k \mapsto \gamma, \imath^{\kappa_1,\kappa_2} \triangleright \epsilon\})$ as we have a single run-time reference count of $\imath$ and only one thread may have $\imath$ in its context, namely thread k. By combining those two facts we have that $\delta[k \mapsto \gamma] = \delta \setminus \{k \mapsto \gamma, \imath^{\kappa_1,\kappa_2} \triangleright \epsilon\}$. Let $\delta' = \delta \setminus \{k \mapsto \gamma, \imath^{\kappa_1,\kappa_2} \triangleright \epsilon\}$ then the following hold by inversion of cns for S:

- $Q_0(\delta)$ holds, hence $Q_0(\delta')$ also holds as $\delta' \subseteq \delta$.

- $Q_3(S, \delta)$, $\forall \jmath \in dom(\delta).\ Q_5(\delta, \jmath, S) \vee Q_6(\delta, \jmath, S)$ hold, hence $Q_3(S_1, \delta')$, $\forall \jmath \in dom(\delta).\ Q_5(\delta', \jmath, S_1) \vee Q_6(\delta', \jmath, S_1)$ also hold as $\delta' \subseteq \delta$, $\forall I \in dom_I(S_1).\ rn(I) \not\equiv \imath \Rightarrow I \in dom_I(S)$, and $\forall \jmath \in dom(\delta').live(S_1, \{\jmath\})$ hold. The last property holds as there exists no child region of $\imath$ in δ'. This can be deduced by observing rule cap: All child regions must have a region count of zero before decrementing $\imath$. $Q_0(\delta, S)$ tells us that region counts of regions that belong in δ must be non-zero. δ' is a subset of δ so it also does not contain child regions of $\imath$. As for remaning regions, other than $\imath$ and its child regions, $live(S_1, \{\jmath\})$ holds by $Q_3(S, \delta)$ and the fact that $\delta' \subseteq \delta$ and $\forall I \in dom_I(S_1)$. $rn(I) \not\equiv \imath \Rightarrow I \in dom_I(S)$.

- From $tcap$ assumption we have $\kappa_2 = 0$, otherwise $tcap$ judgement would not hold. As mentioned earlier, $\imath$ belongs only to thread k, and $Q_3(S, \delta)$ gives us that the sum of lock counts of all threads must be equal to the run-time lock count. We know that we have a single thread k that owns $\imath$ and $k \mapsto \gamma, \imath^{1,0} \triangleright \epsilon \in \delta$ as $\kappa_2 = 0$. Therefore, if $I \in dom_I(S)$ and $I' \in dom_I(S_1)$ and $rn(I) = rn(I') = \imath$, then $lc(I) = 0$. I' differs in respect to I in that its region count is zero therefore: $I' = (\imath, 0, 0, 0)$ (the last element of this tuple is zero as when a lock reaches the value zero, the thread owner identifier becomes zero as well). Now, we have that $Q_4(S, \delta)$, $\delta = \delta', k \mapsto \gamma, \imath^{1,0} \triangleright \epsilon$, $\forall I \in dom_I(S_1)$. $rn(I) \not\equiv \imath \Rightarrow I \in dom_I(S)$, and $I' = (\imath, 0, 0, 0)$, thus $Q_4(S_1, \delta')$ holds.

Case $\psi \equiv L$: By inversion of the str assumption we have that $\forall \jmath \in dom(\delta).\ Q_5(\delta, \jmath, S) \vee Q_6(\delta, \jmath, S)$ holds, hence $Q_5(\delta, \imath, S) \vee Q_6(\delta, \imath, S)$ also holds ($tcap$ assumption implies that $\imath \in dom(\delta)$). We proceed by performing a case analysis:

- Case $Q_5(\delta, \imath, S)$ holds then the lock count of $\imath$ is zero as well as its the thread owner. More formally, $\exists I \in dom_I(S).rn(I) \equiv \imath \wedge lc(I) \equiv 0 \wedge ln(I) \equiv 0$. Q_5 also gives us that each element of $q(\delta, \imath)$ is of the form $\imath^{\kappa_1,0} \triangleright \epsilon$. Assumption $tcap$ can only hold if $\eta = 1$. Therefore, $ar(I, 1, k)$ is definable. To complete our proof that cap holds, we need to $live(S, \{\imath\})$. This can be obtained from $Q_3(S, \delta)$. Thus, $\mathrm{updcap}_k(\imath, S, L, 1) \equiv S_1$ holds. Function ar modifies I, hence $S \not\equiv S$. Therefore, we need to prove that $M; R; \delta[k \mapsto \gamma] \vdash_{str} S_1$. As mentioned above, S_1 differs from S in respect to the region header I. Hence, all premises of str assumption (except for cns) that hold for S also hold for S_1. Let $\delta' = \delta[k \mapsto \gamma]$, then it is obvious that it suffices to prove $\delta' \vdash_{cns} S_1$ in order to complete our proof. By observation of $tcap$ it is obvious that only effect $\imath$ changes. Hence if $\delta(k) = \gamma_1, \imath^{\kappa_1,0} \triangleright \epsilon$ then $\gamma = \gamma_1, \imath^{\kappa_1,1} \triangleright \epsilon$.

- $Q_0(\delta)$ holds, hence $Q_0(\delta')$ also holds as $\delta' \subseteq \delta$.

- $Q_3(S,\delta)$, $Q_4(S,\delta)$ hold by *cns* inversion and $Q_3(S_1,\delta')$, $Q_4(S_1,\delta')$ also hold as S_1 differs from S at the lock count of header I. The region counts remain intact so Q_3 and Q_4 hold for S_1 and δ'.

- $\forall \jmath \in dom(\delta).\ Q_5(\delta,\jmath,S) \vee Q_6(\delta,\jmath,S)$ hold by *cns* inversion, hence $\forall \jmath \in dom(\delta' \setminus q(\delta',\imath)).\ Q_5(\delta',\jmath,S_1) \vee Q_6(\delta',\jmath,S_1)$ as S_1 differs from S at the lock count of header I. To complete the proof it suffices to show that $Q_6(q(\delta',\imath),\imath,S_1$ holds. We mentioned earlier, that for each thread the lock capability of region $\imath$, κ_2 is zero. Now we have a single lock capability for thread k. Thus, the run-time lock count of $\imath$ in S_1 equals the single thread capability of thread k. All other lock capabilities remain zero. Therefore, $Q_6(q(\delta',\imath),\imath,S_1)$ holds.

- Case $Q_6(\delta,\imath,S)$ holds then only one thread has a non-zero lock count. If that thread is not k then $\kappa_2 = 0$ and $\eta = 1$ by observation of the $tcap$ assumption. Therefore, $ar(I,1,k)$ is definable and will return I intact as another thread owns region $\imath$. To complete our proof that *cap* holds, we need to $live(S,\{\imath\})$. This can be obtained from $Q_3(S,\delta)$ premise of $\delta \vdash_{cns} S$. We have that $ar(I,1,k)$ is definable as well as $live(S,\{\imath\})$. Therefore, $\text{updcap}_k(\imath,S,L,1) \equiv S$. S is equal to S_1 hence the proof is complete for the case thread k does not hold the lock. If k holds the lock then assuming that $\delta' = \delta[k \mapsto \gamma]$, we only need to prove that $Q_6(\delta',\imath,S_1)$ holds. The remaining premises of $\delta' \vdash_{cns} S_1$ hold as $\delta \vdash_{cns} S$ holds and S_1 differs from S in respect to the lock count of region $\imath$. This can be shown by following similar reasoning to the previous proof.

Lemma 25 (Preservation-Spawn-Capability Modification) $M;R;\delta \vdash_{str} S \wedge \delta(\imath);\mathsf{par} \vdash \gamma_1;\gamma_2 \Rightarrow \gamma_3 \wedge \bar{\gamma_1} = \text{linear}(\gamma_1) \Rightarrow M;R;\delta[\jmath \mapsto \bar{\gamma_1}, \imath \mapsto \gamma_3] \vdash_{str} S$

Proof. To prove $M;R;\delta[\imath \mapsto \gamma_3, \jmath \mapsto \gamma_4] \vdash_{str} S$, it suffices to show that $\delta[\imath \mapsto \gamma_3, \jmath \mapsto \gamma_4] \vdash_{cns} S$ as all premises of *str* except for *cns*, do not take into account δ but only $R;M$. Further, we know that these premises hold by inversion of the first assumption so we can use them to prove $M;R;\delta[\imath \mapsto \gamma_3, \jmath \mapsto \gamma_4] \vdash_{str} S$.

Let $\delta' = \delta[\imath \mapsto \gamma_3, \jmath \mapsto \gamma_4]$, $\delta_1 = \bigcup_{p \in dom(\gamma_4[\jmath \mapsto \emptyset])} q(\delta',p)$, and $\delta_2 = \bigcup_{p \notin dom(\gamma_4[\jmath \mapsto \emptyset]))} q(\delta',p)$.

By inversion of $M;R;\delta \vdash_{str} S$ we have that $\delta \vdash_{cns} S$. By applying inversion on *cns* we have that:

– $Q_0(\delta) = Q_0(\delta[\jmath \mapsto \delta(\jmath)]) = Q_0(\delta[\jmath \mapsto \emptyset]) \uplus Q_0(\jmath \mapsto \delta(\jmath))$.

– $Q_3(S,\delta) = Q_3(S,\delta_1) \uplus Q_3(S,\delta_2)$.

– $Q_4(S,\delta)$

– $\forall k \in dom(\delta).Q_5(\delta,k,S) \vee Q_6(\delta,k,S) \equiv \forall k \in dom(\delta_1).Q_5(\delta_1,k,S) \vee Q_6(\delta_1,k,S) \wedge \forall k \in dom(\delta_2).Q_5(\delta_2,k,S) \vee Q_6(\delta_2,k,S)$

To complete our proof we need to show that the above premises hold for δ'.

– $Q_4(S,\delta')$ holds as Q_4 only examines the domain of δ' and the second and third assumption imply that $dom(\delta') = dom(\delta)$.

– $Q_0(\delta(\jmath))$ by observing the derivation tree of the first assumption: To type-check a function input effect its region capability (see function F_3) must be n or $\bar{n}$, while the environment input effect must also have a region capability of the form n_1 or $\bar{n_1}$ respectively. By $Q_0(\delta)$ we can deduce that the environment input effect is of the form n_1. Therefore, the function input effects are also of the form n, and so is the summarized function input effects $\delta(\jmath)$. Hence, $Q_0(\delta(\jmath))$ holds. We can combine this fact with an earlier fact, in particular $Q_0(\lceil\delta\rceil_\jmath)$ to prove that $Q_0(\delta')$ holds.

– As mentioned in the previous case, for each region k that belongs in $dom(\gamma_4)$ has a region capability of the form n, then k belongs in $dom(\delta(\imath))$, and if k belongs to $dom(\gamma_3)$ then its region capability is $n_1 - n$. Therefore, if k belongs to $dom(\gamma_3)$ the overall region capability sum of γ_3, γ_4 for region k, $(n_1 - n) + n = n_1$. If k does not belong to $dom(\gamma_3)$ then the overall region capabilty sum is n, but according to F_3 in that case $n_1 = n$. Thus, in both cases the overall region capability sum is n_1. We have from $Q_3(S,\delta_1)$ that $I \in dom_I(S) \wedge rn(I) = k \Rightarrow rc(I) = n_1$, therefore $Q_3(S,\delta_1)$ holds. $Q_3(S,\delta_2)$ also holds as it represents the all other regions than k that were not affected (so it holds from $Q_3(S,\delta \setminus q(\delta,k))$, which can be derived from *cns* of the first assumption). By combinding $Q_3(S,\delta_2)$ and $Q_3(S,\delta_1)$ we have that $Q_3(S,\delta')$.

– The proof for $\forall k \in dom(\delta').Q_5(\delta',k,S) \vee Q_6(\delta',k,S)$ is similar to the earlier case . Briefly, if the lock capability of some region k of γ_3 is $\perp$ then k will not exist in the output effect list γ_4. If the lock capability of k in $\delta(\jmath)$ is zero then the proof for region k is immediate. Finally, if the lock capability is non-zero then the corresponding lock capability of γ_3 will have the same value whereas the lock capability in γ_4 will be zero. It is obvious that in all cases the original lock capability (i.e., $\forall k \in dom(\delta).Q_5(\delta,k,S) \vee Q_6(\delta,k,S)$) is preserved.

Lemma 26 (Progress-Add Location) $M;R;\delta \vdash_{str} S \wedge R;M;\emptyset;\emptyset \vdash v : \tau \,\&\, (\emptyset;\emptyset) \wedge \text{live}(\delta(k),\{\imath\}) \Rightarrow \exists S_1,\ell.(\ell,S_1) = \text{alloc}(\jmath,S,v)\ell \notin dom(M)$

Proof. By observation of if_ℓ judgement it suffices to show that:

- $live(S, \{k\})$: This is immediate by inversion of the first assumption, which gives us that $\delta \vdash_{cns} S$. By the third assumption we can deduce that $k \in dom(\delta)$. By combining those two facts we can deduce from cns premise Q_3 that $live(S, \{k\})$.
- Assume that $I : (H, S_2)$ is a subregion of S and $rn(I) = k$ then $\ell \mapsto v$ can be added to H. This is immediate as ℓ is a fresh location. This fact implies in conjuction with the first premise that ℓ does not exist in the locations of the entire store. Therefore, it can be added to H.

Lemma 27 (Progress-Add Region) $M; R; \delta \vdash_{str} S \wedge \text{live}(\delta(\imath), \{k\}) \Rightarrow \exists S_1, \jmath.(\jmath, S_1) \equiv \text{newrgn}(k, S) \wedge \jmath \notin R$

Proof. By observation of ins_S judgement it suffices to show that:

- $live(S, \{k\})$: This is immediate by inversion of the first assumption, which gives us that $\delta \vdash_{cns} S$. By the second assumption we can deduce that $k \in dom(\delta)$. By combining those two facts we can deduce from cns premise Q_3 that $live(S, \{k\})$.
- $\jmath$ is a fresh region (third assumption). By inversion of the second assumption it is obvious that it does not exist in the region names of S, thus it can be inserted.

Lemma 28 (Progress-Update Value) $M; R; \delta \vdash_{str} S \wedge \text{accessible}(\delta(\jmath), \{\imath\}) \wedge R; M; \Delta; \Gamma \vdash v : \tau \;\&\; (\gamma_1;\gamma_1) \wedge \ell \in dom(M) \Rightarrow \exists S', v_1.(S', v_1) \equiv \text{xupdate}_\jmath(S, \ell, v)$

Proof. By observation of the second assumption we have that $\jmath \in dom(\delta)$. We can invert the first assumption to obtain $\delta \vdash_{cns} S$. By inversion of cns and the fact that $\jmath \in dom(\delta)$ we have $Q_5(\delta, \jmath, S) \vee Q_6(\delta, \jmath, S)$. We now perform a case analysis on Q_5 and Q_6:

- Assume $Q_5(\delta, \jmath, S)$ holds. This is a contradiction as the second assumption implies that thread $\jmath$ has a lock capability for region $\imath$ but Q_5 implies that for all threads the region capability of $\imath$ is zero. Hence Q_5 cannot hold.
- It is obvious from the above that $Q_6((\delta, \jmath, S)$ holds. Informally, Q_6 implies that there exists exactly one thread whose lock capability of region $\imath$ is non-zero. The second assumption tells us that this thread is $\jmath$. Let κ_2 be the lock capability of region $\imath$ of thread $\jmath$. We can deduce from Q_6 that $Q_2(\kappa_2, \imath, \jmath, S)$ holds. Lock capability κ_2 can have two possible values:

  If $Q_2(\bot, \imath, \jmath, S)$ $\kappa_2 = \bot$ holds then by the definition of Q_2 $\exists I \in dom_I(S).rn(I) = \imath \wedge lc(I) = -1 \wedge ln(I) = 0$.

  If $Q_2(n, \imath, \jmath, S)$ $\kappa_2 = \bot$ holds then by the definition of Q_2 $\exists I \in dom_I(S).rn(I) = \imath \wedge lc(I) = n \wedge ln(I) = \jmath$.
- By the first and fourth assumption we have that ℓ belongs to the locations of S

Therefore, all premises of upd_ℓ are satisfied, thus $\exists S', v_1.(S', v_1) \equiv \text{xupdate}_\jmath(S, \ell, v)$

Lemma 29 (Context Weakening Generalization) $R; M; \Delta; \Gamma \vdash e : \tau \& (\overline{\gamma_1}; \overline{\gamma_3}) \wedge R; \Delta \vdash \gamma_4 \wedge R; \Delta \vdash \gamma_5 \wedge \text{output}(\gamma_4, \mathsf{seq}, \gamma_1, \gamma_2) \equiv \gamma_5 \Rightarrow \exists \gamma_6.R; \Delta \vdash \gamma_6 \wedge \text{output}(\gamma_4, \mathsf{seq}, \gamma_1, \gamma_3) \equiv \gamma_6 \wedge \text{output}(\gamma_6, \mathsf{seq}, \gamma_3, \gamma_2) \equiv \gamma_5 \wedge R; M; \Delta; \Gamma \vdash e : \tau \& (\gamma_4; \gamma_6)$

Proof. By induction on the expression typing derivation.

Case *T-I*, *T-U*, *T-F*, *T-L*, *T-R*, *T-V*,*T-PRF*: In this case the typing derivation is of the form $R; M; \Delta; \Gamma \vdash v : \tau \& (\overline{\gamma_1}; \overline{\gamma_1})$. The application of lemma 5 to this derivation yields $\vdash_D R; M; \Delta; \Gamma; \overline{\gamma_1}; \overline{\gamma_1}$. The assumptions $R; \Delta \vdash \gamma_4$ $R; \Delta \vdash \gamma_5$ and the well-formedness fact imply that $\vdash_D R; M; \Delta; \Gamma; \gamma_5; \gamma_5$. We can then re-construct the value typing as follows: $R; M; \Delta; \Gamma \vdash v : \tau \& (\gamma_5; \gamma_5)$.

Case *T-Proj*, *T-RApp*, *T-EP*: These typing derivations have a single typing sub-derivation hence the proof is immediate by applying the induction hypothesis on the sub-derivations and re-constructing the typing derivation using the induction hypothesis conclusions. The proof is trivial for one additional reason: there exist no capability checking or modification predicates in their premises.

Case *T-App*: We can apply the induction hypothesis on the leftmost premise $R; M; \Delta; \Gamma \vdash e_1 : \tau_a \overset{\gamma_a \to \gamma_b}{\longrightarrow} \tau_b \;\&\; (\overline{\gamma_1};\overline{\gamma_{1a}})$ (along with the remaining assumptions of this lemma) to obtain that $\exists \gamma_6.R; \Delta \vdash \gamma_6 \wedge \text{output}(\gamma_4, \mathsf{seq}, \gamma_1, \gamma_{1a}) \equiv \gamma_6 \wedge \text{output}(\gamma_6, \mathsf{seq}, \gamma_{1a}, \gamma_2) \equiv \gamma_5 \wedge R; M; \Delta; \Gamma \vdash e_1 : \tau_a \overset{\gamma_a \to \gamma_b}{\longrightarrow} \tau_b \;\&\; (\gamma_4;\gamma_6)$.

By performing the induction hypothesis on $R; M; \Delta; \Gamma \vdash e_2 : \tau_a \;\&\; (\overline{\gamma_{1a}};\overline{\gamma_{1b}})$ and and using that $\text{output}(\gamma_6, \mathsf{seq}, \gamma_{1a}, \gamma_2) \equiv \gamma_5$, $R; \Delta \vdash \gamma_6$, and $R; \Delta \vdash \gamma_5$ (from the previous application) we have that: $\exists \gamma_7.R; \Delta \vdash \gamma_7 \wedge \text{output}(\gamma_6, \mathsf{seq}, \gamma_{1a}, \gamma_{1b}) \equiv \gamma_7 \wedge \text{output}(\gamma_7, \mathsf{seq}, \gamma_{1b}, \gamma_3) \equiv \gamma_5 \wedge R; M; \Delta; \Gamma \vdash e_2 : \tau_a \;\&\; (\gamma_6;\gamma_7)$

By applying lemma 30 on $\text{output}(\gamma_7, \mathsf{seq}, \gamma_{1b}, \gamma_3) \equiv \gamma_5$ and the $\overline{\gamma_3} = \text{output}(\overline{\gamma_{1b}}, \xi, \gamma_a, \gamma_b)$ premise the application typing derivation we have that $\gamma_5 = \text{output}(\gamma_7, \xi, \gamma_a, \gamma_b)$. Therefore, $R; M; \Delta; \Gamma \vdash (e_1\ e_2)^\xi : \tau_b \;\&\; (\gamma_4;\gamma_5)$

Case *T-EO*, *T-Tu*: Similar to the *T-App* case.

Case *T-NRG*: Similar to the *T-App* case. In contrast with *T-App*, we have an additional obligatation. We need to prove that given $\mathrm{live}(\overline{\gamma_{1b}}, \{r\})$, $\mathrm{live}(\gamma_7, \{r\})$ holds. This is immediate by applying lemma 31.

Case *T-CP*: Similar to the *T-Proj* case. In contrast with *T-Proj*, we have an additional obligatation. We need to prove that given $\mathrm{live}(\overline{\gamma_{1b}}, \{r\})$, $\mathrm{live}(\gamma_7, \{r\})$ holds. This is immediate by applying lemma 33.

Case *T-D*,*T-NR*, *T-A*: The first case is similar to *T-Proj* case, whereas the remaining two cases are similar to *T-App*. In all three cases, we have an additional obligatation. We need to prove that given $\mathrm{accessible}(\overline{\gamma_{1b}}, \{r\})$, $\mathrm{accessible}(\gamma_7, \{r\})$ holds. This is immediate by applying lemma 32.

Lemma 30 ("Implies"-Weakening) $\mathrm{output}(\overline{\gamma_1}, \xi, \gamma_3, \gamma_4) \equiv \overline{\gamma_2} \wedge \mathrm{output}(\gamma_5, \mathsf{seq}, \gamma_1, \gamma_2) \equiv \gamma_6 \Rightarrow \mathrm{output}(\gamma_5, \xi, \gamma_3, \gamma_4) \equiv \gamma_6$

Proof. To prove the conclusion we need to prove its premises. Therefore, we need to prove $\mathrm{inv}(\gamma_6)$, $\mathrm{live}(\gamma_5, \epsilon_3)$ and $\mathrm{live}(\gamma_6, \epsilon_3)$. The first obligation comes for free as it is a premise of $\exists \epsilon_2.(\gamma_6, \epsilon_2) \equiv \mathrm{tout}(\mathsf{seq}, \mathrm{sumt}(\gamma_1), \mathrm{sumt}(\gamma_2), \gamma_5)$.

By observation of the tout relation and the above facts we have that $\epsilon_3 \subseteq \epsilon_1 \cup \epsilon_2$. The above facts also imply that $\mathrm{live}(\gamma_5, \epsilon_2)$, and $\mathrm{live}(\gamma_6, \epsilon_2)$. Hence to prove $\mathrm{live}(\gamma_5, \epsilon_3)$ and $\mathrm{live}(\gamma_6, \epsilon_3)$ it suffices to prove that $\mathrm{live}(\gamma_5, \epsilon_1)$ and $\mathrm{live}(\gamma_6, \epsilon_1)$.

The definition of tout implies that all effects of γ_3 also exist in γ_4 and the region capability κ_2 of each output effect is defined as $\kappa_2 \equiv \mathrm{out}(\mathsf{rg}, \psi, \kappa_3, \kappa_4, \kappa_1)$, where κ_3,κ_4, and κ_1 are the region capabilities of the corresponding effects of γ_3,γ_4, and γ_1 respectively. By the second assumption the same applies for γ_5 and γ_1.

By the definition of out it is immediate that when a region in γ_3 is live then the corresponding region in γ_1 must also be live. The same applies for regions in γ_4 and γ_2. By the second assumption we also have the same constraints for γ_1 and γ_5, and and γ_2 and γ_6. We also have from the first assumption that $\mathrm{live}(\gamma_1, \epsilon_1)$ and $\mathrm{live}(\gamma_2, \epsilon_1)$. By the above three facts we can conclude that $\mathrm{live}(\gamma_5, \epsilon_1)$ and $\mathrm{live}(\gamma_6, \epsilon_1)$.

To complete the proof we need to prove the remaining premise of the conclusion, namely $\exists \epsilon_1.(\overline{\gamma_2}, \epsilon_1) \equiv \mathrm{tout}(\mathsf{seq}, \mathrm{sumt}(\gamma_3), \mathrm{sumt}(\gamma_4), \overline{\gamma_1})$ (by the first assumption) and $\exists \epsilon_2.(\gamma_6, \epsilon_2) \equiv \mathrm{tout}(\mathsf{seq}, \mathrm{sumt}(\gamma_1), \mathrm{sumt}(\gamma_2), \gamma_5)$ (by the second assumption) then $\exists \epsilon_3.(\gamma_6, \epsilon_3) \equiv \mathrm{tout}(\mathsf{seq}, \mathrm{sumt}(\gamma_3), \mathrm{sumt}(\gamma_4), \gamma_5)$. Intuitively, all region and lock capabilities modified from the transition of γ_5 to γ_6 are exactly the ones modified from the transition of $\overline{\gamma_1}$ to $\overline{\gamma_2}$ as a result of the transition of γ_3 to γ_5. Therefore, we can substitute γ_3 and γ_5 for $\overline{\gamma_1}$ to $\overline{\gamma_2}$ in $\exists \epsilon_2.(\gamma_6, \epsilon_2) \equiv \mathrm{tout}(\mathsf{seq}, \mathrm{sumt}(\gamma_1), \mathrm{sumt}(\gamma_2), \gamma_5)$ and obtain that the resulting output predicate holds. Of course, the resulting ϵ in the exists clause will be sligtly larger but we have proven above that the live predicate will still hold.

Lemma 31 (Live Region-Weakening) $\mathrm{output}(\gamma_7, \mathsf{seq}, \gamma_{1b}, \gamma_3) \equiv \gamma_5 \wedge \mathrm{live}(\overline{\gamma_{1b}}, \{r\}) \Rightarrow \mathrm{live}(\gamma_7, \{r\})$

Proof. The second assumption implies that $\forall r_1 \in (\{r\} \cup \mathit{parents}(\gamma_{1b}, r)).r_1^{\kappa_1, \kappa_2} \triangleright \epsilon \in \mathit{linear}(\gamma_{1b}) \Rightarrow \kappa_1 \not\equiv 0 \wedge \kappa_1 \not\equiv \bar{0}$.

$\mathrm{output}(\gamma_7, \mathsf{seq}, \gamma_{1b}, \gamma_3) \equiv \gamma_5$. It sufficies to show that $\forall r_1 \in (\{r\} \cup \mathit{parents}(\gamma_7, r)).r_1^{\kappa_5, \kappa_6} \triangleright \epsilon \in \mathit{linear}(\gamma_7) \Rightarrow \kappa_5 \not\equiv 0 \wedge \kappa_1 \not\equiv \bar{0}$.

The first assumption implies that $\exists \epsilon.(\gamma_5, \epsilon) \equiv \mathrm{tout}(\mathsf{seq}, \mathrm{sumt}(\gamma_{1b}), \mathrm{sumt}(\gamma_3), \gamma_7)$. In turn the definition of tout implies that all effects of γ_{1b} also exist in γ_7 and the region capability κ_7 of each output effect is defined as $\kappa_7 \equiv \mathrm{out}(\mathsf{rg}, \psi, \kappa_1, \kappa_3, \kappa_5)$, where κ_1,κ_3, and κ_5 are the region capabilities of the corresponding effects of γ_{1b},γ_3, and γ_7 respectively. By observation of out relation we have that $\kappa_1 \not\equiv 0 \wedge \kappa_1 \not\equiv \bar{0}$ implies $\kappa_7 \not\equiv 0 \wedge \kappa_1 \not\equiv \bar{0}$. Therefore, $\forall r_1 \in (\{r\} \cup \mathit{parents}(\gamma_7, r)).r_1^{\kappa_5, \kappa_6} \triangleright \epsilon \in \mathit{linear}(\gamma_7) \Rightarrow \kappa_5 \not\equiv 0 \wedge \kappa_1 \not\equiv \bar{0}$ holds.

Lemma 32 (Accessible Region-Weakening) $\mathrm{output}(\gamma_7, \mathsf{seq}, \gamma_{1b}, \gamma_3) \equiv \gamma_5 \wedge \mathrm{accessible}(\overline{\gamma_{1b}}, \{r\}) \Rightarrow \mathrm{accessible}(\gamma_7, \{r\})$

Proof. Similar reasoning to lemma 31.

Lemma 33 (Capability Modification-Weakening) $\mathrm{output}(\gamma_7, \mathsf{seq}, \gamma_{1b}, \gamma_3) \equiv \gamma_5 \wedge \mathrm{modcap}(\overline{\gamma_{1b}}, \psi, \eta, r) \Rightarrow \mathrm{modcap}(\gamma_7, \psi, \eta, r)$

Proof. By case analysis on ψ.

Case $\psi \equiv \mathsf{rg}$ then the second assumption implies that
$x \equiv r^{\kappa_1, \kappa_2} \triangleright \epsilon_1 \wedge x \in \gamma_{1b} \wedge \kappa_1' \equiv \mathrm{mod}_\eta(\kappa_1) \wedge \gamma_2 \equiv \gamma_1[r^{\kappa_1', \kappa_2} \triangleright \epsilon_1 / x] \wedge \mathrm{inv}(\gamma_2) \wedge \mathrm{live}(\gamma_{1b}, \{r\}) \wedge \kappa_1' \equiv 0 \Rightarrow \kappa_2 \equiv 0 \wedge \kappa_1' \equiv \bar{0} \Rightarrow \kappa_2 \equiv \bar{0}$. It suffices to prove that $x \equiv r^{\kappa_5, \kappa_6} \triangleright \epsilon_1 \wedge x \in \gamma_7 \wedge \kappa_5' \equiv \mathrm{mod}_\eta(\kappa_5) \wedge \gamma_5 \equiv \gamma_1[r^{\kappa_5', \kappa_6} \triangleright \epsilon_1 / x] \wedge \mathrm{inv}(\gamma_5) \wedge \mathrm{live}(\gamma_7, \{r\}) \wedge \kappa_5' \equiv 0 \Rightarrow \kappa_6 \equiv 0 \wedge \kappa_5' \equiv \bar{0} \Rightarrow \kappa_6 \equiv \bar{0}$.

The first assumption implies that $\exists\epsilon.(\gamma_5, \epsilon) \equiv \mathrm{tout}(\mathsf{seq}, \mathrm{sumt}(\gamma_{1b}), \mathrm{sumt}(\gamma_3), \gamma_7)$. In turn the definition of tout implies that all effects of γ_{1b} also exist in γ_7 and the region capability κ_7 of each output effect is defined as $\kappa_7 \equiv \mathrm{out}(\mathsf{rg}, \psi, \kappa_1, \kappa_3, \kappa_5)$, where κ_1, κ_3, and κ_5 are the region capabilities of the corresponding effects of γ_{1b},γ_3, and γ_7 respectively.

By observation of mod_η and out relation we have that $\mathrm{mod}_\eta(\kappa_5)$ is defined. We also have that $\mathrm{inv}(\gamma_7)$ for free by the definition of output. $\mathrm{live}(\gamma_7, \{r\})$ holds by lemma 31. Finally, $\kappa_5' \equiv 0 \Rightarrow \kappa_6 \equiv 0 \wedge \kappa_5' \equiv \bar{0} \Rightarrow \kappa_6 \equiv \bar{0}$ holds by $\mathrm{inv}(\gamma_7)$: Assume κ_5' is equal to 0 or $\bar{0}$ whereas κ_6' is non-zero. Then inv implies that that $\neg\mathrm{iszero}(\kappa_5', \kappa_6') \Rightarrow \mathrm{live}(\gamma_7, \{r\})$, where r is the region corresponding to these effects. However, the above does not hold is κ_5' has a zero value. This is a contradiction.

Case $\psi \equiv \mathsf{lk}$ Similar reasoning can be applied for this case as well.