Race-free and Memory-safe Multithreading: Design and Implementation in Cyclone

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

We present the design of a formal low-level multi-threaded language with advanced region-based memory management and synchronization primitives, where well-typed programs are memory safe and race free. In our language, regions and locks are combined in a single hierarchy and are subject to uniform ownership constraints imposed by a hierarchical structure: deallocating a region causes its sub-regions to be deallocated. Similarly, when a region is protected, then its sub-regions are also protected. We discuss aspects of the integration and implementation of the formal language to Cyclone and evaluate the performance of code produced by the modified Cyclone compiler against highly optimized C programs using atomic operations, pthreads, and OpenMP. Although our implementation is still in a preliminary stage, our results show that the performance overhead for guaranteed race freedom and memory safety is acceptable.

1. Introduction

With the emergence of commodity multicore architectures, exploiting the performance benefits of multi-threaded execution has become increasingly important to the extent that doing so is arguably a necessity these days. Programming languages that retain the transparency and control of memory such as C, seem best-suited to exploit the performance benefits of multicore machines, except for the fact that programs written in them often compromise memory safety by allowing invalid memory accesses, buffer overruns, space leaks, etc. and become susceptible to data races by careless uses of locks. Thus, a challenge for programming language research is to design and implement multithreaded low-level languages providing static guarantees for memory safety and freedom from data races and at the same time allow for a relatively smooth conversion of legacy C code to its multi-threaded counterpart.

Towards this challenge, we present the design of a formal low-level concurrent language that employs advanced region-based management and hierarchical lock-based synchronization primitives. Similar to other approaches, our memory regions are organized in a hierarchical manner where each region is physically allocated within a single parent region and may contain multiple children regions. Our language allows deallocation of complete subtrees in the presence of region sharing between threads and deallocation is allowed to occur at any program point. Each region is associated with an implicit lock. Thus, locks also follow the hierarchical structure of regions and in this setting each region is protected by its own lock as well as the locks of all its ancestors. As opposed to the majority of type systems and analyses that guarantee race freedom for lexically-scoped locking constructs [8, 12, 20], our language employs non-lexically scoped locking primitives, which are more suitable for languages at the C level of abstraction. Furthermore, the formal language allows regions and locks to be safely aliased and escape the lexical scope when passed to a new thread. These features are invaluable for expressing numerous idioms of multi-threaded programming such as *sharing*, *region ownership* or *lock ownership transfers*, and *region migration*.

Our formal language is not just a paper design with some nice properties. We have integrated our language in Cyclone [21], a strongly-typed dialect of C which preserves explicit control and representation of memory without sacrificing memory soundness. We have opted for Cyclone both because it has a publicly available implementation but also because it is more than a safe variant of C. Cyclone offers modern programming language features such as first-class polymorphism, exceptions, tuples, namespaces, (extensible) algebraic data types, and region-based memory management. We will discuss how these features interact with our language and the additions that were needed to Cyclone's implementation.

The contributions of this paper are as follows:

- We improve on our earlier work [18] by providing an operational semantics for our language that not only provides explicit guarantees for race freedom and memory safety, but also explicit guarantees as to when a subtree is deallocated, thereby avoiding temporary memory leaks.
- We discuss the integration of our formal language to Cyclone. The resulting language is a concurrent language at the C level of abstraction that enjoys the benefits of the formal system: it offers memory safety and race freedom guarantees and allows regions to be deallocated in bulk and also be locked atomically.
- We discuss implementation issues related to analysis, code generation and additions to the run-time system that were required in order to make the integration possible. the complex protocols used by Cyclone programs that wish to gain explicit control over region lifetimes.

The next section starts by reviewing the Cyclone language. We present our language and its operational semantics in Section 3. We describe the interaction of our language with Cyclone in Section 4, followed by a presentation of implementation (Section 5) and performance (Section 6) aspects of its integration. The paper ends by short sections presenting future improvements to our implementation (Section 7), discussing related work (Section 8), and with some concluding remarks.

2. Cyclone: A memory-safe dialect of C

In this section, we provide a high-level overview of memory management in Cyclone. In particular, we discuss how traditional regions are used in Cyclone and identify some shortcomes that are alleviated in our language. Additionally, we clarify through several examples that Cyclone's memory safety guarantees only hold

for sequential programs. Cyclone has no race freedom guarantees, which are important at the language-level and the compiler-level.

2.1 Memory management in Cyclone

Cyclone employs a uniform treatment of different memory segments such as the main heap, the stack and individual regions. More specifically, memory segments are mapped into *logical memory partitions*. Each data object is allocated at a single memory segment, but references to objects may refer to multiple segments. Hereon, we overload the term "region" to mean a type-level logical memory partition, a run-time entity that enables fast allocation and bulk deallocation of objects, or a memory segment such as the heap and the stack.

For instance, a stack frame is treated as a region holding the values of variables declared in a lexical block. As another example, the main heap is an immortal region that contains all global variables. The type system of Cyclone tracks the set of *live* regions at each program point and verifies that the regions associated with each accessed object are indeed a subset of the live regions.

```
{ region < `r > h ;          // live regions:   {`r}
  int *z = rnew(h) 10;       //                 {`r}
      ...                    //                 {`r}
}                            //                 { }
```

The above example illustrates how a scoped region can be created and used: the first statement allocates a fresh memory segment, and associates this segment with a fresh type-level region (i.e., `r). Following Cyclone's terminology, we use a leading backquote for type-level names, e.g., `r. (We will often use the same name without the backquote for the corresponding region handle, which is here explicitly named *h*.) The comments on the right-hand side of the example's code show the live region set (i.e., the *effect*) at each program point.

The new region can be accessed via its *region handle* (*h*), which is given the *singleton type* `region_t < `r >`. The second statement uses *h* to allocate memory for a single integer and initializes it to the value 10. At the type level, the type of the fresh reference is annotated with region `r (i.e., `int * `r`). The type system ensures that the reference can only be accessed when `r is in the current effect.

The uniform treatment of memory allows for polymorphism over different kinds of memory segments.

```
void swap (int * `r1 x , int * `r2 y );
```

For instance, the above line of code defines a function that swaps the contents of the variables *x* and *y* located at regions $`r_1$ and $`r_2$ respectively. Both $`r_1$ and $`r_2$ are polymorphic and can be instantiated with *any* region. The following line of code invokes swap by explicitly instantiating both $`r_1$ and $`r_2$ to `r.

```
{ region < `r > h ;
  int *z = rnew(h) 10;
  int *y = rnew(h) 15;
      swap(z, y);              // effect of swap   {`r, `r}
}
```

As shown in the above example, type-level regions can be freely aliased in a effect (e.g., {`r, `r}). The downside of this approach is that scoped regions can only be deallocated implicitly by the run-time system when a region's scope ends.

To overcome this restriction, the authors of Cyclone have extended the region system with three powerful features, namely *tracked types*, the notion of *borrowing* tracked types and *existential types*. Tracked types, which are closely related to linear types, disallow aliasing of *tracked* references. Borrowing can be used to convert a tracked reference to an aliasable reference within a particular scope. The aliasable reference is accessible within the scope, whereas the tracked reference becomes inaccessible for the duration of the scope. Finally, existential types serve as the means for overcoming lexically scoped region names, by permitting the on-demand concealment and disclosure of region names. Cyclone allows access and deallocation of non-lexically scoped (i.e., *dynamically* scoped) regions as follows:

- a request is made to the run-time system to allocate a fresh dynamic region.
- the run-time returns an existential package containing some region name `r and a *key* (i.e., a tracked reference) to the handle of the fresh region. The handle is also annotated with `r.
- the existential value is unpacked and `r is brought into scope as well as the key.
- the program can immediately deallocate the new region by deallocating the key.
- or it may temporarily yield access to the key by allowing it to be *borrowed* within a scope. During that scope `r is added to the effect and the region handle referred by the key is usable.

The following example illustrates a similar scenario:

```
void access_and_deallocate (NewDynRgn pr ) {
  let NewDynRgn{< `r > key} = pr;     // open existential
  {region h = open(key);              // borrow key for this scope
   let x = rnew(h) 5;
   ...
  }
  free_ukey(key);                     // deallocate region
}
```

It should be noted that a dynamic region cannot be deallocated when its key has been *borrowed*. Additionally, Cyclone allows tracked references to leak and thus allows dynamic regions to leak as well. To tackle this issue, an intra-procedural analysis can be used to report tracked reference leaks. In practice, this analysis is impractical as it produces a large number of false positives [27]. For instance, when a function call takes place between the allocation and deallocation point of a tracked reference, the analysis must report that the tracked reference may leak as an uncaught exception may be thrown during the call. For a detailed discussion about Cyclone we refer the reader to [27].

As it will be discussed in latter sections, our work disallows memory leaks in the presence of a complex shared memory management scheme with bulk region deallocation, allows region deallocation at *any program point* and simplifies the process of creating, using and deallocating explicitly freeable regions.

2.2 Concurrency in Cyclone

Cyclone does not have language support for concurrency. Instead, it provides an interface to the pthreads library, which allows programmers to spawn new threads and use numerous synchronization primitives to control the interaction between threads. The interface to the pthreads library ensures that the run-time data structures are correctly initialized before a new thread runs.

To preserve memory safety (e.g., absence of dangling pointers), Cyclone requires that all memory regions passed to a new thread must live at least as long as the immortal (main) heap. This implies that threads can interact with other threads via dynamically allocated references that reside in the heap or global variables. This restriction diminishes the explicit memory management benefits of Cyclone (aliasable heap references can only be garbage collected).

The following definition has been extracted from Cyclone's interface to pthreads library:

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

The most interesting part of the above definition is the part `regions('b)` ≤ `'H`, which says that all region names occurring in the type that will instantiate the type variable `'b` must be live at least as long as the the immortal heap (`'H`). Tracked pointers cannot be passed to threads.

The memory safety guarantees of Cyclone can be compromised in the presence of multi-threading. Here, we only mention a few cases which can violate memory safety. Firstly, the data flow analysis performed for identifying where dynamic checks (e.g., null pointer and array bounds checks) should be inserted is unsound in a concurrent setting. Consider the following code fragment:

```
void foo (int * 'r * 'r x) {
    if (x != NULL && *x != NULL) **x = 20;
}
```

Assuming that x is a shared *possibly null* reference, then the analysis will deduce that $**x$ can be accessed within the conditional statement as x and $*x$ are definitely *not null*. This property does not hold for concurrent programs that share x, but do not synchronize accesses to it.

Secondly, some features of Cyclone such as pattern matching, accesses to wide references (i.e., *fat pointers*) and *swap* operations between tracked references must be performed *atomically*. The lack of atomicity in swap operations and wide references can trivially compromise memory safety and cause cause dangling pointer dereferences and double "free" operations. As mentioned earlier, the run-time system does handle the initialization of new threads, but the region allocation subsystem is not reentrant.

Last but not least, Cyclone's type system does not guard against data races. The absence of data races gives additional guarantees to the programmer and allows a thread-aware compiler to perform certain kinds of optimizations that should only be applied to sequential programs. As it will be shown in the sections that follow, we have solved some of the issues stated above by implementing an adjusted version of our type system and operational semantics in Cyclone. We have also re-engineered the run-time system so that it is mostly non-blocking and thread-safe. The next section introduces our type system and operational semantics.

3. Formal language

Earlier work on the hierarchical region type system provides an operational semantics that satisfies our design goals, but admits *temporary leaks* of region and locks [18]. The memory leaks are temporary as the type system enforces that all regions and locks will eventually be released and can be avoided, by disallowing the release of *aliased* regions. However, this restriction would impede the expressiveness and the benefits of our language, as region aliasing is the rule not the exception.

We improve on earlier work, by providing semantics for the same type system that disallows leaks and proving it safe. The key idea is the introduction of dynamic effects and the preservation of an exact correspondence between dynamic and static effects.

3.1 Language description

The language syntax is illustrated in Figure 1.[1] The core expressions comprise of variables (x), constants (c), functions, and function application. Function application terms are annotated with a *calling mode* (ξ). The calling mode specifies whether a function application should be executed sequentially (seq) or in parallel (par).

Monomorphic functions ($\lambda x.\, e$) must be annotated with their type (τ). Our language is region-polymorphic and thus includes polymorphic functions ($\Lambda\rho.\, f$) and region application ($e[\rho]$).

The construct $\texttt{newrgn}^r\ \rho, x\ \texttt{at}\ e_1\ \texttt{in}\ e_2$ allocates a fresh region ρ at the region indicated by handle e_1, and binds x to the *handle* of ρ. Both ρ and x are lexically bound to the scope of e_2. The new region must be explicitly released within e_2. The region allocation construct is annotated with the parent region name r, which is only required for the type safety proof.

The constructs for manipulating references are standard. A newly allocated memory cell is returned by $\texttt{new}\ e_1\ \texttt{at}\ e_2$, where e_1 is an initializer expression for the new cell and handle e_2 indicates the region in which the new cell will be allocated. Standard assignment and dereference operators complete the picture. A region can be released 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 accessed directly with no additional overhead. This is our equivalent of a thread-local region.

By using the construct $\texttt{cap}^r_\eta\ e$, a thread can *increment* or *decrement* the capability counts of some region r whose handle is specified in e. The cap construct annotation (r) is only required for the type safety proof. The capability operator η can be, e.g., rg+ (meaning that the region count is to be incremented) or lk− (meaning that the lock count is to be decremented). 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, but it may still be *protected* by a locked ancestor region. 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 protected.

A capability of the form (n_1, n_2) is called a *pure* capability, whereas a capability of the form $(\overline{n_1}, \overline{n_2})$ is called an *impure* capability. In both cases, it is implied that the current thread can decrement the region count n_1 times and the lock count n_2 times. Impure ca-

Function	f	::=	$\lambda x.\, e\ \texttt{as}\ \tau \xrightarrow{\gamma\to\gamma} \tau \mid \Lambda\rho.\, f$
Expression	e	::=	$x \mid \texttt{c} \mid f \mid (e\ e)^{\xi} \mid e[r] \mid \texttt{new}\ e\ \texttt{at}\ e$
		|	$e := e \mid \texttt{deref}\ e \mid \texttt{newrgn}^r\ \rho, x\ \texttt{at}\ e\ \texttt{in}\ e$
		|	$\texttt{cap}^r_\eta\ e \mid \texttt{rgn}_\iota \mid \texttt{loc}_l \mid \texttt{pop}_\gamma\ e$
Type	τ	::=	$b \mid \langle\rangle \mid \tau \xrightarrow{\gamma\to\gamma} \tau \mid \forall\rho.\, \tau \mid \texttt{ref}(\tau, r) \mid \texttt{rgn}(r)$
Effect	γ	::=	$\emptyset \mid \gamma, r^{\kappa} \triangleright \pi$
Capability kind	ψ	::=	rg | lk
Capability op	η	::=	$\psi+ \mid \psi-$
Region	r	::=	$\rho \mid \iota \mid \iota@n$
Capability	κ	::=	$n, n \mid \overline{n, n}$
Region parent	π	::=	$r \mid \bot$
Calling mode	ξ	::=	seq | par

Figure 1. Syntax.

[1] The constructs $\texttt{rgn}_\iota$, $\texttt{loc}_l$ and $\texttt{pop}_\gamma\ e$ are not considered as a part of the language. They are only introduced during program evaluation. We defer the discussion about them until section 3.2.

Stack	σ	::=	$\emptyset \mid \sigma;\gamma$
Hierarchy	δ	::=	$\emptyset \mid \delta, n \mapsto \sigma$
Contents	H	::=	$\emptyset \mid H, \ell \mapsto v$
Region list	S	::=	$\emptyset \mid S, \iota \mapsto H$
Threads	T	::=	$\emptyset \mid T, n : e$
Configuration	C	::=	$\delta; S; T$

$$E ::= \square \mid (E\ e)^{\xi} \mid (v\ E)^{\xi} \mid E[r] \mid \texttt{newrgn}^{r}\,\rho, x\ \texttt{at}\ E\ \texttt{in}\ e \mid \texttt{cap}^{r}_{\eta}\ E \mid \texttt{new}\ v\ \texttt{at}\ E \mid \texttt{deref}\ E \mid E := e \mid v := E \mid \texttt{new}\ E\ \texttt{at}\ e \mid \texttt{pop}_{\gamma}\ E$$

Figure 2. Configuration, store, threads and evaluation contexts.

pabilities are obtained by splitting pure or other impure capabilities into several pieces, e.g., $(3,2) = (\overline{2,1}) + (\overline{1,1})$, in the same spirit as *fractional capabilities* [10]. Splitting a linear resource into multiple pieces is particularly useful for region aliasing (e.g. the same region can 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 access the same region. Capability splitting takes place automatically with function application.

3.2 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 global hierarchy δ, an abstract *store* S and a thread map T.[2] The global hierarchy δ maps thread identifiers (n) to stacks (σ). A thread stack σ is a list of frames (γ) and represents a hierarchy of regions accessible to a thread. Each frame γ represents the portion of σ that is accessible to a function. Notice, that frames include region counts. A frame is a list of elements of the form $r^{\kappa} \triangleright \pi$, denoting that region r is associated with count κ and has parent π, which can be another region or $\bot$. Regions whose parents are $\bot$ are considered as roots in a region hierarchy. A store S maps region identifiers (ι) to heaps (H). A heap H, maps memory locations to values. A thread map T associates thread identifiers to expressions (i.e., threads).

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 [24]. 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, have released all regions used by them, and have been reduced to *unit* values, represented as (), are removed from the active thread list (rule *E-T*). Rule *E-S* reduces some thread n via the expression evaluation relation. Notice, that rule *E-S* only modifies the stack of thread n and requires that the resulting hierarchy δ' is *consistent* ($\vdash \delta'$): regions accessible to thread n should be inaccessible to other threads and regions having positive pure capabilities can only be live at a *single* stack frame of thread n.[3] Therefore, the operational semantics will get stuck if the mutual exclusion protocol is unsatisfied. Our approach differs from related work, e.g. the work of Grossman [20], where a special kind of value $junk_v$ is often used as an intermediate step when assigning a value v to a location, before the real assignment takes place, and type safety guarantees that no junk values are ever read.

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 calling mode of the application term is changed from parallel to sequential. The topmost frame of the spawning thread (γ) is split into two frames γ' and γ_1 so that the intersection of regions locked in γ' and in γ_1 is empty.[4] If it is impossible to split γ, the thread evaluation relation gets stuck. Notice, that γ_1 is an effect annotation of the function abstraction. Frame γ is then replaced by γ' and γ_1 becomes the initial frame of the new thread.

The expression evaluation relation (defined in Figure 4) rewrites tuples of the form $\sigma; S; e$, where σ is a thread local stack, S is the global store, and e is an expression. We have elided the fact that constant regions may be of the form $\iota@n$, which is a constant region ι tagged with a unique identifier n. The region application (*E-RP*) rule introduces tagged regions during substitution so as to prevent the existence duplicate region names in function effects. As mentioned earlier, effects are used as stack frames. Region application using traditional substitution could introduce region aliases in function effects and thus cause non-determinism in the expression evaluation rules: region lookup on stack frames would yield multiple regions.

Hereon, the symbol γ means "the topmost frame of the currently executed thread n". The sequential function application (*E-A*) rule splits γ into two stack frames γ_1 and γ_r such that γ_1 matches the effect expected by the lambda abstraction, and substitutes the sequence of stack frames $\gamma_r; \gamma_1$ for γ. The function body is placed within a pop construct, which is annotated with frame γ_r. A pop construct must not be contained in the original program, and must only appear during program evaluation. Rule *E-E* eliminates pop constructs, when the function body has been reduced to a value and the annotation γ_r of pop matches the frame preceding the topmost frame γ'. Frames γ_r are γ' are joined to form a new frame γ'', which replaces them on the current stack.

The remaining rules make use the judgements $is\_live(\gamma, r)$ and $is\_accessible(\gamma, r)$ (Figure 7) to establish that a region r is *live* and *accessible* in a frame γ. A region r is *live* in γ when the region count of each region in the path between r and the root region is positive. A region r is *accessible* in γ when it is *live* and there exists at least one region in the path between r and the root region with a positive lock count. We also define the following partial functions: $\bar{r}$ removes the unique identifier from a tagged region, $[\![\eta]\!](\kappa)$ decrements or increments the region or lock field of κ by one, according to operation specified by η, and finally live(γ) selects a subset of γ so that all regions in that subset are *live*.

Rule *E-NG* requires that region r is *live* in γ, adds a fresh and empty region ι to S and adds the dynamic effect of ι to γ, which specifies that r is the parent of ι and that ι has region and lock count of one. Rule *E-C* requires that region r is *live* in γ, substitutes

[2] The order of elements in comma-separated lists, e.g. in a store S or in a list of threads T, is unimportant; we consider all list permutations as equivalent.

[3] The second invariant ensures that regions with positive pure capabilities can safely be passed to other threads (e.g. locked). This is sound when the current thread has no more counts of such regions in other stack frames.

[4] The rules for splitting effects are defined in Figure 6 and discussed in Section 3.3.

$$\frac{e' \equiv (v_1\ v)^{\mathsf{par}} \quad \text{fresh } n' \qquad v_1 \equiv \lambda x.\, e \ \mathtt{as}\ \tau_1 \xrightarrow{\gamma_1 \to \gamma_2} \tau_2 \quad \mathsf{par} \vdash \gamma' = \emptyset \oplus (\gamma \ominus \gamma_1) \qquad e'' \equiv (v_1\ v)^{\mathsf{seq}} \quad \delta' = \delta'', n \mapsto \sigma; \gamma, n' \mapsto \emptyset; \gamma_1}{\delta; S; T, n : E[e'] \rightsquigarrow \delta'; S; T, n : E[()], n' : e''} \ (E\text{-}SN)$$

$$\frac{\delta = \delta'', n \mapsto \sigma \quad \sigma; S; e \to \sigma'; S'; e' \quad \delta' = \delta'', n \mapsto \sigma' \vdash \delta'}{\delta; S; T, n : E[e] \rightsquigarrow \delta'; S'; T, n : E[e']} \ (E\text{-}S) \qquad \frac{\delta = \delta', n \mapsto (\emptyset; \emptyset)}{\delta; S; T, n : () \rightsquigarrow \delta'; S; T} \ (E\text{-}T)$$

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

$$\frac{\sigma = \sigma'; \gamma \quad \mathsf{seq} \vdash \gamma = \gamma_1 \oplus \gamma_r \quad \sigma'' = \sigma'; \gamma_r; \gamma_1}{\sigma; S; ((\lambda x.\, e \ \mathtt{as}\ \tau_1 \xrightarrow{\gamma_1 \to \gamma_2} \tau_2)\ v)^{\mathsf{seq}} \to \sigma''; S; \mathtt{pop}_{\gamma_r}\ e[v/x]} \ (E\text{-}A) \qquad \frac{\sigma = \sigma'; \gamma_r; \gamma' \quad \mathsf{seq} \vdash \gamma'' = \gamma' \oplus (\gamma_r \ominus \emptyset) \quad \sigma'' = \sigma'; \gamma''}{\sigma; S; \mathtt{pop}_{\gamma_r}\ v \to \sigma''; S; v} \ (E\text{-}E)$$

$$\frac{\sigma = \sigma'; \gamma \quad \text{is\_live}(\gamma, r) \quad \text{fresh } \iota \quad \sigma'' = \sigma'; \gamma, \iota^{1,1} \triangleright r}{\sigma; S; \mathtt{newrgn}^{r}\ \rho, x\ \mathtt{at}\ \mathtt{rgn}_{\bar{r}}\ \mathtt{in}\ e \to \sigma''; S, \iota \mapsto \emptyset; e[\iota/\rho][\mathtt{rgn}_{\iota}/x]} \ (E\text{-}NG) \qquad \frac{\begin{array}{c}\sigma = \sigma'; \gamma \quad \text{is\_live}(\gamma, r) \quad \gamma = \gamma', r^{\kappa} \triangleright \pi \\ \kappa' = [\![\eta]\!](\kappa) \quad \sigma'' = \sigma'; \text{live}(\gamma', r^{\kappa'} \triangleright \pi)\end{array}}{\sigma; S; \mathtt{cap}^{r}_{\eta}\ \mathtt{rgn}_{\bar{r}} \to \sigma''; S; ()} \ (E\text{-}C)$$

$$\frac{\sigma = \sigma'; \gamma \quad \text{is\_live}(\gamma, r) \quad \text{fresh } \ell}{\sigma; S; \mathtt{new}\ v\ \mathtt{at}\ \mathtt{rgn}_{\bar{r}} \to \sigma; S[\bar{r} \mapsto S(\bar{r}), \ell \mapsto v]; \mathtt{loc}_{\ell}} \ (E\text{-}NR) \qquad \frac{\sigma = \sigma'; \gamma \quad \text{is\_accessible}(\gamma, r) \quad (\ell \mapsto v_1) \in S(\bar{r})}{\sigma; S; \mathtt{loc}_{\ell} := v \to \sigma; S(\bar{r})[\ell \mapsto v]; ()} \ (E\text{-}AS)$$

$$\frac{\sigma = \sigma'; \gamma \quad \text{is\_accessible}(\gamma, r) \quad (\ell \mapsto v) \in S(\bar{r})}{\sigma; S; \mathtt{deref}\ \mathtt{loc}_{\ell} \to \sigma; S; v} \ (E\text{-}D) \qquad \frac{\text{fresh } n'}{\sigma; S; (\Lambda\rho.\, f)[r] \to \sigma; S; f[\bar{r}@n'/\rho]} \ (E\text{-}RP)$$

Figure 4. Expression evaluation relation $\sigma; S; e \to \sigma'; S'; e'$.

$[\![\eta]\!](\kappa)$ for κ in γ at the exponent of r, and removes dead regions from the resulting frame. Rule *E-NR* requires that region r is *live* in γ and updates the heap of r with a fresh location ℓ mapping to value v. Notice, that r may be *unlocked*. Rules *E-AS* and *E-D* require that *some region* r, which contains the location (ℓ) being accessed, must be *accessible* in γ. Therefore, the semantics will get stuck when a thread attempts to access a memory location without having acquired an appropriate lock for this location.

3.3 Static semantics

We discuss the most interesting aspects of our type system. We employ a *type and effect system* to enforce memory and race safety invariants. Effects (γ) are used to statically track region capabilities.

The syntax of types is defined in Figure 1. A collection of base types b is assumed; the syntax of values belonging to these types and operations upon such values are omitted from this paper. We assume the existence of a *unit* base type, which we denote by $\langle\rangle$. Region handle types $\mathtt{rgn}(r)$ and reference types $\mathtt{ref}(\tau, r)$ are associated with a type-level region r. Monomorphic function types carry an *input* and an *output effect*. A well-typed expression e has a type τ under an input effect γ and results in an output effect γ'. The typing relation (see Figure 5) is denoted by $R; M; \Delta; \Gamma \vdash e : \tau \,\&\, (\gamma;\gamma')$ and uses four typing contexts: a set of region literals (R), a mapping of locations to types (M), a set of region variables (Δ), and a mapping of term variables to types (Γ).

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

- the liveness invariant for γ'', i.e., $\gamma'' = \mathit{live}(\gamma')$;
- the consistency of γ and γ'', i.e., regions cannot change parent and capabilities cannot switch from pure to impure or vice versa; the domain of γ'' is a subset of the domain of γ (i.e rules *ES-C* and *CS*);
- regions having pure capabilities must appear once in the function input (γ_1) and output (γ_2) effects, i.e., $ok(\gamma_1; \gamma_2)$. Furthermore, the capabilities required by a function's input effect (γ_1) must be present in the environment (γ — rule *ES-C*).
- for parallel application, the thread output effect must be empty, the thread input effect must not contain impure capabilities with positive lock counts.
- finally $a \simeq b$ holds when by erasing region tags from a and b (i.e., replacing $\iota@n$ with ι), we obtain identical results.

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*. Notice that effect typing is left-to-right, which is consistent with the left-to-right evaluation in the operational semantics. The output effect of the rightmost subexpression of each construct is always (except for rule *T-NG*) used for checking the liveness and accessibility invariants. The rule for creating new regions (*T-NG*) checks that e_1 is a handle for some live region r'. Expression e_2 is type checked in an extended typing context (i.e., ρ and $x : \mathtt{rgn}(\rho)$ are appended to Δ and Γ respectively) and an extended input effect (i.e., a new effect is appended to the input effect such that the new region is live and accessible to this thread). The rule also checks that the type and the output effect of e_2 do not contain any occurrence of region variable ρ. This implies that ρ must be *consumed* by the end of the scope of e_2. The capability manipulation rule (*T-CP*) checks that e is a handle of a live region r. It then modifies the capability count of r as dictated by function $[\![\eta]\!]$, which increases or decreases the region or the lock count of its argument, according to the value of η. The dynamic semantics ensures that an operational step is performed if the updated hierarchy preserves the invariant that protected regions are accessible to a single thread at instance of time. For instance, if the lock of region r is held by some other executing thread, the evaluation of $\mathtt{cap}_{\mathsf{lk}+}$ must be suspended until the lock can be obtained. On the other hand, the evaluation of $\mathtt{cap}_{\mathsf{rg}-}$ does not need to suspend but may not be able to physically deallocate a region, as it may be used by other threads.

$$\frac{R;M;\Delta;\Gamma \vdash e_1 : \tau_1 \xrightarrow{\gamma_1 \to \gamma_2} \tau_2 \,\&\, (\gamma;\gamma_3) \qquad \xi = \mathsf{par} \Rightarrow \tau_2 = \langle\rangle \qquad R;M;\Delta;\Gamma \vdash e_2 : \tau_1 \,\&\, (\gamma_3;\gamma_4) \qquad \xi \vdash \gamma_5 = \gamma_2 \oplus (\gamma_4 \ominus \gamma_1)}{R;M;\Delta;\Gamma \vdash (e_1\ e_2)^{\xi} : \tau_2 \,\&\, (\gamma;\gamma_5)}\ \textit{(T-AP)}$$

$$\frac{R;M;\Delta;\Gamma \vdash e : \mathtt{ref}(\tau, r) \,\&\, (\gamma;\gamma') \qquad \mathit{is\_accessible}(\gamma', r)}{R;M;\Delta;\Gamma \vdash \mathtt{deref}\ e : \tau \,\&\, (\gamma;\gamma')}\ \textit{(T-D)} \qquad \frac{R;M;\Delta;\Gamma \vdash e_1 : \mathtt{rgn}(r) \,\&\, (\gamma;\gamma') \qquad \mathit{is\_live}(\gamma', r) \qquad R;\Delta \vdash \tau \qquad R;M;\Delta,\rho;\Gamma, x : \mathtt{rgn}(\rho) \vdash e_2 : \tau \,\&\, (\gamma', \rho^{1,1} \triangleright r;\gamma'') \qquad \rho \notin dom(\gamma'')}{R;M;\Delta;\Gamma \vdash \mathtt{newrgn}^r\ \rho, x\ \mathtt{at}\ e_1\ \mathtt{in}\ e_2 : \tau \,\&\, (\gamma;\gamma'')}\ \textit{(T-NG)}$$

$$\frac{R;M;\Delta;\Gamma \vdash e_1 : \tau \,\&\, (\gamma;\gamma') \qquad \mathit{is\_live}(\gamma'', r) \qquad R;M;\Delta;\Gamma \vdash e_2 : \mathtt{rgn}(r) \,\&\, (\gamma';\gamma'')}{R;M;\Delta;\Gamma \vdash \mathtt{new}\ e_1\ \mathtt{at}\ e_2 : \mathtt{ref}(\tau, r) \,\&\, (\gamma;\gamma'')}\ \textit{(T-NR)} \qquad \frac{\mathit{is\_live}(\gamma', r^{\kappa} \triangleright \pi, r) \qquad \gamma'' = \mathit{live}(\gamma', r^{\kappa'} \triangleright \pi) \qquad R;M;\Delta;\Gamma \vdash e_1 : \mathtt{rgn}(r) \,\&\, (\gamma;\gamma', r^{\kappa} \triangleright \pi) \qquad \kappa' = [\![\eta]\!](\kappa)}{R;M;\Delta;\Gamma \vdash \mathtt{cap}^r_{\eta}\ e_1 : \langle\rangle \,\&\, (\gamma;\gamma'')}\ \textit{(T-CP)}$$

Figure 5. Selected typing rules.

$$\frac{}{\xi \vdash \gamma = \emptyset \oplus \gamma}\ \textit{(ES-N)} \qquad \frac{\xi \vdash \gamma, r^{\kappa_2} \triangleright \pi = \gamma_1 \oplus \gamma_2 \qquad \xi \vdash \kappa = \kappa_1 + \kappa_2 \qquad \pi \simeq \pi' \qquad r' \simeq r}{\xi \vdash \gamma, r^{\kappa} \triangleright \pi = \gamma_1, r'^{\kappa_1} \triangleright \pi' \oplus \gamma_2}\ \textit{(ES-C)} \qquad \frac{\xi \vdash \gamma = \gamma_1 \oplus \gamma_r \qquad \xi \vdash \gamma' = \gamma_2 \oplus \gamma_r \qquad \gamma'' = \mathit{live}(\gamma') \qquad \mathit{ok}(\gamma_1;\gamma_2) \qquad \xi = \mathsf{par} \Rightarrow \gamma_2 = \emptyset}{\xi \vdash \gamma'' = \gamma_2 \oplus (\gamma \ominus \gamma_1)}\ \textit{(ESJ)}$$

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

Figure 6. Effect and capability splitting.

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

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

Figure 7. Auxiliary predicates: region liveness and accessibility.

The type safety formulation is based on proving the *preservation* and *progress* lemmata.[5] Deadlocked threads are not considered to be stuck. A well-typed configuration $\delta; S; T$ 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,the *progress* and *preservation* lemmata definitions are standard.

4. Interaction with Cyclone

We have integrated the type system and operational semantics presented in earlier sections to Cyclone. In this section, we provide an in-depth description of the interaction between our system and Cyclone. We have identified five important goals that the integration should accomplish or preserve:

- Memory safety: dangling and null pointer dereferences as well as buffer overruns should be prevented.
- Thread safety: The type system should guarantee that shared data accesses should be race free. Furthermore, the run-time system must be re-entrant and thread safe. We defer the discussion about implementation issues until the next section.
- Separate compilation: it should be possible to compile and link separate modules independently.
- Backwards compatibility: sequential Cyclone code should work as expected with no further modifications.
- Allow accessing local data without synchronization: thread-local data is often the rule, not the exception. It should be possible to access thread-local data with no additional overhead (e.g. synchronization).

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

4.1 Extended regions

In contrast with traditional lexically scoped regions, which are allocated in a LIFO manner, our *extended* regions can be allocated at any extended region ancestor. We consider the main heap (‘H) as the root of our region hierarchy. Thus, the *heap_handle* can also be used for allocating an extended region.

The following example illustrates this point. In particular, we allocate a fresh extended region (‘*child*) within an existing region ‘*parent*. The type system ensures that ‘*parent* is *live* at the allocation point, but not necessarily *accessible*.

```
{ region child @ parent   // add ‘child^{1,1} ▷ ‘parent to effect
  ...
  xdec(child);            // remove ‘child from effect
  ...
}
```

As in traditional regions, *child* is the handle to the fresh region. This form of allocation generalizes the stack-based region organization to a tree-based organization and enables finer-grained control of region lifetimes.

As in the operational semantics, region ‘*child* is a *sharable* region, but no synchronization is required for accessing its data, as it is initially *accessible* to the thread allocating ‘*child*.

4.2 Kind system

Our current design draws a line between our regions and traditional regions. In this way, we are able to restrict what *kinds* of regions can be shared. Traditional lexically scoped regions cannot be shared

safely. For instance, the *stack frame* of a function is treated as a region and sharing the stack in a safe manner would have a severe impact on concurrency between threads.

The type system of Cyclone uses *kinds* to group types. Traditional region type variables are of kind R. As mentioned earier, such regions cannot be shared among threads. In order to distinguish between sharable and unsharable regions, we have introduced a new kind X as a subkind of R. Subkinding allows regions of kind X to be treated as regions of kind R. However, our regions differ in respect to traditional Cyclone regions in that they can be unlocked or released at *any* program point. Therefore, we disallow function calls, which treat extended region effects as traditional Cyclone effects. We expect future implementations to allow a higher degree of interaction between regions of different kinds.

4.3 Traditional Cyclone effects

The *effect* system of Cyclone tracks the set of *accessible* regions at each program point. Functions are annotated with a *single* effect, which can be automatically inferred by calculating the union of region variables occuring in the types of function parameters. As mentioned, traditional regions cannot be deallocated once they have been added in a function's effect, as unrestricted region aliasing within an effect is admitted. Therefore, a function effect serves as both a precondition and a postcondition of the regions that are accessible before and after calling a function respectively.

The following example illustractes a function, which has been annotated explicitly with the effect $\{\text{`}r\}$. This effect implies that region `$r$ is both *live* and *accessible* for the entire scope of foo.

```
void foo ( region_t < `r > h ; {`r});
```

We have decided to place our effects in separate annotations as full effect inference for our regions is beyond the scope of this work and we wish to preseve backwards compatibility with traditional Cyclone programs that enjoy full inference.

Our effects are mutually exclusive with traditional effects so a region name may not exist in both effects. We expect that future implementations will integrate the two different kinds of effects into a single effect and will enjoy effect inference for both kinds of regions. For instance, it is possible to have a single effect for all kinds of regions, by using positive and negative capabilities.

The following example shows how we can write function *foo* so that it uses the extended regions:

```
void foo ( region_t < `r > h ) @ieffect({`r, i(1,1), `H})
                               @oeffect({`r, i(1,1), `H});
```

The `@ieffect` and `@oeffect` annotations denote the input and output effects of function *foo*. These effects consist solely of extended regions or `H.[6] The equivalent type of function *foo* in our formal type system would be:

$$\forall \text{`}r.\, \texttt{region\_t} < \text{`}r > \xrightarrow{\gamma \to \gamma} \langle\rangle \quad \text{where } \gamma \equiv \text{`}r^{\overline{\mathtt{I},\mathtt{I}}} \triangleright \bot$$

The heap region is mapped to $\bot$ as it is immortal. The impure capability $\overline{n_1, n_2}$ maps to $\mathtt{i}(n_1, n_2)$, whereas the pure capability n_1, n_2 maps to $\mathtt{p}(n_1, n_2)$. Pure capabilities are most useful when transferring lock capabilites to other threads. It is therefore expected that impure capabilities would be the common case. Therefore, the above definition can be abbreviated as follows:

```
void foo( region_t < `r > h ) @ieffect({`r, 1, 1, `H})
                              @oeffect({`r, 1, 1, `H});
```

Finally, it is possible to omit the output effect annotation when the functions consumes the regions declared at the input effect.

[6] `H can only occur as a parent annotation.

4.4 Hierarchy abstraction

In order to allow a function to access a region without having to pass all its ancestors explicitly, its ancestors can be abstracted from an effect for the duration of a function call. To maintain soundness, we require that 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. The definition of *foo* can be further simplified, by using hierarchy abstraction:

```
void foo ( region_t < `r > h ) @ieffect({`r, 1, 1})
                               @oeffect({`r, 1, 1});
```

4.5 Operating on capabilities

The `cap` operator of the formal semantics has been encoded as a set of library functions:

```
void xdec ( region_t <`r>; ) @ieffect(`r, 1, 0);
void xinc  ( region_t <`r>; ) @ieffect(`r, 1, 0)
                              @oeffect(`r, 2, 0);
void xldec( region_t <`r>; ) @ieffect(`r, 1, 1)
                              @oeffect(`r, 1, 0);
void xlinc (region_t <`r>; ) @ieffect(`r, 1, 0)
                              @oeffect(`r, 1, 1);
```

For instance, the first function *xdec* encodes the operator $\mathtt{cap}_{\mathsf{rg}-}$ for any region `$r$. It requires that the calling context has at least one region capability. This invariant is encoded in its input effect. The output effect of *xdec* is omitted, thus exactly one region capability is consumed. Similary the remaning functions encode the remaining functionality of operator `cap`. It would be preferable to use dependent types to express arbitrary additions or subtractions. To the best of our knowledge this is impossible to express at the type level, in Cyclone's type system. However, we plan on extending the type-level expressiveness in future versions.

4.6 Exceptions

Having static guarantees about the control flow of a program plays a crucial role in manual memory management. As mentioned in earlier sections, Cyclone allows memory leaks[7] of tracked pointers even when they point at dynamic regions.

We decided that our regions should *always* be reclaimed manually. Towards this goal, we have made possible to annotate Cyclone function declarations with uncaught exception names that may be thrown from a function's body.[8] We have not opted for an interprocedural analysis as this would violate separate compilation.

In addition, exact knowledge of a function's control-flow graph is required to guarantee soundness: if the body of a function does not satisfy the `@oeffect` postcondition of that function (as a result of a statically unknown exception), then it is possible to introduce dangling pointers. There exist three kinds of annotations for exceptions:

- the `@throws`(...) clause enumerates all exceptions that may be thrown from a function body.
- the `@nothrow` annotation is an abbreviation for `@throws`().
- finally, `@throwsany` acts as a wildcard for any exception that may be thrown. This annotation may be useful for legacy library prototypes and code.

The default annotation for functions is `@throwsany`. Exceptions may be thrown *explicitly* by the programmer or *implicitly* by the run-time system. Implicit exceptions arise in situations where:

[7] These pointers will eventually be reclaimed by the garbage collector.

[8] We have noticed that Cyclone has a `@throws` clause but it is undocumented and not functioning.

- a *null* pointer is dereferenced.
- an *out of bounds* array access is performed.
- the system has *insufficent memory* to fulfill an allocation request.
- a value *cannot be matched* against any of the available patterns.

The exception analysis takes into consideration both explicit and implicit exceptions. In the future we plan on relaxing the exception analysis and adding run-time support to ensure that function postconditions are always satisfied.

4.7 Re-entrant functions

Global data are implicitly shared by all threads and this may cause a data race. To preserve race-freedom, we have constrained our language so that it only admits access to *sharable* regions.[9] Thus, traditional regions cannot be passed to a new thread. To enforce, this policy we require that each explictly spawned thread must be declared as `@re_entrant`. In addition, a function annotated as `@re_entrant` yields access to global variables, the immortal heap, tracked objects and it can only invoke `@re_entrant` functions. Function *main*, is not `@re_entrant`. Global data and tracked objects can still be directly accessed by any non-reentrant function invoked directly or indirectly by function *main*. Therefore, sequential programs have full access to global data. In the future, we intend to relax type-checking so that tracked objects can be passed to threads, provided that these objects are consumed from the environment performing a spawn operation.

4.8 Thread creation

Threads can be explicitly created by the means of the *spawn* operator. This operator takes two expressions e_1 and e_2, i.e., *spawn* (e_1) e_2), and spawns a new thread. The first expression is a list of thread-specific parameters such as the stack size. The second expression e_2 must be a function call and the function must be annotated as `@re_entrant@nothrow` and its `@oeffect` annotation must be either empty or omitted. Further, the traditional Cyclone effect must be *empty* so that unsharable regions cannot be used in the new thread. Both expressions e_1 and e_2 are evaluated from left to right. The spawning thread does not block and returns immediately.

4.9 Type polymorphism

Cyclone effects are not polymorphic. To allow the invocation of functions, which have polymorphic arguments (e.g., say ‘*a*), Cyclone programmers use the `regions`(‘*a*) operator. The purpose of this operator is to defer effect checking until the function call is performed, where the calling environment must prove that all regions occuring in the type that instantiates ‘*a* are present in the environment's effect:

`void` *foo* (‘*a* ; `regions`(‘*a*));

In terms of extended regions, the `regions` operator would require that all regions occuring in ‘*a* are *live* and *accessible* for the scope of the function call. However, this is beyond the scope of our type system. Further,we cannot provably guarantee memory safety if this construct is used in the way described above. Therefore, the type checker disallows uses of `regions` operator within our effects.

This limitation could be improved in future work but as a workaround we allow extended regions to interoperate with traditional regions. We explain this feature in the section that follows.

[9] In practice, we allow reading global variables that are declared as constant.

4.10 Interoperability with traditional regions

The distinction between traditional and extended regions may be limiting for programs that require both kinds of regions. We introduce a language construct similar to the `alias` and `open` constructs of Cyclone, that borrows a part (or a fraction) of an accessible extended region for a certain scope. Consider the following example:

```
{ region child @ parent;
  { region h = xopen(child);  // consume one lock capability
                  ...
  } // restore lock capability
  xdec(child);
}
```

The `xopen` construct *borrows* exactly one lock capability from the extended region ‘*child* for the scope of the `xopen` construct. The type system requires that region ‘*child* is *live* by the end of the *xopen* scope and creates a fresh logical region ‘*h*, which can be used as a traditional Cyclone region. It should be noted that ‘*child* is *still* live and possibly accessible (if it has more than one lock capability) during the scope of `xopen`. On the downside, region ‘*child must* remain locked for the scope of `xopen`.

4.11 Memory consistency

Our formal language semantics assumes a *sequentially consistent* memory model [24], which implies that concurrent read and write operations are viewed as an interleaving of *atomic* steps.

Modern, processors are implemented with much weaker memory consistency specifications as sequential consistency restricts common compiler and hardware optimizations. Research on relaxed memory models [1, 19] has shown that *race-free*[10] programs running on relaxed memory systems have a sequentially consistent view of memory operations.

Assuming that the compilation process preserves the original Cyclone code semantics, then we obtain *race-free* native code with sequential consistency guarantees. At the implementation level, we must guarantee that memory operations to extended regions cannot escape the scope of a “lock/unlock” primitive as locking operations *synchronize memory*. This situation may arise as a result of compiler optimizations such as *register promotion* [7]. We have taken the most conservative approach and require that extended region data objects are compiled down to C as *volatile*.[11] According to the GCC manual, “*an implementation is free to reorder and combine volatile accesses which occur between sequence points, but cannot do so for accesses across a sequence point*” [17]. Our locking primitives are translated to sequence points and thus the compilation process will not reorder volatile accesses in an unsafe manner.

5. Implementation

5.1 Compiler

We have implemented extended region checking as a separate compiler pass in Cyclone. First, the type well-formedness of our annotations (effects, exceptions, types) is checked. During type checking, we disregard control-flow and verify that the extended regions being accessed exist in a function's scope (i.e., by inspecting the `@ieffect` annotation and tracking fresh regions). This allows us to catch common errors early. Once type checking is finished, the compiler enters the static analysis stage where it performs data- and control-flow analyses and determines where dynamic checks should be placed.

[10] Read and write operations to shared memory locations only occur within memory synchronization primitives.

[11] GCC is invoked by the Cyclone compiler to generate native code.

As illustrated in Section 2, memory access checks in concurrent programs should be optimized away when the accessed memory is thread-local or shared and protected by a lock. The compiler eliminates such checks by utilizing programmer-inserted checks. In this case, the analysis should ensure that the lock is not released between the programmer-inserted check and the memory access location. Our current implementation, is highly conservative and only allows dynamic check elimination for trivial cases of shared memory accesses.

An exception analysis, utilizing information from the previous passes about where *implicit* exceptions may be thrown, performs a control-flow sensitive analysis to verify that uncaught exceptions that *may* be thrown from a function body are included in the function's `@throws` specification.

Finally, a control-flow sensitive analysis is performed. The analysis propagates effects through the control flow graph and verifies that the output effect of the function body matches the function's `@oeffect` specification. This analysis also utilizes function attributes, when checking function calls. For instance, effects are not propagated from function calls that never return to the calling context (i.e., `__attribute((noreturn))__`).

5.2 Code generation and run-time system

As discussed in the Section 3.2, each thread *must* maintain a local view of the hierarchy. Otherwise, some regions of the hierarchy may become falsely shared and thus reduce concurrency between threads.

Consider the case where a thread owning an unlocked region ρ *shares* that region with a new thread, which in turn allocates a fresh region ρ_1 at region ρ. The second thread uses ρ_1 locally and then deallocates it. With respect to the global view of the hierarchy, ρ_1 is *owned* by the second thread. If the first thread attempts to lock ρ, then it will have to block until ρ_1 is deallocated by the second thread. Here we use the notion of a global view for simplicity. The implementation only makes use of local views.

Deallocating regions *en masse* may temporarily cause a *region leak*, if hierarchies passed to functions are not tracked dynamically. (Note that the leak is temporary because the region containing the leaked regions will eventually deallocate them and roots of the hierarchy cannot leak.)

At the type-level, when a non-leaf node of a hierarchy is removed from the current effect, then the the entire subtree of that node will be removed from the effect. If the run-time system only decrements the reference count of the node being removed, then the node's subtree may leak. If the node being removed is pure, then it is safe to deallocate its subtree from the local hierarchy without requiring additional information. Otherwise, the compiler has two options: issue a warning about a possible leak or generate code that dynamically tracks the hierarchy passed to a function. That is, the compiler could preserve an exact correspondence between run-time views and static effects. The advantage of the second approach is that it prevents temporary memory leaks. On the downside, it places an overhead for each function call that uses non-trivial hierarchies.

The current implementation strictly adheres to the formal semantics and implements the second option. As an optimization, we avoid code generation for function calls that use hierarchies of height one. However, it is entirely possible to allow the programmer to decide whether such leaks should be prevented, by adding annotations to functions (e.g. `@noleak`), or by introducing new compiler flags.

In the paragraphs that follow we discuss how the code generator assists the run-time system with type information so that the it can disallow *false sharing* and prevent *region leaks*. We also discuss about new features that have been added to the run-time system.

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

- Translate `spawn` statements to low-level primitives, which require (un)packing of the function arguments and placing the call into a wrapper function, which acts as a glue between the call and thread that will executing it.
- Generate specialized code for allocating extended regions and references.
- Generate code for allowing "dynamic effect tracking" before *some* function calls. The sub-tree passed to a call is not actually copied. Instead we use a form of dynamic scoping (shallow binding in particular) so as to map type-level region names to nodes of the local tree. Each dynamic scope is pushed into the virtual stack frame of the run-time system. An additional `pop` statement is added after the call.

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

- It registers fresh regions to the local thread hierarchy in which they are allocated.
- When a subtree has to be deallocated, it uses the dynamic scoping structures to retrieve nodes of the local hierarchy and update their dynamic counts accordingly. Notice that all remaining region locks are released during the deallocation phase.
- The implementation of `spawn` uses a similar technique to construct the subtree passed to new thread and makes this tree accessible to the new thread. It also performs capability accounting tasks so that the dynamic trees of both threads match the static effects.
- Region locking is implemented in a straightforward manner by traversing the local hierarchy. To avoid deadlocks, subtrees are always locked in a top-down left-to-right manner.
- The region allocation subsystem has been re-engineered so that it can serve concurrent allocation requests in a non-blocking manner (i.e., using atomic operations).

6. Performance evaluation

We evaluated our implementation on five concurrent benchmark programs, taken from "The Computer Language Benchmarks Game" (`http://shootout.alioth.debian.org/u32q/`). As a basis for our evaluation we used the fastest version of the programs in C, with one exception mentioned below, which we translated to our language as directly as possible. The results are summarized in Table 1. The five benchmark programs were:[12]

binary-trees a program that allocates, traverses and deallocates many binary trees. The original program (#7) uses GCC OpenMP and memory pools, as implemented in the Apache Portable Runtime Library.

chameneos-redux a program that simulates the interaction of a number of creatures, using symmetrical thread rendez-vous. Our basis for the comparison is the second fastest version in C (#2); it uses pthreads and mutex locks. On our testing machine, it only produced the correct result when compiled with `-O2` and we compiled our program with the same option. The fastest version in C (#5) uses the processor's "compare and swap" instruction, instead of locks, and explicitly schedules threads to processor cores; it cannot be translated directly to our language.

[12] Our implementation and the benchmark programs are available from: `http://www.softlab.ntua.gr/~pgerakios/cyc_reglock.tgz`.

lang	CPU (s)	memory (KB)	load per core (%)				elapsed (s)	factor
Benchmark: **binary-trees**								
gcc	21.34	100,688	46	88	50	89	7.54	1.00
cyc	23.88	122,412	73	81	88	79	7.21	0.96
Benchmark: **chameneos-redux**								
gcc	56.38	576	68	90	72	91	17.04	1.00
cyc	276.08	1,112	85	99	84	100	76.69	4.50
Benchmark: **fannkuch**								
gcc	152.69	572	97	100	97	97	39.18	1.00
cyc	177.28	1,032	99	99	100	99	44.63	1.14
Benchmark: **mandelbrot**								
gcc	24.24	28,260	100	99	100	100	8.13	1.00
cyc	46.21	32,176	95	95	95	97	16.15	1.99
Benchmark: **thread-ring**								
gcc	143.09	4,536	26	38	24	16	133.04	1.00
cyc	254.38	18,108	13	49	22	16	246.95	1.86

Table 1. Performance overhead, compared to GCC, for five benchmarks taken from "The Computer Language Benchmarks Game."

fannkuch a program that performs indexed access to small sequences of integer numbers. The original program (#2) uses pthreads. On our testing machine, it only produced the correct result when compiled with optimization turned off and we did the same for our program.

mandelbrot a program that plots a bitmap of the Mandelbrot set. The original program (#6) uses pthreads and special SSE2 128-bit floating-point instructions. Our translation implements the same algorithm but is based on a simpler C# version of the program, using normal double precision numbers.

thread-ring a program that creates a large number of threads, organized in a ring, and repeatedly passes a token from one thread to the next. The original program (#1) uses pthreads and mutex locks. (We should mention that the original program performs very poorly, compared to versions in other languages.)

The testing machine is a quad-core 2.4GHz Intel, with 2GB of RAM, running a Linux 2.6.30 kernel. Our implementation used GCC 4.3.2 as a back end, which was also used to compile the C programs. We used `-O3`, except as explained above. In our Cyclone implementation we turned off Boehm's garbage collector, which is only used for Cyclone's original regions and is not need for these benchmarks.

As shown in Table 1, the benchmark programs fall in three categories. First, in binary-trees and fannkuch, the Cyclone program runs a little faster (7%) and a little slower (15%) than the original C program, respectively. Especially for the case of binary-trees, this result is particularly interesting as the two compared programs use both the same algorithm and the same region-based memory management scheme. Second, in mandelbrot and thread-ring, the Cyclone program runs almost twice as slow, by a factor of 86% and 99%, respectively. In the case of mandelbrot, this is attributed to our Cyclone's inability to exploit the CPU's specialized number crunching instructions and, we believe, is not very interesting for the purposes of this paper. On the other hand, thread ring is an extreme case of benchmark, stressing thread communication; we believe that we can achieve better results here by further tuning the performance of our locks.

Third, we observe a very heavy performance penalty in chameneos-redux, which runs 4.5 times slower than the original program. This is the most interesting of our benchmarks, as it reports an inherent limitation of locking supported by the type system. The original program uses one lock for the meeting place, where the creatures meet. In addition, our program also uses a second lock for the entire array holding the creatures' data. In our program, the array must be locked because it is not possible to convince the type system that the creature who waits in the meeting room *will never* access its data, but this will be updated by its peer and therefore no data race will occur. We believe that it is this second lock which causes the performance penalty.

7. Future work

The benchmarks of the previous section show that Cyclone extended with our language constructs provides static memory safety guarantees without significantly compromizing performance compared with optimized C. Still, there are plenty of optimizations and improvements that could be done to our implementation. Here, we identify the most important ones according to our current benchmarking experiences.

Waiting for threads. A lexically-scoped Cilk-like [14] construct for allowing parent threads to wait for the children threads to terminate would be highly desirable:

```
join {
    for (int i = 0; i < size; i++)
        spawn worker(a[i]);
}
```

The compiler and run-time system could utilize *join* information so as to make better scheduling decisions.

Read only data. The multiple readers, single writer lock idiom has proven to be invaluable. Concurrent applications often use data structures as read-only for certain parts of a concurrent computation. Our type system can straightforwardly be extended to support this idiom, by introducing a new dimension for our capabilities:

$$\kappa' \;:=\; n, n \mid \overline{n, n}$$
$$\kappa \;:=\; \mathtt{ro}(\kappa') \mid \mathtt{rw}(\kappa')$$

A region capability can be *read only* (i.e., $ro(\kappa')$) or read/write (i.e., $rw(\kappa')$), which which permit read only operations or read/write operations to a region respectively.

Finer grained locking. For certain applications it would be preferable to associate locks to individual references as opposed to regions. It is possible to extend our system to support finer-grained locking by blurring the distinction between regions and references. That is, a fresh region effect could be assigned to new lockable references. In turn, this could be implemented by introducing a new language construct or utilizing existing methods such as tracked pointers and existential types (i.e., a similar mechanism to dynamic regions). At run-time the new reference would be allocated in the parent region's space and explicit deallocation would still be possible by using reaps.

Run-time system improvements. Undoubtedly, there are plenty of optimizations that could be performed in our run-time system. However, we strongly believe that the most important feature that should be added is scheduling support for efficiently mapping kernel threads to processors. A desirable feature would be to schedule tightly-coupled threads sharing the same regions on the same processor. The integration of cooperative threads (and possibly adding language support for such threads) would be highly desirable as the cooperative model seems to be highly scalable for event-based applications [29].

8. Related work

The first statically checked stack-based region system was developed by Tofte and Talpin [28]. Since then, several memory-safe systems that enabled early region deallocation for a sequential language were proposed [2, 13, 22, 30]. RC [15] and Cyclone [21] 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 multi-threaded programs. Grossman proposed a type system for safe multi-threading in Cyclone [20]. Race freedom is guaranteed by statically tracking locksets within lexically-scoped synchronization constructs. Grossman's proposal allows for fine-grained locking, but 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. In addition, Grossman's system provides no support for data migration or lock transfers.

Statically checked region systems have also been proposed [9, 31, 32] for real-time Java to rule out dynamic checks imposed by its specification. Boyapati et al. [9] introduce hierarchical regions in ownership types but the approach suffers from similar disadvantages as Grossman's work. Additionally, their type system only allows sub-regions for *shared* regions, whereas we do not have this limitation. Boyapati also proposed an ownership-based type system that prevents deadlocks and data races [8]; 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 [32]. Their main advantage is that programs require fewer annotations compared to programs with explicit region constructs. In the same track, Zhao et al. [31] 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.

At the program verification side, Concurrent C minor [23] is a concurrent version of C with threads, shared memory and first-class locks, which uses a variant of separation logic to reason about programs. Even though, their specifications are finer-grained and more flexible than our type system, they require interactive proofs.

Many systems, such as Safe-C [6], CCured [25], and Deputy [11], aim to make C code safe. Some of these systems drop soundness guarantees, so as to reduce the annotation burden, drop the explicit memory representation of C programs or provide no guarantees for concurrent programs. There also are numerous of static analyses for avoiding data races [26] and valid data sharing [4, 5] for C programs. However, these systems drop soundness guarantees, so as to reduce the annotation burden and require whole program analyses, which are not always possible in the presence of libraries and legacy code.

There are numerous languages at the C level of abstraction with explicit concurrency features [3, 14, 16] but to the best of our knowledge none of them provides both memory safety and race freedom guarantees.

9. Concluding remarks

We have presented the design and implementation of a formal, concurrent language employing region-based memory management and locking primitives to Cyclone, a variant of C that guarantees memory safety for sequential programs. The formal language provides memory safety and race freedom guarantees for well-typed programs. We discussed the integration of the formal language to Cyclone and argued about the decisions that we have made in order to guarantee memory safety and race freedom at the implementation level. Finally, we evaluated the performance of our programs against highly optimized C programs and report the results.

The main contribution of our work is the development of an extension to Cyclone that supports race-free and memory-safe multi-threading. To the best of our knowledge, it is the first variant of Cyclone that has been implemented with these properties, and one of the very few programming languages at this level of abstraction that have been designed and implemented with this goal in mind. Although our implementation is still not very mature, the benchmark results that were reported are acceptable and promising.

References

[1] S. V. Adve and M. D. Hill. Weak ordering – a new definition. In *Proceedings of the Annual International Symposium on Computer Architecture*, pages 2–14, New York, NY, USA, 1990. ACM.

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

[3] T. Anderson, N. Glew, P. Guo, B. T. Lewis, W. Liu, Z. Liu, L. Petersen, M. Rajagopalan, J. M. Stichnoth, G. Wu, and D. Zhang. Pillar: A parallel implementation language. In *Proceedings of the International Workshop on Languages and Compilers for Parallel Computing*, pages 141–155, Berlin, Heidelberg, 2008. Springer-Verlag.

[4] Z. R. Anderson, D. Gay, R. Ennals, and E. Brewer. SharC: Checking data sharing strategies for multithreaded C. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 149–158, New York, NY, USA, 2008. ACM.

[5] Z. R. Anderson, D. Gay, and M. Naik. Lightweight annotations for controlling sharing in concurrent data structures. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 98–109, New York, NY, USA, 2009. ACM.

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

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

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

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

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

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

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

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

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

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

[16] D. Gay, P. Levis, J. R. von Behren, M. Welsh, E. A. Brewer, and D. E. Culler. The nesC language: A holistic approach to networked embedded systems. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–11, New York, NY, USA, 2003. ACM.

[17] *GCC Online Documentation*. http://gcc.gnu.org/onlinedocs/.

[18] P. Gerakios, N. Papaspyrou, and K. Sagonas. A concurrent language with a uniform treatment of regions and locks. In *Preliminary Proceedings of the Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software*, 2009. See also technical report available at http://lazy.softlab.ntua.gr/~pgerakios/papers/reglock_techrep09.pdf.

[19] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the Annual International Symposium on Computer Architecture*, pages 15–26, 1990.

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

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

[22] F. Henglein, H. Makholm, and H. Niss. A direct approach to control-flow sensitive region-based memory management. In *Proceedings of the 3rd ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, pages 175–186, New York, NY, USA, 2001. ACM.

[23] A. Hobor, A. W. Appel, and F. Z. Nardelli. Oracle semantics for concurrent separation logic. In *Programming Language and Systems: Proceedings of the European Symposium on Programming*, volume 4960 of *LNCS*, pages 353–367. Springer, 2008.

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

[25] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy software. *ACM Trans. Prog. Lang. Syst.*, 27(3):477–526, 2005.

[26] P. Pratikakis, J. S. Foster., and M. Hicks. Locksmith: context-sensitive correlation analysis for race detection. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 320–331, New York, NY, USA, 2006. ACM.

[27] N. Swamy, M. Hicks, G. Morrisett, D. Grossman, and T. Jim. Safe manual memory management in Cyclone. *Sci. Comput. Program.*, 62(2):122–144, 2006.

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

[29] R. von Behren, J. Condit, and E. Brewer. Why events are a bad idea (for high-concurrency servers). In *Proceedings of the Conference on Hot Topics in Operating Systems*, page 4, Berkeley, CA, USA, 2003. USENIX Association.

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

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

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

Appendix

Language Syntax & Substitution Relation

Value	v	::=	$f \mid c \mid \mathtt{rgn}_\iota \mid \mathtt{loc}_l$
Expression	e	::=	$x \mid \mathsf{c} \mid f \mid (e\ e)^\xi \mid e[r] \mid \mathtt{new}\, e\, \mathtt{at}\, e$
		|	$e := e \mid \mathtt{deref}\, e \mid \mathtt{newrgn}^r\, \rho, x\, \mathtt{at}\, e\, \mathtt{in}\, e$
		|	$\mathtt{cap}^r_\eta\, e \mid \mathtt{rgn}_\iota \mid \mathtt{loc}_l \mid \mathtt{pop}_\gamma\ e$
Capability kind	ψ	::=	$\mathsf{rg} \mid \mathsf{lk}$
Capability op	η	::=	$\psi+ \mid \psi-$
Region	r	::=	$\rho \mid \iota \mid \iota@n$
Capability	κ	::=	$n, n \mid \overline{n, n}$
Region parent	π	::=	$r \mid \bot$
Effect	γ	::=	$\emptyset \mid \gamma, r^\kappa \triangleright \pi$
Type	τ	::=	$b \mid \langle\rangle \mid \tau \xrightarrow{\gamma\to\gamma} \tau \mid \forall\rho.\tau \mid \mathtt{ref}(\tau, r) \mid \mathtt{rgn}(r)$
Function	f	::=	$\lambda x.e\ \mathtt{as}\ \tau \xrightarrow{\gamma\to\gamma} \tau \mid \Lambda\rho.f$
Calling mode	ξ	::=	$\mathsf{seq} \mid \mathsf{par}$

$x_l[v/x]$	=	v		$x_1 \equiv x$
	|	x_1		*otherwise*
$r_l[r/r']$	=	r		$r_1 \equiv r'$
	|	r_1		*otherwise*
$\pi[r_l/r_2]$	=	$\bot \mid r[r_1/r_2]$		
$e[v/x]$	=	$x[v/x] \mid \mathsf{c} \mid \mathtt{rgn}_\iota \mid \mathtt{cap}^{r'}_\eta\ e_1[v/x] \mid \mathtt{pop}_\gamma\ e[v/x]$		
	|	$\mathtt{new}\, e_1[v/x]\, \mathtt{at}\, e_2[v/x] \mid \mathtt{deref}\, e_1[v/x] \mid e_1[v/x] := e_2[v/x]$		
	|	$\mathtt{loc}_l \mid f \mid (e_1[v/x]\ e_2[v/x])^\xi \mid (e_1[v/x])[r]$		
	|	$\mathtt{newrgn}^r\, \rho, y\, \mathtt{at}\, e_1[v/x]\, \mathtt{in}\, e_2[v/x]$		$y \not\equiv x$
$f[r/r_2]$	=	$\lambda x.e[r/r_2]\ \mathtt{as}\ \tau_1[r/r_2] \xrightarrow{\gamma_1[r/r_2]\to\gamma_2[r/r_2]} \tau_2[r/r_2] \mid \Lambda\rho'.f[r/r_2]$		$\rho' \not\equiv r_2$
$e[r/r_2]$	=	$x \mid \mathsf{c} \mid \mathtt{rgn}_\iota \mid \mathtt{cap}^{r'[r/r_2]}_\eta\ e_1[r/r_2] \mid (e_1[r/r_2]\ e_2[r/r_2])^{\xi[r/r_2]} \mid \mathtt{pop}_{\gamma[r/r_2]}\ e[r/r_2]$		
	|	$\mathtt{new}\, e_1[r/r_2]\, \mathtt{at}\, e_2[r/r_2] \mid \mathtt{deref}\, e_1[r/r_2] \mid e_1[r/r_2] := e_2[r/r_2]$		
	|	$\mathtt{loc}_l \mid f[r/r_2] \mid (e_1[r/r_2])[r_1[r/r_2]]$		
	|	$\mathtt{newrgn}^{r'[r/r_2]}\, \rho', x\, \mathtt{at}\, e_1[r/r_2]\, \mathtt{in}\, e_2[r/r_2]$		$\rho' \not\equiv r_2$
$\tau[r_l/r_2]$	=	$b \mid \langle\rangle \mid \mathtt{rgn}(r[r_1/r_2]) \mid \mathtt{ref}(\tau[r_1/r_2], r[r_1/r_2])$		
	|	$\tau_1[r_1/r_2] \xrightarrow{\gamma_1[r_1/r_2]\to\gamma_2[r_1/r_2]} \tau_2[r_1/r_2]$		
	|	$\forall\rho'.\tau[r_1/r_2]$		$\rho' \not\equiv r_2$
$\Gamma[r/\rho]$	=	$\emptyset \mid \Gamma_1[r/\rho], x : \tau[r/\rho]$		
$\xi[r_l/r_2]$	=	$\mathsf{seq} \mid \mathsf{par}$		
$\gamma[r_l/r_2]$	=	$\emptyset \mid \gamma'[r_1/r_2], r[r_1/r_2]^\kappa \triangleright \pi[r_1/r_2]$		

Operational Semantics

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

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

$$\frac{\sigma \simeq \sigma_1; \gamma, \iota^{\kappa} \triangleright \pi + \sigma_2 \quad \mathrm{lk}(\kappa) > 0 \vee (\pi = r \wedge \mathrm{is\_accessible}(\sigma, \bar{r}))}{\mathrm{is\_accessible}(\sigma, \iota)} \qquad \frac{\sigma \simeq \sigma_1; \gamma, \iota^{\kappa} \triangleright \pi \Rightarrow \mathrm{is\_pure}(\kappa) \wedge \mathrm{rg}(\kappa) = 0 \wedge \iota \notin \mathrm{dom}(\emptyset; \gamma) \wedge \sigma_1 \neq \emptyset \Rightarrow \mathrm{zero\_pure}(\sigma_1, \iota)}{\mathrm{zero\_pure}(\sigma, \iota)}$$

$$\frac{\vdash \delta \quad \sigma \vdash \delta}{\vdash \delta, n \mapsto \sigma} \qquad \frac{}{\vdash \emptyset}$$

$$\frac{\sigma \vdash \delta \quad \forall \iota \in \mathrm{dom}(\sigma).\mathrm{is\_accessible}(\sigma, \iota) \Rightarrow \neg \mathrm{is\_accessible}(\sigma', \iota)}{\sigma \vdash \delta, n \mapsto \sigma'} \qquad \frac{\forall \iota \in \mathrm{dom}(\sigma).\sigma \simeq \sigma_1; \gamma, \iota^{\kappa} \triangleright \pi + \sigma_2 \wedge \mathrm{rg}(\kappa) > 0 \wedge \mathrm{is\_pure}(\kappa) \Rightarrow \mathrm{zero\_pure}(\sigma_1, \iota) \wedge \iota \notin \mathrm{dom}(\sigma_2; \gamma)}{\sigma \vdash \emptyset}$$

$$\begin{array}{lcl}
\bar{r} & = & \begin{cases} \iota & \text{if } r = \iota \\ \bar{r'} & \text{if } r = r' @ n' \end{cases} \\
\sigma \simeq \sigma' & = & \begin{cases} \sigma_1 \simeq \sigma_2 \wedge \gamma_1 \simeq \gamma_2 & \text{if } \sigma = \sigma_1; \gamma_1 \wedge \sigma' = \sigma_2; \gamma_2 \\ \sigma_1 \equiv \sigma_2 & \text{otherwise} \end{cases} \\
\mathrm{rg}(\kappa) & = & n_1 \quad \text{if } \kappa = n_1, n_2 \vee \kappa = \overline{n_1, n_2} \\
\mathrm{lk}(\kappa) & = & n_2 \quad \text{if } \kappa = n_1, n_2 \vee \kappa = \overline{n_1, n_2} \\
\mathrm{live}(\gamma) & = & \{ r^{\kappa} \triangleright \pi \mid (r^{\kappa} \triangleright \pi) \in \gamma \wedge \mathrm{is_live}(\gamma, r) \} \\
\sigma_1 + \sigma_2 & = & \begin{cases} (\sigma_1 + \sigma); \gamma_n & \text{if } \sigma_2 \equiv \sigma; \gamma_n \\ \sigma_1 & \text{if } \sigma_2 \equiv \emptyset \end{cases} \\
\gamma_1 + \gamma_2 & = & \begin{cases} (\gamma_1 + \gamma), r^{\kappa} \triangleright \pi & \text{if } \gamma_2 \equiv \gamma, r^{\kappa} \triangleright \pi \\ \gamma_1 & \text{if } \gamma_2 \equiv \emptyset \end{cases} \\
\mathrm{dom}(\gamma) & = & \{ r \mid (r^{\kappa} \triangleright \pi) \in \gamma \} \\
\mathrm{dom}(\sigma) & = & \begin{cases} \{\bar{r} \mid (r^{\kappa} \triangleright \pi) \in \gamma\} \cup \mathrm{dom}(\sigma') & \text{if } \sigma = \sigma'; \gamma \\ \emptyset & \text{if } \sigma = \emptyset \end{cases} \\
\mathrm{pops}(\sigma : e) & = & \begin{cases}
\mathrm{pops}(\sigma : e_1) \wedge \mathrm{pops}(\emptyset; \emptyset : e_2) & \text{if } e \equiv (e_1\ e_2)^{\xi} \wedge e_1 \neq v \\
\mathrm{pops}(\sigma : e_2) \wedge \mathrm{pops}(\emptyset; \emptyset : v) & \text{if } e \equiv (v\ e_2)^{\xi} \\
\mathrm{pops}(\sigma : e_1) & \text{if } e \equiv (e_1)\,[v] \\
\mathrm{pops}(\sigma : e_1) \wedge \mathrm{pops}(\emptyset; \emptyset : e_2) & \text{if } e \equiv \mathtt{newrgn}^{r}\, \rho, x\, \mathtt{at}\, e_1\, \mathtt{in}\, e_2 \\
\mathrm{pops}(\sigma : e_1) & \text{if } e \equiv \mathtt{cap}^{r'}_{\eta}\ e_1 \\
\mathrm{pops}(\sigma : e_1) \wedge \mathrm{pops}(\emptyset; \emptyset : v) & \text{if } e \equiv \mathtt{new}\ v\ \mathtt{at}\ e_1 \\
\mathrm{pops}(\sigma : e_1) \wedge \mathrm{pops}(\emptyset; \emptyset : e_2) & \text{if } e \equiv \mathtt{new}\ e_1\ \mathtt{at}\ e_2 \wedge e_1 \neq v \\
\mathrm{pops}(\sigma : e_1) & \text{if } e \equiv \mathtt{deref}\ e_1 \\
\mathrm{pops}(\sigma : e_1) \wedge \mathrm{pops}(\emptyset; \emptyset : e_2) & \text{if } e \equiv e_1 := e_2 \wedge e_1 \neq v \\
\mathrm{pops}(\sigma : e_1) \wedge \mathrm{pops}(\emptyset; \emptyset : v) & \text{if } e \equiv v := e_1 \\
\mathrm{pops}(\sigma' : e_1) & \text{if } e \equiv \mathtt{pop}_{\gamma_r}\ e_1 \wedge \sigma = \emptyset; \gamma_r + \sigma' \\
\sigma \equiv \emptyset; \gamma \wedge \mathrm{pops}(\emptyset; \emptyset : e') & \text{if } e \equiv \lambda x.\, e'\ \mathtt{as}\ \tau \\
\sigma \equiv \emptyset; \gamma \wedge \mathrm{pops}(\emptyset; \emptyset : f) & \text{if } e \equiv \Lambda \rho.\, f \\
\sigma \equiv \emptyset; \gamma & \text{if } e \in \{\mathtt{loc}_{\ell}, \mathtt{rgn}_{\iota}, ()\}
\end{cases}
\end{array}$$

Stack	σ	::=	$\emptyset \mid \sigma; \gamma$
Hierarchy	δ	::=	$\emptyset \mid \delta, n \mapsto \sigma$
Contents	H	::=	$\emptyset \mid H, \ell \mapsto v$
Region list	S	::=	$\emptyset \mid S, \iota \mapsto H$
Threads	T	::=	$\emptyset \mid T, n : e$
Configuration	C	::=	$\delta; S; T$

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

$$\frac{\delta = \delta'', n \mapsto \sigma \quad \sigma; S; e \to \sigma'; S'; e' \quad \delta' = \delta'', n \mapsto \sigma' \vdash \delta'}{\delta; S; T, n : E[e] \rightsquigarrow \delta'; S'; T, n : E[e']}\ (E\text{-}S) \qquad \frac{e' \equiv (v_1\ v)^{\mathsf{par}} \quad v_1 \equiv \lambda x.\, e\ \mathtt{as}\ \tau_1 \xrightarrow{\gamma_1 \to \gamma_2} \tau_2 \quad e'' \equiv (v_1\ v)^{\mathsf{seq}} \quad \delta = \delta'', n \mapsto \sigma; \gamma \quad \text{fresh } n' \quad \mathsf{par} \vdash \gamma' = \emptyset \oplus (\gamma \ominus \gamma_1) \quad \delta' = \delta'', n \mapsto \sigma; \gamma', n' \mapsto \emptyset; \gamma_1}{\delta; S; T, n : E[e'] \rightsquigarrow \delta'; S; T, n : E[()], n' : e''}\ (E\text{-}S$$

$$\frac{\sigma = \sigma'; \gamma \quad \text{is\_live}(\gamma, r) \quad \gamma = \gamma', r^{\kappa} \triangleright \pi \quad \kappa' = [\![\eta]\!](\kappa) \quad \sigma'' = \sigma'; \text{live}(\gamma', r^{\kappa'} \triangleright \pi)}{\sigma; S; \mathtt{cap}^{r}_{\eta}\ \mathtt{rgn}_{\bar{r}} \to \sigma''; S; ()}\ (E\text{-}C)$$

$$\frac{\delta = \delta', n \mapsto (\emptyset; \emptyset)}{\delta; S; T, n : () \rightsquigarrow \delta'; S; T}\ (E\text{-}T) \qquad \frac{\sigma = \sigma'; \gamma \quad \mathsf{seq} \vdash \gamma = \gamma_1 \oplus \gamma_r \quad \sigma'' = \sigma'; \gamma_r; \gamma_1}{\sigma; S; ((\lambda x.\, e\ \mathtt{as}\ \tau_1 \xrightarrow{\gamma_1 \to \gamma_2} \tau_2)\ v)^{\mathsf{seq}} \to \sigma''; S; \mathsf{pop}_{\gamma_r}\ e[v/x]}\ (E\text{-}A)$$

$$\frac{\text{fresh } n'}{\sigma; S; (\Lambda\rho.\, f)[r] \to \sigma; S; f[\bar{r}@n'/\rho]}\ (E\text{-}RP) \qquad \frac{\sigma = \sigma'; \gamma \quad \text{is\_accessible}(\gamma, r) \quad (\ell \mapsto v) \in S(\bar{r})}{\sigma; S; \mathtt{deref}\ \mathtt{loc}_{\ell} \to \sigma; S; v}\ (E\text{-}D)$$

$$\frac{\sigma = \sigma'; \gamma \quad \text{is\_live}(\gamma, r) \quad \text{fresh } \ell}{\sigma; S; \mathtt{new}\ v\ \mathtt{at}\ \mathtt{rgn}_{\bar{r}} \to \sigma; S[\bar{r} \mapsto S(\bar{r}), \ell \mapsto v]; \mathtt{loc}_{\ell}}\ (E\text{-}NR) \qquad \frac{\sigma = \sigma'; \gamma \quad \text{is\_accessible}(\gamma, r) \quad (\ell \mapsto v_1) \in S(\bar{r})}{\sigma; S; \mathtt{loc}_{\ell} := v \to \sigma; S(\bar{r})[\ell \mapsto v]; ()}\ (E\text{-}AS)$$

$$\frac{\sigma = \sigma'; \gamma \quad \text{is\_live}(\gamma, r) \quad \text{fresh } \iota \quad \sigma'' = \sigma'; \gamma, \iota^{1,1} \triangleright r}{\sigma; S; \mathtt{newrgn}^{r}\ \rho, x\ \mathtt{at}\ \mathtt{rgn}_{\bar{r}}\ \mathtt{in}\ e \to \sigma''; S, \iota \mapsto \emptyset; e[\iota/\rho][\mathtt{rgn}_{\iota}/x]}\ (E\text{-}NG) \qquad \frac{\sigma = \sigma'; \gamma_r; \gamma' \quad \mathsf{seq} \vdash \gamma'' = \gamma' \oplus (\gamma_r \ominus \emptyset) \quad \sigma'' = \sigma'; \gamma''}{\sigma; S; \mathsf{pop}_{\gamma_r}\ v \to \sigma''; S; v}\ (E\text{-}E)$$

Static Semantics

$$\begin{array}{lcl}
\text{is_pure}(\kappa) & = & \exists n_1.\, \exists n_2.\, \kappa = n_1, n_2 \\
[\![\eta]\!](\kappa) & = & \kappa' \quad \text{if } \eta \equiv \psi \pm \wedge \text{is_pure}(\kappa) \Leftrightarrow \text{is_pure}(\kappa') \wedge \\
 & & (\psi = \mathsf{rg} \Rightarrow \text{rg}(\kappa') = \text{rg} \pm 1 \wedge \text{lk}(\kappa') = \text{lk}(\kappa)) \wedge \\
 & & (\psi = \mathsf{lk} \Rightarrow \text{lk}(\kappa') = \text{lk} \pm 1 \wedge \text{rg}(\kappa') = \text{rg}(\kappa)) \\
\text{valid_pure}(\gamma) & = & \forall r^{\kappa} \triangleright \pi \in \gamma. \exists \gamma_1. \gamma = \gamma_1, r^{\kappa} \triangleright \pi \wedge \text{is_pure}(\kappa) \Rightarrow \forall r' \simeq r. r' \notin \text{dom}(\gamma_1) \\
\text{ok}(\gamma_1; \gamma_2) & = & \text{valid_pure}(\gamma_1) \wedge \text{valid_pure}(\gamma_2) \\
\text{valid}(\gamma_1; \gamma_2) & = & (\forall (r^{\kappa} \triangleright \pi) \in \gamma_1.\, \forall (r^{\kappa'} \triangleright \pi') \in \gamma_2.\, \pi = \pi' \wedge (\text{is_pure}(\kappa) \Leftrightarrow \\
 & & \text{is_pure}(\kappa'))) \wedge \text{live}(\gamma_1) = \gamma_1 \wedge \text{live}(\gamma_2) = \gamma_2 \wedge \text{dom}(\gamma_2) \subseteq \text{dom}(\gamma_1) \\
r_1 \simeq r_2 & \equiv & \begin{cases} \iota_1 \equiv \iota_2 & \text{if } r_1 \equiv \iota_1@n_1 \wedge r_2 \equiv \iota_2@n_2 \\ r_1 \equiv r_2 & \text{otherwise} \end{cases} \\
\pi_1 \simeq \pi_2 & \equiv & \begin{cases} r_1 \simeq r_2 & \text{if } \pi_1 \equiv r_1 \wedge \pi_1 \equiv r_2 \\ \pi_1 \equiv \pi_2 & \text{otherwise} \end{cases} \\
\gamma_1 \simeq \gamma_2 & \equiv & \begin{cases} \gamma_3 \simeq \gamma_4 \wedge r_1 \simeq r_2 \wedge \pi_1 \simeq \pi_2 & \text{if } \gamma_1 = \gamma_3, r_1^{\kappa} \triangleright \pi_1 \wedge \gamma_2 = \gamma_4, r_2^{\kappa} \triangleright \pi_2 \\ \gamma_1 \equiv \gamma_2 & \text{otherwise} \end{cases} \\
\tau_1 \simeq \tau_2 & \equiv & \begin{cases} \tau_1 \equiv \tau_2 & \text{if } (\{\tau_1\} \cup \{\tau_2\}) \cap \{\langle\rangle, b\} \\ \tau_3 \simeq \tau_4 \wedge r_1 \simeq r_2 & \text{if } \tau_1 = \mathtt{ref}(\tau_3, r_1) \wedge \tau_2 = \mathtt{ref}(\tau_4, r_2) \\ r_1 \simeq r_2 & \text{if } \tau_1 = \mathtt{rgn}(r_1) \wedge \tau_2 = \mathtt{rgn}(r_2) \\ \tau_3 \simeq \tau_5 \wedge \tau_4 \simeq \tau_6 \wedge \gamma_1 \simeq \gamma_3 \wedge \gamma_2 \simeq \gamma_4 & \text{if } \tau_1 \equiv \tau_3 \xrightarrow{\gamma_1 \to \gamma_2} \tau_4 \wedge \tau_2 \equiv \tau_5 \xrightarrow{\gamma_3 \to \gamma_4} \tau_6 \\ \tau_3[\rho/\rho_1] \simeq \tau_4[\rho/\rho_2] & \text{if } \tau_1 \equiv \forall \rho_1.\, \tau_3 \wedge \tau_2 \equiv \forall \rho_2.\, \tau_4 \wedge \text{fresh } \rho \end{cases} \\
\text{set}(\gamma; \gamma') & = & (\forall (r^{\kappa} \triangleright \pi) \in \gamma. \exists \gamma_1. \gamma = \gamma_1, r^{\kappa} \triangleright \pi \wedge r \notin \text{dom}(\gamma_1)) \wedge (\forall (r^{\kappa} \triangleright \pi) \in \gamma'. \exists \gamma_1. \gamma' = \gamma_1, r^{\kappa} \triangleright \pi \wedge r \notin \text{dom}(\gamma_1))
\end{array}$$

Region List $R ::= \emptyset \mid R, \iota$

Type variable list $\Delta ::= \emptyset \mid \Delta, \rho$

Memory List $M ::= \emptyset \mid M, \ell \mapsto (\tau, \iota)$

Variable list $\Gamma ::= \emptyset \mid \Gamma, x : \tau$

Constraint Well-formedness

$$\frac{}{R; \Delta \vdash \emptyset} \qquad \frac{R; \Delta \vdash \gamma_1 \quad R; \Delta \vdash r_1 \quad r_1 \neq \pi \quad \pi = r_2 \Rightarrow R; \Delta \vdash r_2}{R; \Delta \vdash \gamma_1, r_1^{\kappa} \triangleright \pi}$$

Region Well-formedness

$$\frac{r \in \Delta \uplus R}{R; \Delta \vdash r} \qquad \frac{R; \Delta \vdash \iota}{R; \Delta \vdash \iota@n}$$

Type Well-formedness

$$\frac{}{R; \Delta \vdash b} \qquad \frac{R; \Delta \vdash r}{R; \Delta \vdash \mathtt{rgn}(r)} \qquad \frac{R; \Delta, \rho \vdash \tau}{R; \Delta \vdash \forall \rho.\, \tau} \qquad \frac{R; \Delta \vdash \tau \quad R; \Delta \vdash r}{R; \Delta \vdash \mathtt{ref}(\tau, r)}$$

$$\frac{\text{valid}(\gamma_1; \gamma_2) \quad R; \Delta \vdash \tau_1 \quad R; \Delta \vdash \gamma_1 \quad R; \Delta \vdash \tau_2 \quad R; \Delta \vdash \gamma_2}{R; \Delta \vdash \tau_1 \xrightarrow{\gamma_1 \to \gamma_2} \tau_2} \qquad \frac{}{R; \Delta \vdash \langle\rangle}$$

Variable Context Well-formedness

$$\frac{R;\Delta \vdash \tau_1 \qquad x \notin dom(\Gamma_1) \qquad R;\Delta \vdash \Gamma_1}{R;\Delta \vdash \Gamma_1, x:\tau_1}$$

Memory Location Well-formedness

$$\frac{R \vdash M_1 \qquad \ell \notin dom(M_1) \qquad R;\emptyset \vdash \mathtt{ref}(\tau_1,\iota)}{R \vdash M_1, \ell \mapsto (\tau_1,\iota)}$$

Program Typing Context Well-formedness

$$\frac{\mathrm{set}(\gamma;\gamma') \qquad \mathrm{ok}(\gamma;\gamma') \qquad R \vdash M \qquad R;\Delta \vdash \Gamma \qquad R;\Delta \vdash \gamma \qquad R;\Delta \vdash \gamma'}{\vdash R;M;\Delta;\Gamma;\gamma;\gamma'}$$

$$\frac{\vdash R;M;\Delta;\Gamma;\gamma;\gamma \qquad (x:\tau)\in\Gamma \qquad \tau \simeq \tau'}{R;M;\Delta;\Gamma \vdash x : \tau' \,\&\, (\gamma;\gamma)}\ (T\text{-}V) \qquad \frac{\vdash R;M;\Delta;\Gamma;\gamma;\gamma}{R;M;\Delta;\Gamma \vdash \mathtt{c} : b \,\&\, (\gamma;\gamma)}\ (T\text{-}I) \qquad \frac{\vdash R;M;\Delta;\Gamma;\gamma;\gamma}{R;M;\Delta;\Gamma \vdash () : \langle\rangle \,\&\, (\gamma;\gamma)}\ (T\text{-}U)$$

$$\frac{\vdash R;M;\Delta;\Gamma;\gamma;\gamma \qquad R;\Delta \vdash \iota \qquad r \simeq \iota}{R;M;\Delta;\Gamma \vdash \mathtt{rgn}_\iota : \mathtt{rgn}(r) \,\&\, (\gamma;\gamma)}\ (T\text{-}R) \qquad \frac{\vdash R;M;\Delta;\Gamma;\gamma;\gamma \qquad (\ell \mapsto (\tau,\iota)) \in M \qquad \tau' \simeq \mathtt{ref}(\tau,\iota)}{R;M;\Delta;\Gamma \vdash \mathtt{loc}_\ell : \tau' \,\&\, (\gamma;\gamma)}\ (T\text{-}L)$$

$$\frac{\mathrm{is\_live}(\gamma', r^\kappa \triangleright \pi, r) \qquad \gamma'' = \mathrm{live}(\gamma', r^{\kappa'} \triangleright \pi) \qquad R;M;\Delta;\Gamma \vdash e_1 : \mathtt{rgn}(r) \,\&\, (\gamma;\gamma', r^\kappa \triangleright \pi) \qquad \kappa' = [\![\eta]\!](\kappa)}{R;M;\Delta;\Gamma \vdash \mathtt{cap}^r_\eta\ e_1 : \langle\rangle \,\&\, (\gamma;\gamma'')}\ (T\text{-}$$

$$\frac{\vdash R;M;\Delta;\Gamma;\gamma;\gamma \qquad \tau \equiv \tau_1 \xrightarrow{\gamma_1 \to \gamma_2} \tau_2 \qquad \mathrm{set}(\gamma_1;\gamma_2) \qquad \tau \simeq \tau' \qquad \mathrm{ok}(\gamma_1;\gamma_2) \Rightarrow R;M;\Delta;\Gamma, x:\tau_1 \vdash e : \tau_2 \,\&\, (\gamma_1;\gamma_2)}{R;M;\Delta;\Gamma \vdash \lambda x.\, e\ \mathtt{as}\ \tau : \tau' \,\&\, (\gamma;\gamma)}\ (T\text{-}F) \qquad \frac{R;M;\Delta,\rho;\Gamma \vdash f : \tau \,\&\, (\gamma;\gamma)}{R;M;\Delta;\Gamma \vdash \Lambda\rho.\, f : \forall\rho.\,\tau \,\&\, (\gamma;\gamma)}\ (T\text{-}RF)$$

$$\frac{R;\Delta \vdash r \qquad R;M;\Delta;\Gamma \vdash e : \forall\rho.\,\tau \,\&\, (\gamma;\gamma')}{R;M;\Delta;\Gamma \vdash e\ [r] : \tau[r/\rho] \,\&\, (\gamma;\gamma')}\ (T\text{-}$$

$$\frac{R;M;\Delta;\Gamma \vdash e_1 : \tau_1 \xrightarrow{\gamma_1 \to \gamma_2} \tau_2 \,\&\, (\gamma;\gamma_3) \qquad \xi = \mathsf{par} \Rightarrow \tau_2 = \langle\rangle \qquad R;M;\Delta;\Gamma \vdash e_2 : \tau_1 \,\&\, (\gamma_3;\gamma_4) \qquad \xi \vdash \gamma_5 = \gamma_2 \oplus (\gamma_4 \ominus \gamma_1)}{R;M;\Delta;\Gamma \vdash (e_1\ \ e_2)^\xi : \tau_2 \,\&\, (\gamma;\gamma_5)}\ (T\text{-}AP)$$

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

$$\frac{R;M;\Delta;\Gamma \vdash e_1 : \tau \,\&\, (\gamma;\gamma') \qquad \mathrm{is\_live}(\gamma'', r) \qquad R;M;\Delta;\Gamma \vdash e_2 : \mathtt{rgn}(r) \,\&\, (\gamma';\gamma'')}{R;M;\Delta;\Gamma \vdash \mathtt{new}\ e_1\ \mathtt{at}\ e_2 : \mathtt{ref}(\tau,r) \,\&\, (\gamma;\gamma'')}\ (T\text{-}NR) \qquad \frac{R;M;\Delta;\Gamma \vdash e_1 : \mathtt{ref}(\tau,r) \,\&\, (\gamma;\gamma') \qquad R;M;\Delta;\Gamma \vdash e_2 : \tau \,\&\, (\gamma';\gamma'') \qquad \mathrm{is\_accessible}(\gamma'', r)}{R;M;\Delta;\Gamma \vdash e_1 := e_2 : \langle\rangle \,\&\, (\gamma;\gamma'')}\ (T\text{-}A)$$

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

$$\frac{\mathsf{seq} \vdash \gamma' = \gamma_2 \oplus (\gamma_r \ominus \emptyset) \qquad R;\Delta \vdash \gamma_r \qquad \mathrm{ok}(\gamma_r;\emptyset) \qquad R;M;\Delta;\Gamma \vdash e : \tau' \,\&\, (\gamma_1;\gamma_2) \qquad \tau \simeq \tau' \qquad \mathrm{set}(\gamma_r;\emptyset)}{R;M;\Delta;\Gamma \vdash \mathtt{pop}_{\gamma_r}\ e : \tau \,\&\, (\gamma_1;\gamma')}\ (T\text{-}E)$$

$$\frac{}{\xi \vdash \gamma = \emptyset \oplus \gamma}\ (ES\text{-}N) \qquad \frac{\xi \vdash \gamma, r^{\kappa_2} \triangleright \pi = \gamma_1 \oplus \gamma_2 \qquad \xi \vdash \kappa = \kappa_1 + \kappa_2 \qquad \pi \simeq \pi' \qquad r' \simeq r}{\xi \vdash \gamma, r^\kappa \triangleright \pi = \gamma_1, r'^{\kappa_1} \triangleright \pi' \oplus \gamma_2}\ (ES\text{-}C)$$

$$\frac{\xi \vdash \gamma = \gamma_1 \oplus \gamma_r \qquad \xi \vdash \gamma' = \gamma_2 \oplus \gamma_r \qquad \gamma'' = \mathrm{live}(\gamma') \qquad \mathrm{ok}(\gamma_1;\gamma_2) \qquad \xi = \mathsf{par} \Rightarrow \gamma_2 = \emptyset}{\xi \vdash \gamma'' = \gamma_2 \oplus (\gamma \ominus \gamma_1)}\ (ESJ)$$

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

Type Safety Judgements

$$\begin{aligned}\mathrm{nolock}(\delta,n,e) \quad &\equiv e = E[\mathtt{cap}^r_{+\mathsf{lk}}\ \mathtt{rgn}_\mathsf{j}] \wedge \exists\delta'',\pi,\kappa,\kappa'.\ \delta = \delta'', n \mapsto \sigma;\gamma, r^\kappa \triangleright \pi \wedge \kappa' = [\![\mathsf{lk}+]\!](\kappa) \wedge \neg \vdash \delta'', n \mapsto \sigma;\gamma, r^{\kappa'} \triangleright \pi \\ \mathrm{redex}(e) \quad &= (\exists\,\sigma,\sigma',S,S',e',n.\ \ \sigma;S;e \to \sigma';S';e') \vee (\exists v_1,v_2,\gamma_1.e = (v_1\ \ v_2)^{\mathsf{par}})\end{aligned}$$

$$\frac{\bigcup_{(n\mapsto H)\in S}\{\ell \mid (\ell \mapsto v) \in H\} = \{\ell \mid (\ell \mapsto (\tau,j)) \in M\}}{M \vdash S} \qquad \frac{R = \{\iota \mid (\iota \mapsto H) \in S\}}{R \vdash S} \qquad \frac{\forall(n \mapsto \sigma) \in \delta.\forall\gamma \in \sigma.\forall(r^\kappa \triangleright \pi) \in \gamma.\ \bar{r} \in R \wedge (\pi = r' \Rightarrow \bar{r'} \in R)}{R \vdash \delta}$$

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

Store Typing

$$\frac{\vdash \delta \qquad R;M \vdash S \qquad R \vdash \delta}{R;M \vdash \delta;S}$$

Not Stuck

$$\frac{\forall(n:e) \in T.\ (\delta;S;T \rightsquigarrow \delta;S';T' \wedge (n:e) \notin T') \vee \mathrm{nolock}(\delta,n,e)}{\vdash \delta;S;T}$$

Thread Typing

$$\frac{}{R;M;\emptyset \vdash \emptyset} \qquad \frac{R;M;\emptyset;\emptyset \vdash e : \langle\rangle \,\&\, (\gamma;\emptyset) \qquad R;M;\delta' \vdash T \qquad \forall(n':e') \in T.n' \neq n \qquad \delta = \delta', n \mapsto \sigma;\gamma \qquad \mathrm{pops}(\sigma;\gamma:e)}{R;M;\delta \vdash T, n:e}$$

Configuration Typing

$$\frac{R;M;\delta \vdash T \qquad R;M \vdash \delta;S}{R;M \vdash \delta;S;T}$$

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

Type Safety Proof

Definition Type Safety Initial Environment

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

Theorem 1 (Type safety) *Let R_0, δ_0, S_0 and T_0 be defined as in Definition . If the operational semantics takes any number of steps $\delta_0; S_0; T_0 \rightsquigarrow^n \delta_n; S_n; T_n$, then the resulting configuration $\delta_n; S_n; T_n$ is not stuck.*

Proof. The proof is trivial: Lemma 1 is applied to the assumptions $\delta; S; T$ is well-typed and the operational semantics performs n steps, to obtain that $\delta_n; S_n; T_n$ is well-typed for some $R_n; M_n$. Then, lemma 53 is applied to the latter fact to prove that $\delta_n; S_n; T_n$ is not stuck.

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

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

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

Proof. By case analysis on the thread evaluation relation:

Case *E-T*: The premise of this rule are $T_1, n : () = T$ and $\delta_1, n \mapsto (\emptyset; \emptyset) = \delta$, for some δ_1 and T_1. By applying lemma 3 to the configuration typing assumption we have that $R; M \vdash \delta_1, n \mapsto (\emptyset; \emptyset); S; T_1, n : ()$ holds.

By inversion of the latter configuration typing derivation we obtain the store $(R; M \vdash \delta_1, n \mapsto (\emptyset; \emptyset); S)$, and thread $(R; M; \delta_1, n \mapsto (\emptyset; \emptyset) \vdash T_1, n : ())$ typing derivations. By inversion of the thread typing derivation, we have that $R; M; \delta_1 \vdash T_1$ is well-typed. Lemma 9 is applied to the store typing derivation $(R; M \vdash \delta_1, n \mapsto (\emptyset; \emptyset); S)$ to obtain that $R; M \vdash \delta_1; S$ holds. The new store and thread typing derivations give us $R; M \vdash \delta_1; S; T_1$.

Case *E-S* : The premises of this rule are $T_1, n : E[e] = T$, $\delta = \delta_1, n \mapsto \sigma$, $\vdash \delta'$, $\delta' = \delta_1, n \mapsto \sigma'$ and $\sigma; S; e \rightarrow \sigma'; S'; e'$. The resulting configuration is $\delta'; S'; T_1, n : E[e']$. By applying lemma 3 to the latter configuration typing derivation, we have that $R; M \vdash \delta; S; T_1, n : E[e]$ holds. By inverting the *configuration typing* we obtain that $R; M; \delta \vdash T_1, n : E[e]$ and $R; M \vdash \delta; S$ holds. By inversion of the thread typing derivation we have that $R; M; \emptyset; \emptyset \vdash E[e] : \langle\rangle\&(\gamma; \emptyset)$, $\sigma = \sigma''; \gamma$, $\text{pops}(\sigma''; \gamma : E[e])$ holds, and $R; M; \delta_1 \vdash T_1$ holds. By applying lemma 11 to the typing derivation of $E[e]$ we obtain that $R; M; \emptyset; \emptyset \vdash e : \tau, (\gamma; \gamma')$ for some γ' and τ. By applying lemma 29 to $\vdash \delta[n \mapsto \sigma']$ (rule *E-S*), the typing derivation of e, the *expression evaluation* step $(\sigma; S; e \rightarrow \sigma'; S'; e')$ and the store typing derivation $(R; M \vdash \delta; S)$, we obtain that e' is also well-typed $(R'; M'; \emptyset; \emptyset \vdash e' : \tau\&(\gamma''; \gamma'))$, where γ'' is the top stack frame for thread n $(\sigma' = \sigma'''; \gamma''$ for some $\sigma''')$, for some $R \subseteq R'$, $M \subseteq M'$, and the resulting store $\delta[n \mapsto \sigma']; S'$ is also well-typed $(R'; M' \vdash \delta[n \mapsto \sigma']; S')$. By applying lemma 12 to the typing derivation of e' we have that $\vdash R'; M'; \emptyset; \emptyset; \gamma''; \gamma'$. By inversion of the latter derivation we have that $R' \vdash M'$. By applying lemma 4 to $R; M; \delta \vdash T_1, n : E[e]$, $R \subseteq R'$, $M \subseteq M'$ and $R' \vdash M'$, we have that $R'; M'; \delta \vdash T_1, n : E[e]$ holds. By inversion of the latter derivation we have that $R'; M'; \emptyset; \emptyset \vdash E[e] : \langle\rangle\&(\gamma; \emptyset)$. By lemma 23 we can substitute e' for e in the evaluation context E (all well-typed in $R'; M'$) to obtain $R'; M'; \emptyset; \emptyset \vdash E[e'] : \langle\rangle\&(\gamma''; \emptyset)$.

The application of lemma 6 to $\text{pops}((\sigma''; \gamma) : E[e])$ implies that $\exists \sigma_1, \sigma_2 . \sigma''; \gamma = \sigma_1 + \sigma_2 \wedge \text{pops}(\sigma_2 : e))$. The application of lemma 7 to $\text{pops}((\sigma''; \gamma) : E[e])$, $(\sigma''; \gamma); S; e \rightarrow \sigma'; S'; e'$, $\sigma''; \gamma = \sigma_1 + \sigma_2$, and $\text{pops}(\sigma_2 : e)$, gives us that $\text{pops}(\sigma_1 + \sigma_2' : E[e'])$, where $\sigma' = \sigma_1 + \sigma_2'$.

By inversion of $R'; M'; \delta \vdash T_1, n : E[e]$ we have that $R'; M'; \delta_1 \vdash T_1$ and $\forall (n' : e') \in T_1 . n' \neq n$. We can reconstruct a similar derivation by using the latter derivations along with $\text{pops}(\sigma_1 + \sigma_2' : E[e'])$, $\delta' = \delta_1, n \mapsto \sigma_1 + \sigma_2'$ and $R'; M'; \emptyset; \emptyset \vdash E[e'] : \langle\rangle\&(\gamma''; \emptyset)$: $R'; M'; \delta' \vdash T_1, n : E[e']$. Both $R'; M' \vdash \delta'; S'$ and $R'; M'; \delta' \vdash T_1, n : E[e']$ imply that $R'; M' \vdash \delta'; S'; T_1, n : E[e']$ holds.

Case *E-SN*: The *program evaluation* assumption gives us that $e \equiv E[e']$, such that e' is a parallel application redex, and its premise asserts that e' is moved to a new thread as a local application redex e''. It also gives that $T_1, n : E[e'] = T$ and that n' is *fresh*. The resulting store map δ' is equal to $\delta'', n \mapsto \sigma; \gamma', n' \mapsto \gamma_1$, where δ equals $\delta'', n \mapsto \sigma; \gamma$ and $\mathsf{par} \vdash \gamma' = \emptyset \oplus (\gamma \ominus \gamma_1)$ holds. By applying lemma 3 to the configuration typing derivation, $R; M \vdash \delta; S; T$, $T_1, n : E[e'] = T$ and $\delta = \delta'', n \mapsto \sigma; \gamma$, we have that $R; M \vdash \delta'', n \mapsto \sigma; \gamma; S; T_1, n : E[e']$ holds.

We need to prove that $R; M \vdash \delta'; S; T_1, n : E[()], n' : e''$ holds. It suffices to prove that $R; M; \delta' \vdash T_1, n : E[()], n' : e''$ and $R; M \vdash \delta'; S$.

Thread typing: The following obligations must be proved:

- $R; M; \emptyset; \emptyset \vdash E[()] : \langle\rangle \,\&\, (\gamma'; \emptyset)$: we must prove that that $R; M; \delta'', n \mapsto \sigma; \gamma \vdash T_1, n : E[e']$. By inversion of this obligation it suffices to prove that $R; M; \emptyset; \emptyset \vdash E[e'] : \langle\rangle \& (\gamma; \emptyset)$ holds. By applying lemma 11 to the typing derivation of $E[e']$, we obtain that e' is well-typed in the context $R; M; \emptyset; \emptyset$ with effect $(\gamma; \gamma'')$ for some γ''. The application of lemma 10 to the latter derivation implies that $\mathsf{par} \vdash \gamma'' = \emptyset \oplus (\gamma \ominus \gamma_1')$, where $\gamma_1' \simeq \gamma_1$. The application of lemma 33 implies that $\mathsf{par} \vdash \gamma'' = \emptyset \oplus (\gamma \ominus \gamma_1)$ holds. The capability addition derivation rule is deterministic thus, that γ'' is equal to γ'. Thus, e' is well-typed with effect $(\gamma; \gamma')$.

  By applying lemma 12 to the typing derivation of e', we have that $\vdash R; M; \emptyset; \emptyset; \gamma; \gamma'$. Thus, $\vdash R; M; \emptyset; \emptyset; \gamma'; \gamma'$ also holds (trivial). Consequently, by rule *T-U* we have that $R; M; \emptyset; \emptyset \vdash () : \langle\rangle \& (\gamma'; \gamma')$ holds. Now we can substitute the well-typed unit value described above for e' in the evaluation context E, by using lemma 23, to obtain that $E[()]$ is well-typed in the typing context $R; M; \emptyset; \emptyset$ with effect $(\gamma'; \emptyset)$.
- $\forall (n'' : e''') \in T_1 . n'' \neq n$: immediate from the thread typing assumption (premise), which can be obtained by inversion of the original configuration typing derivation.
- $\delta = \delta'', n \mapsto \sigma; \gamma'$: immediate.
- $\mathrm{pops}(\sigma; \gamma' : E[()])$: this is immediate by the application of lemma 8 to the fact that $\mathrm{pops}(\sigma; \gamma : E[e'])$.
- $R; M; \emptyset; \emptyset \vdash e'' : \langle\rangle \,\&\, (\gamma_1; \emptyset)$: the application of lemma 24 to the fact that e' is well-typed in the context $R; M; \emptyset; \emptyset$ with effect $(\gamma; \gamma')$, yields that e'' is well-typed in the context $R; M; \emptyset; \emptyset$ with effect $(\gamma_1; \emptyset)$.
- $\forall (n'' : e''') \in T_1 . n'' \neq n'$ and $n' \neq n$: immediate from the fact that n' is *fresh*.
- $\delta = \delta'', n \mapsto \sigma; \gamma', n' \mapsto \emptyset; \gamma_1$: immediate.
- $\mathrm{pops}(\emptyset; \gamma_1 : e'')$: the asssumption that $\mathrm{pops}(\sigma; \gamma : E[e'])$ and lemma 7 imply that $\sigma; \gamma = \sigma_a + \sigma_b$ and $\mathrm{pops}(\sigma_b : e')$. Expression e' comprises of values thus by the definition of *pops* we have that $\sigma_b = \emptyset; \gamma$. The definition of *pops* also allows us to derive $\mathrm{pops}(\emptyset; \gamma_1 : e')$. Thus, $\mathrm{pops}(\emptyset; \gamma_1 : e'')$ also holds.

Store typing: by applying lemma 12 to the typing derivation (as shown earlier) of e' implies that $\mathrm{ok}(\gamma; \gamma')$ holds. The proof is immediate by the application of lemma 28 to the fact that $\mathrm{ok}(\gamma; \gamma')$ holds, $R; M \vdash \delta; S$ holds, δ' is equal to $\delta'', n \mapsto \sigma; \gamma', n' \mapsto \gamma_1$, $\mathrm{live}(\gamma_1) = \gamma_1$ (by inversion of the function type well-formedness derivation, which is a premise of the function typing derivation) and $\mathsf{par} \vdash \gamma' = \emptyset \oplus (\gamma \ominus \gamma_1)$.

Lemma 3 (Reordering) $R; M \vdash \delta; S; T \wedge T' = T \wedge \delta' = \delta \Rightarrow R; M \vdash \delta'; S; T'$

Proof. Trivial.

Lemma 4 (Thread Weakening) $R; M; \delta \vdash T \wedge R \subseteq R' \wedge M \subseteq M' \wedge R' \vdash M' \Rightarrow R'; M'; \delta \vdash T$

Proof. Proof by induction on the shape of T.

- $\emptyset$: $R'; M'; \delta \vdash \emptyset$ trivially holds.
- $T', n : e$: By inversion of this derivation we have that
  - $R; M; \emptyset; \emptyset \vdash e : \langle\rangle \,\&\, (\gamma; \emptyset)$: The application of lemma 21 to $R \subseteq R'$ and the typing derivation of e gives us $R'; M; \emptyset; \emptyset \vdash e : \langle\rangle \,\&\, (\gamma; \emptyset)$. The application of lemma 22 to the latter derivation, $M \subseteq M'$ and $R' \vdash M'$ gives us $R'; M'; \emptyset; \emptyset \vdash e : \langle\rangle \,\&\, (\gamma; \emptyset)$.
  - $R; M; \delta' \vdash T'$: By the induction hypothesis $R'; M'; \delta' \vdash T'$ holds.
  - $\forall (n' : e') \in T' . n' \neq n$
  - $\delta = \delta', n \mapsto \sigma; \gamma$
  - $\mathrm{pops}(\sigma; \gamma : e)$

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

Lemma 5 (*pops* expression preservation) $\sigma_0; S; e \rightarrow \sigma_0'; S'; e' \wedge \text{pops}(\sigma_2 : e) \wedge \sigma_0 = \sigma_1 + \sigma_2 \Rightarrow \exists \sigma_2'. \wedge \sigma_0' = \sigma_1 + \sigma_2' \wedge \text{pops}(\sigma_2' : e')$

Proof. Proof by case analysis on the operational rules.

Case *E-C*: the premises of this rule tell us that $\sigma_0 = \sigma; \gamma$ and $\sigma_0' = \sigma; \text{live}(\gamma', r^{\kappa'} \triangleright \pi)$. Thus, $\sigma; \gamma = \sigma + \emptyset; \gamma$ and $\sigma_0' = \sigma + \emptyset; \text{live}(\gamma', r^{\kappa'} \triangleright \pi)$. By the definition of predicate *pops* we can derive $\text{pops}(\emptyset; \text{live}(\gamma', r^{\kappa'} \triangleright \pi) : ())$.

Case *E-A*: the premises of this rule tell us that $\sigma_0 = \sigma; \gamma$ and $\sigma_0' = \sigma; \gamma_r; \gamma_1$. Thus, $\sigma; \gamma = \sigma + \emptyset; \gamma$ and $\sigma_0' = \sigma + \emptyset; \gamma_r; \gamma_1$. By inversion of the assumption that $\text{pops}(\sigma : (\lambda x.\, e \ \texttt{as}\ \tau\ v)^{\text{seq}})$ holds we have that $\text{pops}(\emptyset; \emptyset : e)$. It is trivial to show that $\text{pops}(\emptyset; \gamma_1 : e)$ also holds by induction on predicate *pops*. We combine the latter fact with $\emptyset; \gamma_r; \gamma_1 = \emptyset; \gamma_r + \emptyset; \gamma_1$ to derive $\text{pops}(\emptyset; \gamma_r; \gamma_1 : \texttt{pop}_{\gamma_r}\ e)$.

Case *E-NG*: the premises of this rule tell us that $\sigma_0 = \sigma; \gamma$ and $\sigma_0' = \sigma; \gamma, r^{1,1} \triangleright r'$. Thus, $\sigma; \gamma = \sigma + \emptyset; \gamma$ and $\sigma_0' = \sigma + \emptyset; \gamma, r^{1,1} \triangleright r'$. By inversion of the assumption that $\text{pops}(\sigma : \texttt{newrgn}^{r'}\ \rho, x\ \texttt{at}\ \texttt{rgn}_r\ \texttt{in}\ e)$ holds we have that $\text{pops}(\emptyset; \emptyset : e)$. It is trivial to show that $\text{pops}(\emptyset; \gamma, r^{1,1} \triangleright r' : e)$ also holds by induction on predicate *pops*. Thus, the proof is completed if σ_2 is equal to $\emptyset; \gamma, r^{1,1} \triangleright r'$.

Case *E-E*: the premises of this rule tell us that $\sigma_0 = \sigma; \gamma_r; \gamma'$ and $\sigma_0' = \sigma; \gamma''$. Thus, $\sigma; \gamma = \sigma + \emptyset; \gamma_r; \gamma'$ and $\sigma_0' = \sigma + \emptyset; \gamma''$. By inversion of the assumption $\text{pops}(\emptyset; \gamma_r; \gamma' : \texttt{pop}_{\gamma_r}\ v)$ we have that $\text{pops}(\emptyset; \gamma' : v)$ holds. By the definition of predicate *pops* we can rewrite the latter fact as $\text{pops}(\emptyset; \gamma'' : v)$. Thus, the proof is completed if σ_2 is equal to $\emptyset; \gamma''$.

Case *E-RP*: this rule implies that $\sigma_0 = \sigma_0' = \sigma$. The assumption that $\text{pops}(\sigma_2 : (f)\,[r])$ holds tells us that $\sigma_2 = \emptyset; \gamma$ for some γ. Thus, $\sigma = \sigma_1 + \emptyset; \gamma$. $\sigma_0' = \sigma = \sigma_1 + \emptyset; \gamma$. Thus $\sigma_2' = \emptyset; \gamma$. The assumption that $\text{pops}(\sigma_2 : (f)\,[r])$ also tells us that $\text{pops}(\emptyset; \emptyset : f)$ holds. By the definition of *pops*, $\text{pops}(\emptyset; \gamma : f)$ also holds. Thus, $\text{pops}(\sigma_2' : f)$ holds.

Case *E-D*,*E-NR*,*E-AS*: similar to the previous case.

Lemma 6 (*pops* implication) $\text{pops}(\sigma : E[e]) \Rightarrow \exists \sigma_1, \sigma_2. \sigma = \sigma_1 + \sigma_2 \wedge \text{pops}(\sigma_2 : e))$

Proof. We perform induction on the shape of E:

Case $\square[e]$: Let σ_1 and σ_2 be equal to $\emptyset$ and σ respectively. By the latter facts and the assumption pops(: $E[e]$) we have that $\text{pops}(\sigma : e)$ (assumption) holds.

Case $((E'\ e_2)^{\xi})[e]$: this is equivalent to $(E'[e]\ e_2)^{\xi}$. By inversion of the assumption we have that $pops(\sigma : E'[e])$ holds ($E'[e]$ is not value; this is dealt with in the next case). By the induction hypothesis there exists σ_1 and σ_2 such that $\sigma = \sigma_1 + \sigma_2$ and $pops(\sigma_2 : e)$.

Case $((v_1\ E')^{\xi})[e]$: this is equivalent to $(v_1\ E'[e])^{\xi}$. By inversion of the assumption we have that $pops(\sigma : E'[e])$ holds By the induction hypothesis there exists σ_1 and σ_2 such that $\sigma = \sigma_1 + \sigma_2$ and $pops(\sigma_2 : e)$.

Case $(\texttt{pop}_{\gamma}\ E'))[e]$,$(E'\ [r])[e]$: this is equivalent to $\texttt{pop}_{\gamma_r}\ E'[e]$. By inversion of the assumption we have that $\sigma = 0; \gamma_r + \sigma'$ and $\text{pops}(\sigma' : E'[e])$. By applying the induction hypothesis we have that $\sigma' = \sigma_1' + \sigma_2$ and $\text{pops}(\sigma_2 : e)$. Thus, the proof is completed if σ_1 equals $\emptyset; \gamma_r + \sigma_1'$.

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

Lemma 7 (*pops* evaluation context preservation) $\text{pops}(\sigma_1 + \sigma_2 : E[e]) \wedge \text{pops}(\sigma_2 : e) \wedge \sigma_1 + \sigma_2; S; e \rightarrow \sigma'; S'; e' \Rightarrow \exists \sigma_3. \sigma' = \sigma_1 + \sigma_3 \wedge \text{pops}(\sigma' : E[e'])$.

Proof. We proceed by induction on the structure of E:

Case $\square[e]$: the application of lemma 5 to $\text{pops}(\sigma_2 : e)$ and $\sigma_1 + \sigma_2; S; e \rightarrow \sigma'; S'; e'$ implies that there exists an σ_3 such that $\sigma' = \sigma_1 + \sigma_3$ and $\text{pops}(\sigma_3 : e')$ holds. The assumption implies that $\text{pops}(\sigma_1 + \sigma_2 : \square[e])$ and $\text{pops}(\sigma_2 : e)$. This can only hold if $\sigma_1 = \emptyset$. We have shown that $\text{pops}(\sigma_3 : e')$ holds. Thus, $\text{pops}(\sigma_1 + \sigma_3 : \square[e'])$ holds.

Case $((E'\ e_2)^{\xi})[e]$: By the definition of the evaluation context and the assumption that $\text{pops}(\sigma_1 + \sigma_2 : E[e])$ holds, we have that $\text{pops}(\sigma_1 + \sigma_2 : (E'[e]\ e_2)^{\xi})$ holds. $E'[e]$ is not a value as e is a redex (operational step assumption). Therefore, by inversion of $\text{pops}(\sigma_1 + \sigma_2 : (E'[e]\ e_2)^{\xi})$ and the latter fact we obtain that $\text{pops}(\sigma_1 + \sigma_2 : E'[e])$ and $\text{pops}(\emptyset; \emptyset : e_2)$. By applying the induction hypothesis we have that there exists an σ_3' such that $\text{pops}(\sigma' : E'[e'])$ and $\sigma' = \sigma_1 + \sigma_3'$ holds. Therefore, we can combine $\text{pops}(\sigma' : E'[e'])$ and $\text{pops}(\emptyset; \emptyset : e_2)$ to derive $\text{pops}(\sigma' : (E'[e]\ e_2)^{\xi})$.

Case $((v_1\ E')^{\xi})[e]$: By the definition of the evaluation context and the assumption that $\mathsf{pops}(\sigma_1 + \sigma_2 : E[e])$ holds, we have that $\mathsf{pops}(\sigma_1 + \sigma_2 : (v_1\ E'[e])^{\xi})$ holds. By inversion of $\mathsf{pops}(\sigma_1 + \sigma_2 : (v_1\ E'[e])^{\xi})$ and the latter fact we obtain that $\mathsf{pops}(\sigma_1 + \sigma_2 : E'[e])$ and $\mathsf{pops}(\emptyset;\emptyset : v_1)$. By applying the induction hypothesis we have that there exists an σ'_3 such that $\mathsf{pops}(\sigma' : E'[e'])$ and $\sigma' = \sigma_1 + \sigma'_3$ holds. Therefore, we can combine $\mathsf{pops}(\sigma' : E'[e'])$ and $\mathsf{pops}(\emptyset;\emptyset : v_1)$ to derive $\mathsf{pops}(\sigma' : (v_1\ E'[e'])^{\xi})$.

Case $(\mathtt{pop}_{\gamma}\ E'))[e]$: By the definition of the evaluation context and the assumption that $\mathsf{pops}(\sigma_1 + \sigma_2 : E[e])$ holds, we have that $\mathsf{pops}(\sigma_1 + \sigma_2 : \mathtt{pop}_{\gamma_r}\ E'[e])$ holds. By inversion of the latter fact we obtain that $\sigma_1 + \sigma_2 = \emptyset;\gamma_r + \sigma_x$ and $\mathsf{pops}(\sigma_x; E'[e])$. σ_2 is a postfix of σ_x as $\mathsf{pops}(\sigma_2 : e)$ would not hold otherwise (definition of *pops* predicate). Thus, there exists an σ'_x such that $\sigma_x = \sigma'_x + \sigma_2$ and $\sigma_1 = 0;\gamma_r + \sigma'_x$. By applying the induction hypothesis we obtain that $\mathsf{pops}(\sigma'_x + \sigma_3 : E'[e'])$ for some σ_3. We can combine the latter fact with $\sigma_1 = 0;\gamma_r + \sigma'_x$ to derive $\mathsf{pops}(\sigma_1 + \sigma_3 : E'[e'])$.

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

Lemma 8 (*pops* — Replace value) $\mathsf{pops}(\sigma;\gamma : E[(v\ v')^{\xi}]) \Rightarrow \forall\gamma'.\mathsf{pops}(\sigma;\gamma' : E[()])$

Proof. Proof by induction on the shape of E:

Case $\square$: this is immediate by the definition of *pops* for the unit value.

Case $(E'\ e_2)^{\xi'}$: By the definition of the evaluation context and the assumption that $\mathsf{pops}(\sigma;\gamma : E[(v\ v')^{\xi'}])$ holds, $\mathsf{pops}(\sigma;\gamma : (E'[(v\ v')^{\xi}]\ e_2)^{\xi'})$ also holds. By inversion of the latter judgement and the fact that the hole does not contain a value we obtain that $\mathsf{pops}(\sigma;\gamma : E'[(v\ v')^{\xi}])$ and $\mathsf{pops}(\emptyset;\emptyset : e_2)$. By applying the induction hypothesis we obtain that $\forall\gamma'.\mathsf{pops}(\sigma;\gamma' : E'[()])$. We can use the latter fact and $\mathsf{pops}(\emptyset;\emptyset : e_2)$ to derive $\forall\gamma'.\mathsf{pops}(\sigma;\gamma' : (E'[()]\ e_2)^{\xi'})$.

Case $(v_1\ E')^{\xi'}$: By the definition of the evaluation context and the assumption that $\mathsf{pops}(\sigma;\gamma : E[(v\ v')^{\xi'}])$ holds, $\mathsf{pops}(\sigma;\gamma : (v_1\ E'[(v\ v')^{\xi}])^{\xi'})$ also holds. By inversion of the latter judgement we obtain that $\mathsf{pops}(\sigma;\gamma : E'[(v\ v')^{\xi}])$ and $\mathsf{pops}(\emptyset;\emptyset : v_1)$. By applying the induction hypothesis we obtain that $\forall\gamma'.\mathsf{pops}(\sigma;\gamma' : E'[()])$. We can use the latter fact and $\mathsf{pops}(\emptyset;\emptyset : v_1)$ to derive $\forall\gamma'.\mathsf{pops}(\sigma;\gamma' : (v_1\ E'[()])^{\xi'})$.

Case $(\mathtt{pop}_{\gamma}\ E'))[e]$: By the definition of the evaluation context and the assumption that $\mathsf{pops}(\sigma;\gamma : E[(v\ v')^{\xi'}])$ holds, $\mathsf{pops}(\sigma;\gamma : \mathtt{pop}_{\gamma_r}\ E'[(v\ v')^{\xi}])$ also holds. By inversion of the latter judgement we obtain that $\sigma;\gamma = \emptyset;\gamma_r + \sigma'$ and $\mathsf{pops}(\sigma' : E'[(v\ v')^{\xi}])$. $\sigma;\gamma = \emptyset;\gamma_r + \sigma'$ implies that there exists a σ'' such that $\sigma' = \sigma'';\gamma$. Thus, $\mathsf{pops}(\sigma' : E'[(v\ v')^{\xi}])$ becomes $\mathsf{pops}(\sigma'';\gamma : E'[(v\ v')^{\xi}])$. The application of the induction hypothesis to the latter fact gives us that $\forall\gamma'.\mathsf{pops}(\sigma'';\gamma' : E'[(v\ v')^{\xi}])$. Thus, we can derive from the above facts that $\mathsf{pops}(\sigma;\gamma' : \mathtt{pop}_{\gamma_r}\ E'[()])$ holds.

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

Lemma 9 (Store Strengthening — Empty γ) *If store $\delta, n \mapsto (\emptyset;\emptyset); S$ is well-typed in the context $R; M$, then $\delta; S$ is also well-typed in the same context.*

Proof. By inversion of the store typing assumption we have that

- $R; M \vdash S$
- $R \vdash \delta, n \mapsto (\emptyset;\emptyset)$
- $\vdash \delta, n \mapsto (\emptyset;\emptyset)$

$R \vdash \delta$ trivially holds by observing the premise of $R \vdash \delta, n \mapsto (\emptyset;\emptyset)$. By inversion of $\vdash \delta, n \mapsto (\emptyset;\emptyset)$, we obtain that $\vdash \delta$ holds.

Therefore, the latter facts and $R; M \vdash S$ imply that $R; M \vdash \delta; S$ also holds.

Lemma 10 (Inversion)
$R; M; \Delta; \Gamma \vdash x : \tau' \,\&\, (\gamma_a;\gamma_b) \Rightarrow \gamma_a = \gamma_b = \gamma \wedge \ \vdash R; M; \Delta; \Gamma; \gamma; \gamma \ \wedge\ (x : \tau) \in \Gamma \ \wedge\ \tau \simeq \tau'$
$\wedge$
$R; M; \Delta; \Gamma \vdash \mathtt{c} : \tau \,\&\, (\gamma_a;\gamma_b) \Rightarrow \gamma_a = \gamma_b = \gamma \wedge \ \vdash R; M; \Delta; \Gamma; \gamma; \gamma \ \wedge\ \tau = b$
$\wedge$
$R; M; \Delta; \Gamma \vdash () : \tau \,\&\, (\gamma_a;\gamma_b) \Rightarrow \gamma_a = \gamma_b = \gamma \wedge \ \vdash R; M; \Delta; \Gamma; \gamma; \gamma \ \wedge\ \tau = \langle\rangle$
$\wedge$
$R; M; \Delta; \Gamma \vdash \mathtt{rgn}_{\iota} : \tau \,\&\, (\gamma_a;\gamma_b) \Rightarrow \gamma_a = \gamma_b = \gamma \wedge \ \vdash R; M; \Delta; \Gamma; \gamma; \gamma \ \wedge\ \tau \simeq \mathtt{rgn}(\iota) \ \wedge\ R; \Delta \vdash \iota$
$\wedge$

$R; M; \Delta; \Gamma \vdash \mathtt{loc}_l : \tau \,\&\, (\gamma_a;\gamma_b) \Rightarrow \gamma_a = \gamma_b = \gamma \wedge \vdash R; M; \Delta; \Gamma; \gamma; \gamma \wedge \tau \simeq \mathtt{ref}\ M(\ell)$
$\wedge$
$R; M; \Delta; \Gamma \vdash \mathtt{cap}^r_\eta\ e_1 : \tau \,\&\, (\gamma;\gamma'') \Rightarrow \tau = \langle\rangle \wedge R; M; \Delta; \Gamma \vdash e_1 : \mathtt{rgn}(r) \,\&\, (\gamma;\gamma', r^\kappa \triangleright \pi) \wedge \kappa' = [\![\eta]\!](\kappa) \wedge \gamma'' = \mathrm{live}(\gamma', r^{\kappa'} \triangleright \pi) \wedge \mathrm{is\_live}(\gamma', r^\kappa \triangleright \pi)$
$\wedge$
$R; M; \Delta; \Gamma \vdash \lambda x.e\ \mathtt{as}\ \tau : \tau' \,\&\, (\gamma_a;\gamma_b) \Rightarrow \gamma_a = \gamma_b = \gamma \wedge \vdash R; M; \Delta; \Gamma; \gamma; \gamma \wedge R; \Delta \vdash \tau \wedge \tau \equiv \tau_1 \xrightarrow{\gamma_1 \to \gamma_2} \tau_2 \wedge \gamma' = \gamma_1 \wedge \mathrm{set}(\gamma_1;\gamma_2) \wedge (\mathrm{ok}(\gamma_1;\gamma_2) \Rightarrow R; M; \Delta; \Gamma, x : \tau_1 \vdash e : \tau_2 \,\&\, (\gamma_1;\gamma_2)) \wedge \tau' \simeq \tau$
$\wedge$
$R; M; \Delta; \Gamma \vdash \Lambda\rho.\, f : \tau \,\&\, (\gamma_a;\gamma_b) \Rightarrow \gamma_a = \gamma_b = \gamma \wedge \tau = \forall\rho.\, \tau' \wedge R; M; \Delta, \rho; \Gamma \vdash f : \tau' \,\&\, (\gamma;\gamma)$
$\wedge$
$R; M; \Delta; \Gamma \vdash e\ [r] : \tau \,\&\, (\gamma;\gamma') \Rightarrow \tau = \tau'[r/\rho] \wedge R; \Delta \vdash r \wedge R; M; \Delta; \Gamma \vdash e : \forall\rho.\, \tau' \,\&\, (\gamma;\gamma')$
$\wedge$
$R; M; \Delta; \Gamma \vdash (e_1\ e_2)^\xi : \tau_2 \,\&\, (\gamma;\gamma_5) \Rightarrow R; M; \Delta; \Gamma \vdash e_1 : \tau_1 \xrightarrow{\gamma_1 \to \gamma_2} \tau_2 \,\&\, (\gamma;\gamma_3) \wedge \mathsf{par} \Rightarrow \tau_2 = \langle\rangle \wedge R; M; \Delta; \Gamma \vdash e_2 : \tau_1 \,\&\, (\gamma_3;\gamma_4) \wedge \xi \vdash \gamma_5 = \gamma_2 \oplus (\gamma_4 \ominus \gamma_1)$
$\wedge$
$R; M; \Delta; \Gamma \vdash \mathtt{newrgn}^{r'}\ \rho, x\ \mathtt{at}\ e_1\ \mathtt{in}\ e_2 : \tau \,\&\, (\gamma;\gamma'') \Rightarrow R; M; \Delta; \Gamma \vdash e_1 : \mathtt{rgn}(r) \,\&\, (\gamma;\gamma') \wedge r' \equiv r \wedge \mathrm{is\_live}(\gamma', r) \wedge R; \Delta \vdash \tau \wedge R; M; \Delta, \rho; \Gamma, x : \mathtt{rgn}(\rho) \vdash e_2 : \tau \,\&\, (\gamma', \rho^{1,1} \triangleright r;\gamma'') \wedge \rho \notin \mathrm{dom}(\gamma'')$
$\wedge$
$R; M; \Delta; \Gamma \vdash \mathtt{new}\ e_1\ \mathtt{at}\ e_2 : \tau \,\&\, (\gamma;\gamma'') \Rightarrow \tau = \mathtt{ref}(\tau', r) \wedge R; M; \Delta; \Gamma \vdash e_1 : \tau' \,\&\, (\gamma;\gamma') \wedge \mathrm{is\_live}(\gamma'', r) \wedge R; M; \Delta; \Gamma \vdash e_2 : \mathtt{rgn}(r) \,\&\, (\gamma';\gamma'')$
$\wedge$
$R; M; \Delta; \Gamma \vdash e_1 := e_2 : \tau' \,\&\, (\gamma;\gamma_2) \Rightarrow \tau' = \langle\rangle \wedge R; M; \Delta; \Gamma \vdash e_1 : \mathtt{ref}(\tau, r) \,\&\, (\gamma;\gamma') \wedge R; M; \Delta; \Gamma \vdash e_2 : \tau \,\&\, (\gamma';\gamma'') \wedge \mathrm{is\_accessible}(\gamma'', r)$
$\wedge$
$R; M; \Delta; \Gamma \vdash \mathtt{deref}\ e : \tau \,\&\, (\gamma;\gamma') \Rightarrow R; M; \Delta; \Gamma \vdash e : \mathtt{ref}(\tau, r) \,\&\, (\gamma;\gamma') \wedge \mathrm{is\_accessible}(\gamma', r)$
$\wedge$
$R; M; \Delta; \Gamma \vdash \mathtt{pop}_{\gamma_r}\ e : \tau \,\&\, (\gamma;\gamma') \Rightarrow \mathrm{ok}(\gamma_r;\emptyset) \wedge R; M; \Delta; \Gamma \vdash e : \tau \,\&\, (\gamma_1;\gamma_2) \wedge \mathsf{seq} \vdash \gamma' = \gamma_2 \oplus (\gamma_r \ominus \emptyset) \wedge R; \Delta \vdash \gamma_r \wedge \tau \simeq \tau' \wedge \mathrm{set}(\gamma_r;\emptyset)$

Proof. Straightforward pattern matching on the typing derivations.

Lemma 11 (Context Inversion) *If $E[e]$ is a well-typed expression in the typing context $R; M; \Delta; \Gamma$ with effect $(\gamma_1;\gamma_2)$, then e is also a well-typed expression for some type τ, in the same typing context with effect $(\gamma_1;\gamma_3)$ for some γ_3.*

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

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

Case $((E'\ e_2)^\xi)[e]$: An equivalent expression for this case is $(E'[e]\ e_2)^\xi$. By the assumption, $(E'[e]\ e_2)^\xi$ is a well-typed application term. Lemma 10 implies that $E[e]$ is well-typed in the same typing context with effect $(\gamma_1;\gamma')$, where γ' is its output effect. The application of the induction hypothesis to the the latter typing derivation yields that e is a well-typed term in the same typing context with effect $(\gamma_1;\gamma'')$ for some γ''.

Case $((v_1\ E')^\xi)[e]$: An equivalent expression for this case is $(v_1\ E'[e])^\xi$. Lemma 10 implies that $(v_1\ E'[e])^\xi$, $E'[e]$ and v_1 are well-typed. In addition, v_1 is a value with effect $(\gamma_1;\gamma_1)$ (this is immediate by performing a case analysis on v and applying lemma 10). Thus, the input effect of $E'[e]$ is γ_1. The application of the induction hypothesis to the latter fact implies that e is well-typed for some type τ with effect $(\gamma_1;\gamma_3)$, for some γ_3.

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

Lemma 12 (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 cases are the ones of rules *T-AP* and *T-E*:

- *T-A*: By applying lemma 10 to the typing derivation of e we have that e_1 is well-typed with effect $(\gamma;\gamma_x)$, e_2 is well-typed with effect $(\gamma_x;\gamma_y)$ and $\xi \vdash \gamma' = \gamma_2 \oplus (\gamma \ominus \gamma_y)$. By applying the induction hypothesis to e_1 and e_2 we obtain that $\vdash R; M; \Delta; \Gamma; \gamma; \gamma_x$ and $\vdash R; M; \Delta; \Gamma; \gamma_x; \gamma_y$ respectively. It suffices to prove the following obligations:
  - $R \vdash M$: immediate by inversion of $\vdash R; M; \Delta; \Gamma; \gamma; \gamma_x$.

- $R; \Delta \vdash \Gamma$: immediate by inversion of $\vdash R; M; \Delta; \Gamma; \gamma; \gamma_x$.
- $R; \Delta \vdash \gamma$: immediate by inversion of $\vdash R; M; \Delta; \Gamma; \gamma; \gamma_x$.
- $R; \Delta \vdash \gamma'$: the effect addition assumption implies that the regions of γ' is a subset of the regions of γ. Thus, $R; \Delta \vdash \gamma'$ follows from the fact that $R; \Delta \vdash \gamma$ holds as shown earlier.
- set$(\gamma; \gamma')$: by inversion of $\vdash R; M; \Delta; \Gamma; \gamma; \gamma_x$ we obtain that set$(\gamma; \emptyset)$ holds. The effect addition assumption implies that the regions of γ' are contained in the regions of γ. Thus, set$(\gamma'; \emptyset)$ is immediate from the fact that set$(\gamma; \emptyset)$. Hence set$(\gamma; \gamma')$.
- ok$(\gamma; \gamma')$: by inversion of $\vdash R; M; \Delta; \Gamma; \gamma; \gamma_x$ we obtain that ok$(\gamma; \emptyset)$ holds. The effect addition assumption implies that the regions of γ' are contained in the regions of γ and the *purity* of each atomic effect of γ' is identical to the *purity* of the same effect in γ. Thus, ok$(\gamma; \gamma')$ holds.

Case *T-E*: By applying lemma 10 to the typing derivation of e we obtain that e_1 is well-typed with effect $(\gamma; \gamma_2)$, $\mathsf{seq} \vdash \gamma' = \gamma_2 \oplus (\gamma_r \ominus \emptyset)$, ok$(\gamma_r; \emptyset)$, set$(\gamma_r; \emptyset)$ and $R; \Delta \vdash \gamma_r$. By applying the induction hypothesis to e_1 we obtain that $\vdash R; M; \Delta; \Gamma; \gamma; \gamma_2$. It suffices to prove the following obligations:

- $R \vdash M$: immediate by inversion of $\vdash R; M; \Delta; \Gamma; \gamma; \gamma_2$.
- $R; \Delta \vdash \Gamma$: immediate by inversion of $\vdash R; M; \Delta; \Gamma; \gamma; \gamma_2$.
- $R; \Delta \vdash \gamma$: immediate by inversion of $\vdash R; M; \Delta; \Gamma; \gamma; \gamma_2$.
- $R; \Delta \vdash \gamma'$: the effect addition assumption implies that the regions of γ' is a subset of the regions of γ_r. Thus, $R; \Delta \vdash \gamma'$ follows from the fact that $R; \Delta \vdash \gamma_r$.
- set$(\gamma; \gamma')$: the effect addition assumption implies that the regions of γ' are contained in the regions of γ_r. Thus, set$(\gamma'; \emptyset)$ is immediate from the fact that set$(\gamma_r; \emptyset)$. By inversion of $\vdash R; M; \Delta; \Gamma; \gamma; \gamma_2$ we obtain that set$(\gamma; \emptyset)$ holds. Hence set$(\gamma; \gamma')$.
- ok$(\gamma; \gamma')$: we have shown that ok$(\gamma_r; \emptyset)$ holds. The effect addition assumption implies that the regions of γ' are contained in the regions of γ_r and the *purity* of each atomic effect of γ' is identical to the *purity* of the same effect in γ_r. Hence, ok$(\gamma'; \emptyset)$ holds. By inversion of $\vdash R; M; \Delta; \Gamma; \gamma; \gamma_2$ we obtain that ok$(\gamma; \emptyset)$ holds. Thus, ok$(\gamma; \gamma')$ holds.

Lemma 13 (Value-Effect — Using well-formedness) *If value v is well-typed in the typing context $R; M; \Delta; \Gamma$, with effect $(\gamma; \gamma)$ and $\vdash R; M; \Delta; \Gamma; \gamma_1; \gamma_2$, then v is well-typed in the same typing context with effect $(\gamma_1; \gamma_1)$ and $(\gamma_2; \gamma_2)$.*

Proof. The proof is trivial, but we provide the key steps behind the proof. The assumption implies that $\vdash R; M; \Delta; \Gamma; \gamma_1; \gamma_1$ and also $\vdash R; M; \Delta; \Gamma; \gamma_2; \gamma_2$ hold (trivial). By lemma 10 we obtain the well-formedness derivation as well as some other premises (in the case of rules *T-L,T-R,T-V,T-F*). We may use the latter premises of value typing, which *still hold* (same typing context), along with the latter two well-formedness derivations to formulate the new value typing derivations with effect $(\gamma_1; \gamma_1)$ and $(\gamma_2; \gamma_2)$ respectively. The case for rule *T-RF* can be shown trivially by induction (the base case is the same as for rule *T-F*).

Lemma 14 (Value-Effect) *If value v is well-typed in the typing context $R; M; \Delta; \Gamma$, with effect $(\gamma; \gamma)$, and e is well-typed in the same typing context with effect $(\gamma'; \gamma'')$, then v is well-typed in the same typing context with effect $(\gamma''; \gamma'')$ and $(\gamma'; \gamma')$.*

Proof. By applying lemma 10 to the typing derivation of v, we have that $\vdash R; M; \Delta; \Gamma; \gamma; \gamma$. Similarly, the application of lemma 12 to the typing derivation of e implies that $\vdash R; M; \Delta; \Gamma; \gamma'; \gamma''$. The proof is completed by applying lemma 13.

Lemma 15 (R Well-Formedness Weakening) $R; \Delta \vdash r \wedge R \subseteq R' \Rightarrow R'; \Delta \vdash r$

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

- $\iota@n$: By inversion of this derivation we have that $R; \Delta \vdash \iota$. We can use the induction hypothesis to complete the proof.
- $r \neq \iota@n$: By inversion of this derivation $\bar{r} \in R \uplus \Delta$ holds. Thus, $\bar{r} \in R' \uplus \Delta$ also holds.

Lemma 16 (Effect Well-formedness Weakening) $R; \Delta \vdash \gamma \wedge R \subseteq R' \Rightarrow R'; \Delta \vdash \gamma$

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

- $\emptyset$: $R'; \Delta \vdash \emptyset$ trivially holds.
  $R; \Delta \vdash \gamma', r^{\star} \triangleright \pi$: $R'; \Delta \vdash \gamma'$ holds by the induction hypothesis. $R'; \Delta \vdash r$ holds by lemma 15. If $\pi = r'$, then $R'; \Delta \vdash r'$ holds by lemma 15.

Lemma 17 (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 τ:

- b: $R'; \Delta \vdash b$ trivially holds.
- $\langle\rangle$: $R'; \Delta \vdash \langle\rangle$ trivially holds.
- $\texttt{rgn}(r)$: $R'; \Delta \vdash r$ holds by lemma 15.
- $\texttt{ref}(\tau', r)$: $R'; \Delta \vdash r$ holds by lemma 15. $R'; \Delta \vdash \tau'$ holds by the induction hypothesis.
- $\forall \rho.\, \tau'$: $R'; \Delta, \rho \vdash \tau'$ holds by the induction hypothesis.
- $\tau' \xrightarrow{\gamma_1 \to \gamma_2} \tau''$: $R'; \Delta \vdash \tau'$ holds by the induction hypothesis. $R'; \Delta \vdash \tau''$ holds by the induction hypothesis. $R'; \Delta \vdash \gamma_1$ holds by lemma 16. $R'; \Delta \vdash \gamma_2$ holds by lemma 16.

Lemma 18 (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 Γ:

- $\emptyset$: $R'; \Delta \vdash \emptyset$ trivially holds.
  $R; \Delta \vdash \Gamma', x : \tau$: $R'; \Delta \vdash \Gamma'$ holds by the induction hypothesis. $R'; \Delta \vdash \tau$ holds by lemma 17.

Lemma 19 (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:

- $\emptyset$: $R' \vdash \emptyset$ trivially holds.
  $R \vdash M', \ell \mapsto (\tau, \iota)$: $R' \vdash M'$ holds by the induction hypothesis. $R'; \emptyset \vdash \texttt{ref}(\tau, \iota)$ holds by lemma 17.

Lemma 20 (Typing Context Well-formedness Weakening) $\vdash R; M; \Delta; \Gamma; \gamma_1; \gamma_2 \land R \subseteq R' \Rightarrow \vdash R'; M; \Delta; \Gamma; \gamma_1; \gamma_2$

Proof. Immediate by lemmas 19, 18, 16.

Lemma 21 (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 12 to the typing derivation of e we have that $\vdash R; M; \Delta; \Gamma; \gamma_1; \gamma_2$. Lemma 20 implies that $\vdash R'; M; \Delta; \Gamma; \gamma_1; \gamma_2$ holds.

- *T-I*: Immediate by applying rule *T-I* to $\vdash R'; M; \Delta; \Gamma; \gamma_1; \gamma_2$.
- *T-U*: Immediate by applying rule *T-U* to $\vdash R'; M; \Delta; \Gamma; \gamma_1; \gamma_2$.
- *T-R*: By applying lemma 10 to this derivation we have that $R; \Delta \vdash \iota$ and $r \simeq \iota$. Lemma 15 implies that $R'; \Delta \vdash \iota$ holds. Thus, we can apply rule *T-R* to the latter fact, $r \simeq \iota$ and $\vdash R'; M; \Delta; \Gamma; \gamma_1; \gamma_2$ to complete the proof.
- *T-L*: By applying lemma 10 to this derivation we have that $(\ell \mapsto (\tau', \iota)) \in M$ and $\texttt{ref}\ M(\ell) \simeq \tau$. Thus, we can apply rule *T-L* to $\vdash R'; M; \Delta; \Gamma; \gamma_1; \gamma_2$, $(\ell \mapsto (\tau', \iota)) \in M$ and $\texttt{ref}\ M(\ell) \simeq \tau$ to to complete the proof.
- *T-V*: By applying lemma 10 to this derivation we have that $(x : \tau') \in \Gamma$ and $\tau' \simeq \tau$. Thus, we can apply rule *T-V* to $\vdash R'; M; \Delta; \Gamma; \gamma_1; \gamma_2$, $(x : \tau') \in \Gamma$ and $\tau' \simeq \tau$ to complete the proof.

- *T-F*: By applying lemma 10 to this derivation we have that
  - $\vdash R; M; \Delta; \Gamma; \gamma; \gamma$: We have shown that $\vdash R'; M; \Delta; \Gamma; \gamma_1; \gamma_2$ holds.
  - $R; \Delta \vdash \tau$: $R'; \Delta \vdash \tau$ holds by lemma 17.
  - $\tau' \simeq \tau$
  - $\tau \equiv \tau_1 \xrightarrow{\gamma_1 \to \gamma_2} \tau_2$
  - $\text{set}(\gamma_1; \gamma_2)$
  - $\text{ok}(\gamma_1; \gamma_2) \Rightarrow R; M; \Delta; \Gamma, x : \tau_1 \vdash e' : \tau_2 \,\&\, (\gamma_1;\gamma_2)$: Assuming that $\text{ok}(\gamma_1; \gamma_2)$ holds, we apply the induction hypothesis to the derivation of e' to derive that $R'; M; \Delta; \Gamma, x : \tau_1 \vdash e' : \tau_2 \,\&\, (\gamma_1;\gamma_2)$ holds.

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

Case *T-AP*, *T-CP*, *T-RP*, *T-NG*, *T-NR*, *T-D*, *T-RF*, *T-E*, *T-A*: similar reasoning is performed to prove the remaining cases. Lemmas 15 and 17 can be used for premises of the form $R; \Delta \vdash r$ and $R; \Delta \vdash \tau$ respectively.

Lemma 22 (Memory Context Weakening) *If expression e is well-typed in the typing context* $R; M; \Delta; \Gamma$, $R \vdash M'$ *holds, and* M' *is a superset of M, then e is well-typed in the context* $R; M'; \Delta; \Gamma$ *withthe same type and effect.*

Proof. By applying lemma 12 to the typing derivation of v we have that $\vdash R; M; \Delta; \Gamma; \gamma_1; \gamma_2$. Thus, we can substitute premise $R \vdash M$ with $R \vdash M'$ to obtain $\vdash R; M'; \Delta; \Gamma; \gamma_1; \gamma_2$.

- *T-I*: Immediate by applying rule *T-I* to $\vdash R; M'; \Delta; \Gamma; \gamma_1; \gamma_2$.
- *T-U*: Immediate by applying rule *T-U* to $\vdash R; M'; \Delta; \Gamma; \gamma_1; \gamma_2$.
- *T-R*: By applying lemma 10 to this derivation we have that $R; \Delta \vdash \iota$. The proof is completed by applying rule *T-R* to $R; \Delta \vdash \iota$. and $\vdash R; M'; \Delta; \Gamma; \gamma_1; \gamma_2$.
- *T-L*: By applying lemma 10 to this derivation we have that $(\ell \mapsto (\tau', \iota)) \in M$ and $\texttt{ref } M(\ell) \simeq \tau$. Thus, $(\ell \mapsto (\tau', \iota)) \in M'$ and $\texttt{ref } M'(\ell) \simeq \tau$ also hold as $M \subseteq M'$. We can apply rule *T-L* to $\vdash R; M'; \Delta; \Gamma; \gamma_1; \gamma_2$, $(\ell \mapsto (\tau', \iota)) \in M'$ and $\texttt{ref } M'(\ell) \simeq \tau$ to to complete the proof.
- *T-V*: By applying lemma 10 to this derivation we have that $(x : \tau') \in \Gamma$ and $\tau' \simeq \tau$. Thus, we can apply rule *T-V* to $\vdash R; M'; \Delta; \Gamma; \gamma_1; \gamma_2$, $(x : \tau') \in \Gamma$ and $\tau' \simeq \tau$ to complete the proof.
- *T-F*: By applying lemma 10 to this derivation we have that
  - $\vdash R; M; \Delta; \Gamma; \gamma; \gamma$: We have shown that $\vdash R; M'; \Delta; \Gamma; \gamma_1; \gamma_2$ holds.
  - $\text{set}(\gamma_1; \gamma_2)$
  - $R; \Delta \vdash \tau$
  - $\tau' \simeq \tau$
  - $\tau \equiv \tau_1 \xrightarrow{\gamma_1 \to \gamma_2} \tau_2$
  - $\text{ok}(\gamma_1; \gamma_2) \Rightarrow R; M; \Delta; \Gamma, x : \tau_1 \vdash e' : \tau_2 \,\&\, (\gamma_1;\gamma_2)$: assume that $\text{ok}(\gamma_1; \gamma_2)$ holds. By applying the induction hypothesis to this derivation we have that $R; M'; \Delta; \Gamma, x : \tau_1 \vdash e' : \tau_2 \,\&\, (\gamma_1;\gamma_2)$ holds.

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

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

Lemma 23 (Replacement) *If expressions* $E[e_1]$, e_1 *and* e_2 *are well-typed in the typing context* $R; M; \Delta; \Gamma$, *with effects* $(\gamma_1; \gamma_2)$,$(\gamma_1; \gamma_3)$ *and* $(\gamma_4; \gamma_3)$ *respectively, then expression* $E[e_2]$ *is also well-typed in the same typing context with effect* $(\gamma_4; \gamma_2)$.

Proof. By straightforward induction on the shape of the evaluation context. The intuition behind this proof is that the substitution of e_2 for e_1 in the evaluation context E will not surpise its environment as both e_1 and e_2 yield the same output effect. In regards to the input effect, we know that the environment will not be surprised as the expressions preceding e_1 will definitely be values and can be given the input effect of e_2 (by lemma 14).

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

Case $(\texttt{new}\ v\ \texttt{at}\ E')[e]$: Lemma 10 implies that that $R; M; \Delta; \Gamma \vdash v : \tau_1 \& (\gamma_1; \gamma_1)$. The application of lemma 14 to the latter judgement and the fact e_2 is well-typed with effect $(\gamma_4; \gamma_3)$ yields $R; M; \Delta; \Gamma \vdash v : \tau_1 \& (\gamma_4; \gamma_4)$. By applying lemma 10 to the memory allocation construct typing derivation yields is_live(γ_3, r) and $R; M; \Delta; \Gamma \vdash E'[e] : \texttt{rgn}(r) \& (\gamma_1; \gamma_2)$. The application of the induction hypothesis on the derivation of $E'[e_2]$ and the derivation of e_2 (assumption) yields $R; M; \Delta; \Gamma \vdash E'[e_2] : \tau_1 \,\&\, (\gamma_4; \gamma_2)$. Now, *T-NR* can be applied to the latter judgment, the new derivation of v, and the fact that is_live(γ_3, r) to obtain $R; M; \Delta; \Gamma \vdash \texttt{new}\ v\ \texttt{at}\ E'[e_2] : \texttt{ref}(\tau_1, r) \,\&\, (\gamma_4; \gamma_2)$ or equivalently $R; M; \Delta; \Gamma \vdash (\texttt{new}\ v\ \texttt{at}\ E')[e_2] : \texttt{ref}(\tau_1, r) \,\&\, (\gamma_4; \gamma_2)$.

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

Lemma 24 (Parallel-Sequential typing implication) *If a parallel application term is well-typed ($R; M; \Delta; \Gamma \vdash (v_1\ v_2)^{\mathsf{par}} : \langle\rangle \& (\gamma; \gamma')$), where $v_1 \equiv \lambda x.\, e\ \texttt{as}\ \tau_1 \xrightarrow{\gamma_1 \to \emptyset} \langle\rangle$, then the corresponding sequential application term ($(v_1\ v_2)^{\mathsf{seq}}$) is also well-typed in the same typing context, with effect $(\gamma_1; \emptyset)$.*

Proof. Lemma 10 implies that v_1 and v_2 are well typed in the same typing context $R; M; \Delta; \Gamma$, with effects $(\gamma; \gamma)$ and $(\gamma; \gamma)$ respectively. It also implies that $R; M; \Delta; \Gamma, x : \tau_1 \vdash e : \langle\rangle \& (\gamma_1; \emptyset)$. By applying lemma 14 to the typing derivations of v_1,v_2, and the fact that e is well-typed with effect $(\gamma_1; \emptyset)$, we obtain that v_1 and v_2 are well-typed in the same typing context with effect $(\gamma_1; \gamma_1)$. We can derive $\mathsf{seq} \vdash \emptyset = \emptyset \oplus (\gamma_1 \ominus \gamma_1)$. By applying *T-AP* to the latter facts, we have that $R; M; \Delta; \Gamma \vdash (v_1\ v_2)^{\mathsf{seq}} : \langle\rangle \& (\gamma_1; \emptyset)$ holds.

Lemma 25 (Store Typing Preservation for $\vdash \delta$ — Helper 1) *If $\sigma; \gamma \vdash \delta''$, $\mathrm{ok}(\gamma; \gamma')$ and $\mathsf{par} \vdash \gamma' = \emptyset \oplus (\gamma \ominus \gamma_1)$ hold, $\delta_y \subseteq \delta''$ then $\sigma; \gamma' \vdash \delta_y$ also holds.*

Proof. Proof by induction on the structure of δ_y:

- $\emptyset$: given that $\sigma; \gamma' \simeq \sigma_1; \gamma_x, \iota^{\kappa} \triangleright \pi + \sigma_2$, $\mathrm{rg}(\kappa) > 0$ and is_pure(κ) hold, it suffices to prove that zero_pure(σ_1, ι) and $\iota \notin \mathrm{dom}(\sigma_2; \gamma_x)$ for all ι in the domain of $\sigma; \gamma'$. The assumption that $\sigma; \gamma \vdash \delta''$ holds implies that $\sigma; \gamma \simeq \sigma_3; \gamma_y, \iota^{\kappa'} \triangleright \pi' + \sigma_4$, $\iota \notin \mathrm{dom}(\sigma_4; \gamma_y)$ and zero_pure(σ_3, ι). We proceed by performing a case analysis:
  - ι belongs in the domain of $\emptyset; \gamma$: the following constraints hold from the above facts: $\sigma_2 = \sigma_4 = \emptyset$ and $\sigma_1 \simeq \sigma_3 \simeq \sigma$, $\gamma \simeq \gamma_y, \iota^{\kappa'} \triangleright \pi'$ and $\gamma' \simeq \gamma_x, \iota^{\kappa} \triangleright \pi$. Thus, zero_pure$(\sigma_1, \iota)$ holds and the assumption $\mathrm{ok}(\gamma; \gamma')$ implies that $\iota \notin \mathrm{dom}(\sigma_2; \gamma_x)$.
  - ι does not belong in the domain of $\emptyset; \gamma$: the following constraints hold from the above facts: $\sigma_1; \gamma_x, \iota^{\kappa} \triangleright \pi \simeq \sigma_3; \gamma_y, \iota^{\kappa'} \triangleright \pi'$, $\sigma_2 \simeq \sigma_a; \gamma'$, $\sigma_4 \simeq \sigma_b; \gamma$ and $\sigma_a \simeq \sigma_b$. Thus, zero_pure(σ_1, ι) is immediate. The equalties and $\iota \notin \mathrm{dom}(\sigma_4; \gamma_y)$ imply that $\iota \notin \mathrm{dom}(\sigma_b)$ and thus $\iota \notin \mathrm{dom}(\sigma_a)$. It suffices to show that $\iota \notin \gamma_x$. This is immediate by the fact that $\iota \notin \mathrm{dom}(\sigma_2; \gamma_y)$ and $\mathrm{dom}(\emptyset; \gamma_y) \subseteq \mathrm{dom}(\emptyset; \gamma_x)$ (by the capability addition assumption).

Case $\delta_1, n_1 \mapsto \sigma_1$: by applying the induction hypothesis we have that $\sigma; \gamma' \vdash \delta_1$ holds. Given that is_accessible$(\sigma; \gamma', \iota)$ holds for all ι that belong in the domain of $\sigma; \gamma'$, it suffices to prove that $\neg$is_accessible(σ_1, ι) holds. The assumption that $\sigma; \gamma \vdash \delta''$ holds implies that is_accessible$(\sigma; \gamma, \iota) \Rightarrow \neg$is_accessible$(\sigma_1, \iota)$. The capability addition assumption implies that if is_accessible$(\sigma; \gamma', \iota)$, then is_accessible$(\sigma; \gamma, \iota)$. Thus, the latter two facts imply that $\neg$is_accessible(σ_1, ι).

Lemma 26 (Store Typing Preservation for $\vdash \delta$ — Helper 2) *If $\vdash \delta'', n \mapsto \sigma; \gamma$, $\mathsf{par} \vdash \gamma' = \emptyset \oplus (\gamma \ominus \gamma_1)$ hold and $\delta_y \subseteq \delta'', n \mapsto \sigma; \gamma'$, then $\emptyset; \gamma_1 \vdash \delta_y$ holds.*

Proof. Proof by induction on the shape of δ_y.

- $\emptyset$: given that $\emptyset; \gamma_1 \simeq \sigma_1; \gamma_x, \iota^{\kappa} \triangleright \pi + \sigma_2$, $\mathrm{rg}(\kappa) > 0$ and is_pure(κ) hold, then it suffices to prove that zero_pure(σ_1) and $\iota \notin \mathrm{dom}(\sigma_2; \gamma)$. for all ι in the domain of $\emptyset; \gamma_1$. This is immediate by the fact that σ_1 and σ_2 are empty and $\mathrm{ok}(\gamma_1; \gamma_2)$ (obtained by inversion of the effect addition assumption).
- $\delta_1, n_1 \mapsto \sigma_1$: $\emptyset; \gamma_1 \vdash \delta_1$ is immediate by applying the induction hypothesis. Given that is_accessible$(0; \gamma_1, \iota)$, it suffices to prove $\neg$is_accessible(σ_1, ι) holds for all ι that belong in the domain of γ_1. If ι exists in the domain of γ_1, then the effect addition assumption tells us that ι exists in γ and is_accessible(γ, ι) holds. Thus, by inversion of $\vdash \delta'', n \mapsto \sigma; \gamma$ we have that $\neg$is_accessible(σ_1, ι) holds, when $n_1 \neq n$.

  To complete the proof it must be proved that $\neg$is_accessible$(\sigma; \gamma', \iota)$ holds. The capability addition assumption implies that ι or at least one of its ancestors has a positive *pure* capability in both γ and γ_1. It also tells us that there exists no positive *impure* capability in γ_1. Assume $\jmath$ is a region protecting ι (may be equal to ι) with a positive and pure capability. By inversion of $\vdash \delta'', n \mapsto \sigma; \gamma$ we have that $\sigma; \gamma \vdash \emptyset$ holds. By inversion of the latter derivation we have that zero_pure$(\sigma_1, \jmath)$ and $\jmath \notin \mathrm{dom}(\sigma_2; \gamma_3)$. Region $\jmath$ is positive in γ thus, $\sigma_1 = \sigma$, $\sigma_2 = \emptyset$ and $\gamma = \gamma_3, \jmath^{\kappa'} \triangleright \pi'$. Consequently, $\neg$is_accessible$(\sigma; \gamma', \jmath)$ holds by the latter fact and the effect addition assumption.

Lemma 27 (Store Typing Preservation for $\vdash \delta$ — Spawn) *If $\delta = \delta'', n \mapsto \sigma; \gamma$, $\delta = \delta'', n \mapsto \sigma; \gamma', n' \mapsto \emptyset; \gamma_1$, $\text{ok}(\gamma; \gamma')$, $\vdash \delta$ and $\mathsf{par} \vdash \gamma' = \emptyset \oplus (\gamma \ominus \gamma_1)$ hold, then $\vdash \delta'$ holds.*

Proof. It suffices to show that:

- $\vdash \delta''$: immediate by inversion of $\vdash \delta$.
- $\sigma; \gamma' \vdash \delta''$: by inversion of $\vdash \delta$ we obtain that $\sigma; \gamma \vdash \delta''$. The proof for this case is completed by the application of lemma 25.
- $\emptyset; \gamma_1 \vdash \delta'', n \mapsto \sigma; \gamma'$: the proof for this case is immediate by lemma 26.

Lemma 28 (Store Typing Preservation — Spawn) *If $\delta; S$ is a well-typed store in respect to $R; M$ where δ equals $\delta'', n \mapsto \sigma; \gamma$, $\text{ok}(\gamma; \gamma')$ holds, $\mathsf{par} \vdash \gamma' = \emptyset \oplus (\gamma \ominus \gamma_1)$ holds, $\text{live}(\gamma_1) = \gamma_1$, and δ' equals $\delta'', n \mapsto \sigma; \gamma', n' \mapsto \emptyset; \gamma_1$ (fresh n'), then $\delta'; S$ is well-typed in respect to $R; M$.*

Proof. By inversion of the store typing assumption, we have that:

- $R \vdash \delta$
- $R; M \vdash S$
- $\vdash \delta$

The capability addition assumption implies that the regions of γ_1 and γ' are subsets of the regions of γ. Therefore $R \vdash \delta$ implies that $R \vdash \delta'$ holds. We have that $R; M \vdash S$, thus it suffices to show that $\vdash \delta$ and $\mathsf{par} \vdash \gamma' = \emptyset \oplus (\gamma \ominus \gamma_1)$ imply that $\vdash \delta'$ holds. This is immediate by lemma 27.

Lemma 29 (Preservation — Expressions) *Let e be a well-typed expression with, $\vdash \delta[n \mapsto \sigma'; \gamma']$, $\delta(n) = \sigma; \gamma$, $R; M; \emptyset; \emptyset \vdash e : \tau \,\&\, (\gamma; \gamma'')$ and $R; M \vdash \delta; S$. If the operational semantics takes a step $\delta(n); S; e \to (\sigma'; \gamma'); S'; e'$, then there exist $R' \supseteq R$ and $M' \supseteq M$, such that the resulting expression and the resulting store are well-typed with $R'; M'; \emptyset; \emptyset \vdash e' : \tau \,\&\, (\gamma'; \gamma'')$, $R; M \vdash \delta[n \mapsto \sigma'; \gamma']; S'$*

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

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

Case *T-E*: The shape of u is $\mathsf{pop}_{\gamma_r}\ v$ for some value v. By applying lemma 10 to rule *T-E*, we have that $\text{ok}(\gamma_r; \emptyset)$, $\mathsf{seq} \vdash \gamma' = \gamma_2 \oplus (\gamma_r \ominus \emptyset)$ and $R; M; \emptyset; \emptyset \vdash v : \tau' \& (\gamma_1; \gamma_2)$, where γ_1 and γ' is the input and output effect of $\mathsf{pop}_{\gamma_r}\ v$ respectively, and $\tau' \simeq \tau$. By applying lemma 10 to the latter fact we have that $\gamma_1 = \gamma_2$. Thus, the earlier facts can be rewritten as $\mathsf{seq} \vdash \gamma' = \gamma_1 \oplus (\gamma_r \ominus \emptyset)$ and $R; M; \emptyset; \emptyset \vdash v : \tau' \& (\gamma_1; \gamma_1)$. The application of lemma 34 to the latter derivation and $\tau \simeq \tau'$ implies that $R; M; \emptyset; \emptyset \vdash v : \tau \& (\gamma_1; \gamma_1)$ holds.

The operational rule that matches the shape of u is *E-E* and gives us that $\delta; S; \mathsf{pop}_{\gamma_r}\ v$ evaluates to $\delta'; S; v$. The premises of rule *E-E* are $\mathsf{seq} \vdash \gamma'' = \gamma_1 \oplus (\gamma_r \ominus \emptyset)$, where δ and δ' equal $\delta'', n \mapsto \sigma; \gamma_r; \gamma_1$ and $\delta'', n \mapsto \sigma; \gamma''$ respectively. The capability addition rule is deterministic, thus γ'' equals γ'. The application of lemma 13 to $\vdash R; M; \emptyset; \emptyset; \gamma'; \gamma'$ and $R; M; \emptyset; \emptyset \vdash v : \tau \& (\gamma_1; \gamma_2)$, yields $R; M; \emptyset; \emptyset \vdash v : \tau \& (\gamma'; \gamma')$. To complete the proof, we need to show that $R; M \vdash \delta'; S$. This is immediate by the application of lemma 52 to $R; M \vdash S$, $\text{ok}(\gamma_r; \gamma_1)$ (obtained by $\text{ok}(\gamma_r; \emptyset)$ and $\text{ok}(\gamma_1; \gamma_2)$; $\text{ok}(\gamma_1; \gamma_2)$ is immediate by applying lemma 12 to the typing derivation of v), $\mathsf{seq} \vdash \gamma'' = \gamma_1 \oplus (\gamma_r \ominus \emptyset)$, $\delta = \delta'', n \mapsto \sigma; \gamma_r; \gamma_1$ and $\delta' = \delta'', n \mapsto \sigma; \gamma''$.

Case *T-RP*: The typing derivation of *T-RP* gives us that u is of the form $(e)\,[r]$. The operational rule that matches the shape of u is *E-RP*. Thus, u is of the form $(\Lambda\rho.\, f)\,[r]$. We can apply lemma 10 to the latter derivation to obtain that $R; M; \emptyset, \rho; \emptyset \vdash f : \tau \& (\gamma; \gamma)$, where γ equals $\delta(n)$. The application of lemma 35 to the latter fact, $\overline{\bar{r}@n'} \in R$ (premise $R; \emptyset \vdash r$ of rule *T-RP*), the fact that $\bar{r}@n'$ is *fresh* (premise of rule *E-RP*), $R; \emptyset \vdash \gamma$ (premise of $\vdash R; M; \emptyset; \emptyset; \gamma; \gamma$; the well-formedness fact is immediate by the application of lemma 12 to the typing derivation of type application), $\overline{\bar{r}@n'} \simeq r$ (by the premise of rule *E-RP* and the definition of relation $\simeq$), gives us that $R; M; \emptyset; \emptyset \vdash f[\bar{r}@n'/\rho] : \tau[r/\rho] \& (\gamma; \gamma)$.

Therefore, typing is preserved. The resulting store is identical to the input store, thus it is also well-typed by the assumption of this lemma.

Case *T-CP*: **Expression typing:** The application of lemma 12 to the typing derivation of the assumption gives us that $\vdash R; M; \emptyset; \emptyset; \gamma; \gamma''$, where γ is the equal to $\delta(n)$. Thus, $\vdash R; M; \emptyset; \emptyset; \gamma''; \gamma''$ also holds. The application of rule *T-U* to the latter fact gives us $R; M; \emptyset; \emptyset \vdash () : \langle\rangle\&(\gamma''; \gamma'')$.

Store typing: The operational rule *E-C* matches the shape of u. Thus, we need to prove that $R; M \vdash \delta'; S$ holds, where $\delta' = \delta[n \mapsto \sigma'; \gamma']$. It suffices to show that $\vdash \delta'$ holds. This is immediate by the assumptions of this lemma.

Case *T-NG*: Rule *E-NG* matches the shape of u. This rule implies that $\sigma; S; \texttt{newrgn}^r\ \rho, x\ \texttt{at}\ \texttt{rgn}_{\bar{r}}\ \texttt{in}\ e_2 \rightarrow (\sigma; \gamma, \iota^{1,1} \triangleright r); S'; e_2[\iota/\rho][\texttt{rgn}_\iota/x]$, $\sigma = \sigma_0; \gamma$, is_live$(\gamma, r)$, $S' = S, \iota \mapsto \emptyset$, fresh ι and $\delta' = \delta, n \mapsto \sigma; \gamma, \iota^{1,1} \triangleright r$ hold.

Store typing: We must prove that $R, \iota; M \vdash \delta'; S'$ hold given that $R; M \vdash \delta; S$ holds. The latter derivation gives us that $R; M \vdash S$, $R \vdash S$ and $\vdash \delta$.

- $R, \iota; M \vdash S'$:
  - $M \vdash S'$: Trivially holds as $M \vdash S$ holds and $S' = S, \iota \mapsto \emptyset$.
  - $R, \iota \vdash S'$: Trivially holds as $R \vdash S$ holds and $S' = S, \iota \mapsto \emptyset$.
- $R, \iota \vdash \delta'$: $R \vdash \delta$ holds and the only new region introduced in δ' is ι.
- $\vdash \delta[n \mapsto \sigma_0; \gamma, \iota^{1,1} \triangleright r]$: immediate by the assumption of this lemma.

Expression typing: The store typing derivation of $\delta'; S'$ implies that $\iota \notin R$. Lemma 10 implies that $R; M; \emptyset, \rho; \emptyset, x : \texttt{rgn}(\rho) \vdash e_2 : \tau\&(\gamma, \rho^{1,1} \triangleright r; \gamma'')$, such that $\rho \notin dom(\gamma'')$. The application of lemma 21 to the typing derivation of e_2 and the fact that $\iota \notin R$ yields $R, \iota; M; \emptyset, \rho; \emptyset, x : \texttt{rgn}(\rho) \vdash e_2 : \tau\&(\gamma, \rho^{1,1} \triangleright r; \gamma'')$. By applying lemma 12 to the original typing derivation of *newrgn* construct we obtain the well-formedness derivation. By inversion of the latter derivation we have that ok$(\gamma; \gamma'')$, thus ok$(\gamma; \emptyset)$ holds. ρ is a fresh type variable so it does not exist in the domain of γ. Thus, ok$(\gamma, \rho^{1,1} \triangleright r)$ also holds. We then apply lemma 46 on the latter fact, the derivation of e_2 and the fact that ι is *fresh* to obtain $R, \iota; M; \emptyset; \emptyset, x : \texttt{rgn}([\iota/\rho]) \vdash e_2[\iota/\rho] : \tau[\iota/\rho]\&(\gamma[\iota/\rho], \iota^{1,1} \triangleright r; \gamma''[\iota/\rho])$. By applying lemma 12 to the original typing derivation of *newrgn* construct we have that the typing the context (including γ and γ'') is not defined in terms of ρ (i.e. ρ is *fresh*). Further, the premise of *newrgn* derivation suggests that τ is also independent of ρ (i.e. $R; \emptyset \vdash \tau$). Hence, the above facts and the definition of the substitution relation imply that the typing derivation of e_2 becomes $R, \iota; M; \emptyset; \emptyset, x : \texttt{rgn}(\iota) \vdash e_2[\iota/\rho] : \tau\&(\gamma, \iota^{1,1} \triangleright r; \gamma'')$. By the application of lemma 12 to the fact that e_2 is well-typed, we have that $\vdash R, \iota; M; \emptyset; \emptyset; \gamma, \rho^{1,1} \triangleright r; \gamma''$ is well formed. By the definition of well-formedness, $\vdash R, \iota; M; \emptyset; \emptyset; \emptyset; \emptyset$ also holds. The definition of the typing rule *T-R*, the latter fact and the fact that $R, \iota \vdash \iota$ holds imply that $\texttt{rgn}_\iota$ is well-typed (with type $\texttt{rgn}(\iota)$) in the context $R, \iota; M; \emptyset; \emptyset$ with effect $(\emptyset; \emptyset)$. By applying lemma 36 to the latter derivation and the fact that $R, \iota; M; \emptyset; \emptyset, x : \texttt{rgn}(\iota) \vdash e_2[k/\rho] : \tau\&(\gamma, \iota^{1,1} \triangleright r; \gamma'')$ we obtain $R, \iota; M; \emptyset; \emptyset \vdash e_2[\iota/\rho][\texttt{rgn}_\iota/x] : \tau\&(\gamma, \iota^{1,1} \triangleright r; \gamma'')$.

Case *T-D*: Rule *E-D* matches the shape of u. Its premises also imply that the value read from the store is equal to $S(\bar{r})(\ell)$, for some r such that $\bar{r} = \iota$. The store typing assumption yields that $R; M; \emptyset; \emptyset \vdash v : \tau'\&(\emptyset; \emptyset)$, where $v = S(\iota)(\ell)$ and $M(\ell) = (\tau', \iota)$.

The application of lemma 12 to the typing derivation of *deref* gives us $\vdash R; M; \emptyset; \emptyset; \gamma; \gamma$. By applying lemma 13 to the latter derivation and $R; M; \emptyset; \emptyset \vdash v : \tau'\&(\emptyset; \emptyset)$ gives us that $R; M; \emptyset; \emptyset \vdash v : \tau'\&(\gamma; \gamma)$. By applying lemma 10 to the typing derivation of *deref* we have that if $\texttt{ref}(\tau'', r)$ is the type assigned to $\texttt{loc}_\ell$, then $\texttt{ref}(\tau'', r) \simeq \texttt{ref}\ M(\ell)$ holds. Thus $\tau'' \simeq \tau'$ also holds. We can use lemma 34, the latter fact and $R; M; \emptyset; \emptyset \vdash v : \tau'\&(\gamma; \gamma)$ (lemma 10) to derive $R; M; \emptyset; \emptyset \vdash v : \tau''\&(\gamma; \gamma)$. The output store is identical to the input store hence it is also well-typed.

Case *T-A*:

Expression typing: The application of lemma 12 to the typing derivation of e yields that $R; M; \emptyset; \emptyset; \gamma; \gamma'$ holds. Thus, $R; M; \emptyset; \emptyset; \gamma'; \gamma'$ holds. The application of rule *T-U* to the latter fact yields that $R; M; \emptyset; \emptyset \vdash () : \langle\rangle\&(\gamma'; \gamma')$.

Store typing: The store preservation proof is as follows: Lemma 10 implies that the following hold: $R; M; \emptyset; \emptyset \vdash \texttt{loc}_\ell : \texttt{ref}(\tau, r)\&(\gamma; \gamma)$, where γ is equal to $\delta(n)$, $R; M; \emptyset; \emptyset \vdash v : \tau\&(\gamma; \gamma)$ and $\texttt{ref}(\tau, r) \simeq \texttt{ref}\ M(\ell)$ (this also implies that $\texttt{ref}(\tau, r)$ contains no type variables). Let $M(\ell)$ be equal to (τ', r') for some τ' and r', then we also have that $\tau \simeq \tau'$.

By applying lemma 34 to $\tau \simeq \tau'$ and $R; M; \emptyset; \emptyset \vdash v : \tau\&(\gamma; \gamma)$, we have that $R; M; \emptyset; \emptyset \vdash v : \tau'\&(\gamma; \gamma)$. The application of lemma 12 to the latter derivation implies $\vdash R; M; \emptyset; \emptyset; \gamma; \gamma$. Thus, $\vdash R; M; \emptyset; \emptyset; \emptyset; \emptyset$ also holds. The application of lemma 13 to the latter fact and $R; M; \emptyset; \emptyset \vdash v : \tau'\&(\gamma; \gamma)$ gives us $R; M; \emptyset; \emptyset \vdash v : \tau'\&(\emptyset; \emptyset)$.

The premise of the operational rule *E-AS* implies that if the input store is $\delta; S$, then the output store is $\delta; S[\bar{r} \mapsto S(\bar{r}), \ell \mapsto v]$. We have from the original store typing assumption that:

- $\vdash \delta$
- $R \vdash \delta$
- $R; M \vdash S$: $R \vdash S$ and $M \vdash S$

Thus, it suffices to show that $R; M \vdash S'$ holds. $R \vdash S'$ holds as $R \vdash S$ holds as no regions are added to S'. $M \vdash S'$ holds as $M \vdash S$ holds for all other locations than ℓ, and ℓ itself contains now the updated value v with typing derivation $R; M; \emptyset; \emptyset \vdash v : \tau'\&(\emptyset; \emptyset)$. Thus, $M \vdash S'$ holds.

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

By applying lemma 10 to the typing derivation of construct *new* we have that:

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

Let τ' be such that $\tau' \simeq \tau$ and τ' contains no region names of the form $\iota@n$. By applying lemma 34 to the latter fact we have that $R; M; \emptyset; \emptyset \vdash v : \tau'\&(\gamma;\gamma)$.

The application of lemma 30 to the latter typing derivation of v tells us that $R; \emptyset \vdash \tau'$ holds. The application of lemma 10 to the typing derivation of $\texttt{rgn}_{\bar{r}}$ gives us that $R; \emptyset \vdash \iota$. Thefore, $R; \emptyset \vdash \texttt{ref}(\tau', \iota)$ holds. By applying lemma 12 to the typing derivation of v we have that $\vdash R; M; \emptyset; \emptyset; \gamma; \gamma$. By inversion of the latter derivation $R \vdash M$ holds. Location ℓ is *fresh* so it does not belong to the domain of M. Consequently, we can combine the latter facts to derive that $R \vdash M, \ell \mapsto (\tau', \bar{r})$.

Expression typing: The latter derivation is substituted for $R \vdash M$ in the premises of $\vdash R; M; \emptyset; \emptyset; \gamma; \gamma$ to derive that $\vdash R; M, \ell \mapsto (\tau', \bar{r}); \emptyset; \emptyset; \gamma; \gamma$ holds. By applying rule *T-L* to the latter fact, $M, \ell \mapsto (\tau', \bar{r})$ and $\texttt{ref}(\tau, r) \simeq \texttt{ref}\ (\tau', \bar{r})$ we obtain that $R; M, \ell \mapsto (\tau', \bar{r}); \emptyset; \emptyset \vdash \texttt{loc}_\ell : \texttt{ref}(\tau, r)\&(\gamma;\gamma)$.

Store typing: By applying lemma 12 to the typing derivation of construct *new* we have that $\vdash R; M; \emptyset; \emptyset; \gamma; \gamma''$, where γ'' equals γ. Thus, $\vdash R; M; \emptyset; \emptyset; \emptyset; \emptyset$ also holds. By applying lemma 13 to the latter fact and $R; M; \emptyset; \emptyset \vdash v : \tau'\&(\gamma;\gamma)$ we have that $R; M; \emptyset; \emptyset \vdash v : \tau'\&(\emptyset;\emptyset)$ holds. By applying lemma 22 to the latter derivation $R \vdash M, \ell \mapsto (\tau', \bar{r})$ and $M \subseteq M, \ell \mapsto (\tau', \bar{r})$ we have that $R; M, \ell \mapsto (\tau', \bar{r}); \emptyset; \emptyset \vdash v : \tau'\&(\emptyset;\emptyset)$.

By inversion of the store typing assumption we have that $M \vdash S$ and $\forall(\ell' \mapsto (\tau'', j)) \in M.R; M; \emptyset; \emptyset \vdash S(j)(\ell') : \tau''\ \&\ (\emptyset;\emptyset)$. We must show that $R; M, \ell \mapsto (\tau', \bar{r}) \vdash \delta; S'$. It suffices to show that the following hold:

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

Case *T-AP*: The only operational rule that matches the shape of the application term is rule *E-A*:

- $\xi = \mathsf{seq}$
- $\mathsf{seq} \vdash \gamma = \gamma_1 \oplus \gamma_r$
- $\delta = \delta'', n \mapsto \sigma; \gamma$
- $\delta' = \delta'', n \mapsto \sigma; \gamma_r; \gamma_1$
- $\delta(n); S; ((\lambda x.\, e\ \texttt{as}\ \tau)\ v)^{\mathsf{seq}} \rightarrow \delta(n)'; S; \texttt{pop}_{\gamma_r}\ e[v/x]$

Expression typing: The proof for the typing preservation is similar to the previous proofs. By applying lemma 10 to the derivation of the application term we obtain the following premises:

- $R; M; \emptyset; \emptyset \vdash \lambda x.\, e_1\ \texttt{as}\ \tau_x : \tau_1' \xrightarrow{\gamma_1' \to \gamma_2'} \tau_2'\&(\gamma;\gamma)$: γ is equal to $\delta(n)$. by applying lemma 10 to this derivation we obtain that:
  - $\mathrm{ok}(\gamma_1;\gamma_2) \Rightarrow R; M; \emptyset; \emptyset, x : \tau_1 \vdash e_1 : \tau_2\&(\gamma_1;\gamma_2)$
  - $\tau_x \equiv \tau_1 \xrightarrow{\gamma_1 \to \gamma_2} \tau_2$.
  - $\tau' \simeq \tau$: by inversion of this fact we obtain that $\gamma_1 \simeq \gamma_1'$, $\gamma_2 \simeq \gamma_2'$, $\tau_1' \simeq \tau_1$ and $\tau_2' \simeq \tau_2$.
- $\mathsf{seq} \vdash \gamma'' = \gamma_2' \oplus (\gamma \ominus \gamma_1')$: by applying lemma 33 to $\mathsf{seq} \vdash \gamma'' = \gamma_2' \oplus (\gamma \ominus \gamma_1')$, $\gamma_1 \simeq \gamma_1'$ and $\gamma_2 \simeq \gamma_2'$, we have that $\mathsf{seq} \vdash \gamma'' = \gamma_2 \oplus (\gamma \ominus \gamma_1)$. By inversion of the latter derivation we obtain that $\mathrm{ok}(\gamma_1;\gamma_2)$ holds. By applying lemma 31 to $\mathsf{seq} \vdash \gamma = \gamma_1 \oplus \gamma_r$ and the latter fact we have that $\mathsf{seq} \vdash \gamma'' = \gamma_2 \oplus (\gamma_r \ominus \emptyset)$. $R; \emptyset \vdash \gamma_r$ holds as $R; \emptyset \vdash \gamma$ holds (premise of $\vdash R; M; \emptyset; \emptyset; \gamma; \gamma''$, which is immediate by lemma 12) and the regions of γ_r is a subset of the of regions γ (by $\mathsf{seq} \vdash \gamma = \gamma_1 \oplus \gamma_r$). $\vdash R; M; \emptyset; \emptyset; \gamma; \gamma''$ also tells us that $\mathrm{ok}(\gamma;\gamma'')$ holds. As mentioned earlier, γ_r is a subset of γ, thus $\mathrm{set}(\gamma_r;\emptyset)$ holds.
- $R; M; \emptyset; \emptyset \vdash v : \tau_1'\&(\gamma;\gamma)$: the application of lemma 12 to the typing derivation of the application term gives us that $\vdash R; M; \emptyset; \emptyset; \gamma; \gamma''$. Thus, $\vdash R; M; \emptyset; \emptyset; \emptyset; \emptyset$ also holds. We can use the latter fact and the derivation of value v along with lemma 13 to obtain $R; M; \emptyset; \emptyset \vdash v : \tau_1'\&(\emptyset;\emptyset)$. The application of lemma 34 to the latter derivation and $\tau_1' \simeq \tau_1$ gives us $R; M; \emptyset; \emptyset \vdash v : \tau_1\&(\emptyset;\emptyset)$.

We have shown that $\mathrm{ok}(\gamma_1;\gamma_2) \Rightarrow R; M; \emptyset; \emptyset, x : \tau_1 \vdash e_1 : \tau_2\&(\gamma_1;\gamma_2)$ and $\mathrm{ok}(\gamma_1;\gamma_2)$ holds, thus $R; M; \emptyset; \emptyset, x : \tau_1 \vdash e_1 : \tau_2\&(\gamma_1;\gamma_2)$ holds. Lemma 36 is applied to the typing derivation of v and e yields: $R; M; \emptyset; \emptyset \vdash e[v/x] : \tau_2\&(\gamma_1;\gamma_2)$.

The application of rule *T-E* to $\tau_2' \simeq \tau_2$, $\mathrm{set}(\gamma_r;\emptyset)$, $\mathsf{seq} \vdash \gamma'' = \gamma_2 \oplus (\gamma_r \ominus \emptyset)$, $R; \emptyset \vdash \gamma_r$ and $R; M; \emptyset; \emptyset \vdash e[v/x] : \tau_2\&(\gamma_1;\gamma_2)$. gives us $R; M; \emptyset; \emptyset \vdash \texttt{pop}_{\gamma_r}\ e[v/x] : \tau_2'\&(\gamma_1;\gamma'')$.

Store typing: The application of lemma 49 to the store typing assumption $(R; M \vdash \delta; S)$, $\mathsf{seq} \vdash \gamma = \gamma_1 \oplus \gamma_r$ and $\mathrm{ok}(\gamma;\gamma_1)$ (obtained by $\mathrm{ok}(\gamma_1;\gamma_2)$ and $\mathrm{ok}(\gamma;\gamma'')$ holds as shown earlier) implies that $R; M \vdash \delta'; S$ holds.

Lemma 30 (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 31 (Capability Addition Implication) $\xi \vdash \gamma = \gamma_1 \oplus \gamma_r \wedge \xi \vdash \gamma' = \gamma_2 \oplus (\gamma \ominus \gamma_1) \Rightarrow \xi \vdash \gamma' = \gamma_2 \oplus (\gamma_r \ominus \emptyset)$

Proof. By inversion of $\xi \vdash \gamma' = \gamma_2 \oplus (\gamma \ominus \gamma_1)$ we have that:

- $\xi \vdash \gamma = \gamma_1 \oplus \gamma_{r'}$: the capability addition rule *ES-C* is determistic, thus $\gamma_{r'} = \gamma_r$.
- $\xi \vdash \gamma' = \gamma_2 \oplus \gamma_{r'}$: similarly $\xi \vdash \gamma' = \gamma_2 \oplus \gamma_r$ holds. rule *ES-N* implies that $\xi \vdash \gamma_r = \emptyset \oplus \gamma_r$ holds.
- $\gamma'' = \text{live}(\gamma')$
- $\text{ok}(\gamma_1; \gamma_2)$: $\text{ok}(\gamma_2; \emptyset)$ trivially holds.
- $\xi = \mathsf{par} \Rightarrow \gamma_2 = \emptyset$

Rule *ESJ* is applied to the above facts to derive $\xi \vdash \gamma' = \gamma_2 \oplus (\gamma_r \ominus \emptyset)$.

Lemma 32 (Effect Addition Implication) $\xi \vdash \gamma' = \gamma'_1 \oplus \gamma_r \wedge \gamma_1 \simeq \gamma'_1 \Rightarrow \xi \vdash \gamma = \gamma_1 \oplus \gamma_r$

Proof. If γ_1 is empty then the conclusion holds by rule *ES-N*. Otherwise,rule *ES-C* applies and gives us the following facts:

- $\gamma = \gamma', r^{\kappa} \triangleright \pi$
- $\gamma_1 = \gamma'', r''^{\kappa_1} \triangleright \pi''$: $\gamma_1 \simeq \gamma'_1$ implies that $\gamma'_1 = \gamma''_1, r'''^{\kappa_1} \triangleright \pi'''$, $r''' \simeq r''$, $\pi''' \simeq \pi''$ and $\gamma'' \simeq \gamma''_1$.
- $\xi \vdash \gamma', r^{\kappa_2} \triangleright \pi = \gamma'' \oplus \gamma_r$: we can apply the induction hypothesis to this fact and $\gamma'' \simeq \gamma''_1$ to obtain $\xi \vdash \gamma', r^{\kappa_2} \triangleright \pi = \gamma''_1 \oplus \gamma_r$.
- $\xi \vdash \kappa = \kappa_1 + \kappa_2$
- $\pi \simeq \pi''$: we use the fact that $\pi''' \simeq \pi''$ to obtain $\pi \simeq \pi'''$.
- $r \simeq r''$: we use the fact $r''' \simeq r''$, and that to obtain $r''' \simeq r$.

Thus, $\xi \vdash \gamma = \gamma'_1 \oplus \gamma_r$ holds by applying rule *E-SC* to the above facts.

Lemma 33 (Effect Addition/Subtraction Implication) $\xi \vdash \gamma'' = \gamma'_2 \oplus (\gamma \ominus \gamma'_1) \wedge \gamma_1 \simeq \gamma'_1 \wedge \gamma_2 \simeq \gamma'_2 \Rightarrow \xi \vdash \gamma'' = \gamma_2 \oplus (\gamma \ominus \gamma_1)$

Proof. By inversion of the effect addition/subtraction assumption we have that

- $\xi \vdash \gamma = \gamma'_1 \oplus \gamma_r$: the application of lemma 32 to $\xi \vdash \gamma = \gamma'_1 \oplus \gamma_r$ and $\gamma_1 \simeq \gamma'_1$ implies that $\xi \vdash \gamma = \gamma_1 \oplus \gamma_r$.
- $\xi \vdash \gamma' = \gamma'_2 \oplus \gamma_r$: the application of lemma 32 to $\xi \vdash \gamma' = \gamma'_2 \oplus \gamma_r$ and $\gamma_2 \simeq \gamma'_2$ implies $\xi \vdash \gamma' = \gamma_2 \oplus \gamma_r$.
- $\gamma'' = \text{live}(\gamma')$
- $\xi = \mathsf{par} \Rightarrow \gamma_2 = \emptyset$
- $\text{ok}(\gamma'_1; \gamma'_2)$: $\text{ok}(\gamma'_1; \gamma'_2)$ trivially holds.

Lemma 34 (Value Type Implication) $R; M; \Delta; \Gamma \vdash v : \tau \& (\gamma; \gamma) \wedge \tau \simeq \tau' \Rightarrow R; M; \Delta; \Gamma \vdash v : \tau' \& (\gamma; \gamma)$

Proof. Trivial proof by case analysis on the shape of value v.

Lemma 35 (Polymorphic value substitution) $R, \bar{r}; \emptyset \vdash \gamma \wedge R, \bar{r}; M; \Delta, \rho; \emptyset \vdash f : \tau \& (\gamma; \gamma) \wedge \mathit{fresh}\ r \wedge r \simeq r' \Rightarrow R, \bar{r}; M; \Delta; \emptyset \vdash f[r/\rho] : \tau[r'/\rho] \& (\gamma; \gamma)$

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

Case $f \equiv \lambda x.\, e\ \mathsf{as}\ \tau'$: By inversion (lemma 10) of the assumption typing derivation we have that $\text{ok}(\gamma_1; \gamma_2) \Rightarrow R, \bar{r}; M; \Delta, \rho; \emptyset \vdash \lambda x.\, e\ \mathsf{as}\ \tau' : \tau \& (\gamma; \gamma)$ holds. If $\text{ok}(\gamma_1[r/\rho]; \emptyset)$ does not hold then the proof is immediate. Otherwise, the application of lemma 46 to the latter derivation, the fact that r is *fresh*, and $\text{ok}(\gamma_1[r/\rho]; \emptyset)$ gives us $R, \bar{r}; M; \Delta; \emptyset \vdash (\lambda x.\, e\ \mathsf{as}\ \tau')[r/\rho] : \tau[r/\rho] \& (\gamma[r/\rho]; \gamma[r/\rho])$. The assumption implies that γ is defined independently of ρ ($R, \bar{r}; \emptyset \vdash \gamma$). Thus, $R, \bar{r}; M; \Delta; \emptyset \vdash (\lambda x.\, e\ \mathsf{as}\ \tau')[r/\rho] : \tau[r/\rho] \& (\gamma; \gamma)$ also holds. By lemma 10 we obtain the premises of the latter derivation. We can use rule *T-F*, the premises and the fact that $\tau[r/\rho] \simeq \tau[r'/\rho]$ ($r \simeq r'$) to derive $R, \bar{r}; M; \Delta; \emptyset \vdash (\lambda x.\, e\ \mathsf{as}\ \tau')[r/\rho] : \tau[r'/\rho] \& (\gamma; \gamma)$.

Case $f \equiv \Lambda\rho'.\, f'$: By inversion (lemma 10) of the typing derivation of the assumption we have that $R, \bar{r}; M; \Delta, \rho, \rho'; \emptyset \vdash f' : \tau \& (\gamma; \gamma)$. We can use the induction hypothesis to derive that $R, \bar{r}; M; \Delta, \rho'; \emptyset \vdash f'[r/\rho] : \tau[r'/\rho] \& (\gamma; \gamma)$. The application of rule *T-RF* to the latter derivation yields $R, \bar{r}; M; \Delta; \emptyset \vdash \Lambda\rho'.\, f'[r/\rho] : \forall\rho'.\, \tau[r'/\rho] \& (\gamma; \gamma)$.

Lemma 36 (Variable Substitution) $R; M; \Delta; \Gamma, x : \tau_1 \vdash e : \tau_2 \,\&\, (\gamma_1;\gamma_2) \wedge R; M; \emptyset; \emptyset \vdash v : \tau_1 \,\&\, (\emptyset;\emptyset) \Rightarrow R; M; \Delta; \Gamma \vdash e[v/x] : \tau_2 \,\&\, (\gamma_1;\gamma_2)$

Proof. Straightforward induction on the expression typing derivation.

Lemma 37 (Region substitution preserves *ok*) $\xi \vdash \gamma_3 = \gamma_2 \oplus \gamma_1 \wedge \mathrm{ok}(\gamma_3; \emptyset) \wedge \mathrm{ok}(\gamma_2; \emptyset) \wedge \mathrm{ok}(\gamma_3[r_x/\rho]; \emptyset) \Rightarrow \mathrm{ok}(\gamma_2[r_x/\rho]; \emptyset)$

Proof. If ρ does not exist in γ_2 then the proof is immediate by the assumption that $\mathrm{ok}(\gamma_2; \emptyset)$. Otherwise we assume that ρ must exist in both γ_2 and γ_3 (by $\xi \vdash \gamma_3 = \gamma_2 \oplus \gamma_1$ we have that the domain of γ_2 is a subset of the domain of γ_3). Assume that r_x belongs in the domain of γ_3. This is a contradiction as the assumption $\mathrm{ok}(\gamma_3[r_x/\rho]; \emptyset)$ does not hold. We have mentioned that the regions of γ_2 are a subset of the regions of γ_3. Therefore, r_x does not belong in the domain of γ_2 either. By the assumption that $\mathrm{ok}(\gamma_2; \emptyset)$ holds, the definition of predicate *ok* and the fact that r_x does not occur in the domain of γ_2 implies that $\mathrm{ok}(\gamma_2[r_x/\rho]; \emptyset)$ holds.

Lemma 38 (Region substitution preserves $\oplus$) $\xi \vdash \gamma_3 = \gamma_2 \oplus \gamma_1 \Rightarrow \xi[r/\rho] \vdash \gamma_3[r/\rho] = \gamma_2[r/\rho] \oplus \gamma_1[r/\rho]$

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

- $\pi \simeq \pi'$: $\pi[r/\rho] \simeq \pi'[r/\rho]$ is immediate.
- $r' \simeq r$: $r'[r/\rho] \simeq r[r/\rho]$ is immediate.
- $\xi \vdash \gamma_{31}, r'^{\kappa_2} \triangleright \pi = \gamma_{12} \oplus \gamma_1$: $\xi[r/\rho] \vdash (\gamma_{31}, r'^{\kappa_2} \triangleright \pi)[r/\rho] = \gamma_{12}[r/\rho] \oplus \gamma_1[r/\rho]$ holds by the induction hypothesis.
- $\gamma_3 = \gamma_{31}, r'^{\kappa_2} \triangleright \pi'$: $\gamma_3[r/\rho] = (\gamma_{31}, r'^{\kappa_2} \triangleright \pi')[r/\rho]$ is immediate.
- $\gamma_1 = \gamma_{12}, r^{\kappa_1} \triangleright \pi$: $\gamma_1[r/\rho] = (\gamma_{12}, r^{\kappa_1} \triangleright \pi)[r/\rho]$ is immediate.

By using rule *ES-C* we obtain that: $\xi[r/\rho] \vdash \gamma_3[r/\rho] = \gamma_2[r/\rho] \oplus \gamma_1[r/\rho]$

Lemma 39 (Valid implication — $\oplus$) $\xi \vdash \gamma = \gamma_1 \oplus \gamma_r \wedge \mathrm{valid}(\gamma_a; \gamma_b) \wedge \gamma_1 \subseteq \gamma_a \wedge \gamma_2 \subseteq \gamma_b \wedge \mathrm{dom}(\gamma_2) \subseteq \mathrm{dom}(\gamma_1) \Rightarrow \xi \vdash \gamma = \gamma_2 \oplus \gamma'_r$

Proof. Proof by induction on the structure of γ_2:

- $\emptyset$: immediate by rule *ES-N*.
- $\gamma_2 = \gamma_{21}, r^{\kappa_3} \triangleright \pi$: $\mathrm{valid}(\gamma_a; \gamma_b)$, $\gamma_1 \subseteq \gamma_a$, $\gamma_2 \subseteq \gamma_b$ and $\mathrm{dom}(\gamma_2) \subseteq \mathrm{dom}(\gamma_1)$ imply that $\gamma_1 = \gamma_{12}, r^{\kappa_1} \triangleright \pi$ and $\mathrm{is\_pure}(\kappa_1) \Leftrightarrow \mathrm{is\_pure}(\kappa_3)$. By inversion of the assumption $\xi \vdash \gamma = \gamma_1 \oplus \gamma_r$ we have that:
  - $\gamma_3 = \gamma_{31}, r'^{\kappa_2} \triangleright \pi'$.
  - $\pi \simeq \pi'$ and $r' \simeq r$.
  - $\xi \vdash \kappa = \kappa_1 + \kappa_2$: $\xi \vdash \kappa = \kappa_3 + \kappa'_2$ also holds for some κ'_2 as a result of $\mathrm{is\_pure}(\kappa_1) \Leftrightarrow \mathrm{is\_pure}(\kappa_3)$.
  - $\xi \vdash \gamma_{31}, r'^{\kappa_2} \triangleright \pi = \gamma_{12} \oplus \gamma'_r$: $\xi \vdash \gamma_{31}, r'^{\kappa_2} \triangleright \pi = \gamma_{21} \oplus \gamma''_r$ holds by induction hypothesis.

  By applying rule *ES-C* to the latter facts we obtain that $\xi \vdash \gamma = \gamma_2 \oplus \gamma''_r$.

Lemma 40 (Region substitution preserves $\oplus/\ominus$) $\xi \vdash \gamma_3 = \gamma_2 \oplus (\gamma \ominus \gamma_1) \wedge \mathrm{ok}(\gamma; \gamma_3) \wedge \mathrm{ok}(\gamma[r/\rho]; \emptyset) \wedge \mathrm{valid}(\gamma_1; \gamma_2) \wedge \mathit{fresh}\ r \Rightarrow \xi[r/\rho] \vdash \gamma_3[r/\rho] = \gamma_2[r/\rho] \oplus (\gamma[r/\rho] \ominus \gamma_1[r/\rho])$

Proof. The assumption that $\mathrm{ok}(\gamma; \gamma_3)$ holds implies that $\mathrm{ok}(\gamma; \emptyset)$ and $\mathrm{ok}(\gamma_3; \emptyset)$ hold. By inversion of the first assumption we obtain the following facts:

- $\mathrm{ok}(\gamma_1; \gamma_2)$: this fact implies that $\mathrm{ok}(\gamma_1; \emptyset)$ and $\mathrm{ok}(\gamma_2; \emptyset)$ hold. The application of lemma 37 to $\xi \vdash \gamma = \gamma_1 \oplus \gamma_r$, $\mathrm{ok}(\gamma_1; \emptyset)$, $\mathrm{ok}(\gamma; \emptyset)$ and $\mathrm{ok}(\gamma[r/\rho]; \emptyset)$ implies that $\mathrm{ok}(\gamma_1[r/\rho]; \emptyset)$ holds. The application of lemma 39 to $\mathrm{valid}(\gamma_1; \gamma_2)$, $\gamma_1 \subseteq \gamma_1$, $\gamma_2 \subseteq \gamma_2$ and $\mathrm{dom}(\gamma_2) \subseteq \mathrm{dom}(\gamma_1)$ (by inversion of $\mathrm{valid}(\gamma_1; \gamma_2)$), we have that $\xi \vdash \gamma = \gamma_2 \oplus \gamma'_r$, for some γ'_r. The application of lemma 37 to $\xi \vdash \gamma = \gamma_2 \oplus \gamma'_r$, $\mathrm{ok}(\gamma_2; \emptyset)$, $\mathrm{ok}(\gamma; \emptyset)$ and $\mathrm{ok}(\gamma[r/\rho]; \emptyset)$ implies that $\mathrm{ok}(\gamma_2[r/\rho]; \emptyset)$ holds. Thus, $\mathrm{ok}(\gamma_1[r/\rho]; \gamma_2[r/\rho])$ holds.

- $\xi \vdash \gamma = \gamma_1 \oplus \gamma_r$: $\xi[r/\rho] \vdash \gamma[r/\rho] = \gamma_1[r/\rho] \oplus \gamma_r[r/\rho]$ immediate by the application of lemma 38 to $\xi \vdash \gamma = \gamma_1 \oplus \gamma_r$.
- $\xi \vdash \gamma_a = \gamma_2 \oplus \gamma_r$: $\xi[r/\rho] \vdash \gamma_a[r/\rho] = \gamma_2[r/\rho] \oplus \gamma_r[r/\rho]$ immediate by the application of lemma 38 to $\vdash \gamma_a = \gamma_2 \oplus \gamma_r$.
- $\gamma_3 = \text{live}(\gamma_a)$: it suffices to prove that $(\text{live}(\gamma_a))[r/\rho] = \text{live}(\gamma_a[r/\rho])$. This is trivial to show given that r is fresh (i.e. it does not belong in the domain of γ_a).
- $\xi = \mathsf{par} \Rightarrow \gamma_2 = \emptyset$: $\xi[r/\rho] = \mathsf{par} \Rightarrow \gamma_2[r/\rho] = \emptyset$ trivially holds.

Lemma 41 (R Well-Formedness Substitution) $R, \bar{r'}; \Delta, \rho \vdash r \Rightarrow R, \bar{r'}; \Delta \vdash r[r'/\rho]$

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

- $\iota@n$: By inversion of this derivation we have that $R, \bar{r'}; \Delta, \rho \vdash \iota$. The proof is completed by applying the induction hypothesis.
- $r \neq \iota@n$: By inversion of this derivation we have that $\bar{r} \in R, \bar{r'} \uplus \Delta, \rho$. Thus, $\bar{r}[r'/\rho] \in R, \bar{r'} \uplus \Delta$ also holds as $\bar{r}[r'/\rho]$ cannot be contained in Δ. Therefore, $R, \bar{r'}; \Delta \vdash r[r'/\rho]$ holds.

Lemma 42 (Effect Well-formedness Substitution) $R, \bar{r''}; \Delta, \rho \vdash \gamma \Rightarrow R, \bar{r''}; \Delta \vdash \gamma[r''/\rho]$

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

- $\emptyset$: $R, \bar{r''}; \Delta \vdash \emptyset$ trivially holds.
  $R, \bar{r''}; \Delta, \rho \vdash \gamma', r^{\kappa} \triangleright \pi$: $R, \bar{r''}; \Delta \vdash \gamma'[r''/\rho]$ holds by the induction hypothesis. $R, \bar{r''}; \Delta \vdash r[r''/\rho]$ holds by lemma 41. If $\pi = r'$, then $R, \bar{r''}; \Delta \vdash r'[r''/\rho]$ holds by lemma 41.

Lemma 43 (Type Context Well-formedness Substitution) $R, \bar{r'}; \Delta, \rho \vdash \tau \wedge \mathit{fresh}\ r' \Rightarrow R, \bar{r'}; \Delta \vdash \tau[r'/\rho]$

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

- b: $R, \bar{r'}; \Delta \vdash b$ trivially holds.
- $\langle\rangle$: $R, \bar{r'}; \Delta \vdash \langle\rangle$ trivially holds.
- $\texttt{rgn}(r)$: $R, \bar{r'}; \Delta \vdash r[r'/\rho]$ holds by lemma 41.
- $\texttt{ref}(\tau', r)$: $R, \bar{r'}; \Delta \vdash r[r'/\rho]$ holds by lemma 41. $R, \bar{r'}; \Delta \vdash \tau'[r'/\rho]$ holds by the induction hypothesis.
- $\forall \rho'.\, \tau'$: $R, \bar{r'}; \Delta, \rho' \vdash \tau'[r'/\rho]$ holds by the induction hypothesis.
- $\tau' \xrightarrow{\gamma_1 \to \gamma_2} \tau''$: $R, \bar{r'}; \Delta \vdash \tau'[r'/\rho]$ holds by the induction hypothesis. $R, \bar{r'}; \Delta \vdash \tau''[r'/\rho]$ holds by the induction hypothesis. $R, \bar{r'}; \Delta \vdash \gamma_1[r'/\rho]$ holds by lemma 42. $R, \bar{r'}; \Delta \vdash \gamma_2[r'/\rho]$ holds by lemma 42. We have that $\text{valid}(\gamma_1; \gamma_2)$ and we must prove that $\text{valid}(\gamma_1[r'/\rho]; \gamma_2[r'/\rho])$ holds. It suffices to show that:
  - if $(r[r'/\rho]^{\kappa} \triangleright \pi[r'/\rho]) \in \gamma_1[r'/\rho]$ and $(r[r'/\rho]^{\kappa'} \triangleright \pi'[r'/\rho]) \in \gamma_2[r'/\rho]$ for some r, then $\pi = \pi' \wedge (\text{is\_pure}(\kappa) \Leftrightarrow \text{is\_pure}(\kappa'))$: this is immediate by $(r^{\kappa} \triangleright \pi) \in \gamma_1$ and $(r^{\kappa'} \triangleright \pi') \in \gamma_2$, then $\pi = \pi' \wedge (\text{is\_pure}(\kappa) \Leftrightarrow \text{is\_pure}(\kappa'))$, which can be obtained by inversion of $\text{valid}(\gamma_1; \gamma_2)$.
  - $\text{live}(\gamma_1[r'/\rho]) = \gamma_1[r'/\rho]$ and $\text{live}(\gamma_2[r'/\rho]) = \gamma_2[r'/\rho]$: immediate by inversion of $\text{valid}(\gamma_1; \gamma_2)$, the definition of substitution and the fact that r' is *fresh*.
  - $\text{dom}(\gamma_2) \subseteq \text{dom}(\gamma_1)$: $\text{dom}(\gamma_2[r'/\rho]) \subseteq \text{dom}(\gamma_1[r'/\rho])$ is immediate.

Lemma 44 (Variable Context Well-formedness Substitution) $R, \bar{r}; \Delta, \rho \vdash \Gamma \wedge \mathit{fresh}\ r \Rightarrow R, \bar{r}; \Delta \vdash \Gamma[r/\rho]$

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

- $\emptyset$: $R'; \Delta \vdash \emptyset$ trivially holds.
  $R; \Delta \vdash \Gamma', x : \tau$: $R'; \Delta \vdash \Gamma'[r/\rho]$ holds by the induction hypothesis. $R'; \Delta \vdash \tau[r/\rho]$ holds by lemma 43 and the fact that r is *fresh*.

Lemma 45 (Well-formedness Substitution) $R, \bar{r}; M; \Delta, \rho; \Gamma; \gamma_1; \gamma_2 \wedge \mathit{fresh}\ r \wedge \mathrm{ok}(\gamma_1[r/\rho]; \gamma_2[r/\rho]) \Rightarrow R, \bar{r}; M; \Delta; \Gamma[r/\rho]; \gamma_1[r/\rho]; \gamma_2[r/\rho]$

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

- $R, \bar{r} \vdash M$
- $R, \bar{r}; \Delta, \rho \vdash \Gamma$: $R, \bar{r}; \Delta \vdash \Gamma[r/\rho]$ immediate by lemma 44 and the fact that r is *fresh*.
- $R, \bar{r}; \Delta, \rho \vdash \gamma_1$: $R, \bar{r}; \Delta \vdash \gamma_1[r/\rho]$ immediate by lemma 42.
- $R, \bar{r}; \Delta, \rho \vdash \gamma_2$: $R, \bar{r}; \Delta \vdash \gamma_2[r/\rho]$ immediate by lemma 42.
- $\mathrm{ok}(\gamma_1; \gamma_2)$: $\mathrm{ok}(\gamma_1[r/\rho]; \gamma_2[r/\rho])$ immediate from the assumption.

Lemma 46 (Region Substitution) $R, \bar{r}; M; \Delta, \rho; \Gamma \vdash e : \tau \,\&\, (\gamma_1;\gamma_2) \wedge \mathit{fresh}\ r\ \wedge \mathrm{ok}(\gamma_1[r/\rho]; \emptyset) \Rightarrow\ R, \bar{r}; M; \Delta; \Gamma[r/\rho] \vdash e[r/\rho] : \tau[r/\rho] \,\&\, (\gamma_1[r/\rho];\gamma_2[r/\rho])$

Proof. Proof by induction on the expression typing derivation.

Case *T-I*: by applying lemma 10 to the derivation of e we have that $\gamma_1 = \gamma_2$. Thus, $\mathrm{ok}(\gamma_2[r/\rho]; \emptyset)$ is immediate from $\gamma_1 = \gamma_2$ and the assumption that $\mathrm{ok}(\gamma_1[r/\rho]; \emptyset)$ holds. The application of lemma 45 to the latter derivation, the fact that r is *fresh* and $\mathrm{ok}(\gamma_1[r/\rho]; \gamma_2[r/\rho])$ implies that $\vdash R, \bar{r}; M; \Delta; \Gamma[r/\rho];\ \gamma_1[r/\rho]; \gamma_2[r/\rho]$ holds. The proof for this case is completed by applying rule *T-I*.

Case *T-U*, by applying lemma 10 to the derivation of e we have that $\gamma_1 = \gamma_2$. Thus, $\mathrm{ok}(\gamma_2[r/\rho]; \emptyset)$ is immediate from $\gamma_1 = \gamma_2$ and the assumption that $\mathrm{ok}(\gamma_1[r/\rho]; \emptyset)$ holds. The application of lemma 45 to the latter derivation, the fact that r is *fresh* and $\mathrm{ok}(\gamma_1[r/\rho]; \gamma_2[r/\rho])$ implies that $\vdash R, \bar{r}; M; \Delta; \Gamma[r/\rho];\ \gamma_1[r/\rho]; \gamma_2[r/\rho]$ holds. The proof for this case is completed by applying rule *T-U*.

Case *T-R*: the application of lemma 10 to the derivation of e yields:

- $\gamma_1 = \gamma_2$: $\mathrm{ok}(\gamma_2[r/\rho]; \emptyset)$ is immediate from $\gamma_1 = \gamma_2$ and the assumption that $\mathrm{ok}(\gamma_1[r/\rho]; \emptyset)$ holds. the application of lemma 45 to the latter derivation, the fact that r is *fresh* and $\mathrm{ok}(\gamma_1[r/\rho]; \gamma_2[r/\rho])$ implies that $\vdash R, \bar{r}; M; \Delta; \Gamma[r/\rho]; \gamma_1[r/\rho]; \gamma_2[r/\rho]$ holds.
- $\vdash R, \bar{r}; M; \Delta, \rho, \Gamma; \gamma; \gamma$: we have already shown that $\vdash R, \bar{r}; M; \Delta; \Gamma[r/\rho]; \gamma[r/\rho]; \gamma[r/\rho]$ holds.
- $R; \Delta \vdash \iota$: $R, \bar{r}; \Delta \vdash \iota[r/\rho]$ holds by lemma 41.
- $r' \simeq \iota$: $r'[r/\rho] \simeq \iota[r/\rho]$ trivially holds.

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

Case *T-L*: the application of lemma 10 to the derivation of e yields:

- $\gamma_1 = \gamma_2$: $\mathrm{ok}(\gamma_2[r/\rho]; \emptyset)$ is immediate from $\gamma_1 = \gamma_2$ and the assumption that $\mathrm{ok}(\gamma_1[r/\rho]; \emptyset)$ holds. The application of lemma 45 to the latter derivation, the fact that r is *fresh* and $\mathrm{ok}(\gamma_1[r/\rho]; \gamma_2[r/\rho])$ implies that $\vdash R, \bar{r}; M; \Delta; \Gamma[r/\rho]; \gamma_1[r/\rho]; \gamma_2[r/\rho]$ holds.
- $\vdash R, \bar{r}; M; \Delta, \rho, \Gamma; \gamma; \gamma$: we have already shown that $\vdash R, \bar{r}; M; \Delta; \Gamma[r/\rho]; \gamma[r/\rho]; \gamma[r/\rho]$ holds.
- $(\ell \mapsto (\tau, \iota)) \in M$
- $\tau' \simeq \mathtt{ref}(\tau, \iota)$: $\tau'[r/\rho] \simeq \mathtt{ref}(\tau, \iota)[r/\rho]$ trivially holds.

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

Case *T-V*: the application of lemma 10 to the derivation of e yields:

- $\gamma_1 = \gamma_2$: $\mathrm{ok}(\gamma_2[r/\rho]; \emptyset)$ is immediate from $\gamma_1 = \gamma_2$ and the assumption that $\mathrm{ok}(\gamma_1[r/\rho]; \emptyset)$ holds. The application of lemma 45 to the latter derivation, the fact that r is *fresh* and $\mathrm{ok}(\gamma_1[r/\rho]; \gamma_2[r/\rho])$ implies that $\vdash R, \bar{r}; M; \Delta; \Gamma[r/\rho]; \gamma_1[r/\rho]; \gamma_2[r/\rho]$ holds.
- $\vdash R, \bar{r}; M; \Delta, \rho, \Gamma; \gamma; \gamma$: we have already shown that $\vdash R, \bar{r}; M; \Delta; \Gamma[r/\rho]; \gamma[r/\rho]; \gamma[r/\rho]$ holds.
- $(x : \tau) \in \Gamma$: $(x : \tau[r/\rho]) \in \Gamma[r/\rho]$ trivially holds.
- $\tau \simeq \tau'$: $\tau[r/\rho] \simeq \tau'[r/\rho]$ trivially holds.

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

Case *T-F*: the application of lemma 10 to the derivation of e yields:

- $\gamma_1 = \gamma_2$: $\text{ok}(\gamma_2[r/\rho]; \emptyset)$ is immediate from $\gamma_1 = \gamma_2$ and the assumption that $\text{ok}(\gamma_1[r/\rho]; \emptyset)$ holds. The application of lemma 45 to the latter derivation, the fact that r is *fresh* and $\text{ok}(\gamma_1[r/\rho]; \gamma_2[r/\rho])$ implies that $\vdash R, \bar{r}; M; \Delta; \Gamma[r/\rho]; \gamma_1[r/\rho]; \gamma_2[r/\rho]$ holds.
- $\vdash R, \bar{r}; M; \Delta, \rho, \Gamma; \gamma; \gamma$: we have already shown that $\vdash R, \bar{r}; M; \Delta; \Gamma[r/\rho]; \gamma[r/\rho]; \gamma[r/\rho]$ holds.
- $R, \bar{r}; \Delta, \rho \vdash \tau$: lemma 43 and the fact that r is *fresh* imply that $R, \bar{r}; \Delta \vdash \tau[r/\rho]$ holds.
- $\tau' \simeq \tau$: $\tau'[r/\rho] \simeq \tau[r/\rho]$ trivially holds.
- $\tau \equiv \tau_1 \xrightarrow{\gamma_a \to \gamma_b} \tau_2$: the function type after substitution is $\tau[r/\rho] \equiv \tau_1[r/\rho] \xrightarrow{\gamma_a[r/\rho] \to \gamma_b[r/\rho]} \tau_2[r/\rho]$.
- $\text{set}(\gamma_a; \gamma_b)$: $\text{set}(\gamma_a[r/\rho], \gamma_b[r/\rho])$ trivially holds as r is *fresh*.
- $\text{ok}(\gamma_a; \gamma_b) \Rightarrow R, \bar{r}; M; \Delta, \rho; \Gamma, x : \tau_1 \vdash e : \tau_2 \,\&\, (\gamma_a; \gamma_b)$: Let us assume that $\text{ok}(\gamma_a[r/\rho]; \gamma_b[r/\rho])$ holds, then, $R, \bar{r}; M; \Delta; (\Gamma, x : \tau_1)[r/\rho] \vdash e[r/\rho] : \tau_2[r/\rho] \,\&\, (\gamma_a[r/\rho]; \gamma_b[r/\rho])$ holds by the induction hypothesis.

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

Case *T-AP*: the application of lemma 10 to the derivation of e yields:

- $R, \bar{r}; M; \Delta, \rho; \Gamma \vdash e_1 : \tau_1 \xrightarrow{\gamma_a \to \gamma_b} \tau_2 \,\&\, (\gamma_1; \gamma_3)$: By applying lemma 30 to the derivation of e_1 we obtain that $R, \bar{r}; \Delta, \rho \vdash \tau_1 \xrightarrow{\gamma_a \to \gamma_b} \tau_2$. By inversion of the latter fact $\text{valid}(\gamma_a; \gamma_b)$ holds.

  $R, \bar{r}; M; \Delta; \Gamma[r/\rho] \vdash e_1[r/\rho] : (\tau_1 \xrightarrow{\gamma_a \to \gamma_b} \tau_2)[r/\rho] \,\&\, (\gamma_1[r/\rho]; \gamma_3[r/\rho])$ holds by the induction hypothesis, the assumption that r is *fresh* and $\text{ok}(\gamma_1[r/\rho]; \emptyset)$. By applying lemma 12 to the latter fact and performing inversion to the resulting well-formedness derivation we have that $\text{ok}(\gamma_3[r/\rho]; \emptyset)$.
- $\mathsf{par} \Rightarrow \tau_2 = \langle\rangle$: $\mathsf{par} \Rightarrow \tau_2[r/\rho] = \langle\rangle$ trivially holds.
- $R, \bar{r}; M; \Delta, \rho; \Gamma \vdash e_2 : \tau_1 \,\&\, (\gamma_3; \gamma_4)$: $R, \bar{r}; M; \Delta; \Gamma[r/\rho] \vdash e_2[r/\rho] : \tau_1[r/\rho] \,\&\, (\gamma_3[r/\rho]; \gamma_4[r/\rho])$ holds by the induction hypothesis and $\text{ok}(\gamma_3[r/\rho]; \emptyset)$. By applying lemma 12 to the latter fact and performing inversion to the resulting well-formedness derivation we have that $\text{ok}(\gamma_4[r/\rho]; \emptyset)$.
- $\xi \vdash \gamma_2 = \gamma_b \oplus (\gamma_4 \ominus \gamma_a)$: we have shown that $\text{ok}(\gamma_4[r/\rho]; \emptyset)$ and $\text{valid}(\gamma_a; \gamma_b)$. $\xi \vdash \gamma_2[r/\rho] = \gamma_b[r/\rho] \oplus (\gamma_4[r/\rho] \ominus \gamma_a[r/\rho])$ is immediate by lemma 40, $\text{ok}(\gamma_4[r/\rho]; \emptyset)$, $\text{valid}(\gamma_a; \gamma_b)$ and $\text{ok}(\gamma_4; \gamma_2)$, which can be obtained by applying lemma 12 to the typing derivation of e, and the fact that r is *fresh*.

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

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

Lemma 47 (Store typing preservation — Push Helper 2) $\sigma; \gamma \vdash \delta \wedge \text{ok}(\gamma; \gamma_1) \wedge \mathsf{seq} \vdash \gamma = \gamma_1 \oplus \gamma_r \Rightarrow \sigma; \gamma_r; \gamma_1 \vdash \delta$

Proof. Proof by induction on the shape of δ:

- $\emptyset$: it must be shown that for all ι that belong in the domain of $\text{dom}(\sigma; \gamma_r; \gamma_1)$, an given that $\sigma; \gamma_r; \gamma_1 \simeq \sigma_1; \gamma_x, \iota^\kappa \triangleright \pi + \sigma_2$, $\text{rg}(\kappa) > 0$ and $\text{is\_pure}(\kappa)$ hold, then both $\text{zero\_pure}(\sigma_1)$ and $\iota \notin \text{dom}(\sigma_2; \gamma_x)$ hold. We proceed by performing a case analysis as follows:

  - ι does belong in the domain of $\emptyset; \gamma$: stack $\sigma_1 \simeq \sigma$ and $\sigma_2 = \emptyset$. By inversion of $\sigma; \gamma \vdash \emptyset$ (obtained by $\sigma; \gamma \vdash \delta$) we have that $\text{zero\_pure}(\sigma)$ (or $\text{zero\_pure}(\sigma_1)$) and $\iota \notin \text{dom}(\emptyset; \gamma_x)$. The assumption that $\text{ok}(\gamma; \gamma_1)$ holds implies that ι, which is associated with a pure capability, belongs in the domain of either $\emptyset; \gamma_r$ or $\emptyset; \gamma_1$ with a pure and positive capability. If it does belong in the domain of $\emptyset; \gamma_r$, then $\gamma_r \simeq \gamma_x, \iota^\kappa \triangleright \pi$ and $\iota \notin \text{dom}(\emptyset; \gamma_1; \gamma_x)$ trivially hold from the above fact, the assumption that $\text{ok}(\gamma; \gamma_1)$ and $\mathsf{seq} \vdash \gamma = \gamma_1 \oplus \gamma_r$. Otherwise, $\gamma_1 \simeq \gamma_x, \iota^\kappa \triangleright \pi$, $\text{zero\_pure}(\sigma; \gamma_r)$ and $\iota \notin \text{dom}(\emptyset; \gamma_x)$ trivially hold from the above facts and the assumption that $\text{ok}(\gamma; \gamma_1)$.
  - ι does not belong in the domain of $\emptyset; \gamma$: stack $\sigma_2 \simeq \sigma_2'; \gamma_r; \gamma_1$ for some stack σ_2' (the assumption that $\mathsf{seq} \vdash \gamma = \gamma_1 \oplus \gamma_r$ holds implies that $\text{dom}(\emptyset; \gamma_r) \subseteq \text{dom}(\emptyset; \gamma)$ and $\text{dom}(\emptyset; \gamma_1) \subseteq \text{dom}(\emptyset; \gamma)$). By inversion of $\sigma; \gamma \vdash \emptyset$ (obtained by $\sigma; \gamma \vdash \delta$) we have that $\text{zero\_pure}(\sigma_1)$ and $\iota \notin \text{dom}(\sigma_2)$. (or $\iota \notin \text{dom}(\sigma_2'; \gamma; \gamma_x)$). The latter fact implies that $\iota \notin \text{dom}(\sigma_2'; \gamma_r; \gamma_1; \gamma_x)$.

Case $\delta_1, n_1 \mapsto \sigma_1$: it suffices to show that

- $\sigma; \gamma_r; \gamma_1 \vdash \delta_1$: by inversion of $\sigma; \gamma \vdash \delta$ we obtain $\sigma; \gamma \vdash \delta_1$. The proof for this case is completed by applying the induction hypothesis.

– Given that is_accessible($\sigma; \gamma_r; \gamma_1, \iota$), then prove that ¬is_accessible(σ_1, ι) for all regions ι that belong in the domain of $\sigma; \gamma_r; \gamma_1$: the capability addition assumption implies that is_accessible($\sigma; \gamma_r; \gamma_1, \iota$) implies is_accessible($\sigma; \gamma, \iota$). By inversion of $\sigma; \gamma \vdash \delta$ and the latter fact, and the fact that ι belongs in the domain of $\emptyset; \gamma$ (capability addition assumption), we obtain that is_accessible($\sigma; \gamma, \iota$) and thus ¬is_accessible(σ_1, ι).

Lemma 48 (Store typing preservation — Push Helper 1) $\vdash \delta, n \mapsto \sigma; \gamma \wedge \text{ok}(\gamma; \gamma_1) \wedge \mathsf{seq} \vdash \gamma = \gamma_1 \oplus \gamma_r \Rightarrow \vdash \delta, n \mapsto \sigma; \gamma_r; \gamma_1$

Proof. It suffices to prove that:

- $\vdash \delta$: immediate by inversion inversion of the assumption $\vdash \delta, n \mapsto \sigma; \gamma$.
- $\sigma; \gamma_r; \gamma_1 \vdash \delta$: by inversion of the assumption $\vdash \delta, n \mapsto \sigma; \gamma$ we have that $\sigma; \gamma \vdash \delta$. The application of lemma 47 completes the proof for this case.

Lemma 49 (Store typing preservation — Push) $R; M \vdash \delta; S \wedge \text{ok}(\gamma; \gamma_1) \wedge \mathsf{seq} \vdash \gamma = \gamma_1 \oplus \gamma_r \wedge \delta = \delta'', n \mapsto \sigma; \gamma \Rightarrow \delta' = \delta'', n \mapsto \sigma; \gamma_r; \gamma_1 \wedge R; M \vdash \delta'; S$

Proof. The store typing assumption implies that the following hold:

- $R; M \vdash S$
- $R \vdash \delta$: immediate by the fact that the regions of γ_1 and γ_r are a subset of γ (by the effect addition assumption).
- $\vdash \delta$: by inversion of $R; M \vdash \delta; S$ we have that $\vdash \delta'', n \mapsto \sigma; \gamma$. The proof is immediate by the application of lemma 48 to the latter fact, ok($\gamma; \gamma_1$) and $\mathsf{seq} \vdash \gamma = \gamma_1 \oplus \gamma_r$.

Lemma 50 (Store typing preservation — Pop Helper 2) $\sigma; \gamma_r; \gamma_1 \vdash \delta \wedge \text{ok}(\gamma_r; \gamma_1) \wedge \mathsf{seq} \vdash \gamma = \gamma_1 \oplus (\gamma_r \ominus \emptyset) \Rightarrow \sigma; \gamma \vdash \delta$

Proof. Proof by induction on the shape of δ:

- $\emptyset$: it must be shown that for all ι that belong in the domain of dom($\sigma; \gamma$), an given that $\sigma; \gamma \simeq \sigma_1; \gamma_x, \iota^{\kappa} \triangleright \pi + \sigma_2$, rg($\kappa$) > 0 and is_pure($\kappa$) hold, then both zero_pure(σ_1) and $\iota \notin$ dom($\sigma_2; \gamma_x$) hold. By inversion of $\sigma; \gamma_r; \gamma_1 \vdash \emptyset$ (obtained by the assumption $\sigma; \gamma_r; \gamma_1 \vdash \delta$) we have that $\sigma; \gamma_r; \gamma_1 \simeq \sigma_3; \gamma_y, \iota^{\kappa'} \triangleright \pi + \sigma_4$, zero_pure($\sigma_3$) and $\iota \notin$ dom($\sigma_2; \gamma_y$).
  Region ι cannot be contained as a positive and pure effect in both γ_r and γ_1 as the effect addition assumption would not hold. We proceed by performing a case analysis as follows:
  - ι does belong in the domain of γ_1: $\sigma_1 \simeq \sigma_3 \simeq \sigma; \gamma_r$, $\sigma_2 = \sigma_4 = \emptyset$, and $\gamma_1 \simeq \gamma_y, \iota^{\kappa'} \triangleright \pi$. Thus, we have that zero_pure(σ_3, ι) (or zero_pure(σ, ι)) and $\iota \notin$ dom($\emptyset; \gamma_y$). The effect addition assumption and the assumption that ok($\gamma_r; \gamma_1$) imply that ok(γ) holds. Thus, $\iota \notin$ dom($\emptyset; \gamma_x$).
  - ι does belong in the domain of γ_r: $\sigma_1 \simeq \sigma_3 \simeq \sigma$, $\sigma_2 \simeq \emptyset$, $\sigma_4 \simeq \emptyset; \gamma_1$, and $\gamma_r \simeq \gamma_y, \iota^{\kappa'} \triangleright \pi$. Thus, we have that zero_pure(σ_3, ι) (or zero_pure(σ, ι)) and $\iota \notin$ dom($\emptyset; \gamma_1; \gamma_y$). The effect addition assumption and the assumption that ok($\gamma_r; \gamma_1$) imply that ok(γ) holds. Thus, $\iota \notin$ dom($\emptyset; \gamma_x$).
  - ι does not belong in the domain of $\emptyset; \gamma_r; \gamma_1$: $\sigma_2 \simeq \sigma_2'; \gamma_r; \gamma_1$, $\sigma_4 \simeq \sigma_2'; \gamma$, for some stack σ_2', $\sigma_3; \gamma_y, \iota^{\kappa'} \triangleright \pi \simeq \sigma_1; \gamma_x, \iota^{\kappa} \triangleright \pi$. Thus, we have that zero_pure(σ_3, ι) (or zero_pure(σ_1, ι)) and $\iota \notin$ dom($\sigma_2'; \gamma_1; \gamma_r; \gamma_y$). The effect addition assumption implies that dom($\emptyset; \gamma$) $\subseteq$ dom($\emptyset; \gamma_r$). Thus, $\iota \notin$ dom($\sigma_2'; \gamma; \gamma_x$).

Case $\delta_1, n_1 \mapsto \sigma_1$: it suffices to show that

- $\sigma; \gamma \vdash \delta_1$: by inversion of $\sigma; \gamma_r; \gamma_1 \vdash \delta$ we obtain that $\sigma; \gamma_r; \gamma_1 \vdash \delta_1$. The proof for this case is completed by applying the induction hypothesis.
- Given that is_accessible($\sigma; \gamma, \iota$), then prove that ¬is_accessible(σ_1, ι) for all regions ι that belong in the domain of $\sigma; \gamma_r; \gamma_1$: the capability addition assumption implies that is_accessible($\sigma; \gamma, \iota$) implies is_accessible($\sigma; \gamma_r; \gamma_1, \iota$). By inversion of $\sigma; \gamma_r; \gamma_1 \vdash \delta$ and the latter fact, and the fact that ι belongs in the domain of $\emptyset; \gamma_r; \gamma_1$ (capability addition assumption), we obtain that is_accessible($\sigma; \gamma_r; \gamma_1, \iota$) and thus, ¬is_accessible(σ_1, ι).

Lemma 51 (Store typing preservation — Pop Helper 1) $\vdash \delta, n \mapsto \sigma; \gamma_r; \gamma_1 \wedge \text{ok}(\gamma_r; \gamma_1) \wedge \mathsf{seq} \vdash \gamma = \gamma_1 \oplus (\gamma_r \ominus \emptyset) \Rightarrow \vdash \delta, n \mapsto \sigma; \gamma$

Proof. It suffices to prove that:

- $\vdash \delta$: immediate by inversion inversion of the assumption $\vdash \delta, n \mapsto \sigma; \gamma_r; \gamma_1$.
- $\sigma; \gamma_r; \gamma_1 \vdash \delta$: by inversion of the assumption $\vdash \delta, n \mapsto \sigma; \gamma_r; \gamma_1$ we have that $\sigma; \gamma_r; \gamma_1 \vdash \delta$. The application of lemma 50 completes the proof for this case.

Lemma 52 (Store typing preservation — Pop) $R; M \vdash \delta; S \wedge \text{ok}(\gamma; \gamma') \wedge \mathsf{seq} \vdash \gamma'' = \gamma' \oplus (\gamma \ominus \emptyset) \wedge \delta = \delta'', n \mapsto \sigma; \gamma; \gamma' \wedge \delta' = \delta'', n \mapsto \sigma; \gamma'' \Rightarrow R; M \vdash \delta'; S$

Proof. The store typing assumption implies that the following hold:

- $R; M \vdash S$
- $R \vdash \delta$: immediate as the regions of γ'' are a subset of γ' and γ (by the effect addition assumption).
- $\vdash \delta'$: by inversion of $R; M \vdash \delta; S$ we have that $\vdash \delta'', n \mapsto \sigma; \gamma; \gamma'$. The proof is immediate by the application of lemma 51 to the latter fact, $\text{ok}(\gamma; \gamma')$ and $\mathsf{seq} \vdash \gamma'' = \gamma' \oplus (\gamma \ominus \emptyset)$.

Lemma 53 (Progress — Program) *Let $S; T$ be a closed well-typed configuration with $R; M \vdash \delta; S; T$, then $S; T$ is not stuck ($\vdash S; T$).*

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 *nolock* predicate holds for it. Without loss of generality, we choose a random thread from the thread list, namely $n : e$ and show that it is not stuck. Thus, $T = T_1, n : e$ for some T_1. We use lemma 3 to obtain $R; M \vdash \delta; S; T_1, n : e$. By inversion of the configuration typing derivation we have that $R; M; \delta \vdash T_1, n : e$ and $R; M \vdash \delta; S$. By inversion of the former derivation we obtain that $R; M; \emptyset; \emptyset \vdash e : \langle\rangle \& (\gamma; \emptyset)$, $\text{pops}(\sigma; \gamma : e)$, $\delta = \delta_1, n \mapsto \sigma; \gamma$ and $R; M; \delta_1 \vdash T_1$.

If e is a value then lemma 56 tells us that e is (). $\text{pops}(\sigma; \gamma : ())$ implies that $\sigma; \gamma \equiv \emptyset; \emptyset$. We already have that $T = T_1, n : e$ for some T_1. Thus, a single step can be performed via rule *E-T*. Otherwise, e is not a value. The application of lemma 55 to the latter fact and the typing derivation of e implies that $\exists e_1, E.\ E[e_1] = e$ and $\text{redex}(e_1)$. Thus, $R; M; \emptyset; \emptyset \vdash E[e_1] : \langle\rangle \& (\gamma; \emptyset)$ is also well-typed. The application of lemma 54 to $\text{redex}(e_1)$, $\delta = \delta_1, n \mapsto \sigma; \gamma$, the typing derivation of $E[e_1]$, $\text{pops}(\sigma; \gamma : E[e_1])$ and $R; M \vdash \delta; S$ implies that one of the following holds:

- $\text{nolock}(\delta, n, e_1)$: the proof is trivially completed as $\text{nolock}(\delta, n, E[e_1])$ also holds.
- $\vdash \delta[n \mapsto \sigma'] \wedge \exists \delta', S', e'.\ \sigma; S; e_1 \rightarrow \sigma'; S'; e'$: A single step can be performed via rule *E-S*.
- $\exists e_2, v, \tau, \gamma_1.\ e_1 \equiv (\lambda x.\, e_2\ \mathtt{as}\ \tau\ v)^{\mathsf{par}}$: A step can only be performed via rule *E-SN*. Thus, it suffices to show that the premises of that rule are satisfied:
  - $T_1, n : e = T$: we have shown that this property holds.
  - fresh n': it is possible to find a thread identifier n' that has never been used previously.
  - $v_1 \equiv \lambda x.\, e_2\ \mathtt{as}\ \tau$: immediate from the assumption.
  - $e_1 \equiv (v_1\ v)^{\mathsf{par}}$: immediate from the assumption.
  - $e' \equiv (v_1\ v)^{\mathsf{seq}}$: the language syntax allows us to formulate this term.
  - $\delta = \delta_1, n \mapsto \sigma; \gamma$: we have shown that this property holds.
  - $\mathsf{par} \vdash \gamma' = \emptyset \oplus (\gamma \ominus \gamma_1)$: The application of lemma 11 to the typing derivation of $E[e_1]$ implies that $R; M; \emptyset; \emptyset \vdash e_1 : \tau \& (\gamma; \gamma'')$ for some τ and γ''. By applying lemma 10 to the latter derivation we obtain that v_1 is a well-typed abstraction and $\mathsf{par} \vdash \gamma'' = \emptyset \oplus (\gamma \ominus \gamma_1')$, where γ_1' is the effect embedded in the type assigned to v_1. By applying lemma 10 to the typing derivation of v_1 we have that if γ_1 the type ascribed on v_1, then $\gamma_1 \simeq \gamma_1'$. The application of lemma 33 to the latter facts give us that $\mathsf{par} \vdash \gamma'' = \emptyset \oplus (\gamma \ominus \gamma_1)$. The effect addition derivation is deterministic thus $\gamma'' = \gamma'$.
  - $\delta' = \delta_1, n \mapsto \sigma; \gamma', n' \mapsto \emptyset; \gamma_1$: the syntax of δ allows us to formulate this context.

Lemma 54 (Progress — Expressions) $\text{redex}(e) \wedge \delta = \delta_1, n \mapsto \sigma; \gamma \wedge R; M; \emptyset; \emptyset \vdash E[e] : \langle\rangle \,\&\, (\gamma; \emptyset) \wedge \text{pops}(\sigma; \gamma : E[e]) \wedge R; M \vdash \delta; S \Rightarrow \text{nolock}(\delta, n, e) \vee (\vdash \delta[n \mapsto \sigma'] \wedge \exists \sigma', S', e'.\ (\sigma; \gamma); S; e \rightarrow \sigma'; S'; e') \vee (\exists e_1, \tau, v.\, e \equiv (\lambda x.\, e_1\ \mathtt{as}\ \tau\ v)^{\mathsf{par}})$

Proof. The application of lemma 11 to the typing derivation of $E[e]$ gives us that $R; M; \emptyset; \emptyset \vdash e : \tau \,\&\, (\gamma;\gamma')$ for some γ' and τ. We proceed by perfoming induction on the typing derivation of e.

Case *T-I*, *T-U*, *T-F*, *T-L*, *T-R*, *T-RF*,*T-V*: this is a contradiction as e should be a value, but we have assumed that e is a *redex*.

Case *T-E*: The conclusion of rule *T-E* implies that shape of e is of the form $\mathtt{pop}_{\gamma_r}\ e'$. We have assumed that e is a *redex*, thus e is of the form $\mathtt{pop}_{\gamma_r}\ v$. The assumption that $\mathrm{pops}(\sigma; \gamma : \mathtt{pop}_{\gamma_r}\ v)$ implies that $\sigma = \sigma'; \gamma_r$. By applying lemma 10 to the derivation of e we obtain that $\mathsf{seq} \vdash \gamma' = \gamma \oplus (\gamma_r \ominus \emptyset)$ holds. We can apply rule *E-E* to the latter fact and the fact that $\exists\, \delta_1.\ \delta = \delta_1, n \mapsto \sigma'; \gamma_r; \gamma$ to perform a single step. Lemma 51, $\mathrm{ok}(\gamma_r; \gamma_1)$[1] and $\mathsf{seq} \vdash \gamma' = \gamma \oplus (\gamma_r \ominus \emptyset)$ we have that $\vdash \delta[n \mapsto \sigma']$ holds, where $\sigma' = \sigma; \gamma'$.

Case *T-AP*: The conclusion of rule *T-AP* implies that shape of e is of the form $(e_1\ e_2)^{\xi}$. We have assumed that e is a *redex*, thus e is of the form $(v_1\ v_2)^{\xi}$. Thus, $R; M; \emptyset; \emptyset \vdash (v_1\ v_2)^{\xi} : \tau \& (\gamma; \gamma')$ holds. If ξ equals par then the proof is trivially completed. Otherwise, $\xi = \mathsf{seq}$ and we can use lemma 10 to derive that $\xi \vdash \gamma' = \gamma'_2 \oplus (\gamma \ominus \gamma'_1)$, if $\tau'_1 \xrightarrow{\gamma'_1 \to \gamma'_2} \tau$ is the type assigned to v_1. The application of lemma 10 to v_1 typing derivation, implies that $\gamma_1 \simeq \gamma'_1$ and $\gamma_2 \simeq \gamma'_2$. Thus, by lemma 33 we have that $\xi \vdash \gamma' = \gamma_2 \oplus (\gamma \ominus \gamma_1)$, where γ_1 and γ_2 are the types ascribed on v_1. The latter derivation implies that $\xi \vdash \gamma = \gamma_1 \oplus \gamma_r$, for some γ_r. Consequently, rule *E-A* can be used to perform a single step. Lemma 48, $\vdash \delta$ (immediate by $R; M \vdash \delta; S$), $\mathrm{ok}(\gamma; \gamma_1)$ (obtained by $\mathrm{ok}(\gamma_1; \gamma_2)$ and $\mathrm{ok}(\gamma; \gamma'')$ holds[2]) and $\xi \vdash \gamma = \gamma_1 \oplus \gamma_r$ imply that $\vdash \delta[n \mapsto \sigma']$ holds, where $\sigma' = \sigma; \gamma_r; \gamma_1$.

Case *T-RP*: rule *E-RP* can be used to perform a single step. $\vdash \delta$ holds by the assumption $R; M \vdash \delta; S$.

Case *T-NG*: Lemma 11 implies that the region allocation construct is well-typed. Lemma 10 implies that $\mathrm{is\_live}(\gamma, r)$ holds, where r is the parent region. Therefore, we can perform a single step by rule *E-NG*. $\vdash \delta[n \mapsto \sigma''; \gamma, \iota^{1,1} \triangleright r]$, where $\sigma = \sigma''; \gamma$, trivially holds as ι is a *fresh* region.

Case *T-NR*: Similar to the previous case. The store and redex typing gives us that region $\bar{r}$ exists in S. Rule *E-NR* can be used to perform a single step. $\vdash \delta$ holds by the assumption $R; M \vdash \delta; S$.

Case *T-D*: Similar to the previous case. The store and redex typing gives us that region $\bar{r}$ and location ℓ exist in S. The application of lemma 10 to the typing derivation of e implies that $\mathrm{is\_accessible}(\gamma, r)$ holds. Rule *E-D* can be used to perform a single step. $\vdash \delta$ holds by the assumption $R; M \vdash \delta; S$.

Case *T-A*: Similar to the previous case. Rule *E-AS* can be used to perform a single step. $\vdash \delta$ holds by the assumption $R; M \vdash \delta; S$.

Case *T-CP*: Lemma 11 implies that the *cap* construct is well-typed. Lemma 10 implies that region $\mathrm{is\_live}(\gamma, r)$. If $\mathrm{nolock}(\delta, n, e)$ holds and the proof is immediate. Otherwise, $\neg\mathrm{nolock}(\delta, n, e)$ holds and we have that at least one of the following holds:

- $\delta \neq \delta'', n \mapsto \sigma; \gamma, r^{\kappa} \triangleright \pi$: this case does not hold as it contradicts an assumption of this lemma and $\mathrm{is\_live}(\gamma, r)$.
- $\eta \neq \mathsf{lk}+$: the proof is completed by applying lemma 57 to $\delta = \delta'', n \mapsto \sigma; \gamma, r^{\kappa} \triangleright \pi$, $\delta' = \delta'', n \mapsto \sigma'$, where $\sigma' = \sigma; \mathrm{live}(\gamma, r^{\kappa'} \triangleright \pi)$, $\vdash \delta$ (store typing assumption), $\kappa' = [\![\eta]\!](\kappa)$ (premise of rule *E-C*) and $\eta \neq \mathsf{lk}+$.
- $\vdash \delta[n \mapsto \sigma']$: if this case holds the proof is trivially completed.

Lemma 55 (Expression — Redex) $R; M; \Delta; \Gamma \vdash e : \tau_1 \,\&\, (\gamma_1;\gamma_2) \wedge e \not\equiv v_1 \Rightarrow \exists\, e', E.E[u] \equiv e \wedge \mathrm{redex}(e)$

Proof. Straightforward proof by induction on the typing derivation.

Case *T-I*, *T-U*, *T-F*, *T-L*, *T-R*, *T-RF* then the proof is immediate as e is a value.

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

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

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

Lemma 56 (Cannonical Forms) $R; M; \Delta; \Gamma \vdash v : \tau \,\&\, (\gamma_1;\gamma_2) \Rightarrow$
$\tau \equiv \langle\rangle \Rightarrow v \equiv () \wedge$
$\tau \equiv \mathtt{rgn}(\bar{r}) \Rightarrow (v \equiv \mathtt{rgn}_{\bar{r}} \wedge \bar{r} \in R) \wedge$
$\tau \equiv \mathtt{ref}(\tau, r) \Rightarrow (v \equiv \mathtt{loc}_{\ell} \wedge \ell \mapsto (\tau, \bar{r}) \in M) \wedge$
$\tau \equiv b \Rightarrow v \equiv n \wedge$
$\tau \equiv \tau_1 \xrightarrow{\gamma_1 \to \gamma_2} \tau_2 \Rightarrow v \equiv \lambda x.\, e\ \mathtt{as}\ \tau_1 \xrightarrow{\gamma_1 \to \gamma_2} \tau_2 \wedge$
$\tau \equiv \forall \rho.\, \tau \Rightarrow v \equiv \Lambda \rho.\, f$

[1] $\mathrm{ok}(\gamma_r; \gamma_1)$ can be obtained in the same way as in lemma 29 case *T-E*.

[2] we can obtain $\mathrm{ok}(\gamma_1; \gamma_2)$ and $\mathrm{ok}(\gamma; \gamma'')$ in the same way as in lemma 29 case *T-AP*.

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

Lemma 57 (Store Progress — $\vdash \delta$) $\delta = \delta'', n \mapsto \sigma; \gamma, r^{\kappa} \triangleright \pi \wedge \delta' = \delta'', n \mapsto \sigma; \text{live}(\gamma, r^{\kappa'} \triangleright \pi) \wedge \vdash \delta \wedge \kappa' = [\![\eta]\!](\kappa) \wedge \eta \neq \mathsf{Ik}+ \Rightarrow \vdash \delta'$.

Proof. To prove that $\vdash \delta'$ holds it suffices to show that its premises hold. By inversion of $\vdash \delta'$ we have that:

- $\vdash \delta''$: The assumption tells us that $\vdash \delta'', n \mapsto \sigma; \gamma, r^{\kappa} \triangleright \pi$ holds. By inversion of the latter fact we have that $\vdash \delta''$ holds.
- $\sigma; \text{live}(\gamma, r^{\kappa'} \triangleright \pi) \vdash \delta''$: Let γ_0 be equal to $\gamma, r^{\kappa} \triangleright \pi$ and γ'_0 be equal to $\text{live}(\gamma, r^{\kappa'} \triangleright \pi)$. The proof is completed by the application of lemma 58 to $\kappa' = [\![\eta]\!](\kappa), \delta = \delta'', n \mapsto \sigma; \gamma_0, \delta' = \delta'', n \mapsto \sigma; \gamma'_0, \sigma; \gamma_0 \vdash \delta''$ (by inversion of $\vdash \delta$), $\delta'' \subseteq \delta''$ and $\eta \neq \mathsf{Ik}+$.

Lemma 58 (Store Progress — Helper lemma 1) $\kappa' = [\![\eta]\!](\kappa) \wedge \delta = \delta'', n \mapsto \sigma; \gamma \wedge \delta' = \delta'', n \mapsto \sigma; \gamma' \wedge \gamma = \gamma'', r^{\kappa} \triangleright \pi \wedge \gamma' = \text{live}(\gamma'', r^{\kappa'} \triangleright \pi) \wedge \sigma; \gamma \vdash \delta'' \wedge \delta_0 \subseteq \delta'' \wedge \eta \neq \mathsf{Ik}+ \Rightarrow \sigma; \gamma' \vdash \delta_0$.

Proof. We proceed by induction on the derivation of δ_0.

- $\emptyset$: given that $\sigma; \gamma' \simeq \sigma_1; \gamma_x, \iota^{\kappa_1} \triangleright \pi + \sigma_2$ and $\text{rg}(\kappa_1) > 0$ and $\text{is\_pure}(\kappa_1)$, it suffices to prove that $\text{zero\_pure}(\sigma_1, \iota)$ and $\iota \notin \text{dom}(\sigma_2; \gamma_x)$ for all regions ι that belong in the domain of $\sigma; \gamma'$. By inversion of $\sigma; \gamma \vdash \emptyset$ (derived by the assumption $\sigma; \gamma \vdash \delta''$), we have that $\sigma; \gamma \simeq \sigma_3; \gamma_y, \iota^{\kappa_2} \triangleright \pi + \sigma_4$, $\text{rg}(\kappa_2) > 0$ and $\text{is\_pure}(\kappa_2)$, $\text{zero\_pure}(\sigma_3, \iota)$, and $\iota \notin \text{dom}(\sigma_4; \gamma_y)$. We proceed by performing a case analysis:
  - If ι does not belong in the domain of $\emptyset; \gamma$, then $\sigma_3 \simeq \sigma_1$, $\gamma_y \simeq \gamma_x$ and there exists a stack σ'_4 such that $\sigma_2 \simeq \sigma'_4; \gamma'$ and $\sigma_4 \simeq \sigma'_4; \gamma$. Hence, $\text{zero\_pure}(\sigma_1, \iota)$ holds. The assumption that $\gamma' = \text{live}(\gamma'', r^{\kappa'} \triangleright \pi)$ and $\gamma = \gamma'', r^{\kappa} \triangleright \pi$ holds implies that $\text{dom}(\emptyset; \gamma') \subseteq \text{dom}(\emptyset; \gamma)$. Consequently, $\iota \notin \text{dom}(\sigma'_4; \gamma'; \gamma_x)$ holds.
  - If ι does belong in the domain of $\emptyset; \gamma$, then $\sigma_4 = \sigma_2 = \emptyset$ and $\sigma_1 \simeq \sigma_3 \simeq s$. Hence, $\text{zero\_pure}(\sigma_1, \iota)$ trivially holds. The assumption that $\gamma' = \text{live}(\gamma'', r^{\kappa'} \triangleright \pi)$ and $\gamma = \gamma'', r^{\kappa} \triangleright \pi$ holds implies that $\text{dom}(\emptyset; \gamma') \subseteq \text{dom}(\emptyset; \gamma)$ and thus, $\text{dom}(\emptyset; \gamma_x) \subseteq \text{dom}(\emptyset; \gamma_y)$. Consequently, $\iota \notin \text{dom}(\emptyset; \gamma_x)$ holds.

Case $\delta_1, n_1 \mapsto \sigma_1$: it suffices to prove that

- $\sigma; \gamma' \vdash \delta_1$: is immediate by applying the induction hypothesis.
- Given that $\text{is\_accessible}(\sigma; \gamma', \iota)$ holds for all ι that belong in the domain of $\sigma; \gamma'$, it suffices to prove that $\neg\text{is\_accessible}(\sigma_1, \iota)$ holds. The assumption that $\sigma; \gamma \vdash \delta''$ holds, the fact that ι is accessible in γ' and $\eta \neq \mathsf{Ik}+$ imply that $\text{is\_accessible}(\sigma; \gamma, \iota)$ and thus $\neg\text{is\_accessible}(\sigma_1, \iota)$.