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 [Tøfte 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 [Tøfte 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 ρ, h at heap in  // {ρ\textsuperscript{1,1}¬ρ_H}
  let z = new 10 at h in
  ...
  z := deref z + 5;
  ...
  free h;       // {} — empty effect, ρ is no longer alive
  ...
```

The comments on the right-hand side of the example’s code show the current effect. An effect is roughly a set of capabilities that are held at a given program point. Right after creation of region ρ, the entry ρ\textsuperscript{1,1}¬ρ_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\textsubscript{1} within the heap. Other regions can be then allocated within ρ\textsubscript{1}, e.g. ρ\textsubscript{2}; this can be done by passing the handle of ρ\textsubscript{1} to the region creation construct.

Similarly, regions ρ\textsubscript{3} and ρ\textsubscript{4} can be allocated within region ρ\textsubscript{2}.

```
newrgn ρ\textsubscript{1}, h at heap in  // {ρ\textsuperscript{1,1}¬ρ_H}
  ...
newrgn ρ\textsubscript{2}, h\textsubscript{1} at h\textsubscript{1} in  // {ρ\textsuperscript{1,1}¬ρ_H, ρ\textsuperscript{2,1}¬ρ\textsubscript{1}}
  ...
newrgn ρ\textsubscript{3}, h\textsubscript{2} at h\textsubscript{2} in  // {ρ\textsuperscript{1,1}¬ρ_H, ρ\textsuperscript{2,1}¬ρ\textsubscript{1}, ρ\textsuperscript{3,1}¬ρ\textsubscript{2}}
  ...
newrgn ρ\textsubscript{4}, h\textsubscript{2} at h\textsubscript{2} in  // {ρ\textsuperscript{1,1}¬ρ_H, ρ\textsuperscript{2,1}¬ρ\textsubscript{1}, ρ\textsuperscript{3,1}¬ρ\textsubscript{2}, ρ\textsuperscript{4,1}¬ρ\textsubscript{2}}
  ...
```

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\textsubscript{1}, in the lexical scope of region ρ\textsubscript{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 ρ₂ within the innermost scope; regions ρ₃ and ρ₄ are also deallocated. They are all removed from the current effect and thus are no longer accessible.

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 = λρ₁. Aheap.
   while (true) do
      newrgn ρ₁, h at heap in // {ρ₁, h} ⊲ ρ₁
         ... newrgn ρ₂, h₂ at h₁ in // {ρ₁, ρ₂} ⊲ ρ₁
            ... newrgn ρ₃, h₃ at h₂ in // {ρ₁, ρ₂, ρ₃} ⊲ ρ₃
               newrgn ρ₄, h₄ at h₂ in // {ρ₁, ρ₂, ρ₃, ρ₄} ⊲ ρ₄
                  ... free h₄; // {ρ₁, ρ₂, ρ₃} ⊲ ρ₃
                     ... ρ₂, ρ₃ and ρ₄ are no longer alive
```

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,
It assumes that both regions are already locked and remain locked when the function returns.

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 $\rho_1$ and $\rho_2$, and swaps their contents. It assumes that both regions are already locked and remain locked when the function returns.

In order to instantiate $\rho_1$ and $\rho_2$ with the same region $\rho$, we can create two lock capabilities by using the lock operator twice on $\rho$’s handle $h$. Of course, the second use of lock will succeed immediately, as the region has already been locked by the same thread.
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).

```python
def swap = λρ₁, ρ₂. (h₁ : rgn(ρ₁), h₂ : rgn(ρ₂)). x : ref(ρ₁, int), y : ref(ρ₂, int)). // ρ₁ and ρ₂ are unlocked
  lock h₁;
  let z = deref x in
    lock h₂;
    x := deref y;
    unlock h₁;
    y := z;
    unlock h₂
```

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 h₁`) 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 `ρ₁` and `ρ₂`, which are both locked. It lets `ρ₁` migrate to a new thread and passes `x` as a parameter. It then assigns a value to `y`.

```python
def bad = λρ₁, ρ₂. λ(x : ref(ρ₁, int), y : ref(ρ₂, int)). // ρ₁ and ρ₂ must be both locked
  spawn f[ρ₁](x);
  y := 7
```

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

```python
swap[ρ][ρ](h, h, a, b);
```

The cause of the unsoundness is that, in this last call to `swap[ρ][ρ]`, we allowed a single capability `ρ²→ρ_H` to be divided in two distinct capabilities `ρ¹→ρ_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, `ρ²→ρ_H` is a pure capability; when we divide it we obtain two impure halves, which we denote as `ρ¹⁺→ρ_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.
### 4 Language Description

The syntax of the language is illustrated in Figure 1. The language core comprises of variables (x), constants (e), functions, and function application. Functions can be region polymorphic (λp.f) and region application is explicit (e[p]). Monomorphic functions (λx.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(y) for spawning a new thread. 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 r1 at r2, where r1 is the value that will be placed in the cell and r2 is a handle of the region in which the new cell will be allocated. Standard assignment and dereference operators complete the picture.

The construct new r1 in r2 allocates a new region r and binds x to the region handle. The new region resides in a parent region, whose handle is given in e1. The scope of r and x is e2, 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 cap 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., r1g+ (meaning that the region count is to be incremented) or l1k− (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 (n1, n2) is called a pure capability, whereas a capability of the form (n1, n2) is called an impure capability. In both cases, it is implied that the current thread can decrement the region count n1 times and the lock count n2 times. Impure capabilities are obtained by splitting pure or other impure capabilities into several pieces, e.g., (3, 2) = (2, 1) + (1, 1), in the same spirit as fractional capabilities. 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.

---

**Figure 1: Syntax.**

<table>
<thead>
<tr>
<th>Value</th>
<th>v ::= f</th>
<th>c</th>
<th>rgn,</th>
<th>loc,</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function</td>
<td>f ::= λx.e as rgn f</td>
<td>Λp.f</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Expression</td>
<td>e ::= x</td>
<td>c</td>
<td>f</td>
<td>(e) e'</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Type</td>
<td>τ ::= b</td>
<td>rgn τ</td>
<td>ref τ</td>
<td>rgn(r)</td>
</tr>
<tr>
<td>Effect</td>
<td>γ ::= 0</td>
<td>γ, r ⇒ π</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Calling mode</th>
<th>ξ ::= seq</th>
<th>par(y)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Capability op</td>
<td>η ::= ψ+</td>
<td>ψ−</td>
</tr>
<tr>
<td>Capability kind</td>
<td>ψ ::= rgn</td>
<td>lk</td>
</tr>
<tr>
<td>Capability</td>
<td>κ ::= n,n</td>
<td>n,n</td>
</tr>
<tr>
<td>Region parent</td>
<td>π ::= r</td>
<td>⊥</td>
</tr>
<tr>
<td>Region</td>
<td>r ::= ρ</td>
<td>t</td>
</tr>
</tbody>
</table>

---

1In the examples of Section 3, we used more intuitive notation: we omitted seq and used the keyword spawn instead of par.
2The region manipulation operators used in Section 3 are simple abbreviations: share ≡ capρ, unloc ≡ capρ, etc.
### 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 $T$. Each thread in $T$ is of the form $n : e$, where $n$ is a thread identifier and $e$ is an expression. The store is a list of regions of the form $i : (θ, H, S)$, where $i$ 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 $i$, 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 $\square$. 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 $(\varepsilon)$, 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 $\text{transfer}(S, n, n', γ_1)$ 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

---

3The 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.
thread map associated with r, or with some ancestor of r, maps n to a positive lock count.

- alloc(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 j and is set to value v. Region 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, j) is used in rule E-NR to create a new region in S. The new region is allocated within j, which must be live. Its thread map is set to n → 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, η, j, n) is used in rule E-C. This operation updates S by modifying the region or lock count of thread n for region 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.


![Figure 4: Expression evaluation relation S; e ↠ S’; e’](image)

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 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 rπ, denoting that region r is associated with capability κ and has parent π, which can be another region, ⊥, or ?. Regions whose parents are ⊥ or ?
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 \( \bot \). 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 \( \rho \).

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 \( \text{unit} \) base type, which we denote by \( \langle \rangle \). Region handle types \( \text{rgn}(r) \) and reference types \( \text{ref}(r, r) \) are associated with a type-level region \( r \). Monomorphic function types carry an \( \text{input} \) and an \( \text{output effect} \). A well-typed expression \( e \) has a type \( \tau \) under an input effect \( \gamma \) and results in an output effect \( \gamma' \). The typing relation (see Figure 5) is denoted by \( R; M; \Delta; \Gamma \vdash e : \tau \& (\gamma; \gamma') \). It then joins the remaining effect with the function’s output effect \( \gamma_2 \). In the case of parallel application, rule \( \text{T-AP} \) also requires that the return type is \( \text{unit} \).

The splitting and joining of effects is controlled by the judgement \( \xi \vdash \gamma'' = \gamma_2 \oplus (\gamma \oplus \gamma_1) \), 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 \( \gamma'' \);
- the consistency of \( \gamma \) and \( \gamma'' \), i.e., regions cannot change parent and capabilities cannot switch from pure to impure or vice versa; the domain of \( \gamma'' \) is a subset of the domain of \( \gamma \);


\[
\begin{array}{c}
(r^* \pi) \in \gamma & rg(k) > 0 & \pi \in \{\bot, ?\}
\end{array}
\]

\[
\frac{}{\text{is\_live}(y, r)}
\]

\[
\begin{array}{c}
(r^* \pi) \in \gamma & lk(k) > 0
\end{array}
\]

\[
\frac{}{\text{is\_accessible}(y, r)}
\]

\[
\begin{array}{c}
(r^* \pi) \in \gamma & is\_live(y, r')
\end{array}
\]

\[
\frac{}{is\_live(y, r)}
\]

\[
\begin{array}{c}
(r^* \pi) \in \gamma & is\_accessible(y, r')
\end{array}
\]

\[
\frac{}{is\_accessible(y, r)}
\]

Figure 7: Auxiliary predicates: region liveness and accessibility.

\[
\begin{array}{l}
rg(k) = n_1 \quad \text{if } k = n_1, n_2 \lor k = \overline{m_1}, \overline{m_2} \\
lk(k) = n_2 \quad \text{if } k = n_1, n_2 \lor k = \overline{m_1}, \overline{m_2} \\
dom(\gamma) = \{ r \mid (r^* \pi) \in \gamma \} \\
live(\gamma) = \{ r^* \pi \mid (r^* \pi) \in \gamma \land \text{is\_live}(y, r) \} \\
is\_pure(k) = \exists n_1. \exists n_2. k = n_1, n_2 \\
consistent(\gamma_1; \gamma_2) = \left( \forall (r^* \pi) \in \gamma_1. \forall (r^* \pi') \in \gamma_2. \pi = \pi' \land (is\_pure(k) \Leftrightarrow is\_pure(k')) \right) \\
\quad \land \text{dom}(\gamma_1) \subseteq \text{dom}(\gamma_2) \land \text{live}(\gamma_1) = \gamma_1 \land \text{live}(\gamma_2) = \gamma_2 \\
abs\_par(\gamma_1; \gamma_2) = \left\{ r \mid (r^* \pi') \in \gamma_1 \land (r^* ?) \in \gamma_2 \right\}
\end{array}
\]

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., \(\rho\) and \(x : rgn(\rho)\) are appended to \(\Delta\) and \(\Gamma\) 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 \(\rho\). This implies that \(\rho\) 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 \(\eta\). 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 \(\text{cap}\_lk\) must be suspended until the lock can be obtained. On the other hand, the evaluation of \(\text{cap}\_rg\) 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 \(\delta\) 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.

\[
\begin{align*}
R; M; \emptyset \vdash_T \emptyset & \quad\quad R; M; \emptyset \vdash \gamma & \quad\quad R; M; \emptyset; \emptyset \vdash e : \langle \rangle & \quad\quad n \notin \text{dom}(\delta) & \quad\quad R; M; \delta, n \mapsto \gamma \vdash_T T, n : e
\end{align*}
\]

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

- **Region consistency**: the set of region names occurring in the co-domain of \( \delta \) 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 \( \delta \) for a particular region \( j \). Additionally, only this thread is allowed to access or lock sub-regions of \( j \).

**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 \( \delta \),
- the set of region names in \( S \) is equal to \( R \),
- the set of locations in \( M \) is equal to the set of locations in \( S \), and
- for each location \( \ell \), 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 \( S; T \leadsto S'; T' \), then there exist \( R' \supseteq R \), \( M' \supseteq M \) and \( \delta' \) such that the resulting configuration is well-typed with \( R'; M'; \delta' \vdash_{C} S'; T' \).

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 \( \text{cap}_{lk}, \text{rgn}_{j} \) and \( j \) is a live but inaccessible region to thread \( n \), or
• $e$ is of the form $(\lambda x. e_1 \text{ as } \tau \triangleright \text{par}(\gamma))$ or

• there exist $S'$ and $e'$ such that $S; e \xrightarrow{n} S'; e'$.

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 $R; M; \delta[n] \vdash \gamma \triangleright S$. If the operational semantics takes a step $S; e \xrightarrow{n} S'; e'$, then there exist $R' \supseteq R$, $M' \supseteq M$ and $\gamma'$ such that the resulting expression and the resulting store are well-typed with $R'; M'; \emptyset; \emptyset \vdash e' : \tau \& (\gamma'; \gamma''')$ and $R'; M'; \delta[n] \vdash \gamma' \triangleright S'$.

The type safety theorem is a direct consequence of Lemmata 1 and 2. Let function $\text{main}$ be the initial program, let $\iota_H$ be global heap region and let the initial typing contexts $R_0$ and $\delta_0$ and the initial program configuration $S_0$; $T_0$ be defined by the following singleton lists:

\[
R_0 = \{ \iota_H \} \\
\delta_0 = \{ 1 \mapsto \iota_H \triangleright \bot \} \\
\theta_0 = \{ 1 \mapsto 1, 0 \} \\
S_0 = \{ \iota_H : (\theta_0, \emptyset, \emptyset) \} \\
T_0 = \{ 1 : (\text{main}; \iota_H, \text{rgn}_H) \triangleright \text{seq} \}
\]

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

The empty (except for $R_0$ that contains only $\iota_H$) contexts that are used when typechecking the initial configuration $S_0; T_0$ guarantee that all functions in the program are closed and that no explicit region values ($\text{rgn}_r$) or location values ($\text{loc}_l$) 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 \cite{tofte1994}. Since then, several memory-safe systems that enabled early region deallocation for a sequential language were proposed \cite{aiken1995}. Henglein et al. \cite{henglein2001}, Walker and Watkins \cite{walker2001}, Fluet et al. \cite{fluet2006}. Cyclone \cite{grossman2002} and RC \cite{gay2001} 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 \cite{grossman2003}. 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 \cite{boyapati2003, zhao2004, zhao2008} for real-time Java to rule out dynamic checks imposed by the language specification. Boyapati et al. \cite{boyapati2003} 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 \cite{boyapati2002}; 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 \cite{zhao2004}. Their main advantage is that programs require fewer annotations compared to programs with explicit region constructs. In the same track, Zhao et al. \cite{zhao2008}
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 hierarchies 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 employing 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


Appendix

Language Syntax & Substitution Relation

Value
\[
\begin{aligned}
\nu &::= f | c | \text{rgn} | \text{loc}_i \\
\rho_1[r/p] &::= r \\
\nu &::= \text{def} | \text{newrgn}_\rho, x \in e \in \text{cap}_\rho e | \text{rgn}
\end{aligned}
\]

Expression
\[
\begin{aligned}
e &::= x | c | f | (e f) | e[r] | \text{new} e \text{at} ee | e::= e | \text{loc}_i, x \in e \\
\text{def} &::= \text{newrgn}_\rho, x \in e \in \text{cap}_\rho e | \text{rgn}
\end{aligned}
\]

Capability kind
\[
\begin{aligned}
\psi &::= \text{rg} | \text{lk} \\
\rho &::= \text{rg} | \text{lk} \\
\eta &::= \psi + | \psi - \\
\kappa &::= n, n \in \overline{m} \\
\pi &::= r \cup \downarrow ? \\
\tau &::= b | \tau \mapsto r | \forall \rho, \tau | \text{ref}(\tau, r) | \text{rgn}(r)
\end{aligned}
\]

Region parent
\[
\begin{aligned}
\psi &::= \text{rg} | \text{lk} \\
\rho &::= \text{rg} | \text{lk}
\end{aligned}
\]

Type
\[
\begin{aligned}
\tau &::= b | \tau \mapsto r | \forall \rho, \tau | \text{ref}(\tau, r) | \text{rgn}(r) \\
\nu &::= f | c | \text{rgn} | \text{loc}_i \\
\nu &::= \text{def} | \text{newrgn}_\rho, x \in e \in \text{cap}_\rho e | \text{rgn}
\end{aligned}
\]

Capability op
\[
\begin{aligned}
\psi &::= \psi + | \psi - \\
\kappa &::= n, n \in \overline{m} \\
\pi &::= r \cup \downarrow ? \\
\tau &::= b | \tau \mapsto r | \forall \rho, \tau | \text{ref}(\tau, r) | \text{rgn}(r)
\end{aligned}
\]

Effect
\[
\begin{aligned}
\psi &::= \emptyset | \eta, r \mapsto \pi \\
\nu &::= f | c | \text{rgn} | \text{loc}_i
\end{aligned}
\]

Function
\[
\begin{aligned}
f &::= \lambda x. e \in \tau \mapsto r | \forall \rho, f \in \text{rgn}(r) \\
\nu &::= f | c | \text{rgn} | \text{loc}_i \\
\nu &::= \text{def} | \text{newrgn}_\rho, x \in e \in \text{cap}_\rho e | \text{rgn}
\end{aligned}
\]

Calling mode
\[
\begin{aligned}
\xi &::= \text{seq} | \text{par}(\gamma) \\
\nu &::= f | c | \text{rgn} | \text{loc}_i \\
\nu &::= \text{def} | \text{newrgn}_\rho, x \in e \in \text{cap}_\rho e | \text{rgn}
\end{aligned}
\]

Operational Semantics

\[
bflatten(S) \equiv \begin{cases} 
\emptyset & \text{if } S = \emptyset \\
\{ t : (\theta, H, S'), bflatten(S'), bflatten(S''') \} & \text{if } S = t : (\theta, H, S'), S'''
\end{cases}
\]

\[
\text{thread\_live}(S, t, m) \equiv \{ f : (\theta, H, S') \in bflatten(S) \in \{ j | \text{dom}(S') \Rightarrow \exists n_1, n_2, (m \mapsto n_1, n_2) \in \theta \land n_1 > 0 \}
\]

\[
live(S, t) \equiv \exists n_1, n_2 \cdot \text{thread\_live}(S, t, m)
\]

\[
dom(S, t) \equiv \{ t | (t \mapsto v) \in H \equiv (j : (\theta, H, S')) \in bflatten(S) \}
\]

\[
\text{dom}(S) \equiv \{ t | (t : (\theta, H, S)) \in bflatten(S) \}
\]

\[
\text{flatten}(S) \equiv \{ t : (\theta', H', S') | (t : (\theta', H', S')) \in bflatten(S') \land \text{live}(S, t) \}
\]

\[
S(\eta, H, S') \equiv \begin{cases} 
S_1, \eta : (\theta, H, S'), S_1 & \text{if } S = S_1, \eta : (\theta, H, S), S_3 \\
S_2, j : (\theta_1, H_1, S_1) & \text{if } \eta \in \text{dom}(S) \land S = S_2, j : (\theta_1, H_1, S_1) \\
\end{cases}
\]

\[
\text{newrgn}(S, n, j) \equiv \begin{cases} 
S(\eta, H, S') = (n \mapsto 1, 1, 0, 0) & \text{if } k \notin \text{dom}(S) \land j : (\theta, H, S') \in \text{flatten}(S) \\
\end{cases}
\]

\[
\text{alloc}(S, v, \nu) \equiv \begin{cases} 
S(\eta, H, \ell \mapsto v, S') & \text{if } \ell \notin \text{dom}(S) \land j : (\theta, H, S') \in \text{flatten}(S) \\
\end{cases}
\]

\[
\text{canlk}(S, t, n) \equiv \begin{cases} 
\forall \nu' : (\eta', H', S'') \in \text{flatten}(S) \land \text{canlk}(S, t, n) \land \text{canlk}(S, t, n') \land \forall \nu' \neq n \sim \text{canlk}(S, t, n') \\
\end{cases}
\]

update(S, f, x, n) \equiv \begin{cases} 
S(\eta, H, \ell \mapsto x, S') & \text{if } (t : (\theta, H, \ell) \mapsto \text{lookup}(S, f, x)) \in \text{flatten}(S) \\
\end{cases}
\]

\[
\text{lookup}(f, l, v, S') \equiv \begin{cases} 
\text{def} & \text{if } y = \emptyset \\
\text{def} & \text{if } y = y', r^{\rho_1, \eta \mapsto \pi} \land \eta : (0, \theta, s \mapsto n_1, n_2, H, S') \in \text{flatten}(S) \\
\end{cases}
\]

\[
\text{transfer}(S, \eta, H, S') \equiv \begin{cases} 
\text{def} & \text{if } y = y', r^{\rho_1, \eta \mapsto \pi} \land \eta : (0, \theta, s \mapsto n_1, n_2, H, S') \in \text{flatten}(S) \\
\end{cases}
\]

\[
\text{updcnt}(n_1, n_2, \eta) \equiv \begin{cases} 
\eta = \text{rg}\eta & n_1 = n_2, n_2 = n_1 \\
\text{if } \eta \neq \text{rg}\eta & n_1 = n_2 \neq n_1 \\
\end{cases}
\]

\[
\text{updcap}(S, \psi, \eta) \equiv \begin{cases} 
S' & \text{if } (t : (\theta, H, S')) \in \text{flatten}(S) \land \theta' = 0[n \mapsto \text{updcnt}(\theta(n), \eta)] \land S' = S(\eta, H, S') \land (\eta = \text{rg}\eta \Rightarrow \text{canlk}(S, t, n))
\end{cases}
\]
\[\vdash R; M; \Delta; \Gamma; b; \gamma; \gamma \quad (T-V) \]
\[\vdash R; M; \Delta; \Gamma \vdash n : b \quad (T-I) \]
\[\vdash R; M; \Delta; \Gamma \vdash \lambda x.x \quad (T-R) \]
\[\vdash R; M; \Delta; \Gamma; b \quad (T-L) \]
\[\vdash R; M; \Delta; \Gamma \vdash \text{cap}_v e_l : \gamma \quad (T-CP) \]
\[\vdash R; M; \Delta; \Gamma \vdash \text{rngr} : \text{rngr}(\gamma) \quad (T-CP) \]
\[\vdash R; M; \Delta; \Gamma \vdash \tau_\Delta \quad (T-F) \]
\[\vdash R; M; \Delta; \Gamma \vdash \text{f} : \tau \quad (T-RF) \]
\[\vdash R; M; \Delta; \Gamma \vdash \text{ref}(\tau, r) \quad (T-AP) \]
\[\vdash R; M; \Delta; \Gamma \vdash \text{new} e_1 \quad (T-NR) \]
\[\vdash R; M; \Delta; \Gamma \vdash e_1 \quad (T-A) \]
\[\vdash R; M; \Delta; \Gamma \vdash \text{deref} e : \tau \quad (T-D) \]
\[\vdash \xi \vdash \gamma = \gamma_1 \quad (ESJ) \]
\[\vdash \xi \vdash \gamma = \gamma_1 \quad (ES-C) \]

**Type Safety Judgements**

**Program effect**
\[\delta ::= \emptyset \mid \delta, n \mapsto \gamma \]

**Redex**
\[\text{redex}(e) = (\exists S, S', e', n. S; e \rightarrow_S S'; e') \vee (\exists v_1, v_2, \gamma_1, (v_1, v_2)^{\text{par}(\gamma_1)}) \]

**Store**
\[S(l) = \text{v} \quad \text{if } (i : (\theta, H, \Gamma \mapsto v, S')) \in \text{flatten}(S) \]
\[\text{dom}(\Gamma) = \{ x \mid (x : \tau) \in \Gamma \} \]
\[\text{dom}(M) = \{ (i) \mid (i \mapsto (\tau, i)) \in M \} \]
\[\text{pure_once}(y) = \forall l \in \text{dom}(y). \forall n_1, n_2, \gamma, \gamma_1 = \gamma, \gamma, \gamma', \gamma_2, n_1 = \gamma \]
\[\text{cap}(S, i, m) = \left\{ \begin{array}{l} n_1, n_2 \quad \text{if } (i : (\theta, m \mapsto n_1, n_2, H, S')) \in \text{flatten}(S) \end{array} \right. \]
\[\text{cap}(\gamma, i) = \left\{ \begin{array}{l} \gamma_2 \quad \text{if } \gamma = \gamma', \gamma_2 \quad \text{if } \gamma = \gamma', \gamma \end{array} \right. \]
\[\text{count}(S, n, \delta) = n_1 \geq n, n_2 \geq n_4 \]
\[\text{counts}(\delta, S) = \forall (m, \gamma, \gamma', \gamma_2, \delta) \in \Delta \text{thread live}(S, i, m) \geq \text{cap}(\gamma, \delta) \quad \text{is pure}(\gamma) \Rightarrow \gamma \not\in \text{dom}(\gamma) \]
\[\text{mutex}(S, n, \delta) = \delta = \delta_1 \mid m \mapsto \gamma \quad \gamma \rightarrow \gamma_2 \quad \text{link}(S, i, m) \geq \gamma \rightarrow \gamma_2 \quad \forall (n' \mapsto \gamma) \rightarrow \gamma_2 \quad \delta \not\in \text{canal}(S, i, n') \]
\[\text{store}(S, n, \delta) = \forall n \in [0, \ldots, \text{dom}(\gamma)] \text{counts}(\delta, S) \quad \text{mutex}(S, n, \delta) \]
\[\text{block}(S, n, e, \delta) = e = E(\text{cap}_v \text{rngr}) \land \delta = \delta_1 \mid n \mapsto \gamma \land \text{is accessible}(\gamma, \delta) \land \text{canal}(S, i, n) \]

**Not Stuck**
\[\forall (n : e) \in T. S(T) \sim S' \land n(e) \not\in T \vee \text{block}(S, n, e, \delta) \]

**Configuration Typing**
\[R; M; \Delta \vdash \gamma \quad \text{if } \Delta \vdash_{n\delta} S : T \]
\[R; M; \Delta \vdash \gamma \quad \text{if } \Delta \vdash_{\delta\tau} S : T \]
\[ n > 0 \quad \delta; S; T \leadsto^{n-1} \delta_{n-1}; S_{n-1}; T_{n-1} \]
\[ \delta_{n-1}; S_{n-1}; T_{n-1} \leadsto \delta_{n}; S_{n}; T_{n} \quad (E-M1) \]
\[ \delta; S; T \leadsto^n \delta; S; T \quad (E-M2) \]

Progress:
\( R; M; \delta \vdash_C S; T \rightarrow \delta \vdash_C S; T \)

Preservation:
\( R; M; \delta \vdash_C S; T \land S; T \leadsto S'; T' \Rightarrow \exists R' \supseteq R, M' \supseteq M, \delta'. \quad R'; M'; \delta' \vdash_C S'; T' \)

Safety:
\( S_0; T_0 \equiv \{ (1 \rightarrow 1, 0, 0, 0) \rightarrow 1 : (f) \{ \text{rgn} \} \} \rightarrow \emptyset; 0 \quad \Rightarrow \emptyset; \delta_{0}^{f} \vdash_C S_{0}; T_{0} \land S_{0}; T_{0} \leadsto \emptyset; S'; T' \Rightarrow \exists \delta'; \delta' \vdash_C S'; T' \)

Expression Progress
\( \text{redex}(e) \land R; M; \emptyset \vdash e : \tau \land \delta(\tau); n \vdash e' \land R; M; \delta +_\tau \vdash S \Rightarrow \text{block}(S, n, e, \delta) \lor (3S' \land e \rightarrow_{n} S' ; e') \lor (\exists \delta', \tau, \nu, \gamma, \epsilon \equiv (\lambda x. e \text{ as } \tau \nu^{\gamma(x)}) ) \)

Expression Preservation
\( R; M; \emptyset \vdash e : \tau \land \delta(\tau); n \vdash S ; e \rightarrow_{n} S' ; S' \land R; M; \delta +_\tau \vdash S \Rightarrow \exists R_{1}, \gamma_{1}, R_{1} \supseteq R \land M_{1} \supseteq M \land R_{1}; M_{1}; \emptyset \vdash e' : \tau \land \delta[n \mapsto \gamma_{1}]; \nu^{\gamma'(x)} \land R_{1}; M_{1}; \delta[n \mapsto \gamma_{1}] +_\tau \vdash S' \)

1 Type Safety Proof

Theorem 1 (Type Safety) Let \( R_{0}, \delta_{0}, S_{0} \) 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 \( S_{0}; T_{0} \leadsto^{n} S_{n}; T_{n} \), 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 \( S; T \leadsto S'; T' \) in \( n \) steps then there exists a closed well-typed configuration such that \( R'; M'; \delta' \vdash_C S'; T' \), where \( R' \land M' \) are supersets of \( R \land M \) respectively.

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 \( S; T \leadsto S'; T' \lor S; T \leadsto^{n-1} S_{n-1}; T_{n-1} \land S_{n-1}; T_{n-1} \leadsto S'; T' \). 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 \( S_{n-1}; T_{n-1} \leadsto S'; T' \).

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 \( S; T \leadsto S'; T' \) for some thread identifier \( i \), then there exist \( \forall R' \supseteq R, M' \supseteq M \land \delta' \) 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 \( R; M; \emptyset, n \mapsto \emptyset +_\nu S \), and the thread context \( R; M; \emptyset, n \mapsto \emptyset +_\tau T, n \mapsto () \). By inversion of the thread typing derivation, we have that \( T \land (R; M; \delta +_\tau T) \) is well-typed. Lemma 4 is applied to the store typing derivation to obtain that store \( S \) is well-typed in the strengthened context \( \delta (R; M; \delta +_\nu S) \). The new store and thread typing derivation give us that the resulting configuration \( (S; T) \) is well-typed in the strengthened context \( \delta (R; M; \delta +_\nu S) \).

Case \( E-S \): By applying inversion twice to the configuration typing judgement we obtain that the input effect of thread \( n : e \) is \( \gamma (\delta, n \mapsto \gamma) \), \( e \) is well-typed \( (R; M; \emptyset \vdash e : \emptyset ) \land \gamma (\emptyset; \emptyset) \) and that the initial store \( S \) is well-typed \( (R; M; \delta +_\nu S) \). 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 \( (R; M; \emptyset \vdash e' : \gamma \gamma' \land \gamma) \). By applying Lemma 22 to the typing derivation of \( e' \), the expression evaluation step \( S; e' \leadsto S'; e'' \), obtained from the inversion of rule E-S and the store \( S \) typing derivation, we obtain that \( e'' \) is also well-typed \( (R' ; M' ; \emptyset \vdash e' \land \gamma \gamma' \land \gamma) \), and the resulting store \( S' \) is also well-typed \( (R' ; M' ; \delta \mapsto \gamma' \mapsto S') \). By applying lemma 6 to the typing derivation of \( e'' \) we have that \( R' \land M' \land \gamma \land \gamma' \). By inversion of the latter derivation we have that \( R' \supseteq M' \). By applying Lemma 3 to \( R' \land M' \land \gamma \land \gamma' \) we have that \( R' ; M' ; \delta +_\tau T, n \mapsto E[e] \). By Lemma 17 we can substitute \( e' \) in the evaluation context \( E \) (all well-typed in \( R' ; M' \)) to obtain \( R' ; M' ; \emptyset \vdash E[e'] \land \gamma (\emptyset; \emptyset) \). We can combine the latter fact with the typing of \( T \), to derive \( R' ; M' ; \delta \mapsto \gamma' \mapsto T, n \mapsto E[e'] \). We have shown that \( R' ; M' ; \delta \mapsto \gamma' \mapsto S' \), thus the latter two facts imply that the configuration \( S'; T, n \mapsto E[e'] \) is well-typed in the typing context \( R' ; M' ; \delta \mapsto \gamma' \mapsto \gamma'' \).

Case \( E-SN \): The program evaluation assumption implies that \( e \equiv E[e'] \), such that \( e' \) is a parallel application reducible, and its premise asserts that \( e' \) is moved to a new thread as a local application reducible \( e'' \). The resulting store \( S' \) is a function of the old store \( S \) and the regions whose ownership is transferred to the new thread (transfer \( (S, n, \gamma', \gamma) \)). By inversion of
If an expression $e$ is well-typed in the typing context $R; M; \delta, n \mapsto \gamma \vdash_T T, n : E[e']$ and $R; M; \delta, n \mapsto \gamma \vdash_S S$ 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 $\text{par}(\gamma_1) + \gamma' = \emptyset \oplus (\gamma \oplus \gamma_1)$ holds, where $\gamma_1$ is the new thread's input effect. It suffices to prove that $R; M; \delta' \vdash_{\omega} S'$ 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 $S$ $(S' = \text{transf}(S, n, n', \gamma_1))$, and $\text{par}(\gamma_1) + \gamma' = \emptyset \oplus (\gamma \oplus \gamma_1)$ 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 $\vdash R; M; \emptyset; \gamma; \gamma'$. Thus, $\vdash R; M; \emptyset; \gamma; \gamma'$ and $R; M; \emptyset; \emptyset = \vdash \gamma \oplus \gamma'$ hold. The application of lemma 17 to the typing derivation of $E[()]$, and $\gamma'$ 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 $e''$ is well-typed in the context $\vdash R; M; \emptyset; \emptyset$ with effect $(\gamma'; \emptyset)$.

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

**Proof.** Proof by induction on the shape of $T$.

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

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

**Lemma 4 (Store Strengthening - Empty $\gamma$)** 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 $\text{store.ok}(\delta, S)$. By inversion of the assumption we have that $\text{store.ok}(\delta, n \mapsto \emptyset, S)$ holds. We unfold the definition of $\text{store.ok}$: for all $i$ in $\bigcup_{(n \mapsto \gamma) \in \text{dom}(\gamma)} \text{counts.ok}(\delta, n \mapsto \emptyset, i, S)$ and $\text{mutex.ok}(\delta, n \mapsto \emptyset, i, S)$. Thus, it suffices to show that for all $i$ in $\bigcup_{(n \mapsto \gamma) \in \text{dom}(\gamma)} \text{counts.ok}(\delta, i, S)$ and $\text{mutex.ok}(\delta, i, S)$ also holds. Both predicates $\text{counts.ok}$ and $\text{mutex.ok}$ depend on threads with non-empty effects. Thus, $\text{counts.ok}(\delta, i, S)$ and $\text{mutex.ok}(\delta, i, S)$ immediately follow from $\text{counts.ok}(\delta, n \mapsto \emptyset, i, S)$, $\text{mutex.ok}(\delta, n \mapsto \emptyset, i, S)$ 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 $\tau$, in the same typing context with effect $(\gamma_1; \gamma_3)$ for some $\gamma_3$.

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

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

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

Case $[(v_1 E'[\{\}] e_2)[e]]$: An equivalent expression for this case is $(v_1 E'[\{\}] e_2, E'[\{\}] e_2)$. By inversion of the typing derivation of $(v_1 E'[\{\}] e_2, E'[\{\}] e_2)$ and $v_1$ are well-typed. In addition, $v_1$ has 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 $E'[\{\}] e_2$ is $\gamma_1$. The application of the induction hypothesis to the latter case implies that $e$ is well-typed for some type $\tau$ with effect $(\gamma_1; \gamma_2)$ for some $\gamma_3$.

Case $[(\text{cap}, E'[\{\}] e_2).[\text{dere f} E'[\{\}] e_2].[(\text{loc}, E'[\{\}] e_2).[[\text{new v at E'}]](\text{new w at E'})] e_2)]$ Similiar 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 \( T\text{-AP} \):

- \( T\text{-A} \): By inversion of the typing derivation of \( e \) we have that \( e_1 \) is well-typed with effect \( (\gamma; \gamma_1) \), \( e_2 \) is well-typed with effect \( (\gamma; \gamma_2) \) and \( \xi \vdash \gamma' = \gamma_2 \iff (\gamma \xor \gamma_2) \). By applying the induction hypothesis to \( e_1 \) and \( e_2 \) we obtain that

\[
\begin{align*}
\vdash R; M; \Delta; \gamma; \gamma_1 \text{ and } \vdash R; M; \Delta; \gamma; \gamma_2 \text{ respectively. It suffices to prove the following obligations:}
\end{align*}
\]

- \( R \vdash M \): immediate by inversion of \( \vdash R; M; \Delta; \gamma; \gamma_1 \).
- \( R; \Delta \vdash \gamma \): immediate by inversion of \( \vdash R; M; \Delta; \gamma; \gamma_2 \).
- \( R; \Delta \vdash \gamma' \): immediate by inversion of \( \vdash R; M; \Delta; \gamma; \gamma_2 \).
- \( R; \Delta \vdash \gamma' \): the effect addition assumption implies that the regions of \( \gamma' \) is a subset of the regions of \( \gamma \). 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) \( \text{If value } v \text{ is well-typed in the typing context } R; M; \Delta; \Gamma, \text{ with effect } (\gamma; \gamma) \text{ and } \vdash R; M; \Delta; \gamma_1; \gamma_2, \text{ then } v \text{ is well-typed in the same typing context with effect } (\gamma_1; \gamma_1) \text{ and } (\gamma_2; \gamma_2). \)

Proof. The proof is trivial, but we provide the key steps behind the proof. The assumption implies that \( \vdash R; M; \Delta; \gamma_1; \gamma_1 \) and also \( \vdash R; M; \Delta; \gamma_2; \gamma_2 \) 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 \( T\text{-L}, T\text{-R}, T\text{-V}, T\text{-F} \)). 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 \( T\text{-RF} \) can be shown trivially by induction (the base case is the same as for rule \( T\text{-F} \)).

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

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

Lemma 9 (R Well-Formedness Weakening) \( R; \Delta \vdash \gamma \land R \subseteq R' \Rightarrow R'; \Delta \vdash \gamma \)

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) \( R; \Delta \vdash \gamma \land R \subseteq R' \Rightarrow R'; \Delta \vdash \gamma \)

Proof. We proceed by performing a case analysis on \( \gamma \):

- \( \emptyset; R'; \Delta \vdash \emptyset \) trivially holds.
- \( \emptyset; R'; \Delta \vdash \gamma' \), \( r \vdash \pi; R'; \Delta \vdash \gamma' \) 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) \( R; \Delta \vdash \tau \land R \subseteq R' \Rightarrow R'; \Delta \vdash \tau \)

Proof. We proceed by performing a case analysis on \( \tau \):

- \( b; R'; \Delta \vdash b \) trivially holds.
- \( \emptyset; R'; \Delta \vdash \emptyset \) trivially holds.
- \( \text{rgn}(\tau); R'; \Delta \vdash r \) holds by lemma 9.
- \( \text{ref}(\tau', r); R'; \Delta \vdash r \) holds by lemma 9. \( R'; \Delta \vdash \tau' \) holds by the induction hypothesis.
- \( \forall p. \tau' \vdash R'; \Delta, p \vdash \tau' \) holds by the induction hypothesis.
- \( \tau' \vdash R'; \Delta \vdash \tau' \) 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) \( R; \Delta \vdash \Gamma \land R \subseteq R' \Rightarrow R'; \Delta \vdash \Gamma \)

Proof. We proceed by performing a case analysis on \( \Gamma \):
- \( \emptyset; \Delta \vdash \emptyset \) trivially holds.
- \( R; \Delta \vdash \Gamma \vdash R'; \Delta \vdash \Gamma' \) holds by the induction hypothesis. \( R'; \Delta \vdash \tau \) holds by lemma 11.

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

Proof. We proceed by performing a case analysis on \( M \):
- \( \emptyset; \Delta \vdash \emptyset \) trivially holds.
- \( R \vdash M', \ell \mapsto (\tau, i); R' \vdash M' \) holds by the induction hypothesis. \( R'; \emptyset \vdash \text{ref}(\tau, i) \) holds by lemma 11.

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

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 \( \vdash R'; M; \Delta; \Gamma; \gamma_1; \gamma_2 \).

- \( T-I \): 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 \( R; \Delta \vdash \tau \). Lemma 9 implies that \( R'; \Delta \vdash \tau \) 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 \( (\ell \mapsto (\tau', i)) \in M \). Thus, we can apply rule \( T-L \) to \( \vdash R'; M; \Delta; \Gamma; \gamma_1; \gamma_2 \) and \( (\ell \mapsto (\tau', i)) \in M \) 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
  - \( \vdash R; M; \Delta; \Gamma; \gamma_1; \gamma_2 \) We have shown that \( \vdash R'; M; \Delta; \Gamma; \gamma_1; \gamma_2 \).
  - \( R; \Delta \vdash \tau; R'; \Delta \vdash \tau \) holds by lemma 11.
  - \( \tau \equiv \tau_1 \) \( \vdash \tau_2 \) \( \vdash \tau_2 \)
  - \( \vdash R; M; \Delta; \Gamma; x : \tau_1 \vdash e' : \tau_2 \vdash \gamma_1; \gamma_2 \): the induction hypothesis is applied to the derivation of \( e' \) to derive that
  - \( R'; M; \Delta; \Gamma; x : \tau_1 \vdash e' : \tau_2 \vdash \gamma_1; \gamma_2 \).
  - \( \vdash \Delta; \Gamma \vdash \lambda x. e' \) as \( \tau : \tau' \) and \( \gamma \gamma \).

We then apply rule \( T-F \) to the above facts to derive \( R'; M; \Delta; \Gamma \vdash \lambda x. e' \)

Case \( T-\text{AP}, T-\text{CP}, T-\text{RP}, T-\text{NG}, T-\text{NR}, T-\text{D}, T-\text{RF}, T-\text{A} \): similar reasoning is performed to prove the remaining cases.

Lemmas 9 and 11 can be used for premises of the form \( R; \Delta \vdash \tau \) and \( R; \Delta \vdash \tau \) respectively.

Lemma 16 (Memory Context Weakening) If expression e is well-typed in the typing context \( R; M; \Delta; \Gamma \) and \( R' \vdash M' \) holds, and \( M' \) is a superset of \( M \), 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 \( \vdash R; M; \Delta; \Gamma; \gamma_1; \gamma_2 \). Thus, we can substitute premise \( R \vdash M \) with \( R \vdash M' \) to obtain \( R; M'; \Delta; \Gamma; \gamma_1; \gamma_2 \).
- T-I: Immediate by applying rule T-I to \( \Gamma \vdash R; \Delta : \Gamma ; \gamma_1; \gamma_2 \).

- T-U: Immediate by applying rule T-U to \( \Gamma \vdash R; \Delta : \Gamma ; \gamma_1; \gamma_2 \).

- T-L: By inversion of this derivation we have that \( \Gamma \vdash \gamma_1; \gamma_2 \).

- T-V: By inversion of this derivation we have that \( \gamma_1; \gamma_2 \) and \( (x : \tau) \in \Gamma \) to complete the proof.

- T-F: By inversion of this derivation we have that
  - \( \Gamma \vdash R; \Delta \); \( \Delta \vdash \gamma_1; \gamma_2 \) holds.
  - \( \Gamma \vdash \tau \)
  - \( \tau \equiv \tau_1 \)
  - \( (x : \tau) \in \Gamma \)
  - R; \( \Delta ; \Delta ; \gamma_1; \gamma_2 \) and \( (\ell \mapsto (\ell', i)) \in M \) complete the proof.

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

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

Case \( \Gamma \vdash \gamma_1 \) then proof is immediate.

Case \( \Gamma \vdash \gamma_2 \) by inversion of the typing derivation of \( \gamma_2 \), we have that \( \Gamma \vdash \gamma_1 \) and \( \gamma_2 \) respectively, then \( E[e_2] \) is also well-typed in the same typing context with effect \( (\gamma_1; \gamma_2) \).

Lemma 18 (Parallel-Sequential typing implication) If a parallel application term is well-typed \( \Gamma \vdash \gamma \), then the corresponding sequential application term \( \Gamma \vdash \gamma \) is also well-typed in the same typing context, with effect \( (\gamma; \gamma') \).

Proof. By inversion of the parallel application typing derivation, \( \Gamma \vdash \gamma \) and \( \gamma \) are well-typed in the same typing context \( \Gamma \vdash \gamma \), with effects \( (\gamma; \gamma') \) and \( (\gamma; \gamma') \) respectively. It also implies that \( \Gamma \vdash \gamma \), \( \gamma \) and \( \Delta : \gamma \) are well-typed in the same typing context with effect \( (\gamma; \gamma) \). We can derive \( \gamma \) then apply \( \Gamma \vdash \gamma \) to the latter facts, we have that \( \Gamma \vdash \gamma \). By applying \( \Gamma \vdash \gamma \) to the latter facts, we have that \( \Gamma \vdash \gamma \) and \( \gamma \) are well-typed in the same typing context with effect \( (\gamma; \gamma) \).

Lemma 19 (Store Typing Preservation — Spawn Helper 2) \( \gamma_1 = \gamma_1', r \mapsto \pi \land S' = \text{transfer}(S, n, n', \gamma_1) \land \text{live}(\gamma_1) = \gamma_1 \Rightarrow \text{thread}_{\text{live}}(S', \ell, \ell') \land \text{cap}(S', \ell, \ell') \geq \text{cap}(\gamma_1, \ell) \).
\[ \begin{align*}
\kappa &= n_1, n_2; \text{ assumption transfer implies that if } t : (\emptyset, \emptyset, n \mapsto n_2, n_2, H, S^*) \in \text{flatten}(S), n_1 \leq n_2, n_2 \leq n_1 \text{ hold, then } \\
t : (\emptyset, \emptyset, n' \mapsto n_2, n_2, H, S^*) \in \text{flatten}(S') \text{ holds. The assumption that live(}y_i) = y_i \text{ implies that } n_1 \text{ is positive and thus } n_2 \text{ is positive. If } \pi \text{ is } \tau, \text{ then thread}\_\text{live}(S', \tau, n') \text{ trivially holds from the latter facts. Otherwise, } \pi \text{ is equal to some } j. \\
\text{The fact that } y_i = \text{live}(y_j) \text{ implies that } y_i = y_j', f \mapsto \pi' \text{ and thus we can apply the induction hypothesis to derive that thread}\_\text{live}(S', j, n'). \text{ The previous facts imply that thread}\_\text{live}(S', \tau, n') \text{ and } \text{cap}(S', \tau, n') \geq \text{cap}(y_j, i). \\
\kappa &= n, n; \text{ similar to the previous case.}
\end{align*} \]

**Lemma 20 (Store Typing Preservation — Spawn Helper 1)**

\[ \text{counts}_{\text{ok}}(\delta, n \mapsto y, i, S) \land \text{par}(y_1) \lor y' = \emptyset \otimes (y \otimes y_1) \land \text{live}(y_1) = y_1 \land S' = \text{transfer}(S, n, n', y_1) \Rightarrow \text{counts}_{\text{ok}}(\delta, n \mapsto y', n' \mapsto y_1, i, S') \]

**Proof.** If \( t \) does not belong in the domain of \( y_1 \), then \( \text{counts}_{\text{ok}}(\delta, n \mapsto y', n' \mapsto y_1, i, S') \) trivially holds from the fact that its counts of in \( S' \) are identical to the ones in \( S \) and the assumption that \( \text{counts}_{\text{ok}}(\delta, n \mapsto y_1, i, S) \) holds. If \( t \) belongs in the domain of \( y_1 \), then it suffices to prove that for all threads \( m \) that belong in the domain of \( \delta, \Sigma \mapsto y', n' \mapsto y_1 \), such that their effect equals to \( y_1', r' \mapsto \pi \) for some \( k', \pi \) and \( y_j \), then the following conditions must hold: \( \text{is}_{\text{pure}}(k') \Rightarrow \emptyset \notin \text{dom}(y_j). \)

- \( m = n' \): it can be trivially shown that \( \text{counts}_{\text{ok}}(\delta, n \mapsto y, i, S) \) implies that \( \text{pure}\_\text{once}(y) \) holds and thus if \( k \) is pure then \( t \) is not contained in \( \text{dom}(y_j) \). The effect addition assumption and the latter fact implies that \( \text{pure}\_\text{once}(y_1) \) holds. The application of lemma 19 completes the proof for this case.

- \( m = n \): if \( k' \) is pure then \( t \) is not contained in \( \gamma' \) (the effect of thread \( n \)). This is immediate by the effect addition assumption. Moreover, \( i \) is impure and is contained in the domain of \( \gamma' \). Thus, \( i \) is also impure in \( y_j \), by the effect addition assumption. Function \text{transfer} does not modify the lock counts of thread \( n \) when \( k' \) is impure. Additionally, the effect addition assumption and the fact that \( i \) belongs in the domain of \( \gamma' \) implies that function \text{transfer} will remove at most \( \text{rg}(\kappa) - 1 \) counts from region \( \gamma \), where \( \kappa \) is the capability of \( i \) in effect \( y \). Thus, the region count for \( t \) for thread \( n \) in \( S' \) will be positive, by the latter fact and the assumption that \( \text{counts}_{\text{ok}}(\delta, n \mapsto y_1, i, S) \). Therefore, \( \text{cap}(S', \gamma_1, i, n) \geq \text{cap}(y_1', r' \mapsto \pi, i) \) holds. The same consideration can be applied inductively for all ancestors of \( i \) so as to derive that thread\_\text{live}(S', i, n) (if there would exist an ancestor of \( i \) that is not live, then \( i \) would not belong in the domain of \( \gamma' \)); this is a contradiction. The effect addition assumption implies that \( \gamma' \) is a subset of \( \gamma \). Thus, the assumption that \( \text{counts}_{\text{ok}}(\delta, n \mapsto y_1, i, S) \) and the latter fact imply that \( \text{pure}\_\text{once}(k') \Rightarrow \emptyset \notin \text{dom}(y_j). \)

- \( m \notin [n, n'] \): function \text{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\_\text{live}(S', i, m) and \( \text{cap}(S', \gamma_1, i, n) \geq \text{cap}(y_1', r' \mapsto \pi, i) \) trivially hold from the latter fact and the assumption that \( \text{counts}_{\text{ok}}(\delta, n \mapsto y_1, i, S) \).

**Lemma 21 (Store Typing Preservation — Spawn)**

\[ \text{If } S \text{ is a well-typed store in respect to } R; M; \delta, n \mapsto y, \text{ par}(y_2) \lor y' = \emptyset \otimes (y \otimes y_2) \text{ holds, and } S' \text{ is derived from } S \text{ by changing the ownership of each region that is contained in } y_2; (S' = \text{transfer}(S, n, n', y_2)), \text{ then } S' \text{ is well-typed in respect to } R; M \text{ and the effect map } \delta, n \mapsto y' \text{, } \text{cap}(S', i, n) \geq \text{cap}(y_1', r' \mapsto \pi, i) \text{ trivially hold from the latter fact and the assumption that } \text{counts}_{\text{ok}}(\delta, n \mapsto y, i, S). \]

**Proof.** It suffices to prove that for all regions \( i \) of \( \delta, n \mapsto y', n' \mapsto y_1 \) store\_\text{ok}(\delta, n \mapsto y', n' \mapsto y_1, i, S') and mutex\_\text{ok}(\delta, n \mapsto y', n' \mapsto y_1, i, S') hold. The first obligation is immediate by lemma 20. The second obligation can be shown by performing a case analysis as follows:

- \( i \) does not belong in the domain of \( y_1 \): the lock capability of region \( i \) is unmodified for all threads, thus mutex\_\text{ok}(\delta, n \mapsto y', n' \mapsto y_1, i, S') is immediate by the assumption that mutex\_\text{ok}(\delta, n \mapsto y, S).

- \( i \) belongs in the domain of \( y_1 \) and is \text{impure}: the lock capability of region \( i \) is unmodified for all threads except for \( n' \) that receives a zero lock capability (by the \text{transfer} assumption), thus, mutex\_\text{ok}(\delta, n \mapsto y', n' \mapsto y_1, i, S') is immediate by the \text{transfer} assumption. The lock capability of \( i \) is unmodified for all threads except for \( n \) and \( n' \). In particular, the lock and region capability of \( i \) is \text{moved} from thread \( n \) to \( n' \) (by the \text{transfer} assumption) and \( i \) is no longer live nor locked for thread \( n \). The above facts imply that mutex\_\text{ok}(\delta, n \mapsto y', n' \mapsto y_1, i, S') holds.

**Lemma 22 (Preservation — Expressions)**

\[ \text{Let } e \text{ be a well-typed expression with } R; M; \emptyset \triangleright e : \tau \land (\delta(n); y'') \text{ and } R; M; \delta \triangleright S. \text{ If the operational semantics takes a step } S; e \longrightarrow S'; e', \text{ then there exist } R' \subseteq R \text{ and } M' \supseteq M, \text{ such that the resulting expression and the resulting store are well-typed with } R'; M'; \emptyset \triangleright e' : \tau \land (y''); R; M; \delta[n \mapsto y'] \triangleright S'. \]

24
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 immediately to stress that $u$ is a redex.


Case $T-RP$: The typing derivation of $T-RP$ gives us that $u$ is of the form $(e,v)[t]$. The operational rule that matches the shape of $u$ is $E-RP$. Thus, $u$ is of the form $(\lambda p. f)[t]$. By inversion of the latter derivation we obtain that $R;\ M; \emptyset; \emptyset \vdash f : \tau \land (\gamma;\gamma)\ where\ \gamma$ equals $\delta(n)$. The premise $R;\emptyset \vdash f$ of rule $T-RP$ implies that $t \in \mathcal{R}$. The application of lemma 6 to the latter facts, and $R;\emptyset \vdash f$ (premise of $\vdash R;\ M; \emptyset; \emptyset \vdash f[l/p] : \tau[l/p]\land(\gamma;\gamma)$, the well-formedness fact is immediate by the application of lemma 6 to the typing derivation of type assertion $\vdash S$. The application of lemma 6 to the typing derivation of type assertion $\vdash S$ gives us that $R;\ M; \emptyset; \emptyset \vdash f[l/p] : \tau[l/p]\land(\gamma;\gamma)$. 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 $\gamma$ 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 $R;\ M; \emptyset; \emptyset \vdash t : (\gamma;\gamma)$. Store typing: The operational rule $E-C$ matches the shape of $u$. Thus, we need to prove that $\vdash R;\ M; \emptyset; \emptyset; \gamma;\gamma,$ holds, where $S'$ equals $\text{updcap}(S,\eta,\delta,\gamma,\delta,\gamma)$. It suffices to show that $\text{store}_{ok}(\delta[n \mapsto \gamma],\gamma,\delta,\gamma)\ holds$. Consequently, for all regions $\gamma$ contained in $\delta[n \mapsto \gamma]$ we must show that:

- $\text{counts}_{ok}(\delta[n \mapsto \gamma],\delta,\gamma,\gamma,\delta,\gamma)$: the store typing assumption implies that $\text{counts}_{ok}(\delta,\delta,\gamma,\delta,\gamma)$ holds. Given that, $\delta[n \mapsto \gamma] = \delta'[m] = \gamma$, $\delta'[\pi],\gamma,\delta,\tau$ for some $\delta'[\pi],\gamma,\delta,\tau$ where $\gamma$ equals $\delta(n)$. The first obligation is immediate by $\text{counts}_{ok}(\delta,\delta),\delta,\gamma,\delta,\gamma)\ where\ \gamma$ equals $\delta(n)$. The premise $\vdash R;\emptyset \vdash f$ of rule $T-RP$ implies that $t \in \mathcal{R}$. The application of lemma 6 to the latter facts, and $\vdash R;\emptyset \vdash f$ (premise of $\vdash R;\ M; \emptyset; \emptyset \vdash f[l/p] : \tau[l/p]\land(\gamma;\gamma)$, the well-formedness fact is immediate by the application of lemma 6 to the typing derivation of type assertion $\vdash S$. The application of lemma 6 to the typing derivation of type assertion $\vdash S$ gives us that $\vdash R;\ M; \emptyset; \emptyset \vdash f[l/p] : \tau[l/p]\land(\gamma;\gamma)$. 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.
Expression typing: the store typing derivation of $S'$ implies that $i \notin R$. By inversion of the typing derivation of $u$ we have that $R, M; \emptyset; \rho; x : \text{rgn}(\rho) + e_2 : \tau(\gamma, \rho_{1\rightarrow} j; \gamma')$, such that $\rho \notin \text{dom}(\gamma')$. The application of lemma 15 to the typing derivation of $e_2$ and the fact that $i \notin R$ yields $R, i; M; \emptyset; \rho; x : \text{rgn}(\rho) + e_2 : \tau(\gamma, \rho_{1\rightarrow} j; \gamma')$. We then apply lemma 33 on the derivation of $e_2$ to obtain $R, i; M; \emptyset; x : \text{rgn}(i/\rho) + e_2[i/\rho] : \tau[i/\rho](\gamma[i/\rho], \rho_{1\rightarrow} j; \gamma'[i/\rho])$. By applying lemma 6 to the original typing derivation of $\text{newrgn}$ construct we have that the typing the context (including $\gamma$ and $\gamma'$) is not defined in terms of $\rho$ (i.e. $\rho$ is fresh). Further, the premise of $\text{newrgn}$ derivation suggests that $\tau$ is also independent of $\rho$ (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 $R, i; M; \emptyset; x : \text{rgn}(i) + e_2[i/\rho] : \tau(\gamma, \rho_{1\rightarrow} j; \gamma')$. By the application of lemma 6 to the fact that $e_2$ is well-typed, we have that $R, i; M; \emptyset; \gamma, \rho, \rho_{1\rightarrow} j; \gamma'$ is well-formed. By the definition of well-formedness, $R, r; M; \emptyset; \gamma, \rho, \rho_{1\rightarrow} j; \gamma'$ also holds. The definition of the typing rule $\text{T-R}$, the latter fact and the fact that $R, i; \emptyset ; i \vdash \tau$ implies that $\text{rgn}$ is well-typed (with type $\text{rgn}(i)$) in the context $R, i; M; \emptyset$ with effect $(\emptyset; \emptyset)$. By applying lemma 34 to the latter derivation and the fact that $R, i; M; \emptyset ; i \vdash \tau(\emptyset; \emptyset)$ we obtain $R, i; M; \emptyset ; i \vdash \tau(\gamma, \rho_{1\rightarrow} j; \gamma')$.

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 assertions that $R, M; \emptyset; \emptyset ; \gamma, \gamma'$ holds, where $\gamma$ and $\gamma'$ 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 $R, M; \emptyset; \emptyset, \emptyset \vdash \tau(\gamma, \gamma')$. By inversion of the typing derivation of $\text{deref}$ gives us $R, M; \emptyset; \gamma, \gamma'$, where $\gamma$ is equal to $\delta(n)$. By applying lemma 7 to the latter derivation and $R, M; \emptyset ; i \vdash \tau(\emptyset; \emptyset)$ we get $R, M; \emptyset ; i \vdash \tau(\gamma, \gamma')$. 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 $\gamma$ and $\gamma'$ 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 $R, M; \emptyset; \emptyset, \emptyset \vdash \tau(\gamma, \gamma')$. By the application of $\text{ref}$ gives us that $R; \emptyset \vdash \tau(\gamma, \gamma')$. By inversion of the typing derivation of $\text{ref}$ we have that $R; \emptyset \vdash \tau(\gamma, \gamma')$. By applying lemma 6 to the typing derivation of $\text{ref}$ we have that $R; \emptyset \vdash \tau(\gamma, \gamma')$. By inversion of the latter derivation $R; \emptyset \vdash \tau(\gamma, \gamma')$. By the application of lemma 7 to the latter fact and the fact that $R; \emptyset \vdash \tau(\gamma, \gamma')$ we get $R; \emptyset \vdash \tau(\gamma, \gamma')$.

Expression typing: the latter derivation is substituted for $\text{ref}$ with $\text{rgn}$ gives us that $R; \emptyset \vdash \tau(\gamma, \gamma')$. By inversion of the latter derivation $R; \emptyset \vdash \tau(\gamma, \gamma')$. By the applyiong of lemma 7 to the latter fact and the fact that $R; \emptyset \vdash \tau(\gamma, \gamma')$ we get $R; \emptyset \vdash \tau(\gamma, \gamma')$. By inversion of the store typing assumption we have that $\text{dom}(M) = \text{dom}(S)$ and $\forall \ell' \rightarrow (\ell, j) \in M; R; M; \emptyset ; \ell \vdash S(\ell') : \tau(\emptyset; \emptyset)$, such that $\forall \ell' \rightarrow (\ell, j) \in M; R; M; \emptyset ; \ell \vdash S(\ell') : \tau(\emptyset; \emptyset)$. We must show that both hold in the extended memory typing context and the new store. It suffices to show that the following holds:

- $\text{dom}(M, \ell \rightarrow (\tau, i)) = \text{dom}(S')$: The locations contained in store $S'$ are equal to the location contained in $S$ except for an additional location $\ell$. Thus, the latter fact and $M \vdash S$ imply that $M, \ell \rightarrow (\tau, i) \vdash S'$ holds.
- $\exists \ell' \rightarrow (\ell, j) \in M, \ell \rightarrow (\tau, i) ; M ; \emptyset ; \ell \vdash S' : \tau(\emptyset; \emptyset)$: immediate by $\forall \ell' \rightarrow (\ell, j) \in M, \ell \rightarrow (\tau, i) ; M ; \emptyset ; \ell \vdash S(\ell') : \tau(\emptyset; \emptyset)$ and $M, \ell \rightarrow (\tau, i) ; M ; \emptyset ; i \vdash \tau(\emptyset; \emptyset)$.

Case T-AP: We only need to consider the case where $\ell = \emptyset$ 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
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) \( R; r; \emptyset \vdash \gamma \land R; r; M; \Delta; \rho; \emptyset \vdash f : \tau \& (\gamma; \gamma) \land \Rightarrow R; r; M; \Delta; \emptyset \vdash f[I/\rho] : \tau[I/\rho] \& (\gamma; \gamma) \)

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

Case \( f \equiv \lambda e. e \) as \( \tau \): by inversion of the assumption typing derivation we have that \( R; r; M; \Delta; \rho; \emptyset \vdash \lambda e. e \) as \( \tau \), and by definition of the typing derivation of value \( e \) we have that \( \gamma \) is defined independently of \( \rho \) \( R; r; \emptyset \vdash \gamma \). Thus, \( R; r; M; \Delta; \emptyset \vdash (\lambda e. e \) as \( \tau[I/\rho] \) : \( \tau[I/\rho] \& (\gamma; \gamma) \) also holds.

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

Lemma 25 (Region substitution preserves \( \oplus \)) \( \exists \beta \) \( \exists \gamma_3 = \gamma_2 \& \gamma_1 \Rightarrow \exists \beta \gamma_3[I/\rho] = \gamma_2[I/\rho] \& \gamma_1[I/\rho] \)

Proof. If \( \gamma_1 \) is empty then rule ES-N implies that \( \gamma_1 \) equals \( \gamma_1 \). Thefore, \( \exists \beta \gamma_3[I/\rho] = \emptyset \& \gamma_1[I/\rho] \) holds. It can be trivially shown that if \( \exists \beta \gamma_3[I/\rho] = \gamma_1[I/\rho] \) then \( \gamma_1[I/\rho] \) holds. If \( \gamma_1 \) is not empty then rule ES-C applies. By inversion of this rule we have that the following hold:

- \( \gamma_3 = \gamma_3[I/\rho] = \gamma_3[I/\rho] \)
- \( \gamma_1[I/\rho] = \gamma_1[I/\rho] \)
- \( \gamma_2[I/\rho] = \gamma_2[I/\rho] \)
- \( \gamma_3[I/\rho] = \gamma_3[I/\rho] \)
- \( \gamma_1[I/\rho] = \gamma_1[I/\rho] \)
- \( \gamma_2[I/\rho] = \gamma_2[I/\rho] \)

By using rule ES-C we obtain that: \( \exists \beta \gamma_3[I/\rho] = \gamma_3[I/\rho] \& \gamma_2[I/\rho] \& \gamma_1[I/\rho] \)

Lemma 26 (Region substitution preserves \( \oplus \& \oplus \)) \( \exists \beta \gamma_3 = \gamma_2 \& \gamma_1 \Rightarrow \exists \beta \gamma_3[I/\rho] = \gamma_2[I/\rho] \& \gamma_1[I/\rho] \)

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

\[
\text{(a)} \exists \beta \gamma = \gamma_1 \& \gamma_2 \\
\text{(b)} \exists \beta \gamma_3 = \gamma_2 \& \gamma_2 \\
\text{(c)} \exists \beta \gamma_3 = \text{live}(\gamma_3) \\
\]

We can apply Lemma 25 to (a) to derive that \( \exists \beta \gamma[I/\rho] = \gamma_1[I/\rho] \& \gamma_2[I/\rho] \). We can apply the same reasoning to on (b) to derive \( \exists \beta \gamma_3[I/\rho] = \gamma_2[I/\rho] \& \gamma_2[I/\rho] \). We can trivially show that \( \exists \beta \gamma[I/\rho] = \gamma_1[I/\rho] \& \gamma_2[I/\rho] \) holds, by using (d) and by observing that the live parents substituted in \( \gamma[I/\rho] \) correspond to the live parents substituted in \( \gamma_1[I/\rho] \).

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

Proof. The assumption implies that \( r \) belongs in \( (R; r) \cup \Delta \). If \( r \) is then \( r[I/\rho] \) belongs in \( (R; r) \cup \Delta \). Otherwise, \( r \in (R; r) \cup \Delta \) trivially holds.
Lemma 31 (Variable Context Well-formedness Substitution) \( R; r; \Delta, \rho \vdash \gamma \Rightarrow R; r; \Delta \vdash \gamma[t/\rho] \)

Proof. We proceed by performing a case analysis on \( \gamma \):
- \( \emptyset : R; r; \Gamma \vdash \emptyset \) trivially holds.
- \( \langle \rangle : R; r; \Delta \vdash \langle \rangle \) trivially holds.
- \( \text{rgn}(r) : R; r; \Delta \vdash r[t/\rho] \) holds by lemma 27.
- \( \text{ref}(\tau', r) : R; r; \Delta \vdash \tau'[t/\rho] \) holds by the induction hypothesis.
- \( \forall \tau' : R; r; \Delta, \rho' \vdash \tau'[t/\rho] \) holds by the induction hypothesis.
- \( \tau' \vdash \tau' : R; r; \Delta \vdash \tau'[t/\rho] \) holds by the induction hypothesis. \( R; r; \Delta \vdash \gamma[t/\rho] \) holds by lemma 28. \( R; r; \Delta \vdash \gamma[t/\rho] \) holds by lemma 28. We have that \( \text{consistent}(\gamma_1; \gamma_2) \) and we must prove that \( \text{consistent}(\gamma_1[t/\rho]; \gamma_2[t/\rho]) \) holds. This is immediate by lemma 29.

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

Proof. By inversion of the first typing context and effect well-formedness assumption we have that
- \( R; t \vdash M \)
- \( R; r; \Delta, \rho \vdash \Gamma : R; r; \Delta \vdash \Gamma[t/\rho] \) immediate by lemma 31.
- \( R; r; \Delta, \rho \vdash \gamma_1 : R; r; \Delta \vdash \gamma_1[t/\rho] \) immediate by lemma 28.
- \( R; r; \Delta, \rho \vdash \gamma_2 : R; r; \Delta \vdash \gamma_2[t/\rho] \) immediate by lemma 28.
- \( \text{consistent}(\gamma_1; \gamma_2) \): the proof for \( \text{consistent}(\gamma_1[t/\rho]; \gamma_2[t/\rho]) \) is immediate by the application of lemma 29.
Lemma 33 (Region Substitution) \( R, r; M; \Delta, \rho; \Gamma \vdash e : \tau \land (\gamma_1; \gamma_2) \Rightarrow R, r; M; \Delta; \Gamma[t / \rho] \vdash e[t / \rho] : \tau[t / \rho] \land (\gamma_1[t / \rho]; \gamma_2[t / \rho]) \)

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 \( \vdash R, r; M; \Delta; \Gamma[t / \rho]; \gamma_1[t / \rho]; \gamma_2[t / \rho] \) 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 \( \vdash R, r; M; \Delta; \Gamma[t / \rho]; \gamma_1[t / \rho]; \gamma_2[t / \rho] \) holds. The proof for this case is completed by applying rule \( T-U \).

Case \( T-R \): by inversion of the derivation of \( e \)
- \( \vdash R, r; M; \Delta, \rho; \Gamma; \gamma_1; \gamma_2 \): the application of lemma 32 to the well-formedness premise implies that \( \vdash R, r; M; \Delta; \Gamma[t / \rho]; \gamma_1[t / \rho]; \gamma_2[t / \rho] \) holds.
- \( R; \Delta \vdash j : R, r; \Delta \vdash j[t / \rho] \) 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:
- \( \vdash R, r; M; \Delta, \rho; \Gamma; \gamma_1; \gamma_2 \): the application of lemma 32 to the well-formedness premise implies that \( \vdash R, r; M; \Delta; \Gamma[t / \rho]; \gamma_1[t / \rho]; \gamma_2[t / \rho] \) holds.
- \( (\ell \mapsto (\tau, j)) \in M \)

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:
- \( \vdash R, r; M; \Delta, \rho; \Gamma; \gamma_1; \gamma_2 \): the application of lemma 32 to the well-formedness premise implies that \( \vdash R, r; M; \Delta; \Gamma[t / \rho]; \gamma_1[t / \rho]; \gamma_2[t / \rho] \) holds.
- \( (x : \tau) \in \Gamma : (x : \tau[t / \rho]) \in \Gamma[t / \rho] \) 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:
- \( \vdash R, r; M; \Delta, \rho; \Gamma; \gamma; \gamma \): the application of lemma 32 to the well-formedness premise implies that \( \vdash R, r; M; \Delta; \Gamma[t / \rho]; \gamma_1[t / \rho]; \gamma_2[t / \rho] \) holds.
- \( R, \Delta, \rho \vdash \tau \): lemma 30 implies that \( \vdash R, r; \Delta \vdash \tau[t / \rho] \) holds.
- \( \tau \equiv \gamma_1 \xrightarrow{\gamma_2} \tau_2 \): the function type after substitution is \( \tau[t / \rho] \equiv \gamma_1[t / \rho] \xrightarrow{\gamma_2[t / \rho]} \tau_2[t / \rho] \).
- \( R, r; M; \Delta, \rho; \Gamma; x : \tau_1 \vdash e : \tau_2 \land (\gamma_1; \gamma_2); R, r; M; \Delta; \Gamma[t / \rho] \vdash e[t / \rho] : \tau_2[t / \rho] \land (\gamma_1[t / \rho]; \gamma_2[t / \rho]) \) 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:
- \( \vdash R, r; M; \Delta, \rho; \Gamma \vdash e_1 \equiv \gamma_1 \xrightarrow{\gamma_2} \gamma_3 \): By applying lemma 23 to the derivation of \( e_1 \) we obtain that \( \vdash R, r; \Delta, \rho \vdash \gamma_1 \xrightarrow{\gamma_2} \gamma_3 \).
- \( R, r; M; \Delta; \Gamma[t / \rho] \vdash e_1[t / \rho] : (\gamma_1 \xrightarrow{\gamma_2} \gamma_3[t / \rho]) \) holds by the induction hypothesis.
- \( \xi = \text{par}(\gamma_1) \Rightarrow \tau_2 = () : \xi[t / \rho] = \text{par}(\gamma_1)[t / \rho] \Rightarrow \tau_2[t / \rho] = () \) trivially holds.
- \( R, r; M; \Delta, \rho; \Gamma \vdash e_2 : \tau_1 \land (\gamma_3; \gamma_4); R, r; M; \Delta; \Gamma[t / \rho] \vdash e_2[t / \rho] : \tau_1[t / \rho] \land (\gamma_3[t / \rho]; \gamma_4[t / \rho]) \) holds by the induction hypothesis.
- \( \xi \vdash \gamma_2 \equiv \gamma_3 \land \gamma_4 \land \xi \vdash \gamma_3[t / \rho] \lor (\gamma_3[t / \rho] \land \gamma_4[t / \rho]) \) is immediate by the application of lemma 26.

Case \( T-CP \), \( T-RP \), \( T-NG \), \( T-NR \), \( T-D \), \( T-RE \), \( T-TE \), \( T-A \): We can perform similar reasoning to prove the remaining cases. The key point is to prove in the remaining cases that \( \text{live}(\gamma_4)[t / \rho] = \text{live}(\gamma_3[t / \rho]) \), where \( \gamma_4 \) is the effect of interest. The proof can be summarized as follows:
- \( \rho \) is a leaf element in \( \gamma_4 \); liveness for this regions is unaffected as its parents are unaffected.
- \( \rho \) is an intermediate node in \( \gamma_4 \); assuming that there exist an immediate and live descendant \( r' \), then its parent annotation is \( \rho \). Thus after substitution \( r' \) will still be live.
Lemma 34 (Variable Substitution) \( R; M; Δ; Γ, x : τ₁ + e : τ₂ & (γ₁; γ₂) \land R; M; 0; 0 + v : τ₁ & (0; 0) \Rightarrow R; M; Δ; Γ + e[v/x] : τ₂ & (γ₁; γ₂) \)

Proof. Straightforward induction on the expression typing derivation.

Lemma 35 (Value Strengthening) \( R; M; Δ; Γ + v : τ & (γ₁; γ₂) \Rightarrow R; M; 0; 0 + v : τ & (0; 0) \)

Proof. By case analysis on the value typing derivations. The proof is trivial as it suffices to prove well-formedness effect strengthening, which is immediate from the definition of well-formedness: \( ⊢ R; M; Δ; Γ; γ₁; γ₂ \) can be immediately be strengthened to \( ⊢ R; M; Δ; Γ; 0; 0 \). We can then combine the strengthened well-formedness derivation with the remaining premises of each value derivation, which remain intact, to obtain \( R; M; 0; 0 + v : τ & (0; 0) \).

Lemma 36 (Context Weakening) \( R; M; Δ; Γ + e : τ₁ & (γ₁; γ₂) \land R; M; Δ; Γ; γ; γ'' \land seq + γ'' = γ₂ \land (γ \land γ') \Rightarrow R; M; Δ; Γ + e : τ₁ & (γ₁; γ'') \)

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 \( e₁ \) and \( e₂ \) are well-typed in the typing context \( R; M; Δ; Γ \) with effects \( (γ₁; γ₂) \) respectively. By applying lemma 6 to the typing derivation of \( e₁ \) we have that consistent\( (γ₁; γ₂) \) holds. By applying lemma 40 to the latter fact and the assumption that \( seq + y'' = γ₂ \land (γ \land γ') \) we obtain that \( seq + γ₁ = γ₁ \land (γ \land γ') \) for some \( γ \). By applying lemma 6 to the typing of \( e₂ \) we obtain that consistent\( (γ₁; γ₂) \). By applying lemma 40 to the latter facts we have that \( seq + γ₁ = γ₂ \land (γ \land γ') \) and \( seq + γ'' = γ₂ \land (γ \land γ') \) for some \( γ \). The above facts (judgements) and the well-formedness assumption tell us that \( ⊢ R; M; Δ; Γ; γ; γ₁, γ₂ \) and \( ⊢ R; M; Δ; Γ; γ; γ₂, γ₁ \). We can apply the induction hypothesis to derive that \( e₁ \) and \( e₂ \) are well-typed in the same typing context with effects \( (γ₁; γ₂) \) and \( (γ₂; γ₁) \) respectively. The next step is to apply 37 to the fact obtained that \( seq + γ'' = γ₂ \land (γ \land γ') \) and the application premise \( γ \land γ₁ \) to derive \( γ \land γ'' = γ₂ \land (γ \land γ') \). Hence, premises which are related to \( dom, accessible, \) or \( [k] \) \( (k) \) that hold for \( γ₁ \) and \( γ₂ \) also hold for \( γ'' \) and \( γ \).

Lemma 37 (Context Weakening) \( γ \land γ₁ \)

Proof. The assumptions yield the following facts:

\(~(γ\landγ₁)\)
\(seq + γ = γ₁ \land γ₂\)
\(ε \land γ₁ = γ₂ \land γ₂\)
\(ε \land γ₂ = γ₁ \land γ₂\)
\(ε \land γ₃ = \text{live}(γ₃)\)
\(abs\_par(γ; γ₁) \subseteq \text{dom}(γ')\)

By applying lemma 38 to (a) and (e) we obtain that \( ∃γ''₈. ε \land γ = γ₁ \land γ₂ \land ∀r'' \pi \in γ''₈. ∃r'' \pi' \in γ₁, r'' \pi'' \in γ₂. rg(γ) = rg(π₁) + rg(π₂) \land lk(π₁) = lk(κ₁) + lk(κ₂). Facts (f) and (b) as well as the third assumption imply that \( ε \land γ₂ = γ₁ \land γ₂ \), where \( γ''₈ \) is a subset of \( γ₂ \) such that \( \text{dom}(γ''₈) = \text{dom}(γ₂) \). By applying lemma 38 to the latter derivation and (b) we obtain that \( ∃γ''₈. ε \land γ₂ = γ₁ \land γ₂ \land ∀r'' \pi \in γ''₈. ∃r'' \pi' \in γ₁, r'' \pi'' \in γ₂. rg(γ) = rg(κ₁) + rg(κ₂) \land lk(π₁) = lk(κ₁) + lk(κ₂). Given the above facts we can deduce that \( γ'' = γ''₈ \). To complete the proof we need to show that \( \text{abs\_par}(γ; γ₁) \subseteq \text{dom}(γ') \). This is immediately by facts (d), (g), (b) and (c): \( \text{abs\_par}(γ; γ₁) = \text{abs\_par}(γ₁, γ₁) \subseteq \text{dom}(γ₂) \subseteq \text{dom}(γ') \) (by simple observation of the definition of \( \text{abs\_par} \)).

Lemma 38 (Context Weakening) \( γ \land γ₂ \)

Proof. By induction on the structure of \( γ₂ \):

Case \( γ₂ = 0 \): The conclusion trivially holds for \( γ'' = γ \).

Case \( γ₂ = γ₂', r'' \pi'' \): The assumptions yield the following facts:
By applying the induction hypothesis on (k) and (l) we have that $3y, e + y' = γ_1 + γ_2$. By the definition of rule CS, and facts (i) and (j) it is trivial to show that there exists a $κ_k$ such that $η + κ_k = κ + κ_k$ and $rg(κ_k) = rg(κ(k)) + rg(κ(η))$. We can construct a fresh $κ$ such that $seq + κ = n_1, n_2 + n_1, n_2$ by using facts from predicate consistent. The above facts and rule ES-C imply that $seq + γ', r + ζ = γ_2, r + ζ \subseteq \gamma_1, r + ζ \pi$ holds.

Case $γ_2 = γ_2, r + ζ \pi$: Similar to the previous case.

Lemma 39 (opl Implication) $seq + γ = γ_1 + γ_2 \land \text{consistent}(γ_1, γ_2) \Rightarrow 3y'. seq + y' = γ_2 + γ_2$

Proof. By induction on the structure of $γ_2$:

Case $γ_2 = Θ$: Then the conclusion trivially holds for $γ = γ_2$.

Case $γ_2 = γ_2, r^{n_1 \to π}π'$: The second assumption yields that there exists a $γ'$ such that $γ_2 = γ_1, r^{n_2 \to π}π'$. Consequently, the first assumption becomes (rule ES-S): $seq + γ_2, r^{n_2 \to π}π = γ_1, r^{n_1 \to π}π + γ_2, r^{n_2 \to π}π$ for some $γ_1$ and $γ_2$. By inversion of the latter derivation we obtain that $seq + γ = γ_1 + γ_2$ holds. The second assumption and the above facts imply that consistent($γ_1, γ_2$) also holds. By induction hypothesis we have that there exists a $γ''$ such that $seq + γ = γ_2 + γ_2$. We can construct a fresh $κ$ such that $seq + κ = n_1, n_2 + n_1, n_2$ by using facts from predicate consistent. The above facts and rule ES-C imply that $seq + γ', r + ζ = γ_2, r + ζ \subseteq \gamma_1, r + ζ \pi$ holds.

Case $γ_2 = γ_2, r^{n_1 \to π}π'$: Similar to the previous case.

Lemma 40 (opl/opl Implication) $seq + γ' = γ_2 + (γ \odot γ_1) \land \text{consistent}(γ_1, γ_2') \Rightarrow 3y''. seq + y'' = γ' + (γ \odot γ_1) \land seq + γ = γ_2 + (γ'' \odot γ_2')$

Proof. By inversion of the first assumption we obtain: $seq + γ = γ_1 + γ_2$ and $seq + γ_2 = γ_2 + γ_2$, for some $γ_2$. $γ' = \text{live}(γ_2)$, and $abs pari(γ, γ_1) \subseteq \text{dom}(γ')$. By the application of lemma 39 to $seq + γ = γ_1 + γ_2$, we have that $seq + γ_2 = γ_2 + γ_2$ holds for some $γ_2$.

To prove the left of the conjuction of the conclusion it suffices to show that $abs pari(γ, γ_1) \subseteq \text{dom}(\text{live}(γ_2))$. This can be shown by proving that $\text{dom}(\text{live}(γ_2)) \subseteq \text{dom}(\text{live}(γ_2))$. This is immediate by the second assumption, which implies that the domain of $γ_2$ is a subset of the domain of $γ_1$, which contains positive region counts (γ_1 is live by assumption consistent). Thus, $γ_2$ contains at least as many live regions as $γ_2$. To prove the right of the conjuction it suffices to show that $abs pari(γ_2, γ'_2) \subseteq \text{dom}(γ''')$. As mentioned earlier, $γ'_2$ is a subset of $γ_1$ so it contains at most as many abstracted parents as $γ_1$. Therefore, $abs pari(γ_2, γ'_2) \subseteq abs pari(γ, γ_2)$. Thus, $abs pari(γ_2, γ'_2) \subseteq \text{dom}(γ''')$ holds.

Lemma 41 (Progress — Program) Let $S; T$ be a closed well-typed configuration with $R; M; δ ⊢ C S; T$. 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 $n : ❨(l)⟧$: Then proof is immediate by rule E-T.

Case $n : E[(lx, e x as τ) par(n)]$: it suffices to prove that there exists an $S'$, such that $S'$ is equal to transfer($S, n, n', γ_1$). 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 $γ_1 = γ_2 + (γ \odot γ_2)$ (obtained by performing inversion on the configuration typing derivation, then a subsequent inversion on the thread 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 $R; M; δ ; 0 + e : τ & δ(0); 0$. If $e$ is a value then lemma 46 tells us that $e$ is (). We already have that $T = T_i, n : e$ for some $T_i$. 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 $3e_1, E[e_1] = e$ and redex(e_1). Thus, $R; M; δ ; 0 + E[e_1] : (γ; δ(0); 0)$ is also well-typed. The application of lemma 44 to redex(e_1), the typing derivation of $e_1$ and the store typing assumption ($M; R; δ ⊢ w S$) yields $3S_1, e_2, S_1, e_1 : S_1 → S_2; e_2$ or block($S, n, e_1, δ$) or $3S, γ_1, e_2, τ, e_1 = (lx, e x as τ) par(n)$. 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 have 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 $i$ to binary tuples of natural numbers.

- $\text{sum}(\gamma) = \mathbb{z}$ if ($\forall i \in \text{dom}(\gamma).z(i) = \text{cap}(\gamma, i)$) and ($\forall i \notin \text{dom}(\gamma).z(i) = 0, 0$)
- $\mathbb{z} \leq \mathbb{z}'$ if ($\forall i \in \text{dom}(\mathbb{z}) \cup \text{dom}(\mathbb{z}').z(i) \leq z'(i)$)
- $\text{cnt}(S, n) = \mathbb{z}$ if ($\forall i.\text{thread}\_\text{live}(S, i, n) \Rightarrow z(i) = \text{cap}(S, i, n)$) and ($\forall i.\neg \text{thread}\_\text{live}(S, i, n) \Rightarrow z(i) = 0, 0$)
- $\text{owns}(S, \gamma, n) = \forall i \in \text{dom}(\gamma), (z : (\theta, H, S') \in \text{bflattened}(S)) \land n \in \text{dom}(\theta)$

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

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

Case $\gamma_1 = \gamma', \mathbb{e}^n \rightarrow (\pi \rightarrow \pi)$: Similar proof to the previous case.

Lemma 43 (Progress — Spawn) $M; R; \delta, n \mapsto \gamma + S \land \xi + \gamma_3 = \gamma_2 \oplus (\gamma \oplus \gamma_1) \land n'$ 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 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 pure\_once($\gamma_1$) hold. The proof is completed by applying lemma 42 to $\text{owns}(S, n, \gamma_1)$, pure\_once($\gamma_1$) and $\text{sum}(\gamma_1) \leq \text{cnt}(S, n)$.

Lemma 44 (Progress — Expressions) $\text{redex}(e)\rightarrow R; M; \delta; \emptyset \mapsto e : \tau & (\delta(n); \gamma') \land R; M; \delta + S \Rightarrow \text{block}(S, n, e', \delta) & (S', e'; e \rightarrow S, S'; e') \lor (\exists e, e. e' \equiv (a, e, e_1 \rightarrow_\pi \forall x.e'))$

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 redex($e$).

Case T-AP, T-RP: The proof is 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 $v_1 \equiv \text{gn}_n$. It suffices to show that the premise of E-NRG, namely $(S_1, j) = \text{newgn}(S, n, m)$ is satisfied. This can be shown by applying lemma 46 to store typing derivation $M; R; \delta \mapsto S$, which is obtained by the assumption of the progress theorem, and the fact that $\delta = \delta, n \mapsto \gamma', m \rightarrow \pi$, which is obtained by the premise of rule T-NRG that requires that $m$ belongs in the domain of $\gamma$.

Case T-NR: The application of lemma 46 to the typing derivation of $v_2$, which is obtained by inverting T-NR, yields that $v_2 \equiv \text{gn}_n$. 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 $(S_1, \ell) = \text{alloc}(j, S, v_1)$. By inversion of T-NR we have that $R; M; \emptyset; \emptyset \mapsto \ell : \tau \& (\gamma; \gamma)$, and $j \in \text{dom}(\gamma)$. By applying lemma 45 on the typing derivation of $v_1$ we obtain $R; M; \emptyset; \emptyset \mapsto \tau : \tau \& (\emptyset; \emptyset)$. By applying lemma 47 to the latter derivation as well as $j \in \text{dom}(\gamma)$ and the store typing $R; R; \delta, \delta, j \mapsto \ell \mapsto \gamma \mapsto S$ we have that $(\ell, S, j) = \text{alloc}(j, S, v_1)$.

Case T-D: The application of lemma 46 to the typing derivation of $v$, which is obtained by inverting T-D, yields $v \equiv \text{loc}_\gamma$. Further, the typing of $v$ is accessible, Therefore, rule E-D can be applied to perform a single step.
Lemma 45 (Expression — Redex) \( R; M; \Delta; \Gamma \vdash e : \tau_1 \& (\gamma; \gamma_2) \& \epsilon \neq \epsilon_1 \Rightarrow \exists \epsilon', E.E[\epsilon] \equiv e \land \text{redex}(e) \)

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 \( E \equiv \emptyset \) and \( L \equiv \bot \).
- \( T-NR \): By observing the shape of the expression of \( T-NR \) typing derivation, \( e \equiv \text{new} e_1 \). Then \( e_1 \) and \( e_2 \) are both values then the proof is immediate \( (E \equiv \emptyset \land u \equiv \text{new} e_1) \). Otherwise, if \( e_1 \) is not a value the application of the induction hypothesis on the typing derivation of \( e_1 \) (obtained from \( T-NR \) inversion) yields that \( \exists E[u].E[u] \equiv e_1 \land u \equiv u_2 \). Consequently, \( \exists E.\text{new} E[u] \) at \( e_2 \equiv e \land u \equiv u_2 \) or equivalently \( \exists E.\text{new} E \) at \( e_2 \). The last case is that \( e_1 \) is a value and \( e_2 \) is not. By applying similar reasoning we can prove that \( \exists E.\text{new} e_1 \) at \( E[u] \equiv e \land u \equiv u_2 \).
- \( T-AP, T-RP, T-NG, T-CP, T-D, T-A \): We can perform similar reasoning to prove the remaining cases.

Lemma 46 (Canonical Forms) \( R; M; \Delta; \Gamma \vdash v : \tau \& (\gamma; \gamma_2) \Rightarrow \)

\[\begin{align*}
\tau \equiv (\gamma) & \Rightarrow \nu \equiv (\gamma) \land \\
\tau \equiv \text{rgn}(i) & \Rightarrow (v \equiv \text{rgn}, \land i \in R) \land \\
\tau \equiv \text{ref}(r,i) & \Rightarrow (v \equiv \text{loc}_{\ell} \land \ell \equiv (r,i)) \land \\
\tau \equiv b & \Rightarrow v \equiv \nu \land \\
\tau \equiv \tau_1 \Rightarrow \tau_2 & \Rightarrow v \equiv \lambda x.e \text{ as } \tau_1 \Rightarrow \tau_2 \land \\
\tau \equiv \forall \rho, \tau & \Rightarrow v \equiv \forall \rho, f \land \\
\tau \equiv \nu & \Rightarrow v \equiv \nu \land \\
\end{align*}\]

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

Lemma 47 (Progress — Add Location) \( M; R; \delta, k \mapsto \gamma, f \mapsto \pi \land \nu, S \land R; M; \emptyset; \emptyset \vdash v : \tau \& (\emptyset; \emptyset) \land \Rightarrow \exists S_1, \ell.(S_1, \ell) = \text{alloc}(j, S, v) \land \ell \not\in \text{dom}(M) \)

Proof. To prove that \( \text{alloc} \) is defined we only need to show that region \( j \) is live. This is immediate by inversion of the first assumption.

Lemma 48 (Progress — Add Region) \( M; R; \delta, k \mapsto \gamma, m' \mapsto \pi \land \nu, S \land R \Rightarrow \exists S_1, j(j, S_1) = \text{newrgn}(S, i, m) \land j \notin R \)

Proof. To prove that \( \text{newrgn} \) is defined we only need to show that region \( j \) is live. This is immediate by inversion of the first assumption.

Lemma 49 (Progress — Lookup Value) \( M, \ell \mapsto (\tau, j); R; \delta, n \mapsto \gamma \land \nu, S \land j \in \text{accessible}(\gamma) \Rightarrow \exists v. v = \text{lookup}(S, \ell, n) \)

Proof. To prove that \( \text{newrgn} \) is defined we need to show that region \( j \) is live, where \( j \) is the region that contains \( f \) in its heap, \( \text{canlk}(S, j, n) \) only holds for thread \( n \), and \( \ell \) and \( j \) 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 \( \text{counts-ok} \) of the store typing assumption, whereas the last obligation is immediate by the store typing premise that all locations in \( M, \ell \mapsto (\tau, j) \) exist in \( S \). The last obligation is immediate by the store typing premise \( \text{mutex-ok} \) and the assumption that \( j \) is accessible in the effect of thread \( n \).