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, multithreaded 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 transferred or consumed during execution.

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

\[
\begin{align*}
\text{create	extunderscore list} & : \forall \rho. \text{rgn}(\rho) \rightarrow \text{List}[\rho] \\
\text{thread	extunderscore fun}_1 & : \forall \rho. (\text{rgn}(\rho) \times \text{List}[\rho]) \rightarrow () \\
\text{thread	extunderscore fun}_2 & : \forall \rho. (\text{rgn}(\rho) \times \text{List}[\rho]) \rightarrow ()
\end{align*}
\]

Notice that the functions are \textit{region polymorphic}.

Function \text{create	extunderscore list} takes a \textit{region handle} and returns a list allocated in this region. Similarly, \text{thread	extunderscore fun}_1 and \text{thread	extunderscore 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:

\[
\text{def par	extunderscore comp} = \Lambda \rho. \lambda x. \\
\text{newrgn } \rho_1, x_1 \text{ at } x \text{ in} \\
\text{newrgn } \rho_2, x_2 \text{ at } x \text{ in} \\
\text{let } l_1 = \text{call create	extunderscore list }[\rho_1] \ x_1 \text{ in} \\
\text{spawn thread	extunderscore fun}_1 [\rho_1](x_1, l_1); \\
\text{let } l_2 = \text{call create	extunderscore list }[\rho_2] \ x_2 \text{ in} \\
\text{ldec } x_2; \text{rinc } x_2; \\
\text{spawn thread	extunderscore fun}_2 [\rho_2](x_2, l_2); \\
\text{call thread	extunderscore fun}_2 [\rho_2](x_2, l_2)
\]

Function \text{par	extunderscore comp} is a region polymorphic function, which takes a region handle \(x\) and returns the unit value. Using the \text{newrgn} construct, two regions \(\rho_1\) and \(\rho_2\) are allocated within region \(\rho\). The handles of these two regions are \(x_1\) and \(x_2\). By invoking function \text{create	extunderscore list}, a list is then allocated in region \(\rho_1\).
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 $\rho_1$ starts its life with a thread-local lock capability and a single region capability. Subsequently, passing the handle of region $\rho_1$ with a thread-local lock capability to the spawn operation has two implications. First, the obligation for deallocating $\rho_1$ 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 transferred to the spawned thread, which can now access $\rho_1$ without requiring locking. Therefore, $\rho_1$ migrates from the spawning to the spawned thread.

In the following let, a new list is allocated at region $\rho_2$ 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 $\rho_2$ 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 $\text{idec } x_2$ allows the spawning thread to yield the local access right by transforming the lock capability of $\rho_2$ from thread-local to sharable-unlocked. Thus, we have established that $\rho_2$ will not migrate.

The second spawn will definitely consume at least one capability. Therefore, we need to establish that $\rho_2$ will remain live after the spawn operation. Expression $\text{rinc } x_2$ provides the spawning thread with an additional region capability for region $\rho_2$ so that $\rho_2$ will remain live after the spawn operation.

The last expression is an ordinary function application, which applies thread_fun_2 to a pair consisting of the region handle to $\rho_2$ as well as the variable $l_2$. There exists no explicit region decrement command after invoking thread_fun_2 for deallocating $\rho_2$. This is because thread_fun_2 will definitely deallocate $\rho_2$. 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 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 deallocate_both function, which deallocates both regions $\rho_1$ and $\rho_2$, whose type is shown below.

\[
deallocate\_both : \forall \rho_1, \forall \rho_2. (rgn(\rho_1) \times rgn(\rho_2)) \rightarrow \langle \rangle
\]

\(^1\)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 $\rho_1$ and $\rho_2$ with the same region $\rho$ 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 $\rho$ is created with a single region capability, whereas function deallocate_both [$\rho$][$\rho$] attempts to consume two region capabilities from the same region $\rho$.

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 $\rho$ starts (newrgn construct) with a single region capability, and rinc provides $\rho$ with an additional capability. Therefore, its pure region capability is two. The function call aliases the pure region capability of $\rho$ so the type system splits $\rho$ 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 $\ell_1$ is acquired before another lock $\ell_2$, then $\ell_1$ may be released before $\ell_2$ is released.

In the presence of region polymorphism, it is possible to write a function $\text{unlock}_{\text{fst}}\text{access}_{\text{snd}}$ that accepts two region handles and assumes (in an effect annotation, which has been elided in the following example) that the two regions $\rho_1$ and $\rho_2$ are locked (i.e., have a single lock capability).

```
\text{unlock}_{\text{fst}}\text{access}_{\text{snd}} : \forall \rho_1, \forall \rho_2. <\text{rgn}(\rho_1) \times \text{rgn}(\rho_2)> \rightarrow \{
```

As the name of the above function suggests, its body releases the first lock handle and accesses data allocated in the second handle $\rho_2$, as $\rho_2$ 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 * }
```

As in the region deallocation example, this example will also be rejected by our type system as the function call requires two capabilities and $\rho$ has a single lock capability. To preserve soundness, we have adapted the counter-based capability approach for lock capabilities. The following example will type-check as we produce an additional lock capability for $\rho$ before invoking function $\text{unlock}_{\text{fst}}\text{access}_{\text{snd}}$. 

```
```
The type of new treat a pure or fractional capability as a (smaller) fractional capability. Capabilities can be split into several fractional capabilities. Capability subtyping allows to temporarily region and lock capability (new This definition of new noted that thread-local or

Let region and lock capabilities be denoted as \( \kappa_1 \) and \( \kappa_2 \) respectively, then the region effect of region \( \rho_1 \) with parent region \( \rho_2 \) is denoted as \( \rho_1^{\kappa_1,\kappa_2} \triangleright \rho_2 \).

\[
\text{new_int : } \forall \rho_1. \forall \rho_2. (\text{rgn}(\rho_1) \times \text{int}) \xrightarrow{\gamma} \text{ref}(\text{int}, \rho_1) \\
\{ \text{ where } \gamma \equiv \rho_1^{1,1} \triangleright \rho_2, \rho_2^{1,1} \triangleright \emptyset \}
\]

Function new_int is an integer constructor which, given a region handle and an integer value, returns a fresh reference to a location at region \( \rho_1 \) that is initialized to the integer value of the second parameter. The input and output effect lists (\( \gamma \)) 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 new_int invokes no operations that consume capabilities or, if it does, then it restores the original capabilities of \( \rho_1 \) and \( \rho_2 \).

The input effect list consists of two elements: \( \rho_1^{1,1} \triangleright \rho_2 \) and \( \rho_2^{1,1} \triangleright \emptyset \). Both elements have a single region and lock capability (1, 1). Region \( \rho_2 \) 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.

\[
\text{new_int : } \forall \rho_1. \forall \rho_2. (\text{rgn}(\rho_1) \times \text{int}) \xrightarrow{\gamma} \text{ref}(\text{int}, \rho_1) \\
\{ \text{ where } \gamma \equiv \rho_1^{\top,\top} \triangleright \rho_2, \rho_2^{\top,\top} \triangleright \emptyset \}
\]

The type of new_int can now be expressed in terms of fractional capabilities. The function type assumes that the input effect \( \gamma \) contains two region names \( \rho_1 \) and \( \rho_2 \), 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 \( * \).

\[
\text{new_int : } \forall \rho_1. \forall \rho_2. (\text{rgn}(\rho_1) \times \text{int}) \xrightarrow{\gamma} \text{ref}(\text{int}, \rho_1) \\
\{ \text{ where } \gamma \equiv \rho_1^{*,*} \triangleright \rho_2, \rho_2^{*,*} \triangleright \emptyset \}
\]

This definition of 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 \( \rho_1 \triangleright \rho_2 \), which means that \( \rho_1 \) is a subregion of \( \rho_2 \) (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

\[
\text{linc } x_2; \\
\text{call unlock_fst_access_snd }[\rho][\rho](x_2, x_2)
\]

\{ \text{\( x_2 \) is a handle to a locked region} \}
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:

\[
\text{access} : \forall \rho_1. \forall \rho_2. (\text{rgn}(\rho_1) \times \text{rgn}(\rho_2)) \rightarrow \gamma \rightarrow \gamma
\]

{\text{where } \gamma \equiv \rho_1 \triangleright \rho_2, \rho_2 \triangleright \emptyset}

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:

\[
\text{access} : \forall \rho. \text{rgn}(\rho) \rightarrow \gamma \rightarrow \gamma
\]

{\text{where } \gamma \equiv \rho \triangleright \emptyset}

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:

\[
\text{pack } \rho, x \text{ as } \exists \rho. \text{rgn}(\rho)
\]

Existential values may escape the lexical scope of a region, in this case the scope of region \( \rho \), 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 \( \rho \) has a greater lexical scope than \( (\text{outlives}) \rho' \) then the value in the above example can be given the following type:

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

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 \( \epsilon \). To access that region all regions that belong in \( \epsilon \) must be accessible.

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

### 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 (\( \text{rgn} \)) and memory locations (\( \text{loc} \)). Expression syntax is similar to value syntax and has additional elements such as constructs for manipulating references: memory allocation.
Value \( v::= n | (v_1, \ldots, v_n) | \text{rgn} | \text{loc} \)

Expressions \( e::= x | n | (e_1, \ldots, e_n) | \text{prj}_n e \)

| | newrgn \( \rho, x \) at \( e \) in \( \text{rgn} \) | \( \text{cap}^\psi \) \( e \)
| | new \( e \) at \( \rho \) | deref \( e \) | \( e::= e | \text{loc} \)

Capability kind \( \psi::= \text{rg} | \text{lk} \)

Capability op \( \eta::= + | − \)

Figure 1: Core syntax

| Region \( r::= \rho | 1 \)
| Region list \( e::= \emptyset | e, r \)
| Capability \( \kappa::= n | \pi | \perp | * \)
| Effect list \( \gamma::= \emptyset | \gamma, r^{n, \kappa} \triangleright e | \gamma, r^\top \triangleright e \)
| Type \( \tau::= \text{int} | \text{rgn}(r) | (\tau \times \cdots \times \tau) | \text{ref}(\tau, r) \)

| Function \( f::= \lambda x. e \) as \( \tau \triangleleft \gamma \tau \Lambda \rho. f \)
| Operation \( \xi::= \text{seq} | \text{par} \)

Value \( v::= \ldots | f | \text{pack} r, v \) as \( \tau \)

Expressions \( e::= \ldots | f | (e e)^\xi | e[r] \)

| | pack \( r, e \) as \( \tau \) | open \( e \) as \( \rho, x \) in \( e \)
| Program \( P::= \text{def} x = f; | \text{def} x = f; P \)

Figure 2: Full syntax

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

Most constructs are standard, except for the region manipulation constructs. The newrgn \( \rho, x \) at \( e_1 \) in \( e_2 \) construct allocates a new region \( \rho \) 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 \( \rho \) 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 cap\( ^\psi \) \( e \) construct enables region reference count and lock count manipulation. This construct is parameterized by \( \psi \), which can be either a region (rg) or a lock (lk), and \( \eta \), 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 metavariable \( r \), which can be a region variable (\( \rho \)) or a concrete name (\( i \)). The run-time region state is statically tracked via effects. Elements of an effect list \( \gamma \) are of the form \( r^{\text{state} | \triangleright e} \), which means that region \( r \) is at state and depends on regions defined in \( e \). Region state can either be unknown (\( \perp \)) or some state \( \kappa_1, \kappa_2 \) for the region and lock reference count correspondingly.

A capability \( \kappa \) can be a pure capability \( n \), a fractional capability \( \pi \) or an aliasable capability (\( * \)). This applies to both regions and locks. Additionally, a lock capability may also be equal to \( \perp \) 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

\[ I ::= (\check{\iota}, n_1, n_2, n_3) \]

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

\[ E ::= \Box \mid (v, \ldots, E, \ldots, e) \mid \text{proj} \, E \mid (E \, e)^{\xi} \mid (v \, E)^{\xi} \mid E[v] \]
\[ \mid \text{newrgn} \rho, x \text{ at } E \text{ in } e \mid \text{cap}_E^\rho \]
\[ \mid \text{new } E \text{ at } e \mid \text{new } v \text{ at } E \mid \text{deref } E \mid E : e \mid v : E \]
\[ \mid \text{pack } r, E \text{ as } \tau \mid \text{open } E \text{ as } \rho, x \text{ in } e \]

Figure 4: Evaluation contexts

Types consist of integers (\texttt{int}), region handles (\texttt{rgn}(r)), tuples (\langle \tau_1 \times \cdots \times \tau_n \rangle), references (\texttt{ref} (\tau, r)), functions (\tau_1 \rightarrow \tau_2), polymorphic functions (\forall \rho. \tau) and bounded existential (\exists \rho[e]. \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 \( \epsilon \).

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 (\( \check{\iota} \)), 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 \( \check{\iota} \), 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
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-P 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 \( \ell \) when it is live, its lock count is greater than zero and its thread owner is equal to \( \ell \) (in other words, \( \ell \) has successfully obtained a lock for this region).

- \( \text{alloc}(\ell, S, v) \) is used in rule E-NRF for creating a new reference. It allocates a new object in \( S \). The object is placed in region \( \ell \) and is set initially to the value \( v \). Region \( \ell \) must be live. Upon success, the function returns a pair \((\ell, S')\) containing a fresh location of the new object and the new store.

- \( \text{lookup}_i(S, \ell) \) is used in rule E-D to look up the value of location \( \ell \) in \( S \). The region in which \( \ell \) resides must be live and accessible from the currently executed thread \( i \). Upon success, the function returns the value \( v \) held in \( \ell \).

The 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.

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.
• update$_i(S, \ell, v)$ is used in rule E-AS to assign the value $v$ to location $\ell$ in $S$. The region in which $\ell$ resides must be live and accessible from the currently executed thread $i$. Upon success, the function returns the new store $S'$.

• newrgn$(\gamma, S)$ is used in rule E-NR to create a new region in $S$. The new region is made a subregion of $\gamma$, 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.

• updcap$_i(\gamma, S, \psi, \eta)$ is used in rules E-C and E-CB. It requests to obtain or drop (depending on $\eta$) a capability of kind $\psi$ (i.e., region or lock) for region $\gamma$ 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 ($i$) 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 $\tau$ under an input effect list $\gamma$ and results in an output effect list $\gamma'$. 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, $\Delta$ is a set of region variables, and $\Gamma$ 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 $\alpha$-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 $\gamma_1$ and $\gamma_2$. The rationale behind the summarization process is that region substitution in an effect list may violate the
Figure 7: Static semantics
Region capabilities in parallel (i.e., we have region migration). On the other hand, we can permanently convert the local lock capability ⊥ 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 $\gamma_{in}$ 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 $\gamma_{out}$ in this case is defined as the difference between $\kappa_{in}$ and $\kappa_{1}$. 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 $\epsilon$ 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.

<table>
<thead>
<tr>
<th>$\psi$</th>
<th>$\xi$</th>
<th>$\kappa_{1}$</th>
<th>$\kappa_{2}$</th>
<th>$\kappa_{in}$</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>any seq</td>
<td>seq</td>
<td>$n_{3}$</td>
<td>$n_{2}$</td>
<td>$n_{1}$</td>
<td>$n_{2} \leq n_{3}$</td>
</tr>
<tr>
<td>any seq</td>
<td>seq</td>
<td>$n_{3}$</td>
<td>$n_{2}$</td>
<td>$n_{1}$</td>
<td>$n_{2} + n_{3} - n_{1} \leq n_{3}$</td>
</tr>
<tr>
<td>any seq</td>
<td>*</td>
<td>*</td>
<td>$n_{3}$</td>
<td>$n_{2}$</td>
<td>$n_{2} + n_{3} - n_{1} \leq n_{3}$</td>
</tr>
<tr>
<td>any seq</td>
<td>*</td>
<td>*</td>
<td>$n_{3}$</td>
<td>$n_{2}$</td>
<td>$0 &lt; n_{3}$</td>
</tr>
<tr>
<td>any seq</td>
<td>seq</td>
<td>*</td>
<td>*</td>
<td>$n_{3}$</td>
<td>$0 &lt; n_{3}$</td>
</tr>
<tr>
<td>any seq</td>
<td>seq</td>
<td>*</td>
<td>*</td>
<td>$n_{3}$</td>
<td>$n_{3}$</td>
</tr>
<tr>
<td>lk seq</td>
<td>seq</td>
<td>⊥</td>
<td>⊥</td>
<td>⊥</td>
<td>0</td>
</tr>
<tr>
<td>lk seq</td>
<td>seq</td>
<td>⊥</td>
<td>0</td>
<td>⊥</td>
<td>0</td>
</tr>
<tr>
<td>lg seq</td>
<td>seq</td>
<td>*</td>
<td>*</td>
<td>⊥</td>
<td>0</td>
</tr>
<tr>
<td>lg seq</td>
<td>seq</td>
<td>*</td>
<td>*</td>
<td>⊥</td>
<td>0</td>
</tr>
<tr>
<td>rg par</td>
<td>par</td>
<td>$n_{1}$</td>
<td>$n_{2}$</td>
<td>$n_{1}$</td>
<td>$0 &lt; n_{1} \leq n_{2}$</td>
</tr>
<tr>
<td>rg par</td>
<td>par</td>
<td>$n_{1}$</td>
<td>$n_{2}$</td>
<td>$n_{1}$</td>
<td>$0 &lt; n_{1} \leq n_{2}$</td>
</tr>
<tr>
<td>lk par</td>
<td>par</td>
<td>0</td>
<td>0</td>
<td>$n_{1}$</td>
<td>$n_{1}$</td>
</tr>
<tr>
<td>lk par</td>
<td>par</td>
<td>0</td>
<td>0</td>
<td>$n_{1}$</td>
<td>$n_{1}$</td>
</tr>
</tbody>
</table>

Table 1 defines the output capability for any region $r$, given the capability kind $\psi$, the type of application $\xi$ and the three capabilities $\kappa_{in}$, $\kappa_{1}$ and $\kappa_{2}$ for $r$ that are found in $\gamma_{in}$, $\gamma_{1}$ and $\gamma_{2}$, 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 ⊥ 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 ⊥ 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.
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 $\gamma_{\text{out}}$ 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 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 $\text{rgn}(r_1)$ and $\tau$, respectively, and that $\tau$ does not contain any occurrences of the fresh region variable $\rho$. The body expression $e_2$ is typechecked in a context extended with the fresh region $\rho$ and a new effect element is added to the input effect list of $e_2$ ($\rho^1, \tau \triangleright r_1$), which states that there exists a new region $\rho$ with region capability of 1, lock capability $\bot$ (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 $\text{live}(\gamma, r_1)$), 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 modcap($\gamma, \psi, \eta, r$), which modifies by $\eta$ the capability of kind $\psi$ for region $r$ in the effect list $\gamma$ 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 $\epsilon$. The rule for opening an existential package in the scope of an expression $e'$ adds the effect $r^1 \triangleright \epsilon$ 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 $\epsilon$. This form of bounded quantification allows a finer control of region abstraction. Using such effects may be restrictive as all regions in $\epsilon$ 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 $\delta$ 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 \textit{live} regions at the corresponding output effect list.

\[
\begin{align*}
R; M; \Delta; \Gamma; \delta \vdash_T \emptyset \\
R; M; \Delta; \Gamma \vdash e : () & \& (\delta(n) ; \gamma) \\
\text{linear}(\gamma) \equiv \emptyset \\
R; M; \Delta; \Gamma; \delta \vdash_T n : e, T
\end{align*}
\]

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

- The set of region names occurring in the co-domain of \( \delta \) is a subset of the set of region names in \( S \).
- Each region that appears in \( \delta \) (and all of its parents) is live in \( S \). Furthermore, the sum of all region capabilities (for all different threads) for a region in \( \delta \) is equal to the region’s reference count in \( S \).
- All regions that appear in \( S \) but not in the co-domain of \( \delta \) are deallocated (zero region and lock count).
- For each region that appears in \( \delta \), at most one thread in \( \delta \) 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 \textit{well-typed} with respect to \( R; M; \delta \) (we denote this by \( R; M; \delta \vdash_{\text{str}} S \)) when

- \( S \) is consistent with respect to \( \delta \),
- 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 \( \ell \) 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 \textit{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)**

\[
\begin{align*}
R; M; \Delta; \Gamma; \delta \vdash_T T & \quad R; M; \delta \vdash_{\text{str}} S \\
R; M; \Delta; \Gamma; \delta \vdash_C S; T
\end{align*}
\]

**Definition 5 (Not stuck)** A configuration \( S; T \) is \textit{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 \textit{progress} and \textit{preservation} lemmata are first formalized at the \textit{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 $S; T \rightsquigarrow^P S'; T'$, then there exist $R' \supseteq R$, $M' \supseteq M$ and $\delta'$ 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 $R; M; \emptyset; \emptyset \vdash e : \tau \& (\delta(n) \cdot \gamma)$. Then exactly one of the following is true:

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

Lemma 4 (Preservation – Expression) Let $e$ be a well-typed expression with $R; M; \Delta; \Gamma; e \vdash \tau \& (\delta(n) \cdot \gamma)$ 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 $\gamma'$ such that the resulting expression and the resulting store are well-typed with $R'; M'; \Delta; \Gamma; e' : \tau \& (\gamma'; \gamma)$ 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 $\delta$. The input effect $\gamma'$ of the resulting expression $e'$ is placed back in the global mapping $\delta$ and we denote this by $\delta[n \mapsto \gamma']$; a mapping which is identical to $\delta$, only it maps $n$ to $\gamma'$. 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 $\delta_0$ be such that it only maps thread 1 to the input effect list $H^{1,1} \triangleright \emptyset$, in other words, the heap region has reference count equal to 1 and is initially thread-local. Let $I_0 \equiv (H, 1, \neg 1, 0)$ 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 $e_0 \equiv (\text{main} \mid H \ rgn_H)^\text{seq}$.

Theorem 1 (Type Safety) If the initial configuration is well-typed with $R_0; \emptyset; \emptyset; \emptyset \vdash_C S_0; T_0$ and the operational semantics takes any number of steps $S_0; T_0 \rightsquigarrow^P S; T$, 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 ($\text{rgn}_i$) or location values ($\text{loc}_c$) 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 dealllocation 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 \( p \) 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


Appendix

Language Syntax

Value
\[ v ::= f \mid n \mid (v, \ldots, v) \mid \text{rgn} \mid \text{loc}_i \mid \text{pack} \, r, v \text{ as } \tau \]

Expressions
\[ e ::= x \mid n \mid (e, \ldots, e) \mid \text{prj}_n \, e \mid \text{newrg} \, \rho, x \text{ at } e \text{ in } e \mid \text{rgn} \mid \text{cap} \, v, e \]
\[ = \mid \text{new} \, e \text{ at } e \mid \text{deref} \, e \mid e ::= e \mid \text{loc}_i \mid f \mid (e \, e) \mid e \, r \mid \text{pack} \, r, e \text{ as } \tau \]
\[ = \mid \text{open} \, e \text{ as } \rho, x \text{ in } e \]

Capability kind
\[ \psi ::= \text{rg} \mid \text{lk} \]

Capability op
\[ \eta ::= + \mid - \]

Region
\[ r ::= \rho \mid \iota \]

Region list
\[ \epsilon ::= \emptyset \mid \epsilon, \tau \]

Capability
\[ \kappa ::= n \mid \pi \mid \perp \mid \ast \]

Effect list
\[ \gamma ::= \emptyset \mid \gamma, r^n \mid o \epsilon \mid \gamma, r^\top \mid o \epsilon \]

Type
\[ \tau ::= \text{int} \mid \text{rgn} \text{(r)} \mid \langle \tau \times \ldots \times \tau \rangle \mid \text{ref} \,(\tau, r) \]
\[ = \tau \mid \top \mid \bot \mid \exists \, \rho(e). \tau \]

Function
\[ f ::= \lambda x. e \text{ as } \tau \rightarrow \tau \mid \Lambda \rho. \, f \]

Operation
\[ \xi ::= \text{seq} \mid \text{par} \]

Program
\[ P ::= \text{def} \, x = f_1 \mid \text{def} \, x = f \; P \]

Substitution Relation

Definition 1 (Variable Substitution)
\[
x_1[v/x] = v \quad \text{if } x_1 \equiv x \quad \text{otherwise}
\]

Definition 2 (Region substitution)
\[
r_1[r/p] = r \quad \text{if } r_1 \equiv p \quad \text{otherwise}
\]

Definition 3 (Expression Substitution - Term Variable)
\[
e[v/x] = x[v/x] \mid n \mid (e_1[v/x], \ldots, e_n[v/x]) \mid \text{prj}_n \, e_1[v/x] \mid \text{rgn} \mid \text{cap} \, v, e_1[v/x] \]
\[ = \mid \text{new} \, e_1[v/x] \text{ at } e_2[v/x] \mid \text{deref} \, e_1[v/x] \mid \text{in} \, \rho, y \text{ in } e_2[v/x] \]
\[ \text{as } \tau \mid \text{open} \, e_1[v/x] \text{ as } \rho, y \text{ in } e_2[v/x] \quad y \not= x \]
\[ = \mid \text{newrg} \, \rho, y \text{ at } e_1[v/x] \text{ in } e_2[v/x] \quad y \not= x
\]

Definition 4 (Expression Substitution - Type Variable)
\[
f = \lambda x. e[r/p] \text{ as } \tau_1[r/p] \rightarrow \tau_2[r/p] \mid \Lambda \rho'. \, f[r/p] \quad \rho' \not= \rho
\]
\[
e[r/p] = x \mid n \mid (e_1[r/p], \ldots, e_n[r/p]) \mid \text{prj}_n \, e_1[r/p] \mid \text{rgn} \mid \text{cap} \, v, e_1[r/p] \]
\[ = \mid \text{new} \, e_1[r/p] \text{ at } e_2[r/p] \mid \text{deref} \, e_1[r/p] \mid \text{in} \, \rho, y \text{ in } e_2[r/p] \]
\[ = \mid \text{pack} \, r_1, e_1[r/p] \text{ as } \tau[r/p] \mid \text{open} \, e_1[r/p] \text{ as } \rho', x \text{ in } e_1[r/p] \quad \rho' \not= \rho
\]
\[ = \mid \text{newrg} \, \rho', x \text{ at } e_1[r/p] \text{ in } e_1[r/p] \quad \rho' \not= \rho
\]

Definition 5 (Type Substitution)
\[
\tau[r_1/p] = \text{int} \mid \text{rgn} \,(r_1[r_1/p]) \mid \langle \tau_1[r_1/p] \times \ldots \times \tau_n[r_1/p] \rangle \mid \text{ref} \,(\tau_1[r_1/p], \tau_1[r_1/p])
\]
\[ = \tau_1[r_1/p] \rightarrow_{\tau_1[r_1/p]}^{\tau_1[r_1/p]} \tau_2[r_1/p] \quad \rho' \not= \rho
\]
\[ = \exists \rho'(e[r_1/p]) \cdot \tau[r_1/p] \quad \rho' \not= \rho
\]
Definition 6 ($\Gamma$ Substitution - Type Variable)
\[ \Gamma'[r/\rho] = \emptyset | \Gamma_1[r/\rho], x : \sigma[r/\rho] \]

Definition 7 ($\gamma$ Substitution)
\[ \gamma[r_1/\rho] = \emptyset | \gamma_1[r_1/\rho], r[r_1/\rho] \mapsto \epsilon_2[r_1/\rho] \]

\[ \gamma_1[r_1/\rho], r[r_1/\rho] \mapsto \epsilon_2[r_1/\rho] \]

Definition 8 ($\epsilon$ Substitution)
\[ \epsilon[r_1/\rho] = \emptyset | \epsilon_1[r_1/\rho], r[r_1/\rho] \]

Operational Semantics

Helper Judgements and Abbreviations

Definition 9 (Region header-related functions)
Obtain region count $rc$: $rc((i, n_1, n_2, n_3)) \equiv n_1$
Obtain lock count $lc$: $lc((i, n_1, n_2, n_3)) \equiv n_2$
Obtain region name $rn$: $rn((i, n_1, n_2, n_3)) \equiv 2$
Obtain lock name $ln$: $ln((i, n_1, n_2, n_3)) \equiv n_3$
Modify region count $mrc$: $mrc(I, k) \equiv (rn(I), rc(I) + k, le(I), ln(I))$ where $rc(I) + k \geq 0$ and $rc(I) > 0$
Modify lock count $mle$: $mle(I, k) \equiv (rn(I), rc(I), le(I) + k, ln(I))$ where $le(I) + k \geq 0$ and $le(I) > 0$
Modify lock name $mln$: $mln(I, k) \equiv (rn(I), rc(I), lc(I), k)$ where $k \geq 0$

Acquire/Release lock $ar$: $ar((i, n_1, n_2, n_3), n_A, k) \equiv \{ (i, n_1, n_2 + k, n_A) \text{ if } n_3 \equiv n_A \neq 0, n_2 + k \geq 1 \\
(1, n_1, 0, 0) \text{ if } n_3 \equiv n_A \neq 0, n_2 + k = 0 \\
(1, n_1, n_2 + k, n_A) \text{ if } n_3 \equiv 0, n_A \neq 0, n_2 \equiv 0, k \geq 1 \\
(1, n_1, n_2, n_3) \text{ if } n_3 \neq n_A, n_2 + n_3 > 0 \\
(1, n_1, 0, 0) \text{ if } n_3 \equiv 0, n_2 \equiv -1, k \equiv -1 \}

Definition 10 (Store-related Functions and Invariants)
\[ dom_H(S) \equiv \{ 1 | (i, n_1, n_2, n_3) : (H, S_i) \in S \wedge i \in dom_H(S_i) \} \]
\[ dom_I(S) \equiv \{ \ell | (I : (H, S_i) \in S \wedge \ell \rightarrow v \in H) \wedge \ell \in dom_I(S_i) \} \]
\[ dom(S) \equiv \{ \ell \rightarrow v | (I : (H, S_i) \in S \wedge \ell \rightarrow v \in H) \wedge \ell \in dom(S_i) \} \]
\[ children(S, i) \equiv \{ I_1 | I : (H, S_i) \in S \wedge (\forall n | I_1 \in dom_I(S_i)) \wedge I_1 \in children(S, n) \} \]
\[ parents(S, i) \equiv \{ I | I \in dom_I(S) \wedge \exists n | children(S, rn(I)) \} \]
\[ live(S, i) \equiv \forall I \in parents(S, i).rc(I) > 0 \wedge \exists I \in dom_I(S).rn(I) \equiv I \wedge rc(I) > 0 \]

Definition 11 (Search and Update region in $\delta$)
\[ \vdash S_1 = S'_1, I : (H', S'_2) \quad \vdash S_2 = S', I : (H, S_2) \quad \vdash S_1 = S'_1, I : (H, S_2) \]
\[ S_1 : H ; S_2 \vdash_{su} S_3 : I ; H' ; S'_2 \]

Definition 12 (Capability $cap$)
Definition 13 (Abbreviations)

newrgn(j, S) ≡ \{k, S''\} if k /∈ \{rn(I) | dom_t(S) \land live(S, j) \land S; I; H' \land S'; (k, 1, -1, 0) : (\emptyset, \emptyset) \vdash_S S''; I; H' \land S' \land rn(I')\}

updcap(j, S, ψ, η) ≡ S' if updcap(j, S, ψ, η) ≡ S'

alloc(j, S, v) ≡ (ℓ, S') if ℓ /∈ dom_t(S) \land live(S, j) \land S; I; H, ℓ \mapsto v; S' \vdash_S S''; I; H; S' \land rn(I)\equiv κ

xupdate(S, ℓ, v) ≡ (S'', v₁) if S; I; H', ℓ \mapsto v; S' \vdash_S S''; I; H', ℓ \mapsto v₁; S' \land live(S, rn(I)) \land lc(I) \equiv 1 \Rightarrow ln(I) \equiv 1 \land lc(I) > 0

lookup(S, ℓ, v) = \exists I, η \in I : \{I(0, 0) : (ρ, x) \mapsto S; ℓ, v\}

update(S, ℓ, v) ≡ S' if (S', v₁) \equiv xupdate(S, ℓ, v)

threadid(T) ≡ \{1 | i : e ∈ T\}

Operational Semantics

Definition 14 (Operational Semantics Syntax)

Contents

H ::= \emptyset | H, ℓ \mapsto v

Run-time region

I ::= \{i, n₁, n₂, n₃\}

Region list

S ::= \emptyset | S, I : (H, S)

Threads

T ::= \emptyset | T, n : e

Configuration

C ::= S; T

Definition 15 (Operational Semantics - Evaluation Context)

E ::= □ | (v₁, ..., vₘ, e₁, ..., eₙ) | pr_j, E | (E; e) | (E; e) | E[v] | newrgn ρ, x at E in e | cap^n E

| new E at e | new v at E | deref E | E ::= e | v ::= E | pack r, E as τ | open E as ρ, x in e

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

S; i \mapsto ℓ, S'; e' | S; T₁, i : E[e], T₂ \mapsto_P S; T₁, i : E[e'], T₂

(Ε-S)

\begin{align*}
S; T₁, i : E[e], T₂ \mapsto_P S; T₁, i : E[e'], T₂
\end{align*}

(Ε-T)

\begin{align*}
S; T₁, i : E[e], T₂ \mapsto_P S; T₁, i : E[e'], T₂
\end{align*}

(Ε-SN)

\begin{align*}
1 \leq i \leq n
\end{align*}

(Ε-P)

\begin{align*}
S; T₁, i : E[(\langle \lambda x. e \ as \ τ \rangle v)\mapsto], T₂ \mapsto_P S; T₁, i : E[()], T₂, j : e[v/x]
\end{align*}

(Ε-E)

\begin{align*}
S; T₁, i : E[(\langle \lambda x. e \ as \ τ \rangle v)\mapsto], T₂ \mapsto_P S; T₁, i : E[()], T₂, j : e[v/x]
\end{align*}

(Ε-A)

\begin{align*}
S; (\langle \lambda p. f \rangle[r] \mapsto_P S; f[r/p])
\end{align*}

(Ε-EP)

\begin{align*}
\langle \ell, S' \rangle = alloc(j, S, v)
\end{align*}

(Ε-NRF)

\begin{align*}
S; new v at rgn, \mapsto_P S'; loc_c
\end{align*}

(Ε-D)

\begin{align*}
\langle \ell, S' \rangle = update(S, ℓ, v)
\end{align*}

(Ε-AS)

\begin{align*}
\langle k, S' \rangle = newrgn(j, S)
\end{align*}

(Ε-NR)

\begin{align*}
S; newrgn ρ, x at rgn, \mapsto_P S'; e[k/ρ][rgn/d/x]
\end{align*}

(Ε-CN)

\begin{align*}
\langle k, S' \rangle = newrgn(j, S)
\end{align*}

(Ε-OR)

\begin{align*}
S; open (pack j, v as τ) as ρ, x in e \mapsto_P S; e[j/ρ[v/x]]
\end{align*}
Table 1: Capability conversion – Function \( \text{out}(\psi, \xi, \kappa_1, \kappa_2, \kappa_3) \)

<table>
<thead>
<tr>
<th>( \psi )</th>
<th>( \xi )</th>
<th>( \kappa_1 )</th>
<th>( \kappa_2 )</th>
<th>( \kappa_3 )</th>
<th>( \text{out}(\kappa_3, \psi, \xi, \kappa_1, \kappa_2) )</th>
<th>constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>any</td>
<td>seq</td>
<td>( \bar{n}_1 )</td>
<td>( \bar{n}_2 )</td>
<td>( n_4 )</td>
<td>( n_2 )</td>
<td></td>
</tr>
<tr>
<td>any</td>
<td>seq</td>
<td>( \bar{n}_1 )</td>
<td>( \bar{n}_2 )</td>
<td>( n_3 )</td>
<td>( n_2 + n_3 - n_1 )</td>
<td>( n_1 \leq n_3 )</td>
</tr>
<tr>
<td>any</td>
<td>seq</td>
<td>( \bar{n}_1 )</td>
<td>( \bar{n}_2 )</td>
<td>( \bar{n}_3 )</td>
<td>( n_2 + n_3 - n_1 )</td>
<td>( n_1 \leq n_3 )</td>
</tr>
<tr>
<td>any seq</td>
<td>*</td>
<td>*</td>
<td>( n_3 )</td>
<td></td>
<td>0 &lt; ( n_3 )</td>
<td></td>
</tr>
<tr>
<td>any seq</td>
<td>*</td>
<td>*</td>
<td>( \bar{n}_3 )</td>
<td>( \bar{n}_3 )</td>
<td>0 &lt; ( n_3 )</td>
<td></td>
</tr>
<tr>
<td>lk seq</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td></td>
<td>*</td>
<td></td>
</tr>
<tr>
<td>lk seq</td>
<td>( \perp )</td>
<td>( \perp )</td>
<td>( \perp )</td>
<td></td>
<td>( \perp )</td>
<td></td>
</tr>
<tr>
<td>lk seq</td>
<td>( \perp )</td>
<td>0</td>
<td>( \perp )</td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>lk seq</td>
<td>*</td>
<td>*</td>
<td>( \perp )</td>
<td></td>
<td>( \perp )</td>
<td></td>
</tr>
<tr>
<td>rg par</td>
<td>( n_1 )</td>
<td>0</td>
<td>( n_2 )</td>
<td>( n_1 - n_2 )</td>
<td>( 0 \leq n_1 \leq n_2 )</td>
<td></td>
</tr>
<tr>
<td>rg par</td>
<td>( n_1 )</td>
<td>0</td>
<td>( \bar{n}_2 )</td>
<td>( n_1 - n_2 )</td>
<td>( 0 \leq n_1 \leq n_2 )</td>
<td></td>
</tr>
<tr>
<td>lk par</td>
<td>0</td>
<td>0</td>
<td>( n_1 )</td>
<td>( n_1 )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>lk par</td>
<td>0</td>
<td>0</td>
<td>( \bar{n}_2 )</td>
<td>( \bar{n}_2 )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Static Semantics

Static Semantics Helper Judgements and Abbreviations

Definition 17 (Abbreviations)

\[
\begin{align*}
\text{dom}(\gamma) & \equiv \{ r | r^{\kappa_1,\kappa_2} \triangleright\triangleright \varepsilon_1 \in \gamma \} \cup \{ r | r^{\top} \triangleright\triangleright \varepsilon_1 \in \gamma \} \\
\text{select}(\gamma, r_1) & \equiv \{ r^{\kappa_1,\kappa_2} \triangleright\triangleright \varepsilon_1 | r^{\kappa_1,\kappa_2} \triangleright\triangleright \varepsilon_1 \in \gamma \land r \equiv r_1 \} \cup \{ r^{\top} \triangleright\triangleright \varepsilon_1 | r^{\top} \triangleright\triangleright \varepsilon_1 \in \gamma \land r \equiv r_1 \} \\
\text{linear}(\gamma) & \equiv \{ r^{\kappa_1,\kappa_2} \triangleright\triangleright \varepsilon_1 | r^{\kappa_1,\kappa_2} \triangleright\triangleright \varepsilon_1 \in \gamma \} \\
\text{dom}(\Gamma) & \equiv \{ x | x : \tau \in \Gamma \} \\
\text{dom}(M) & \equiv \{ \ell | \ell \rightarrow (\tau, r) \in M \} \\
\text{dom}(P) & \equiv \{ x | \text{def } x = f \in P \} \\
\text{dom}(\delta) & \equiv \{ i | j \rightarrow \gamma \in \delta \land \gamma \notin \emptyset \land \gamma \in \text{dom}(\gamma) \} \\
\text{parents}(\gamma, r) & \equiv \{ r_1 | (r^{\kappa_1,\kappa_2} \triangleright\triangleright \varepsilon \in \gamma \land r^{\top} \triangleright\triangleright \varepsilon \in \gamma) \land r_1 \in \gamma \land r_1 \in \bigcup_{r \in \gamma} \text{parents}(\gamma, r_2) \} \\
\text{children}(\gamma, r) & \equiv \{ r_1 | r_1 \in \text{dom}(\gamma) \land r \in \text{parents}(\gamma, r_1) \} \\
\text{live}(\gamma, \varepsilon) & \equiv \varepsilon \neq \emptyset \land \forall r \in \varepsilon, \forall r_1 \in (\{ r \} \cup \text{parents}(\gamma, r_1)) \cdot r^{\kappa_1,\kappa_2} \triangleright\triangleright \varepsilon \in \text{linear}(\gamma) \Rightarrow \kappa_1 \neq 0 \land \kappa_1 \neq \emptyset \\
\text{accessible}(\gamma, \varepsilon) & \equiv \text{live}(\gamma, \varepsilon) \land \forall r \in \varepsilon, (r^{\kappa_1,\kappa_2} \triangleright\triangleright \varepsilon_1 \in \gamma \land \kappa_2 \neq 0 \land \kappa_2 \neq \emptyset) \lor (r^{\top} \triangleright\triangleright \varepsilon_1 \in \gamma \land \text{accessible}(\gamma, \varepsilon_1))
\end{align*}
\]

Definition 18 (Effect summarization judgement \( \text{sum} \))

\[
\begin{align*}
\text{add}(\kappa_1, \kappa_2, n_3) & \equiv \begin{cases} 
\frac{n_1 + n_2; n_3}{\bar{n}_2} & \text{if } \kappa_1 \equiv \bar{n}_1 \land \kappa_2 \equiv \bar{n}_2 \\
\frac{n_1 + (1 - n_3); 1}{\bar{n}_3} & \text{if } \kappa_1 \equiv \bar{n}_1 \land \kappa_2 \equiv \bar{n}_3 \\
\frac{n_1 + (1 - n_3); 1}{\bar{n}_3} & \text{if } \kappa_1 \equiv \bar{n}_2 \land \kappa_2 \equiv \bar{n}_3 \\
\frac{(1 - n_3); 1}{\bar{n}_3} & \text{if } \kappa_1 \equiv \bar{n}_2 \land \kappa_2 \equiv \bar{n}_2 
\end{cases} \\
\text{sum}(\gamma, n_1, n_2) & \equiv \begin{cases} 
\emptyset & \text{if } \gamma \equiv \emptyset \\
0, r^{\kappa_1,\kappa_2} \triangleright\triangleright \varepsilon & \text{if } \gamma \equiv \emptyset, r^{\kappa_1,\kappa_2} \triangleright\triangleright \varepsilon \\
\text{sum}(\gamma' \cup r^{\kappa_1,\kappa_2} \triangleright\triangleright \varepsilon_1 \cap \gamma_2, n_3, n_4) & \text{if } \gamma \equiv \gamma', r^{\kappa_1,\kappa_2} \triangleright\triangleright \varepsilon_1, r^{\kappa_2,\kappa_3} \triangleright\triangleright \varepsilon_2 \land \text{add}(\kappa_1, \kappa_3, n_3) \equiv (\kappa_5, n_3) \land \text{add}(\kappa_2, \kappa_4, n_2) \equiv (\kappa_6, n_2) \\
\text{notzero}(r^{\kappa_1,\kappa_2} \triangleright\triangleright \varepsilon) & \equiv \begin{cases} 
r^{\kappa_1,\kappa_2} \triangleright\triangleright \varepsilon & \text{if } \kappa_1 \neq 0 \lor \kappa_2 \neq \emptyset \\
\emptyset & \text{otherwise} 
\end{cases} \\
\text{notzero}(x) & \equiv \bigcup_{\gamma \in \text{sum}(\gamma)} \text{notzero}(x)
\end{cases}
\end{align*}
\]

\[
\text{sum}(\gamma) \equiv (\bigcup_{\gamma \in \text{dom}(\text{linear}(\gamma))} \text{sum}(\text{select}((\text{linear}(\gamma), r), 0, 0) \cup (\gamma \setminus (\text{linear}(\gamma) \cup \{ i^{\top} \triangleright\triangleright \varepsilon | i^{\top} \triangleright\triangleright \varepsilon \in \gamma \}))
\]
Definition 19 (Output constraint judgement out)

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

tout(\xi, \gamma_1, \gamma_2, \gamma_3) \equiv \left\{
\begin{array}{ll}
(\emptyset, \emptyset) & \text{if } \gamma_3 \equiv \gamma_2 \equiv 0 \\
(y \cup r \triangleright e, z) & \text{if } y, z \equiv \text{tout}(\xi, \gamma_1 \setminus x, \gamma_2 \setminus x, \gamma_3 \setminus x) \setminus x \equiv r \triangleright e \wedge x \in \gamma_1 \cap \gamma_2 \cap \gamma_3 \\
(y \cup a, z \cup (e \setminus e_1)) & \text{if } (y, z) \equiv \text{tout}(\xi, \gamma_1', \gamma_2', \gamma_3') \land \gamma_1 = \gamma_1', r^{n_1, n_2} \triangleright e_1 \wedge \gamma_2 = \gamma_2', r^{n_3, n_4} \triangleright e_1 \wedge \gamma_3 = \gamma_3', r^{n_5, n_6} \triangleright e_1 \leq e \wedge \kappa_7 \equiv \text{out}(rg, \psi, \kappa_1, \kappa_3, \kappa_5) \wedge \kappa_8 \equiv \text{out}(lk, \psi, \kappa_2, \kappa_4, \kappa_6) \wedge (k_1 \neq 0 \vee k_1 \neq 0) \wedge a \equiv \text{notzero}(r^{n_7, n_8} \triangleright e) \wedge \xi; \kappa_6 \equiv \text{par}; \bot \Rightarrow \kappa_5 \equiv n \equiv k_1
\end{array}
\right.

Definition 20 (Invariant judgement)

\[
iszero(\kappa_1, \kappa_2) \equiv (k_1, k_2 \equiv 00 \lor \kappa_1, \kappa_2 \equiv 00)
\]

\[
\text{invg}(\gamma_1) \equiv \forall r \in r^{n_1, n_2} \triangleright e \in \text{linear}(\gamma_1). (\text{iszero}(\kappa_1, \kappa_2) \Rightarrow \forall r^{n_3, n_4} \triangleright e_1 \in \text{linear(children}(\gamma_1, r)), \text{iszero}(\kappa_3, \kappa_4)) \land (\neg \text{iszero}(\kappa_1, \kappa_2) = \text{invc}(\gamma_1) \equiv \text{invg}(\gamma_1) \land \text{invk}(\gamma_2))
\]

Definition 21 (Environment output constraint judgement)

\[
\text{output}(\gamma_{\text{in}}, \xi, \gamma_1, \gamma_2) \equiv \gamma_{\text{out}} \text{if } (\gamma_{\text{out}}, e_1) \equiv \text{tout}(\xi, \text{sum}(\gamma_1), \text{sum}(\gamma_2), \gamma_{\text{in}}) \land \text{live}(\gamma_{\text{in}}, e_1) \land \text{live}(\gamma_{\text{out}}, e_1) \land \text{inv}(\gamma_{\text{out}})
\]

Definition 22 (Capability mod)

\[
\text{mod}(\gamma, \psi, \eta, r) \equiv \left\{
\begin{array}{ll}
\text{n}_2 + \eta & \text{if } \kappa \equiv \text{n} \land n + \eta \geq 0 \\
\text{n}_2 + \eta & \text{if } \kappa \equiv \text{n} \land n + \eta \geq 0 \\
0 & \text{if } \kappa \eta \equiv -1, \bot
\end{array}
\right.
\]

\[
\text{modcap}(\gamma', \psi, \eta, r) \equiv \left\{
\begin{array}{ll}
\gamma_2 & \text{if } \psi \equiv \text{rg} \land x \equiv r^{n_1, n_2} \triangleright e_1 \land x \in \gamma_1 \land \kappa' \equiv \text{mod}(\kappa_1) \land \gamma_2 \equiv \gamma_1[r^{n_1, n_2} \triangleright e_1 / x] \land \text{inv}(\gamma_2) \land \text{live}(\gamma_1, \{r\}) \\
\gamma_2 & \text{if } \psi \equiv \text{lk} \land x \equiv r^{n_1, n_2} \triangleright e_1 \land x \in \gamma_1 \land \kappa' \equiv \text{mod}(\kappa_2) \land \gamma_2 \equiv \gamma_1[r^{n_1, n_2} \triangleright e_1 / x] \land \text{inv}(\gamma_2) \land \text{live}(\gamma_1, \{r\}) \\
\eta; \kappa_2 \equiv -1; \bot \Rightarrow \kappa_1 \equiv n
\end{array}
\right.
\]

Static Semantics Syntax

Region List \( R ::= \emptyset | R, v \)

Type variable list \( \Delta ::= \emptyset | \Delta, \rho \)

Memory List \( M ::= \emptyset | M, \ell \rightarrow (\tau, \iota) \)

Variable list \( \Gamma ::= \emptyset | \Gamma, x : \tau \)

Typing Context Well-formedness Judgements

Well-formednessJudge

<table>
<thead>
<tr>
<th>Region Well-formedness</th>
<th>Effect Well-formedness</th>
</tr>
</thead>
<tbody>
<tr>
<td>( R; \Delta, \rho \vdash_R p )</td>
<td>( R; \Delta \vdash e_1 )</td>
</tr>
<tr>
<td>( R, \iota; \Delta \vdash_R e )</td>
<td>( R; \Delta \vdash e_1 )</td>
</tr>
<tr>
<td>( R; \Delta \vdash\emptyset )</td>
<td>( R; \Delta \vdash e_1 )</td>
</tr>
<tr>
<td>( \text{Constraint Well-formedness} )</td>
<td>( R; \Delta \vdash_R \gamma )</td>
</tr>
<tr>
<td>( R; \Delta \vdash e )</td>
<td>( R; \Delta \vdash e_1 )</td>
</tr>
<tr>
<td>( R; \Delta \vdash e_1 )</td>
<td>( R; \Delta \vdash e_1 )</td>
</tr>
<tr>
<td>( \text{Type Well-formedness} )</td>
<td>( R; \Delta \vdash_R r )</td>
</tr>
<tr>
<td>( R; \Delta \vdash_R r )</td>
<td>( R; \Delta \vdash_R r )</td>
</tr>
<tr>
<td>( \forall \iota \in [0, n - 1], R; \Delta \vdash_{\iota} \tau_1 )</td>
<td>( R; \Delta \vdash_{\iota} \tau_1 )</td>
</tr>
<tr>
<td>( R; \Delta \vdash_{\iota} \tau )</td>
<td>( R; \Delta \vdash_{\iota} \tau )</td>
</tr>
<tr>
<td>Predicate</td>
<td>Logical Form</td>
</tr>
<tr>
<td>-----------</td>
<td>-------------</td>
</tr>
<tr>
<td>Program Well-formedness</td>
<td>$\text{dom}(P) = \text{dom}(\Gamma_0)$ $\forall x : f \in P, \emptyset ; \emptyset ; \emptyset \vdash f : \Gamma_0(x)$ $\emptyset ; \emptyset$</td>
</tr>
<tr>
<td>Expression Typing Judgements</td>
<td>$\vdash_D R ; M; \Delta; \Gamma; \gamma; \tau ; P \vdash \Gamma_0$ (T-V)</td>
</tr>
<tr>
<td>$\vdash_D R ; M; \Delta; \Gamma; \gamma; \gamma; \tau$ $\vdash_D R ; M; \Delta; \Gamma; \gamma; \gamma; \tau$ $\vdash_D R ; M; \Delta; \Gamma; \gamma; \gamma; \tau$ $\vdash_D R ; M; \Delta; \Gamma; \gamma; \gamma; \tau$ (T-I)</td>
<td></td>
</tr>
<tr>
<td>$\vdash_D R ; M; \Delta; \Gamma; \gamma; \gamma; \tau$ $\vdash_D R ; M; \Delta; \Gamma; \gamma; \gamma; \tau$ (T-U)</td>
<td></td>
</tr>
<tr>
<td>$\vdash_D R ; M; \Delta; \Gamma; \gamma; \gamma; \tau$ $\vdash_D R ; M; \Delta; \Gamma; \gamma; \gamma; \tau$ (T-F)</td>
<td></td>
</tr>
<tr>
<td>$\vdash_D R ; M; \Delta; \Gamma; \gamma; \gamma; \tau$ $\vdash_D R ; M; \Delta; \Gamma; \gamma; \gamma; \tau$ (T-R)</td>
<td></td>
</tr>
<tr>
<td>$\vdash_D R ; M; \Delta; \Gamma; \gamma; \gamma; \tau$ $\vdash_D R ; M; \Delta; \Gamma; \gamma; \gamma; \tau$ (T-Tu)</td>
<td></td>
</tr>
<tr>
<td>$\vdash_D R ; M; \Delta; \Gamma; \gamma; \gamma; \tau$ $\vdash_D R ; M; \Delta; \Gamma; \gamma; \gamma; \tau$ (T-App)</td>
<td></td>
</tr>
<tr>
<td>$\vdash_D R ; M; \Delta; \Gamma; \gamma; \gamma; \tau$ $\vdash_D R ; M; \Delta; \Gamma; \gamma; \gamma; \tau$ (T-NR)</td>
<td></td>
</tr>
<tr>
<td>$\vdash_D R ; M; \Delta; \Gamma; \gamma; \gamma; \tau$ $\vdash_D R ; M; \Delta; \Gamma; \gamma; \gamma; \tau$ (T-A)</td>
<td></td>
</tr>
<tr>
<td>$\vdash_D R ; M; \Delta; \Gamma; \gamma; \gamma; \tau$ $\vdash_D R ; M; \Delta; \Gamma; \gamma; \gamma; \tau$ (T-NRG)</td>
<td></td>
</tr>
<tr>
<td>$\vdash_D R ; M; \Delta; \Gamma; \gamma; \gamma; \tau$ $\vdash_D R ; M; \Delta; \Gamma; \gamma; \gamma; \tau$ (T-EP)</td>
<td></td>
</tr>
<tr>
<td>$\vdash_D R ; M; \Delta; \Gamma; \gamma; \gamma; \tau$ $\vdash_D R ; M; \Delta; \Gamma; \gamma; \gamma; \tau$ (T-TP)</td>
<td></td>
</tr>
</tbody>
</table>

- **Program Well-formedness**: $\text{dom}(P) = \text{dom}(\Gamma_0)$ $\forall x : f \in P, \emptyset ; \emptyset ; \emptyset \vdash f : \Gamma_0(x)$ $\emptyset ; \emptyset$.
- **Expression Typing Judgements**: $\vdash_D R ; M; \Delta; \Gamma; \gamma; \gamma; \tau ; P \vdash \Gamma_0$ (T-V) $\vdash_D R ; M; \Delta; \Gamma; \gamma; \gamma; \tau$ (T-I) $\vdash_D R ; M; \Delta; \Gamma; \gamma; \gamma; \tau$ (T-U) $\vdash_D R ; M; \Delta; \Gamma; \gamma; \gamma; \tau$ (T-F) $\vdash_D R ; M; \Delta; \Gamma; \gamma; \gamma; \tau$ (T-R) $\vdash_D R ; M; \Delta; \Gamma; \gamma; \gamma; \tau$ (T-Tu) $\vdash_D R ; M; \Delta; \Gamma; \gamma; \gamma; \tau$ (T-App) $\vdash_D R ; M; \Delta; \Gamma; \gamma; \gamma; \tau$ (T-NR) $\vdash_D R ; M; \Delta; \Gamma; \gamma; \gamma; \tau$ (T-A) $\vdash_D R ; M; \Delta; \Gamma; \gamma; \gamma; \tau$ (T-NRG) $\vdash_D R ; M; \Delta; \Gamma; \gamma; \gamma; \tau$ (T-EP) $\vdash_D R ; M; \Delta; \Gamma; \gamma; \gamma; \tau$ (T-TP).
Type Safety

Type Safety Syntax

Definition 23 (Program Effect)

Program Effect  \( \delta ::= 0 | \delta, n \mapsto \gamma \)

Type Safety Helper Judgements and Abbreviations

Definition 24 (Abbreviations)

\[
\begin{align*}
rgcap(\delta, m, i) & \equiv \begin{cases} \{n_1\} & \text{if } r^{n_1,m_2} \triangleright \epsilon \in \delta(m) \\ 0 & \text{otherwise} \end{cases} \\
rs\text{um}(\gamma, i) & \equiv \sum_{m \in \text{dom}_i(\delta)} \text{rgcap}(\delta, m, i)
\end{align*}
\]

\[
\begin{align*}
q(\delta, i) & = \bigcup_{m \in \text{dom}_i(\delta)} \{m \mapsto 0, r^{n_1,m_2} \triangleright \epsilon | r^{n_1,m_2} \triangleright \epsilon \in \delta(m)\} \\
Q_0(\delta) & \equiv \forall m \in \text{dom}_i(\delta). \forall i \in \text{dom}(\delta(m)). r^{n_1,m_2} \triangleright \epsilon \in \delta(m) \land n_1 \in \mathbb{N}^* \land \exists n_2 \in \mathbb{N} \land n_2 \lor n_2 \equiv \perp \\
Q_1(\delta) & \equiv \forall m \in \text{dom}_i(\delta). \forall i \in \text{dom}(\delta(m)). r^{n_1,0} \triangleright \epsilon \in \delta(m) \\
Q_2(k, i, m, S) & \equiv (i, n, x, n_2) \in \text{dom}_1(S) \land (k, n_2, x) \in \{(\perp, 0, -1), (0, 0, 0)\} \cup \{(y, m, y) | y \in \mathbb{N}^* \} \land n_1 \in \mathbb{N}^* \\
Q_3(S, \delta) & \equiv \forall i \in \text{dom}(\delta), \text{live}(S, i) \land \exists (t, n, x, n_1) \in \text{dom}_i(S). \text{rs\text{um}}(\delta, i) \equiv n \in \mathbb{N}^* \\
Q_4(S, \delta) & \equiv \forall I \in \text{dom}_i(S), \text{rn}(I) \not\in \text{dom}(\delta) \Rightarrow I \equiv (\text{rn}(I), 0, 0, 0) \\
Q_5(S, \delta, t, S) & \equiv Q_1(q(\delta, i)) \land Q_2(0, i, 0, S) \\
Q_6(\delta, t) & \equiv q(\delta, i) = \delta_i \lor m \mapsto 0, r^{n_1,0} \triangleright \epsilon \land Q_1(\delta_i) \land Q_2(\kappa_2, i, m, S) \land (\kappa_2 \equiv \perp \Rightarrow \delta_i \equiv \emptyset)
\end{align*}
\]

Type Safety Helper Judgements

Definition 25 (Store Consistency)

\[
\begin{align*}
Q_0(\delta) & \quad Q_3(S, \delta) & \quad Q_4(S, \delta) & \quad \forall i \in \text{dom}(\delta). Q_5(\delta, i, S) \lor Q_6(\delta, i, S) \\
\delta \vdash_{\text{cons}} S
\end{align*}
\]

Definition 26 (Store Typing)

\[
\begin{align*}
\delta \vdash_{\text{cons}} S & \quad R = \{\text{rn}(I) | I \in \text{dom}_1(S)\} & \quad \text{dom}_\ell(S) = \text{dom}(M) & \quad \forall \ell \in \text{dom}(M). R; M; \emptyset ; \emptyset \vdash (\text{dom}_H(S))(|) : M(|) & \forall (\emptyset; \emptyset) \\
R; M; \delta \vdash_{\text{str}} S
\end{align*}
\]

Definition 27 (Threads Typing)

\[
\begin{align*}
R; M; \Delta; \Gamma; \delta \vdash_{T} \emptyset & \quad R; M; \Delta; \Gamma; \delta \vdash T & \quad \text{linear}(\gamma) \equiv \emptyset \\
R; M; \Delta; \Gamma; \delta \vdash_{\text{str}} \emptyset & \quad R; M; \Delta; \Gamma; \delta \vdash_{T} n : e, T \\
R; M; \Delta; \Gamma; \delta \vdash_{\text{str}} S; T
\end{align*}
\]

Definition 28 (Configuration Typing)

\[
\begin{align*}
R; M; \Delta; \Gamma; \delta \vdash_{T} T & \quad R; M; \delta \vdash_{\text{str}} S \\
R; M; \Delta; \Gamma; \delta \vdash_{\text{str}} S; T
\end{align*}
\]

Definition 29 (Not stuck)

\[
\begin{align*}
\vdash_{\text{n.s.}} S; T & \quad \forall S^*; T^*. S; T \rightsquigarrow_{P} S^*; T^* & \quad R; M; \Delta; \Gamma; \delta \vdash_{\text{str}} S; T
\end{align*}
\]

26
Proof

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

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

Case 1: $E\{\lambda x. e \ as \ \tau \ v\}_\text{Par}$: Immediate by rule $E-T$.

Case 2: $E[e]$ By applying inversion twice to the configuration typing judgement we have that $R; M; \Delta; \Gamma; \delta \vdash_{\text{c}} S; T$. We use the latter fact in conjunction with the fact that $\Delta, \Gamma$ such that the resulting configuration is well-typed with $R'; M'; \Delta; \Gamma; \delta' \vdash_{\text{c}} S'; T'$.

Proof. By case analysis on the thread evaluation relation:

Case 1: By inversion of configuration typing assumption we obtain the thread typing derivation. By inversion of $R; M; \Delta; \Gamma; \delta \vdash_{\text{c}} S; T$, we get that $R; M; \Delta; \Gamma; \delta \vdash T_1$ and $R; M; \Delta; \Gamma; \delta \vdash T_2$. By observation of the thread typing relation it is obvious that we require a $\delta$ such that there exists a mapping for each thread identifier. We use the latter facts in conjunction with the fact that $\delta \notin \text{threadid}(T_1, T_2)$ to deduce that $R; M; \Delta; \Gamma; \delta[v \mapsto \emptyset] \vdash T_1, T_2$. The output store is identical to the input store, thus it is well-typed by the store typing assumption.

Case 2: By applying inversion to the configuration typing judgement twice we obtain that $R; M; \Delta; \Gamma; \delta \vdash_{\text{c}} S; T$. By applying Lemma 7 we obtain that $R; M; \Delta; \Gamma; \delta \vdash_{\text{c}} S; T$. By applying Lemma 4 to the typing derivation of $e$, the assumption (obtained from $E-S$ rule) $S; e \vdash e' S', e'$, and $R; M; \Delta; \Gamma; \delta \vdash_{\text{c}} S'$ (assumption) we obtain that $\exists e \subseteq R_2, M \subseteq M_2, M_2; \Delta; \Gamma; \delta \vdash e' \vdash \tau, \nu$, and $R; M; \Delta; \Gamma; \delta \vdash_{\text{c}} S'$. 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 $R_2; M_2; \Delta; \Gamma; \delta \vdash_{\text{c}} S_2; T_1, T_2$. By Lemma 8 we can replace $e'$ for $e$ in the evaluation context $E$ (all well-typed in $R_2; M_2$) to obtain $R_2; M_2; \Delta; \Gamma; \delta \vdash_{\text{c}} S_2; T_1, T_2$. We have shown that $R_2; M_2; \Delta; \Gamma; \delta[v \mapsto \emptyset] \vdash_{\text{c}} S_2; T_1, T_2$. The latter two facts imply that the configuration $S'; T_1, T_2$ is well-typed.

Case 3: By performing inversion on the configuration typing judgement twice we obtain that $(\delta) E(\lambda x. e \ as \ \tau \ v)_{\text{Par}}$ is well-typed. By lemma 7 we have that $(\Delta, \Gamma, x : \tau)_{\text{Par}}$ is well-typed. A well-typed unit value is then constructed by using the $\tau$-D rule $R; M; \Delta; \Gamma; \delta \vdash_{\text{c}} \tau, \nu$, which is an immediate result of $R; M; \Delta; \Gamma; \delta \vdash_{\text{c}} \tau, \nu$ (obtained by the application of Lemma 5 on the typing derivation of $(\lambda x. e \ as \ \tau \ v)_{\text{Par}}$). By applying Lemma 8 to the typing derivations of the above facts we have $E[\emptyset]$ is well-typed. We also have that $R; M; \Delta; \Gamma; \delta \vdash_{\text{c}} T_1, T_2$ from the thread typing derivation. Let $\delta' \equiv \delta[v \mapsto \emptyset]$ then by $\delta \notin \text{threadid}(T_1, T_2)$ and the definition of threads typing judgement we have that $R; M; \Delta; \Gamma; \delta' \vdash_{\text{c}} T_1, T_2$.

We have established that the application term is well-typed: $R; M; \Delta; \Gamma; \delta \vdash_{\text{c}} (\lambda x. e \ as \ \tau \ v)_{\text{Par}} : (\lambda \delta) (\tau, \nu)$. By inverting the application term derivation twice we have that $\forall \gamma \in \{\tau_1 \rightarrow \tau_2\}$, $R; M; \Delta; \Gamma; \delta \vdash_{\text{c}} e[v/x] : \tau_2, (\gamma_1; \gamma_2)$. We have established that $R; M; \Delta; \Gamma; \delta \vdash_{\text{c}} T_1, T_2$. To prove the fact that $\gamma \in \text{threadid}(T_1, T_2)$, we have shown that $R; M; \Delta; \Gamma; \delta \vdash_{\text{c}} T_1, T_2$. Thus, to show that the thread list is well-typed it suffices to prove that $e[v/x]$ is well-typed and $\exists \gamma \equiv \text{threadid}(\gamma_1, \gamma_2)$. By applying Lemma 20 to the established fact $\gamma \equiv \text{output}(\delta(i), 1, \gamma_1, \gamma_2)$ we can deduce that $\text{threadid}(\gamma_1, \gamma_2) \equiv \emptyset$.

By the application of Lemma 18 to the typing derivation of $e[v/x]$ and the fact that $\exists \gamma \equiv \text{threadid}(\gamma_1, \gamma_2)$, we have that $R; M; \emptyset; \emptyset \vdash_{\text{c}} \gamma \rightarrow v : \tau_1, (\gamma_1; \gamma_2)$. We use $\gamma$ in the body of $e$ by using Lemma 11 to obtain $R; M; \Delta; \Gamma; \delta \vdash_{\text{c}} e[v/x] : \tau_2, (\gamma_1; \gamma_2)$. Thus, there exists a $\delta \notin \text{threadid}(T_1, T_2) \cup \{\gamma\}$ such that $R; M; \Delta; \Gamma; \delta' \gamma \vdash_{\text{c}} T_1, T_2 : E[\emptyset], T_2 : e[v/x]$. By applying Lemma 18 to the typing derivation of the function abstraction we have that the abstraction is well-typed under an empty $\Delta$ and $\Gamma$. Therefore, there exist no elements of the form $\rho$ 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 $\Delta$. Thus, by the definition of $\gamma$, which excludes elements of the form $\gamma' \rightarrow e$, we have that $\text{threadid}(\gamma_1) \equiv \emptyset$. By applying Lemma 25 to the latter fact, the store typing assumption, and the latter facts imply that the configuration $T, e \equiv \text{output}(\delta(i), 1, \gamma_1, \gamma_2)$ we obtain that $R; M; \Delta; \Gamma; \delta[v \mapsto \emptyset] \vdash_{\text{c}} S$ and $\gamma$ 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 $E[(\lambda x. e \ as \ \tau) v]_{\omega}^{out}$, or
- Case $e$ is of the form $E[e']$ and there exist $S'$ and $e''$ such that $S; e' \rightarrow^P_{\omega} S'; e''$.

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 $T$-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 $v_1 \equiv \text{rgn}_u$. It suffices to show that the premise of $E$-NRG, namely $(\ell, S_1) = \text{newgn}(k, S)$ 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 $\text{live}(\delta(\ell), \{k\})$, 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 $v_2 \equiv \text{rgn}_u$. 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 $(\ell, S_1) = \text{alloc}(j, S, v_1)$. By inversion of $T$-NR we have that $R; M; \emptyset; \emptyset \vdash v_1 : \tau \& (\delta(\ell); \delta())$, and accessible$(\delta(\ell), \{j\})$. By applying lemma 18 on the derivation of $v_1$ we obtain $R; M; \emptyset; \emptyset \vdash v_1 : \tau \& \text{access}(\emptyset; \emptyset)$. By applying lemma 26 to the latter derivation as well as accessible$(\delta(\ell), \{j\})$ and the store typing $M; R; \delta \vdash_{str} S$ we have that $(\ell, S_1) = \text{alloc}(j, S, v_1)$.

Case $T$-CP: The application of lemma 6 to the typing derivation of $v_1$, which is obtained by inverting $T$-NR, yields that $v_2 \equiv \text{rgn}_u$. To perform a single step, we need to prove that $E-C$ or $E-BC$ 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 $\text{updcap}((k, S, \psi, \eta))$ is defined. If $\text{updcap}$ returns an identical store then rule $E-BC$ applies otherwise $E-C$ can be used. By applying lemma 24 to $\text{modcap}(\gamma, \psi, \eta, i)$, which is obtained by inverting $T$-CP, and the store typing derivation, which is obtained from the progress theorem assumption, we have that $\text{updcap}_k(t, S, \psi, \eta)$ is defined.

Case $T$-D: The application of lemma 6 to the typing derivation of $v$, which is obtained by inverting $T$-D, yields $v \equiv \lambda x. e$. Further, inversion of the typing derivation of $\text{Loc}_x$ yields that $\ell \in \text{dom}(M)$ and $\ell \vdash_{E} R; M; \emptyset; \emptyset; \delta(); \delta()$. The latter derivation can be used to construct a typing derivation of a dummy value $v$ of type $M(\ell); R; M; \emptyset; \emptyset; \emptyset; v : M(\ell & \text{access}(\emptyset; \emptyset); \delta())$. By applying lemma 28 on the derivation for the dummy value $v$, $\text{live}(\delta(\ell), \{j\})$, which is obtained by inversion of $T$-D, the fact that $\ell \in \text{dom}(M)$, and the store typing assumption, we have that $(S', v_1) \equiv \text{update}_u(S, \ell, v)$. This implies $v_2 \equiv \text{lookup}_u(S, \ell)$. 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 $S; e \rightarrow^P_{\omega} S'; e'$, then there exist $R' \supseteq R$, $M' \supseteq M$ and $\gamma'$ such that the resulting expression and the resulting store are well-typed with $R'; M'; \Delta; \Gamma \vdash e' : \tau \& (\gamma'; \gamma)$ and $R; M; \delta[n \rightarrow \gamma'] \vdash_{str} S'$.

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 $j^{th}$ 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 $x : \tau \in \Gamma_0$, $\Gamma_0$ is a list of program function names mapped to their types. The premise of T-V $P \vdash \Gamma_0$, in conjunction with the premise of rule E-F, namely $\text{def } x = f \in P$, imply that $\emptyset; \emptyset; \emptyset \vdash f : \tau \& (\emptyset; \emptyset)$. 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, $R; M; \Delta; \Gamma \vdash (\cdot) : (\cdot) (\delta(n); \delta(n))$.

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 context to the input store $S$. Inversion of the T-CP derivation yields $\gamma' \equiv \text{modcap}(\delta(n), \eta, k)$. By applying lemma 24 to the store typing assumption and the modcap fact, we have that $S \neq S' \Rightarrow M; R; [\delta[n \mapsto \gamma']]; \tau_{st}; S'$. We have established that $S' \neq S$, thus the new store $S'$ is well-typed: $M; R; [\delta[n \mapsto \gamma']] \vdash_{st}; S'$.

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

Store: The premise of $S; \text{newrgn } \rho, x \at \text{rgn}_j, \text{in } e_2 \rightarrow \ell'; S_1; e_2[k/\rho][k/x] \rightarrow \Gamma \vdash \delta(n), \delta(n)$.

Finally, the inversion of the typing derivation of newrgn construct yields that $\text{live}(\delta(n), \delta(n))$ holds. The application of lemma 21 to the latter two facts and the store typing assumptions $M; R; k; [\delta[k \mapsto \delta(n), \ell'; \delta; \Gamma]] \vdash_{st}; S_1$.

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 $R; M; \Delta; \rho, \Gamma; x : \text{rgn}(\rho) \vdash e_2 : \tau \& (\gamma_1, \gamma_2)$, where $\gamma_1 \equiv \delta(n), \rho, \ell'; \delta; \Gamma$ and $\rho \notin \text{dom}(\gamma_2)$.

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

The inversion of the typing derivation of newrgn construct yields $R; M; \Delta; \Gamma \vdash \text{rgn}_k : \tau \& (\gamma_1, \gamma_2)$. By applying inversion twice on the typing derivation of 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 deref $v$ is $\vdash_{st}; R; M; \Delta; \Gamma ; \delta(n); \delta(n)$. We can use well-formedness to construct a new derivation for $v$: $R; M; \Delta; \Gamma ; v : \tau \& (\delta(n); \delta(n))$. 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 typing assumption 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 $\text{accessible}(\delta(n), j)$, and $\text{ref}(M(\ell), j)$ is the type of the location $\text{loc}_e$ at the left hand side of the assignment construct. The premise of the operational rule E-AS implies that $(S', v_1) \equiv \text{updates}_e(S, \ell, v)$. 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 ($M; R; \delta \vdash_{st}; S_1$).

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 constructs yield $\text{live}(\delta(n), \{j\})$. The premise of the operational rule E-NRF gives $\ell \equiv \text{alloc}(j, S, \ell, v)$. We can apply lemma 23 to the above facts to obtain that the new store $S'$ is well-typed ($M; \ell \mapsto ((\tau, j); R; \delta \vdash_{st}; S')$. It should be noted that $\tau$ is the type of the initialization value $v$ of the allocation construct.

Case T-E0: By performing a similar proof to T-NRG case, we obtain $R; M; \Delta; \Gamma \vdash e[v/\rho][v_1/x] : \tau \& (\gamma_1, \gamma_4)$, where $\gamma_4 \equiv \delta(n), \tau \vdash e[v/\rho], \gamma_4 \equiv \gamma_1, \tau \vdash e[v/\rho]$ and $\gamma_4$ 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 of $v$ we have $R; M; \Delta; \Gamma \vdash e[v/\rho][v_1/x] : \tau \& (\delta(n), \gamma_4)$. The resulting store is identical to the input store, hence it is well-typed by the preservation assumption.

Case T-App: The store typing 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 $R; M; \Delta; \Gamma \vdash x : \tau_1 \vdash e : \tau_2(\tau_1, \tau_2)$. By inversion of the application derivation we have that $R; M; \Delta; \Gamma \vdash v : \tau_1 \& (\delta(n), \delta(n))$ and $\gamma_4 \equiv \text{output}(\delta(n), \text{seq}, \gamma_1, \gamma_2)$. 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: $R; M; \Delta; \Gamma \vdash e[v/x] : \tau_2(\tau_1, \tau_2) \& (\tau_1, \tau_2)$ and $\gamma_4 \equiv \text{output}(\delta(n), \text{seq}, \gamma_1, \gamma_2)$ to obtain $R; M; \Delta; \Gamma \vdash e[v/x] : \tau_2(\delta(n), \gamma_4)$. 29
Theorem 1 (Type safety) Let $R_0, \delta_0, S_0$ and $T_0$ be defined as in page 9. If the initial configuration $S_0; T_0$ is well-typed with $R_0; \emptyset; \emptyset; \delta_0 \vdash C; S_0; T_0$ and the operational semantics takes any number of steps $S_0; T_0 \rightarrow^* S_n; T_n$, then the resulting configuration $S_n; T_n$ is not stuck.

Proof. The proof is trivial. Assume that we perform $n$ steps, where $n \in \mathbb{N}$. We obtain that $S_n; T_n$ is well typed, in with respect to some $R_n, M_n$ and $\delta_n$, 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) $R; M; \Delta; \Gamma \vdash e : \tau \& (\gamma_1; \gamma_2) \Rightarrow WF R; M; \Delta; \Gamma; \gamma_1; \gamma_2$

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

Lemma 6 (Canonical Forms) $R; M; \Delta; \Gamma \vdash v : \tau \& (\gamma_1; \gamma_2) \Rightarrow$

$\tau \equiv \emptyset \Rightarrow v \equiv \emptyset \land$

$\tau \equiv (\tau_1, \ldots, \tau_n) \Rightarrow v \equiv (v_1, \ldots, v_n) \land$

$\tau \equiv \mathrm{rgn}(i) \Rightarrow (v \equiv \mathrm{rgn}, \land i \in R) \land$

$\tau \equiv \mathrm{ref}(\tau, i) \Rightarrow (v \equiv \mathrm{loc}, \land \ell \leftrightarrow (\tau, i) \in M) \land$

$\tau \equiv \mathrm{int} \Rightarrow v \equiv \emptyset \land$

$\tau \equiv \tau_1 \oplus \tau_2 \Rightarrow v \equiv \lambda x.e \ \mathrm{as} \ \tau_1 \oplus \tau_2 \land$

$\tau \equiv \forall \rho. \tau \Rightarrow v \equiv \exists \rho[e]. \tau \land v \equiv v \equiv \mathrm{pack} \ r, v \equiv \tau$

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

Case $(\mathrm{pr}_j, E)[e]$: The assumption implies that $R; M; \Delta; \Gamma \vdash (\mathrm{pr}_j, E)[e] : \tau_1 \& (\gamma_1; \gamma_2)$ or equivalently $R; M; \Delta; \Gamma \vdash \mathrm{pr}_j, E[e] : \tau_1 \& (\gamma_1; \gamma_2)$. 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 $((e_1, \ldots, E, \ldots e_n))[e]$: Similar to the above proof structure.

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

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

Case $\mathrm{cap}_E[e]$: Similar to the above proof structure.

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

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

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

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

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

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

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

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

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

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 surprise 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).
Lemma 10 (Region Substitution)

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

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

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

Case $((e_1, \ldots, E, \ldots e_n))[e]$: Similar to the above proof structure.

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

Case $(\text{cap}_E e)[e]$: Similar to the above proof structure.

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

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

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

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

Case $(\text{let } r \ := \ E)[e]$: Similar to the above proof structure.

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

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

Lemma 9 (Expression-Redex) $R; M; V; G; \Delta; \Gamma \vdash e : \tau_1 \land (\gamma_1; \gamma_2) \land e \neq v_1 \Rightarrow \exists E[u], E[u] \equiv e \land u \neq v_2$

Proof. Straightforward proof by induction on the typing derivation.

Case $T\text{-I, T-U, T-F, T-L, T-R, T-PRF}$ then the proof is immediate as $e$ is not a value.

Case $T\text{-V}$: Immediate as it holds for $E \equiv \Box$ and $u \equiv x \neq v$.

Case $T\text{-PRF}$: By observing the shape of the expression of $T\text{-PRF}$ typing derivation, $e \equiv \text{prj}_j, e_1$. If $e_1$ is a value then the proof is immediate ($E \equiv \Box$ and $u \equiv \text{prj}_j, e_1$). Otherwise, the application of the induction hypothesis on the typing derivation of $e_1$ (obtained from $T\text{-PRF}$ inversion) yields that $\exists E[u], E[u] \equiv e_1 \land u \neq v_2$. Consequently, $\exists E, \text{prj}_j, E[u] \equiv e \land u \neq v_2$. Equivalently, $\exists E, \text{prj}_j, E[u] \equiv e \land u \neq v_2$.

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

Case $T\text{-App, T-RApp, T-V, T-E0, T-NRG, T-CP, T-D, T-A, T-EP, T-Tu}$: We can perform similar reasoning to prove the remaining cases.

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

Proof. By induction on the typing derivation for expressions.
Case T-I, T-U, T-L, T-R: These derivations have a single premise: ⊢_D R; M; Δ, ρ; Γ; γ, γ_1, γ_2. We shall only discuss the assumptions of the well-formedness derivation that will be affected by the substitution of ρ for ρ, namely R; Δ, ρ ⊢ γ and R; Δ, ρ ⊢ γ_1. The substitution of each occurrence of ρ in Γ and γ_1 allow us to conclude that R; Δ, ρ ⊢ γ_1/r/ρ and R; Δ ⊢ γ_1/r/ρ (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: ⊢_D R; M; Δ; Γ[r/ρ]; γ_1[r/ρ]; γ_1[r/ρ]. Hence it is possible to construct a derivation of the form R; M; Δ; Γ[r/ρ]; v = r & (γ_1[r/ρ]; γ_1[r/ρ]), where v is an integer, a unit value, a region handle or a run-time location.

It should be noted that the type π does not get affected by the region substitution as in the case of T-R the type is of the form rgn(π) and v ∈ R. The same applies to T-L as well-formedness suggests that R ⊢ 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: ⊢_D R; M; Δ; Γ[r/ρ]; γ_1[r/ρ]; γ_1/r/ρ . By applying the induction hypothesis on the derivation of the function body e we have that R; M; Δ; Γ[r/ρ], x : τ[r/ρ] ⊢ e[r/ρ] : τ_1[r/ρ] & (γ_1[r/ρ]; γ_2[r/ρ]; γ_4[r/ρ]). The proof that γ_1[r/ρ] ⊢ out_γ; γ_2 ⊢ output(γ_4[r/ρ]; γ_1[r/ρ]; γ_4[r/ρ]) is trivial. Therefore, we can safely deduce that R; M; Δ; Γ[r/ρ] ⊢ (∆. e & A).e & π : τ[r/ρ] & (γ_1[r/ρ]; γ_4[r/ρ]).

Case T-V: By weakening the well-formedness assumption as we did in the previous case we have that: ⊢_D R; M; Δ; Γ[r/ρ]; γ_1[r/ρ]; γ_1/r/ρ . After substitution of ρ, the premise (x → τ[r/ρ] ∈ (Γ_0, Γ[r/ρ]) still holds as the domain of Γ_0, Γ is the same before and after substitution. Hence the proof we can derive that R; M; Δ; Γ[r/ρ] ⊢ (∆. e & A).e & π : τ[r/ρ] & (γ_1[r/ρ]; γ_4[r/ρ]).

Case T-App: We apply the induction hypothesis on both subexpressions e_1 and e_2 to obtain that R; Δ; Γ[r/ρ] ⊢ e_1 : (τ_1 = κ_1, κ_2) & r/ρ & (γ_1[r/ρ]; γ_2[r/ρ]; γ_4[r/ρ]) and R; M; Δ; Γ[r/ρ] ⊢ e_2 : τ_1[r/ρ] & (γ_1[r/ρ]; γ_4[r/ρ]). The proof that γ_1[r/ρ] ⊢ out_γ; κ_1, κ_2, γ_4 ⊢ output(γ_4[r/ρ]; κ_1, κ_2, γ_4[r/ρ]; γ_4[r/ρ]) is trivial and has not been included here. Therefore, we may conclude that R; M; Δ; Γ[r/ρ] ⊢ (e_1 & e_2) : τ_2[r/ρ] & (γ_1[r/ρ]; γ_4[r/ρ]).

Case T-CP, T-EP, T-Tu, T-PRF, T-Proj, T-RA, 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; Δ; Γ, x : τ_1 ⊢ e : τ_2 & (γ_1; γ_2) ∧ R; M; ∅; σ, v : τ_1 (σ) ⊢ R; M; Δ; Γ ⊢ e[v/x] : τ_2 & (γ_1; γ_2)

Proof. Straightforward induction on the expression typing derivation.

Lemma 12 (Context Weakening) R; M; Δ, Γ ⊢ e : τ_1 & (γ_1; γ_2) ∧ γ_8 ⊢ output(γ_5, seq, γ_6, γ_7) ⇒ R; M; Δ, Γ ⊢ e : τ_1 & (γ_1; γ_2)

Proof. This lemma can be directly derived by applying lemma 29 to the assumptions of this lemma. The application yields ∃γ_9, R; Δ ⊢ γ_9 ∧ γ_9 ⊢ output(γ_5, seq, γ_6, γ_7) ∧ γ_8 ⊢ output(γ_4, seq, γ_6, γ_7) ∧ R; M; Δ, Γ ⊢ e : τ_1 (γ_1; γ_9).

By observing γ_9 ⊢ output(γ_6, seq, γ_6, γ_8) it is immediate that γ_9 ⊢ output(γ_6, seq, γ_6, γ_8). Thus, R; M; Δ, Γ ⊢ e : τ_1 (γ_1; γ_9)

Lemma 13 (Value-Effect) R; M; Δ; Γ ⊢ v : τ_1 & (γ_1; γ_2) ∧ R; Δ ⊢ γ_3 ⇒ R; M; Δ; Γ ⊢ v : τ_1 & (γ_1; γ_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: ⊢_D R; M; Δ, Γ; γ_1; γ_2. By observing the premises of this derivation we have R; Δ ⊢ γ_1. It is possible to replace R; Δ with γ_1 with the assumption derivation, namely R; Δ ⊢ γ_1 to obtain ⊢_D R; M; Δ, Γ; γ_3; γ_1. 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; Δ, Γ ⊢ v : τ_1 & (γ_1; γ_3).

Lemma 14 (Region Context Weakening) R ⊆ R_1 & R; M; Δ, Γ ⊢ e : τ & (γ_1; γ_3) ⇒ R_1; M; Δ, Γ ⊢ e : τ & (γ_1; γ_3)

Proof. Straightforward induction on the expression typing derivation.

The base case is to weaken the well-formedness judgement of value typing derivations so that ⊢_D R; M; Δ, Γ; γ_1; γ_2 becomes ⊢_D R_1; M; Δ, Γ; γ_1; γ_2. The well-formedness 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 remaining premises of value typing derivations to obtain R_1; M; Δ, Γ ⊢ v : τ & (γ_1; γ_3). In the case of T-R we need to show that v ∈ R_1 but this is immediate as v ∈ R and R ⊆ R_1.

The inductive step is to prove that for each non-value typing derivation it is possible to prove R weakening. This can be shown by applying the induction hypothesis on its subexpression typing derivations and re-writing the derivation using the weakened typing premises. In some derivations we also need to prove that if R; Δ ⊢ r & R; Δ ⊢ τ then R_1; Δ ⊢ r & R; Δ ⊢ τ respectively. Again, both proofs are immediate by observing their structure and using facts from the well-formedness weakening (i.e. ∀v ∈ R, R_1; Δ ⊢ r sufficiently to prove both).
Lemma 15 (Memory Context Weakening) \( M \subseteq M_1 \land R; M; \Delta; \Gamma \vdash e : \tau \& (\gamma_1; \gamma_1) \Rightarrow R; M_1; \Delta_1; \Gamma \vdash e : \tau \& (\gamma_1; \gamma_1) \)

Proof. Similar to the previous proof.

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

Proof. Similar to the previous proof.

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

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 strengthening, 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) \( R; M; \Delta; \Gamma \vdash e : \tau \& (\gamma_1; i^T \triangleright \epsilon, i; \gamma_2; i^T \triangleright \epsilon, i) \Rightarrow R; M; \Delta; \Gamma \vdash e : \tau \& (\gamma_1; \gamma_2) \)

Proof. By induction on the expression typing derivations.

Case T-I, T-U, T-T, 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 \( \vdash_D R; M; \Delta; \Gamma; \gamma_1; \gamma_1 \) 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): \( \vdash_D R; M; \Delta; \Gamma; \emptyset; \emptyset \). 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-App, 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 \( R; M; \Delta; \Gamma \vdash e_1 : \tau_1; \gamma_1 \triangleright \epsilon, \eta; \gamma_1 \triangleright \epsilon, \epsilon \) 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): \( \vdash_D R; M; \Delta; \Gamma; \emptyset; \emptyset \). The strengthened well-formedness derivation along with the unaffected premises of the value derivations yield: \( R; M; \Delta; \Gamma \vdash e_1 : \tau_1; \gamma_1 \triangleright \epsilon, \eta; \gamma_1 \triangleright \epsilon, \epsilon \).

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 \( R; M; \Delta; \Gamma \vdash e_1 : x; \gamma_1 \triangleright \epsilon, \eta; \gamma_1 \triangleright \epsilon, \epsilon \). The second step requires to prove that \( \gamma_2; i^T \triangleright \epsilon, \epsilon, \psi; \eta; \tau \vdash_{\text{temp}} \gamma_3; i^T \triangleright \epsilon, \epsilon \). This is immediate by observing that \( \text{temp} \) judgement ignores elements of the form \( i^T \triangleright \epsilon, i \).

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 \( \gamma_2; i^T \triangleright \epsilon, \epsilon, \psi; \eta; \tau \vdash_{\text{liv}} \gamma_3; i^T \triangleright \epsilon, \epsilon, \eta \) implies \( \gamma_2; \tau_1 \vdash_{\text{liv}} \gamma_3; i^T \triangleright \epsilon, \epsilon, \eta \). This can be proven by a straightforward observation of \( \text{liv} \) relation: elements of the form \( j^T \triangleright \epsilon, \epsilon \) are ignored when encountered in \( \gamma \). Therefore, it is possible to remove \( i^T \triangleright \epsilon, \epsilon \) from \( \gamma_2; i^T \triangleright \epsilon, \epsilon, \eta \) and still obtain the same outcome \( \gamma_2; r_1 \vdash_{\text{liv}} \gamma_3; i^T \triangleright \epsilon, \epsilon, \eta \).

Case T-DRG: Similar to the previous for T-NRG.

Case T-D, T-A: Similarly, we apply the induction hypothesis on the sub-expression derivations to remove \( i^T \triangleright \epsilon, \epsilon \) from the input and output effects. We also have to prove that \( \gamma_2; i^T \triangleright \epsilon, \epsilon, \psi; \eta; \tau \vdash_{\text{adl}} \gamma_3; i^T \triangleright \epsilon, \epsilon, \eta \) implies \( \gamma_2 \vdash_{\text{adl}} \tau_1 \). The proof is the same as for \( \text{liv} \) (see \( T - NRG \)).

Lemma 20 (Spawn-Deallocate) \( \text{output}(\gamma_1; \text{par}, \gamma_2; \gamma_3) \equiv \gamma_4 \Rightarrow \text{linear}(\gamma_4) \equiv \emptyset \)
Lemma 22 (Preservation-Update Value) \( \forall \gamma_2, \gamma_3 : \gamma_2 \equiv \gamma_3 \) implies \( \exists \xi. (\gamma_4, \xi) \equiv \text{tout}(\text{par}, \text{sumt}(\gamma_2), \text{sumt}(\gamma_3), \gamma_1) \). By the definition of \( \gamma_2, \gamma_3 \) 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 \( \text{tout} \) for effects, whose exponent is of the form \( \kappa_1, \kappa_2 \) (as mentioned earlier other kinds of effects are not transformed), and given the fact that \( \xi \equiv \text{par} \), the capability conversion table 1 yields that the region capability \( \kappa'_1 \) of each transformed effect will be zero.

Lemma 21 (Preservation-Add Region) \( M; R; \delta \vdash_{\text{str}} S \land \text{live}(\delta(k), \{j\}) \land (i, S_i) \equiv \text{newrgn}(j, S) \land t \notin \{\text{rn}(I) \mid I \in \text{dom}_i(S_i)\} \Rightarrow M; R; k; \delta[k \mapsto \delta(k), 1^{1 \mapsto} j] \vdash_{\text{str}} S_1 \)

Proof. To prove \( M; R; k; \delta[k \mapsto \delta(k), 1^{1 \mapsto} j] \vdash_{\text{str}} S_1 \) it suffices to prove only two of its premises, namely \( \delta[k \mapsto \delta(k), 1^{1 \mapsto} j] \vdash_{\text{str}} S_1 \) and \( R; t \equiv \{\text{rn}(I) \mid I \in \text{dom}_i(S_i)\} \), 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 \( S \). Therefore, those premises, which hold for \( S \) also hold for \( S_1 \).

By inversion of the store typing derivation of \( S \) we have that \( R = \{\text{rn}(I) \mid I \in \text{dom}(S)\} \). Thus, it is immediate that \( R; t = \{\text{rn}(I) \mid I \in \text{dom}_i(S_i)\} = \text{dom}(S), (i, 1 - 1, 0) \).

To prove \( \delta[k \mapsto \delta(k), 1^{1 \mapsto} j] \vdash_{\text{str}} S_1 \) then \( Q_0(\delta, k \mapsto \gamma, 1^{1 \mapsto} j) \), \( Q_3(S_1, \delta, k \mapsto \gamma, 1^{1 \mapsto} j) \), \( Q_4(S_1, \delta, k \mapsto \gamma, 1^{1 \mapsto} j) \) and \( \forall t \in \text{dom}(\delta) \). From inversion of \( \text{ns} \) for store \( S \) we have that \( Q_0(\delta), Q_4(S, \delta), Q_3(S, \delta), \) and \( \forall t \in \text{dom}(\delta) \). \( Q_5(\delta, t, S) \land Q_6(\delta, t, S) \).

Therefore \( \delta[k \mapsto \gamma, 1^{1 \mapsto} j] \vdash_{\text{str}} S_1 \).

Lemma 22 (Preservation-Update Value) \( M; R; \delta \vdash_{\text{str}} S \land \text{accessible}(\delta(j), \{t\}) \land (S, v') \equiv \text{xupdate}(S, t, v) \land R; M; \Delta; \Gamma \vdash v : M(\ell) \land (\gamma_1; \gamma_1) \Rightarrow M; R; \delta \vdash_{\text{str}} S_1 \)

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

The third assumption also tells us that \( \ell \mapsto v_1 \in \text{dom}_i(S) \). The first assumption implies that \( R; M; \emptyset ; \emptyset \vdash v_1 : M(\ell) \land (\emptyset; \emptyset) \). It suffices to show \( R; M; \emptyset ; \emptyset \vdash v : M(\ell) \land (\emptyset; \emptyset) \) 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) \( M; R; \delta \vdash_{\text{str}} S \land R; M; \Delta; \Gamma \vdash \tau \land (\gamma_1; \gamma_1) \land \text{live}(\delta(k), \{t\}) \land (\ell, S_i) \equiv \text{alloc}(t, S, v) \Rightarrow M; \ell \vdash \tau; R; \delta \vdash_{\text{str}} S_1 \)

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

The fifth assumption also tells us that \( \ell \notin \{t_1 | t_1 \mapsto v_1 \in \text{dom}_i(S)\} \). It suffices to show \( R; M; \ell \mapsto \tau; \emptyset ; \emptyset \vdash v : \tau \land (\emptyset; \emptyset) \) as for all other heap locations and their corresponding values, which are identical to the ones of \( S \), we have from \( \text{str} \) assumption that \( \forall t_2 \in \{t_1 | t_1 \mapsto v_1 \in \text{dom}_i(S)\} \). Thus by applying lemma 15 to each one of them we have that: \( \forall t_2 \in \{t_1 | t_1 \mapsto v_1 \in \text{dom}_i(S)\}. R; M; \ell \mapsto \tau; \emptyset ; \emptyset \vdash v_2 : M(\ell_2) \land (\emptyset; \emptyset) \). As mentioned earlier, these locations are identical for \( S_1 \) hence, \( \forall t_2 \in \{t_1 | t_1 \mapsto v_1 \in \text{dom}_i(S_1)\} \backslash \{\ell\}. R; M; \ell \mapsto \tau; \emptyset ; \emptyset \vdash v_2 : M(\ell_2) \land (\emptyset; \emptyset) \).

The proof is completed by applying lemma 15 and lemma 18 to the second assumption.

Lemma 24 (Preservation/Progress-Capability Modification) \( M; R; \delta \vdash_{\text{str}} S \land \gamma' \equiv \text{modcap}(\gamma, \psi, \eta, i) \Rightarrow \text{updcap}_i(t, S, \psi, \eta) \equiv S_1 \land (S_i \neq S \Rightarrow M; R; \delta[k \mapsto \gamma'] \vdash_{\text{str}} S_1) \)

Proof. Proof by case analysis on \( \psi \).

34
Case $\psi \equiv R$: Assume that $\delta(k) = \gamma_1, r^{1-r_2} \triangleright \epsilon, \gamma = \gamma_1, r^{3-r_2} \triangleright \epsilon$. Then by inversion of the second assumption we have that:

- $\kappa_3 \equiv \text{mod}_\eta(\kappa_2)$
- $\kappa_3 > 0 \lor \kappa_3 \geq 0$.

The store typing (first) assumption implies $\delta \vdash_{cns, S}$ which in turn implies $Q_0(\delta)$. $Q_0$ states that $\kappa_3$ is of the form $n_1$ (i.e., not of the form $n_2$). Hence it holds that $\kappa_3 \geq 0$.

- case $(\eta \equiv -1 \land \kappa_2 \geq 1) \land (\eta \equiv 1)$: By inversion of $cns$ we also have $Q_0(S, \delta)$, which states the fact that $i$ belongs to $\text{dom}(\delta)$ implies $\text{live}(S, \{i\})$ and there exists some $I \in \text{dom}(S)$ such that $rn(I) = 1$ and $rc(I) = n_1 \in \mathbb{N}^\ast$. From the last two facts we can derive $\text{updcap}(i, S, \psi, \eta) \equiv S_1, S_1 \neq S$ hence the last step is to prove $M; R; \delta[k \mapsto \gamma] |_{s \neq S_1}$.

It suffices to prove $\delta[k \mapsto \gamma] |_{cns, S_1}$ as the remaining premises of $\text{str}$ 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 $\delta \vdash_{cns, S}$:

- $Q_0(\delta), Q_0(S, \delta), \forall \gamma \in \text{dom}(\delta), Q_0(S, \delta, S) \lor Q_0(S, \delta, 3)$

- $Q_0(S, \delta)$. The first group of invariants, namely $\psi_0, \psi_1, \psi_2$ also hold for $S_1$ and $\delta[k \mapsto \gamma]$. This is because $\delta[k \mapsto \gamma]$ and $\delta$ differ only in $\kappa_3$. $Q_0$ holds as $\kappa_3 \in \mathbb{N}^\ast$. $Q_0$ holds as $\text{dom}(\delta) = \text{dom}(\delta[k \mapsto \gamma])$, $\psi_1$ and $\psi_2$ are related to lock capabilities of each region in $\text{dom}(\delta[k \mapsto \gamma])$, which are identical to the ones of $\delta$. Similarly, the run-time lock counts of region headers are identical to the ones of $S$. Therefore, $Q_0, Q_2, Q_3$ hold for $S_1$ and $\delta[k \mapsto \gamma]$. It suffices to prove that $Q_0(S_1, \delta[k \mapsto \gamma])$ to complete the proof. Assume that $\delta' = \delta \triangleright (\delta[k \mapsto \gamma])$ and $\delta'' = (\delta[k \mapsto \gamma]) \setminus (\delta'(k \mapsto \gamma), i)$ then $\delta'' = \delta$ as other region effects than $i$ are identical. Therefore, we can deduce that $Q_0(S_1', \delta)$ by using the fact that $Q_0(S_1, \delta)$ (directly derived from $Q_0(S, \delta)$) as well as the fact that regions of $S_1$ other than $i$ are identical as the ones of $S$. Therefore, we only need to prove $Q_0(S_1, \delta)$.

- case $\eta \equiv 1$: Assume that $I \in \text{dom}(S)$ and $rn(I) = 1$ then from $Q_0(S, \delta, i)$ we have that $\text{rsum}(\delta[i \mapsto \gamma]) = rc(I) = n_1$ or $\text{rsum}(\delta[i \mapsto \gamma]) = rc(I) = n_1$. Therefore, $\text{rsum}(\delta[k \mapsto \gamma]) = rc(I) = n_1$ or $\text{rsum}(\delta[k \mapsto \gamma]) = rc(I) = n_2 + n_3$. By the fact that $\kappa_3 \equiv \text{mod}_\eta(\kappa_2)$ or $\kappa_3 = n_3 + 1$ we can deduce that $\text{rsum}(\delta[k \mapsto \gamma]) = rc(I) + 1$. This holds as the new region header $I'$ of store $S_1$ has the following properties $rn(I') = 1$ and $rc(I') = rc(I) + 1$. Therefore, $Q_0(S_1, \delta)$.

- case $\eta \equiv -1$ and $\kappa_2 > 1$: As in the previous case we prove the proof of the previous case.

- case $\eta \equiv -1 \land \kappa_2 \equiv 1$: As in the previous case we have that $\text{live}(S, \{i\})$ by inversion of $cns$ and the fact that $i \in \text{dom}(\delta)$. If subregions of $i$ have a non-zero region count then $\psi_0$ will return an $S_1$ such that $S_1 \neq S$. In this case the proof is immediate as $S_1 \neq S$ does not hold. If $S_1 \neq S$ then it suffices to prove $\delta[k \mapsto \gamma] |_{cns, S_1}$ in order to prove $\text{str}$, 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 $\text{str}$ for $S$ also hold for $S_1$ (except for $cns$ that will be proved shortly).

- $Q_0(S, \delta)$, $\forall \gamma \in \text{dom}(\delta), Q_0(S, \delta, S) \lor Q_0(S, \delta, 3)$ hold, hence $Q_0(S_1, \delta), \forall \gamma \in \text{dom}(\delta), Q_0(S_1', \delta, S)$ also hold as $\delta' \leq \delta$. $\forall I \in \text{dom}(S_1)$, $rn(I) \neq 1 \implies I \in \text{dom}(S_1)$, and $\forall \gamma \in \text{dom}(\delta), \text{live}(S_1, \{i\})$ hold. The last property holds as there exists no child region of $i$ in $\delta'$. This can be deduced by observing rule $\forall cap$: All child regions must have a region count of zero before decrementing $i$. $Q_0(S, \delta)$ tells us that region counts of regions that belong in $\delta$ must be non-zero. $\delta'$ is a subset of $\delta$ so it also does not contain child regions of $i$. As for remaining regions, other than $i$ and its child regions, $\text{live}(S_1, \{i\})$ holds by $Q_0(S, \delta)$ and the fact that $\delta' \leq \delta$ and $\forall I \in \text{dom}(S_1)$. $rn(I) \neq 1 \implies I \in \text{dom}(S_1)$.

- From $\text{tcap}$ assumption we have $\kappa_2 = 0$, otherwise $\text{tcap}$ judgement would not hold. As mentioned earlier, $i$ belongs only to thread that $k$ is described that the sum of all thread counts must be equal to the run-time lock count. We know that we have a single thread $k$ that owns $i$ and $k \mapsto \gamma, r^{1-r_2} \triangleright \epsilon$ as $\kappa_2 = 0$. Therefore, if $I \in \text{dom}(S)$ and $I' \in \text{dom}(S)$ and $rn(I) = \text{rsum}(I') = 1$, then $lc(I) = 1$. $I'$ differs in respect to $I$ in that its region count is zero therefore: $I' = (t, 0, 0, 0)$ (the last element of this tuple is zero as when a lock reaches the value zero, the thread identifier becomes zero as well). Now, that we have $Q_0(S, \delta), \delta = \delta', k \mapsto \gamma, r^{1-r_2} \triangleright \epsilon, \forall I \in \text{dom}(S_1)$, $rn(I) \neq 1 \implies I \in \text{dom}(S_1), \text{and} I' = (t, 0, 0, 0)$, thus $Q_0(S_1, \delta')$ holds.

Case $\psi \equiv L$: By inversion of the $\text{str}$ assumption we have that $\forall \gamma \in \text{dom}(\delta), Q_0(S, \delta, S) \lor Q_0(S, \delta, S)$, hence $Q_0(S, \delta, S) \lor Q_0(S, \delta, S)$ also holds ($\text{tcap}$ assumption implies that $i \in \text{dom}(\delta)$). We proceed by performing a case analysis:

- Case $Q_0(S, \delta, S)$ holds then the lock count of $i$ is zero as well as its the thread owner. More formally, $\exists I \in \text{dom}(S), \forall 0 

\text{Assumption } \text{tcap} \text{ can only hold if } \eta = 1. \text{ Therefore, } ar(I, 1, k)$ is definable. To complete our proof that $\text{cap}$ holds, we need to prove $\text{live}(S, \{i\})$. This can be obtained from $Q_0(S, \delta)$. Thus, $\text{updcap}(i, S, \{i\}) \equiv S_1$ holds. $\text{Function } ar \text{ modifies } I$, hence $S \neq S$. Therefore, we need to prove that $M; R; \delta[k \mapsto \gamma] |_{s \neq S_1}$. As mentioned above, $S_1$ differs from $S$ in respect to the region header $I$. Hence, all premises of $\text{str}$ 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 $\delta' \vdash_{cns, S} \text{in order to complete our proof.}$ By observation of $\text{tcap}$ is obvious that only effect $i$ changes. Hence if $\delta(k) = \gamma_1, r^{1-r_2} \triangleright \epsilon$ then $\gamma = \gamma_1, r^{1-r_2} \triangleright \epsilon$. 

35
Lemma 25 (Preservation-Spawn-Capability Modification) \( M; R; \delta \vdash \text{str}, S \& \delta(i); \text{par} \vdash \gamma_1; \gamma_2 \Rightarrow \gamma_3 \& \gamma_1 = \text{linear}(\gamma_1) \Rightarrow M; R; \delta(j) \mapsto \gamma_3, i \mapsto \gamma_3 \vdash \text{str}, S \).

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

Let \( \delta' = \delta[j] \mapsto \gamma_3, j \mapsto \gamma_4 \), \( \delta_1 = \bigcup_{p \in \text{dom}(\gamma_3)[j=\emptyset]} q(\delta', p) \), and \( \delta_2 = \bigcup_{p \not\in \text{dom}(\gamma_3)[j=\emptyset]} q(\delta', p) \).

By inversion of \( M; R; \delta \vdash \text{str}, S \) we have that \( \delta \vdash \text{cns}, S \). By applying inversion on \( \text{cns} \) we have that:

- \( Q_0(\delta) = Q_0(\delta[j] \mapsto (\delta(j)) = Q_0(\delta[j] \mapsto \emptyset) \subseteq Q_0(j \mapsto \delta(j)). \)
- \( Q_3(S, \delta) = Q_3(S, \delta_1) \cup Q_3(S, \delta_2). \)
- \( Q_4(S, \delta) \)
- \( \forall k \in \text{dom}(\delta). Q_5(\delta, k, S) \land Q_6(\delta, k, S) \equiv \forall k \in \text{dom}(\delta_1). Q_5(\delta_1, k, S) \land Q_6(\delta_1, k, S) \land \forall k \in \text{dom}(\delta_2). Q_5(\delta_2, k, S) \land Q_6(\delta_2, k, S) \)

To complete our proof we need to show that the above premises hold for \( \delta' \).

- \( Q_4(S, \delta') \) holds as \( Q_4 \) only examines the domain of \( \delta' \) and the second and third assumption imply that \( \text{dom}(\delta') = \text{dom}(\delta) \).
- \( Q_0(\delta(j)) \) by observing the derivation tree of the first assumption: To type-check a function input effect its region capability (see function \( F_1 \)) 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 \( \delta(j) \). Hence, \( Q_0(\delta(j)) \) holds. We can combine this fact with an earlier fact, in particular \( Q_0(\delta[j]) \) to prove that \( Q_0(\delta') \) holds.

As mentioned in the previous case, for each region \( k \) that belongs in \( \text{dom}(\gamma_4) \) has a region capability of the form \( n \), then \( k \) belongs in \( \text{dom}(\delta) \), and if \( k \) belongs to \( \text{dom}(\gamma_3) \) then its region capability is \( n-1 \). Therefore, if \( k \) belongs to \( \text{dom}(\gamma_3) \) then the overall region capability sum of \( \gamma_3 \) is \( n-1 \). If \( k \) does not belong to \( \text{dom}(\gamma_3) \) then the overall region capability sum is \( n \); however, \( F_3 \) is in that case \( n_1 \). Thus, in both cases the overall region capability sum is \( n_1 \). We have from \( Q_1(S, \delta) \) that \( I \in \text{dom}_1(S) \land \text{rn}(I) = k \Rightarrow \alpha(I) = n_1 \); 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 \( \text{cns} \) of the first assumption). By combining \( Q_3(S, \delta_1) \) and \( Q_3(S, \delta_2) \) we have that \( Q_3(S, \delta') \).

The proof for \( \forall k \in \text{dom}(\delta'). Q_5(\delta', k, S) \land Q_6(\delta', k, S) \) is similar to the earlier case. Briefly, if the lock capability of some region \( k \) of \( \gamma_3 \) is \( \emptyset \) then \( k \) will not exist in the output effect list \( \gamma_4 \). If the lock capability of \( k \) in \( \delta(j) \) is zero then the proof for region \( k \) is immediate. Finally, if the lock capability is non-zero then the corresponding lock capability of \( \gamma_3 \) will have the same value whereas the lock capability in \( \gamma_4 \) will be zero. It is obvious that in all cases the original lock capability (i.e., \( \forall k \in \text{dom}(\delta). Q_5(\delta, k, S) \land Q_6(\delta, k, S) \)) is preserved.

Lemma 26 (Progress-Add Location) \( M; R; \delta \vdash \text{str}, S \& R; M; \emptyset; \emptyset \vdash v : \tau \& \text{live}(\delta(k), \{1\}) \Rightarrow \exists S_1, \ell(\ell, S_1) = \text{alloc}(j, S, v) \ell \not\in \text{dom}(M) \)
Proof. By observation of \( if \) judgement it suffices to show that:

- \( \text{live}(S, \{k\}) \): This is immediate by inversion of the first assumption, which gives us that \( \delta \vdash_{\text{cns}} S \). By the third assumption we can deduce that \( k \in \text{dom}(\delta) \). By combining those two facts we can deduce from \( \text{cns} \) premise \( Q_3 \) that \( \text{live}(S, \{k\}) \).

- Assume that \( I = (H, S_2) \) is a subregion of \( S \) and \( \text{rn}(I) = k \) then \( \ell \mapsto v \) can be added to \( H \). This is immediate as \( \ell \) is a fresh location. This fact implies in conjunction with the first premise that \( \ell \) does not exist in the locations of the entire store. Therefore, it can be added to \( H \).

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

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

- \( \text{live}(S, \{k\}) \): This is immediate by inversion of the first assumption, which gives us that \( \delta \vdash_{\text{cns}} S \). By the second assumption we can deduce that \( k \in \text{dom}(\delta) \). By combining those two facts we can deduce from \( \text{cns} \) premise \( Q_4 \) that \( \text{live}(S, \{k\}) \).

- \( j \) 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) \( M; R; \delta \vdash_{\text{str}} S \wedge \text{accessible}(\delta(j), \{\{\}\}) \Rightarrow \exists S', v_1, (S', v_1) \equiv \text{xupdate}_v(S, \ell, v) \)

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

- Assume \( Q_5(\delta, j, S) \) holds. This is a contradiction as the second assumption implies that thread \( j \) has a lock capability for region \( i \) but \( Q_5 \) implies that for all threads the region capability of \( i \) is zero. Hence \( Q_5 \) cannot hold.

- It is obvious from the above that \( Q_6(\delta, j, S) \) holds. Informally, \( Q_6 \) implies that there exists exactly one thread whose lock capability of region \( i \) is non-zero. The second assumption tells us that this thread is \( j \). Let \( k_2 \) be the lock capability of region \( i \) of thread \( j \). We can deduce from \( Q_6 \) that \( Q_2(k_2, i, j, S) \) holds. Lock capability \( k_2 \) can have two possible values:

  - If \( Q_2(\ell, i, j, S) \) \( k_2 = \ell \) it holds then by the definition of \( Q_2 \) \( \exists I \in \text{dom}(\ell), \text{rn}(I) = i \land \ell(I) = -1 \land \text{ln}(I) = 0 \).

  - If \( Q_2(\ell, i, j, S) \) \( k_2 = \ell \) it holds then by the definition of \( Q_2 \) \( \exists I \in \text{dom}(\ell), \text{rn}(I) = i \land \ell(I) = n \land \text{ln}(I) = j \).

- By the first and fourth assumption we have that \( \ell \) belongs to the locations of \( S \)

Therefore, all premises of \( \text{xupdate} \) are satisfied, thus \( \exists S', v_1, (S', v_1) \equiv \text{xupdate}_v(S, \ell, v) \)

Lemma 29 (Context Weakening Generalization) \( R; M; \Delta; \Gamma \vdash v : \tau \land \text{cns}(\gamma_1; \gamma_2) \Rightarrow \exists \gamma \equiv \text{newrgn}(\gamma_1, R; \Delta \vdash \gamma \land \text{output}(\gamma_1, \sigma, \gamma_1, \gamma_2) \equiv \gamma \land \text{output}(\gamma_1, \sigma, \gamma_1, \gamma_2) \equiv \gamma \land \text{output}(\gamma_1, \sigma, \gamma_1, \gamma_2) \equiv \gamma \land R; M; \Delta; \Gamma \vdash v : \tau \land \text{cns}(\gamma_1; \gamma_2) \)

Proof. By induction on the expression typing derivation.

Case \( \text{T-\text{I}}, \text{T-U}, \text{T-F}, \text{T-L}, \text{T-R}, \text{T-V}, \text{T-PRF} \): In this case the typing derivation is of the form \( R; M; \Delta; \Gamma \vdash v : \tau \land \text{cns}(\gamma_1; \gamma_2) \). The application of lemma 5 to this derivation yields \( \vdash_{\text{str}} R; M; \Delta; \Gamma; \gamma_1; \gamma_2 \). The assumptions \( R; \Delta \vdash \gamma_1 \land \gamma_2 \) and the well-formedness fact imply that \( \vdash_{\text{str}} R; M; \Delta; \Gamma; \gamma_1; \gamma_2 \). We can then re-construct the value typing as follows:

\[ R; M; \Delta; \Gamma \vdash v : \tau \land \text{cns}(\gamma_1; \gamma_2) \]

Case \( \text{T-Proj}, \text{T-RApp}, \text{T-EP} \): 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 in all cases: there exist no capability checking or modification predicates in their premises.

Case \( \text{T-App} \): We can apply the induction hypothesis on the leftmost premise \( R; M; \Delta; \Gamma \vdash v : \tau \land \text{cns}(\gamma_1; \gamma_2) \) (along with the remaining assumptions of this lemma) to obtain that \( \exists \gamma_0. R; \Delta \vdash \gamma_0 \land \text{output}(\gamma_1, \sigma, \gamma_1, \gamma_0) \equiv \gamma_0 \land \text{output}(\gamma_1, \sigma, \gamma_1, \gamma_0) \equiv \gamma_0 \land \text{output}(\gamma_1, \sigma, \gamma_1, \gamma_0) \equiv \gamma_0 \land R; M; \Delta; \Gamma \vdash v : \tau \land \text{cns}(\gamma_1; \gamma_2) \).

By performing the induction hypothesis on \( R; M; \Delta; \Gamma \vdash v : \tau \land \text{cns}(\gamma_1; \gamma_2) \) and using that \( \text{output}(\gamma_0, \sigma, \gamma_1, \gamma_2) \equiv \gamma_0, R; \Delta \vdash \gamma_0 \land \text{output}(\gamma_1, \sigma, \gamma_1, \gamma_0) \equiv \gamma_0 \land \text{output}(\gamma_1, \sigma, \gamma_1, \gamma_0) \equiv \gamma_0 \land \text{output}(\gamma_1, \sigma, \gamma_1, \gamma_0) \equiv \gamma_0 \land \text{output}(\gamma_1, \sigma, \gamma_1, \gamma_0) \equiv \gamma_0 \land R; M; \Delta; \Gamma \vdash v : \tau \land \text{cns}(\gamma_1; \gamma_2) \).

By applying lemma 30 on \( \text{output}(\gamma_0, \sigma, \gamma_1, \gamma_2) \equiv \gamma_0 \) and the \( \text{cns}(\gamma_1; \gamma_2) \) premise the application typing derivation we have that \( \gamma_0 = \text{output}(\gamma_0, \xi, \gamma_0) \). Therefore, \( R; M; \Delta; \Gamma \vdash v : \tau \land \text{cns}(\gamma_1; \gamma_2) \)

37
Lemma 30 ("Implies"-Weakening) \( \text{output}(\gamma_7, \xi, \gamma_3, \gamma_4) \equiv \gamma_7 \land \text{output}(\gamma_6, \text{seq}, \gamma_1, \gamma_2) \equiv \gamma_6 \Rightarrow \text{output}(\gamma_5, \xi, \gamma_3, \gamma_4) \equiv \gamma_6 \)

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

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 \( \text{live}(\gamma_5, \epsilon_2) \) and \( \text{live}(\gamma_6, \epsilon_2) \). Hence to prove \( \text{live}(\gamma_5, \epsilon_2) \) and \( \text{live}(\gamma_6, \epsilon_2) \) it suffices to prove that \( \text{live}(\gamma_5, \epsilon_1) \) and \( \text{live}(\gamma_6, \epsilon_1) \).

The definition of tout implies that all effects of \( \gamma_7 \) also exist in \( \gamma_4 \) and the region capability \( \kappa_2 \) of each output effect is defined as \( \kappa_2 \equiv \text{out}(\text{rg}, \psi, \kappa_3, \kappa_4, \kappa_1) \), where \( \kappa_3, \kappa_4, \) and \( \kappa_1 \) are the region capabilities of the corresponding effects of \( \gamma_3, \gamma_4 \), and \( \gamma_1 \) respectively. By the second assumption the same applies for \( \gamma_5 \) and \( \gamma_7 \).

By the definition of \( \text{out} \) it is immediate that when a region in \( \gamma_3 \) is live then the corresponding region in \( \gamma_1 \) must also be live. The same applies for regions in \( \gamma_4 \) and \( \gamma_2 \). By the second assumption we also have the same constraints for \( \gamma_1 \) and \( \gamma_7 \), and \( \gamma_3 \) and \( \gamma_6 \). We also have from the first assumption that \( \text{live}(\gamma_1, \epsilon_1) \) and \( \text{live}(\gamma_2, \epsilon_1) \). By the above three facts we can conclude that \( \text{live}(\gamma_5, \epsilon_1) \) and \( \text{live}(\gamma_6, \epsilon_1) \).

To complete the proof we need to prove the remaining premise of the conclusion, namely \( \exists \epsilon_1. (\gamma_7, \epsilon_1) \equiv \text{tout}(\text{seq}, \text{sumt}(\gamma_3), \text{sumt}(\gamma_4), \gamma_7) \) (by the first assumption) and \( \exists \epsilon_2. (\gamma_6, \epsilon_2) \equiv \text{tout}(\text{seq}, \text{sumt}(\gamma_1), \text{sumt}(\gamma_2), \gamma_5) \) (by the second assumption) then \( \exists \epsilon_3. (\gamma_6, \epsilon_3) \equiv \text{tout}(\text{seq}, \text{sumt}(\gamma_3), \text{sumt}(\gamma_4), \gamma_5) \).

Intuitively, all region and lock capabilities modified from the transition of \( \gamma_5 \) to \( \gamma_6 \) are exactly the ones modified from the transition of \( \gamma_7 \) to \( \gamma_5 \) as a result of the transition of \( \gamma_3 \) to \( \gamma_5 \). Therefore, we can substitute \( \gamma_3 \) and \( \gamma_5 \) for \( \gamma_7 \) and \( \gamma_5 \) in \( \exists \epsilon_2. (\gamma_6, \epsilon_2) \equiv \text{tout}(\text{seq}, \text{sumt}(\gamma_1), \text{sumt}(\gamma_2), \gamma_5) \) and obtain that the resulting output predicate holds. Of course, the resulting \( \epsilon \) in the exists clause will be slightly larger but we have proven above that the live predicate will still hold.

Lemma 31 (Live Region-Weakening) \( \text{output}(\gamma_7, \text{seq}, \gamma_{1b}, \gamma_3) \equiv \gamma_5 \land \text{live}(\gamma_{1b}, \{r\}) \Rightarrow \text{live}(\gamma_7, \{r\}) \)

Proof. The second assumption implies that \( \forall r \in \{r\} \cup \text{parents}(\gamma_{1b}, r).r^{\kappa_1, \kappa_2} \not\equiv b \in \text{linear}(\gamma_{1b}) \Rightarrow \kappa_1 \not\equiv 0 \land \kappa_1 \not\equiv 0 \).

\( \text{output}(\gamma_7, \text{seq}, \gamma_{1b}, \gamma_3) \equiv \gamma_5 \). It suffices to show that \( \forall r \in \{r\} \cup \text{parents}(\gamma_7, r).r^{\kappa_3, \kappa_4} \not\equiv b \in \text{linear}(\gamma_7) \Rightarrow \kappa_5 \not\equiv 0 \land \kappa_5 \not\equiv 0 \).

The first assumption implies that \( \exists \epsilon_3. (\gamma_3, \epsilon) \equiv \text{tout}(\text{seq}, \text{sumt}(\gamma_{1b}), \text{sumt}(\gamma_3), \gamma_7) \). In turn the definition of tout implies that all effects of \( \gamma_{1b} \) also exist in \( \gamma_7 \) and the region capability \( \kappa_7 \) of each output effect is defined as \( \kappa_7 \equiv \text{out}(\text{rg}, \psi, \kappa_3, \kappa_4, \kappa_5) \), where \( \kappa_3, \kappa_4, \) and \( \kappa_5 \) are the region capabilities of the corresponding effects of \( \gamma_{1b}, \gamma_3 \), and \( \gamma_7 \) respectively. By observation of our relation we have that \( \kappa_1 \not\equiv 0 \land \kappa_1 \not\equiv 0 \) implies \( \kappa_7 \not\equiv 0 \land \kappa_7 \not\equiv 0 \). Therefore, \( \forall r \in \{r\} \cup \text{parents}(\gamma_7, r).r^{\kappa_3, \kappa_4} \not\equiv b \in \text{linear}(\gamma_7) \Rightarrow \kappa_5 \not\equiv 0 \land \kappa_5 \not\equiv 0 \) holds.

Lemma 32 (Accessible Region-Weakening) \( \text{output}(\gamma_7, \text{seq}, \gamma_{1b}, \gamma_3) \equiv \gamma_5 \land \text{accessible}(\gamma_{1b}, \{r\}) \Rightarrow \text{accessible}(\gamma_7, \{r\}) \)

Proof. Similar reasoning to lemma 31.

Lemma 33 (Capability Modification-Weakening) \( \text{output}(\gamma_7, \text{seq}, \gamma_{1b}, \gamma_3) \equiv \gamma_5 \land \text{modcap}(\gamma_{1b}, \psi, \eta, r) \Rightarrow \text{modcap}(\gamma_7, \psi, \eta, r) \)

Proof. By case analysis on \( \psi \).

Case \( \psi \equiv \text{rg} \) then the second assumption implies that \( x \equiv r^{\kappa_1, \kappa_2} \not\equiv b \in \chi \times \chi \times \chi \land \chi \times \chi \equiv \text{mod}(\kappa_1) \land \gamma_2 \equiv \gamma_1[r^{\kappa_1, \kappa_2} \not\equiv b \in \chi \times \chi ] \land \text{inv}(\gamma_2) \land \text{live}(\gamma_{1b}, \{r\}) \land \kappa_1 \equiv 0 \Rightarrow \kappa_2 \equiv 0 \land \kappa_1 \equiv 0 \Rightarrow \kappa_2 \equiv 0 \). It suffices to prove that \( x \equiv r^{\kappa_3, \kappa_4} \not\equiv b \in \chi \times \chi \times \chi \equiv \text{mod}(\kappa_3) \land \gamma_3 \equiv \gamma_1[r^{\kappa_3, \kappa_4} \not\equiv b \in \chi \times \chi ] \land \text{inv}(\gamma_3) \land \text{live}(\gamma_7, \{r\}) \land \kappa_5 \equiv 0 \Rightarrow \kappa_6 \equiv 0 \land \kappa_5 \equiv 0 \Rightarrow \kappa_6 \equiv 0 \).
The first assumption implies that $\exists \varepsilon. (\gamma_5, \varepsilon) \equiv \text{tout}(\text{seq}, \sumt(\gamma_4), \sumt(\gamma_2), \gamma_7)$. In turn the definition of tout implies that all effects of $\gamma_{16}$ also exist in $\gamma_7$ and the region capability $\kappa_7$ of each output effect is defined as $\kappa_7 \equiv \text{out}(\text{rg}, \psi, \kappa_1, \kappa_3, \kappa_5)$, where $\kappa_1$, $\kappa_3$, and $\kappa_5$ are the region capabilities of the corresponding effects of $\gamma_{16}, \gamma_3$, and $\gamma_7$ respectively.

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

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