A Concurrent Language with a Uniform Treatment of Regions and Locks

Prodromos Gerakios Nikolaos Papaspyrou Konstantinos Sagonas

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

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

Abstract

A challenge for programming language research is to design and implement multi-threaded low-level languages providing static guarantees for memory safety and freedom from data races. Towards this goal, we present a concurrent language employing safe region-based memory management and hierarchical locking of regions. Both regions and locks are treated uniformly, and the language supports ownership transfer, early deallocation of regions and early release of locks in a safe manner.

1 Introduction

Writing safe and robust code is a hard task; writing safe and robust multi-threaded low-level code is even harder. In this paper we present a minimal, low-level concurrent language with advanced region-based memory management and hierarchical lock-based synchronization primitives.

Region-based memory management achieves efficiency by bulk allocation and deallocation of objects in segments of memory called regions. Similar to other approaches, our regions are organized in a hierarchical manner such that each region is physically allocated within a single parent region and may contain multiple child regions. This hierarchical structure imposes an ownership relation as well as lifetime constraints over regions. Unlike other languages employing hierarchical regions, our language allows early subtree deallocation in the presence of region sharing between threads. In addition, each thread is obliged to release each region it owns by the end of its scope.

Multi-threaded programs that interact through shared memory generate random execution interleavings. A data race occurs in a multi-threaded program when there exists an interleaving such that some thread accesses a memory location while some other thread attempts to write to it. So far, type systems and analyses that guarantee race freedom Flanagan and Abadi [1999] have mainly focused on lexically-scoped constructs. The key idea in those systems is to statically track or infer the lockset held at each program point. In the language presented in this paper, implicit reentrant locks are used to protect regions from data races. Our locking primitives are non-lexically scoped. Locks also follow the hierarchical structure of regions so that each region is protected by its own lock as well as the locks of all its ancestors.

Furthermore, our language allows regions and locks to be safely aliased, escape the lexical scope when passed to a new thread, or become logically separated from the remaining hierarchy. These features are invaluable for expressing numerous idioms of multi-threaded programming such as *sharing*, *region ownership* or *lock ownership transfers*, *thread-local regions* and *region migration*.

2 Language Design

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

Low-level and concurrent. Our language must efficiently support systems programming. As such, it should cater for memory management and concurrency. It also needs to be low-level: it is not intended to be used by programmers but as a target language of higher-level systems programming languages.

Static safety guarantees. We define safety in terms of *memory safety* and absence of *data races*. A static type system should guarantee that well-typed programs are safe, with minimal run-time overhead.

Safe region-based memory management. Similarly to other languages for safe systems programming (e.g. Cyclone) our language employs region-based memory management, which achieves efficiency by *bulk allocation* and *deallocation* of objects in segments of memory (*regions*). Statically typed regions Tofte and Talpin [1994], Walker et al. [2000] guarantee the absence of dangling pointer dereferences, multiple release operations of the same memory area, and memory leaks. Traditional stack-based regions Tofte and Talpin [1994] are limiting as they cannot be deallocated early. Furthermore, the stack-based discipline fails to model region lifetimes in concurrent languages, where the lifetime of a shared region depends on the lifetime of the longest-lived thread accessing that region. In contrast, we want regions that can be *deallocated early* and that can safely be *shared* between concurrent threads.

We opt for a *hierarchical region* Gay and Aiken [2001] organization: each region is physically allocated within a single parent region and may contain multiple child regions. Early region deallocation in our multi-level hierarchy automatically deallocates the immediate subtree of a region without having to deallocate each region of the subtree recursively. The hierarchical region structure imposes the constraint that a child region is *live* only when its ancestors are live. In order to allow a function to access a region without having to pass all its ancestors explicitly, we allow ancestors to be abstracted (i.e., our language supports *hierarchy abstraction*) for the duration of the function call. To maintain the *liveness* invariant we require that the abstracted parents are *live* before and after the call. Regions whose parent information has been abstracted cannot be passed to a new thread as this may be unsound.

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

Region polymorphism and aliasing. Our language supports *region polymorphism*: it is possible to pass regions as parameters to functions or concurrent threads. This enables *region aliasing*: one actual region could be passed in the place of two distinct formal region parameters. In the presence of mutual exclusion and early region deallocation, aliasing is dangerous. Our language allows safe region aliasing with minimal restrictions. The mechanism that we employ for this purpose also allows us to encode numerous useful idioms of concurrent programming, such as *region migration*, *lock ownership transfers*, *region sharing*, and *thread-local regions*.

3 Language Features through Examples

Our regions are lexically-scoped first-class citizens; they are manipulated via explicit handles. For instance, a region handle can be used for releasing a region early, for allocating references and regions within it, or for locking it. Our language uses a *type and effect system* to guarantee that regions and their contents are properly used. The details will be made clear in Sections 4 and 6. Here, we present the main features of our language through examples. We try to avoid technical issues as much as possible; however, some characteristics of the type and effect system are revealed in this section and their presence is justified. Furthermore, to simplify the presentation in this section, we use abbreviations for a few language constructs that we expect the readers will find more intuitive.

Example 1 (Simple Region Usage) This example shows a typical region use. New regions are allocated via the `newrgn` construct. This construct requires a handle to an existing region (*heap* in this case), in which the new region will be allocated, and introduces a type-level name (ρ) and a fresh handle (h) for the new region. The handle h is then used to allocate a new integer in region ρ; a reference to this integer (z) is created. Finally, the region is deallocated before the end of its lexical scope.

```
newrgn $\rho, h$ at $heap$ in              // $\{\rho^{1,1} \triangleright \rho_H\}$
    let $z$ = new 10 at $h$ in
        ...
        $z$ := deref $z + 5$ ;
        ...
        free $h$ ;                          // $\{\}$    — empty effect, $\rho$ is no longer alive
        ...
```

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

Example 2 (Hierarchical Regions) In the previous example a trivial hierarchy was created by allocating region ρ within the *heap* region. It is possible to construct richer region hierarchies. As in the previous example, the code below allocates a new region ρ_1 within the heap. Other regions can be then allocated within ρ_1, e.g. ρ_2; this can done by passing the handle of ρ_1 to the region creation construct. Similarly, regions ρ_3 and ρ_4 can be allocated within region ρ_2.

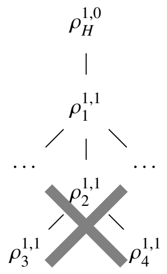
```
newrgn $\rho_1, h$ at $heap$ in          // $\{\rho_1^{1,1} \triangleright \rho_H\}$
    ...
    newrgn $\rho_2, h_2$ at $h_1$ in      // $\{\rho_1^{1,1} \triangleright \rho_H,\ \rho_2^{1,1} \triangleright \rho_1\}$
        ...
        newrgn $\rho_3, h_3$ at $h_2$ in  // $\{\rho_1^{1,1} \triangleright \rho_H,\ \rho_2^{1,1} \triangleright \rho_1,\ \rho_3^{1,1} \triangleright \rho_2\}$
            newrgn $\rho_4, h_4$ at $h_2$ in  // $\{\rho_1^{1,1} \triangleright \rho_H,\ \rho_2^{1,1} \triangleright \rho_1,\ \rho_3^{1,1} \triangleright \rho_2,\ \rho_4^{1,1} \triangleright \rho_2\}$
            ...
```

$$\begin{tikzcd}
 & \rho_H^{1,0} \arrow[d, dash] & \\
 & \rho_1^{1,1} \arrow[dl, dash] \arrow[d, dash] \arrow[dr, dash] & \\
\cdots & \rho_2^{1,1} \arrow[dl, dash] \arrow[dr, dash] & \cdots \\
\rho_3^{1,1} & & \rho_4^{1,1}
\end{tikzcd}$$

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

Example 3 (Bulk Region Deallocation) In the first example a single region was deallocated. That region was a *leaf* node in the hierarchy; it contained no sub-regions. In the general case, when a region is deallocated, the entire subtree below that region is also deallocated. This is what happens if, in the code of the previous example, we deallocate region ρ_2 within the innermost scope; regions ρ_3 and ρ_4 are also deallocated. They are all removed from the current effect and thus are no longer accessible.

```
newrgn ρ1, h at heap in              // $\{\rho_1^{1,1} \triangleright \rho_H\}$
  ...
  newrgn ρ2, h2 at h1 in             // $\{\rho_1^{1,1} \triangleright \rho_H,\ \rho_2^{1,1} \triangleright \rho_1\}$
    ...
    newrgn ρ3, h3 at h2 in           // $\{\rho_1^{1,1} \triangleright \rho_H,\ \rho_2^{1,1} \triangleright \rho_1,\ \rho_3^{1,1} \triangleright \rho_2\}$
      newrgn ρ4, h4 at h2 in         // $\{\rho_1^{1,1} \triangleright \rho_H,\ \rho_2^{1,1} \triangleright \rho_1,\ \rho_3^{1,1} \triangleright \rho_2,\ \rho_4^{1,1} \triangleright \rho_2\}$
        ...
        free h2;                     // $\{\rho_1^{1,1} \triangleright \rho_H\}$
        ...                          // ρ2, ρ3 and ρ4 are no longer alive
```



Example 4 (Region Migration) A common multi-threaded programming idiom is to use *thread-local* data. At any time, only one thread will have access to such data and therefore no locking is required. A thread can transfer thread-local data to another thread but, doing so, it loses access to the data. This idiom is known as *migration*. Our language encodes thread-local data and data migration. As we have seen, newly created regions are considered thread-local; a capability for them is added to the current effect. We support data migration by allowing such capabilities to be transferred to other threads.

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

```
def server = Λρ_H. λheap.
  while (true) do
    newrgn ρ, h at heap in           // $\{\rho^{1,1} \triangleright \rho_H\}$
      let z = wait_data[ρ](h) in     // region ρ is thread-local
        process(z);
        spawn output[ρ](h, z);       // { }   — empty effect, ρ migrates to output
        ...                          // ρ cannot be accessed here
```

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

Example 5 (Region Sharing) In the previous examples, capabilities for all regions were "1, 1" which, as we roughly explained, corresponds to thread-local. In general, a capability for a region consists of two natural numbers; the first denotes the *region count*, whereas the second denotes the *lock count*. When the region count is positive, the region is definitely alive. Similarly, when the lock count is positive, memory accesses to this region's contents are guaranteed to be race free. Capabilities with counts other than 1 can be used for *sharing* regions between threads.

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

```
def server = $\Lambda\rho_H.\ \lambda heap.$
    while (true) do
        newrgn $\rho, h$ at $heap$ in                    // $\{\rho^{1,1} \triangleright \rho_H\}$
            let $z$ = wait_data[$\rho$]($h$) in
                share $h$; unlock $h$;                  // $\{\rho^{2,0} \triangleright \rho_H\}$
                spawn output[$\rho$]($h, z$);           // $\{\rho^{1,0} \triangleright \rho_H\}$    — output consumes $\rho^{1,0} \triangleright \rho_H$
                while (!$finished$) do
                    lock $h$;                           // $\{\rho^{1,1} \triangleright \rho_H\}$
                    process($z$);
                    unlock $h$                          // $\{\rho^{1,0} \triangleright \rho_H\}$
```

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

Example 6 (Hierarchical Locking) In the previous example, locking and unlocking was performed on a leaf region. In general, locking a region in the hierarchy has the effect of atomically locking its subregions as well. A region is accessible when it has been locked by the current thread or when at least one of its ancestors has been locked.

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

```
lock $h$;                                        // the handle of a common ancestor of $\rho_1$ and $\rho_2$
    let $obj$ = hash_remove[$\rho_1$]($tbl_1, key$) in
        hash_insert[$\rho_2$]($tbl_2, key, obj$);
unlock $h$
```

Example 7 (Region Aliasing) An expressive language with regions will have to support region polymorphism, which invariably leads to *region aliasing*. This must be handled with caution, as a naïve approach may cause unsoundness. In the examples that follow, we discuss how region aliasing is used in our language as well as the restrictions that we impose to guarantee safety.

Function swap accepts two integer references, residing in regions ρ_1 and ρ_2, and swaps their contents. It assumes that both regions are already locked and remain locked when the function returns.

```
def swap = $\Lambda\rho_1.\ \Lambda\rho_2.\ \lambda(x : \texttt{ref}(\rho_1, \texttt{int}),\ y : \texttt{ref}(\rho_2, \texttt{int}))$.// $\rho_1$ and $\rho_2$ must be both locked
    let $z$ = deref $x$ in                                   // OK: $\rho_1$ is locked
        $x$ := deref $y$;                                    // OK: $\rho_1$ and $\rho_2$ are locked
        $y$ := $z$                                           // OK: $\rho_2$ is locked
```

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

```
...                                        // $\{\rho^{2,0} \triangleright \rho_H\}$
lock $h$; lock $h$;                        // $\{\rho^{2,2} \triangleright \rho_H\}$
swap[$\rho$][$\rho$]($a, b$);              // each $\rho$ parameter requires $\rho^{1,1} \triangleright \rho_H$
unlock $h$; unlock $h$                     // $\{\rho^{2,0} \triangleright \rho_H\}$
```

Example 8 (Reentrant locks) Region aliasing introduces the need for reentrant locks. To see this, let us change the swapping function of the previous example, so that it receives two references in unlocked regions. For swapping their contents, it will have to acquire locks for the two regions (and release them, when they are no longer needed).

```
def swap = $\Lambda\rho_1.\ \Lambda\rho_2.\ \lambda(h_1 : $rgn$(\rho_1),\ h_2 :$ rgn$(\rho_2).$
                   $x :$ ref$(\rho_1,$ int$),\ y :$ ref$(\rho_2,$ int$)).$// $\rho_1$ and $\rho_2$ are unlocked
    lock $h_1$;
    let $z$ = deref $x$ in                 // OK: $\rho_1$ is locked
        lock $h_2$;
        $x$ := deref $y$;                  // OK: $\rho_1$ and $\rho_2$ are locked
        unlock $h_1$;
        $y$ := $z$;                        // OK: $\rho_2$ is locked
        unlock $h_2$                       // all locks can be released
```

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

```
...                                        // $\{\rho^{2,0} \triangleright \rho_H\}$
swap[$\rho$][$\rho$]($h, h, a, b$);        // each $\rho$ parameter requires $\rho^{1,0} \triangleright \rho_H$
```

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

Example 9 (Pure and Impure Capabilities) Unrestricted region aliasing leads to unsoundness. Consider function bad, which accepts two integer references (x and y) in regions ρ_1 and ρ_2, which are both locked. It lets ρ_1 migrate to a new thread and passes x as a parameter. It then assigns a value to y.

```
def bad = $\Lambda\rho_1.\ \Lambda\rho_2.\ \lambda(x :$ ref$(\rho_1,$ int$),\ y :$ ref$(\rho_2,$ int$)).$ // $\rho_1$ and $\rho_2$ must be both locked
    spawn f[$\rho_1$]($x$);                // $\rho_1$ migrates to f while locked
    $y$ := 7                               // OK: $\rho_2$ is still locked — WRONG!
```

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

```
swap[$\rho$][$\rho$]($a, a$);              // each $\rho$ parameter requires $\rho^{1,1} \triangleright \rho_H$
```

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

$$\begin{array}{llll}
\textbf{Value} & v & ::= & f \mid c \mid \texttt{rgn}_\iota \mid \texttt{loc}_l \\
\textbf{Function} & f & ::= & \lambda x.\, e\ \mathsf{as}\ \tau \xrightarrow{\gamma\to\gamma} \tau \mid \Lambda\rho.\, f \\
\textbf{Expression} & e & ::= & x \mid c \mid f \mid (e\ e)^{\xi} \mid e\,[r] \mid \texttt{new}\ e\ \texttt{at}\ e\epsilon \mid e := e \mid \texttt{loc}_l \\
 & & \mid & \texttt{deref}\ e \mid \texttt{newrgn}\ \rho, x\ \texttt{at}\ e\ \texttt{in}\ e \mid \texttt{cap}_\eta\ e \mid \texttt{rgn}_\iota \\
\textbf{Type} & \tau & ::= & b \mid \tau \xrightarrow{\gamma\to\gamma} \tau \mid \forall\rho.\, \tau \mid \texttt{ref}(\tau, r) \mid \texttt{rgn}(r) \\
\textbf{Effect} & \gamma & ::= & \emptyset \mid \gamma, r^{\kappa} \triangleright \pi
\end{array}
\qquad
\begin{array}{llll}
\textbf{Calling mode} & \xi & ::= & \mathsf{seq} \mid \mathsf{par}(\gamma) \\
\textbf{Capability op} & \eta & ::= & \psi+ \mid \psi- \\
\textbf{Capability kind} & \psi & ::= & \mathsf{rg} \mid \mathsf{lk} \\
\textbf{Capability} & \kappa & ::= & n, n \mid \overline{n, n} \\
\textbf{Region parent} & \pi & ::= & r \mid \perp \mid ? \\
\textbf{Region} & r & ::= & \rho \mid \iota
\end{array}$$

Figure 1: Syntax.

4 Language Description

The syntax of the language is illustrated in Figure 1. The language core comprises of variables (x), constants (c), functions, and function application. Functions can be region polymorphic ($\Lambda\rho.\, f$) and region application is explicit ($e[\rho]$). Monomorphic functions ($\lambda x.\, e$) must be annotated with their type. The application of monomorphic functions is annotated with a *calling mode* (ξ), which is seq for normal (sequential) application and par(γ) for spawning a new thread.[1] Parallel application is annotated with the input effect of the new thread (γ); this annotation can be automatically inferred by the type checker. The constructs for manipulating references are standard. A newly allocated memory cell is returned by `new` e_1 `at` e_2, where e_1 is the value that will be placed in the cell and e_2 is a handle of the region in which the new cell will be allocated. Standard assignment and dereference operators complete the picture.

The construct `newrgn` ρ, x `at` e_1 `in` e_2 allocates a new region ρ and binds x to the region *handle*. The new region resides in a *parent* region, whose handle is given in e_1. The scope of ρ and x is e_2, which must consume the new region by the end of its execution. A region can be consumed either by deallocation or by transferring its ownership to another thread. At any given program point, each region is associated with a *capability* (κ). Capabilities consist of two natural numbers, the *capability counts*: the *region* count and *lock* count, which denote whether a region is live and locked respectively. When first allocated, a region starts with capability $(1, 1)$, meaning that it is live and locked, so that it can be exclusively accessed by the thread that allocated it. As we have seen, this is our equivalent of a thread-local region.

By using the construct $\texttt{cap}_\eta\ e$, a thread can *increment* or *decrement* the capability counts of the region whose handle is specified in e. The capability operator η can be, e.g., rg+ (meaning that the region count is to be incremented) or lk− (meaning that the lock count is to be decremented).[2] When a region count reaches zero, the region may be physically deallocated and no subsequent operations can be performed on it. When a lock count reaches zero, the region is unlocked. As we explained, capability counts determine the validity of operations on regions and references. All memory-related operations require that the involved regions are live, i.e., the region count is greater than zero. Assignment and dereference can be performed only when the corresponding region is live and locked.

A capability of the form (n_1, n_2) is called a *pure* capability, whereas a capability of the form $(\overline{n_1, n_2})$ is called an *impure* capability. In both cases, it is implied that the current thread can decrement the region count n_1 times and the lock count n_2 times. Impure capabilities are obtained by splitting pure or other impure capabilities into several pieces, e.g., $(3, 2) = (\overline{2, 1}) + (\overline{1, 1})$, in the same spirit as *fractional capabilities* Boyland [2003]. As we explained in Example 9 of Section 3, these pieces are useful for region aliasing, when the same region is to be passed to a function in the place of two distinct region parameters. An impure capability implies that our knowledge of the region and lock count is inexact. The use of such capabilities must be restricted; e.g., an impure capability with a non-zero lock count cannot be passed to another thread, as it is unsound to allow two threads to simultaneously hold the same lock. Capability splitting takes place automatically with function application.

[1] In the examples of Section 3, we used more intuitive notation: we omitted seq and used the keyword spawn instead of par.

[2] The region manipulation operators used in Section 3 are simple abbreviations: `share` $\equiv \texttt{cap}_{\mathsf{rg}+}$, `unlock` $\equiv \texttt{cap}_{\mathsf{lk}-}$, etc.

$$\begin{array}{llll}
\textbf{Configuration} & C & ::= & S\,;T \\
\textbf{Threads} & T & ::= & \emptyset \mid T, n : e \\
\textbf{Store} & S & ::= & \emptyset \mid S, \iota : (\theta, H, S) \\
\textbf{Thread map} & \theta & ::= & \emptyset \mid \theta, n_1 \mapsto n_2, n_3 \\
\textbf{Memory heap} & H & ::= & \emptyset \mid H, \ell \mapsto v
\end{array}$$

$$\begin{array}{lll}
E & ::= & \Box \mid (E\ e)^{\xi} \mid (v\ E)^{\xi} \mid E\,[r] \\
 & \mid & \texttt{newrgn}\,\rho, x\ \texttt{at}\ E\ \texttt{in}\ e \mid \texttt{cap}_{\eta}\ E \\
 & \mid & \texttt{new}\ E\ \texttt{at}\ e\epsilon \mid \texttt{new}\ v\ \texttt{at}\ E\epsilon \\
 & \mid & \texttt{deref}\ E \mid E := e \mid v := E
\end{array}$$

Figure 2: Configurations, store, threads and evaluation contexts.

$$\frac{\begin{array}{cc} e' \equiv ((\lambda x.\, e\ \texttt{as}\ \tau)\ v)^{\mathsf{par}(\gamma_1)} & e'' \equiv ((\lambda x.\, e\ \texttt{as}\ \tau)\ v)^{\mathsf{seq}} \\ \text{fresh}\ n' & S' = \text{transfer}(S, n, n', \gamma_1) \end{array}}{S\,;T, n : E[e'] \rightsquigarrow S'; T, n : E[()], n' : e''}\ \textit{(E-SN)}$$

$$\frac{S\,;e \rightarrow_n S'; e'}{S\,;T, n : E[e] \rightsquigarrow S'; T, n : E[e']}\ \textit{(E-S)} \qquad \frac{}{S\,;T, n : () \rightsquigarrow S\,;T}\ \textit{(E-T)}$$

Figure 3: Thread evaluation relation $C \rightsquigarrow C'$.

5 Operational Semantics

We define a *small-step* operational semantics for our language, using two evaluation relations, at the level of *threads* and *expressions* (Figures 3 and 4 on the next page). The thread evaluation relation transforms *configurations*. A configuration C (see Figure 2) consists of an abstract *store* S and a list of threads T.[3] 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 $\iota : (\theta, H, S)$, where ι is a *region identifier*, θ is a thread map, H is a memory heap and S is the list of subregions in the region hierarchy. The thread map associates thread identifiers with capability counts for region ι, whereas the memory heap represents the region's contents, mapping locations to values.

A *thread evaluation context* E (Figure 2) is defined as an expression with a *hole*, represented as $\Box$. The hole indicates the position where the next reduction step can take place. Our notion of evaluation context imposes a call-by-value evaluation strategy to our language. Subexpressions are evaluated in a left-to-right order.

We assume that concurrent reduction events can be totally ordered. At each step, a random thread (n) is chosen from the thread list for evaluation (Figure 3). It should be noted that the thread evaluation rules are the only *non-deterministic* rules in the operational semantics of our language; in the presence of more than one active 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*). Rule *E-S* reduces some thread n via the expression evaluation relation. When a parallel function application redex is detected within the evaluation context of a thread, a new thread is created (rule *E-SN*). The redex is replaced with a unit value in the currently executed thread and a new thread is added to the thread list, with a *fresh* thread identifier. The *partial* function transfer(S, n, n', γ_1) updates the thread maps of all regions specified in γ_1, transferring capabilities between threads n and n'. It is undefined when this transfer is not possible.

The expression evaluation relation is defined in Figure 4. The rules for reducing function application (*E-A*) and region application (*E-RP*) are standard. 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 is *live*. A region is *live* when the sum of all region counts in the thread map associated with that region is positive and all ancestors of the region are *live* as well. In addition to liveness, some of these functions require that some region is *accessible* to the currently executed thread. Region r is accessible to some thread n (and *inaccessible* to all other threads) when r is live and the

[3]The order of elements in comma-separated lists, e.g. in a store S or in a list of threads T, is not important; we consider all list permutations as equivalent.

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

$$\frac{(S',k) = \mathrm{newrgn}(S,n,\texttt{j})}{S\,;\texttt{newrgn}\,\rho,x\ \texttt{at}\ \texttt{rgn}_{\texttt{j}}\ \texttt{in}\ e \to_n S';e[k/\rho][\texttt{rgn}_k/x]}\ \textit{(E-NG)} \qquad \frac{S' = \mathrm{updcap}(S,\eta,\texttt{j},n)}{S\,;\texttt{cap}_\eta\ \texttt{rgn}_{\texttt{j}} \to_n S';()}\ \textit{(E-C)}$$

$$\frac{(S',\ell) = \mathrm{alloc}(\texttt{j},S,v)}{S\,;\texttt{new}\ v\ \texttt{at}\ \texttt{rgn}_{\texttt{j}}\epsilon \to_n S';\texttt{loc}_\ell}\ \textit{(E-NR)} \qquad \frac{S' = \mathrm{update}(S,\ell,v,n)}{S\,;\texttt{loc}_\ell := v \to_n S';()}\ \textit{(E-AS)} \qquad \frac{v = \mathrm{lookup}(S,\ell,n)}{S\,;\texttt{deref}\ \texttt{loc}_\ell \to_n S\,;v}\ \textit{(E-D)}$$

Figure 4: Expression evaluation relation $S\,;e \to_n S';e'$.

thread map associated with r, or with some ancestor of r, maps n to a positive lock count.

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

- lookup(S, ℓ, n) is used in rule *E-D* to look up the value of location ℓ in S. The region in which ℓ resides must be accessible to the currently executed thread n. Upon success, the function returns the value v stored at ℓ.

- update(S, ℓ, v, n) is used in rule *E-AS* to assign the value v to location ℓ in S. The region in which ℓ resides must be accessible to the currently executed thread n. Upon success, the function returns the new store S'.

- newrgn($S, n, \texttt{j}$) is used in rule *E-NR* to create a new region in S. The new region is allocated within $\texttt{j}$, which must be live. Its thread map is set to $n \mapsto 1, 1$. Upon success, the function returns a pair (S', k) containing the new store and a fresh region name for the new region.

- updcap($S, \eta, \texttt{j}, n$) is used in rule *E-C*. This operation updates S by modifying the region or lock count of thread n for region $\texttt{j}$. Upon success, the function returns the new store S'. When a lock update is requested and the lock is held by another thread, the result is undefined. In this case, rule *E-C* cannot be applied and the operation will block, until the lock is available.

The operational semantics may get stuck when a *deadlock* occurs. Our semantics will also get stuck when a thread attempts to access a memory location without having acquired an appropriate lock for this location. In this case, update(S, ℓ, v, n) and lookup(S, ℓ, n) are undefined and it is impossible to perform a single step via rules *E-AS* or *E-D*. The same is true in several other situations (e.g. when referring to a non-existent region or location). Threads that may cause a data race will definitely get stuck.

We follow a different approach from related work, e.g. the work of Grossman Grossman [2003], where a special kind of value $junk_v$ is often used as an intermediate step when assigning a value v to a location, before the real assignment takes place, and type safety guarantees that no junk values are ever used. As described above, we use a more direct approach by incorporating the locking mechanism in the operational semantics. Our progress lemma in Section 7 guarantees that, at any time, *all* threads can make progress and, therefore, a possible implementation does not need to check liveness or accessibility at run-time.

6 Static Semantics

In this section we discuss the most interesting parts of our type system. As we sketched in Section 3, to enforce our safety invariants, we use a *type and effect system*. Effects are used to statically track region capabilities. An effect (γ) is a list of elements of the form $r^\kappa \triangleright \pi$, denoting that region r is associated with capability κ and has parent π, which can be another region, $\perp$, or ?. Regions whose parents are $\perp$ or ?

$$\frac{R;\Delta \vdash \tau \qquad \tau \equiv \tau_1 \xrightarrow{\gamma_1 \to \gamma_2} \tau_2 \qquad R;M;\Delta;\Gamma, x:\tau_1 \vdash e : \tau_2 \,\&\, (\gamma_1;\gamma_2)}{R;M;\Delta;\Gamma \vdash \lambda x.\, e \texttt{ as } \tau : \tau \,\&\, (\gamma;\gamma)}\ \textit{(T-F)}$$

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

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

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

$$\frac{R;M;\Delta;\Gamma \vdash e_1 : \tau \,\&\, (\gamma;\gamma') \qquad R;M;\Delta;\Gamma \vdash e_2 : \texttt{rgn}(r) \,\&\, (\gamma';\gamma'') \qquad r \in \textit{dom}(\gamma'')}{R;M;\Delta;\Gamma \vdash \texttt{new } e_1 \texttt{ at } e_2 \epsilon : \texttt{ref}(\tau,\rho) \,\&\, (\gamma;\gamma'')}\ \textit{(T-NR)}$$

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

Figure 5: Selected typing rules.

$$\frac{\xi \vdash \gamma = \gamma_1 \oplus \gamma_r \qquad \xi \vdash \gamma' = \gamma_2 \oplus \gamma_r \qquad \gamma'' = \textit{live}(\gamma') \qquad \textit{consistent}(\gamma;\gamma'') \qquad \xi = \mathsf{seq} \Rightarrow \textit{abs\_par}(\gamma;\gamma_1) \subseteq \textit{dom}(\gamma'') \qquad \xi = \mathsf{par}(\gamma''') \Rightarrow \gamma_1 = \gamma''' \wedge \gamma_2 = \emptyset}{\xi \vdash \gamma'' = \gamma_2 \oplus (\gamma \ominus \gamma_1)}\ \textit{(ESJ)}$$

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

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

Figure 6: Effect and capability splitting.

are considered as roots in our region hierarchy. We assume that there is an initial (physical) root region corresponding to the entire heap, whose handle is available to the main program. The parent of the heap region is $\perp$. More (logical) root regions can be created using hierarchy abstraction. The abstract parent of a region that is passed to a function is denoted by ?.

The syntax of types in Figure 1 (on page 7) is more or less standard. A collection of base types b is assumed; the syntax of values belonging to these types and operations upon such values are omitted from this paper. We assume the existence of a *unit* base type, which we denote by $\langle\rangle$. Region handle types $\texttt{rgn}(r)$ and reference types $\texttt{ref}(\tau, r)$ are associated with a type-level region r. Monomorphic function types carry an *input* and an *output effect*. A well-typed expression e has a type τ under an input effect γ and results in an output effect γ'. The typing relation (see Figure 5) is denoted by $R;M;\Delta;\Gamma \vdash e : \tau \,\&\, (\gamma;\gamma')$ and uses four typing contexts: a set of region literals (R), a mapping of locations to types (M), a set of region variables (Δ), and a mapping of term variables to types (Γ). The effects that appear in our typing relation must satisfy a *liveness invariant*: all regions that appear in the effect are *live*, i.e., their region counts and those of all their ancestors are positive. Thus, in order to check if a region r is live in the effect γ, we only need to check that $r \in \textit{dom}(\gamma)$.

The typing rule for lambda abstraction (*T-F*) requires that the body e is well-typed with respect to the effects ascribed on its type. The typing rule for function application (*T-AP*) splits the output effect of e_2 (γ'') by subtracting the function's input effect (γ_1). It then joins the remaining effect with the function's output effect (γ_2). In the case of parallel application, rule *T-AP* also requires that the return type is unit. The splitting and joining of effects is controlled by the judgement $\xi \vdash \gamma'' = \gamma_2 \oplus (\gamma \ominus \gamma_1)$, which is defined in Figure 6 (the auxiliary functions and predicates are defined in Figures 7 and 8). It enforces the following properties:

- the liveness invariant for γ'';
- the consistency of γ and γ'', i.e., regions cannot change parent and capabilities cannot switch from pure to impure or vice versa; the domain of γ'' is a subset of the domain of γ;

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

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

Figure 7: Auxiliary predicates: region liveness and accessibility.

$$\begin{array}{lcl}
\mathit{rg}(\kappa) & = & n_1 \quad \text{if } \kappa = n_1,n_2 \vee \kappa = \overline{n_1,n_2} \\
\mathit{lk}(\kappa) & = & n_2 \quad \text{if } \kappa = n_1,n_2 \vee \kappa = \overline{n_1,n_2} \\
\mathit{dom}(\gamma) & = & \{\, r \mid (r^{\kappa}\triangleright\pi)\in\gamma \,\} \\
\mathit{live}(\gamma) & = & \{\, r^{\kappa}\triangleright\pi \mid (r^{\kappa}\triangleright\pi)\in\gamma \wedge \mathit{is\_live}(\gamma,r) \,\} \\
\mathit{is\_pure}(\kappa) & = & \exists n_1.\, \exists n_2.\, \kappa = n_1,n_2 \\
\mathit{consistent}(\gamma_1;\gamma_2) & = & (\forall (r^{\kappa}\triangleright\pi)\in\gamma_1.\, \forall (r^{\kappa'}\triangleright\pi')\in\gamma_2.\, \pi=\pi' \wedge (\mathit{is\_pure}(\kappa) \Leftrightarrow \mathit{is\_pure}(\kappa'))) \\
 & & \quad \wedge\, \mathit{dom}(\gamma_2)\subseteq\mathit{dom}(\gamma_1) \wedge \mathit{live}(\gamma_1)=\gamma_1 \wedge \mathit{live}(\gamma_2)=\gamma_2 \\
\mathit{abs\_par}(\gamma_1;\gamma_2) & = & \left\{ r \mid (r^{\kappa}\triangleright r')\in\gamma_1 \wedge (r^{\kappa'}\triangleright ?)\in\gamma_2 \right\}
\end{array}$$

Figure 8: Auxiliary functions and predicates.

- for sequential application, all parent regions that become abstracted for the duration of the function call must be live after the function returns;
- for parallel application, the thread output effect must be empty, the thread input effect must not contain impure capabilities with positive lock counts and hierarchy abstraction is disallowed.

The typing rules for references are standard. In Figure 5 we only show the rules for dereference (*T-D*) and reference allocation (*T-NR*). The former checks that region r is *accessible*. The latter only checks that the region r is live. The rule for creating new regions (*T-NG*) checks that e_1 is a handle for some live region r'. Expression e_2 is type checked in an extended typing context (i.e., ρ and $x : \texttt{rgn}(\rho)$ are appended to Δ and Γ respectively) and an extended input effect (i.e., a new effect is appended to the input effect such that the new region is live and accessible to this thread). The rule also checks that the type and the output effect of e_2 do not contain any occurrence of region variable ρ. This implies that ρ must be *consumed* by the end of the scope of e_2. The capability manipulation rule (*T-CP*) checks that e is a handle of a live region r. It then modifies the capability count of r as dictated by function $[\![\eta]\!]$, which increases or decreases the region or the lock count of its argument, according to the value of η. The dynamic semantics ensures that an operational step is performed when the actual counts are consistent with the desired output effect. For instance, if the lock of region r is held by some other executing thread, the evaluation of $\texttt{cap}_{\mathsf{lk}+}$ must be suspended until the lock can be obtained. On the other hand, the evaluation of $\texttt{cap}_{\mathsf{rg}-}$ does not need to suspend but may not be able to physically deallocate a region, as it may be used by other threads.

7 Type Safety

In this section we discuss the fundamental theorems that prove type safety of our language.[4] The type safety formulation is based on proving the *preservation* and *progress* lemmata. Informally, a program written in our language is safe when for each thread of execution an evaluation step can be performed or that thread is waiting for a lock (*blocked*). As discussed in Section 5, a thread may become stuck when it accesses a region that is not live or accessible (these are obviously the interesting cases in our concurrent setting; of course a thread may become stuck when it performs a non well-typed operation). Deadlocked threads are not considered to be stuck.

Definition 1 (Thread Typing) Let T be a collection of threads. Let $R; M; \delta$ be a global typing context, in which δ is a mapping from thread identifiers to effects, used only for metatheoretic purposes. For each

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

thread $n : e$ in T, we take $\delta(n)$ to be the input effect that corresponds to the evaluation of expression e. The following rules define *well-typed* threads.

$$\frac{}{R; M; \emptyset \vdash_T \emptyset} \qquad \frac{R; M; \delta \vdash_T T \qquad R; M; \emptyset; \emptyset \vdash e : \langle\rangle \,\&\, (\gamma;\emptyset) \qquad n \notin dom(\delta)}{R; M; \delta, n \mapsto \gamma \vdash_T T, n : e}$$

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

- *Region consistency*: the set of region names occurring in the co-domain of δ is a subset of the set of region names in S.
- *Static-dynamic count consistency*: for each region, the dynamic region and lock counts of some thread must be greater than or equal to the corresponding static counts of the same thread.
- *Mutual exclusion*: only one thread may have a positive lock count in δ for a particular region $\mathtt{j}$. Additionally, only this thread is allowed to access or lock sub-regions of $\mathtt{j}$.

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

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

Definition 4 (Configuration Typing) A configuration $S; T$ is *well-typed* with respect to $R; M; \delta$ (we denote this by $R; M; \delta \vdash_C S; T$) when the collection of threads T is well-typed with respect to $R; M; \delta$ and the store S is well-typed with respect to $R; M; \delta$.

Definition 5 (Not stuck) A configuration $S; T$ is *not stuck* when each thread in T can take one of the evaluation steps in Figure 3 (*E-S*, *E-T* or *E-SN*) or it is waiting for a lock held by some other thread.

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

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

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

An expression-level 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 $R; M; \delta, n \mapsto \gamma \vdash_{str} S$ and let e be a closed well-typed *redex* with $R; M; \emptyset; \emptyset \vdash e : \tau \,\&\, (\gamma;\gamma')$. Then exactly one of the following is true:

- e is of the form $\mathtt{cap}_{\mathsf{lk+}}\ \mathtt{rgn}_{\mathtt{j}}$ and $\mathtt{j}$ is a live but inaccessible region to thread n, or

- e is of the form $(\lambda x.\, e_1 \texttt{ as } \tau\ v)^{\mathsf{par}(\gamma)}$ or
- there exist S' and e' such that $S\,; e \rightarrow_n S'; e'$.

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

The *type safety* theorem is a direct consequence of Lemmata 1 and 2. Let function `main` be the initial program, let ι_H be global heap region and let the initial typing contexts R_0 and δ_0 and the initial program configuration $S_0; T_0$ be defined by the following singleton lists:

$$\begin{array}{lcl} R_0 & = & \{\iota_H\} \\ \delta_0 & = & \{1 \mapsto \iota_H^{1,0} \triangleright \bot\} \\ \theta_0 & = & \{1 \mapsto 1, 0\} \\ S_0 & = & \{\iota_H : (\theta_0, \emptyset, \emptyset)\} \\ T_0 & = & \{1 : (\texttt{main}[\iota_H]\ \texttt{rgn}_{\iota_H})^{\mathsf{seq}}\} \end{array}$$

Theorem 1 (Type Safety) If the initial configuration $S_0; T_0$ is well-typed with $R_0; \emptyset; \delta_0 \vdash_C S_0; T_0$ and the operational semantics takes any number of steps $S_0; T_0 \rightsquigarrow^n S_n; T_n$, then the resulting configuration $S_n; T_n$ is not stuck.

The empty (except for R_0 that contains only ι_H) contexts that are used when typechecking the initial configuration $S_0; T_0$ guarantee that all functions in the program are closed and that no explicit region values ($\texttt{rgn}_\iota$) or location values ($\texttt{loc}_\ell$) are used in the source of the original program.

8 Related Work

The first statically checked stack-based region system was developed by Tofte and Talpin Tofte and Talpin [1994]. Since then, several memory-safe systems that enabled early region deallocation for a sequential language were proposed Aiken et al. [1995], Henglein et al. [2001], Walker and Watkins [2001], Fluet et al. [2006]. Cyclone Grossman et al. [2002] and RC Gay and Aiken [2001] were the first imperative languages to allow safe region-based management with explicit constructs. Both allowed early region deallocation and RC also introduced the notion of multi-level region hierarchies. RC programs may throw region-related exceptions, whereas our approach is purely static. Both Cyclone and RC make no claims of memory safety or race freedom for concurrent programs. Grossman proposed a type system for safe multi-threading in Cyclone Grossman [2003]. Race freedom is guaranteed by statically tracking locksets within lexically-scoped synchronization constructs. Grossman's proposal allows for fine-grained locking, but only deals with stack-based regions and does not enable early release of regions and locks. In contrast, we support hierarchical locking, as opposed to just primitive locking, and bulk region deallocation.

Statically checked region systems have also been proposed Boyapati et al. [2003], Zhao et al. [2004, 2008] for real-time Java to rule out dynamic checks imposed by the language specification. Boyapati et al. Boyapati et al. [2003] introduce hierarchical regions in ownership types but the approach suffers from the same disadvantages as Grossman's work. Additionally, their type system only allows sub-regions for *shared* regions, whereas we do not have this limitation. Boyapati also proposed an ownership-based type system that prevents deadlocks and data races Boyapati et al. [2002]; in contrast to his system, we support locking of arbitrary nodes in the region hierarchy. Static region hierarchies (depth-wise) have been used by Zhao Zhao et al. [2004]. Their main advantage is that programs require fewer annotations compared to programs with explicit region constructs. In the same track, Zhao et al. Zhao et al. [2008]

proposed implicit ownership annotations for regions. Thus, classes that have no explicit owner can be allocated in any static region. This is a form of *existential ownership*. In contrast, we allow a region to completely abstract its owner/ancestor information by using the *hierarchy abstraction* mechanism. None of the above approaches allow full ownership abstraction for region subtrees.

Cunningham et al. Cunningham et al. [2007] proposed a universe type system to guarantee race freedom in a calculus of objects. Similarly to our system, object hierachies can be atomically locked at any level. Unlike our system, they do not support early lock releases and lock ownership transfers between threads. Consequently, their system cannot encode two important aspects of multi-threaded programming: thread-locality and data migration. Finally, our system provides explicit memory management and supports separate compilation.

The main limitation of our work is that we require explicit annotations regarding ownership and region capabilities. Moreover, our locking system offers coarser-grained locking than most other related works. The use of hierarchical locking avoids some, though not all, deadlocks.

9 Concluding Remarks

In this paper, we have presented a concurrent language emloying region-based memory management and locking primitives. Regions and locks are organized in a common hierarchy and treated uniformly. Our language allows atomic deallocation and locking of entire subtrees at any level of the hierarchy; it also allows region and lock capabilities to be transferred between threads, encoding useful idioms of concurrent programming such as *thread-local data* and *data migration*. The type system guarantees the absence of memory access violations and data races in the presence of region aliasing.

We are currently integrating our system in Cyclone. In the future, we are planning to extend our type system to achieve an exact correspondence between static and dynamic capability counts, and provide deadlock freedom guarantees.

References

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

C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 211–230, New York, NY, USA, Nov. 2002. ACM Press.

C. Boyapati, A. Salcianu, W. S. Beebee, and M. Rinard. Ownership types for safe region-based memory management in real-time Java. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 324–337, New York, NY, USA, June 2003. ACM Press.

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

D. Cunningham, S. Drossopoulou, and S. Eisenbach. Universe Types for Race Safety. In *VAMP 07*, pages 20–51, September 2007. URL `http://pubs.doc.ic.ac.uk/universes-races/`.

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

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

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

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

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

F. Henglein, H. Makholm, and H. Niss. A direct approach to control-flow sensitive region-based memory management. In *Proceedings of the 3rd ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 175–186, New York, NY, USA, 2001. ACM. ISBN 1-58113-388-X.

M. Tofte and J.-P. Talpin. Implementation of the typed call-by-value λ-calculus using a stack of regions. In *Conference Record of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 188–201, New York, NY, USA, Jan. 1994. ACM Press.

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

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

T. Zhao, J. Noble, and J. Vitek. Scoped types for real-time Java. In *Proceedings of the 25th IEEE International Real-Time Systems Symposium*, pages 241–251. IEEE Computer Society, 2004. ISBN 0-7695-2247-5.

T. Zhao, J. Baker, J. Hunt, J. Noble, and J. Vitek. Implicit ownership types for memory management. *Sci. Comput. Program.*, 71(3):213–241, 2008.

Appendix

Language Syntax & Substitution Relation

$$
\begin{array}{llll}
\textbf{Value} & v & ::= & f \mid c \mid \texttt{rgn}_\imath \mid \texttt{loc}_l \\
\textbf{Expression} & e & ::= & x \mid c \mid f \mid (e\ e)^\xi \mid e\,[r] \mid \texttt{new}\,e\,\texttt{at}\,e\epsilon \mid e := e \mid \texttt{loc}_l \\
 & & \mid & \texttt{deref}\,e \mid \texttt{newrgn}\,\rho, x\,\texttt{at}\,e\,\texttt{in}\,e \mid \texttt{cap}_\eta\,e \mid \texttt{rgn}_\imath \\
\textbf{Capability kind} & \psi & ::= & \mathsf{rg} \mid \mathsf{lk} \\
\textbf{Capability op} & \eta & ::= & \psi+ \mid \psi- \\
\textbf{Region} & r & ::= & \rho \mid \imath \\
\textbf{Capability} & \kappa & ::= & n, n \mid \overline{n, n} \\
\textbf{Region parent} & \pi & ::= & r \mid \bot \mid ? \\
\textbf{Region} & r & ::= & \rho \mid \imath \\
\textbf{Effect} & \gamma & ::= & \emptyset \mid \gamma, r^\kappa \triangleright \pi \\
\textbf{Type} & \tau & ::= & b \mid \tau \xrightarrow{\gamma \to \gamma} \tau \mid \forall \rho.\, \tau \mid \texttt{ref}(\tau, r) \mid \texttt{rgn}(r) \\
\textbf{Value} & v & ::= & f \mid c \mid \texttt{rgn}_\imath \mid \texttt{loc}_l \\
\textbf{Function} & f & ::= & \lambda x.\, e\,\texttt{as}\,\tau \xrightarrow{\gamma \to \gamma} \tau \mid \Lambda \rho.\, f \\
\textbf{Calling mode} & \xi & ::= & \mathsf{seq} \mid \mathsf{par}(\gamma)
\end{array}
$$

$$
\begin{array}{llll}
x_1[v/x] & = & v & x_1 \equiv x \\
 & \mid & x_1 & \textit{otherwise} \\
r_1[r/\rho] & = & r & r_1 \equiv \rho \\
 & \mid & r_1 & \textit{otherwise}
\end{array}
$$

$$
\begin{array}{llll}
\pi[r_1/r_2] & = & \bot \mid ? \mid r[r_1/r_2] & \\
e[v/x] & = & x[v/x] \mid c \mid \texttt{rgn}_\imath \mid \texttt{cap}_\eta\, e_1[v/x] & \\
 & \mid & \texttt{new}\, e_1[v/x]\,\texttt{at}\, e_2[v/x]\epsilon \mid \texttt{deref}\, e_1[v/x] \mid e_1[v/x] := e_2[v/x] & \\
 & \mid & \texttt{loc}_l \mid f \mid (e_1[v/x]\ e_2[v/x])^\xi \mid (e_1[v/x])[r] & \\
 & \mid & \texttt{newrgn}\,\rho, y\,\texttt{at}\, e_1[v/x]\,\texttt{in}\, e_2[v/x] & y \not\equiv x \\
f & = & \lambda x.\, e[r/\rho]\,\texttt{as}\,\tau_1[r/\rho] \xrightarrow{\gamma_1[r/\rho] \to \gamma_2[r/\rho]} \tau_2[r/\rho] \mid \Lambda \rho'.\, f[r/\rho] & \rho' \not\equiv \rho \\
e[r/\rho] & = & x \mid c \mid \texttt{rgn}_\imath \mid \texttt{cap}_\eta\, e_1[r/\rho] & \\
 & \mid & \texttt{new}\, e_1[r/\rho]\,\texttt{at}\, e_2[r/\rho]\epsilon[r/\rho] \mid \texttt{deref}\, e_1[r/\rho] \mid e_1[r/\rho] := e_2[r/\rho] & \\
 & \mid & \texttt{loc}_l \mid f[r/\rho] \mid (e_1[r/\rho]\ e_2[r/\rho])^\xi \mid (e_1[r/\rho])[r_1[r/\rho]] & \\
 & \mid & \texttt{newrgn}\,\rho', x\,\texttt{at}\, e_1[r/\rho]\,\texttt{in}\, e_2[r/\rho] & \rho' \not\equiv \rho \\
\tau[r_1/\rho] & = & b \mid \texttt{rgn}(r[r_1/\rho]) \mid \texttt{ref}(\tau[r_1/\rho], r_1[r_1/\rho]) & \\
 & \mid & \tau_1[r_1/\rho] \xrightarrow{\gamma_1[r_1/\rho] \to \gamma_2[r_1/\rho]} \tau_2[r_1/\rho] & \\
 & \mid & \forall \rho'.\, \tau[r_1/\rho] & \rho' \not\equiv \rho \\
\Gamma[r/\rho] & = & \emptyset \mid \Gamma_1[r/\rho], x : \tau[r/\rho] & \\
\gamma[r_1/\rho] & = & \emptyset \mid \gamma_1[r_1/\rho], r[r_1/\rho]^\kappa \triangleright \pi[r_1/\rho] &
\end{array}
$$

Operational Semantics

$$
\begin{array}{ll}
\mathrm{bflatten}(S) & \equiv \begin{cases} \emptyset & \text{if } S \equiv \emptyset \\ \imath : (\theta, H, S'), \mathrm{bflatten}(S'), \mathrm{bflatten}(S'') & \text{if } S = \imath : (\theta, H, S'), S'' \end{cases} \\
\mathrm{thread\_live}(S, \imath, m) & \equiv \forall \jmath : (\theta, H, S') \in \mathrm{bflatten}(S).\imath \in \{\jmath\} \cup \mathrm{dom}(S') \Rightarrow \exists n_1.\exists n_2.(m \mapsto n_1, n_2) \in \theta \wedge n_1 > 0 \\
\mathit{live}(S, \imath) & \equiv \exists m.\mathrm{thread\_live}(S, \imath, m) \\
\mathrm{dom}_\ell(S, \imath) & \equiv \{\ell \mid (\ell \mapsto v) \in H \wedge (\jmath : (\theta, H, S')) \in \mathrm{bflatten}(S)\} \\
\mathrm{dom}(S) & \equiv \{\imath \mid (\imath : (\theta, H, S_1)) \in \mathrm{bflatten}(S)\} \\
\mathrm{flatten}(S) & \equiv \{(\imath : (\theta', H', S')) \mid (\imath : (\theta', H', S')) \in \mathrm{bflatten}(S') \wedge \mathrm{live}(S, \imath)\} \\
S(\imath, \theta, H, S') & \equiv \begin{cases} S_1, \imath : (\theta, H, S'), S_3 & \text{if } S = S_1, \imath : (\theta_1, H_2, S_2), S_3 \\ S_2, \jmath : (\theta_1, H_1, S_1(\imath, \theta, H, S')) & \text{if } \imath \in \mathrm{dom}(S_1) \wedge S = S_2, \jmath : (\theta_1, H_1, S_1) \end{cases} \\
\mathrm{newrgn}(S, n, \jmath) & \equiv S(\jmath, \theta, H, S', k : (n \mapsto 1, 1, \emptyset, \emptyset)) \qquad \text{if } k \notin \mathrm{dom}(S) \wedge \jmath : (\theta, H, S') \in \mathrm{flatten}(S) \\
\mathrm{alloc}(\jmath, S, v) & \equiv S(\jmath, \theta, H, \ell \mapsto v, S') \qquad \text{if } \ell \notin \mathrm{dom}_\ell(S) \wedge \jmath : (\theta, H, S') \in \mathrm{flatten}(S) \\
\mathrm{canlk}(S, \imath, n) & \equiv \forall m : (\theta, H, S') \in \mathrm{flatten}(S).\imath \in \mathrm{dom}(S') \vee (\imath : (\theta', H', S'') \in \mathrm{flatten}(S) \wedge m \in \{\imath\} \cup \mathrm{dom}(S'')) \Rightarrow \\
 & \forall (p \mapsto n_1, n_2) \in \theta.p \neq n \Rightarrow n_2 = 0 \\
\mathrm{lookup}(S, \ell, n) & \equiv v \qquad \text{if } \imath : (\theta, H, \ell \mapsto v, S') \in \mathrm{flatten}(S) \wedge \mathrm{canlk}(S, \imath, n) \wedge \forall n' \neq n.\neg\mathrm{canlk}(S, \imath, n') \\
\mathrm{update}(S, \ell, x, n) & \equiv S(\imath, \theta, H, \ell \mapsto x, S') \qquad \text{if } \imath : (\theta, H, \ell \mapsto \mathrm{lookup}(S, \ell, n), S') \in \mathrm{flatten}(S) \\
\mathrm{transfer}(S, s, d, \gamma) & \equiv \begin{cases} S & \text{if } \gamma = \emptyset \\ \mathrm{transfer}(S(\imath, \theta, d \mapsto n_3, n_4, H, S'), s, d, \gamma') & \text{if } \gamma = \gamma', \imath^{n_1, n_2} \triangleright \pi \wedge \imath : (\emptyset, \theta, s \mapsto n_3, n_4, H, S') \in \mathrm{flatten}(S) \\ & \wedge n_1 \leq n_3 \wedge n_2 \leq n_4 \\ \mathrm{transfer}(S(\imath, \theta', H, S'), s, d, \gamma') & \text{if } \gamma = \gamma', \imath^{\overline{n_1, 0}} \triangleright \pi \wedge \imath : (\theta, s \mapsto n_1 + n_2, n_3, H, S') \in \mathrm{flatten}(S) \wedge \\ & (n \in \mathrm{dom}(\theta) \Rightarrow \theta' = \theta, s \mapsto n_2, n_3, d \mapsto \mathrm{rg}(\theta(d)) + n_1, 0) \wedge \\ & (n \notin \mathrm{dom}(\theta) \Rightarrow \theta' = \theta, s \mapsto n_2, n_3, d \mapsto n_1, 0) \end{cases} \\
\mathrm{updcnt}(n_1, n_2, \eta) & \equiv \begin{cases} n_1 \pm 1, n_2 & \text{if } \eta = \mathsf{rg}\pm \\ n_1, n_2 \pm 1 & \text{if } \eta = \mathsf{lk}\pm \end{cases} \\
\mathrm{updcap}(S, \psi, \imath, n) & \equiv S' \qquad \text{if } \imath : (\theta, H, S'') \in \mathrm{flatten}(S) \wedge \theta' = \theta[n \mapsto \mathrm{updcnt}(\theta(n), \eta)] \wedge S' = S(\imath, \theta', H, S'') \wedge (\eta = \mathsf{lk}\pm \Rightarrow \mathrm{canlk}(S, \imath, n))
\end{array}
$$

Thread map	θ	$::=$	$\emptyset \mid \theta, n_1 \mapsto n_2, n_3$
Memory heap	H	$::=$	$\emptyset \mid H, \ell \mapsto v$
Store	S	$::=$	$\emptyset \mid S, \iota : (\theta, H, S)$
Threads	T	$::=$	$\emptyset \mid T, n : e$
Configuration	C	$::=$	$S ; T$

$$\begin{array}{rcl}
E & ::= & \square \mid (E\ e)^{\xi} \mid (v\ E)^{\xi} \mid E\,[r] \\
 & \mid & \mathtt{newrgn}\,\rho, x\,\mathtt{at}\,E\,\mathtt{in}\,e \mid \mathtt{cap}_{\eta}\,E \\
 & \mid & \mathtt{new}\,E\,\mathtt{at}\,e\epsilon \mid \mathtt{new}\,v\,\mathtt{at}\,E\epsilon \\
 & \mid & \mathtt{deref}\,E \mid E := e \mid v := E
\end{array}$$

$$\frac{S ; e \to_n S' ; e'}{S ; T, n : E[e] \rightsquigarrow S' ; T, n : E[e']}\ \textit{(E-S)} \qquad \frac{\begin{array}{cc} e' \equiv ((\lambda x.\, e\ \mathtt{as}\ \tau)\ v)^{\mathsf{par}(\gamma_1)} & e'' \equiv ((\lambda x.\, e\ \mathtt{as}\ \tau)\ v)^{\mathsf{seq}} \\ \text{fresh } n' & S' = \text{transfer}(S, n, n', \gamma_1) \end{array}}{S ; T, n : E[e'] \rightsquigarrow S' ; T, n : E[()], n' : e''}\ \textit{(E-SN)} \qquad \frac{S' = \text{updcap}(S, \eta, \mathtt{j}, n)}{S ; \mathtt{cap}_{\eta}\ \mathtt{rgn}_{\mathtt{j}} \to_n S' ; ()}\ \textit{(E-C)}$$

$$\frac{}{S ; T, n : () \rightsquigarrow S ; T}\ \textit{(E-T)} \qquad \frac{}{S ; ((\lambda x.\, e\ \mathtt{as}\ \tau)\ v)^{\mathsf{seq}} \to_n S ; e[v/x]}\ \textit{(E-A)} \qquad \frac{}{S ; (\Lambda\rho.\, f)[r] \to_n S ; f[r/\rho]}\ \textit{(E-RP)} \qquad \frac{v = \text{lookup}(S, \ell, n)}{S ; \mathtt{deref}\ \mathtt{loc}_{\ell} \to_n S ; v}\ \textit{(E-D)}$$

$$\frac{(S', \ell) = \text{alloc}(\mathtt{j}, S, v)}{S ; \mathtt{new}\ v\ \mathtt{at}\ \mathtt{rgn}_{\mathtt{j}}\epsilon \to_n S' ; \mathtt{loc}_{\ell}}\ \textit{(E-NR)} \qquad \frac{S' = \text{update}(S, \ell, v, n)}{S ; \mathtt{loc}_{\ell} := v \to_n S' ; ()}\ \textit{(E-AS)} \qquad \frac{(S', k) = \text{newrgn}(S, n, \mathtt{j})}{S ; \mathtt{newrgn}\ \rho, x\ \mathtt{at}\ \mathtt{rgn}_{\mathtt{j}}\ \mathtt{in}\ e \to_n S' ; e[k/\rho][\mathtt{rgn}_k/x]}\ \textit{(E-NG)}$$

Static Semantics

$$\frac{(r^{\kappa} \triangleright \pi) \in \gamma \qquad \text{rg}(\kappa) > 0 \qquad \pi \in \{\bot, ?\}}{\text{is\_live}(\gamma, r)} \qquad \frac{(r^{\kappa} \triangleright r') \in \gamma \qquad \text{rg}(\kappa) > 0 \qquad \text{is\_live}(\gamma, r')}{\text{is\_live}(\gamma, r)}$$

$$\frac{(r^{\kappa} \triangleright \pi) \in \gamma \qquad \text{lk}(\kappa) > 0}{\text{is\_accessible}(\gamma, r)} \qquad \frac{(r^{\kappa} \triangleright r') \in \gamma \qquad \text{is\_accessible}(\gamma, r')}{\text{is\_accessible}(\gamma, r)}$$

$$\begin{array}{lcl}
\text{rg}(\kappa) & = & n_1 \quad \text{if } \kappa = n_1, n_2 \vee \kappa = \overline{n_1, n_2} \\
\text{lk}(\kappa) & = & n_2 \quad \text{if } \kappa = n_1, n_2 \vee \kappa = \overline{n_1, n_2} \\
\text{dom}(\gamma) & = & \{\, r \mid (r^{\kappa} \triangleright \pi) \in \gamma \,\} \\
\text{live}(\gamma) & = & \{\, r^{\kappa} \triangleright \pi \mid (r^{\kappa} \triangleright \pi) \in \gamma \wedge \text{is\_live}(\gamma, r) \,\} \\
\text{is\_pure}(\kappa) & = & \exists n_1.\, \exists n_2.\, \kappa = n_1, n_2 \\
\text{consistent}(\gamma_1 ; \gamma_2) & = & (\forall (r^{\kappa} \triangleright \pi) \in \gamma_1.\, \forall (r^{\kappa'} \triangleright \pi') \in \gamma_2.\, \pi = \pi' \wedge (\text{is\_pure}(\kappa) \Leftrightarrow \text{is\_pure}(\kappa'))) \\
 & & \quad \wedge\, \text{dom}(\gamma_2) \subseteq \text{dom}(\gamma_1) \wedge \text{live}(\gamma_1) = \gamma_1 \wedge \text{live}(\gamma_2) = \gamma_2 \\
\text{abs\_par}(\gamma_1 ; \gamma_2) & = & \left\{ r \mid (r^{\kappa} \triangleright r') \in \gamma_1 \wedge (r^{\kappa'} \triangleright ?) \in \gamma_2 \right\}
\end{array}$$

$$[\![\eta]\!]\,(\kappa) \quad = \quad \begin{array}{ll} \kappa' & \text{if } \eta \equiv \psi \pm \wedge \text{is\_pure}(\kappa) \Leftrightarrow \text{is\_pure}(\kappa') \wedge \\ & (\psi = \mathsf{rg} \Rightarrow \text{rg}(\kappa') = \text{rg} \pm 1 \wedge \text{lk}(\kappa') = \text{lk}(\kappa)) \wedge \\ & (\psi = \mathsf{lk} \Rightarrow \text{lk}(\kappa') = \text{lk} \pm 1 \wedge \text{rg}(\kappa') = \text{rg}(\kappa)) \end{array}$$

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

Constraint Well-formedness

$$\frac{}{R; \Delta \vdash_{\gamma} \emptyset} \qquad \frac{R; \Delta \vdash_{\gamma} \gamma_1 \qquad R; \Delta \vdash_R r_1 \qquad \pi = r_2 \Rightarrow r_2 \neq r_1 \wedge R; \Delta \vdash_R r_2}{R; \Delta \vdash_{\gamma} \gamma_1, r_1^{\kappa} \triangleright \pi}$$

Region Well-formedness

$$\frac{r \in \Delta \cup R}{R; \Delta \vdash_R r}$$

Type Well-formedness

$$\frac{}{R; \Delta \vdash b} \qquad \frac{R; \Delta \vdash_R r}{R; \Delta \vdash \mathtt{rgn}(r)} \qquad \frac{R; \Delta, \rho \vdash \tau}{R; \Delta \vdash \forall \rho.\, \tau} \qquad \frac{\begin{array}{c} R; \Delta \vdash \tau \\ R; \Delta \vdash_R r \end{array}}{R; \Delta \vdash \mathtt{ref}(\tau, r)} \qquad \frac{\begin{array}{cc} R; \Delta \vdash \tau_1 & R; \Delta \vdash_{\gamma} \gamma_1 \\ R; \Delta \vdash \tau_2 & R; \Delta \vdash_{\gamma} \gamma_2 \\ \multicolumn{2}{c}{\text{consistent}(\gamma_1 ; \gamma_2)} \end{array}}{R; \Delta \vdash \tau_1 \xrightarrow{\gamma_1 \to \gamma_2} \tau_2} \qquad \frac{}{R; \Delta \vdash \langle\rangle}$$

Variable Context Well-formedness

$$\frac{}{R; \Delta \vdash_{\Gamma} \emptyset} \qquad \frac{\begin{array}{cc} R; \Delta \vdash \tau_1 & x \notin dom(\Gamma_1) \\ \multicolumn{2}{c}{R; \Delta \vdash_{\Gamma} \Gamma_1} \end{array}}{R; \Delta \vdash_{\Gamma} \Gamma_1, x : \tau_1}$$

Memory Location Well-formedness

$$\frac{}{R \vdash_M \emptyset} \qquad \frac{\begin{array}{cc} R \vdash_M M_1 & \ell \notin dom(M_1) \\ \multicolumn{2}{c}{R; \emptyset \vdash_T \mathtt{ref}(\tau_1, \iota)} \end{array}}{R \vdash_M M_1, \ell \mapsto (\tau_1, \iota)}$$

Program Typing Context Well-formedness

$$\frac{\begin{array}{cc} R \vdash_M M & R; \Delta \vdash_{\Gamma} \Gamma \\ R; \Delta \vdash_{\gamma} \gamma & R; \Delta \vdash_{\gamma} \gamma' \\ \multicolumn{2}{c}{\text{consistent}(\gamma_1 ; \gamma_2)} \end{array}}{\vdash R; M; \Delta; \Gamma; \gamma; \gamma'}$$

$$\dfrac{\begin{array}{c}\vdash R; M; \Delta; \Gamma; \gamma; \gamma \\ (x : \tau) \in \Gamma\end{array}}{R; M; \Delta; \Gamma \vdash x : \tau \,\&\, (\gamma;\gamma)}\;\textit{(T-V)}\qquad \dfrac{\vdash R; M; \Delta; \Gamma; \gamma; \gamma}{R; M; \Delta; \Gamma \vdash n : b \,\&\, (\gamma;\gamma)}\;\textit{(T-I)}\qquad \dfrac{\vdash R; M; \Delta; \Gamma; \gamma; \gamma}{R; M; \Delta; \Gamma \vdash () : \langle\rangle \,\&\, (\gamma;\gamma)}\;\textit{(T-U)}$$

$$\dfrac{\begin{array}{c}\vdash R; M; \Delta; \Gamma; \gamma; \gamma \\ R; \Delta \vdash_R \iota\end{array}}{R; M; \Delta; \Gamma \vdash \texttt{rgn}_\iota : \texttt{rgn}(\iota) \,\&\, (\gamma;\gamma)}\;\textit{(T-R)}\qquad \dfrac{\begin{array}{c}\vdash R; M; \Delta; \Gamma; \gamma; \gamma \\ (\ell \mapsto (\tau, \iota)) \in M\end{array}}{R; M; \Delta; \Gamma \vdash \texttt{loc}_l : \texttt{ref}(\tau, \iota) \,\&\, (\gamma;\gamma)}\;\textit{(T-L)}\qquad \dfrac{\begin{array}{c}R; M; \Delta; \Gamma \vdash e_1 : \texttt{rgn}(r) \,\&\, (\gamma;\gamma', r^{\kappa}\triangleright\pi) \\ \kappa' = [\![\eta]\!](\kappa) \qquad \gamma'' = \text{live}(\gamma', r^{\kappa'}\triangleright\pi)\end{array}}{R; M; \Delta; \Gamma \vdash \texttt{cap}_\eta\; e_1 : \langle\rangle \,\&\, (\gamma;\gamma'')}\;\textit{(T-CP)}$$

$$\dfrac{\begin{array}{c}R; \Delta \vdash \tau \qquad \tau \equiv \tau_1 \overset{\gamma_1\to\gamma_2}{\longrightarrow} \tau_2 \\ R; M; \Delta; \Gamma, x : \tau_1 \vdash e : \tau_2 \,\&\, (\gamma_1;\gamma_2)\end{array}}{R; M; \Delta; \Gamma \vdash \lambda x.\, e\; \texttt{as}\; \tau : \tau \,\&\, (\gamma;\gamma)}\;\textit{(T-F)}\qquad \dfrac{R; M; \Delta, \rho; \Gamma \vdash f : \tau \,\&\, (\gamma;\gamma)}{R; M; \Delta; \Gamma \vdash \Lambda\rho.\, f : \forall\rho.\tau \,\&\, (\gamma;\gamma)}\;\textit{(T-RF)}\qquad \dfrac{\begin{array}{c}R; \Delta \vdash_R r \\ R; M; \Delta; \Gamma \vdash e : \forall\rho.\tau \,\&\, (\gamma;\gamma')\end{array}}{R; M; \Delta; \Gamma \vdash e\;[r] : \tau[r/\rho] \,\&\, (\gamma;\gamma')}\;\textit{(T-RP)}$$

$$\dfrac{\begin{array}{c}R; M; \Delta; \Gamma \vdash e_1 : \tau_1 \overset{\gamma_1\to\gamma_2}{\longrightarrow} \tau_2 \,\&\, (\gamma;\gamma') \qquad \xi \neq \textsf{seq} \Rightarrow \tau_2 = \langle\rangle \\ R; M; \Delta; \Gamma \vdash e_2 : \tau_1 \,\&\, (\gamma';\gamma'') \qquad \xi \vdash \gamma''' = \gamma_2 \oplus (\gamma'' \ominus \gamma_1)\end{array}}{R; M; \Delta; \Gamma \vdash (e_1\;\, e_2)^{\xi} : \tau_2 \,\&\, (\gamma;\gamma''')}\;\textit{(T-AP)}\qquad \dfrac{\begin{array}{c}R; M; \Delta; \Gamma \vdash e_1 : \texttt{rgn}(r) \,\&\, (\gamma;\gamma') \qquad r \in \text{dom}(\gamma') \qquad R; \Delta \vdash \tau \\ R; M; \Delta, \rho; \Gamma, x : \texttt{rgn}(\rho) \vdash e_2 : \tau \,\&\, (\gamma', \rho^{1,1}\triangleright r;\gamma'') \qquad \rho \notin \text{dom}(\gamma'')\end{array}}{R; M; \Delta; \Gamma \vdash \texttt{newrgn}\; \rho, x\; \texttt{at}\; e_1\; \texttt{in}\; e_2 : \tau \,\&\, (\gamma;\gamma'')}\;\textit{(T-NG)}$$

$$\dfrac{\begin{array}{c}R; M; \Delta; \Gamma \vdash e_1 : \tau \,\&\, (\gamma;\gamma') \\ R; M; \Delta; \Gamma \vdash e_2 : \texttt{rgn}(r) \,\&\, (\gamma';\gamma'') \qquad r \in \text{dom}(\gamma'')\end{array}}{R; M; \Delta; \Gamma \vdash \texttt{new}\; e_1\; \texttt{at}\; e_2\epsilon : \texttt{ref}(\tau, \rho) \,\&\, (\gamma;\gamma'')}\;\textit{(T-NR)}\qquad \dfrac{\begin{array}{c}R; M; \Delta; \Gamma \vdash e_1 : \texttt{ref}(\tau_1, r) \,\&\, (\gamma;\gamma') \\ R; M; \Delta; \Gamma \vdash e_2 : \tau \,\&\, (\gamma';\gamma'') \\ \text{is\_accessible}(\gamma'', r)\end{array}}{R; M; \Delta; \Gamma \vdash e_1 := e_2 : \langle\rangle \,\&\, (\gamma;\gamma'')}\;\textit{(T-A)}\qquad \dfrac{\begin{array}{c}R; M; \Delta; \Gamma \vdash e : \texttt{ref}(\tau, r) \,\&\, (\gamma;\gamma') \\ \text{is\_accessible}(\gamma', r)\end{array}}{R; M; \Delta; \Gamma \vdash \texttt{deref}\; e : \tau \,\&\, (\gamma;\gamma')}\;\textit{(T-D)}$$

$$\dfrac{\begin{array}{c}\xi \vdash \gamma = \gamma_1 \oplus \gamma_r \qquad \xi \vdash \gamma' = \gamma_2 \oplus \gamma_r \qquad \gamma'' = \text{live}(\gamma') \qquad \text{consistent}(\gamma; \gamma'') \\ \xi = \textsf{seq} \Rightarrow \text{abs\_par}(\gamma; \gamma_1) \subseteq \text{dom}(\gamma'') \qquad \xi = \textsf{par}(\gamma''') \Rightarrow \gamma_1 = \gamma''' \wedge \gamma_2 = \emptyset\end{array}}{\xi \vdash \gamma'' = \gamma_2 \oplus (\gamma \ominus \gamma_1)}\;\textit{(ESJ)}$$

$$\dfrac{}{\xi \vdash \gamma = \emptyset \oplus \gamma}\;\textit{(ES-N)}\qquad \dfrac{\pi' \in \{\pi, ?\} \qquad \xi = \textsf{par}(\gamma') \Rightarrow \pi' \neq ? \qquad \xi \vdash \kappa = \kappa_1 + \kappa_2 \qquad \xi \vdash \gamma = \gamma_1 \oplus \gamma_2}{\xi \vdash \gamma, r^{\kappa}\triangleright\pi = \gamma_1, r^{\kappa_1}\triangleright\pi' \oplus \gamma_2, r^{\kappa_2}\triangleright\pi}\;\textit{(ES-C)}$$

$$\dfrac{\begin{array}{c}\text{rg}(\kappa) = \text{rg}(\kappa_1) + \text{rg}(\kappa_2) \qquad \text{lk}(\kappa) = \text{lk}(\kappa_1) + \text{lk}(\kappa_2) \qquad \text{rg}(\kappa_1) > 0 \\ \text{is\_pure}(\kappa_1) \Leftrightarrow \text{is\_pure}(\kappa_2) \qquad \text{is\_pure}(\kappa_1) \Rightarrow \kappa = \kappa_1 \qquad \xi \neq \textsf{seq} \wedge \neg\text{is\_pure}(\kappa_1) \Rightarrow \text{lk}(\kappa_2) = 0\end{array}}{\xi \vdash \kappa = \kappa_1 + \kappa_2}\;\textit{(CS)}$$

Type Safety Judgements

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

$$\begin{array}{ll}
\text{redex}(e) & = (\exists\, S, S', e', n.\;\; S\,; e \rightarrow_n S'; e') \vee (\exists v_1, v_2, \gamma_1.(v_1\;\, v_2)^{\textsf{par}(\gamma_1)}) \\
\text{S}(\ell) & \equiv v \qquad \text{if } (\iota : (\theta, H, \ell \mapsto v, S')) \in \text{flatten}(S) \\
\text{dom}(\Gamma) & \equiv \{x \mid (x : \tau) \in \Gamma\} \\
\text{dom(M)} & \equiv \{\ell \mid (\ell \mapsto (\tau, \iota)) \in M\} \\
\text{dom}(\delta) & \equiv \{n \mid (n \mapsto \gamma) \in \delta\} \\
\text{pure\_once}(\gamma) & = \forall\iota \in \text{dom}(\gamma).\exists\gamma', n_1, n_2, \pi.\gamma_1 = \gamma', \iota^{n_1,n_2}\triangleright\pi \Rightarrow \iota \notin \text{dom}(\gamma') \\
\text{cap}(S, i, m) & \equiv n_1, n_2 \qquad \text{if } (\iota : (\theta, m \mapsto n_1, n_2, H, S')) \in \text{flatten}(S) \\
\text{cap}(\gamma, \iota) & = \begin{cases} \text{rg}(\kappa) + n_1, \text{lk}(\kappa) + n_2 & \text{if } \gamma = \gamma', \iota^{\kappa}\triangleright\pi \wedge n_1, n_2 = \text{cap}(\gamma', \iota) \\ \text{cap}(\gamma', \iota) & \text{if } \gamma = \gamma', \jmath^{\kappa}\triangleright\pi \wedge \iota \neq \jmath \\ 0, 0 & \text{if } \gamma = \emptyset \end{cases} \\
(n_1, n_2) \geq (n_3, n_4) & = n_1 \geq n_3 \wedge n_2 \geq n_4 \\
\text{counts\_ok}(\delta, \iota, S) & \equiv \forall(m \mapsto \gamma, \iota^{\kappa}\triangleright\pi) \in \delta.\text{thread\_live}(S, \iota, m) \wedge \text{cap}(S, \iota, m) \geq \text{cap}(\gamma, \iota^{\kappa}\triangleright\pi, \iota) \wedge (\text{is\_pure}(\kappa) \Rightarrow \iota \notin \text{dom}(\gamma)) \\
\text{mutex\_ok}(\delta, \iota, S) & \equiv \delta = \delta_1 \uplus m \mapsto \gamma, \iota^{\kappa}\triangleright\pi \wedge \text{lk}(\kappa) > 0 \Rightarrow \text{canlk}(S, \iota, m) \wedge (\forall(n' \mapsto \gamma') \in \delta_1.\gamma' \neq \emptyset \Rightarrow \neg\text{canlk}(S, \iota, n')) \\
\text{store\_ok}(\delta, S) & \equiv \forall\iota \in \bigcup_{(n\mapsto\gamma)\in\delta} \text{dom}(\gamma).\text{counts\_ok}(\delta, \iota, S) \wedge \text{mutex\_ok}(\delta, \iota, S) \\
\text{block}(S, n, e, \delta) & \equiv e \equiv E[\texttt{cap}_{+\textsf{lk}}\; \texttt{rgn}_{\texttt{j}}] \wedge \delta = \delta_1, n \mapsto \gamma \wedge \neg\text{is\_accessible}(\gamma, \texttt{j}) \wedge \neg\text{canlk}(S, \jmath, n)
\end{array}$$

Store Typing

$$\dfrac{\begin{array}{c}R = \text{dom}(S) \qquad \text{store\_ok}(\delta, S) \qquad \text{dom}_\ell(S) = \text{dom}(M) \\ \forall\ell \in \text{dom}(M).R; M; \emptyset; \emptyset \vdash S(\ell) : M(\ell) \,\&\, (\emptyset;\emptyset)\end{array}}{R; M; \delta \vdash_{str} S}$$

Not Stuck

$$\dfrac{\forall(n : e) \in T.\; (S\,; T \rightsquigarrow S'; T' \wedge (n : e) \notin T') \vee \text{block}(S, n, e, \delta)}{\delta \vdash_{ns} S\,; T}$$

Threads Typing

$$\dfrac{}{R; M; \emptyset \vdash_T \emptyset}\qquad \dfrac{R; M; \delta \vdash_T T \qquad R; M; \emptyset; \emptyset \vdash e : \langle\rangle \,\&\, (\gamma;\emptyset) \qquad n \notin \text{dom}(\delta)}{R; M; \delta, n \mapsto \gamma \vdash_T T, n : e}$$

Configuration Typing

$$\dfrac{R; M; \delta \vdash_T T \qquad R; M; \delta \vdash_{str} S}{R; M; \delta \vdash_C S\,; T}$$

$$\frac{n > 0 \quad \delta; S; T \rightsquigarrow^{n-1} \delta_{n-1}; S_{n-1}; T_{n-1} \quad \delta_{n-1}; S_{n-1}; T_{n-1} \rightsquigarrow \delta_n; S_n; T_n}{\delta; S; T \rightsquigarrow^{n} \delta_n; S_n; T_n}\ \textit{(E-M1)} \qquad \frac{}{\delta; S; T \rightsquigarrow^{0} \delta; S; T}\ \textit{(E-M2)}$$

Progress:	$R; M; \delta \vdash_C S; T \Rightarrow \delta \vdash_{ns} S; T$
Preservation:	$R; M; \delta \vdash_C S; T \wedge S; T \rightsquigarrow S'; T' \Rightarrow \exists R' \supseteq R, M' \supseteq M, \delta'. \ R'; M'; \delta' \vdash_C S'; T'$
Safety:	$S_0; T_0 \equiv \iota_0 : (1 \mapsto 1, 0, \emptyset, \emptyset); 1 : ((f)\,[\iota_0]\ \texttt{rgn}_{\iota_0})^{\mathsf{seq}} \wedge \iota_0; \emptyset; 1 \mapsto \emptyset, \iota_0^{1,0} \triangleright \bot \vdash_C S_0; T_0 \wedge S_0; T_0 \rightsquigarrow^n S'; T' \Rightarrow \exists \delta'. \delta' \vdash_{ns} S'; T'$
Expression Progress	$\mathrm{redex}(e) \wedge R; M; \emptyset; \emptyset \vdash e : \tau \,\&\, (\delta(n); \gamma') \wedge R; M; \delta \vdash S \Rightarrow \mathrm{block}(S, n, e, \delta) \vee (\exists S', e'.\ S; e \rightarrow_n S'; e')$ $\vee (\exists e_1, \tau, v, \gamma''.\ e \equiv (\lambda x.\, e_1\ \texttt{as}\ \tau\ v)^{\mathsf{par}(\gamma'')})$
Expression Preservation	$R; M; \emptyset; \emptyset \vdash e : \tau \,\&\, (\delta(n); \gamma') \wedge S; e \rightarrow_n S'; e' \wedge R; M; \delta \vdash_{str} S \Rightarrow \exists R_1, M_1, \gamma_1. R_1 \supseteq R \wedge M_1 \supseteq M \wedge$ $R_1; M_1; \emptyset; \emptyset \vdash e' : \tau \,\&\, (\delta[n \mapsto \gamma_1]; \gamma') \wedge R_1; M_1; \delta[n \mapsto \gamma_1] \vdash_{str} S'$

1 Type Safety Proof

Theorem 1 (Type safety) *Let R_0, δ_0, S_0 and T_0 be defined as in page 13. If the initial configuration $S_0; T_0$ is well-typed with $R_0; \emptyset; \delta_0 \vdash_C S_0; T_0$ and the operational semantics takes any number of steps $S_0; T_0 \rightsquigarrow^n S_n; T_n$, then the resulting configuration $S_n; T_n$ is not stuck.*

Proof. The proof is trivial: Lemma 1 is applied on the assumptions that $S; T$ is well-typed and that the operational semantics performs n steps, to obtain that $S_n; T_n$ is well-typed for some $R_n; M_n$. Then, we apply lemma 41 to the latter fact to prove that $S_n; T_n$ is not stuck.

Lemma 1 (Multi-step Program Preservation) *Let S;T be a* closed well-typed configuration *such that $R; M; \delta \vdash_C S; T$ for some R;M. If the operational semantics evaluates $S; T$ to $S'; T'$ in n steps then there exists a* closed well-typed configuration *such that $R'; M'; \delta' \vdash_C S'; T'$, where R′ and M′ are supersets of R and M respectively.*

Proof. Proof by induction on the number of steps n. When no steps are performed the proof is immediate from the assumption. If n steps are performed we have that $S; T \rightsquigarrow^n S'; T'$ or $S; T \rightsquigarrow^{n-1} S_{n-1}; T_{n-1}$ and $S_{n-1}; T_{n-1} \rightsquigarrow S'; T'$. By applying the induction hypothesis on the fact that $S; T$ is well-typed and that $n - 1$ steps are performed we obtain that there exists a configuration context $R_{n-1}; M_{n-1}; \delta'$ such that $R_{n-1}; M_{n-1}; \delta' \vdash_C S_{n-1}; T_{n-1}$. We complete the proof by applying lemma 2 on the latter fact and $S_{n-1}; T_{n-1} \rightsquigarrow S'; T'$.

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

Proof. By case analysis on the thread evaluation relation:

Case *E-T*: By inversion of the configuration typing assumption we obtain the typing derivation of the store $(R; M; \delta, n \mapsto \emptyset \vdash_{str} S)$, and the thread context $(R; M; \delta, n \mapsto \emptyset \vdash_T T, n : ())$. By inversion of the thread typing derivation, we have that T $(R; M; \delta \vdash_T T)$ is well-typed. Lemma 4 is applied to the store typing derivation to obtain that store S is well-typed in the strengthened context δ $(R; M; \delta \vdash_{str} S)$. The new store and thread typing derivation give us that the resulting configuration $(S; T)$ is well-typed in the strengthened context δ $(R; M; \delta \vdash_C S; T)$.

Case *E-S* : By applying inversion twice to the *configuration typing* judgement we obtain that the input effect of thread $n : e$ is γ $(\delta, n \mapsto \gamma)$, e is well-typed $(R; M; \emptyset; \emptyset \vdash E[e] : \langle\rangle \& (\gamma; \emptyset))$ and that the initial store S is well-typed $(R; M; \delta \vdash_{str} S)$. The *program evaluation* assumption suggests that $e \equiv E[e']$ for some e'. By applying Lemma 5 to the typing derivation of e we obtain that e' is well-typed $(R; M; \emptyset; \emptyset \vdash e' : \tau\&(\gamma, \gamma')$ for some γ' and $\tau)$. By applying Lemma 22 to the typing derivation of e', the *expression evaluation* step $(S; e' \rightarrow_n S'; e''$, obtained from the inversion of rule *E-S*) and the store S typing derivation, we obtain that e'' is also well-typed $(R'; M'; \emptyset; \emptyset \vdash e'' : \tau\&(\gamma'', \gamma')$ for some $R \subseteq R'$, $M \subseteq M'$, $\gamma'')$, and the resulting store S' is also well-typed $(R'; M'; \delta, n \mapsto \gamma'' \vdash_{str} S')$. By applying lemma 6 to the typing derivation of e' we have that $\vdash R'; M'; \emptyset; \emptyset; \gamma''; \gamma'$. By inversion of the latter derivation we have that $R' \vdash M'$. By applying lemma 3 to $R; M; \delta \vdash T, n : E[e]$, $R \subseteq R'$, $M \subseteq M'$ and $R' \vdash M'$, we have that $R'; M'; \delta \vdash T, n : E[e]$ holds. By lemma 17 we can substitute e'' for e' in the evaluation context E (all well-typed in $R'; M'$) to obtain $R'; M'; \emptyset; \emptyset \vdash E[e''] : \langle\rangle\&(\gamma''; \emptyset)$. We can combine the latter fact with the typing of T, to derive $R'; M'; \delta, n \mapsto \gamma'' \vdash_T T, n : E[e'']$. We have shown that $R'; M'; \delta[n \mapsto \gamma''] \vdash_{str} S'$, thus the latter two facts imply that the configuration $S'; T, n : E[e'']$ is well-typed in the typing context $R'; M'; \delta, n \mapsto \gamma''$.

Case *E-SN*: The *program evaluation* assumption implies that $e \equiv E[e']$, such that e' is a parallel application redex, and its premise asserts that e' is moved to a new thread as a local application redex e''. The resulting store S' is a function of the old store S and the regions whose ownership is transferred to the new thread $(\mathrm{transfer}(S, n, n', \gamma_1))$. By inversion of the configuration typing anssumption, $R; M; \delta, n \mapsto \gamma \vdash_T T, n : E[e']$ and $R; M; \delta, n \mapsto \gamma \vdash_{str} S$ hold. By inversion of the thread typing assumption, $E[e']$ is well-typed in the typing context $R; M; \emptyset; \emptyset$ with effect $(\gamma; \emptyset)$. By applying lemma 5 to the typing derivation of $E[e']$ we obtain that e' is well-typed in the context $R; M; \emptyset; \emptyset$ with effect $(\gamma; \gamma')$. By inversion of the latter derivation $\mathsf{par}(\gamma_1) \vdash \gamma' = \emptyset \oplus (\gamma \ominus \gamma_1)$ holds, where γ_1 is the new thread's input effect. It suffices to prove that $R; M; \delta' \vdash_{str} S'$ and $R; M; \delta' \vdash_T T, n : E[()], n' : e''$ hold, where $\delta' = \delta[n \mapsto \gamma'], n' \mapsto \gamma_1$. The proof of the former obligation is immediate by the application of lemma 21 to the fact that S is a well-typed store, S' is derived from S ($S' = \text{transfer}(S, n, n', \gamma_1)$), and $\mathsf{par}(\gamma_1) \vdash \gamma' = \emptyset \oplus (\gamma \ominus \gamma_1)$ holds.

The latter obligation can be eliminated by proving that $E[()]$ and e'' are well-typed with effects $(\gamma'; \emptyset)$ and $(\gamma_1; \emptyset)$ respectively ($R; M; \emptyset; \emptyset$ is fixed). The application of lemma 6 to the typing derivation of e' yields $\vdash R; M; \emptyset; \emptyset; \gamma; \gamma'$. Thus, $\vdash R; M; \emptyset; \emptyset; \gamma'; \gamma'$ and $R; M; \emptyset; \emptyset \vdash () : \langle\rangle \& (\gamma'; \gamma')$ hold. The application of lemma 17 to the typing derivation of $E[()]$, e' and $()$ implies that $E[()]$ is well-typed in the typing context $R; M; \emptyset; \emptyset$ with effect $(\gamma'; \emptyset)$. Finally, the application of lemma 18 to the typing derivation of e' yields that e'' is well-typed in the context $R; M; \emptyset; \emptyset$ with effect $(\gamma_1; \emptyset)$.

Lemma 3 (Thread Weakening) $R; M; \delta \vdash T \wedge R \subseteq R' \wedge M \subseteq M' \wedge R' \vdash M' \Rightarrow R'; M'; \delta \vdash T$

Proof. Proof by induction on the shape of T.

- $\emptyset$: $R'; M'; \delta \vdash \emptyset$ trivially holds.
- $T', n : e$: By inversion of this derivation we have that
  - $R; M; \emptyset; \emptyset \vdash e : \langle\rangle \& (\gamma; \emptyset)$: The application of lemma 15 to $R \subseteq R'$ and the typing derivation of e gives us $R'; M; \emptyset; \emptyset \vdash e : \langle\rangle \& (\gamma; \emptyset)$. The application of lemma 16 to the latter derivation, $M \subseteq M'$ and $R' \vdash M'$ gives us $R'; M'; \emptyset; \emptyset \vdash e : \langle\rangle \& (\gamma; \emptyset)$.
  - $R; M; \delta' \vdash T'$: by the induction hypothesis $R'; M'; \delta' \vdash T'$ holds.

  We can use the above facts to derive $R'; M'; \delta \vdash T', n : e$ holds.

Lemma 4 (Store Strengthening - Empty γ) *If store S is well-typed in the context $M; R; \delta, n \mapsto \emptyset$, then it is also well-typed in the context $M; R; \delta$.*

Proof. It suffices to prove that store_ok(δ, S). By inversion of the assumption we have that store_ok$(\delta, n \mapsto \emptyset, S)$ holds. We unfold the definition of *store_ok*: for all ι in $\bigcup_{(n \mapsto \gamma) \in \delta, n \mapsto \emptyset} \text{dom}(\gamma)$ counts_ok$(\delta, n \mapsto \emptyset, \iota, S)$ and mutex_ok$(\delta, n \mapsto \emptyset, \iota, S)$. Thus, it suffices to show that for all ι in $\bigcup_{(n \mapsto \gamma) \in \delta} \text{dom}(\gamma)$ counts_ok(δ, ι, S) and mutex_ok(δ, ι, S) also holds. Both predicates *counts_ok* and *mutex_ok* depend on threads with non-empty effects. Thus, counts_ok(δ, ι, S) and mutex_ok(δ, ι, S) immediately follow from counts_ok$(\delta, n \mapsto \emptyset, \iota, S)$, mutex_ok$(\delta, n \mapsto \emptyset, \iota, S)$ and the latter fact.

Lemma 5 (Context Inversion) *If $E[e]$ is a well-typed expression in the typing context $R; M; \Delta; \Gamma$ with effect $(\gamma_1; \gamma_2)$, then e is also a well-typed expression for some type τ, in the same typing context with effect $(\gamma_1; \gamma_3)$ for some γ_3.*

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

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

Case $((E'\ e_2)^{\xi})[e]$: An equivalent expression for this case is $(E'[e]\ e_2)^{\xi}$. By the assumption, $(E'[e]\ e_2)^{\xi}$ is a well-typed application term. By inversion of the typing derivation of $(E'[e]\ e_2)^{\xi}$, $E[e]$ is well-typed in the same typing context with effect $(\gamma_1; \gamma')$, where γ' is its output effect. The application of the induction hypothesis to the the latter typing derivation yields that e is a well-typed term in the same typing context with effect $(\gamma_1; \gamma'')$ for some γ''.

Case $((v_1\ E')^{\xi})[e]$: An equivalent expression for this case is $(v_1\ E'[e])^{\xi}$. By inversion of the typing derivation of $(v_1\ E'[e])^{\xi}$, $E'[e]$ and v_1 are well-typed. In addition, v_1 is a value with effect $(\gamma_1; \gamma_1)$ (this is immediate by performing a case analysis on v and applying inversion). Thus, the input effect of $E'[e]$ is γ_1. The application of the induction hypothesis to the latter fact implies that e is well-typed for some type τ with effect $(\gamma_1; \gamma_3)$, for some γ_3.

Case $(\texttt{cap}_\eta\ E')[e]$,$(\texttt{deref}\ E')[e]$,$(E' := e_2)[e]$, $(\texttt{loc}_\ell := E')[e]$, $(\texttt{new}\ E'\ \texttt{at}\ e_2))[e]$, $(\texttt{new}\ v\ \texttt{at}\ E'))[e]$,$(E'\ [r])[e]$, $(\texttt{newrgn}\ \rho, x\ \texttt{at}\ E'\ \texttt{in}\ e_2)[e]$: Similar to the above proof structure.

Lemma 6 (Well-Formedness) *If an expression e is well-typed in the typing context $R; M; \Delta; \Gamma$, with effect $(\gamma; \gamma')$, then $\vdash R; M; \Delta; \Gamma; \gamma; \gamma'$ holds.*

Proof. Straightforward proof by induction on the expression typing derivation. The most interesting case is the one of rule *T-AP*:

- *T-A*: By inversion of the typing derivation of e we have that e_1 is well-typed with effect $(\gamma;\gamma_x)$, e_2 is well-typed with effect $(\gamma_x;\gamma_y)$ and $\xi \vdash \gamma' = \gamma_2 \oplus (\gamma \ominus \gamma_y)$. By applying the induction hypothesis to e_1 and e_2 we obtain that $\vdash R; M; \Delta; \Gamma; \gamma; \gamma_x$ and $\vdash R; M; \Delta; \Gamma; \gamma_x; \gamma_y$ respectively. It suffices to prove the following obligations:
  - $R \vdash M$: immediate by inversion of $\vdash R; M; \Delta; \Gamma; \gamma; \gamma_x$.
  - $R; \Delta \vdash \Gamma$: immediate by inversion of $\vdash R; M; \Delta; \Gamma; \gamma; \gamma_x$.
  - $R; \Delta \vdash \gamma$: immediate by inversion of $\vdash R; M; \Delta; \Gamma; \gamma; \gamma_x$.
  - $R; \Delta \vdash \gamma'$: the effect addition assumption implies that the regions of γ' is a subset of the regions of γ. Thus, $R; \Delta \vdash \gamma'$ follows from the fact that $R; \Delta \vdash \gamma$ holds as shown earlier.
  - consistent$(\gamma;\gamma')$: by inversion of the effect addition assumption, consistent$(\gamma;\gamma')$ holds.

Lemma 7 (Value-Effect — Using well-formedness) *If value v is well-typed in the typing context $R; M; \Delta; \Gamma$, with effect $(\gamma;\gamma)$ and $\vdash R; M; \Delta; \Gamma; \gamma_1; \gamma_2$, then v is well-typed in the same typing context with effect $(\gamma_1;\gamma_1)$ and $(\gamma_2;\gamma_2)$.*

Proof. The proof is trivial, but we provide the key steps behind the proof. The assumption implies that $\vdash R; M; \Delta; \Gamma; \gamma_1; \gamma_1$ and also $\vdash R; M; \Delta; \Gamma; \gamma_2; \gamma_2$ hold (trivial). By inversion of the value typing derivation we obtain the well-formedness derivation as well as some other premises (in the case of rules *T-L,T-R,T-V,T-F*). We may use the latter premises of value typing, which *still hold* (same typing context), along with the latter two well-formedness derivations to formulate the new value typing derivations with effect $(\gamma_1;\gamma_1)$ and $(\gamma_2;\gamma_2)$ respectively. The case for rule *T-RF* can be shown trivially by induction (the base case is the same as for rule *T-F*).

Lemma 8 (Value-Effect) *If value v is well-typed in the typing context $R; M; \Delta; \Gamma$, with effect $(\gamma;\gamma)$, and e is well-typed in the same typing context with effect $(\gamma';\gamma'')$, then v is well-typed in the same typing context with effect $(\gamma'';\gamma'')$ and $(\gamma';\gamma')$.*

Proof. By inversion of the typing derivation of v, $\vdash R; M; \Delta; \Gamma; \gamma; \gamma$ holds. Similarly, the application of lemma 6 to the typing derivation of e implies that $\vdash R; M; \Delta; \Gamma; \gamma'; \gamma''$. The proof is completed by applying lemma 7.

Lemma 9 (R Well-Formedness Weakening) $R; \Delta \vdash r \wedge R \subseteq R' \Rightarrow R'; \Delta \vdash r$

Proof. By inversion of the assumption, $r \in R \cup \Delta$ and thus $r \in R' \cup \Delta$ holds as $R \subseteq R'$. Therefore, $R'; \Delta \vdash r$ holds.

Lemma 10 (Effect Well-formedness Weakening) $R; \Delta \vdash \gamma \wedge R \subseteq R' \Rightarrow R'; \Delta \vdash \gamma$

Proof. We proceed by performing a case analysis on γ:

- $\emptyset$: $R'; \Delta \vdash \emptyset$ trivially holds.
  $R; \Delta \vdash \gamma', r^{\kappa} \triangleright \pi$: $R'; \Delta \vdash \gamma'$ holds by the induction hypothesis. $R'; \Delta \vdash r$ holds by lemma 9. If $\pi = r'$, then $R'; \Delta \vdash r'$ holds by lemma 9.

Lemma 11 (Type Context Well-formedness Weakening) $R; \Delta \vdash \tau \wedge R \subseteq R' \Rightarrow R'; \Delta \vdash \tau$

Proof. We proceed by performing a case analysis on τ:

- b: $R'; \Delta \vdash b$ trivially holds.
- $\langle\rangle$: $R'; \Delta \vdash \langle\rangle$ trivially holds.
- `rgn`(r): $R'; \Delta \vdash r$ holds by lemma 9.
- `ref`(τ', r): $R'; \Delta \vdash r$ holds by lemma 9. $R'; \Delta \vdash \tau'$ holds by the induction hypothesis.
- $\forall \rho. \tau'$: $R'; \Delta, \rho \vdash \tau'$ holds by the induction hypothesis.
- $\tau' \xrightarrow{\gamma_1 \to \gamma_2} \tau''$: $R'; \Delta \vdash \tau'$ holds by the induction hypothesis. $R'; \Delta \vdash \tau''$ holds by the induction hypothesis. $R'; \Delta \vdash \gamma_1$ holds by lemma 10. $R'; \Delta \vdash \gamma_2$ holds by lemma 10.

Lemma 12 (Variable Context Well-formedness Weakening) $R; \Delta \vdash \Gamma \wedge R \subseteq R' \Rightarrow R'; \Delta \vdash \Gamma$

Proof. We proceed by performing a case analysis on Γ:

- $\emptyset$: $R'; \Delta \vdash \emptyset$ trivially holds.
  $R; \Delta \vdash \Gamma', x : \tau$: $R'; \Delta \vdash \Gamma'$ holds by the induction hypothesis. $R'; \Delta \vdash \tau$ holds by lemma 11.

Lemma 13 (Memory Context Well-formedness Weakening — R) $R \vdash M \wedge R \subseteq R' \Rightarrow R' \vdash M$

Proof. We proceed by performing a case analysis on M:

- $\emptyset$: $R' \vdash \emptyset$ trivially holds.
  $R \vdash M', \ell \mapsto (\tau, \iota)$: $R' \vdash M'$ holds by the induction hypothesis. $R'; \emptyset \vdash \texttt{ref}(\tau, \iota)$ holds by lemma 11.

Lemma 14 (Typing Context Well-formedness Weakening) $\vdash R; M; \Delta; \Gamma; \gamma_1; \gamma_2 \wedge R \subseteq R' \Rightarrow \vdash R'; M; \Delta; \Gamma; \gamma_1; \gamma_2$

Proof. Immediate by lemmas 13, 12, 10.

Lemma 15 (Typing Context Weakening — R) *If expression e is well-typed in the typing context $R; M; \Delta; \Gamma$ and R' is a superset of R, then e is well-typed in the context $R'; M; \Delta; \Gamma$ with the same type and effect.*

Proof. By applying lemma 6 to the typing derivation of e we have that $\vdash R; M; \Delta; \Gamma; \gamma_1; \gamma_2$. Lemma 14 implies that $\vdash R'; M; \Delta; \Gamma; \gamma_1; \gamma_2$ holds.

- *T-I*: Immediate by applying rule *T-I* to $\vdash R'; M; \Delta; \Gamma; \gamma_1; \gamma_2$.
- *T-U*: Immediate by applying rule *T-U* to $\vdash R'; M; \Delta; \Gamma; \gamma_1; \gamma_2$.
- *T-R*: By inversion of this derivation we have that $R; \Delta \vdash \iota$. Lemma 9 implies that $R'; \Delta \vdash \iota$ holds. Thus, we can apply rule *T-R* to the latter fact and $\vdash R'; M; \Delta; \Gamma; \gamma_1; \gamma_2$ to complete the proof.
- *T-L*: By inversion of this derivation we have that $(\ell \mapsto (\tau', \iota)) \in M$. Thus, we can apply rule *T-L* to $\vdash R'; M; \Delta; \Gamma; \gamma_1; \gamma_2$ and $(\ell \mapsto (\tau', \iota)) \in M$ to to complete the proof.
- *T-V*: By inversion of this derivation we have that $(x : \tau') \in \Gamma$. Thus, we can apply rule *T-V* to $\vdash R'; M; \Delta; \Gamma; \gamma_1; \gamma_2$ and $(x : \tau') \in \Gamma$ to complete the proof.
- *T-F*: By inversion of this derivation we have that
  – $\vdash R; M; \Delta; \Gamma; \gamma; \gamma$: We have shown that $\vdash R'; M; \Delta; \Gamma; \gamma_1; \gamma_2$ holds.
  – $R; \Delta \vdash \tau$: $R'; \Delta \vdash \tau$ holds by lemma 11.
  – $\tau \equiv \tau_1 \xrightarrow{\gamma_1 \to \gamma_2} \tau_2$
  – $R; M; \Delta; \Gamma, x : \tau_1 \vdash e' : \tau_2 \,\&\, (\gamma_1;\gamma_2)$: the induction hypothesis is applied to the derivation of e' to derive that $R'; M; \Delta; \Gamma, x : \tau_1 \vdash e' : \tau_2 \,\&\, (\gamma_1;\gamma_2)$.

  We then apply rule *T-F* to the above facts to derive $R'; M; \Delta; \Gamma \vdash \lambda x. e'$ $\texttt{as}$ $\tau : \tau' \,\&\, (\gamma;\gamma)$.

Case *T-AP*, *T-CP*, *T-RP*, *T-NG*, *T-NR*, *T-D*, *T-RF*, *T-A*: similar reasoning is performed to prove the remaining cases. Lemmas 9 and 11 can be used for premises of the form $R; \Delta \vdash r$ and $R; \Delta \vdash \tau$ respectively.

Lemma 16 (Memory Context Weakening) *If expression e is well-typed in the typing context $R; M; \Delta; \Gamma$, $R \vdash M'$ holds, and M' is a superset of M, then e is well-typed in the context $R; M'; \Delta; \Gamma$ withthe same type and effect.*

Proof. By applying lemma 6 to the typing derivation of v we have that $\vdash R; M; \Delta; \Gamma; \gamma_1; \gamma_2$. Thus, we can substitute premise $R \vdash M$ with $R \vdash M'$ to obtain $\vdash R; M'; \Delta; \Gamma; \gamma_1; \gamma_2$.

- *T-I*: Immediate by applying rule *T-I* to $\vdash R; M'; \Delta; \Gamma; \gamma_1; \gamma_2$.
- *T-U*: Immediate by applying rule *T-U* to $\vdash R; M'; \Delta; \Gamma; \gamma_1; \gamma_2$.
- *T-R*: By inversion of this derivation we have that $R; \Delta \vdash \iota$. The proof is completed by applying rule *T-R* to $R; \Delta \vdash \iota$. and $\vdash R; M'; \Delta; \Gamma; \gamma_1; \gamma_2$.
- *T-L*: By inversion to this derivation we have that $(\ell \mapsto (\tau', \iota)) \in M$. Thus, $(\ell \mapsto (\tau', \iota)) \in M'$ also holds as $M \subseteq M'$. We can apply rule *T-L* to $\vdash R; M'; \Delta; \Gamma; \gamma_1; \gamma_2$ and $(\ell \mapsto (\tau', \iota)) \in M'$ to complete the proof.
- *T-V*: By inversion of this derivation we have that $(x : \tau') \in \Gamma$. Thus, we can apply rule *T-V* to $\vdash R; M'; \Delta; \Gamma; \gamma_1; \gamma_2$ and $(x : \tau') \in \Gamma$ to complete the proof.
- *T-F*: By inversion of this derivation we have that
  - $\vdash R; M; \Delta; \Gamma; \gamma; \gamma$: We have shown that $\vdash R; M'; \Delta; \Gamma; \gamma_1; \gamma_2$ holds.
  - $R; \Delta \vdash \tau$
  - $\tau \equiv \tau_1 \xrightarrow{\gamma_1 \to \gamma_2} \tau_2$
  - $R; M; \Delta; \Gamma, x : \tau_1 \vdash e' : \tau_2 \,\&\, (\gamma_1;\gamma_2)$: the application of the induction hypothesis to this derivation yields $R; M'; \Delta; \Gamma, x : \tau_1 \vdash e' : \tau_2 \,\&\, (\gamma_1;\gamma_2)$ holds.

  We can apply rule *T-F* to the above facts to derive $R; M'; \Delta; \Gamma \vdash \lambda x. e'$ as $\tau : \tau' \,\&\, (\gamma;\gamma)$.

Case *T-AP*, *T-CP*, *T-RP*, *T-NG*, *T-NR*, *T-D*, *T-RF*, *T-A*: We can perform similar reasoning to prove the remaining cases.

Lemma 17 (Replacement) *If expressions $E[e_1]$, e_1 and e_2 are well-typed in the typing context $R; M; \Delta; \Gamma$, with effects $(\gamma_1; \gamma_2)$,$(\gamma_1; \gamma_3)$ and $(\gamma_4; \gamma_3)$ respectively, then expression $E[e_2]$ is also well-typed in the same typing context with effect $(\gamma_4; \gamma_2)$.*

Proof. By straightforward induction on the shape of the evaluation context. The intuition behind this proof is that the substitution of e_2 for e_1 in the evaluation context E will not surpise its environment as both e_1 and e_2 yield the same output effect. In regards to the input effect, we know that the environment will not be surprised as the expressions preceding e_1 will definitely be values and can be given the input effect of e_2 (by lemma 8).

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

Case $(\texttt{new}\ v\ \texttt{at}\ E')[e]$: by inversion of the typing derivation of $(\texttt{new}\ v\ \texttt{at}\ E')[e]$, we have that that $R; M; \Delta; \Gamma \vdash v : \tau_1 \& (\gamma_1; \gamma_1)$. The application of lemma 8 to the latter judgement and the fact e_2 is well-typed with effect $(\gamma_4; \gamma_3)$ yields $R; M; \Delta; \Gamma \vdash v : \tau_1 \& (\gamma_4; \gamma_4)$. By inverson of the memory allocation construct typing derivation we have that $\text{is\_live}(\gamma_3, r)$ and $R; M; \Delta; \Gamma \vdash E'[e] : \texttt{rgn}(r) \& (\gamma_1; \gamma_2)$. The application of the induction hypothesis on the derivation of $E'[e_2]$ and the derivation of e_2 (assumption) yields $R; M; \Delta; \Gamma \vdash E'[e_2] : \tau_1 \,\&\, (\gamma_4;\gamma_2)$. Now, *T-NR* can be applied to the latter judgment, the new derivation of v, and the fact that $\text{is\_live}(\gamma_3, r)$ to obtain $R; M; \Delta; \Gamma \vdash \texttt{new}\ v\ \texttt{at}\ E'[e_2] : \texttt{ref}(\tau_1, r) \,\&\, (\gamma_4;\gamma_2)$ or equivalently $R; M; \Delta; \Gamma \vdash (\texttt{new}\ v\ \texttt{at}\ E')[e_2] : \texttt{ref}(\tau_1, r) \,\&\, (\gamma_4;\gamma_2)$.

Case $((E'\ e_2)^{\xi})[e]$, $((v\ E')^{\xi})[e]$, $(\texttt{cap}_\eta\ E')[e]$,$(\texttt{deref}\ E')[e]$,$(E' := e_2)[e]$, $(\texttt{loc}_\ell := E')[e]$, $(\texttt{new}\ E'\ \texttt{at}\ e_2))[e]$, ,$(E'\ [r])[e]$, $(\texttt{newrgn}\ \rho, x\ \texttt{at}\ E'\ \texttt{in}\ e_2)[e]$: Similar to the above proof structure.

Lemma 18 (Parallel-Sequential typing implication) *If a parallel application term is well-typed ($R; M; \Delta; \Gamma \vdash (v_1\ v_2)^{\mathsf{par}(\gamma_1)} : \langle\rangle \& (\gamma; \gamma')$), where $v_1 \equiv \lambda\gamma_1. x$ as $e \tau_1 \xrightarrow{\gamma_1 \to \emptyset} \langle\rangle$, then the corresponding sequential application term ($(v_1\ v_2)^{\mathsf{seq}}$) is also well-typed in the same typing context, with effect $(\gamma_1; \emptyset)$.*

Proof. By inversion of the parallel application typing derivation, v_1 and v_2 are well typed in the same typing context $R; M; \Delta; \Gamma$, with effects $(\gamma; \gamma)$ and $(\gamma; \gamma)$ respectively. It also implies that $R; M; \Delta; \Gamma, x : \tau_1 \vdash e : \langle\rangle \& (\gamma_1; \emptyset)$. By applying lemma 8 to the typing derivations of v_1,v_2, and the fact that e is well-typed with effect $(\gamma_1; \emptyset)$, we obtain that v_1 and v_2 are well-typed in the same typing context with effect $(\gamma_1; \gamma_1)$. We can derive $\texttt{seq} \vdash \emptyset = \emptyset \oplus (\gamma_1 \ominus \gamma_1)$. By applying *T-AP* to the latter facts, we have that $R; M; \Delta; \Gamma \vdash (v_1\ v_2)^{\mathsf{seq}} : \langle\rangle \& (\gamma_1; \emptyset)$ holds.

Lemma 19 (Store Typing Preservation — Spawn Helper 2) $\gamma_1 = \gamma_1', \iota^{\kappa} \triangleright \pi \wedge S' = \text{transfer}(S, n, n', \gamma_1) \wedge \text{live}(\gamma_1) = \gamma_1 \Rightarrow \text{thread\_live}(S', \iota, n') \wedge \text{cap}(S', \iota, n') \geq \text{cap}(\gamma_1, \iota)$

Proof. Proof by induction on the height of γ_1 (the fact that $\gamma_1 = \text{live}(\gamma_1)$ implies that the tree formed by the elements of γ_1 is finite). We perform a case analysis on κ:

- $\kappa = n_1,n_2$: assumption $transfer$ implies that if $\iota : (\emptyset, \theta, n \mapsto n_3, n_4, H, S'') \in \text{flatten}(S)$, $n_1 \leq n_3$, $n_2 \leq n_4$ hold, then $\iota : (\emptyset, \theta, n' \mapsto n_3, n_4, H, S'') \in \text{flatten}(S')$ holds. The assumption that $\text{live}(\gamma_1) = \gamma_1$ implies that n_1 is positive and thus n_3 is positive. If π is $\perp$ then thread_live(S', ι, n') trivially holds from the latter facts. Otherwise, π is equal to some $\jmath$. The fact that $\gamma_1 = \text{live}(\gamma_1)$ implies that $\gamma_1 = \gamma_1'', \jmath^{\kappa'} \triangleright \pi'$ and thus we can apply the induction hypothesis to derive that thread_live$(S', \jmath, n')$. The previous facts imply that thread_live(S', ι, n') and $\text{cap}(S', \iota, n') \geq \text{cap}(\gamma_1, \iota)$.
- $\kappa = \overline{n_1, n_2}$: similar to the previous case.

Lemma 20 (Store Typing Preservation — Spawn Helper 1) $\text{counts\_ok}(\delta, n \mapsto \gamma, \iota, S) \wedge \text{par}(\gamma_1) \vdash \gamma' = \emptyset \oplus (\gamma \ominus \gamma_1) \wedge \text{live}(\gamma_1) = \gamma_1 \wedge S' = \text{transfer}(S, n, n', \gamma_1) \Rightarrow \text{counts\_ok}(\delta, n \mapsto \gamma', n' \mapsto \gamma_1, \iota, S')$

Proof. If ι does not belong in the domain of γ_1, then $\text{counts\_ok}(\delta, n \mapsto \gamma', n' \mapsto \gamma_1, \iota, S')$ trivially holds from the fact that its counts of in S' are identical to the ones in S and the assumption that $\text{counts\_ok}(\delta, n \mapsto \gamma, \iota, S)$ holds. If ι does belong in the domain of γ_1, then it suffices to prove that for all threads m that belong in the domain of $\delta, n \mapsto \gamma', n' \mapsto \gamma_1$, such that their effect equals to $\gamma_x, \iota^{\kappa'} \triangleright \pi$ for some κ',π and γ_x, then the following conditions must hold: $(\text{is\_pure}(\kappa') \Rightarrow \iota \notin \text{dom}(\gamma_x))$, thread_live$(S', \iota, m)$ and $\text{cap}(S', \iota, m) \geq \text{cap}(\gamma_x, \iota^{\kappa'} \triangleright \pi, \iota)$. The first proof obligation can be trivially shown by using the first and the second assumption. The remaining proof obligation can be shown by performing a case analysis on m:

- $m = n'$: it can be trivially shown that $\text{counts\_ok}(\delta, n \mapsto \gamma, \iota, S)$ implies that pure_once(γ) holds and thus if κ is pure then ι is not contained in $\text{dom}(\gamma_x)$. The effect addition assumption and the latter fact implies that pure_once(γ_1) holds. The application of lemma 19 completes the proof for this case.
- $m = n$: if κ' is *pure* then ι is not contained in γ' (the effect of thread n). This is immediate by the effect addition assumption. Otherwise, ι is *impure* and is contained in the domain of γ'. Thus, ι is also impure in γ_1, by the effect addition assumption. Function $transfer$ does not modify the lock counts of thread n when κ' is impure. Additionally, the effect addition assumption and the fact that ι belongs in the domain of γ' implies that function $transfer$ will remove at most $\text{rg}(\kappa) - 1$ counts from region ι, where κ is the capability of ι in effect γ. Thus, the region count for ι for thread n in S' will be positive, by the latter fact and the assumption that $\text{counts\_ok}(\delta, n \mapsto \gamma, \iota, S)$. Therefore, $\text{cap}(S', \iota, n) \geq \text{cap}(\gamma_x, \iota^{\kappa'} \triangleright \pi, \iota)$ holds. The same consideration can be applied inductively for all ancestors of ι so as to derive that thread_live(S', ι, n) (if there would exist an ancestor of ι that is not live, then ι would not belong in the domain of γ'; this is a contradiction). The effect addition assumption implies that γ' is a subset of γ. Thus, the assumption that $\text{counts\_ok}(\delta, n \mapsto \gamma, \iota, S)$ and the latter fact imply that $\text{is\_pure}(\kappa) \Rightarrow \iota \notin \text{dom}(\gamma_x)$.
- $m \notin \{n, n'\}$: function $transfer$ modifies the counts of threads n and n' only. Therefore, the counts of thread m in S' are identical to the counts of m in S. Thus, thread_live(S', ι, m) and $\text{cap}(S', \iota, m) \geq \text{cap}(\gamma_x, \iota^{\kappa'} \triangleright \pi, \iota)$ trivially hold from the latter fact and the assumption that $\text{counts\_ok}(\delta, n \mapsto \gamma, \iota, S)$.

Lemma 21 (Store Typing Preservation — Spawn) *If S is a well-typed store in respect to $R; M; \delta, n \mapsto \gamma$, $\text{par}(\gamma_2) \vdash \gamma' = \emptyset \oplus (\gamma \ominus \gamma_2)$ holds, and S' is derived from S by changing the ownership of each region that is contained in γ_2 ($S' = \text{transfer}(S, n, n', \gamma_2)$), then S' is well-typed in respect to $R; M$ and the effect map $\delta, n \mapsto \gamma', n' \mapsto \gamma_1$ for some fresh thread identifier n'.*

Proof. It suffices to prove that for all regions ι of $\delta, n \mapsto \gamma', n' \mapsto \gamma_1$ $\text{store\_ok}(\delta, n \mapsto \gamma', n' \mapsto \gamma_1, \iota, S')$ and $\text{mutex\_ok}(\delta, n \mapsto \gamma', n' \mapsto \gamma_1, \iota, S')$ hold. The first obligation is immediate by lemma 20. The second obligation can be shown by perfoming a case analysis as follows:

- ι does not belong in the domain of γ_1: the lock capability of region ι is unmodified for all threads, thus $\text{mutex\_ok}(\delta, n \mapsto \gamma', n' \mapsto \gamma_1, \iota, S')$ is immediate by the the assumption that $\text{mutex\_ok}(\delta, n \mapsto \gamma, S)$.
- ι belongs in the domain of γ_1 and is *impure*: the lock capability of region ι is unmodified for all threads except for n' that receives a zero lock capability (by the $transfer$ assumption), thus, $\text{mutex\_ok}(\delta, n \mapsto \gamma', n' \mapsto \gamma_1, \iota, S')$ is immediate by the the assumption that $\text{mutex\_ok}(\delta, n \mapsto \gamma, S)$.
- ι belongs in the domain of γ_1 and is *pure*: if the lock capability of ι is zero then the proof is similar to the previous case. Otherwise, ι is locked by thread n and no other thread may hold a lock to the ancestors of ι ($R; M; \delta, n \mapsto \gamma \vdash_{str} S$ assumption). The lock capability of region ι is unmodified for all threads except for n' and n. In particular, the lock and region capability of ι is *moved* from thread n to n' (by the $transfer$ assumption) and ι is no longer live nor locked for thread n. The above facts imply that $\text{mutex\_ok}(\delta, n \mapsto \gamma', n' \mapsto \gamma_1, \iota, S')$ holds.

Lemma 22 (Preservation — Expressions) *Let e be a well-typed expression with $R; M; \emptyset; \emptyset \vdash e : \tau \,\&\, (\delta(n);\gamma'')$ and $R; M; \delta \vdash S$. If the operational semantics takes a step $S; e \rightarrow_n S'; e'$, then there exist $R' \supseteq R$ and $M' \supseteq M$, such that the resulting expression and the resulting store are well-typed with $R'; M'; \emptyset; \emptyset \vdash e' : \tau \,\&\, (\gamma';\gamma'')$, $R; M; \delta[n \mapsto \gamma'] \vdash S'$*

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

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

Case *T-RP*: The typing derivation of *T-RP* gives us that u is of the form $(e_x)\,[\iota]$. The operational rule that matches the shape of u is *E-RP*. Thus, u is of the form $(\Lambda\rho.\, f)\,[\iota]$. By inversion of the latter derivation we obtain that $R; M; \emptyset, \rho; \emptyset \vdash f : \tau\&(\gamma; \gamma)$, where γ equals $\delta(n)$. The premise $R; \emptyset \vdash \iota$ of rule *T-RP* implies that $\iota \in R$. The application of lemma 24 to the latter facts, and $R; \emptyset \vdash \gamma$ (premise of $\vdash R; M; \emptyset; \emptyset; \gamma; \gamma$; the well-formedness fact is immediate by the application of lemma 6 to the typing derivation of type application) gives us that $R; M; \emptyset; \emptyset \vdash f[\iota/\rho] : \tau[\iota/\rho]\&(\gamma; \gamma)$. Therefore, typing is preserved. The resulting store is identical to the input store, thus it is also well-typed by the assumption of this lemma.

Case *T-CP*: **Expression typing:** The application of lemma 6 to the typing derivation of the assumption gives us that $\vdash R; M; \emptyset; \emptyset; \gamma; \gamma''$, where γ is the equal to $\delta(n)$. Thus, $\vdash R; M; \emptyset; \emptyset; \gamma''; \gamma''$ also holds. The application of rule *T-U* to the latter fact gives us $R; M; \emptyset; \emptyset \vdash () : \langle\rangle\&(\gamma''; \gamma'')$.

Store typing: The operational rule *E-C* matches the shape of u. Thus, we need to prove that $R; M; \delta[n \mapsto \gamma''] \vdash S'$ holds, where S' equals $\text{updcap}(S, \eta, \jmath, n)$. It suffices to show that $\text{store\_ok}(\delta[n \mapsto \gamma''], S')$ holds. Consequently, for all regions ι contained in $\delta[n \mapsto \gamma'']$ we must show that:

- $\text{counts\_ok}(\delta[n \mapsto \gamma''], \iota, S')$: the store typing assumption implies that $\text{counts\_ok}(\delta, \iota, S)$ holds. Given that, $\delta[n \mapsto \gamma''] = \delta'', m \mapsto \gamma_x, \iota^{\kappa}\triangleright\pi$ for some m, δ'', γ_x, κ and π, then it suffices to show that when κ is pure then ι appears once in γ_x, $\text{thread\_live}(S', \iota, m)$ and $\text{cap}(S', \iota, m) \geq \text{cap}(\gamma_x, \iota^{\kappa}\triangleright\pi, \iota)$. The first obligation is immediate by $\text{counts\_ok}(\delta, \iota, S)$, the fact that γ'' is a subset of γ (premise of rule *T-CP*) in the case where $m = n$, and the fact that $\delta[n \mapsto \gamma''](m) = \delta(m)$ in all other cases where $m \neq n$. The remaining obligations can be shown as follows: assume that ι equals $\jmath$ and m equals n. Let us assume that the capability of $\mathtt{j}$ before the *cap* operation was κ', then $\kappa = [\![\eta]\!]\,(\kappa')$ holds (premise of rule *T-CP*). The *updcap* premise of rule *E-C* implies that $\text{cap}(S', \mathtt{j}, n) = \text{updcnt}(\text{cap}(S, \mathtt{j}, n), \eta)$. Function *updcnt* is the run-time equivalent of function $[\![]\!]$. Thus, the region/lock count is incremented/decremented by one both statically and dynamically. The assumption that $\text{counts\_ok}(\delta, \mathtt{j}, S)$ implies that $\text{thread\_live}(S, \mathtt{j}, n)$, $\text{cap}(S, \mathtt{j}, n) \geq \text{cap}(\gamma, \mathtt{j})$. The above facts imply $\text{cap}(S', \mathtt{j}, n) \geq \text{cap}(\gamma'', \mathtt{j})$. The premise of rule *T-CP* that γ'' is *live* implies that the region capability of κ is positive. Thus, $\text{thread\_live}(S', \mathtt{j}, n)$ holds. Otherwise, if ι is unequal to $\jmath$ or m is unequal to n, then $\text{counts\_ok}(\delta[n \mapsto \gamma''], \iota, S')$ trivially follows from $\text{counts\_ok}(\delta, \iota, S)$ and the fact that $\text{thread\_live}(S', \mathtt{j}, n)$ holds.

- $\text{mutex\_ok}(\delta[n \mapsto \gamma''], \iota, S')$: if η is unequal to $\mathsf{lk}\pm$, then the proof is immediate as the lock capabilities and the corresponding counts of ι are unmodified and $\text{mutex\_ok}(\delta, \iota, S')$ trivially holds. If there exists no thread that contains ι in its effect or the lock capability of ι for that thread is zero, then $\text{mutex\_ok}(\delta, \iota, S')$ trivially holds. Otherwise, there exists some thread m in δ such that it contains region ι with capability κ such that $\text{lk}(\kappa)$ is positive. We also have that if m equals n, and ι equals $\jmath$ then $\kappa = [\![\eta]\!]\,(\kappa')$ holds (premise of rule *T-CP*), where κ' is the initial capability of $\jmath$ for thread n. The store typing assumption implies that $\text{mutex\_ok}(\delta, \iota, S)$ holds. It suffices to show $\text{canlk}(S', \iota, m)$ and for all other threads m' other than m $\neg\text{canlk}(S', \iota, m')$.

  If ι does not belong in a path starting from the root and ending at any leaf such that its ancestor is $\jmath$ (i.e. ι is in the *locking path*), then the proof is immediate as $\text{canlk}(S, \iota, m)$ and for all other threads m' other than m $\neg\text{canlk}(S, \iota, m')$ hold, the capability of ι is unmodified from S to S', and ι is not in the locking path. Let us assume that ι is the locking path.

  If ι is an ancestor of $\jmath$ then if m is unequal to n, then this a contradiction as *updcap* would be undefined (premise of rule *E-C*) as we have assumed that $\text{lk}(\kappa)$ is positive for thread m. If ι is an ancestor of $\jmath$ locked by thread n, then all regions in the locking path of $\jmath$ are already protected by ι thus $\text{canlk}(S', \iota, n)$ and for all other threads m' other than n $\neg\text{canlk}(S', \iota, m')$ as they hold for S.

  If ι is equal to $\jmath$, then *updcap* implies that $m = n$ and $\text{canlk}(S', \jmath, n)$. If $\text{lk}(\kappa')$ is positive then the store typing implies that for all other threads m' other than n $\neg\text{canlk}(S', \jmath, m')$. Otherwise, the *updcap* asssumption implies that $\text{canlk}(S, \jmath, n)$ holds. Thus, $\text{canlk}(S', \jmath, n)$ also holds. Additionally, for all other threads m' other than n $\neg\text{canlk}(S', \jmath, m')$ holds as $\text{canlk}(S, \iota, n)$ holds, and if θ is the thread map of S' for region $\jmath$, then $\text{lk}(\theta(n))$ is positive.

  If ι is a descendant of $\jmath$, then m can only be equal to n as in any other case this would lead to a contradiction (see above). Both $\text{canlk}(S', \iota, m)$ and for all other threads m' other than m $\neg\text{canlk}(S', \iota, m')$ hold as they hold for S, ι is assumed to be locked ($\text{lk}(\kappa) > 0$) and the capability of ι is unmodified.

Case *T-NG*: Rule *E-NG* matches the shape of u. This rule implies that $S; \mathtt{newrgn}\,\rho, x\ \mathtt{at}\ \mathtt{rgn}_\jmath\ \mathtt{in}\ e \rightarrow_n S'; e[\iota/\rho][\mathtt{rgn}_\iota/x]$, $(S', \iota) = \text{newrgn}(S, n, \mathtt{j})$ hold.

Store typing: We must prove that $R, \iota; M; \delta[n \mapsto \gamma, \iota^{1,1}\triangleright\jmath] \vdash S'$ hold given that $R; M; \delta \vdash S$ holds. It suffices to prove the following:

- $R, \iota = \text{dom}(S')$: this is immediate by $R = \text{dom}(S)$, which can be obtained by inversion of $R; M; \delta \vdash S$, and the definition of function *newrgn*.

- store_ok($\delta[n \mapsto \gamma, \iota^{1,1} \triangleright \jmath], S'$): this is immediate by the fact that store_ok(δ, S), which can be obtained by inversion of $R; M; \delta \vdash S$, and the fact that no other thread has access/owns region ι (ι is *fresh*).

Expression typing: the store typing derivation of S' implies that $\iota \notin R$. By inversion of the typing derivation of u we have that $R; M; \emptyset, \rho; \emptyset, x : \mathtt{rgn}(\rho) \vdash e_2 : \tau\&(\gamma, \rho^{1,1} \triangleright \jmath; \gamma'')$, such that $\rho \notin dom(\gamma'')$. The application of lemma 15 to the typing derivation of e_2 and the fact that $\iota \notin R$ yields $R, \iota; M; \emptyset, \rho; \emptyset, x : \mathtt{rgn}(\rho) \vdash e_2 : \tau\&(\gamma, \rho^{1,1} \triangleright \jmath; \gamma'')$. We then apply lemma 33 on the derivation of e_2 to obtain $R, \iota; M; \emptyset; \emptyset, x : \mathtt{rgn}([\iota/\rho]) \vdash e_2[\iota/\rho] : \tau[\iota/\rho]\&(\gamma[\iota/\rho], \iota^{1,1} \triangleright \jmath; \gamma''[\iota/\rho])$. By applying lemma 6 to the original typing derivation of *newrgn* construct we have that the typing the context (including γ and γ'') is not defined in terms of ρ (i.e. ρ is *fresh*). Further, the premise of *newrgn* derivation suggests that τ is also independent of ρ (i.e. $R; \emptyset \vdash \tau$). Hence, the above facts and the definition of the substitution relation imply that the typing derivation of e_2 becomes $R, \iota; M; \emptyset; \emptyset, x : \mathtt{rgn}(\iota) \vdash e_2[\iota/\rho] : \tau\&(\gamma, \iota^{1,1} \triangleright \jmath; \gamma'')$. By the application of lemma 6 to the fact that e_2 is well-typed, we have that $\vdash R, \iota; M; \emptyset; \emptyset; \gamma, \rho^{1,1} \triangleright \jmath; \gamma''$ is well formed. By the definition of well-formedness, $\vdash R, \iota; M; \emptyset; \emptyset; \emptyset; \emptyset$ also holds. The definition of the typing rule *T-R*, the latter fact and the fact that $R, \iota; \emptyset \vdash \iota$ holds imply that $\mathtt{rgn}_\iota$ is well-typed (with type $\mathtt{rgn}(\iota)$) in the context $R, \iota; M; \emptyset; \emptyset$ with effect $(\emptyset; \emptyset)$. By applying lemma 34 to the latter derivation and the fact that $R, \iota; M; \emptyset; \emptyset, x : \mathtt{rgn}(\iota) \vdash e_2[\iota/\rho] : \tau\&(\gamma, \iota^{1,1} \triangleright \jmath; \gamma'')$ we obtain $R, \iota; M; \emptyset; \emptyset \vdash e_2[\iota/\rho][\mathtt{rgn}_\iota/x] : \tau\&(\gamma, \iota^{1,1} \triangleright \jmath; \gamma'')$.

Case *T-D*: Rule *E-D* matches the shape of u. Its premises imply that the value read from the store is equal to $S(\ell)$. The store typing assumption yields that $R; M; \emptyset; \emptyset \vdash v : \tau\&(\emptyset; \emptyset)$, where $v = S(\ell)$ and $M(\ell) = (\tau, \iota)$.

The application of lemma 6 to the typing derivation of *deref* gives us $\vdash R; M; \emptyset; \emptyset; \gamma; \gamma$, where γ is equal to $\delta(n)$. By applying lemma 7 to the latter derivation and $R; M; \emptyset; \emptyset \vdash v : \tau\&(\emptyset; \emptyset)$ gives us that $R; M; \emptyset; \emptyset \vdash v : \tau\&(\gamma; \gamma)$. The output store is identical to the input store hence it is also well-typed.

Case *T-A*:

Expression typing: the application of lemma 6 to the typing derivation of e yields that $R; M; \emptyset; \emptyset; \gamma; \gamma'$ holds, where γ and γ' are equal to $\delta(n)$. Thus, $R; M; \emptyset; \emptyset; \gamma'; \gamma'$ holds. The application of rule *T-U* to the latter fact yields that $R; M; \emptyset; \emptyset \vdash () : \langle\rangle\&(\gamma'; \gamma')$.

Store typing: the store preservation proof is as follows: By invesion of the typing derivation of e the following hold: $R; M; \emptyset; \emptyset \vdash \mathtt{loc}_\ell : \mathtt{ref}(\tau, \iota)\&(\gamma; \gamma)$, where γ is equal to $\delta(n)$, $R; M; \emptyset; \emptyset \vdash v : \tau\&(\gamma; \gamma)$ and The application of lemma 6 to the latter derivation implies $\vdash R; M; \emptyset; \emptyset; \gamma; \gamma$. Thus, $\vdash R; M; \emptyset; \emptyset; \emptyset; \emptyset$ also holds. The application of lemma 7 to the latter fact and $R; M; \emptyset; \emptyset \vdash v : \tau\&(\gamma; \gamma)$ gives us $R; M; \emptyset; \emptyset \vdash v : \tau\&(\emptyset; \emptyset)$.

The premise of the operational rule *E-AS* implies that if the input store is S, then the output store is defined as follows $S' = \text{update}(S, \ell, v, n)$. It suffices to show that the new value v stored in S is well-typed in the empty context. This has been shown above.

Case *T-NR*: the rule that matches this case is rule *E-NR*. This rules implies that the new store $(S', \ell) = \text{alloc}(\mathtt{j}, S, v)$, where v is the new value that is stored in S', and ℓ is a *fresh* location (i.e. ℓ does not exist in S). Therefore, the store typing assumption ($R; M; \delta \vdash S$) implies that ℓ does not belong in the domain of M.

By inversion of the typing derivation of construct *new* we have that:

- $R; M; \emptyset; \emptyset \vdash v : \tau\&(\gamma; \gamma)$
- $R; M; \emptyset; \emptyset \vdash \mathtt{rgn}_\iota : \mathtt{rgn}(r')\&(\gamma; \gamma)$

The application of lemma 23 to the typing derivation of v tells us that $R; \emptyset \vdash \tau$ holds. By inversion of the typing derivation of $\mathtt{rgn}_\iota$ gives us that $R; \emptyset \vdash \iota$. Thefore, $R; \emptyset \vdash \mathtt{ref}(\tau, \iota)$ holds. By applying lemma 6 to the typing derivation of v we have that $\vdash R; M; \emptyset; \emptyset; \gamma; \gamma$. By inversion of the latter derivation $R \vdash M$ holds. Location ℓ is *fresh* so it does not belong to the domain of M. Consequently, we can combine the latter facts to derive that $R \vdash M, \ell \mapsto (\tau, \iota)$.

Expression typing: the latter derivation is substituted for $R \vdash M$ in the premises of $\vdash R; M; \emptyset; \emptyset; \gamma; \gamma$ to derive that $\vdash R; M, \ell \mapsto (\tau, \iota); \emptyset; \emptyset; \gamma; \gamma$ holds. By applying rule *T-L* to the latter fact, $M, \ell \mapsto (\tau, \iota)$ we obtain that $R; M, \ell \mapsto (\tau, \iota); \emptyset; \emptyset \vdash \mathtt{loc}_\ell : \mathtt{ref}(\tau, \iota)\&(\gamma; \gamma)$.

Store typing: by applying lemma 6 to the typing derivation of construct *new* we have that $\vdash R; M; \emptyset; \emptyset; \gamma; \gamma''$, where γ'' equals γ. Thus, $\vdash R; M; \emptyset; \emptyset; \emptyset; \emptyset$ also holds. By applying lemma 7 to the latter fact and $R; M; \emptyset; \emptyset \vdash v : \tau\&(\gamma; \gamma)$ we have that $R; M; \emptyset; \emptyset \vdash v : \tau\&(\emptyset; \emptyset)$ holds. By applying lemma 16 to the latter derivation $R \vdash M, \ell \mapsto (\tau, \iota)$ and $M \subseteq M, \ell \mapsto (\tau, \iota)$ we have that $R; M, \ell \mapsto (\tau, \iota); \emptyset; \emptyset \vdash v : \tau\&(\emptyset; \emptyset)$.

By inversion of the store typing assumption we have that $\text{dom}(M) = \text{dom}_\ell(S)$ and $\forall(\ell' \mapsto (\tau, \jmath)) \in M.R; M; \emptyset; \emptyset \vdash S(\ell') : \tau \,\&\, (\emptyset; \emptyset)$. We must show that both hold in the extended memory typing context and the new store. It suffices to show that the following hold:

- $\text{dom}(M, \ell \mapsto (\tau, \iota)) = \text{dom}_\ell(S')$: The locations contained in store S' are equal to the location contained in S except for an additional location ℓ. Thus, the latter fact and $M \vdash S$ imply that $M, \ell \mapsto (\tau, \iota) \vdash S'$ holds.
- $\forall(\ell' \mapsto (\tau, \jmath)) \in M, \ell \mapsto (\tau, \iota).R; M; \emptyset; \emptyset \vdash S'(\ell') : \tau \,\&\, (\emptyset; \emptyset)$: immediate by $\forall(\ell' \mapsto (\tau, \jmath)) \in M.R; M; \emptyset; \emptyset \vdash S(\ell') : \tau \,\&\, (\emptyset; \emptyset)$ and $R; M, \ell \mapsto (\tau, \iota); \emptyset; \emptyset \vdash v : \tau\&(\emptyset; \emptyset)$.

Case *T-AP*: we only need to consider the case where $\xi = \mathtt{seq}$, as only rule *E-A* matches the shape of u. The store preservation proof is immediate as the output store is identical to the input store. The proof for the typing preservation is similar to the previous proofs. Briefly, the function application typing derivation is inverted twice, so as to obtain $R; M; \emptyset; \emptyset, x : \tau_1 \vdash e : \tau_2\&(\gamma_1; \gamma_2)$. By inversion of the application derivation we have that $R; M; \emptyset; \emptyset \vdash v : \tau_1\&(\gamma; \gamma)$ and $\mathsf{seq} \vdash \gamma'' = \gamma_2 \oplus (\gamma \ominus \gamma_1)$. The application of lemma 35 to the typing derivation of v yields: $R; M; \emptyset; \emptyset \vdash v : \tau_1\&(\emptyset; \emptyset)$. Now lemma 34 is applied to the typing derivation of v and e to obtain: $R; M; \emptyset; \emptyset \vdash e[v/x] : \tau_2\&(\gamma_1; \gamma_2)$. Finally, lemma 36 is applied to $R; M; \emptyset; \emptyset \vdash e[v/x] : \tau_2\&(\gamma_1; \gamma_2)$, $\vdash R; M; \emptyset; \emptyset; \gamma; \gamma''$ (obtained by lemma 6 applied to the typing derivation of value v), and $\mathsf{seq} \vdash \gamma'' = \gamma_2 \oplus (\gamma \ominus \gamma_1)$ to obtain $R; M; \emptyset; \emptyset \vdash e[v/x] : \tau_2\&(\gamma; \gamma'')$.

Lemma 23 (Type Well-formedness) $R; M; \Delta; \Gamma \vdash e : \tau\&(\gamma; \gamma') \Rightarrow R; \Delta \vdash \tau$

Proof. Straightforward induction on the typing rules.

Lemma 24 (Polymorphic value substitution) $R, \iota; \emptyset \vdash \gamma \;\wedge\; R, \iota; M; \Delta, \rho; \emptyset \vdash f : \tau\&(\gamma; \gamma) \wedge \;\Rightarrow\; R, \iota; M; \Delta; \emptyset \vdash f[\iota/\rho] : \tau[\iota/\rho]\&(\gamma; \gamma)$

Proof. We proceed by performing a case analysis on the shape of f:

Case $f \equiv \lambda x.\, e$ as τ: by inversion of the assumption typing derivation we have that $R, \iota; M; \Delta, \rho; \emptyset \vdash \lambda x.\, e$ as $\tau : \tau\&(\gamma; \gamma)$ holds. The application of lemma 33 to the latter derivation, gives us $R, \iota; M; \Delta; \emptyset \vdash (\lambda x.\, e \text{ as } \tau)[\iota/\rho] : \tau[\iota/\rho]\&(\gamma[\iota/\rho]; \gamma[\iota/\rho])$. The assumption implies that γ is defined independently of ρ ($R, \iota; \emptyset \vdash \gamma$). Thus, $R, \iota; M; \Delta; \emptyset \vdash (\lambda x.\, e \text{ as } \tau)[\iota/\rho] : \tau[\iota/\rho]\&(\gamma; \gamma)$ also holds.

Case $f \equiv \Lambda\rho'.\, f'$: by inversion of the typing derivation of the assumption we have that $R, \iota; M; \Delta, \rho, \rho'; \emptyset \vdash f' : \tau\&(\gamma; \gamma)$. We can use the induction hypothesis to derive that $R, \iota; M; \Delta, \rho'; \emptyset \vdash f'[\iota/\rho] : \tau[r'/\rho]\&(\gamma; \gamma)$. The application of rule *T-RF* to the latter derivation yields $R, \iota; M; \Delta; \emptyset \vdash \Lambda\rho'.\, f'[\iota/\rho] : \forall\rho'.\, \tau[r'/\rho]\&(\gamma; \gamma)$.

Lemma 25 (Region substitution preserves $\oplus$) $\xi \vdash \gamma_3 = \gamma_2 \oplus \gamma_1 \;\Rightarrow\; \xi[r/\rho] \vdash \gamma_3[r/\rho] = \gamma_2[r/\rho] \oplus \gamma_1[r/\rho]$

Proof. If γ_1 is empty then rule *ES-N* implies that γ_3 equals γ_1. Thefore, $\xi[r/\rho] \vdash \gamma_1[r/\rho] = \emptyset \oplus \gamma_1[r/\rho]$ holds. It can be trivially shown that if $\xi \vdash \kappa = \kappa_1 + \kappa_2$, then for *any* r, ρ, $\xi[r/\rho] \vdash \kappa = \kappa_1 + \kappa_2$ also holds. If γ_1 is not empty then rule *ES-C* applies. By inversion of this rule we have that the following hold:

- $\gamma_3 = \gamma_{31}, r^{\kappa} \triangleright \pi$: $\gamma_3[r/\rho] = (\gamma_{31}, r^{\kappa} \triangleright \pi)[r/\rho]$ is immediate.
- $\gamma_1 = \gamma_{12}, r^{\kappa_1} \triangleright \pi'$: $\gamma_1[r/\rho] = (\gamma_{12}, r^{\kappa_1} \triangleright \pi')[r/\rho]$ is immediate.
- $\gamma_2 = \gamma_{22}, r^{\kappa_2} \triangleright \pi$: $\gamma_2[r/\rho] = (\gamma_{22}, r^{\kappa_2} \triangleright \pi)[r/\rho]$ is immediate.
- $\pi' \in \{\pi, ?\}$: $\pi'[r/\rho] \in \{\pi[r/\rho], ?[r/\rho]\}$ is immediate.
- $\xi = \mathsf{par}(\gamma_x) \Rightarrow \pi \neq ?$: $\xi[r/\rho] = \mathsf{par}(\gamma_x[r/\rho]) \Rightarrow \pi[r/\rho] \neq ?[r/\rho]$ is immediate.
- $\xi \vdash \gamma_{31} = \gamma_{12} \oplus \gamma_{22}$: $\xi[r/\rho] \vdash (\gamma_{31})[r/\rho] = \gamma_{12}[r/\rho] \oplus \gamma_{22}[r/\rho]$ holds by the induction hypothesis.

By using rule *ES-C* we obtain that: $\xi[r/\rho] \vdash \gamma_{31}[r/\rho], r[r/\rho]^{\kappa} \triangleright \pi[r/\rho] = \gamma_{12}[r/\rho], r[r/\rho]^{\kappa_1} \triangleright \pi'[r/\rho] \oplus \gamma_{22}[r/\rho], r[r/\rho]^{\kappa_2} \triangleright \pi[r/\rho]$

Lemma 26 (Region substitution preserves $\oplus/\ominus$) $\xi \vdash \gamma_3 = \gamma_2 \oplus (\gamma \ominus \gamma_1) \;\Rightarrow\; \xi \vdash \gamma_3[r/\rho] = \gamma_2[r/\rho] \oplus (\gamma[r/\rho] \ominus \gamma_1[r/\rho])$

Proof. By inversion of the first assumption we obtain the following facts:

$$(a) \vdash \gamma = \gamma_1 \oplus \gamma_r \qquad (b) \vdash \gamma_a = \gamma_2 \oplus \gamma_r \qquad (c)\; \gamma_3 = \text{live}(\gamma_a)$$
$$(d)\; \xi = \mathsf{seq} \Rightarrow \text{abs\_par}(\gamma, \gamma_1) \subseteq \text{dom}(\gamma_3) \qquad (e)\; \xi \neq \mathsf{seq} \Rightarrow \xi = \mathsf{par}(\gamma_1) \wedge \gamma_2 = \emptyset$$

We can apply lemma 25 to (a) to derive that $\xi \vdash \gamma[r/\rho] = \gamma_1[r/\rho] \oplus \gamma_r[r/\rho]$. We can apply the same reasoning to on (b) to derive $\xi \vdash \gamma_a[r/\rho] = \gamma_2[r/\rho] \oplus \gamma_r[r/\rho]$. We can trivially to show that $\mathit{abs\_par}(\gamma[r/\rho], \gamma_1[r/\rho]) \subseteq \mathit{dom}(\gamma_3[r/\rho])$ holds, by using (d) and by observing that the *live* parents substituted in $\gamma_1[r/\rho]$ correspond to the *live* parents substituted in $\gamma_3[r/\rho]$.

Lemma 27 (R Well-formedness Substitution) $R, \iota; \Delta, \rho \vdash r \Rightarrow R, \iota; \Delta \vdash r[\iota/\rho]$

Proof. The assumption implies that r belongs in $(R, \iota) \cup (\Delta, \rho)$. If r is ρ then $\rho[\iota/\rho]$ belongs in $(R, \iota) \cup \Delta$. Otherwise, $r \in (R, \iota) \cup \Delta$ trivially holds.

Lemma 28 (Effect Well-formedness Substitution) $R, \iota; \Delta, \rho \vdash \gamma \Rightarrow R, \iota; \Delta \vdash \gamma[\iota/\rho]$

Proof. We proceed by performing a case analysis on γ:

- $\emptyset$: $R, \iota; \Delta \vdash \emptyset$ trivially holds.
  $R, \iota; \Delta, \rho \vdash \gamma', r^{\kappa} \triangleright \pi$: $R, \iota; \Delta \vdash \gamma'[\iota/\rho]$ holds by the induction hypothesis. $R, \iota; \Delta \vdash r[\iota/\rho]$ holds by lemma 27. If $\pi = r'$, then $R, \iota; \Delta \vdash r'[\iota/\rho]$ holds by lemma 27.

Lemma 29 (Consistent Substitution) $\text{consistent}(\gamma_1; \gamma_2) \Rightarrow \text{consistent}(\gamma_1[\iota/\rho]; \gamma_2[\iota/\rho])$

Proof. It suffices to show that:

- if $(r[\iota/\rho]^{\kappa} \triangleright \pi[\iota/\rho]) \in \gamma_1[\iota/\rho]$ and $(r[\iota/\rho]^{\kappa'} \triangleright \pi'[\iota/\rho]) \in \gamma_2[\iota/\rho]$ for some r, then $\pi = \pi' \land (\text{is\_pure}(\kappa) \Leftrightarrow \text{is\_pure}(\kappa'))$: this is immediate by $(r^{\kappa} \triangleright \pi) \in \gamma_1$ and $(r^{\kappa'} \triangleright \pi') \in \gamma_2$, then $\pi = \pi' \land (\text{is\_pure}(\kappa) \Leftrightarrow \text{is\_pure}(\kappa'))$, which can be obtained by inversion of $\text{consistent}(\gamma_1; \gamma_2)$.
- $\text{live}(\gamma_1[\iota/\rho]) = \gamma_1[\iota/\rho]$ and $\text{live}(\gamma_2[\iota/\rho]) = \gamma_2[\iota/\rho]$: immediate by inversion of $\text{consistent}(\gamma_1; \gamma_2)$ and the definition of substitution.
- $\text{dom}(\gamma_2) \subseteq \text{dom}(\gamma_1)$: $\text{dom}(\gamma_2[\iota/\rho]) \subseteq \text{dom}(\gamma_1[\iota/\rho])$ is immediate.

Lemma 30 (Type Context Well-formedness Substitution) $R, \iota; \Delta, \rho \vdash \tau \Rightarrow R, \iota; \Delta \vdash \tau[\iota/\rho]$

Proof. We proceed by performing a case analysis on τ:

- b: $R, \iota; \Delta \vdash b$ trivially holds.
- $\langle\rangle$: $R, \iota; \Delta \vdash \langle\rangle$ trivially holds.
- $\texttt{rgn}(r)$: $R, \iota; \Delta \vdash r[\iota/\rho]$ holds by lemma 27.
- $\texttt{ref}(\tau', r)$: $R, \iota; \Delta \vdash r[\iota/\rho]$ holds by lemma 27. $R, \iota; \Delta \vdash \tau'[\iota/\rho]$ holds by the induction hypothesis.
- $\forall\rho'.\tau'$: $R, \iota; \Delta, \rho' \vdash \tau'[\iota/\rho]$ holds by the induction hypothesis.
- $\tau' \xrightarrow{\gamma_1 \rightarrow \gamma_2} \tau''$: $R, \iota; \Delta \vdash \tau'[\iota/\rho]$ holds by the induction hypothesis. $R, \iota; \Delta \vdash \tau''[\iota/\rho]$ holds by the induction hypothesis. $R, \iota; \Delta \vdash \gamma_1[\iota/\rho]$ holds by lemma 28. $R, \iota; \Delta \vdash \gamma_2[\iota/\rho]$ holds by lemma 28. We have that $\text{consistent}(\gamma_1; \gamma_2)$ and we must prove that $\text{consistent}(\gamma_1[\iota/\rho]; \gamma_2[\iota/\rho])$ holds. This is immediate by lemma 29.

Lemma 31 (Variable Context Well-formedness Substitution) $R, \iota; \Delta, \rho \vdash \Gamma \Rightarrow R, \iota; \Delta \vdash \Gamma[\iota/\rho]$

Proof. We proceed by performing a case analysis on Γ:

- $\emptyset$: $R, \iota; \Delta \vdash \emptyset$ trivially holds.
  $R, \iota; \Delta \vdash \Gamma', x : \tau$: $R, \iota; \Delta \vdash \Gamma'[\iota/\rho]$ holds by the induction hypothesis. $R, \iota; \Delta \vdash \tau[\iota/\rho]$ holds by lemma 30.

Lemma 32 (Well-formedness Substitution) $R, \iota; M; \Delta, \rho; \Gamma; \gamma_1; \gamma_2 \Rightarrow R, \iota; M; \Delta; \Gamma[\iota/\rho]; \gamma_1[\iota/\rho]; \gamma_2[\iota/\rho]$

Proof. By inversion of the first typing context and effect well-formedness assumption we have that

- $R, \iota \vdash M$
- $R, \iota; \Delta, \rho \vdash \Gamma$: $R, \iota; \Delta \vdash \Gamma[\iota/\rho]$ immediate by lemma 31.
- $R, \iota; \Delta, \rho \vdash \gamma_1$: $R, \iota; \Delta \vdash \gamma_1[\iota/\rho]$ immediate by lemma 28.
- $R, \iota; \Delta, \rho \vdash \gamma_2$: $R, \iota; \Delta \vdash \gamma_2[\iota/\rho]$ immediate by lemma 28.
- $\text{consistent}(\gamma_1; \gamma_2)$: the proof for $\text{consistent}(\gamma_1[\iota/\rho]; \gamma_2[\iota/\rho])$ is immediate by the application of lemma 29.

Lemma 33 (Region Substitution) $R, \iota; M; \Delta, \rho; \Gamma \vdash e : \tau \,\&\, (\gamma_1;\gamma_2) \Rightarrow R, \iota; M; \Delta; \Gamma[\iota/\rho] \vdash e[\iota/\rho] : \tau[\iota/\rho] \,\&\, (\gamma_1[\iota/\rho];\gamma_2[\iota/\rho])$

Proof. Proof by induction on the expression typing derivation.

Case *T-I*: by inversion of the derivation of e we have that $\gamma_1 = \gamma_2$. The application of lemma 32 to the well-formedness premise, implies that $\vdash R, \iota; M; \Delta; \Gamma[\iota/\rho];\ \gamma_1[\iota/\rho]; \gamma_2[\iota/\rho]$ holds. The proof for this case is completed by applying rule *T-I*.

Case *T-U*, by inversion of the derivation of e we have that $\gamma_1 = \gamma_2$. The application of lemma 32 to the well-formedness premise, implies that $\vdash R, \iota; M; \Delta; \Gamma[\iota/\rho];\ \gamma_1[\iota/\rho]; \gamma_2[\iota/\rho]$ holds. The proof for this case is completed by applying rule *T-U*.

Case *T-R*: by inversion of the derivation of e

- $\vdash R, \iota; M; \Delta, \rho, \Gamma; \gamma_1; \gamma_2$: the application of lemma 32 to the well-formedness premise implies that $\vdash R, \iota; M; \Delta; \Gamma[\iota/\rho]; \gamma_1[\iota/\rho]; \gamma_2[\iota/\rho]$ holds.
- $R; \Delta \vdash \jmath$: $R, \iota; \Delta \vdash \jmath[\iota/\rho]$ holds by lemma 27.

The proof for this case is completed by applying rule *T-R* to the derived facts.

Case *T-L*: by inversion of the derivation of e we have that:

- $\vdash R, \iota; M; \Delta, \rho, \Gamma; \gamma_1; \gamma_2$: the application of lemma 32 to the well-formedness premise implies that $\vdash R, \iota; M; \Delta; \Gamma[\iota/\rho]; \gamma_1[\iota/\rho]; \gamma_2[\iota/\rho]$ holds.
- $(\ell \mapsto (\tau, \jmath)) \in M$

The proof for this case is completed by applying rule *T-L* to the derived facts.

Case *T-V*: by inversion of the derivation of e we have that:

- $\vdash R, \iota; M; \Delta, \rho, \Gamma; \gamma_1; \gamma_2$: the application of lemma 32 to the well-formedness premise implies that $\vdash R, \iota; M; \Delta; \Gamma[\iota/\rho]; \gamma_1[\iota/\rho]; \gamma_2[\iota/\rho]$ holds.
- $(x : \tau) \in \Gamma$: $(x : \tau[\iota/\rho]) \in \Gamma[\iota/\rho]$ trivially holds.

The proof for this case is completed by applying rule *T-V* to the derived facts.

Case *T-F*: by inversion of the abstraction typing derivation we have that:

- $\vdash R, \iota; M; \Delta, \rho, \Gamma; \gamma; \gamma$: the application of lemma 32 well-formedness premise implies that $\vdash R, \iota; M; \Delta; \Gamma[\iota/\rho]; \gamma_1[\iota/\rho]; \gamma_2[\iota/\rho]$ holds.
- $R, \iota; \Delta, \rho \vdash \tau$: lemma 30 implies that $R, \iota; \Delta \vdash \tau[\iota/\rho]$ holds.
- $\tau \equiv \tau_1 \xrightarrow{\gamma_a \to \gamma_b} \tau_2$: the function type after substitution is $\tau[\iota/\rho] \equiv \tau_1[\iota/\rho] \xrightarrow{\gamma_a[\iota/\rho] \to \gamma_b[\iota/\rho]} \tau_2[\iota/\rho]$.
- $R, \iota; M; \Delta, \rho; \Gamma, x : \tau_1 \vdash e : \tau_2 \,\&\, (\gamma_a;\gamma_b)$: $R, \iota; M; \Delta; (\Gamma, x : \tau_1)[\iota/\rho] \vdash e[\iota/\rho] : \tau_2[\iota/\rho] \,\&\, (\gamma_a[\iota/\rho];\gamma_b[\iota/\rho])$ holds by the induction hypothesis.

The proof for this case is completed by applying rule *T-F* to the derived facts.

Case *T-AP*: by inversion of the application derivation we have that:

- $R, \iota; M; \Delta, \rho; \Gamma \vdash e_1 : \tau_1 \xrightarrow{\gamma_a \to \gamma_b} \tau_2 \,\&\, (\gamma_1;\gamma_3)$: By applying lemma 23 to the derivation of e_1 we obtain that $R, \iota; \Delta, \rho \vdash \tau_1 \xrightarrow{\gamma_a \to \gamma_b} \tau_2$. By inversion of the latter fact valid$(\gamma_a; \gamma_b)$ holds.
  $R, \iota; M; \Delta; \Gamma[\iota/\rho] \vdash e_1[\iota/\rho] : (\tau_1 \xrightarrow{\gamma_a \to \gamma_b} \tau_2)[\iota/\rho] \,\&\, (\gamma_1[\iota/\rho];\gamma_3[\iota/\rho])$ holds by the induction hypothesis.
- $\xi = \mathsf{par}(\gamma_1) \Rightarrow \tau_2 = \langle\rangle$: $\xi[r/\rho] = \mathsf{par}(\gamma_1)[\iota/\rho] \Rightarrow \tau_2[\iota/\rho] = \langle\rangle$ trivially holds.
- $R, \iota; M; \Delta, \rho; \Gamma \vdash e_2 : \tau_1 \,\&\, (\gamma_3;\gamma_4)$: $R, \iota; M; \Delta; \Gamma[\iota/\rho] \vdash e_2[\iota/\rho] : \tau_1[\iota/\rho] \,\&\, (\gamma_3[\iota/\rho];\gamma_4[\iota/\rho])$ holds by the induction hypothesis.
- $\xi \vdash \gamma_2 = \gamma_b \oplus (\gamma_4 \ominus \gamma_a)$: $\xi \vdash \gamma_2[\iota/\rho] = \gamma_b[\iota/\rho] \oplus (\gamma_4[\iota/\rho] \ominus \gamma_a[\iota/\rho])$ is immediate by the application of lemma 26.

Case *T-CP*, *T-RP*, *T-NG*, *T-NR*, *T-D*, *T-RF*, *T-E*, *T-A*: We can perform similar reasoning to prove the remaining cases. The key point is to prove in the remaining cases that $(\text{live}(\gamma_x))[\iota/\rho] = \text{live}(\gamma_x[\iota/\rho])$, where γ_x is the effect of interest. The proof can be summarized as follows:

- ρ is a leaf element in γ_x: liveness for this regions is unaffected as its parents are unaffected.
- ρ is an intermediate node in γ_x: assuming that there exist an immediate and *live* descendant r', then its parent annotation is ρ. Thus after substitution r' will still be *live*.

Lemma 34 (Variable Substitution) $R; M; \Delta; \Gamma, x : \tau_1 \vdash e : \tau_2 \,\&\, (\gamma_1;\gamma_2) \wedge R; M; \emptyset; \emptyset \vdash v : \tau_1 \,\&\, (\emptyset;\emptyset) \Rightarrow R; M; \Delta; \Gamma \vdash e[v/x] : \tau_2 \,\&\, (\gamma_1;\gamma_2)$

Proof. Straightforward induction on the expression typing derivation.

Lemma 35 (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 the value typing derivations. The proof is trivial as it suffices to prove well-formedness effect strenghening, which is immediate from the definition of well-formedness: $\vdash R; M; \Delta; \Gamma; \gamma_1; \gamma_1$ can be immediately be strengthened to $\vdash R; M; \Delta; \Gamma; \emptyset; \emptyset$. 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 36 (Context Weakening) $R; M; \Delta; \Gamma \vdash e : \tau_1 \,\&\, (\gamma_1;\gamma_2) \wedge \vdash R; M; \Delta; \Gamma; \gamma; \gamma'' \wedge \mathsf{seq} \vdash \gamma'' = \gamma_2 \oplus (\gamma \ominus \gamma_1) \Rightarrow R; M; \Delta; \Gamma \vdash e : \tau_1 \,\&\, (\gamma;\gamma'')$

Proof. By induction on the structure of e.

Case *T-R*,*T-L*,*T-F*, *T-V*,*T-I* and *T-U*: Immediate by the second assumption.

Case *T-AP*: by inversion of the typing rule we have that e_1 and e_2 are well-typed in the typing context $R; M; \Delta; \Gamma$ with effects $(\gamma_1; \gamma_3)$ and $(\gamma_3; \gamma_4)$ respectively. By applying lemma 6 to the typing derivation of e_1 we have that consistent$(\gamma_1; \gamma_3)$ holds. By applying lemma 40 to the latter fact and the assumption that $\mathsf{seq} \vdash \gamma'' = \gamma_2 \oplus (\gamma \ominus \gamma_1)$ we obtain that $\mathsf{seq} \vdash \gamma_x = \gamma_3 \oplus (\gamma \ominus \gamma_1)$ and $\mathsf{seq} \vdash \gamma'' = \gamma_2 \oplus (\gamma_x \ominus \gamma_3)$ for some γ_x. By applying lemma 6 to the typing of e_2 we obtain that consistent$(\gamma_3; \gamma_4)$ holds. By applying lemma 40 to the latter facts we have that $\mathsf{seq} \vdash \gamma_y = \gamma_4 \oplus (\gamma_x \ominus \gamma_3)$ and $\mathsf{seq} \vdash \gamma'' = \gamma_2 \oplus (\gamma_y \ominus \gamma_4)$ for some γ_y. The above facts ($\oplus$ judgements) and the well-formedness assumption tell us that $\vdash R; M; \Delta; \Gamma; \gamma; \gamma_x$ and $\vdash R; M; \Delta; \Gamma; \gamma_x; \gamma_y$ hold. We can apply the induction hypothesis to derive that e_1 and e_2 are well-typed in the same typing context with effects $(\gamma; \gamma_x)$ and $(\gamma_x; \gamma_y)$ respectively. The next step is to apply 37 to the fact obtain that $\mathsf{seq} \vdash \gamma'' = \gamma_2 \oplus (\gamma_y \ominus \gamma_4)$ and the application premise $\xi \vdash \gamma_2 = \gamma_b \oplus (\gamma_4 \ominus \gamma_a)$ to derive $\xi \vdash \gamma'' = \gamma_b \oplus (\gamma_y \ominus \gamma_a)$.

Case *T-NG*,*T-NR*, *T-A*, *T-CP*: *T-RF*, *T-RP*, *T-D*: Similar reasoning to the previous case applies to the remaining rules. It is worth mentioning that the domain of γ'' and γ is a superset of the domain of γ_1 and γ_2 respectively. In addition, it can be easily shown that the capability count of each region in the first two effects is greater than or equal to the counts of the last two effects respectively. Hence, premises which are related to *dom*, *accessible*, or $[\![\eta]\!]\,(\kappa)$ that hold for γ_1 and γ_2 also hold for γ'' and γ.

Lemma 37 (Context Weakening — $\oplus/\ominus$) $\mathsf{seq} \vdash \gamma' = \gamma_2 \oplus (\gamma \ominus \gamma_1) \wedge \xi \vdash \gamma_2 = \gamma_4 \oplus (\gamma_1 \ominus \gamma_3) \wedge \text{live}(\gamma_4) \Rightarrow \xi \vdash \gamma' = \gamma_4 \oplus (\gamma \ominus \gamma_3)$

Proof. The assumptions yield the following facts:

$$\begin{array}{cccc}
\text{(a) } \mathsf{seq} \vdash \gamma = \gamma_1 \oplus \gamma_{r1} & \text{(b) } \mathsf{seq} \vdash \gamma_a = \gamma_2 \oplus \gamma_{r1} & \text{(c) } \gamma' = \text{live}(\gamma_a) & \text{(d) abs\_par}(\gamma, \gamma_1) \subseteq \text{dom}(\gamma') \\
\text{(e) } \xi \vdash \gamma_1 = \gamma_3 \oplus \gamma_{r2} & \text{(f) } \xi \vdash \gamma_b = \gamma_4 \oplus \gamma_{r2} & \text{(g) abs\_par}(\gamma_1, \gamma_3) \subseteq \text{dom}(\gamma_2) & \text{(h) } \gamma_2 = \text{live}(\gamma_b) \\
\text{(i) } \xi \neq \mathsf{seq} \Rightarrow \xi = \mathsf{par}(\gamma_3) \wedge \gamma_2 = \emptyset & & &
\end{array}$$

By applying lemma 38 to (a) and (e) we obtain that $\exists \gamma_r''. \xi \vdash \gamma = \gamma_3 \oplus \gamma_r'' \wedge \forall r^{\kappa} \triangleright \pi \in \gamma_r''. \exists r^{\kappa_1} \triangleright \pi' \in \gamma_{r1}, r^{\kappa_2} \triangleright \pi'' \in \gamma_{r2}.\text{rg}(\kappa) = \text{rg}(\kappa_1) + \text{rg}(\kappa_2) \wedge \text{lk}(\kappa) = \text{lk}(\kappa_1) + \text{lk}(\kappa_2)$. Facts (f) and (h) as well as the third assumption imply that $\xi \vdash \gamma_2 = \gamma_4 \oplus \gamma_{r2}'$, where γ_{r2}' is a subset of γ_{r2} such that $\text{dom}(\gamma_{r2}') = \text{dom}(\gamma_2)$ holds. By applying lemma 38 to the latter derivation and (b) we obtain that $\exists \gamma_r'''. \xi \vdash \gamma_a = \gamma_4 \oplus \gamma_r''' \wedge \forall r^{\kappa} \triangleright \pi \in \gamma_r'''. \exists r^{\kappa_1} \triangleright \pi' \in \gamma_{r1}, r^{\kappa_2} \triangleright \pi'' \in \gamma_{r2}'.\text{rg}(\kappa) = \text{rg}(\kappa_1) + \text{rg}(\kappa_2) \wedge \text{lk}(\kappa) = \text{lk}(\kappa_1) + \text{lk}(\kappa_2)$. Given the above facts we can deduce that $\gamma''' = \gamma''$. To complete the proof we need to show that $\text{abs\_par}(\gamma, \gamma_3) \subseteq \text{dom}(\gamma')$. This is immediate by facts (d), (g), (b) and (c): $\text{abs\_par}(\gamma, \gamma_3) = \text{abs\_par}(\gamma_1, \gamma_3) \subseteq \text{dom}(\gamma_2) \subseteq \text{dom}(\gamma')$ (by simple observation of the definition of *abs_par*).

Lemma 38 (Context Weakening — $\oplus$) $\mathsf{seq} \vdash \gamma = \gamma_1 \oplus \gamma_{r2} \wedge \xi \vdash \gamma_1 = \gamma_2 \oplus \gamma_{r1} \Rightarrow \exists \gamma_r''. \xi \vdash \gamma = \gamma_2 \oplus \gamma_r'' \wedge \forall r^{\kappa} \triangleright \pi \in \gamma_r''. \exists r^{\kappa_1} \triangleright \pi' \in \gamma_{r1}, r^{\kappa_2} \triangleright \pi'' \in \gamma_{r2}.\text{rg}(\kappa) = \text{rg}(\kappa_1) + \text{rg}(\kappa_2) \wedge \text{lk}(\kappa) = \text{lk}(\kappa_1) + \text{lk}(\kappa_2)$

Proof. Proof by induction on the structure of γ_2:

Case $\gamma_2 = \emptyset$: The conclusion trivially holds for $\gamma_r'' = \gamma$.

Case $\gamma_2 = \gamma_2', r^{\kappa} \triangleright \pi'$: The assumptions yield the following facts:

(a) $\gamma_{r1} = \gamma'_{r1}, r^{\kappa_1} \triangleright \pi$ (b) $\gamma_1 = \gamma'_1, r^{\kappa_3} \triangleright \pi$ (c) $\gamma_{r2} = \gamma'_{r2}, r^{\kappa_4} \triangleright \pi''$ (d) $\gamma = \gamma', r^{\kappa_5} \triangleright \pi''$
(e) $\pi' \in \{?, \pi\}$ (f) $\xi \neq \mathsf{seq} \Rightarrow \pi' \neq ?$ (g) $\pi \in \{?, \pi''\}$ (h) $\xi \neq \mathsf{seq} \Rightarrow \pi \neq ?$
(i) $\mathsf{seq} \vdash \kappa_5 = \kappa_3 + \kappa_4$ (j) $\mathsf{seq} \vdash \kappa_3 = \kappa + \kappa_1$ (k) $\mathsf{seq} \vdash \gamma' = \gamma'_1 \oplus \gamma'_{r2}$ (l) $\xi \vdash \gamma_1 = \gamma'_2 \oplus \gamma'_{r1}$

By applying the induction hypothesis on (k) and (l) we have that $\exists \gamma_x . \xi \vdash \gamma' = \gamma_1 + \gamma_x$. By the definition of rule *CS*, and facts (i) and (j) it is trivial to show that there exists a κ_x such that $\xi \vdash \kappa_3 = \kappa + \kappa_x$ and $\mathrm{rg}(\kappa_x) = \mathrm{rg}(\kappa_4) + \mathrm{rg}(\kappa_1)$, $\mathrm{lk}(\kappa_x) = \mathrm{lk}(\kappa_4) + \mathrm{lk}(\kappa_1)$ hold . By (e) and (g) we can deduce that $\pi \in \{?, \pi''\}$. Now we can apply rule *ES-C* to the derived facts as well as fact (h) to conclude that $\xi \vdash \gamma = \gamma', r^{\kappa_5} \triangleright \pi'' \gamma_2 = \gamma'_2, r^{\kappa} \triangleright \pi' \oplus \gamma_x, r^{\kappa_x} \triangleright \pi''$ holds.

Lemma 39 ($\oplus$ Implication) $\mathsf{seq} \vdash \gamma = \gamma_1 \oplus \gamma_r \wedge \mathrm{consistent}(\gamma_1, \gamma_2) \Rightarrow \exists \gamma' . \mathsf{seq} \vdash \gamma' = \gamma_2 \oplus \gamma_r$

Proof. By induction on the structure of γ_2:

Case $\gamma_2 = \emptyset$: Then the conclusion trivially holds for $\gamma' = \gamma_r$.

Case $\gamma_2 = \gamma'_2, r^{n_1,n_2} \triangleright \pi'$: The second assumption yields that there exists a γ'_1 such that $\gamma_1 = \gamma'_1, r^{n_3,n_4} \triangleright \pi'$. Consequently, the first assumption becomes (rule *E-SC*): $\mathsf{seq} \vdash \gamma', r^{n_5,n_6} \triangleright \pi = \gamma'_1, r^{n_3,n_4} \triangleright \pi' \oplus \gamma'_r, r^{n_7,n_8} \triangleright \pi$ for some γ' and γ'_r. By inversion of the latter derivation we obtain that $\mathsf{seq} \vdash \gamma' = \gamma'_1 \oplus \gamma'_r$ holds. The second assumption and the above facts imply that $\mathrm{consistent}(\gamma'_1, \gamma'_2)$ also holds. By induction hypothesis we have that there exists a γ'' such that $\mathsf{seq} \vdash \gamma'' = \gamma'_2 \oplus \gamma'_r$. We can construct a fresh κ such that $\mathsf{seq} \vdash \kappa = n_1, n_2 + n_7, n_8$ by using facts from predicate *consistent*. The above facts and rule *ES-C* imply that $\mathsf{seq} \vdash \gamma'', r^{\kappa} \triangleright \pi = \gamma_2, r^{n_1,n_2} \triangleright \pi' \oplus \gamma'_r, r^{n_7,n_8} \triangleright \pi$ holds.

Case $\gamma_2 = \gamma'_2, r^{\overline{n_1,n_2}} \triangleright \pi'$: Similar to the previous case.

Lemma 40 ($\oplus$/$\ominus$ Implication) $\mathsf{seq} \vdash \gamma' = \gamma_2 \oplus (\gamma \ominus \gamma_1) \wedge \mathrm{consistent}(\gamma_1, \gamma'_1) \Rightarrow \exists \gamma'' . \mathsf{seq} \vdash \gamma'' = \gamma'_1 \oplus (\gamma \ominus \gamma_1) \wedge \mathsf{seq} \vdash \gamma' = \gamma_2 \oplus (\gamma'' \ominus \gamma'_1)$

Proof. By inversion of the first assumption we obtain: $\mathsf{seq} \vdash \gamma = \gamma_1 \oplus \gamma_r$ and $\mathsf{seq} \vdash \gamma_a = \gamma_2 \oplus \gamma_r$, for some γ_r, $\gamma' = \mathrm{live}(\gamma_a)$, and $\mathrm{abs\_par}(\gamma, \gamma_1) \subseteq \mathrm{dom}(\gamma')$. By the application of lemma 39 to $\mathsf{seq} \vdash \gamma = \gamma_1 \oplus \gamma_r$ we have that $\mathsf{seq} \vdash \gamma_b = \gamma'_1 \oplus \gamma_r$ holds for some γ_b.

To prove the left term of the conjuction of the conclusion it suffices to show that $\mathit{abs\_par}(\gamma, \gamma_1) \subseteq \mathrm{dom}(\mathrm{live}(\gamma_b))$. This can be shown by proving that $\mathrm{dom}(\mathrm{live}(\gamma_a)) \subseteq \mathrm{dom}(\mathrm{live}(\gamma_b))$. This is immediate by the second assumption, which implies that the domain of γ'_1 is a subset of the domain of γ_1, which contains positive region counts (γ_1 is *live* by assumption *consistent*). Thus, γ_b contains at least as many live regions as γ_a.

To prove the right term of the conjuction it suffices to show that $\mathit{abs\_par}(\gamma_b, \gamma'_1) \subseteq \mathrm{dom}(\gamma'')$. As mentioned earlier, γ'_1 is a subset of γ_1 so it contains at most as many abstracted parents as γ_1. Therefore, $\mathit{abs\_par}(\gamma_b, \gamma'_1) \subseteq \mathit{abs\_par}(\gamma, \gamma_1)$. Thus, $\mathit{abs\_par}(\gamma_b, \gamma'_1) \subseteq \mathrm{dom}(\gamma'')$ holds.

Lemma 41 (Progress — Program) *Let $S\,;T$ be a closed well-typed configuration with $R; M; \delta \vdash_C S\,;T$. Then $S\,;T$ is not stuck.*

Proof. In order to prove that the configuration is not stuck, we need to prove that each of the executing threads can either perform a step or *block* predicate holds for it. Without loss of generality, we choose a random thread from the thread list, namely $n : E[e]$ and show that it is not stuck. By case analysis on the structure of $n : E[e]$ we have the following cases:

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

Case $n : E[(\lambda x. e' \text{ as } \tau\ v)^{\mathsf{par}(\gamma_1)}]$): it suffices to prove that there exists an S', such that S' is equal to $\mathit{transfer}(S, n, n', \gamma_1)$. This is immediate by applying lemma 43 to the assumption that the store is well-typed (obtained by inversion of the configuration typing assumption), n' is a fresh thread identifier (obtained by *E-SN*) and that $\gamma_3 = \gamma_2 \oplus (\gamma \ominus \gamma_1)$ (obtained by performing inversion on the configuration typing derivation, then a subsequent inversion on the thread typing derivation and then lemma 5 applied so as to extract the application typing derivation. Finally, an inversion on the application typing derivation is performed).

Case $n : e$ By applying inversion twice to the *configuration typing* judgement we have that $R; M; \emptyset; \emptyset \vdash e : \tau \& (\delta(n); \emptyset)$. If e is a value then lemma 46 tells us that e is (). We already have that $T = T_1, n : e$ for some T_1. Thus, a single step can be performed via rule *E-T*. Otherwise, e is not a value. The application of lemma 45 to the latter fact and the typing derivation of e implies that $\exists e_1, E.\ E[e_1] = e$ and $\mathrm{redex}(e_1)$. Thus, $R; M; \emptyset; \emptyset \vdash E[e_1] : \langle\rangle \& (\gamma; \emptyset)$ is also well-typed. The application of lemma 44 to $\mathrm{redex}(e_1)$, the typing derivation of e_1 and the store typing assumption ($M; R; \delta \vdash_{str} S$) yields $\exists S_2, e_2 . S_1, e_1 . S\,; e_1 \to_n S_2; e_2$ or $\mathrm{block}(S, n, e_1, \delta)$ or $\exists v, \gamma_1, e_2, \tau . e_1 \equiv (\lambda x. e_2 \text{ as } \tau\ v)^{\mathsf{par}(\gamma_1)}$. If the first case holds, then a single operational step can be performed by rule *E-S* and thus the configuration is not stuck. If the second case holds, then the configuration is also not stuck (by the definition of not-stuck *ns*). The last case cannot has already been dealt with in the second case of this proof.

Definition 1 *Let z be defined as a metavariable representing maps from region identifiers (e.g. ι) to binary tuples of natural numbers.*

- $\text{sum}(\gamma) = z \quad \textit{if}\ (\forall \iota \in \text{dom}(\gamma).z(\iota) = \text{cap}(\gamma, \iota)) \wedge (\forall \iota \notin \text{dom}(\gamma).z(\iota) = 0, 0)$
- $z \leq z' = \forall \iota \in \text{dom}(z) \cup \text{dom}(z').z(\iota) \leq z'(\iota)$
- $\text{cnt}(S, n) = z \quad \textit{if}\ (\forall \iota.\text{thread\_live}(S, \iota, n) \Rightarrow z(\iota) = \text{cap}(S, \iota, n)) \wedge (\forall \iota.\neg\text{thread\_live}(S, \iota, n) \Rightarrow z(\iota) = 0, 0)$
- $\text{owns}(S, \gamma, n) = \forall \iota \in \text{dom}(\gamma).(\iota : (\theta, H, S') \in \text{bflatten}(S)) \wedge n \in \text{dom}(\theta)$

Lemma 42 (Progress — Spawn Helper 1) $\text{sum}(\gamma_1) \leq \text{cnt}(S, n) \wedge \text{owns}(S, \gamma_1, n) \wedge \text{pure\_once}(\gamma_1) \Rightarrow \text{transfer}(S, n, n', \gamma_1)$ *definable.*

Proof. Proof by induction on the structure of γ_1.

- $\gamma_1 = \emptyset$: Then $transfer$ trivially holds.
- $\gamma_1 = \gamma', \iota^{n_1, n_2} \triangleright \pi$: it suffices to prove that
  - $\iota : (\emptyset, \theta, s \mapsto n_3, n_4, H, S') \in \text{flatten}(S)$: this is immediate by the assumption that $\text{owns}(S, \gamma_1, n)$.
  - $n_1 \leq n_3$ and $n_2 \leq n_4$: this is immediate by the assumption that $\text{sum}(\gamma_1) \leq \text{cnt}(S, n)$.
  - $\text{transfer}(S(\iota, \theta, n' \mapsto n_3, n_4, H, S'), n, n', \gamma')$: $\text{pure\_once}(\gamma_1)$ implies $\text{pure\_once}(\gamma')$. $\text{pure\_once}(\gamma_1)$ and $\text{sum}(\gamma_1) \leq \text{cnt}(S, n)$ imply that $\text{sum}(\gamma') \leq \text{cnt}(S(\iota, \theta, n' \mapsto n_3, n_4, H, S'), n)$. The assumption that $\text{owns}(S, \gamma_1, n)$ and $\text{pure\_once}(\gamma_1)$ imply $\text{owns}(S(\iota, \theta, n' \mapsto n_3, n_4, H, S'), \gamma', n)$. The application of the induction hypothesis to the latter facts completes the proof.

Case $\gamma_1 = \gamma', \iota^{\overline{n_1+n_2, n_3}} \triangleright \pi$: Similar proof to the previous case.

Lemma 43 (Progress — Spawn) $M; R; \delta, n \mapsto \gamma \vdash S \ \wedge\ \xi \vdash \gamma_3 = \gamma_2 \oplus (\gamma \ominus \gamma_1) \ \wedge\ n'$ *fresh thread identifier* $\Rightarrow\ S' = \text{transfer}(S, n, n', \gamma_1)$

Proof. The store typing assumption implies (trivial) that $\text{owns}(S, n, \gamma)$, $\text{sum}(\gamma) \leq \text{cnt}(S, n)$ and $\text{pure\_once}(\gamma)$. The capability addition assumption implies that $\text{sum}(\gamma_1) \leq \text{sum}(\gamma)$ and $\text{dom}(\gamma_1) \subseteq \text{dom}(\gamma)$. Therefore, $\text{sum}(\gamma_1) \leq \text{cnt}(S, n)$, $\text{owns}(S, n, \gamma_1)$ and $\text{pure\_once}(\gamma_1)$ hold. The proof is completed by applying lemma 42 to $\text{owns}(S, n, \gamma_1)$, $\text{pure\_once}(\gamma_1)$ and $\text{sum}(\gamma_1) \leq \text{cnt}(S, n)$.

Lemma 44 (Progress — Expressions) $\text{redex}(e) \wedge R; M; \emptyset; \emptyset \vdash e : \tau \,\&\, (\delta(n); \gamma') \wedge R; M; \delta \vdash S \Rightarrow \text{block}(S, n, e', \delta) \vee (\exists S', e'. S; e \rightarrow_n S'; e') \vee (\exists e_1, \tau, v, \gamma''. e \equiv (\lambda x. e_1\ \texttt{as}\ \tau\ v)^{\mathsf{par}(\gamma'')})$

Proof. We proceed by performing a case analysis on the typing derivation of e:

Case *T-I*, *T-U*, *T-F*, *T-L*, *T-R*, *T-RF*,*T-V*: the proof is immediate as e is a value and this contradicts the assumption that $\text{redex}(e)$.

Case *T-AP*,*T-RP*: The proof is a straightforward application of canonical forms lemma and the operational rules *E-A*, *E-RP* respectively. In the case of *T-AP* we may have that e is equal to a parallel application term. In that case the second case of the conclusion holds.

Case *T-NRG*: The application of lemma 46 to the typing derivation of v_1, which is obtained by inverting *T-NG*, yields that $v_1 \equiv \texttt{rgn}_m$. It suffices to show that the premise of *E-NR*, namely $(S_1, \jmath) = \text{newrgn}(S, n, m)$ is satisfied. This can be shown by applying lemma 48 to store typing derivation $M; R; \delta \vdash_{str} S$, which is obtained by the assumption of the progress theorem, and the fact that $\delta = \delta, n \mapsto \gamma', m^{\kappa} \triangleright \pi$, which is obtained by the premise of rule *T-NRG* that requires that m belongs in the domain of γ.

Case *T-NR*:The application of lemma 46 to the typing derivation of v_2, which is obtained by inverting *T-NR*, yields that $v_1 \equiv \texttt{rgn}_k$. To perform a single step, we need to prove that *E-NR* applies. We already have that the term has the appropriate form, thus it suffices to prove the premise of *E-NRG*, which is $(S_1, \ell) = \text{alloc}(\jmath, S, v_1)$. By inversion of *T-NR* we have that $R; M; \emptyset; \emptyset \vdash v_1 : \tau\&(\gamma; \gamma)$, and $\jmath \in \text{dom}(\gamma)$. By applying lemma lemma 35 on the derivation of v_1 we obtain $R; M; \emptyset; \emptyset \vdash v_1 : \tau\&(\emptyset; \emptyset)$. By applying lemma 47 to the latter derivation as well as $\jmath \in \text{dom}(\gamma)$ and the store typing $M; R; \delta, \jmath \mapsto \gamma \vdash_{str} S$ we have that $(\ell, S_1) = \text{alloc}(\jmath, S, v_1)$.

Case *T-D*: The application of lemma 46 to the typing derivation of v, which is obtained by inverting *T-D*, yields $v \equiv \texttt{loc}_\ell$. Further, inversion of the typing derivation of $\texttt{loc}_\ell$ yields that $\ell \in dom(M)$ and $\vdash R; M; \emptyset; \emptyset; \gamma; \gamma$, where γ is the effect of thread ι. By applying lemma 49 to the store typing assumption, where $\ell \in \text{dom}(M)$ and $\delta(\iota) = \gamma$, and the fact that ι belongs in $\text{accessible}(\gamma)$, we have that $\exists v' = \text{lookup}(S, \ell, \iota)$. 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 49 to prove premise of *E-AS*).

Case *T-CP*: The application of lemma 46 to the typing derivation of v_1, which is obtained by inverting *T-NR*, yields that $v_1 \equiv \texttt{rgn}_\iota$. By inversion of rule *T-CP* we have that region ι is *live*. It suffices to prove that the operational semantics performs a step or block(S, ι, e, δ) holds. We proceed by examining all possible values of η:

- *rg*±: *block* predicate does not hold in this case and thus it must be shown that the semantics performs a step. The store typing assumption and the fact that ι is *live* (as shown earlier) tells us that region ι imply that thread_live(S, ι, n). Thus, it possible to decrement or increment its region count once. The liveness fact also satisifes the only precondition of *updcap* premise of the operational rule *E-C*, thus the semantics performs a single step.
- *lk*−: the premise of typing rule *T-CP* tells us that region ι is *live* and *accessible* (positive lock capability). As in the previous case the store typing implies that thread_live(S, ι, n). The *canlk* premise of *updcap* function is trivially satisfied by the store typing premise mutex_ok and the fact that ι is *accessible*.
- *lk*+ and the lock capability of region ι in thread ι is positive: Similar to the previous case.
- *lk*+ and ι is *inaccessible*: As in the previous case the store typing implies that thread_live(S, ι, n) as a consequence of the fact that ι is *live* in the effect of thread n. If canlk(S, ι, n) does hold then a single step can be performed via rule *E-C*. Otherwise, block(S, n, e, δ) holds as a consequence of the aboce facts and the proof is completed.

Lemma 45 (Expression — Redex) $R; M; \Delta; \Gamma \vdash e : \tau_1 \,\&\, (\gamma_1;\gamma_2) \wedge e \not\equiv v_1 \Rightarrow \exists\, e', E.E[u] \equiv e \wedge \text{redex}(e)$

Proof. Straightforward proof by induction on the typing derivation.

- *T-I*, *T-U*, *T-F*, *T-L*, *T-R*, *T-RF* then the proof is immediate as e is a value.
- *T-V*: Immediate as it holds for $E \equiv \square$ and $u \equiv x \not\equiv v$.
- *T-NR*: By observing the shape of the expression of *T-NR* typing derivation, $e \equiv \texttt{new}\, e_1 \texttt{ at } e_2$. If e_1 and e_2 are both values then the proof is immediate ($E \equiv \square$ and $u \equiv \texttt{new}\, e_1 \texttt{ at } e_2$). Otherwise, if e_1 is not a value the application of the induction hypothesis on the typing derivation of e_1 (obtained from *T-NR* inversion) yields that $\exists\, E[u].E[u] \equiv e_1 \wedge u \not\equiv v_2$. Consequently, $\exists\, E.\texttt{new}\; E[u] \texttt{ at } e_2 \equiv e \wedge u \not\equiv v_2$ or equivalently, $\exists\, E.(\texttt{new}\; E \texttt{ at } e_2)[u] \equiv e \wedge u \not\equiv v_2$. The last case is that e_1 is a value and e_2 is not. By applying similar reasoning we can prove that $\exists\, E.(\texttt{new}\, e_1 \texttt{ at } E)[u] \equiv e \wedge u \not\equiv v_2$.
- *T-AP*, *T-RP*, *T-NG*, *T-CP*, *T-D*, *T-A*: We can perform similar reasoning to prove the remaining cases.

Lemma 46 (Cannonical Forms) $R; M; \Delta; \Gamma \vdash v : \tau \,\&\, (\gamma_1;\gamma_2) \Rightarrow$
$\tau \equiv \langle\rangle \Rightarrow v \equiv () \wedge$
$\tau \equiv \texttt{rgn}(\iota) \Rightarrow (v \equiv \texttt{rgn}_\iota \wedge \iota \in R) \wedge$
$\tau \equiv \texttt{ref}(\tau, \iota) \Rightarrow (v \equiv \texttt{loc}_\ell \wedge \ell \mapsto (\tau, \iota) \in M) \wedge$
$\tau \equiv b \Rightarrow v \equiv n \wedge$
$\tau \equiv \tau_1 \xrightarrow{\gamma_1 \to \gamma_2} \tau_2 \Rightarrow v \equiv \lambda x.e \texttt{ as } \tau_1 \xrightarrow{\gamma_1 \to \gamma_2} \tau_2 \wedge$
$\tau \equiv \forall \rho.\tau \Rightarrow v \equiv \Lambda \rho. f \wedge$

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

Lemma 47 (Progress — Add Location) $M; R; \delta, k \mapsto \gamma, \jmath^\kappa \triangleright \pi \vdash_{str} S \wedge R; M; \emptyset; \emptyset \vdash v : \tau \,\&\, (\emptyset;\emptyset) \wedge \Rightarrow \exists S_1, \ell.(S_1, \ell) =$ alloc$(\jmath, S, v) \wedge \ell \notin dom(M)$

Proof. To prove that *alloc* is defined we only need to show that region $\jmath$ is *live*. This is immediate by inversion of the first assumption.

Lemma 48 (Progress — Add Region) $M; R; \delta, \iota \mapsto \gamma, m^\kappa \triangleright \pi \vdash_{str} S \wedge \Rightarrow \exists S_1, \jmath.(\jmath, S_1) =$ newrgn$(S, \iota, m) \wedge \jmath \notin R$

Proof. To prove that *newrgn* is defined we only need to show that region $\jmath$ is *live*. This is immediate by inversion of the first assumption.

Lemma 49 (Progress — Lookup Value) $M, \ell \mapsto (\tau, \jmath); R; \delta, n \mapsto \gamma \vdash_{str} S \wedge \jmath \in$ accessible$(\gamma) \Rightarrow \exists v.v =$ lookup(S, ℓ, n)

Proof. To prove that *newrgn* is defined we need to show that region $\jmath$ is *live*, where $\jmath$ is the region that contains ℓ in its heap, *canlk*$(S, \jmath, n)$ *only* holds for thread n, and ℓ and $\jmath$ exist in store S. The first and last obligations are trivial by store typing assumption. In particular, the first obligation is satisfied by the premise *counts_ok* of the store typing assumption, whereas the last obligation is immediate by the store typing premise that all locations in $M, \ell \mapsto (\tau, \jmath)$ exist in S. The last obligation is immediate by the store typing premise *mutex_ok* and the assumption that $\jmath$ is accessible in the effect of thread n.