# Formal Type Soundness for Cyclone's Region System

Dan Grossman[*] Greg Morrisett[*] Trevor Jim[†]
Mike Hicks[*] Yanling Wang[*] James Cheney[*]

November 2001

**Abstract**

Cyclone is a polymorphic, type-safe programming language derived from C. The primary design goals of Cyclone are to let programmers control data representations and memory management without sacrificing type-safety. In this paper, we focus on the region-based memory management of Cyclone and its static typing discipline. The design incorporates several advancements, including support for region subtyping and a coherent integration with stack allocation and a garbage collector. To support separate compilation, Cyclone requires programmers to write some explicit region annotations, but uses a combination of default annotations, local type inference, and a novel treatment of region effects to reduce this burden. As a result, we integrate C idioms in a region-based framework. In our experience, porting legacy C to Cyclone has required altering about 8% of the code; of the changes, only 6% (of the 8%) were region annotations.

This technical report is really two documents in one: The first part is a paper submitted for publication in November, 2001. The second part is the full formal language and type-safety proof mentioned briefly in the first part. If you have already read a version of, "Region-Based Memory Management in Cyclone", then you should proceed directly to Section 9.

## 1 Introduction

Many software systems, including operating systems, device drivers, file servers, and databases require fine-grained control over data representation (e.g., field layout) and resource management (e.g., memory management). The *de facto* language for coding such systems is C. However, in providing low-level control, C admits a wide class of dangerous — and extremely common — safety violations, such as incorrect type casts, buffer overruns, dangling-pointer dereferences, and space leaks. As a result, building large systems in C, especially ones including third-party extensions, is perilous. Higher-level, type-safe languages avoid these drawbacks, but in so doing, they often fail to give programmers the control needed in low-level systems. Moreover, porting or extending legacy code is often prohibitively expensive. Therefore, a safe language at the C level of abstraction, with an easy porting path, would be an attractive option.

Toward this end, we have developed *Cyclone* [6], a language designed to be extremely close to C while remaining type-safe. We have written or ported over 70,000 lines of Cyclone code, including the Cyclone compiler, an extensive library, lexer and parser generators, compression utilities, a

[*]Cornell University. This material is based on work supported in part by AFOSR grants F49620-00-1-0198, F49620-01-1-0298, F49620-00-1-0209, and F49620-01-1-0312; Sloan grant BR-3734; ONR grant N00014-01-1-0968; NSF grant 9875536; and NSF Graduate Fellowships. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not reflect the views of these agencies.

[†]AT&T Labs Research.

Windows device driver, and a web server. In so doing, we identified many common C idioms that are usually safe, but for which the C type system is too weak to verify. We then augmented the language with modern features and typing technologies so that programmers could continue to use those idioms, but have safety guarantees.

For example, to reduce the need for type casts, Cyclone has features like parametric polymorphism, subtyping, and tagged unions. To prevent bounds violations without making hidden data-representation changes, Cyclone has a variety of pointer types with different compile-time invariants and associated run-time checks. Other projects aimed at making legacy C code safe have addressed these issues with somewhat different approaches, as discussed in Section 7.

In this paper, we focus on the most novel aspect of Cyclone: its system for preventing dangling-pointer dereferences and space leaks. The design addresses several seemingly conflicting goals. Specifically, the system is:

- *Sound:* Programs never dereference dangling pointers.
- *Static:* Dereferencing a dangling pointer is a compile-time error. No run-time checks are needed to determine if memory has been deallocated.
- *Convenient:* We minimize the need for explicit programmer annotations while supporting many C idioms. In particular, C code that manipulates stack pointers often requires no modification.
- *Exposed:* We provide mechanisms that let programmers control the placement and lifetimes of objects. As in C, local declarations are always stack-allocated.
- *Comprehensive:* We treat all memory uniformly, including the stack, the heap (which can be optionally garbage-collected), and "growable" regions.
- *Scalable:* The system supports separate compilation, as all analyses are intra-procedural.

Following the seminal work of Tofte and Talpin [22], the system is *region-based*: each object lives in a distinct region of memory and, with the optional exception of the heap, a region's objects are all deallocated simultaneously. As a static system for an explicitly typed, low-level language, Cyclone's region framework makes several important technical contributions over previous work, notably:

- Region subtyping: A last-in-first-out discipline on region lifetimes induces an "outlives" relationship on regions, which, in turn, allows us to provide a useful subtyping discipline on pointer types.
- Simple effects: We eliminate the need for effect variables (which complicate interfaces) through the use of a "`regions_of`" type operator.
- Local region inference: Though inference is local, a system of defaults minimizes the need for explicit region annotations.
- Integration of existentials: The combination of region subtyping and simple effects makes the integration of first-class abstract data types relatively simple.

In the rest of this paper, we demonstrate these contributions. We begin with a general description of the system suitable for C programmers (Section 2), and then follow with a more technical discussion of our novel effect system and its interaction with existential types (Section 3). We continue with a core formal language that we have proven sound (Section 4), an overview of our implementation (Section 5), and a study of the cost of porting C code to Cyclone (Section 6). We discuss related work in Section 7 and future work in Section 8.

## 2 Using Cyclone Regions

This section presents the programmer's view of Cyclone's memory-management system. It starts with the constructs for creating regions, allocating objects, and so on — this part is simple because the departure from C is small. We next present the corresponding type system, which is more involved because every pointer type carries a region annotation. Then we show how regions' lifetimes induce subtyping on pointer types. At that point, the type syntax is quite verbose, so we explain the features that, in practice, eliminate almost all region annotations. Throughout, we take the liberty of using prettier syntax (e.g., Greek letters) than actual Cyclone. For the ASCII syntax and a less region-oriented introduction to Cyclone, see the user's manual [6].

### 2.1 Basic Operations

In Cyclone, all memory is in some region, of which there are three kinds:

- A single heap region, which conceptually lives forever.
- Stack regions, which correspond to local-declaration blocks, as in C.
- Dynamic regions, which have lexically scoped lifetimes but permit unlimited allocation into them.

Static data are placed in the heap. Primitives `malloc` and `new` create new heap objects. The `new` operation is like `malloc` except that it takes an expression and initializes the memory with it. There is no explicit mechanism for reclaiming heap-allocated objects (e.g., `free`). However, Cyclone programs may optionally link against the Boehm-Demers-Weiser conservative garbage collector [4] to implicitly reclaim unreachable heap-allocated objects. The interaction of the collector with regions is discussed in Section 5.

Stack regions correspond directly to C's local-declaration blocks: entering a block with local declarations creates storage with a lifetime corresponding to the lexical scope of the block. Function parameters are in a stack region corresponding to the function's lifetime. In short, Cyclone local declarations and function parameters have exactly the same layout and lifetimes as in C.

Dynamic regions are created with the construct `region r {`*s*`}`, where `r` is an identifier and *s* is a statement. The region's lifetime is the execution of *s*. In *s*, `r` is bound to a *handle* for the region, which primitives `rmalloc` and `rnew` use to allocate objects into the associated region. For example, `rnew(r) 3` returns a pointer to an `int` allocated in the region of handle `r` and initialized to 3. Handles are first-class values; a caller may pass a handle to a function to allow it to allocate into the associated region. A pre-defined constant `heap_region` is a handle for the heap.

Like a declaration block, a dynamic region is deallocated precisely when execution leaves the body of the enclosed statement. Execution can leave due to unstructured jumps (`continue`, `goto`, etc.), a `return`, or via an exception. Section 5 explains how we compile dynamic-region deallocation.

The region system imposes no changes on the representation of pointers or the meaning of operators such as `&` and `*`. There are no hidden fields or reference counts for maintaining region information at run-time. Pointers to arrays of unknown size (denoted $\tau$ `?`) are implemented with extra fields to support bounds-checks, but this design is orthogonal to regions. As a result, all the infrastructure for preventing dangling-pointer dereferences is in the static type system, making such dereferences a compile-time error.

### 2.2 Basic Type System

**Region Annotations** All pointers point into exactly one region. In principle, pointer types are annotated with the *region name* of the region they point into, though in practice we eliminate most

```
char?ρ   strcpy<ρ,ρ_2>(char?ρ d, const char?ρ_2 s);
char?ρ_H strdup<ρ>(const char?ρ s);
char?ρ   rstrdup<ρ,ρ_2>(region_t<ρ>,const char?ρ_2 s);
size_t   strlen<ρ>(const char?ρ s);
```

Figure 1: Cyclone string library prototypes

annotations. Ignoring subtyping, `int*`$\rho$ describes a pointer to an `int` that is in the region whose name is $\rho$. This invariant—pointers have a particular region—is the basic restriction we impose to make the undecidable problem of detecting dangling-pointer dereferences tractable. Pointer types with different region names are different types. A handle for a region corresponding to $\rho$ has the type `region_t<`$\rho$`>`.

Region names fall into four categories. The region name for the heap is $\rho_H$. A block labeled `L` (such as `L:{int x=0;` $s$`}`) has name $\rho_L$, and refers to the stack region that the block creates. Similarly, the arguments of a function `f` are stored in the stack region $\rho_f$. Finally, the statement `region r {`$s$`}` defines region name $\rho_r$ for the created region. So `r` has type `region_t<`$\rho_r$`>`. In all cases, the scope of a region name corresponds to the lifetime of the corresponding region.

We can now give types to some small examples. If $e_1$ has type `region_t<`$\rho$`>` and $e_2$ has type $\tau$, then `rnew (`$e_1$`)` $e_2$ has type $\tau$`*`$\rho$. If `int x` is declared in block `L`, then `&x` has type `int*`$\rho_L$. Similarly, if $e$ has type $\tau$`*`$\rho$, then `&*`$e$ has type $\tau$`*`$\rho$.

**Preventing dangling-pointer dereferences** To dereference a pointer, safety demands that its region be live. Our goal is to determine at compile-time that no code follows a dangling pointer. It appears that no well-typed pointer could be a dangling reference, because pointer types' region names must be in scope. For example, this code is ill-typed:

```
1. int*ρ_L p;
2. L:{ int x = 0;
3.     p = &x;
4.   }
5. *p = 42;
```

The code creates storage for `x` at line 2 and deallocates it at line 4, so the assignment of `&x` to `p` creates a dangling pointer that is dereferenced in line 5. Cyclone rejects this code because $\rho_L$ is not in scope when `p` is declared. If we change the declaration of `p` to another region, then the assignment `p = &x` fails to type-check because `&x` has type `int*`$\rho_L$.

However, Cyclone's advanced features, notably existential and universal polymorphism, conspire to allow pointers to escape the scope of their regions, just as closures allow pointers to escape in the original Tofte-Talpin work. Therefore, in general, we cannot rely upon simple scoping mechanisms to ensure soundness, and must instead track the set of live regions at each control-flow point. To keep the analysis intra-procedural, we use a novel type-and-effects system to track inter-procedural liveness requirements. We delay the full discussion of effects until Section 3.

**Region Polymorphism** Functions in Cyclone are *region-polymorphic*; they can abstract the actual regions of their arguments or results. That way, functions can manipulate pointers regardless of the region they point into, whether it be the stack, the heap, or a dynamic region.

Figure 1 presents some prototypes from the Cyclone string library, including `strcpy`, `strdup`, and `strlen`, and a region-allocating function `rstrdup`. The `?` is Cyclone notation for a pointer to a dynamically-sized array. These functions all exhibit region polymorphism. In `strcpy`, the parameters' region names $\rho$ and $\rho_2$ are abstracted by the syntax <$\rho, \rho_2$>, meaning they can be instantiated with any actual region name when the function is called. So we can write code like:

```
 L:{ char buf[20];
      strcpy<$\rho_L, \rho_H$>(buf,"a heap pointer"); }
```

Here, the call instantiates $\rho_2$ with the heap region $\rho_H$ and $\rho$ with the stack region $\rho_L$, allowing one to copy a string from the heap to the stack.

Region polymorphism can also guarantee region equalities of unknown regions by using the same region names. For example, in `strcpy` the region names of the first argument and the return value are the same; so the returned pointer must point to the same region as the first argument. Region name equalities are also important for dynamic regions. For example, the `rstrdup` function is a version of `strdup` that copies the source string into a dynamic region. In its prototype, we see that the region name of the returned value $\rho$ matches the region name of the dynamic region handle `region_t`<$\rho$>. In fact, we implement `strdup` by just calling `rstrdup`:

```
char?$\rho_H$ strdup<$\rho$>(const char?$\rho$ s) {
  return rstrdup<$\rho_H, \rho$>(heap_region,s);
}
```

**Polymorphic Recursion** It is often valuable to instantiate the region parameters of a recursive function call with different names than the function's own arguments. As an example, this contrived program has a function `fact` that abstracts a region $\rho$ and takes as arguments a pointer into $\rho$ and an integer.

```
void fact<$\rho$>(int*$\rho$ result, int n) {
 L: { int x = 1;
      if(n > 1) fact<$\rho_L$>(&x,n-1);
      *result = x*n; }
}
int g = 0;
int main() { fact<$\rho_H$>(&g,6); return g; }
```

When executed, the program returns the value 720. In `main`, we pass `fact` a heap pointer (`&g`), so the type of `fact` is instantiated with $\rho_H$ for $\rho$. In contrast, the recursive call instantiates $\rho$ with $\rho_L$, which is the name of the stack region. At run time, the first instance of `fact` modifies `g`; each recursive call modifies the value of `x` in its caller's stack frame.

**Type Definitions** Because `struct` definitions can contain pointers, Cyclone allows `struct`s to be parameterized by region names. For example, here is a type for lists of pointers to ints:

```
struct Lst<$\rho_1$,$\rho_2$> {
  int*$\rho_1$ hd;
  struct Lst<$\rho_1$,$\rho_2$> *$\rho_2$ tl;
};
```

Ignoring subtyping, a value of type `struct Lst`<$\rho_1$,$\rho_2$> will be a list with `hd` fields that point into $\rho_1$ and `tl` fields that point into $\rho_2$. Other invariants are possible: If the type of `tl` were `struct Lst`<$\rho_2$,$\rho_1$>`*` $\rho_2$, we would describe lists where the regions for `hd` and `tl` alternated at each element.

Type abbreviations using `typedef` can also have region parameters. For example, we can define region-allocated lists of heap-allocated pointers with:
`typedef struct Lst`<$\rho_H$,$\rho$>` list_t`<$\rho$>`;`.

```
char?ρ strcpy(char?ρ d, const char? s);
char? strdup(const char? s);
char?ρ rstrdup(region_t<ρ>,const char? s);
size_t strlen(const char? s);
```

Figure 2: Cyclone prototypes minimally-annotated

## 2.3 Subtyping

While the type system we have described thus far is quite powerful, it is not expressive enough in some cases. For example, it is common to define a local variable to alternatively hold the value of one of its arguments:

```
void f<ρ1,ρ2>(int b, int*ρ1 p1, int*ρ2 p2) {
 L: { int*ρL p;
      if(b) p = p1; else p=p2;
      /* ... do something with p ... */ }
}
```

In the type system described thus far, the program fails to type-check because neither `p1` nor `p2` has type `int*`$\rho_L$. We cannot change the type of `p` to `int*`$\rho_1$ or `int*`$\rho_2$, for then one of the assignments would fail.

To solve this problem, we observe that if the region corresponding to $\rho_1$ *outlives* the region corresponding to $\rho_2$, then it is sound to use a value of type $\tau$`*`$\rho_1$ where we expect one of type $\tau$`*`$\rho_2$. Cyclone supports such coercions implicitly. The last-in-first-out region discipline makes such outlives relationships common: when we create a region, we know every region currently alive will outlive it. Simple subtyping based on this outlives relationship allows the above program to type-check.

Region-polymorphic functions can specify outlives relationships among their arguments with explicit pre-conditions that express partial orders on region lifetimes. In practice, we have not used this feature, because the local outlives information has sufficed.

To ensure soundness, we do not allow casting $\tau_1$`*`$\rho$ to $\tau_2$`*`$\rho$, even if $\tau_1$ is a subtype of $\tau_2$, as this cast would allow putting a $\tau_2$ in a location where other code expects a $\tau_1$. (This problem is the usual one with covariant subtyping on references.) However, Cyclone does allow casts from $\tau_1$`*`$\rho$ to `const` $\tau_2$`*`$\rho_2$ when $\tau_1$ is a subtype of $\tau_2$. To ensure soundness, we must enforce read-only access for `const` values (unlike C). This support for "deep" subtyping, when combined with polymorphic recursion, is powerful enough to allow stack allocation of some recursive structures of arbitrary size.

## 2.4 Eliminating Annotations

Although Cyclone is explicitly typed in principle, it would be too onerous to fully annotate every function. Instead, we use a combination of inference and well-chosen defaults to dramatically reduce the number of annotations needed in practice. We emphasize that our approach to inference is purely intra-procedural and that prototypes for functions are never inferred. Rather, we use a default completion of partial prototypes to minimize region annotations. This approach permits separate compilation.

When writing a pointer type (e.g., `int*`), the region annotation is optional; the compiler deduces an appropriate annotation based on context:

1. For local declarations, a unification-based inference engine infers the annotation from the declaration's (intra-procedural) uses. This local inference works well in practice, especially when declarations have initializers.

2. Omitted region names in argument types are filled in with fresh region names that are implicitly generalized. So by default, functions are region polymorphic without any region equalities.

3. In all other contexts (return types, globals, type definitions), omitted region names are filled in with $\rho_H$ (i.e., the heap). This default works well for global variables and for functions that return heap-allocated results. However, it fails for functions like `strcpy` that return one of their parameters. On the other hand, without looking at the function body, we cannot determine which parameter (or component of a parameter) the function might return.

In addition, when calling a region-polymorphic function, the programmer can omit the explicit region-name instantiation and the inference engine discovers it. As a result of these devices, our `fact` example can become annotation-free:

```
void fact(int* result, int n) {
  int x = 1;
  if(n > 1) fact(&x,n-1);
  *result = x*n;
}
```

Taken another way, the function above, when treated as C code, ports to Cyclone with no additional annotations. Figure 2 shows the same string library functions as Figure 1, but minimally annotated. In all cases, the lack of a region annotation on the argument `s` means the type-checker would insert a fresh region name for the pointer type, and generalize it. The lack of an annotation on the return type of `strdup` defaults to the heap. In total, five region annotations were removed and all generalization became implicit.

While the default annotations and inference engine reduce the burden on the programmer and make porting easier, it is still necessary to put in some explicit annotations to express equalities necessary for safety. For example, if we write:

```
void f2(int** pp, int* p) {*pp=p;}
```

then the code elaborates to:

```
void f2<ρ1,ρ2,ρ3>(int *ρ1*ρ2 pp, int *ρ3 p) {*pp=p;}
```

which fails to type-check because `int*`$\rho_1 \neq$ `int*`$\rho_3$. The programmer must insert an explicit region annotation to assert an appropriate equality relation on the parameters:

```
void f2(int*ρ* pp, int*ρ p) { *pp = p; }
```

Finally, we employ another technique that dramatically reduces annotations in practice, with regard to type definitions. we can partially apply parameterized type definitions; elided arguments are filled in via the same rules used for pointer types. Here is an aggressive use of this feature:

```
typedef struct Lst<ρ1,ρ2> *ρ2 l_t<ρ1,ρ2>;
l_t heap_copy(l_t l) {
  l_t ans = NULL;
  for(l_t l2 = l; l2 != NULL; l2 = l2->tl)
    ans = new Lst(new *l2->hd,ans);
  return ans;
}
```

Because of defaults, the parameter type is `l_t<`$\rho_1$`,`$\rho_2$`>` and the return type is `l_t<`$\rho_H$`,`$\rho_H$`>`. Because of inference, the compiler assigns `ans` the type `l_t<`$\rho_H$`,`$\rho_H$`>` and `l2` the type `l_t<`$\rho_1$`,`$\rho_2$`>`.

## 3 Effects

We argued in Section 2.2 that the scope restrictions on region types prevent pointers from escaping the scope of their region. In particular, a function or block cannot return or assign a value of type $\tau$*$\rho$ outside the scope of $\rho$'s definition, simply because you cannot write down a (well-formed) type for the result. Indeed, if Cyclone had no mechanisms for type abstraction, this property would hold.

But if there is some way to hide a pointer's type in a result value, then the pointer could escape the scope of its region. For instance, if Cyclone had (upwards-escaping) closures, then one could hide a pointer to a local variable in the closure's environment, and return the closure outside the scope of the variable, thereby introducing a dangling pointer. This, in and of itself, is not a problem, but if the closure is later invoked, then it might dereference the dangling pointer. This is the critical problem that Tofte and Talpin addresses for functional languages.

Cyclone does not have closures, but it has other typing constructs that hide regions. In particular, Cyclone provides existential types [17], which suffice to encode closures [16] and simple forms of objects [5]. Therefore, it is possible in Cyclone for pointers to escape the scope of their regions.

To address this problem, the Cyclone type system keeps track of the subset of regions that are live at any control-flow point. Following Walker, Crary, and Morrisett [23], we call the set of live regions the *capability.* Before dereferencing a pointer, the type system ensures that the associated region is in the capability. Similarly, before calling a function, Cyclone ensures that regions the function might access are all in the current capability. To this end, function types carry an *effect* that records the set of regions the function might access. The idea of using effects to ensure soundness is due to Tofte and Talpin (hereafter TT). However, our treatment of effects differs substantially from previous work.

The first major departure from TT is that we calculate default effects from the function prototype alone (instead of inferring them from the function body) in order to preserve separate compilation. The default effect is simply the set of regions that appear in the argument or result types. For instance, given the prototype:

```
int*ρ1 f(int*, int*ρ1*);
```

which elaborates to:

```
int*ρ1 f<ρ1,ρ2,ρ3>(int*ρ2, int*ρ1*ρ3);
```

the default effect is {$\rho_1, \rho_2, \rho_3$}. In the absence of polymorphism, this default effect is a conservative bound on the regions the function might access. As with region names in prototypes, the programmer can override the default with an explicit effect. For example, if `f` never dereferences its first argument, we can strengthen its prototype by adding an explicit effect as follows:

```
int*ρ1 f(int*ρ2, int*ρ1*ρ3; {ρ1,ρ3});
```

In practice, we have found that default effects extremely useful. Indeed, for the 70,000 lines of Cyclone code we have written thus far, we have written one non-default effect.

The second major departure from TT is that we do not have *effect variables.* Effect variables are used by TT for three purposes: (1) to simulate subtyping in a unification-based inference framework, (2) to abstract the set of regions that a closure might need to access, and (3) to abstract the set of regions hidden by an abstract type.

In our original Cyclone design, we tried to use TT-style effect variables. However, we found that the approach does not work well in an explicitly-typed language for two reasons. First, the effect variables introduced by TT to support effect subtyping could only occur free in one location, and all effect variables had to be prenex quantified [20]. Their unification algorithm depended crucially upon these structural invariants. In an explicitly-typed language, we found that enforcing these constraints was difficult. Furthermore, the prenex quantification restriction prevented first-class polymorphic functions, which Cyclone supports.

Second, we found that effect variables appeared in some library interfaces, making the libraries harder to understand and use. Consider, for instance, an implementation of polymorphic sets:

```
struct Set<α,ρ,ε> {
  list_t<α,ρ> elts;
  int (*cmp)(α,α; ε);
}
```

A `Set` consists of a list of $\alpha$ elements, with the spine of the list in region $\rho$. We do not know where the elements are allocated until we instantiate $\alpha$. The comparison function `cmp` is used to determine set membership. Because the type of the elements is not yet known, the type of the `cmp` function must use an effect variable $\epsilon$ to abstract the set of regions that it might access when comparing the two $\alpha$ values. And this effect variable, like the type and region variable, must be abstracted by the `Set` structure.

Now the library might export the `Set` structure to clients abstractly (i.e., without revealing its definition):

```
struct Set<α,ρ,ε>;
```

The client must somehow discern the connection between $\alpha$ and $\epsilon$, namely that $\epsilon$ is meant to abstract the set of regions within $\alpha$ that the hidden comparison function might access.

To simplify the system while solving the problems that effect variables solve, we use a type operator, `regions_of(`$\tau$`)`. This novel operator is just part of the type system; it does not exist at run-time. Intuitively, `regions_of(`$\tau$`)` represents the set of regions that occur free in $\tau$. In particular:

$$\begin{array}{l}
\texttt{regions\_of(int)} = \emptyset \\
\texttt{regions\_of(}\tau\texttt{*}\rho\texttt{)} = \{\rho\} \cup \texttt{regions\_of(}\tau\texttt{)} \\
\texttt{regions\_of(}(\tau_1,\ldots,\tau_n) \rightarrow \tau\texttt{)} = \\
\ \texttt{regions\_of(}\tau_1\texttt{)} \cup \ldots \cup \texttt{regions\_of(}\tau_n\texttt{)} \cup \texttt{regions\_of(}\tau\texttt{)}
\end{array}$$

For type variables, `regions_of(`$\alpha$`)` is treated as an abstract set of region variables, much like effect variables. For example, `regions_of(`$\alpha$`*`$\rho$`)` = `{`$\rho$`}` $\cup$ `regions_of(`$\alpha$`)`.

With the addition of `regions_of`, we can rewrite the `Set` example as follows:

```
struct Set<α,ρ> {
  list_t<α,ρ> elts;
  int (*cmp)(α,α; regions_of(α));
}
```

Now the connection between the type parameter $\alpha$ and the comparison function's effect is apparent, and the data structure no longer needs to be parameterized by an effect variable. Moreover, `regions_of(`$\alpha$`)` is the default effect for `int (*cmp)(`$\alpha$`,`$\alpha$`)`, so we need not write it.

Now suppose we wish to build a `Set<int*`$\rho_1$`,`$\rho_2$`>` value using some pre-defined comparison function:

```
int cmp_ptr<ρ1>(int*ρ1 p1, int*ρ1 p2) {
  return (*p1) == (*p2);
}
Set<int*ρ1,ρ2> build_set(list_t<int*ρ1,ρ2> e) {
  return Set{.elts = e, .cmp = cmp_ptr<ρ1>};
}
```

The default effect for `cmp_ptr` is {$\rho_1$}. After instantiating $\alpha$ with `int*`$\rho_1$, the effect of `cmp` becomes `regions_of(int*`$\rho_1$`)`, which equals {$\rho_1$}. As a result, the function `build_set` type-checks. And indeed, using any function with a default effect will always succeed. Consequently, programmers need not explicitly mention effects when designing or using libraries.

In addition, unifying function types becomes somewhat easier when default effects are used because, given the same argument and result types, two functions always have the same default effect.

## 3.1 Interaction with Existentials

As mentioned above, Cyclone supports *existential types*, which allow programmers to encode closures. For example, we can give a type for "call-backs" that return an `int`:

```
struct IntFn ∃α { int (*func)(α env); α env;};
```

Here, the call-back consists of a function pointer and some abstracted state that should be passed to the function. The $\alpha$ is existentially bound: Various objects of type `struct IntFn` can instantiate $\alpha$ differently. When a `struct IntFn` object is created, the type-checker ensures there is a type for $\alpha$ such that the fields are correctly initialized.

To access the fields of an existential object, we need to "open" them by giving a name to the bound type variable. For example, we can write (in admittedly alien syntax):

```
int apply_intfn(struct IntFn pkg) {
  let IntFn<β>{.func = f,.env = y} = pkg;
  return f(y);
}
```

The `let` form binds `f` to `pkg.func` with type `int (*)(`$\beta$`)` and `y` to `pkg.env` with type $\beta$. So the function call appears well-typed. However, the effect for `f` is `regions_of(`$\beta$`)` and we have no evidence that these regions are still live, even though $\beta$ is in scope. Indeed, the regions may not be live as the following code demonstrates:

```
int read<ρ>(int*ρ x) { return *x; }
struct IntFn dangle() {
  L:{int x = 0;
      struct IntFn ans =
         <int*ρL>{.func = read<ρL>, .env = &x};
      return ans; }
}
```

Here, the abstracted type $\alpha$ is instantiated with `int*`$\rho_L$ because the call-back's environment is a pointer to an `int x` in region $\rho_L$. The function for the call-back just dereferences the pointer it is passed. When packaged as an existential, the `int*`$\rho_L$ is hidden and thus the result is well-typed despite the fact that the call-back has a dangling pointer.

In short, to use `struct IntFn` objects, we must "leak" enough information to prove a call is safe. We accomplish this by giving `regions_of(`$\alpha$`)` a *bound*:

```
struct IntFn<ρ> ∃α:>ρ { ... };
```

The bound means `regions_of(`$\alpha$`)` must all *outlive* $\rho$. Therefore, if `pkg` has type `struct IntFn<`$\rho$`>`, then we can call the `func` field so long as $\rho$ is live. In practice, this soundly reduces the "effect" of the call-back to a single region.

$$
\begin{array}{rlll}
\text{kinds} & \kappa & ::= & \mathcal{T} \mid \mathcal{R} \\
\text{type and region vars} & \alpha, \rho & & \\
\text{region sets} & \epsilon & ::= & \alpha_1 \cup \cdots \cup \alpha_n \cup \{\rho_1, \ldots, \rho_m\} \\
\text{region constraints} & \gamma & ::= & \emptyset \mid \gamma, \epsilon <: \rho \\
\text{constructors} & \tau & ::= & \alpha \mid \mathrm{int} \mid \tau_1 \xrightarrow{\epsilon} \tau_2 \mid \tau_1 \times \tau_2 \mid \tau * \rho \mid \mathrm{handle}(\rho) \mid \forall \alpha{:}\kappa \triangleright \gamma.\tau \mid \exists \alpha{:}\kappa \triangleright \gamma.\tau \\
\text{expressions} & e & ::= & x_\rho \mid v \mid e\langle\tau\rangle \mid (e_1, e_2) \mid e.i \mid *e \mid \texttt{rnew}(e_1)e_2 \mid \\
 & & & e_1(e_2) \mid \&e \mid e_1 = e_2 \mid \texttt{pack}\,[\tau_1, e]\,\texttt{as}\,\tau_2 \\
\text{values} & v & ::= & i \mid f \mid \&p \mid \texttt{region}(\rho) \mid (v_1, v_2) \mid \texttt{pack}\,[\tau_1, v]\,\texttt{as}\,\tau_2 \\
\text{paths} & p & ::= & x_\rho \mid p.i \\
\text{functions} & f & ::= & \rho{:}(\tau_1\, x_\rho) \xrightarrow{\epsilon} \tau_2 = \{s\} \mid \Lambda \alpha{:}\kappa \triangleright \gamma.f \\
\text{statements} & s & ::= & e \mid \texttt{return}\, e \mid s_1; s_2 \mid \texttt{if}\ (e)\ s_1\ \texttt{else}\ s_2 \mid \texttt{while}\ (e)\ s \mid \\
 & & & \rho{:}\{\tau\ x_\rho = e;\ s\} \mid \texttt{region}\langle\rho\rangle\ x_\rho\ s \mid \rho{:}\{\texttt{open}\,[\alpha, x_\rho] = e;\ s\} \mid s\,\texttt{pop}[\rho]
\end{array}
$$

Figure 3: Abstract Syntax of Core Cyclone

## 4 Formal Soundness

In a separate technical report [13], we have defined an operational model of core Cyclone, formalized the type system, and proven type soundness. Space constraints prevent us from duplicating the material here, so we summarize the salient details.

The core includes all of the features relevant to memory management, including stack allocation, dynamic region allocation, polymorphism, and existentials. The operational semantics is a small-step, deterministic rewriting relation ($\rightarrow$) from machine states to machine states. A machine state is a triple $(G, S, s)$ consisting of a garbage stack $G$, a stack $S$, and a statement $s$. The stacks are finite maps from region names ($\rho$) to regions ($R$), which in turn are finite maps from locations ($x$) to values ($v$). The garbage stack $G$ is a technical device used to record the deallocated storage so that the program stays closed despite dangling pointers. Note, however, that the abstract machine becomes stuck if the program attempts to read or write a location in the garbage stack. The primary goal of the formalism is to prove that well-typed programs cannot get stuck, and thus the garbage can be safely reclaimed at any point during the execution.

### 4.1 Syntax

Figure 3 gives BNF definitions for the syntax of the statements, expressions, and types for Core Cyclone. Constructors ($\tau$) define syntax for both types and regions. We use a kind discipline to determine whether a type variable represents a type ($\mathcal{T}$) or a region ($\mathcal{R}$).

Types include pairs ($\tau_1 \times \tau_2$) to model primitive structs. Like `structs`, pairs are passed by value (i.e., copy-in/copy-out). They cannot instantiate type variables because we do not duplicate polymorphic code and values of pair types are larger than other types. Types also include type variables, universal types, and existential types. The quantifiers can range over types or regions and include region constraints, which are used to specify partial orders on region lifetimes. A region constraint ($\gamma$) is a list of primitive constraints of the form $\epsilon <: \rho$ where $\epsilon$ is a region set, and $\rho$ is a region. Intuitively, the constraint means that if you can show any of the regions in $\epsilon$ are live, then you can assume $\rho$ is live. Region sets can include either region variables ($\rho$) or the `regions_of` a type variable. (We omit the `regions_of` for conciseness.) Finally, function types include a region set ($\epsilon$), which specifies the function's effect — the set of regions that must be live before calling the function.

Statements consist of expressions, return statements, composition, if-statements, and while-statements. In addition, they include blocks ($\rho : \{\tau\, x_\rho = e;\ s\}$) for declaring a new stack region and a variable within that region, dynamic region declarations ($\texttt{region}\langle\rho\rangle\, x_\rho\, s$), and a mechanism for opening values of existential type. Finally, statements include a special form "$s\,\texttt{pop}[\rho]$" which, when executed, evaluates $s$ to a terminal state and then deallocates the region $\rho$. This special form is not available at the source level, but is used internally by the abstract machine as a marker to indicate when regions should be deallocated.

Expressions include variables $x_\rho$ which double as locations. Each variable $x$ lives in a given region $\rho$; formally $x_\rho$ makes this fact explicit. Expressions also include integers, functions, pointer dereference, function calls, the address-of operator, and assignment as in C. In addition, expressions include polymorphic instantiation, pairs, projection, `rnew`, existential packages, region handles, and paths. Rather than model individual memory locations, paths provide a symbolic way to refer to a component of a compound object. For instance, if the location $x_\rho$ contains the value $((3,4),(5,6))$, then the path $x_\rho.1$ refers to $(3,4)$, and $x_\rho.1.2$ refers to 4.

## 4.2 Static Semantics

The most important typing judgment is the one for statements. It has the form:

$$\Delta; \Gamma; \gamma; \epsilon; \tau \vdash_{\text{stmt}} s$$

Here, $\Delta$ records the set of type and region variables that are in scope, $\Gamma$ records the set of value variables in scope along with their types, $\gamma$ records partial order constraints on region lifetimes from $\Delta$, $\epsilon$ records the static capability (i.e., which regions in $\Delta$ are considered live), and $\tau$ records the return type for the statement.

The inference rules for deriving this judgment allow dangling pointers to be manipulated, but they cannot be dereferenced. This is because the rules for pointer dereference require that the region into which the pointer refers is still live. To establish liveness, it suffices to show that the region name $\rho$ is in $\epsilon$. Often, this can be shown directly. When it cannot, we can try to find another region $\rho'$ in $\epsilon$ such that the constraint $\rho' :> \rho$ is in $\gamma$. The constraint ensures that $\rho$ outlives $\rho'$ and since $\rho'$ is live, $\rho$ must be also.

The other important judgment for statements is

$$\vdash_{\text{ret}} s$$

which asserts that the statement will not "fall off". Rather, if execution of the statement terminates, then the terminal state will be of the form $\texttt{return}(v)$ for some value $v$. This judgment is defined via a simple syntax-directed analysis.

Another important typing judgment is the one that allows us to assert that a garbage stack $G$ and stack $S$ can be described by the context $\Delta; \Gamma; \gamma$:

$$\vdash_{\text{heap}} (G, S) : \Delta; \Gamma; \gamma$$

Here, $\Delta$ is the set of region names that are bound in either $G$ or $S$; $\Gamma$ records the types of the locations bound in either $G$ or $S$; and $\gamma$ records the relative lifetimes of the regions in $S$. This judgment is used to connect assumptions that a statement might make with the reality of the current heap.

With these top-level judgments (informally) defined, we can state the Soundness Theorem for Core Cyclone:

**Theorem 4.1 (Soundness)** *If:*

1. $\vdash_{\text{heap}} (\emptyset, [\rho_H \mapsto R]) : \Delta; \Gamma; \gamma,$

*2.* $\vdash_{\text{ret}} s$*,*

*3.* $\Delta; \Gamma; \gamma; \{\rho_H\}; \text{int} \vdash_{\text{stmt}} s$*, and*

*4.* $s$ *contains no* `pop` *statements*

*then either* $(G, S, s)$ *runs forever or there exists a* $G'$*,* $R'$ *and* $i$ *such that* $(G, [\rho_H \mapsto R], s) \rightarrow^* (G', [\rho_H \mapsto R'], \texttt{return}(i))$

In plain English, if we start with an empty garbage heap, and a stack that contains a single heap region ($[\rho_H \mapsto R]$) that is well-formed, and if statement $s$ doesn't fall off, and $s$ is well formed with respect to the type of the initial heap and promises to return only integers, and $s$ does not contain `pop` statements, then the program cannot get stuck from type errors or dangling-pointer dereferences. Furthermore, if the program terminates, all of the regions it allocated will have been freed and the program will return an integer. The proof details are available in our companion technical report [13].

## 5 Implementing Cyclone Regions

The code-generation and run-time support for Cyclone regions is very simple. Heap and stack manipulation are exactly as in C. Dynamic regions are represented as linked lists of "pages" where each page is twice the size of the previous one. A region handle points to the beginning of the list and the current "allocation point" on the last page, where `rnew` or `rmalloc` place the next object. If there is insufficient space for an object, a new page is allocated. Region deallocation simply frees each page of the list.

When the garbage collector is included, dynamic-region list pages are acquired from the collector. The collector supports explicit deallocation, which we use to free regions. It is important to note that the collector simply treats the region pages as large objects. As they are always reachable from the stack, they are scanned and any pointers to heap-allocated objects are found, ensuring that these objects are preserved. The advantage of this interface is its simplicity, but at some cost: At collection time, every object in every dynamic region appears reachable, and thus no objects within (or reachable from) dynamic regions are reclaimed.

The code generator ensures that regions are deallocated even when their lifetimes end due to unstructured control flow. For each intra-procedural jump or `return`, it is easy to determine statically how many regions should be deallocated before transferring control. When throwing an exception, this is no longer the case. Therefore, we store region handles and exception handlers in an integrated list that operates in a last-in-first-out manner. When an exception is thrown, we traverse the list deallocating regions until we reach an exception handler. We then transfer control with `longjmp`. In this fashion, we ensure that a region is always deallocated when control returns.

## 6 Experimental Results

One of our central goals has been to minimize the number of required region annotations, to simplify both writing new code and porting existing code. To evaluate our design, we examined a large body of Cyclone code, including applications and libraries. In this section, we present our observations, finding that region annotations impose negligible burden on the application writer, but a somewhat larger burden on the library writer.

## 6.1 Application Code

To understand the overhead of porting C code to Cyclone, and particularly the impact of our region system, we ported a number of legacy applications and compared the differences in source code between the original and the Cyclone version. We picked several networking applications because they are part of the "systems" domain in which controlling data representation is important; these include a web server (`mini_httpd`), some web utilities (`http_get`, `http_post`, `http_ping`, and `http_load`), and a simple client (`finger`). We also used some computationally-intense, older C applications that make heavy use of arrays and pointers; these include `cfrac`, `grobner`, and `tile`. Finally, we ported the compression utilities `cacm` and `ncompress`.

We took two approaches to porting. First, we changed all the programs as little as possible to make them correct Cyclone programs. Then, for `cfrac` and `mini_httpd`, we *regionized* the code: We made functions more region polymorphic and, where possible, eliminated heap allocation in favor of dynamic region allocation with `rnew`. We also added compiler-checked "not null" annotations to pointer types where possible to avoid some null checks.

The results of our efforts are summarized in Table 1. For each benchmark program, we show the number of lines of C and Cyclone code, the differences between the two, the region annotations required, and a performance comparison of the Cyclone version with or without bounds/null checks. The `+` column indicates the number of lines added relative to C, and the `-` column is the lines removed. For the annotations, the *total* column is the number of individual region-related alterations, including per-variable annotations and occurrences of `region r {s}` and `rnew`. The *lines* column is the total number of lines in the file that changed due to these annotations. Though not our focus, we give performance estimates as well. The times are the median running times (n=21) on a 750MHz PentiumIII with 256MRam running Linux kernel 2.2.16-12. The percentage for the Cyclone programs is time relative to the C version.

There are two interesting results regarding the difficulty of minimal-porting. First, the overall changes in the programs are relatively small — less than 10% of the program code needed to be changed. The vast majority of the overall differences arise from pointer-syntax alterations. These changes are typically easy to make — e.g., the type of strings are changed from `char *` to `char ?`.

The most encouraging result is that the number of region annotations is small: only 124 changes in total for more than 18,000 lines of code, which account for roughly 6% of the total changes. The majority of these changes were completely trivial, e.g., many programs required adding $\rho_H$ annotations to `argv` so that arguments could be stored in global variables. The program that required the most changes was `grobner`. Interestingly, the majority of these changes arose from the fact that in one place a stack pointer was being stored in a `struct` type. We therefore parameterized the `struct` definition with a region variable, and this parameterization then propagated through the rest of the code. However, the default annotation still worked in many cases: out of 133 total variable declarations of the parameterized `struct` type, only 38 required annotations, or 28%.

The cost of porting a program to use dynamic regions was also reasonable; in this case roughly 13% of the total diffs were region-related. For the web server, we were able to eliminate heap allocation entirely. Because it is event-driven, handling each request as it comes in, we changed the main handler function to create a dynamic region and then pass the region handle to its subroutines in a request structure. After the request is serviced, the region is freed. The majority of the overall changes arose from moving global variables into the request structure and adding the structure as a parameter to various functions. This request structure is parameterized by a region, so many of the functions need annotations to connect the region of the request structure to that of another argument or return value.

We were less successful in regionizing `cfrac`. As in the web server, we changed many functions to allocate using region-handle parameters. It was easy to do dynamic region allocation and deallocation as part of the algorithm's main iteration, but for large inputs, it was difficult to keep regions from

| Program | LOC | | diffs | | annotations | | performance | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | C | Cyc | + | - | total | lines | C time (s) | checked(s) | % | unchecked(s) | % |
| cacm | 340 | 359 | 42 | 23 | 0 | 0 | 1.77 | 3.49 | 97% | 3.03 | 71% |
| cfrac | 4218 | 4214 | 132 | 136 | 2 | 2 | 2.61 | 17.07 | 554% | 17.07 | 554% |
| finger | 158 | 161 | 18 | 15 | 3 | 3 | 0.58 | 0.55 | -5% | 0.48 | -17% |
| grobner | 3244 | 3377 | 438 | 305 | 71 | 40 | 0.07 | 0.20 | 186% | 0.20 | 186% |
| http_get | 529 | 529 | 36 | 36 | 4 | 4 | 0.28 | 0.28 | 0% | 0.28 | 0% |
| http_load | 2072 | 2057 | 115 | 130 | 15 | 13 | 89.37 | 90.22 | 1% | 90.19 | 1% |
| http_ping | 1072 | 1081 | 30 | 21 | 1 | 1 | 0.28 | 0.28 | 0% | 0.28 | 0% |
| http_post | 607 | 608 | 42 | 41 | 8 | 8 | 0.16 | 0.16 | 0% | 0.16 | 0% |
| matxmult | 57 | 48 | 3 | 12 | 3 | 1 | 1.38 | 1.83 | 32% | 1.38 | 0% |
| mini_httpd | 3005 | 3022 | 233 | 216 | 4 | 4 | 3.71 | 3.85 | 4% | 3.86 | 4% |
| ncompress | 1964 | 1982 | 120 | 102 | 10 | 9 | 0.20 | 0.39 | 95% | 0.38 | 90% |
| tile | 1345 | 1366 | 145 | 124 | 2 | 2 | 0.48 | 1.05 | 116% | 0.99 | 104% |
| total | 18611 | 18804 | 1354 | 1161 | 124 | 86 | - | - | - | - | - |

*"regionized" versions of benchmarks*

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| cfrac | 4218 | 4110 | 501 | 528 | 158 | 107 | 2.61 | 10.07 | 286% | 8.80 | 237% |
| mini_httpd | 3005 | 2967 | 500 | 522 | 88 | 54 | 3.71 | 3.83 | 3% | 3.82 | 3% |
| total | 7223 | 7174 | 1001 | 1050 | 246 | 161 | - | - | - | - | - |

Table 1: Porting C code to Cyclone

growing large before deallocation. We conclude that garbage collection is a better match for this code, but others have had more success with regions [11].

As for performance, we achieve near-zero overhead for network or I/O bound applications such as the http clients and servers, but we pay a considerable run-time penalty for processor-intensive benchmarks such as the compression tools. The unusually high overhead for the unregionized cfrac appears due to poor code generation for `*p++` where `p` has type `int?`. The regionized port avoids such expressions. We believe much of the overhead is due to array representation, not regions. We address this issue further in Section 8.

## 6.2 Library Code

We have ported a significant subset of the C and Ocaml libraries to Cyclone. Two illustrative cases are the Cyclone list and string libraries, ported from Ocaml and C respectively. Table 2 summarizes the region annotations in the interfaces and implementations of these libraries. As a rough measure of the effectiveness of default region annotations, we also provide results for "maximally annotated" versions of the interfaces (list-max.h and string-max.h, respectively). The *proto* column lists the number of region type annotations that were necessary in function prototypes; the *rnew* column lists the number of uses of `rnew`, and the *region* column lists the number of uses of dynamic regions.

We found that library code requires more region annotations than application code, but most of these annotations are for the sake of convenience and generality rather than necessity. Library functions that perform allocation tend to come in two flavors: a heap allocating function that has the same signature as the corresponding C or Ocaml function, and a region version that takes an additional region handle. Most of the annotations occur in the latter, and so were made for the sake of the convenience of using of the libraries with arbitrary regions. Most of the changes are to function prototypes; no explicit region annotations were necessary in the bodies of functions. The maximally annotated interfaces require 2-2.4 times more region annotations; that is, the default region annotations suffice 50-60% of the time. Most of the non-default region annotations were

| | LOC | proto | rnew | region |
|---|---|---|---|---|
| string.h | 139 | 57 | 0 | 0 |
| string-max.h | 139 | 135 | 0 | 0 |
| string.cyc | 739 | 68 | 14 | 2 |
| list.h | 364 | 85 | 0 | 0 |
| list-max.h | 364 | 171 | 0 | 0 |
| list.cyc | 819 | 74 | 38 | 0 |

Table 2: Region annotations in libraries

needed to express a "same-region" relationship between arguments and return types or to allow the function to allocate into an arbitrary region; the remainder were needed in type definitions. Moreover, no effect annotations whatsoever were necessary.

Most importantly, our applications, such as the compiler, use the libraries extensively and region instantiation is implicit throughout them. The vast majority of library calls in ported C code require no changes; `malloc`, `realloc`, `memcpy`, etc., are essentially the only exceptions.

## 7 Related Work

In this paper, we have concentrated on the region-based type system for Cyclone, which naturally supports C-style stack allocation, conventional heap allocation, and dynamic region allocation. We feel that Cyclone is a unique and promising point in the programming-language design-space, but many other systems share some of these features.

**Making C Safe** Many systems, including (but not limited to) LCLint [9, 8], SLAM [3], Safe-C [2], and CCured [19] aim to make C code safe. Some of these systems, such as LCLint, are meant to be static bug-finding tools. Like Cyclone, they usually require restricted coding idioms or additional annotations, but unlike Cyclone, they offer no soundness guarantees. In this way, these static tools reduce false positives. In contrast, Cyclone uses a combination of static analysis (for memory management) and run-time checks (for bounds violations) to minimize false positives.

Other systems, such as Safe-C and CCured, ensure soundness by rewriting the code and adding run-time checks. The primary advantage of these systems is that they require (almost) no changes to the C code, unlike Cyclone. However, they do not preserve the same data representations and lifetimes for objects. Furthermore, memory errors are caught at run-time instead of compile time. For instance, when an object is freed under CCured, the (entire) storage is not immediately reclaimed, but rather marked as inaccessible. Subsequent accesses check the mark and signal an error when the object is dereferenced. Ultimately, the "mark" is reclaimed with a garbage collector to avoid leaks. Furthermore, CCured may implicitly move some stack-allocated objects to the heap to avoid dangling-pointer dereferences.

**Static Regions** Tofte and Talpin's seminal work [22] on implementing ML with regions provides the foundation for regions in the ML Kit [21]. Programming with the Kit is convenient, as the compiler automatically infers all region annotations. However, small changes to a program can have drastic, unintuitive effects on object lifetimes. Thus, to program effectively, one must understand the analysis and try to control it indirectly by using certain idioms [21]. More recent work for the ML Kit includes optional support for garbage collection within regions [14].

A number of extensions to the basic Tofte-Talpin framework can avoid the constraints of LIFO region lifetimes. As examples, the ML Kit includes a reset-region primitive [21]; Aiken et al. provide an analysis to free some regions early [1]; and Walker et al. [23, 24] propose general systems for freeing regions based on linear types. All of these systems are more expressive than our framework. For instance, the ideas in the Capability Calculus were used to implement type-safe garbage collectors *within* a language [25, 18]. However, these systems were not designed for source-level programming. Rather, they were designed as compiler intermediate languages or analyses and can thus ignore user issues such as minimizing annotations or providing control to the user.

Two other recent projects, Vault [7] and the work of Henglein et al. [15] aim to provide convenient, source-level control over memory management using regions. Vault's powerful type system allows a region to be freed before it leaves scope and its types can enforce that code *must* free a region. To do so, Vault restricts region aliasing and tracks more fine-grained effects. As a result, programming in Vault requires more annotations. Nevertheless, we find Vault an extremely promising direction and hope to adapt some of these ideas to Cyclone. Henglein et al. [15] have designed a flexible region system that does not require LIFO behavior. However, the system is monomorphic and first-order; it is unclear how to extend it to support polymorphism or existential types.

**Regions in C** Perhaps the most closely related work is Gay and Aiken's RC [11] compiler and their earlier system, C@ [10]. As they note, region-based programming in C is an old idea; they contribute language support for efficient reference counting to detect if a region is deallocated while there remain pointers to it (that are not within it). This dynamic system has no *a priori* restrictions on regions' lifetimes and a pointer can point anywhere, so the RC approach can encode more memory-management idioms. Like Cyclone, they provide pointer annotations. These annotations are never required, but they are often crucial for performance because they reduce the need for reference-counting. One such annotation is very similar to our notion of region subtyping.

RC uses reference counting only for dynamic regions. In fact, one annotation enforces that a pointer never points into a dynamic region, so no reference counting is needed. As a result, RC allows dangling pointers into the stack or heap. Other kinds of type errors also remain. Indeed, we found a number of array-bounds bugs in the benchmarks used to evaluate RC, such as `grobner`. Finally, RC cannot support the kind of polymorphism that Cyclone does because the RC compiler must know statically which objects are pointers.

In summary, some of these systems are more convenient to use than Cyclone (e.g., CCured and the MLKit) but take away control over memory management. Some of the static systems (e.g., the Capability Calculus) provide more powerful region constructs, but were designed as intermediate languages and do not have the programming convenience of Cyclone. Other systems (e.g., RC, Safe-C) are more flexible but offer no static guarantees.

## 8 Future Work

A great deal of work remains to achieve our goals of providing a tool to easily move legacy code to a type-safe environment and providing a type-safe language for building systems where control over data representations and memory management is an issue.

In the near future, we hope to incorporate support for deallocating dynamic regions early. We have experimented briefly with linear type systems in the style of the Capability Calculus or Vault, but have found that this approach is generally too restrictive, especially in the context of exceptions. Instead, we are currently developing a traditional intra-procedural flow analysis to track region aliasing and region lifetimes. Again, for the inter-procedural case, we expect to add support for explicit annotations, and to use experimental evidence to drive the choice of defaults.

We also expect to incorporate better support for first-class regions, in the style of RC. The goal is to give programmers a sufficient range of options that they can use the statically checked regions most of the time, but fall back on the dynamically checked regions when needed.

In addition to enhancements to the region framework, work is needed in other areas. For instance, we have seen run-time overheads ranging from 1 to 3x for the benchmarks presented here. For compute-intensive micro-benchmarks, we have seen performance range from 1 to 10x, depending on the architecture and C compiler. We are currently working to identify the bottlenecks, but an obvious problem is that we perform all array-bounds checks at run-time. For example, when we disabled bounds checks for a matrix multiply program, overhead dropped from 30% to 0%. Fortunately, array-bounds-check elimination is a well-studied issue, and we feel confident that we can adapt a simple, but effective approach to avoid much of this cost.

The other key area we are investigating is data representation: To support dynamically sized arrays and array-bounds checks, we tag such arrays with implicit size information. Similarly, to support type-safe, discriminated unions, we add implicit tags. We are adapting ideas from DML [27] and Xanadu [26] to give programmers control over the placement of these tags. We hope this will make it easier to interface with legacy C code or devices that do not expect these tags on the data. However, we have found that the DML framework does not easily extend to imperative languages such as C. In particular, there are subtle issues involving existential types and the address-of (&) operator [12].

## Acknowledgments

We would like to thank Steve Zdancewic and Jeff Vinocur for help in proofreading this manuscript.

## References

[1] A. Aiken, M. Fähndrich, and R. Levien. Better static memory management: Improving region-based analysis of higher-order languages. In *ACM Conference on Programming Language Design and Implementation*, pages 174–185, La Jolla, CA, 1995.

[2] T. M. Austin, S. E. Breach, and G. S. Sohi. Efficient detection of all pointer and array access errors. In *ACM Conference on Programming Language Design and Implementation*, pages 290–301, June 1994.

[3] T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN 2001, Workshop on Model Checking of Software*, volume 2057 of *Lecture Notes in Computer Science*, pages 103–122. Springer-Verlag, May 2001.

[4] H.-J. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, 18(9):807–820, 1988.

[5] K. B. Bruce, L. Cardelli, and B. C. Pierce. Comparing object encodings. *Information and Computation*, 155:108–133, 1999.

[6] Cyclone user's manual. Technical Report 2001-1855, Department of Computer Science, Cornell University, Nov. 2001. Current version at `http://www.cs.cornell.edu/projects/cyclone/`.

[7] R. DeLine and M. Fähndrich. Enforcing high-level protocols in low-level software. In *ACM Conference on Programming Language Design and Implementation*, pages 59–69, Snowbird, UT, June 2001.

[8] D. Evans. LCLint user's guide. http://lclint.cs.virginia.edu/guide/.

[9] D. Evans. Static detection of dynamic memory errors. In *ACM Conference on Programming Language Design and Implementation*, pages 44–53, Philadelphia, Pennsylvania, May 1996.

[10] D. Gay and A. Aiken. Memory management with explicit regions. In *ACM Conference on Programming Language Design and Implementation*, pages 313–323, Montreal, Canada, June 1998.

[11] D. Gay and A. Aiken. Language support for regions. In *ACM Conference on Programming Language Design and Implementation*, pages 70–80, Snowbird, UT, June 2001.

[12] D. Grossman. Existential types for imperative languages. Oct. 2001. Submitted for publication. Available at `http://www.cs.cornell.edu/home/danieljg/papers/exists_imp.pdf`.

[13] D. Grossman, G. Morrisett, Y. Wang, T. Jim, M. Hicks, and J. Cheney. Formal type soundness for Cyclone's region system. Technical Report 2001-1856, Department of Computer Science, Cornell University, Nov. 2001. Available at `http://www.cs.cornell.edu/home/danieljg/papers/cyclone_regions_tr.pdf`.

[14] N. Hallenberg. Combining garbage collection and region inference in the ML Kit. Master's thesis, Department of Computer Science, University of Copenhagen, 1999.

[15] F. Henglein, H. Makholm, and H. Niss. A direct approach to control-flow sensitive region-based memory management. In *Third International Conference on Principles and Practice of Declarative Programming*, Firenze, Italy, Sept. 2001.

[16] Y. Minamide, G. Morrisett, and R. Harper. Typed closure conversion. In *Twenty-Third ACM Symposium on Principles of Programming Languages*, pages 271–283, St. Petersburg, Jan. 1996.

[17] J. Mitchell and G. Plotkin. Abstract types have existential type. *ACM Trans. on Programming Languages and Systems*, 10(3):470–502, 1988. Preliminary version in Twelfth ACM Symposium on Principles of Programming Languages, 1985.

[18] S. Monnier, B. Saha, and Z. Shao. Principled scavenging. In *ACM Conference on Programming Language Design and Implementation*, pages 81–91, Snowbird, UT, June 2001.

[19] G. C. Necula, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy code. In *Twenty-Ninth ACM Symposium on Principles of Programming Languages*, Portland, OR, Jan. 2002. To appear.

[20] M. Tofte and L. Birkedal. A region inference algorithm. *ACM Transactions on Progamming Languages and Systems*, 20(4):734–767, July 1998.

[21] M. Tofte, L. Birkedal, M. Elsman, N. Hallenberg, T. H. Olesen, and P. Sestoft. Programming with regions in the ML Kit (for version 4). Technical report, IT University of Copenhagen, Sept. 2001.

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

[23] D. Walker, K. Crary, and G. Morrisett. Typed memory management in a calculus of capabilities. *ACM Transactions on Progamming Languages and Systems*, 24(4):701–771, July 2000.

[24] D. Walker and K. Watkins. On regions and linear types. In *ACM International Conference on Functional Programming*, pages 181–192, Sept. 2001.

[25] D. C. Wang and A. W. Appel. Type-preserving garbage collectors. In *Twenty-Eighth ACM Symposium on Principles of Programming Languages*, pages 166–178, London, United Kingdom, Jan. 2001.

[26] H. Xi. Imperative programming with dependent types. In *Proceedings of 15th IEEE Symposium on Logic in Computer Science*, pages 375–387, Santa Barbara, June 2000.

[27] H. Xi and F. Pfenning. Dependent types in practical programming. In *Proceedings of ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 214–227, San Antonio, January 1999.

## 9 Full Formal Language

We now present our full formal language and a syntactic proof of type soundness. We begin with a presentation of the language (syntax, dynamic semantics, and static semantics) and then proceed to the proof. The proof appears in "bottom-up" order, so we try to give some intuition for each lemma's purpose, but in the end, the most important intuition is to realize why each part of the static semantics is needed for a strong enough induction hypothesis to establish soundness.

The dynamic semantics defines a small-step rewriting relation from machine states to machine states. A machine state has the form $(G, S, s)$, where $G$ is the garbage heap, $S$ is the live heap, and $s$ is the current program. No rule allows accessing $G$'s contents; it is just a technical device to keep $S$ and $s$ closed. Both $G$ and $S$ are actually stacks of regions. That is, we impose more order on $G$ and $S$ than on a conventional heap.

The most relevant (but quite minor) aspect of Cyclone that we do not bother to model is that it is impossible to allocate into a "stack region" after the region has been created. We can make this fact true by not allowing initial programs to have explicit region handles, but we do not establish a strong enough induction hypothesis to prove—absolutely formally—that this restriction suffices.

### 9.1 Syntax

Figure 4 presents the syntax for the full formal language.

Types and regions are joined into a single syntactic class of constructors ($\tau$) and distinguished using a kind structure. In particular, if $\tau$ has kind $\mathcal{T}$ then it is a type, whereas if $\tau$ has kind $\mathcal{R}$ then it is a region name. Note that the only constructors of kind $\mathcal{R}$ are variables. We use $\rho$ to range over region variables (also known as region names) and we use $\alpha$ to range over type variables, but we also use $\alpha$ when the kind is unknown.

In addition, types may be boxed types. Boxed types ($\mathcal{B}$) are those types whose values fit into a machine word. Both int and pointer types ($\tau@\rho$) are boxed types, as are existential or universal types that are boxed. In particular, tuples are not considered to be boxed types. Type variables ($\alpha$) may only be introduced as boxed types. The box-type distinction is made in Cyclone because all box-types are represented uniformly; compilation thus does not require code duplication or run-time type information.

A region-set ($\epsilon$) is a set of type variables. We identify region-sets up to associativity, commutativity, idempotence of $\emptyset$, and removal of repeated elements. In other words, we treat them as abstract sets and exploit any necessary algebraic laws as a result. We also blur the distinction between $\alpha$ and $\{\alpha\}$ as syntactically convenient. In function types, a region-set serves as an *effect* and in a typing context it serves as a *capability*, so we use the terms "region-set", "effect", and "capability" interchangeably. We write $\alpha \uplus \epsilon$ for the effect $\alpha \cup \epsilon$ where $\alpha \notin \epsilon$. Intuitively, the meaning of $\alpha \in \epsilon$ depends on the kind of $\alpha$. For region variables, it means the variable is in the effect (or capability) that the set represents. For type variables, it means all the regions that $\alpha$ "mentions" are in the effect (or capability). This intuition is formalized with the definition of regions$(\tau)$ and the definition of substitution, which we present later.

A region-constraint ($\gamma$) is a collection of atomic constraints of the form $\epsilon <: \rho$. The order is unimportant. The intuitive meaning is that every element of $\epsilon$ *outlives* $\rho$. Therefore, liveness of $\rho$

| | | | |
|---|---|---|---|
| kinds | $\kappa$ | ::= | $\mathcal{T} \mid \mathcal{B} \mid \mathcal{R}$ |
| type and region vars | $\alpha, \rho$ | | |
| region sets | $\epsilon$ | ::= | $\emptyset \mid \alpha \mid \epsilon_1 \cup \epsilon_2$ |
| region constraints | $\gamma$ | ::= | $\emptyset \mid \gamma, \epsilon <: \rho$ |
| constructors | $\tau$ | ::= | $\alpha \mid \text{int} \mid \tau_1 \xrightarrow{\epsilon} \tau_2 \mid \tau_1 \times \tau_2 \mid \tau@\rho \mid \text{handle}(\rho) \mid$ |
| | | | $\forall\alpha{:}\kappa \triangleright \gamma.\tau \mid \exists\alpha{:}\kappa \triangleright \gamma.\tau$ |
| | | | |
| identifiers | $x$ | | |
| expressions | $e$ | ::= | $x_\rho \mid v \mid e\langle\tau\rangle \mid (e_1, e_2) \mid e.i \mid *e \mid \texttt{new}(e_1)e_2 \mid$ |
| | | | $e_1(e_2) \mid \&e \mid e_1 = e_2 \mid \texttt{call}\{s\} \mid \texttt{pack}\,[\tau_1, e]\,\texttt{as}\,\tau_2$ |
| values | $v$ | ::= | $i \mid f \mid \&p \mid \texttt{region}(\rho) \mid (v_1, v_2) \mid \texttt{pack}\,[\tau_1, v]\,\texttt{as}\,\tau_2$ |
| paths | $p$ | ::= | $x_\rho \mid p.i$ |
| functions | $f$ | ::= | $\rho{:}(\tau_1\, x_\rho) \xrightarrow{\epsilon} \tau_2 = \{s\} \mid \Lambda\alpha{:}\kappa \triangleright \gamma.f$ |
| statements | $s$ | ::= | $e \mid \texttt{return}\,e \mid s_1; s_2 \mid \texttt{if}\ (e)\ s_1\ \texttt{else}\ s_2 \mid \texttt{while}\ (e)\ s \mid$ |
| | | | $\rho{:}\{\tau\ x_\rho = e;\ s\} \mid \texttt{region}\langle\rho\rangle\ x_\rho\ s \mid$ |
| | | | $\rho{:}\{\texttt{open}\,[\alpha, x_\rho] = e;\ s\} \mid s\,\texttt{pop}[\rho]$ |
| regions | $R$ | ::= | $\emptyset \mid R[x \mapsto v]$ |
| region stacks | $G, S$ | ::= | $\emptyset \mid S[\rho \mapsto R]$ |
| | | | |
| type variable contexts | $\Delta$ | ::= | $\bullet \mid \Delta, \alpha{:}\kappa$ |
| value variable contexts | $\Gamma$ | ::= | $\bullet \mid \Gamma, x_\rho : \tau$ |
| contexts | $C$ | ::= | $\Delta; \Gamma; \gamma; \epsilon$ |

Figure 4: Syntax of Core Cyclone IL

suffices to establish the liveness of any element in $\epsilon$. As we show later, we use $\gamma$ to define a semantic outlives relationship $\Rightarrow$ that is essentially the reflexive, transitive closure of the constraints in $\gamma$.

The constructors $\tau$ include:

- $\alpha$, which represents a type variable or a region name, depending on its kind (which is determined from a type-variable context $\Delta$)
- int, which describes integers
- $\tau_1 \xrightarrow{\epsilon} \tau_2$, which describes functions from $\tau_1$ to $\tau_2$ with effect $\epsilon$. Note that our "unboxed" product types make the restriction to a single parameter a non-issue: Passing a pair does not require allocation.
- $\tau_1 \times \tau_2$, which describes "unboxed" pairs. That is, the "size" of such an object is the "size" of $\tau_1$ plus the "size" of $\tau_2$. Such values are "passed by copy", as is clear from the dynamic semantics.
- $\tau@\rho$, describes pointers to $\tau$ values living in the region described by $\rho$. We use @ because, as in Cyclone, these values are not allowed to be NULL. (We are not modeling NULL.)
- handle($\rho$) describes a handle for (allocating into) the region described by $\rho$.
- $\forall\alpha{:}\kappa \triangleright \gamma.\tau$ describes polymorphic values. Intuitively, $\alpha$ must be instantiated with a type such that the constraints in $\gamma$ are true. (Presumably $\gamma$ is empty or mentions $\alpha$, but neither is actually required.) The syntax for expressions happens to restrict polymorphism to functions. That is, $\tau$ will either be a function type or another universal type. Note that $\alpha$ could stand for a region name or a boxed type, as determined by $\kappa$.
- $\exists\alpha{:}\kappa \triangleright \gamma.\tau$ describes existential packages. Intuitively, the "witness type" in the package is such that instantiating $\alpha$ with it makes the constraints in $\gamma$ true. As with universal types, $\alpha$ could range over region names or boxed types.

The expressions $e$ include:

- $x_\rho$, identifiers. We use identifiers for two purposes: identifiers in programs (as usual) and references into the heap. This reuse reduces the number of language constructs; by implicitly relying on $\alpha$-conversion when we allocate space for $x_\rho$ at run-time due to a binding occurrence, we can assume $x_\rho$ has never been used before. The $\rho$ describes the region where $x$ is (or will be) allocated. We just find it convenient to annotate identifiers this way, rather than looking up "region of $x$" from the context.
- $i$, integer constants.
- $f$, function bodies: $\Lambda\alpha{:}\kappa \triangleright \gamma.f$ abstracts $\alpha$ in $f$; if well-formed, it has a universal type as described above. The innermost $f$ has the form $\rho{:}(\tau_1\, x_\rho) \xrightarrow{\epsilon} \tau_2 = \{s\}$, which has type $\tau_1 \xrightarrow{\epsilon} \tau_2$. The function body is $s$. In the body, $x_\rho$ refers to the function parameter, which has type $\tau_1$.
- `region`($\rho$), a handle into the region named $\rho$. Intuitively, these constructs do not appear in source programs. When we create a "growable region" $\rho$, we bind an identifier to `region`($\rho$).
- $e\langle\tau\rangle$, type application. The term $e$ should evaluate to a polymorphic expression.
- $(e_1, e_2)$, pairs.
- $e.i$, pair projection. $i$ must be 1 or 2.

- $*e$, pointer dereference, as in C.
- `new`$(e_1)e_2$, allocates $e_2$ into a fresh location in the region for which the result of $e_1$ is a handle.
- $e_1(e_2)$, function call, as in C.
- $\&e$, address-of, as in C.
- $e_1 = e_2$, assignment, as in C.
- `call`$\{s\}$ need not appear in source programs. A function call translates to this construct, so that the call-stack is represented in the syntax of the program as it is rewritten.
- `pack` $[\tau_1, e]$ `as` $\tau_2$, an existential package with type $\tau_2$ and witness type $\tau_1$.

The semantics distinguishes left-hand-sides and right-hand-sides, as in C. Top-level expressions are right-hand-sides. A subexpression inherits the "sidedness" of its containing expression, except:

- In $\&e$, $e$ is a left-hand-side (see **DR9**, below).
- In $*e$, $e$ is always a right-hand-side (see **DL2**, below).
- In $e_1 = e_2$, $e_1$ is a left-hand-side (see **DR9**, below).

These rules are just the rules for C, which carefully distinguishes left and right, adapted to our formal language. The point is that the evaluation of $\&e_1$ and $e_1 = e_2$ is not concerned with the value at the location $e_1$, but rather with the location itself. Values ($v$) are the terminal results of evaluating right-hand sides. Paths ($p$) are the terminal results of evaluating left-hand sides. A path is just an identifier followed by some number of projections. Most expression forms make no sense as left-hand-sides (e.g., integer constants); the static semantics forbids them.

Most of the statement forms (expressions, return, sequence, conditional, loop) are just as in C. The three local-binding forms are less familiar:

- $\rho{:}\{\tau\ x_\rho = e;\ s\}$ declares local variable $x$, whose scope is $s$. $x$ lives in a "stack region" $\rho$, which is deallocated when control leaves $s$. If $\tau$ is a pair type, then we can model multiple local variables in the same region with pairing. The outer $\rho$ is redundant syntax, but it serves to emphasize that we are binding a region name and an identifier.
- `region`$\langle\rho\rangle\ x_\rho\ s$ creates a new "growable" region and binds $x_\rho$ to its handle. As with the previous form, we are binding $\rho$ and $x$.
- $\rho{:}\{$`open` $[\alpha, x_\rho] = e;\ s\}$ is just like the first form except that we also do an existential "unpack" or "open". In $s$, $\alpha$ is bound and stands for the witness type. $e$ should, of course, evaluate to an existential package. This form binds $\rho$, $\alpha$, and $x$.

Finally, $s$ `pop`$[\rho]$ should not appear in source programs. Its meaning is to execute $s$ and then deallocate the region $\rho$. We will prove that $\rho$ is always the most-recently-allocated region and that deallocating it does not violate type soundness.

A stack ($G$ or $S$) is an *ordered* map from region names ($\rho$) to regions $R$. In other words, we have partitioned the heap into an ordered live heap $S$ and the garbage heap $G$. The order on $G$ is irrelevant, but there was no reason to use a different construct for it. A region $R$ is an unordered map from locations ($x$ as explained above) to values $v$.

A typing context includes a $\Delta$ (giving type variables kinds), a $\Gamma$ (giving value variables types), a $\gamma$ (reflecting the known outlives ordering), and an $\epsilon$ (reflecting the current capability).

$$(\mathbf{DS1})\quad (G,S,v;s)\xrightarrow{\text{stmt}}(G,S,s)$$

$$(\mathbf{DS2})\quad (G,S,\texttt{if}\ (0)\ s_1\ \texttt{else}\ s_2)\xrightarrow{\text{stmt}}(G,S,s_2)$$

$$(\mathbf{DS3})\quad (G,S,\texttt{if}\ (i)\ s_1\ \texttt{else}\ s_2)\xrightarrow{\text{stmt}}(G,S,s_1)\qquad (i\neq 0)$$

$$(\mathbf{DS4})\quad (G,S,\texttt{while}\ (e)\ s)\xrightarrow{\text{stmt}}(G,S,\texttt{if}\ (e)\ \{s;\texttt{while}\ (e)\ s\}\ \texttt{else}\ 0)$$

$$(\mathbf{DS5})\quad (G,S,\texttt{return}\,v;s)\xrightarrow{\text{stmt}}(G,S,\texttt{return}\,v)$$

$$(\mathbf{DS6})\quad (G,S[\rho\mapsto R],\texttt{return}\,v\,\texttt{pop}[\rho])\xrightarrow{\text{stmt}}(G[\rho\mapsto R],S,\texttt{return}\,v)$$

$$(\mathbf{DS7})\quad (G,S[\rho\mapsto R],v\,\texttt{pop}[\rho])\xrightarrow{\text{stmt}}(G[\rho\mapsto R],S,v)$$

$$(\mathbf{DS8a})\quad (G,S,\rho{:}\{\tau\,x_\rho=v;\ s\})\xrightarrow{\text{stmt}}(G,S[\rho\mapsto\{x\mapsto v\}],s\,\texttt{pop}[\rho])\quad (\rho\notin \mathrm{Dom}(S)\cup\mathrm{Dom}(G))$$

$$(\mathbf{DS8b})\quad (G,S,\texttt{region}\langle\rho\rangle\,x_\rho\,s)\xrightarrow{\text{stmt}}(G,S[\rho\mapsto\{x\mapsto\texttt{region}(\rho)\}],s\,\texttt{pop}[\rho])\quad (\rho\notin \mathrm{Dom}(S)\cup\mathrm{Dom}(G))$$

$$(\mathbf{DS8c})\quad (G,S,\rho{:}\{\texttt{open}\,[\alpha,x_\rho]=\texttt{pack}\,[\tau_1,v]\,\texttt{as}\,\exists\alpha{:}\kappa\triangleright\gamma.\tau_2;s\})\xrightarrow{\text{stmt}}(G,S,\rho{:}\{\tau_2[\tau_1/\alpha]\,x_\rho=v;\ s[\tau_1/\alpha]\})$$

$$(\mathbf{DS9})\quad \frac{(G,S,e)\xrightarrow{\text{rhs}}(G',S',e')}{\begin{array}{c}(G,S,e)\xrightarrow{\text{stmt}}(G',S',e')\\ (G,S,\texttt{return}\,e)\xrightarrow{\text{stmt}}(G',S',\texttt{return}\,e')\\ (G,S,\texttt{if}\ (e)\ s_1\ \texttt{else}\ s_2)\xrightarrow{\text{stmt}}(G',S',\texttt{if}\ (e')\ s_1\ \texttt{else}\ s_2)\\ (G,S,\rho{:}\{\tau\,x_\rho=e;s\})\xrightarrow{\text{stmt}}(G',S',\rho{:}\{\tau\,x_\rho=e';s\})\\ (G,S,\rho{:}\{\texttt{open}\,[\alpha,x_\rho]=e;s\})\xrightarrow{\text{stmt}}(G,S,\rho{:}\{\texttt{open}\,[\alpha,x_\rho]=e';s\})\end{array}}$$

$$(\mathbf{DS10})\quad \frac{(G,S,s)\xrightarrow{\text{stmt}}(G',S',s')}{\begin{array}{c}(G,S,s;s_2)\xrightarrow{\text{stmt}}(G',S',s';s_2)\\ (G,S,s\,\texttt{pop}[\rho])\xrightarrow{\text{stmt}}(G',S',s'\,\texttt{pop}[\rho])\end{array}}$$

Figure 5: Statement Rewriting

$$(\mathbf{DR1})\quad (G, S_1[\rho \mapsto R]S_2, x_\rho) \xrightarrow{\text{rhs}} (G, S_1[\rho \mapsto R]S_2, R(x))$$

$$(\mathbf{DR2})\quad (G, S, (\Lambda\alpha{:}\kappa \triangleright \gamma.f)\langle\tau\rangle) \xrightarrow{\text{rhs}} (G, S, f[\tau/\alpha])$$

$$(\mathbf{DR3})\quad (G, S, (v_1, v_2).i) \xrightarrow{\text{rhs}} (G, S, v_i)$$

$$(\mathbf{DR4})\quad (G, S, *(\&p)) \xrightarrow{\text{rhs}} (G, S, p)$$

$$(\mathbf{DR5})\quad (G, S_1[\rho \mapsto R]S_2, \texttt{new}(\texttt{region}(\rho))\, v) \xrightarrow{\text{rhs}} (G, S_1[\rho \mapsto R[x \mapsto v]]S_2, \&x_\rho) \qquad (x \notin \text{Dom}(R))$$

$$(\mathbf{DR6})\quad (G, S, (\rho{:}(\tau'\, x_\rho) \xrightarrow{\epsilon} \tau = \{s\})(v)) \xrightarrow{\text{rhs}} (G, S, \texttt{call}\{\rho{:}\{\tau'\, x_\rho = v;\ s\}\})$$

$$(\mathbf{DR7})\quad (G, S, \texttt{call}\{\texttt{return}\, v\}) \xrightarrow{\text{rhs}} (G, S, v)$$

$$(\mathbf{DR8})\quad \begin{array}{l} (G, S_1[\rho \mapsto R[x \mapsto v_a]]S_2, x_\rho.i_1.i_2 \ldots i_n = v_b) \xrightarrow{\text{rhs}} \\ \qquad (G, S_1[\rho \mapsto R[x \mapsto \text{update}(v_a, [i_1, i_2, \ldots, i_n], v_b)]]S_2, v_b) \\[1ex] \quad \text{where} \quad \begin{array}{l} \text{update}(v_a, [\,], v_b) = v_b \\ \text{update}((v_1, v_2), [1, i_1, \cdots, i_n], v) = (\text{update}(v_1, [i_1, \cdots, i_n], v_b), v_2) \\ \text{update}((v_1, v_2), [2, i_1, \cdots, i_n], v) = (v_1, \text{update}(v_2, [i_1, \cdots, i_n], v_b)) \end{array} \end{array}$$

$$(\mathbf{DR9})\quad \frac{(G, S, e) \xrightarrow{\text{lhs}} (G', S', e')}{\begin{array}{c}(G, S, \&e) \xrightarrow{\text{rhs}} (G', S', \&e') \\ (G, S, e = e_2) \xrightarrow{\text{rhs}} (G', S', e' = e_2)\end{array}} \qquad (\mathbf{DR10})\quad \frac{(G, S, s) \xrightarrow{\text{stmt}} (G', S', s')}{(G, S, \texttt{call}\{s\}) \xrightarrow{\text{rhs}} (G', S', \texttt{call}\{s'\})}$$

$$(\mathbf{DR11})\quad \frac{(G, S, e) \xrightarrow{\text{rhs}} (G', S', e')}{\begin{array}{cc} (G, S, e\langle\tau\rangle) \xrightarrow{\text{rhs}} (G', S', e'\langle\tau\rangle) & (G, S, \texttt{new}(e)\, e_2) \xrightarrow{\text{rhs}} (G', S', \texttt{new}(e')\, e_2) \\ (G, S, (e, e_2)) \xrightarrow{\text{rhs}} (G', S', (e', e_2)) & (G, S, \texttt{new}(v)\, e) \xrightarrow{\text{rhs}} (G', S', \texttt{new}(v)\, e') \\ (G, S, (v, e)) \xrightarrow{\text{rhs}} (G', S', (v, e')) & (G, S, e(e_2)) \xrightarrow{\text{rhs}} (G', S', e'(e_2)) \\ (G, S, *e) \xrightarrow{\text{rhs}} (G', S', *e') & (G, S, v(e)) \xrightarrow{\text{rhs}} (G', S', v(e')) \\ (G, S, e.i) \xrightarrow{\text{rhs}} (G', S', e'.i) & (G, S, p = e) \xrightarrow{\text{rhs}} (G', S', p = e') \\ \multicolumn{2}{c}{(G, S, \texttt{pack}\,[\tau_1, e]\,\texttt{as}\,\tau_2) \xrightarrow{\text{rhs}} (G', S', \texttt{pack}\,[\tau_1, e']\,\texttt{as}\,\tau_2)} \end{array}}$$

Figure 6: Right-Hand-Side Expression Rewriting

$$(\mathbf{DL1})\quad (G, S, *(\&p)) \xrightarrow{\text{lhs}} (G, S, p)$$

$$(\mathbf{DL2})\quad \frac{(G, S, e) \xrightarrow{\text{lhs}} (G', S', e')}{(G, S, e.i) \xrightarrow{\text{lhs}} (G', S', e'.i)} \qquad (\mathbf{DL3})\quad \frac{(G, S, e) \xrightarrow{\text{rhs}} (G', S', e')}{(G, S, *e) \xrightarrow{\text{lhs}} (G', S', *e')}$$

Figure 7: Left-Hand-Side Expression Rewriting

## 9.2 Dynamic Semantics

Three mutually inductive judgments (Figures 5, 6, and 7) define the dynamic semantics. Rules **DS9**, **DS10**, **DR9**, **DR10**, **DR11**, **DL2**, and **DL3** are congruence rules. They show why the judgments are mutually inductive. We state without proof that the dynamic semantics is deterministic; it is syntax-directed.

We use an allocation-style semantics for term variables, but we use substitution for type variables, written $c[\tau/\alpha]$ for some construct $c$ (for example, an expression) in which we substitute $\tau$ for $\alpha$. For now, a rough intuition of substitution suffices, but we explain below that the definition is non-standard because of the case where a type variable $\alpha$ appears in a region-set $\epsilon$. Although we do not formally prove it, the semantics clearly enjoys a type-erasure property: Types are not actually used at run-time; we substitute them just to prove type preservation.

The remaining rules for statement evaluation are described as follows:

- **DS1** discards the first part of a sequence after we are done evaluating it.
- **DS2** and **DS3** reduce conditionals to the appropriate branch.
- **DS4** "unrolls" a loop to an equivalent expression.
- **DS5** discards the second part of a sequence when the first part returns.
- **DS6** and **DS7** deallocate the region named $\rho$. Notice that these rules apply only when $\rho$ names the most recently allocated region. ($S$ is ordered and newer regions appear on the right.) We leave the entire region $R$ in $G$ as a matter of convenience, but no code will ever access $G$. The type-preservation argument just uses the fact that we can give a consistent type to $G$ and that new regions names will be unique (not in the domain of $G$).
- **DS8a** allocates a new region for a local binding and rewrites the statement to one that first evaluates the inner statement and then deallocates the region. $\alpha$-conversion ensures that the side-condition can always be satisfied.
- **DS8b** allocates a new region, puts a handle for the region in the region itself, and is otherwise quite similar to **DS8a**. Note this rule does not model Cyclone as accurately as the others because region handles in Cyclone are stack-allocated. The important point is that handles are in some region (not stored magically); the exact region is less interesting.
- **DS8c** unpacks an existential package, by substituting the witness type for the bound type variable and rewriting to a local binding (so that **DS8a** will apply to the result).

The remaining rules for right-hand-side evaluation are described as follows:

- **DR1** is variable lookup. Notice how the region annotation on identifiers is used.

- **DR2** is type application; we use a substitution approach for type variables.
- **DR3** is pair projection.
- **DR4** is pointer dereference. Notice how the normal-form for pointers is $\&p$ for some path $p$.
- **DR5** is dynamic allocation. $\alpha$-conversion ensures the side-condition can always be satisfied. Notice how the handle indicates the region and the result is a pointer to an identifier and the identifier indicates the region.
- **DR6** is function call. Notice how we use the `call`$\{s\}$ form to encode the call-stack in the term language. The clever use of the local binding will cause the next step to allocate a region for the actual parameter and bind it to the formal. (For recursive functions, the side-condition in rule **DS8a** ensures that we use distinct storage for each activation.)
- **DR7** shows how functions return: Once a return statement is the top-level statement of a call (thanks to **DS5** and **DS6**), we replace the call with the result $v$.
- **DR8** is assignment. It is complicated because assignment can update *part* of a location (when the location is a (nested) pair and the left-hand-side has a non-empty list of fields). The definition uses the auxiliary inductive definition of update to encode the fact that we update exactly the part of the value that the path $x_\rho.i_1.\cdots.i_n$ refers to.

Notice that no left-hand-side rule applies for $p.i$; this is a terminal configuration for left-hand-sides. The only true left-hand reduction is **DL1**; it is the left-hand-side equivalent of **DR4**. In both cases, the expression after the step is a path $p$. The interesting difference is that, as a left-hand-side, no rule applies for $p$ (it is a terminal left-hand-side), but as a right-hand-side, either **DR1** or **DR3** applies.

We now return to (type) substitution, which was used in **DR2** and **DS8c**. It is formally defined in Figures 8 and 9, but we find an informal description far more illuminating. The substitution of types for type variables through statements, expressions, and other types is completely standard, *except* for what it means to substitute through an $\epsilon$ or a $\gamma$, as occurs in the cases for function types and quantified types. In these cases, we use regions($\tau$), which is an auxiliary function from types to region-sets defined in Figure 10. The idea is to replace $\alpha$ with the region names and type variables in the type for which we are substituting.

However, this substitution won't work if we have a constraint of the form $\epsilon <: \rho$ and we are substituting a $\tau$ for $\rho$ such that regions($\tau$) is not a singleton set. (In the case where it is a singleton set, we're just abusing notation to avoid defining substitution twice.) However, we only ever substitution one region-name for another (all types of kind $\mathcal{R}$ are region names), so we can leave substitution undefined in other cases. Our Substitution Lemmas distinguish as necessary the kind of the variable for which we are substituting.

An obvious question is why we didn't relax the form of constraints to $\epsilon_1 <: \epsilon_2$ to avoid this issue. The answer is subtle: We wanted to support transitivity when using constraints to determine outlives relationships. In fact, we use transitivity in a substitution lemma needed in our type-preservation proof. So if we have $\alpha_1 <: \alpha_2$ and $\alpha_2 <: \alpha_3$, then we would like to conclude $\alpha_1 <: \alpha_3$. But if we could substitute int for for $\alpha_2$, we would have $\alpha_1 <: \emptyset$ and $\emptyset <: \alpha_3$. Under a reasonable interpretation of $<:$—that everything on the left outlives everything on the right—our constraints are now useless. We cannot conclude $\alpha_1 <: \alpha_3$, which would make type preservation break. By restricting the right side of $<:$ to region names, we can preserve transitivity under substitution.

$$\text{Region Sets:}\quad \begin{array}{l} \emptyset[\epsilon/\alpha] = \emptyset \\ \alpha[\epsilon/\alpha] = \epsilon \\ \alpha[\epsilon/\beta] = \alpha \\ (\epsilon_1 \cup \epsilon_2)[\epsilon_3/\alpha] = \epsilon_1[\epsilon_3/\alpha] \cup \epsilon_2[\epsilon_3/\alpha] \end{array}$$

$$\text{Region Constraints:}\quad \begin{array}{l} \emptyset[\epsilon/\alpha] = \emptyset \\ (\gamma, \epsilon <: \rho)[\rho'/\rho] = \gamma[\rho'/\rho], \epsilon[\rho'/\rho] <: \rho' \\ (\gamma, \epsilon <: \rho)[\epsilon'/\alpha] = \gamma[\epsilon'/\alpha], \epsilon[\epsilon/\alpha] <: \rho \end{array}$$

*Note that $\gamma[\epsilon/\rho]$ is not defined if $\gamma = \gamma_1, \epsilon <: \rho, \gamma_2$. Type substitution is undefined if the type contains such a $\gamma$ and the substituted type is not of the form $\rho'$.*

$$\text{Constructors:}\quad \begin{array}{l} \alpha[\tau/\alpha] = \tau \\ \beta[\tau/\alpha] = \beta \\ \text{int}[\tau/\alpha] = \text{int} \\ (\tau_1 \xrightarrow{\epsilon} \tau_2)[\tau/\alpha] = \tau_1[\tau/\alpha] \xrightarrow{\epsilon[\text{regions}(\tau)/\alpha]} \tau_2[\tau/\alpha] \\ (\tau_1 \times \tau_2)[\tau/\alpha] = \tau_1[\tau/\alpha] \times \tau_2[\tau/\alpha](\tau_1@\rho)[\rho'/\rho] = \tau_1[\rho'/\rho]@\rho' \\ (\tau_1@\rho)[\tau/\alpha] = \tau_1[\tau/\alpha]@\rho \\ \text{handle}(\rho)[\rho'/\rho] = \text{handle}(\rho') \\ \text{handle}(\rho)[\tau'/\alpha] = \text{handle}(\rho) \\ (\forall\beta{:}\kappa \rhd \gamma.\tau_1)[\tau/\alpha] = \forall\beta{:}\kappa \rhd \gamma[\text{regions}(\tau)/\alpha].\tau_1[\tau/\alpha] \\ (\exists\beta{:}\kappa \rhd \gamma.\tau_1)[\tau/\alpha] = \exists\beta{:}\kappa \rhd \gamma[\text{regions}(\tau)/\alpha].\tau_1[\tau/\alpha] \end{array}$$

*Note that type substitution of $\tau$ for $\rho$ in $\tau'$ is undefined if $\tau$ is not $\rho'$ for some $\rho'$ and $\tau'$ contains a type of the form $\tau''@\rho$ or* handle$(\rho)$*.*
*Note that substitution through terms and contexts of $\tau$ (not of the form $\rho'$) for a region variable is undefined by extension of the earlier considerations.*

$$\text{Value Variable Contexts:}\quad \begin{array}{ll} \bullet[\tau/\alpha] = \bullet & \\ (\Gamma, x_\rho : \tau')[\rho'/\rho] = \Gamma[\rho'/\rho], x_{\rho'} : \tau'[\rho'/\rho] & \text{This case never occurs.} \\ (\Gamma, x_\rho : \tau')[\tau/\alpha] = \Gamma[\tau/\alpha], x_\rho : \tau'[\tau/\alpha] & \end{array}$$

Figure 8: Substitution, part I (the kind of $\alpha$ may be $\mathcal{B}$ or $\mathcal{R}$)

Expressions:

$x_\rho[\rho'/\rho] = x_{\rho'}$ This case never occurs.
$x_\rho[\tau/\alpha] = x_\rho$
$i[\tau/\alpha] = i$
$(\rho{:}(\tau_1\, x_\rho) \xrightarrow{\epsilon} \tau_2 = \{s\})[\tau/\alpha] = (\rho{:}(\tau_1[\tau/\alpha]\, x_\rho) \xrightarrow{\epsilon[\text{regions}(\tau)/\alpha]} \tau_2[\tau/\alpha] = \{s[\tau/\alpha]\})$
$(\Lambda\beta{:}\kappa \triangleright \gamma.f)[\tau/\alpha] = \Lambda\beta{:}\kappa \triangleright \gamma[\text{regions}(\tau)/\alpha].f[\tau/\alpha]$
$\texttt{region}(\rho)[\rho'/\rho] = \texttt{region}(\rho')$ This case never occurs.
$(e\langle\tau'\rangle)[\tau/\alpha] = e[\tau/\alpha]\langle\tau'[\tau/\alpha]\rangle$
$(e_1, e_2)[\tau/\alpha] = (e_1[\tau/\alpha], e_2[\tau/\alpha])$
$e.i[\tau/\alpha] = e[\tau/\alpha].i$
$(*e)[\tau/\alpha] = *(e[\tau/\alpha])$
$(\texttt{new}(e_1)e_2)[\tau/\alpha] = \texttt{new}(e_1[\tau/\alpha])e_2[\tau/\alpha]$
$(e_1(e_2))[\tau/\alpha] = e_1[\tau/\alpha])(e_2[\tau/\alpha])$
$(\&e)[\tau/\alpha] = \&(e[\tau/\alpha])$
$(e_1 = e_2)[\tau/\alpha] = e_1[\tau/\alpha] = e_2[\tau/\alpha]$
$\texttt{call}\{s\}[\tau/\alpha] = \texttt{call}\{s[\tau/\alpha]\}$
$(\texttt{pack}\,[\tau_1, e]\,\texttt{as}\,\tau_2)[\tau/\alpha] = \texttt{pack}\,[\tau_1[\tau/\alpha], e[\tau/\alpha]]\,\texttt{as}\,\tau_2[\tau/\alpha]$

Statements:

$(\texttt{return}\,e)[\tau/\alpha] = \texttt{return}\,(e[\tau/\alpha])$
$(s_1; s_2)[\tau/\alpha] = s_1[\tau/\alpha]; s_2[\tau/\alpha]$
$(\texttt{if}\ (e)\ s_1\ \texttt{else}\ s_2)[\tau/\alpha] = \texttt{if}\ (e[\tau/\alpha])\ s_1[\tau/\alpha]\ \texttt{else}\ s_2[\tau/\alpha]$
$(\texttt{while}\ (e)\ s)[\tau/\alpha] = \texttt{while}\ (e[\tau/\alpha])\ s[\tau/\alpha]$
$(\rho{:}\{\tau'\ x_\rho = e;\ s\})[\tau/\alpha] = \rho{:}\{\tau'[\tau/\alpha]\ x_\rho = e[\tau/\alpha];\ s[\tau/\alpha]\}$
$(\texttt{region}\langle\rho\rangle\ x_\rho\ s)[\tau/\alpha] = \texttt{region}\langle\rho\rangle\ x_\rho\ s[\tau/\alpha]$
$(\rho{:}\{\texttt{open}\,[\beta, x_\rho] = e;\ s\})[\tau/\alpha] = \rho{:}\{\texttt{open}\,[\beta, x_\rho] = e[\tau/\alpha];\ s[\tau/\alpha]\}$
$(s\,\texttt{pop}[\rho])[\rho'/\rho] = s[\rho'/\rho]\,\texttt{pop}[\rho']$ This case never occurs.
$(s\,\texttt{pop}[\rho])[\tau/\alpha] = s[\tau/\alpha]\,\texttt{pop}[\rho]$

Figure 9: Substitution, part II (the kind of $\alpha$ may be $\mathcal{B}$ or $\mathcal{R}$)

$$\begin{array}{rcl}
\text{regions}(\alpha) & = & \alpha \\
\text{regions}(\text{int}) & = & \emptyset \\
\text{regions}(\tau_1 \xrightarrow{\epsilon} \tau_2) & = & \epsilon \\
\text{regions}(\tau_1 \times \tau_2) & = & \text{regions}(\tau_1) \cup \text{regions}(\tau_2) \\
\text{regions}(\tau@\rho) & = & \rho \cup \text{regions}(\tau) \\
\text{regions}(\text{handle}(\rho)) & = & \rho \\
\text{regions}(\forall\alpha{:}\kappa \triangleright \gamma.\tau) & = & \text{regions}(\tau) - \alpha \\
\text{regions}(\exists\alpha{:}\kappa \triangleright \gamma.\tau) & = & \text{regions}(\tau) - \alpha
\end{array}$$

Figure 10: Regions-Of

## 9.3 Static Semantics

Well-formed programs obey several invariants, which we enforce with separate judgments. We begin with the more conventional-looking "type-checking" judgments for statements and expressions. After extending these judgments to heaps ($G$ and $S$) and machine states, we have a system that prevents type errors (such as treating an int as a function) and prevents dangling pointers. But that is not enough to prove that programs deallocate all of their regions and that functions always diverge or return (as opposed to "fall off the end" as is possible—and unsafe—in C). We therefore add other judgments for these invariants. Finally, we present some technical judgments used but not discussed in detail, namely kind-checking and context well-formedness.

### 9.3.1 Statement and Expression Type-Checking

As with the dynamic semantics, we have three mutually inductive judgments for statements (Figure 11), right-hand-side expressions (Figures 12 and 13), and left-hand-side expressions (Figure 14). We begin by explaining the form of each judgment and then describe the individual rules.

In each judgment, we have a $\Delta$, $\Gamma$, $\gamma$, and $\epsilon$ in the context. These describe the type variables (and region names) in scope, the value variables in scope, the outlives relationships assumed true, and the regions assumed live, respectively. To see that all of these constructs are necessary, consider (informally for the moment) type-checking $*x_\rho$. We need to know that $\rho$ is a region-name (via $\Delta$) for a region that outlives (via $\gamma$) something that we know is live (via $\epsilon$) and that $x_\rho$ is a pointer (via $\Gamma$). (We also need to know $x_\rho$ points into a live region.) Each judgment has a slightly different form:

- $\Delta; \Gamma; \gamma; \epsilon; \tau \vdash_{\text{stmt}} s$ means $s$ is well-typed even though statements do not have types (or have type unit if you prefer). The $\tau$ in the context is used to type-check return statements: It is the return type for the function body being type-checked (int for the top-level program).
- $\Delta; \Gamma; \gamma; \epsilon \vdash_{\text{rhs}} e : \tau$ means $e$ is a well-typed right-hand-side expression of type $\tau$.
- $\Delta; \Gamma; \gamma; \epsilon \vdash_{\text{lhs}} e : \tau @ \rho$ means $e$ is a well-typed left-hand-side expression of type $\tau$ *in the region named* $\rho$.

The point is that right-hand-sides do not have region names ("the region of 42" makes no sense), but left-hand-sides do have region names (the path to which the expression evaluates will begin with an identifier with the region name). Of course, right-hand-sides with pointer types have a region name in their type. In fact, to type-check $\&e$ as a right-hand-side, the region name from type-checking $e$ as a left-hand-side is the region name with which we annotate the pointer type for $\&e$.

We now describe each of the rules for type-checking statements:

- **SS1** through **SS5** are mostly just congruence rules. Notice that in **SS2** we use the $\tau$ in the context as the required type for the expression returned. Also, loop and conditional guards must have type int.
- **SS6** requires that $\rho$ is a region-name and adds $\rho$ to the capability used to type-check $s$. The point is that $\rho$ should not be in the capability *except* in $s$ because the associated region will be deallocated after $s$ executes. As we will see, we type-check whole programs under the empty capability—only descending under the `pop` form provides access to a region. Well, actually we start with an initial capability for the region holding the program's "static data", which the program does not deallocate.
- **SS7** through **SS9** each add a region $\rho$ and a term variable $x_\rho$ for type-checking the contained statement $s$. The new region will be deallocated before any live region, so we add $\epsilon <: \rho$ to $\gamma$. The new region remains live throughout the execution of $s$, so we add $\rho$ to $\epsilon$. The type of $x_\rho$ is

$$(\textbf{SS1})\quad \frac{\Delta;\Gamma;\gamma;\epsilon \vdash_{\text{rhs}} e : \tau' \quad \Delta \vdash_{\text{con}} \tau{:}\mathcal{T}}{\Delta;\Gamma;\gamma;\epsilon;\tau \vdash_{\text{stmt}} e}$$

$$(\textbf{SS2})\quad \frac{C \vdash_{\text{rhs}} e : \tau}{C;\tau \vdash_{\text{stmt}} \texttt{return}\, e}$$

$$(\textbf{SS3})\quad \frac{C;\tau \vdash_{\text{stmt}} s_1 \quad C;\tau \vdash_{\text{stmt}} s_2}{C;\tau \vdash_{\text{stmt}} s_1; s_2}$$

$$(\textbf{SS4})\quad \frac{C \vdash_{\text{rhs}} e : \text{int} \quad C;\tau \vdash_{\text{stmt}} s_1 \quad C;\tau \vdash_{\text{stmt}} s_2}{C;\tau \vdash_{\text{stmt}} \texttt{if}\ (e)\ s_1\ \texttt{else}\ s_2}$$

$$(\textbf{SS5})\quad \frac{C \vdash_{\text{rhs}} e : \text{int} \quad C;\tau \vdash_{\text{stmt}} s}{C;\tau \vdash_{\text{stmt}} \texttt{while}\ (e)\ s}$$

$$(\textbf{SS6})\quad \frac{\Delta;\Gamma;\gamma;\epsilon \uplus \rho;\tau \vdash_{\text{stmt}} s \quad \Delta \vdash_{\text{con}} \rho{:}\mathcal{R}}{\Delta;\Gamma;\gamma;\epsilon;\tau \vdash_{\text{stmt}} s\, \texttt{pop}[\rho]}$$

$$(\textbf{SS7})\quad \frac{\begin{array}{c}\Delta;\Gamma;\gamma;\epsilon \vdash_{\text{rhs}} e : \tau' \quad \Delta \vdash_{\text{con}} \tau{:}\mathcal{T} \\ (\Delta, \rho{:}\mathcal{R}); (\Gamma, x_\rho{:}\tau'); (\gamma, \epsilon <: \rho); (\epsilon \cup \rho); \tau \vdash_{\text{stmt}} s\end{array}}{\Delta;\Gamma;\gamma;\epsilon;\tau \vdash_{\text{stmt}} \rho{:}\{\tau'\, x_\rho = e;\ s\}} \quad (\rho \notin \text{Dom}(\Delta), x_\rho \notin \text{Dom}(\Gamma))$$

$$(\textbf{SS8})\quad \frac{\begin{array}{c}\vdash_{\text{ctxt}} \Delta;\Gamma;\gamma;\epsilon \quad \Delta \vdash_{\text{con}} \tau{:}\mathcal{T} \\ (\Delta, \rho{:}\mathcal{R}); (\Gamma, x_\rho{:}\text{handle}(\rho)); (\gamma, \epsilon <: \rho); (\epsilon \cup \rho); \tau \vdash_{\text{stmt}} s\end{array}}{\Delta;\Gamma;\gamma;\epsilon;\tau \vdash_{\text{stmt}} \texttt{region}\langle\rho\rangle\, x_\rho\, s} \quad (\rho \notin \text{Dom}(\Delta), x_\rho \notin \text{Dom}(\Gamma))$$

$$(\textbf{SS9})\quad \frac{\begin{array}{c}\Delta;\Gamma;\gamma;\epsilon \vdash_{\text{rhs}} e : \exists\alpha{:}\kappa \triangleright \gamma_1.\tau_1 \quad \Delta \vdash_{\text{con}} \tau{:}\mathcal{T} \\ (\Delta, \rho{:}\mathcal{R}, \alpha{:}\kappa); (\Gamma, x_\rho{:}\tau_1); (\gamma, \epsilon <: \rho, \gamma_1); (\epsilon \cup \rho); \tau \vdash_{\text{stmt}} s\end{array}}{\Delta;\Gamma;\gamma;\epsilon;\tau \vdash_{\text{stmt}} \rho{:}\{\texttt{open}\,[\alpha, x_\rho] = e;\ s\}} \quad (\rho, \alpha \notin \text{Dom}(\Delta), x_\rho \notin \text{Dom}(\Gamma))$$

Figure 11: Statement Typing Rules

$$(\mathbf{SR1})\quad \frac{\gamma \vdash_{\text{ei}} \epsilon \Rightarrow \rho \quad \vdash_{\text{ctxt}} \Delta;\Gamma;\gamma;\epsilon}{\Delta;\Gamma;\gamma;\epsilon \vdash_{\text{rhs}} x_\rho : \Gamma(x_\rho)}$$

$$(\mathbf{SR2})\quad \frac{\Delta;\Gamma;\gamma;\epsilon \vdash_{\text{rhs}} e : \tau@\rho_1 \quad \gamma \vdash_{\text{ei}} \rho_2 \Rightarrow \rho_1}{\Delta;\Gamma;\gamma;\epsilon \vdash_{\text{rhs}} e : \tau@\rho_2}$$

$$(\mathbf{SR3})\quad \frac{\vdash_{\text{ctxt}} C}{C \vdash_{\text{rhs}} i : \text{int}}$$

$$(\mathbf{SR4})\quad \frac{\Delta \vdash_{\text{con}} \rho : \mathcal{R} \quad \vdash_{\text{ctxt}} \Delta;\Gamma;\gamma;\epsilon}{\Delta;\Gamma;\gamma;\epsilon \vdash_{\text{rhs}} \texttt{region}(\rho) : \text{handle}(\rho)}$$

$$(\mathbf{SR5})\quad \frac{C \vdash_{\text{rhs}} e_1 : \tau_1 \qquad C \vdash_{\text{rhs}} e_2 : \tau_2}{C \vdash_{\text{rhs}} (e_1, e_2) : \tau_1 \times \tau_2}$$

$$(\mathbf{SR6})\quad \frac{C \vdash_{\text{rhs}} e : \tau_1 \times \tau_2}{C \vdash_{\text{rhs}} e.i : \tau_i}$$

$$(\mathbf{SR7})\quad \frac{\Delta;\Gamma;\gamma;\epsilon \vdash_{\text{rhs}} e : \tau@\rho \quad \gamma \vdash_{\text{ei}} \epsilon \Rightarrow \rho}{\Delta;\Gamma;\gamma;\epsilon \vdash_{\text{rhs}} *e : \tau}$$

$$(\mathbf{SR8})\quad \frac{\begin{array}{c}\gamma \vdash_{\text{ei}} \epsilon \Rightarrow \rho \\ \Delta;\Gamma;\gamma;\epsilon \vdash_{\text{rhs}} e_1 : \text{handle}(\rho) \qquad \Delta;\Gamma;\gamma;\epsilon \vdash_{\text{rhs}} e_2 : \tau\end{array}}{\Delta;\Gamma;\gamma;\epsilon \vdash_{\text{rhs}} \texttt{new}(e_1)\, e_2 : \tau@\rho}$$

Figure 12: Right-Hand-Side Expression Typing Rules (Part I)

$$(\textbf{SR9})\quad \frac{C \vdash_{\text{lhs}} e : \tau@\rho}{C \vdash_{\text{rhs}} \&e : \tau@\rho}$$

$$(\textbf{SR10})\quad \frac{\Delta;\Gamma;\gamma;\epsilon \vdash_{\text{lhs}} e_1 : \tau@\rho \qquad \Delta;\Gamma;\gamma;\epsilon \vdash_{\text{rhs}} e_2 : \tau \qquad \gamma \vdash_{\text{ei}} \epsilon \Rightarrow \rho}{\Delta;\Gamma;\gamma;\epsilon \vdash_{\text{rhs}} e_1 = e_2 : \tau}$$

$$(\textbf{SR11})\quad \frac{C;\tau \vdash_{\text{stmt}} s \quad \vdash_{\text{ret}} s}{C \vdash_{\text{rhs}} \texttt{call}\{s\} : \tau}$$

$$(\textbf{SR12})\quad \frac{\begin{array}{c}\Delta;\Gamma;\gamma;\epsilon \vdash_{\text{rhs}} e_1 : \tau_2 \xrightarrow{\epsilon_1} \tau \qquad \Delta;\Gamma;\gamma;\epsilon \vdash_{\text{rhs}} e_2 : \tau_2 \\ \gamma \vdash_{\text{ei}} \epsilon \Rightarrow \epsilon_1\end{array}}{\Delta;\Gamma;\gamma;\epsilon \vdash_{\text{rhs}} e_1(e_2) : \tau}$$

$$(\textbf{SR13})\quad \frac{\begin{array}{c}\Delta \vdash_{\text{con}} \tau_1 : \kappa \qquad \gamma \vdash_{\text{ord}} \gamma_1[\text{regions}(\tau_1)/\alpha] \\ \Delta;\Gamma;\gamma;\epsilon \vdash_{\text{rhs}} e : \tau_2[\tau_1/\alpha]\end{array}}{\Delta;\Gamma;\gamma;\epsilon \vdash_{\text{rhs}} \texttt{pack}\,[\tau_1, e]\,\texttt{as}\,\exists\alpha{:}\kappa \triangleright \gamma_1.\tau_2 : \exists\alpha{:}\kappa \triangleright \gamma_1.\tau_2} \quad (\alpha \notin \text{Dom}(\Delta), \kappa \neq \mathcal{T})$$

$$(\textbf{SR14})\quad \frac{\begin{array}{c}\Delta;\Gamma;\gamma;\epsilon \vdash_{\text{rhs}} e : \forall\alpha{:}\kappa \triangleright \gamma_1.\tau_2 \\ \Delta \vdash_{\text{con}} \tau_1 : \kappa \qquad \gamma \vdash_{\text{ord}} \gamma_1[\text{regions}(\tau_1)/\alpha]\end{array}}{\Delta;\Gamma;\gamma;\epsilon \vdash_{\text{rhs}} e\langle\tau_1\rangle : \tau_2[\tau_1/\alpha]}$$

$$(\textbf{SR15})\quad \frac{\begin{array}{c}\Delta \vdash_{\text{con}} \tau_1 \xrightarrow{\epsilon_1} \tau_2 \quad \vdash_{\text{ctxt}} \Delta;\Gamma;\gamma;\epsilon \\ (\Delta, \rho{:}\mathcal{R});(\Gamma, x_\rho{:}\tau_1);(\gamma, \epsilon_1 <: \rho);(\epsilon_1 \uplus \rho);\tau_2 \vdash_{\text{stmt}} s \quad \vdash_{\text{ret}} s\end{array}}{\Delta;\Gamma;\gamma;\epsilon \vdash_{\text{rhs}} \rho{:}(\tau_1 x_\rho) \xrightarrow{\epsilon_1} \tau_2 = \{s\} : \tau_1 \xrightarrow{\epsilon_1} \tau_2} \quad (\rho \notin \text{Dom}(\Delta))$$

$$(\textbf{SR16})\quad \frac{(\Delta, \alpha{:}\kappa);\Gamma;(\gamma, \gamma_1);\epsilon \vdash_{\text{rhs}} e : \tau \quad \Delta, \alpha{:}\kappa \vdash_{\text{rc}} \gamma_1 \quad \vdash_{\text{ctxt}} \Delta;\Gamma;\gamma;\epsilon}{\Delta;\Gamma;\gamma;\epsilon \vdash_{\text{rhs}} \Lambda\alpha{:}\kappa \triangleright \gamma_1.e : \forall\alpha{:}\kappa \triangleright \gamma_1.\tau} \quad (\alpha \notin \text{Dom}(\Delta), \kappa \neq \mathcal{T})$$

Figure 13: Right-Hand-Side Expression Typing Rules (Part II)

$$(\textbf{SL1})\quad \frac{\vdash_{\text{ctxt}} C}{C \vdash_{\text{lhs}} x_\rho : \Gamma(x_\rho)@\rho} \qquad (\textbf{SL2})\quad \frac{C \vdash_{\text{rhs}} e : \tau@\rho}{C \vdash_{\text{lhs}} {*e} : \tau@\rho}$$

$$(\textbf{SL3})\quad \frac{C \vdash_{\text{lhs}} e : (\tau_1 \times \tau_2)@\rho}{C \vdash_{\text{lhs}} e.i : \tau_i@\rho}$$

Figure 14: Left-Hand-Side Expression Typing Rules

the expected one for each rule. Finally, in **SS9** we also add $\gamma_1$, the constraints ensured by the existential type, to the context when type-checking $s$. The typing rule for creating existential types ensures these constraints are true, so it is sound to use them. (We cannot use them elsewhere because they may mention $\alpha$, which is added to $\Delta$ only when type-checking $s$.)

We now describe the type-checking rules for right-hand expressions. Several of the rules use antecedents of the form $\gamma \vdash_{\mathrm{ei}} \epsilon_1 \Rightarrow \epsilon_2$ (where $\epsilon_1$ or $\epsilon_2$ may be a single region-name $\rho$). The formal rules for this judgment are presented later. Informally, it means that under constraints $\gamma$, having capability $\epsilon_1$ suffices for establishing capability $\epsilon_2$. In other words, if everything in $\epsilon_1$ is live, then everything in $\epsilon_2$ is live. Similarly, we have $\gamma_1 \vdash_{\mathrm{ord}} \gamma_2$, which means every constraint in $\gamma_2$ is provable given $\gamma_1$.

Turning to the individual rules:

- **SR1**: To type-check a variable, we need to establish that its region is live under the current capability (because the dynamic semantics will "read" from the region). The result is from $\Gamma$.
- **SR2**: This rule is subsumption, allowing subtyping on pointers based on the outlives relationship of regions. Specifically, if we can establish that $\rho_1$ outlives $\rho_2$, then it is sound to cast from $\tau@\rho_1$ to $\tau@\rho_2$. Note that this cast is *not* deep; pointer types are invariant in $\tau$. (This point is subtle for the heap. Safety comes from the fact that we will pick one type for each heap location, so different aliases are unable to view it at different types. Only right-hand-sides may be cast; all aliases "use the same cast".)
- **SR3** through **SR6** are self-explanatory. As expected, the region-name for a handle appears in its type.
- **SR7** is much like **SR1**; we need to know that the region that the expression's evaluation will access is live.
- **SR8** is what we would expect for allocation; the result type is a pointer into the handle's region. Notice that we must know the handle's region; the region-name for the handle to which $e_1$ evaluates must be known at compile-time. Of course, the region must be live.
- **SR9** and **SR10** use the judgment for left-hand-sides appropriately. Notice that **SR9** does not require the region to be live (we permit dangling pointers, just not following them) whereas **SR10** does require the region to be live (because assignment "writes" to the region).
- **SR11** uses the statement judgment and requires that the statement cannot "fall off the end".
- **SR12** is for function calls. As should be expected, the current capability must be sufficient to demonstrate the function's effect. Otherwise, the call's execution could access dead regions.
- **SR13** introduces existential types. The only non-standard feature is the requirement that the constraint $\gamma_1$ in the result type is ensured by the current constraint $\gamma$ (when using the witness type for $\alpha$). Constraints never become false (the region lifetime ordering is unchangeable), so this assumption is what allows the open construct to assume the constraints of the package it opens.
- **SR14** is the elimination form for universal types. It is the dual of **SR13**. The only non-standard feature is the obligation to show that the current constraint suffices to prove the constraint in the universal type. This assumption is what allows us to assume the constraint when type-checking the body of a polymorphic expression.

$$(\mathbf{SH})\quad \frac{\begin{array}{c} G = [\rho'_1 \mapsto R'_1, \ldots, \rho'_m \mapsto R'_m] \qquad S = [\rho_1 \mapsto R_1, \ldots, \rho_n \mapsto R_n] \\ \Delta = \rho'_1{:}\mathcal{R}, \ldots, \rho'_m{:}\mathcal{R}, \rho_1{:}\mathcal{R}, \ldots, \rho_n{:}\mathcal{R} \\ \gamma_G = \epsilon_1 <: \rho'_1, \ldots, \epsilon_m <: \rho'_m \qquad \gamma_S = \rho_1 <: \rho_2, \rho_2 <: \rho_3, \ldots, \rho_{n-1} <: \rho_n \\ \gamma = \gamma_G, \gamma_S \qquad \Delta \vdash_{\text{rc}} \gamma \\ \Gamma = \Gamma_G \uplus \Gamma_S \qquad \Delta \vdash_{\text{vctxt}} \Gamma \\ \Delta; \Gamma; \gamma \vdash_{\text{stack}} G : \Gamma_G \qquad \Delta; \Gamma; \gamma \vdash_{\text{stack}} S : \Gamma_S \end{array}}{\vdash_{\text{heap}} (G, S) : \Delta; \Gamma; \gamma}$$

$$(\mathbf{SStk1})\quad \Delta; \Gamma; \gamma \vdash_{\text{stack}} \emptyset : \bullet$$

$$(\mathbf{SStk2})\quad \frac{\Delta; \Gamma; \gamma \vdash_{\text{stack}} S : \Gamma_1 \quad \Delta; \Gamma; \gamma \vdash_{\text{rgn}} \rho \mapsto R : \Gamma_2}{\Delta; \Gamma; \gamma \vdash_{\text{stack}} S[\rho \mapsto R] : \Gamma_1 \uplus \Gamma_2} \qquad (\rho \notin \text{Dom}(S))$$

$$(\mathbf{SRgn1})\quad \Delta; \Gamma; \gamma \vdash_{\text{rgn}} \rho \mapsto \emptyset : \bullet$$

$$(\mathbf{SRgn2})\quad \frac{\Delta; \Gamma; \gamma \vdash_{\text{rgn}} \rho \mapsto R : \Gamma_1 \quad \Delta; \Gamma; \gamma; \emptyset \vdash_{\text{rhs}} v : \tau \quad \emptyset \vdash_{\text{epop}} v}{\Delta; \Gamma; \gamma \vdash_{\text{rgn}} \rho \mapsto R[x \mapsto v] : \Gamma_1, x_\rho{:}\tau} \qquad (x_\rho \notin \text{Dom}(\Gamma_1))$$

Figure 15: Machine-Level Typing Rules

- **SR15** is for function bodies. These bodies must return a value of the appropriate type. They assume the explicit effect is live (the current capability). Furthermore, the parameter $x_\rho$ is in a live region $\rho$ that every other live region outlives. Note that our formal language actually permits "nested functions" and "downward funargs" even though Cyclone does not. In Cyclone, this rule would only be used for a "top-level" function, so $\Delta$ and $\epsilon$ would have only the heap's region-name and $\Gamma$ would have only the code and static data. ($\gamma$ would be empty.) Also note, as usual, that we do not require $\gamma \vdash_{\text{ei}} \epsilon \Rightarrow \epsilon_1$ (if we did, top-level functions that took non-heap pointers couldn't type-check). The implication is necessary only when calling a function (rule **SR12**).

- **SR16** is for universal introduction. It is the dual of **SS9**. When type-checking the contained expression, we can assume the constraints in $\gamma_1$. The other assumptions are just technical well-formedness conditions (e.g., $\gamma_1$ cannot mention type variables not in $\Delta$).

The rules for left-hand-sides are straightforward. Notice that none of them explicitly require a region to be live: Evaluating left-hand-sides does not read memory unless there is a dereference, in which case (rule **SL2**), the assumption uses $\vdash_{\text{rhs}}$, which will require the necessary capability.

### 9.3.2 Heap Type-Checking

The heap is a collection of dead regions ($G$) and a collection of live regions ($S$) with a fixed ordering. The heap must be well-typed and provide a context under which the current program is well-typed. The relevant rules are in Figure 15.

Because the heap can contain cycles, we need to assume the heap's context when type-checking the heap's contents. For this reason, $\vdash_{\text{stack}}$ and and $\vdash_{\text{rgn}}$ have a $\Delta$, $\Gamma$, and $\gamma$ in their contexts.

$$(\textbf{SRet1})\quad \vdash_{\text{ret}} \texttt{return}\, e \qquad (\textbf{SRet2})\quad \frac{\vdash_{\text{ret}} s_1}{\vdash_{\text{ret}} s_1; s_2}$$

$$(\textbf{SRet3})\quad \frac{\vdash_{\text{ret}} s_2}{\vdash_{\text{ret}} s_1; s_2} \qquad (\textbf{SRet4})\quad \frac{\vdash_{\text{ret}} s_1 \quad \vdash_{\text{ret}} s_2}{\texttt{if}\ (e)\ s_1\ \texttt{else}\ s_2}$$

$$(\textbf{SRet5})\quad \frac{\vdash_{\text{ret}} s}{\begin{array}{l}\vdash_{\text{ret}} \rho{:}\{\tau'\, x_\rho = e;\ s\} \\ \vdash_{\text{ret}} \texttt{region}\langle\rho\rangle\, x_\rho\, s \\ \vdash_{\text{ret}} \rho{:}\{\texttt{open}\,[\alpha, x_\rho] = e;\ s\} \\ \vdash_{\text{ret}} s\,\texttt{pop}[\rho]\end{array}}$$

Figure 16: Statement Must-Return Rules

These judgments "collect" the types of all the values in the heap in their result, if you will. In order to prevent the assumption of nonexistent heap values, **SH** requires that the values collected exactly equal (up to reordering) the assumed $\Gamma$.

The context for $\vdash_{\text{stack}}$ and $\vdash_{\text{rgn}}$ does not need a capability $\epsilon$ because the heap contains only values (not expressions requiring evaluation) and only evaluation requires access to regions. Put another way, heap objects can be in dead regions (or point to dead regions), so long as the program does not access the objects (or dereference them).

Rule **SH** takes a heap $(G, S)$ and "produces" the appropriate $\Delta$, $\Gamma$, and $\gamma$. The $\Gamma$ comes from $\vdash_{\text{stack}}$ and $\vdash_{\text{rgn}}$ as just described. $\Delta$ is simply the region names in $G$ and $S$; as expected, the heap has no free type variables. The interesting part is $\gamma$. Because of the stack-like discipline of regions, we can assume $\rho_i <: \rho_{i+1}$ when $\rho_{i+1}$ is to the right of $\rho_i$ in $S$. In other words, regions to the left of $S$ outlive regions to the right. Type-checking garbage regions may also require outlives relationships (though this point is mostly technical because the garbage regions are inaccessible). Because a garbage region is *already dead*, it is sound to assume that any region outlives a garbage region. That is why $\epsilon_1, \ldots, \epsilon_m$ are unconstrained (provided they are closed under $\Delta$).

When we type-check a top-level statement $s$ (such as in the assumption of Type and Pop Preservation), we use the context that the conclusion of **SH** provides.

### 9.3.3 Return- and Pop-Checking

Progress requires that programs do not get stuck during evaluation. A function that "falls off the end" without returning would be stuck because we would have an "active position" of the form $\texttt{call}\{v\}$, (or at top-level, just $v$). Instead, we insist that all execution paths of a function body (and the top-level program) either diverge or reach a return statement. A simple syntax-directed judgment for statements suffices (but of course is conservative). It appears in Figure 16. We have already seen its uses (rules **SR11**, and **SR15**). It also used to ensure that top-level programs return.

More interesting is proving that Cyclone, in some sense, prohibits memory leaks. One way to say this is that if a program terminates, it deallocates all the regions it allocates. To state this formally, we introduce the mutually inductive judgments $S \vdash_{\text{spop}} s$ and $S \vdash_{\text{epop}} e$ in Figures 17 and 18

If $S \vdash_{\text{spop}} s$ (and similarly for expressions), then execution of $s$ will deallocate *exactly* the regions in $S$, and in the correct order. The order comes from the fact that rule **SP9** requires the popped region to be on the left of $S$—all other regions in $S$ must be popped first by $s$ (which is evaluated before the deallocation caused by the pop statement).

$$(\textbf{SP1})\quad \frac{S \vdash_{\text{epop}} e}{S \vdash_{\text{spop}} e} \qquad (\textbf{SP2})\quad \frac{S \vdash_{\text{epop}} e}{S \vdash_{\text{spop}} \texttt{return}\, e}$$

$$(\textbf{SP3})\quad \frac{S \vdash_{\text{spop}} s_1 \quad \emptyset \vdash_{\text{spop}} s_2}{S \vdash_{\text{spop}} s_1; s_2}$$

$$(\textbf{SP4})\quad \frac{S \vdash_{\text{epop}} e \quad \emptyset \vdash_{\text{spop}} s_1 \quad \emptyset \vdash_{\text{spop}} s_2}{S \vdash_{\text{spop}} \texttt{if}\ (e)\ s_1\ \texttt{else}\ s_2}$$

$$(\textbf{SP5})\quad \frac{\emptyset \vdash_{\text{epop}} e \quad \emptyset \vdash_{\text{spop}} s}{\emptyset \vdash_{\text{spop}} \texttt{while}\ (e)\ s} \qquad (\textbf{SP6})\quad \frac{S \vdash_{\text{epop}} e \quad \emptyset \vdash_{\text{spop}} s}{S \vdash_{\text{spop}} \rho{:}\{\tau\ x_\rho = e;\ s\}}$$

$$(\textbf{SP7})\quad \frac{\emptyset \vdash_{\text{spop}} s}{\emptyset \vdash_{\text{spop}} \texttt{region}\langle\rho\rangle\ x_\rho\ s}$$

$$(\textbf{SP8})\quad \frac{S \vdash_{\text{epop}} e \quad \emptyset \vdash_{\text{spop}} s}{S \vdash_{\text{spop}} \rho{:}\{\texttt{open}\,[\alpha, x_\rho] = e;\ s\}}$$

$$(\textbf{SP9})\quad \frac{S \vdash_{\text{spop}} s}{[\rho \mapsto R]S \vdash_{\text{spop}} s\,\texttt{pop}[\rho]}$$

Figure 17: Statement Pop Rules

$$(\textbf{SE1})\quad \emptyset \vdash_{\text{epop}} x_\rho \qquad (\textbf{SE2})\quad \emptyset \vdash_{\text{epop}} i \qquad (\textbf{SE3})\quad \emptyset \vdash_{\text{epop}} \texttt{region}(\rho)$$

$$(\textbf{SE4})\quad \frac{\emptyset \vdash_{\text{spop}} s}{\emptyset \vdash_{\text{epop}} \rho{:}(\tau_1\, x_\rho) \xrightarrow{\epsilon} \tau_2 = \{s\}} \qquad (\textbf{SE5})\quad \frac{\emptyset \vdash_{\text{epop}} e}{\emptyset \vdash_{\text{epop}} \Lambda\alpha{:}\kappa \rhd \gamma.e}$$

$$(\textbf{SE6})\quad \frac{S \vdash_{\text{epop}} e}{\begin{array}{l} S \vdash_{\text{epop}} e\langle\tau\rangle \\ S \vdash_{\text{epop}} e.i \\ S \vdash_{\text{epop}} *e \\ S \vdash_{\text{epop}} \&e \\ S \vdash_{\text{epop}} \texttt{pack}\,[\tau_1, e]\,\texttt{as}\,\tau_2 \end{array}} \qquad (\textbf{SE7})\quad \frac{S \vdash_{\text{epop}} e_1 \quad \emptyset \vdash_{\text{epop}} e_2}{\begin{array}{l} S \vdash_{\text{epop}} (e_1, e_2) \\ S \vdash_{\text{epop}} \texttt{new}(e_1)e_2 \\ S \vdash_{\text{epop}} e_1(e_2) \\ S \vdash_{\text{epop}} e_1 = e_2 \end{array}}$$

$$(\textbf{SE8})\quad \frac{\emptyset \vdash_{\text{epop}} v \quad S \vdash_{\text{epop}} e_2}{\begin{array}{l} S \vdash_{\text{epop}} (v, e_2) \\ S \vdash_{\text{epop}} \texttt{new}(v)e_2 \\ S \vdash_{\text{epop}} v(e_2) \\ S \vdash_{\text{epop}} v = e_2 \end{array}} \qquad (\textbf{SE9})\quad \frac{S \vdash_{\text{spop}} s}{S \vdash_{\text{epop}} \texttt{call}\{s\}}$$

Figure 18: Expression Pop Rules

$$(\textbf{ST1})\quad \Delta \vdash_{\text{con}} \alpha : \Delta(\alpha) \qquad (\textbf{ST2})\quad \Delta \vdash_{\text{con}} \text{int} : \mathcal{B}$$

$$(\textbf{ST3})\ \frac{\Delta \vdash_{\text{con}} \tau : \mathcal{B}}{\Delta \vdash_{\text{con}} \tau : \mathcal{T}} \qquad (\textbf{ST4})\ \frac{\Delta \vdash_{\text{con}} \rho : \mathcal{R}}{\Delta \vdash_{\text{con}} \text{handle}(\rho) : \mathcal{B}}$$

$$(\textbf{ST5})\ \frac{\Delta \vdash_{\text{con}} \tau_1 : \mathcal{T} \quad \Delta \vdash_{\text{con}} \tau_2 : \mathcal{T} \quad \Delta \vdash_{\text{rset}} \epsilon}{\Delta \vdash_{\text{con}} \tau_1 \xrightarrow{\epsilon} \tau_2 : \mathcal{T}}$$

$$(\textbf{ST6})\ \frac{\Delta \vdash_{\text{con}} \tau_1 : \mathcal{T} \quad \Delta \vdash_{\text{con}} \tau_2 : \mathcal{T}}{\Delta \vdash_{\text{con}} \tau_1 \times \tau_2 : \mathcal{T}} \qquad (\textbf{ST7})\ \frac{\Delta \vdash_{\text{con}} \tau : \mathcal{T} \quad \Delta \vdash_{\text{con}} \rho : \mathcal{R}}{\Delta \vdash_{\text{con}} \tau @ \rho : \mathcal{B}}$$

$$(\textbf{ST8})\ \frac{\Delta, \alpha{:}\kappa \vdash_{\text{rc}} \gamma \quad \Delta, \alpha{:}\kappa \vdash_{\text{con}} \tau : \mathcal{T}}{\begin{array}{c}\Delta \vdash_{\text{con}} \forall \alpha{:}\kappa \rhd \gamma.\tau : \mathcal{T} \\ \Delta \vdash_{\text{con}} \exists \alpha{:}\kappa \rhd \gamma.\tau : \mathcal{T}\end{array}} \quad (\kappa \neq \mathcal{T})$$

$$(\textbf{SC1})\quad \vdash_{\text{tvars}} \emptyset \qquad (\textbf{SC2})\ \frac{\vdash_{\text{tvars}} \Delta \quad \alpha \notin \text{Dom}(\Delta)}{\vdash_{\text{tvars}} \Delta, \alpha : \kappa}$$

$$(\textbf{SC3})\quad \Delta \vdash_{\text{rset}} \emptyset \qquad (\textbf{SC4})\ \frac{\alpha \in \text{Dom}(\Delta)}{\Delta \vdash_{\text{rset}} \alpha} \qquad (\textbf{SC5})\ \frac{\Delta \vdash_{\text{rset}} \epsilon_1 \quad \Delta \vdash_{\text{rset}} \epsilon_1}{\Delta \vdash_{\text{rset}} \epsilon_1 \cup \epsilon_2}$$

$$(\textbf{SC6})\quad \Delta \vdash_{\text{rc}} \emptyset \qquad (\textbf{SC7})\ \frac{\Delta \vdash_{\text{rc}} \gamma \quad \Delta \vdash_{\text{rset}} \epsilon \quad \Delta(\rho) = \mathcal{R}}{\Delta \vdash_{\text{rc}} \gamma, \epsilon <: \rho}$$

$$(\textbf{SC8})\quad \Delta \vdash_{\text{vctxt}} \bullet \qquad (\textbf{SC9})\ \frac{\Delta \vdash_{\text{vctxt}} \Gamma \quad \Delta \vdash_{\text{con}} \rho{:}\mathcal{R} \quad \Delta \vdash_{\text{con}} \tau{:}\mathcal{T} \quad \forall \rho', x_{\rho'} \notin \text{Dom}(\Gamma)}{\Delta \vdash_{\text{vctxt}} \Gamma, x_\rho{:}\tau}$$

$$(\textbf{SC10})\ \frac{\vdash_{\text{tvars}} \Delta \quad \Delta \vdash_{\text{vctxt}} \Gamma \quad \Delta \vdash_{\text{rc}} \gamma \quad \Delta \vdash_{\text{rset}} \epsilon}{\vdash_{\text{ctxt}} \Delta; \Gamma; \gamma; \epsilon}$$

Figure 19: Type and Context Well-Formedness

$$(\mathbf{SC11})\quad \gamma \vdash_{\text{vi}} \alpha \Rightarrow \alpha \qquad (\mathbf{SC12})\quad \frac{\alpha \in \epsilon}{\gamma_1, \epsilon <: \rho, \gamma_2 \vdash_{\text{vi}} \rho \Rightarrow \alpha}$$

$$(\mathbf{SC13})\quad \frac{\gamma \vdash_{\text{vi}} \alpha_1 \Rightarrow \alpha_2 \quad \gamma \vdash_{\text{vi}} \alpha_2 \Rightarrow \alpha_3}{\gamma \vdash_{\text{vi}} \alpha_1 \Rightarrow \alpha_3}$$

$$(\mathbf{SC14})\quad \frac{\text{for all } \alpha_2 \in \epsilon_2 \text{ there exists } \alpha_1 \in \epsilon_1 \text{ such that } \gamma \vdash_{\text{vi}} \alpha_1 \Rightarrow \alpha_2}{\gamma \vdash_{\text{ei}} \epsilon_1 \Rightarrow \epsilon_2}$$

$$(\mathbf{SC15})\quad \gamma \vdash_{\text{ord}} \emptyset \qquad (\mathbf{SC16})\quad \frac{\gamma \vdash_{\text{ord}} \gamma' \quad \gamma \vdash_{\text{ei}} \rho \Rightarrow \epsilon}{\gamma \vdash_{\text{ord}} \gamma', \epsilon <: \rho}$$

Figure 20: Capability Entailment

The judgments are actually more restrictive than "all regions must be deallocated in the correct order". In fact, all the deallocations must occur due to pop statements that contain the active position. So there is no way to derive $[\rho \mapsto R] \vdash_{\text{spop}} \texttt{if}\ (e)\ s_1\,\texttt{pop}[\rho]\ \texttt{else}\ s_2\,\texttt{pop}[\rho]$.

Notice that a value (such as a function body) can never contain an explicit pop statement. We prove this formally, but it is obvious from inspection of the rules.

Using the run-time construct $S$ in the context of a compile-time check is a bit sloppy, but it is not a problem: All that the judgments use is the order of the region-names in $S$. Furthermore, for type-checking "initial programs" (which is all we ever actually do; the rest is for proving type preservation), the judgments degenerate to "this program has no explicit pop statements", which is trivial to check.

### 9.3.4 Technical Details

Only a few judgments remain unexplained. Figure 19 contains several judgments for ensuring $\Delta$, $\Gamma$, $\gamma$, and $\epsilon$ are well-formed. These judgments simply require that a $\Delta$ contains distinct type variables and that the other constructs do not mention type variables not in their context. In the case of $\Gamma$, we further require that the kind of the type of variables is not $\mathcal{R}$, which makes no sense. We also require that $\Gamma$ contains distinct identifiers.

Figure 19 also defines $\vdash_{\text{con}}$, which assigns kinds to types. All of the rules are straightforward. Note that we explicitly prohibit quantifying over "wide types" like products.

Figure 20 defines $\vdash_{\text{vi}}$, $\vdash_{\text{ei}}$, and $\vdash_{\text{ord}}$, each in terms of the previous one. If $\gamma \vdash_{\text{vi}} \alpha_1 \Rightarrow \alpha_2$, then $\gamma$ shows that $\alpha_2$ outlives $\alpha_1$. Put another way, if $\alpha_1$ is live, then $\alpha_2$ is live. The rules just formalize the reflexive (**SC11**), transitive (**SC13**) closure of the syntactic constraints in $\gamma$ (**SC12**). We extend $\vdash_{\text{vi}}$ to $\gamma \vdash_{\text{ei}} \epsilon_1 \Rightarrow \epsilon_2$ with **SC14**. This rule need not be stronger (such as allowing $\gamma \vdash \epsilon \Rightarrow \alpha_2$ for some hypothetical $\vdash$) because of the strict ordering of all regions—using more than one $\alpha_1 \in \epsilon_1$ cannot help given the form of $\gamma$. Finally $\gamma \vdash_{\text{ord}} \gamma'$ extends $\vdash_{\text{ei}}$ to $\gamma'$ in the natural way.

## 10 Theorems

In this section, we prove type soundness. The lemmas are presented in "bottom-up" order so that any necessary assumptions have already been proven. The lemmas mostly follow the conventional structure of a syntactic type-soundness argument, but some features of our language (notably the non-standard definition of substitution, the use of identifiers for heap locations, the modeling of C's "aggregate assignment", and the distinction of left- and right-hand-sides) can make it easy to "lose the forest for the trees". Therefore, we first give a "top-down" overview of the argument.

Many of the theorems have mutually recursive clauses for statements, right-hand-sides and left-hand-sides. In our summary, we mention only the statement clause; the other parts are clearly necessary to prove the statement clause.

We want to prove that a well-formed initial machine state does not "get stuck": either the program runs forever or it returns an integer. Moreover, we want to prove that it deallocates all the regions it allocates. Two preservation lemmas and one progress lemma make this theorem an easy corollary.

Return Preservation just ensures that the "must-return" property ($\vdash_{\text{ret}}$) of a well-typed program is preserved under evaluation.

The other preservation lemma (Type and Pop) is much more interesting. It assumes a well-formed heap in which the live regions can be partitioned into $S_1S_2$ such that $\text{Dom}(S_1)$ is the capability for type-checking the statement $s$ and $S_2$ contains exactly the regions that execution of $s$ deallocates (if it terminates). The conclusion asserts that the heap after the step to $s'$ has live regions of the form $S_1'S_2'$ where $\text{Dom}(S_1') = \text{Dom}(S_1)$, $s'$ type-checks under the same capability as $s$, and $s'$ deallocates all the regions in $S_2'$. Note that $S_2'$ might have more regions or fewer regions than $S_2$. The point is that, at top-level, $S_2$ is all the live regions the program must deallocate and $S_1$ is just the program's static data, which is never deallocated. The stronger formulation with $S_1$ and $S_2$ is necessary for type-checking pop statements.

The Progress Lemma is more conventional. It says that well-typed machines states either are terminal configurations or can take a step. The lemma needs to partition the live regions in $S_1S_2$ like Type and Pop Preservation because "well-typed programs" assume a certain collection of regions $S_1$ remain live while deallocating another collection of regions $S_2$. We cannot just put all the region-names in the capability because of the rule for type-checking pop statements.

As usual, the proof of Progress depends on a Canonical Forms Lemma, which describes the forms of values of particular types. For example, to know that $v_1(v_2)$ is not stuck, we must know that values of function types are functions. Two other lemmas address some technical issues that the structure of our heap causes. First, the Canonical Paths Lemma says that a well-typed path with a live region-annotation corresponds to a location in the live heap that contains a value of the correct type. This conclusion is used for Progress on right-hand-side identifiers (because access to $G$ is forbidden) and assignment (because Update Progress requires the "old value" to have the correct type). Second, Update Progress proves that for a well-typed assignment, the auxiliary update function always applies.

The Constraint Progress Lemma is used to prove the Canonical Paths Lemma. Essentially, it says that regions presumed to outlive live regions are live. The $\gamma$ used to type-check the heap establishes this fact.

The Values Effectless Lemma is used in case **SS6** of Progress and in many cases of Type and Pop Preservation. In part, this Lemma proves that if $S \vdash_{\text{epop}} v$, then $S = \emptyset$. That suffices to show, as is necessary in **SS6**, that the region to be deallocated must be the right-most (most recently allocated) region in the live heap.

We now turn to the auxiliary lemmas for proving Type and Pop Preservation. Substitution Lemmas 10, 12, and 13 are used in the cases where the dynamic step substitutes a type for a type-variable (or region-name). The other Substitution Lemmas are all for proving Substitution Lemma 10. The interesting part is Lemmas 6 and 7, in which we substitute a region-set $\epsilon$ or a region-name $\rho$ for a type-variable or region-name, respectively, and establish that all outlives derivations (in terms of a $\gamma$ through which we do the substitution) are preserved. The Regions-Of Lemma is needed for Substitution Lemma 4.

The Useless Substitution Lemma addresses a complication caused by overloading identifiers as heap locations: The Substitution Lemma says that a statement $s$ continues to type-check after substituting (for example) $\tau$ for $\alpha$ throughout the context and $s$. But the context for "top-level" programs comes from the heap, so we cannot type-check $s[\tau/\alpha]$ under a different context. But $\alpha$ does not occur in the context that comes from the heap, so the substitution is irrelevant (or useless) for the context. (In a conventional presentation, the context for a top-level program is empty, so the same result is trivial.)

The Term Weakening Lemma serves its usual purpose: After we extend the heap, we type-check the old heap elements and the resulting program under a stronger context. The point of Weakening is that nothing fails to type-check as a result. To prove this result, we need the Context Weakening Lemmas for all the static-semantics rules that use the judgments mentioned in these lemmas.

The Context Well-Formedness Lemma is just a technical fact that alleviates our need to constantly mention that we assume all contexts are well-formed.

The Values Effectless Lemma allows us to move a value into the heap, even though heap locations are type-checked under empty capabilities.

The Path Substitution Lemma is needed to establish preservation when an expression of the form $(*(\&x_\rho)).i_1.\cdots.i_n$ rewrites to $v.i_1.\cdots.i_n$.

Finally, we need Update Preservation to establish that a well-typed assignment leaves the assigned-to location with a value of the same type that it had before the assignment.

**Lemma 10.1 (Context Well-Formedness)**
*Let* $C = \Delta;\Gamma;\gamma;\epsilon$.

1. *If* $\vdash_{\text{heap}} (G,S) : \Delta;\Gamma;\gamma$, *then* $\Delta \vdash_{\text{vctxt}} \Gamma$ *and* $\Delta \vdash_{\text{rc}} \gamma$.
2. *If* $C \vdash_{\text{lhs}} e : \tau$, *then* $\vdash_{\text{ctxt}} C$ *and* $\Delta \vdash_{\text{con}} \tau{:}\mathcal{T}$.
3. *If* $C \vdash_{\text{rhs}} e : \tau$, *then* $\vdash_{\text{ctxt}} C$ *and* $\Delta \vdash_{\text{con}} \tau{:}\mathcal{T}$.
4. *If* $C;\tau \vdash_{\text{stmt}} s$, *then* $\vdash_{\text{ctxt}} C$ *and* $\Delta \vdash_{\text{con}} \tau{:}\mathcal{T}$.

**Proof:**

The first part is by inspection of rule **SH**. The rest are by simultaneous induction on the derivations of $C \vdash_{\text{lhs}} e : \tau$, $C \vdash_{\text{rhs}} e : \tau$, and $C;\tau \vdash_{\text{stmt}} s$. There are no difficult cases; explicit well-formedness pre-conditions are in the rules to avoid them.

**Lemma 10.2 (Context Weakening)**

1. *If* $\Delta \vdash_{\text{rset}} \epsilon$ *and* $\vdash_{\text{tvars}} \Delta,\Delta'$, *then* $\Delta,\Delta' \vdash_{\text{rset}} \epsilon$.
2. *If* $\Delta \vdash_{\text{rc}} \gamma$ *and* $\vdash_{\text{tvars}} \Delta,\Delta'$, *then* $\Delta,\Delta' \vdash_{\text{rc}} \gamma$.
3. *If* $\Delta \vdash_{\text{con}} \tau{:}\kappa$ *and* $\vdash_{\text{tvars}} \Delta,\Delta'$, *then* $\Delta,\Delta' \vdash_{\text{con}} \tau{:}\kappa$.
4. *If* $\Delta \vdash_{\text{vctxt}} \Gamma$ *and* $\vdash_{\text{tvars}} \Delta,\Delta'$, *then* $\Delta,\Delta' \vdash_{\text{vctxt}} \Gamma$.
5. *If* $\vdash_{\text{ctxt}} \Delta;\Gamma;\gamma;\epsilon$ *and* $\vdash_{\text{tvars}} \Delta,\Delta'$, *then* $\vdash_{\text{ctxt}} \Delta,\Delta';\Gamma;\gamma;\epsilon$.
6. *If* $\gamma \vdash_{\text{vi}} \alpha_1 \Rightarrow \alpha_2$ *and* $\gamma' \vdash_{\text{ord}} \gamma$, *then* $\gamma' \vdash_{\text{vi}} \alpha_1 \Rightarrow \alpha_2$.
7. *If* $\gamma \vdash_{\text{ei}} \epsilon_1 \Rightarrow \epsilon_2$ *and* $\gamma' \vdash_{\text{ord}} \gamma$, *then* $\gamma' \vdash_{\text{ei}} \epsilon_1 \Rightarrow \epsilon_2$.
8. *If* $\gamma \vdash_{\text{ord}} \gamma''$ *and* $\gamma' \vdash_{\text{ord}} \gamma$, *then* $\gamma' \vdash_{\text{ord}} \gamma''$.

**Proof:**

In each case, by induction on the derivation of the first assumption. The only interesting case is for part 6, rule **SC12**. In this case, the assumption $\gamma' \vdash_{\text{ord}} \gamma$ must have been derived with **SC16**, which has strong enough hypotheses to derive the desired result.

**Lemma 10.3 (Term Weakening)**
*Suppose:*

1. $\vdash_{\text{ctxt}} \Delta\Delta'; \Gamma\Gamma'; \gamma'; \epsilon'$
2. $\gamma' \vdash_{\text{ord}} \gamma$
3. $\gamma' \vdash_{\text{ei}} \epsilon' \Rightarrow \epsilon$

*Then:*

1. *If* $\Delta; \Gamma; \gamma; \epsilon \vdash_{\text{lhs}} e : \tau$*, then* $\Delta\Delta'; \Gamma\Gamma'; \gamma'; \epsilon' \vdash_{\text{lhs}} e : \tau$.
2. *If* $\Delta; \Gamma; \gamma; \epsilon \vdash_{\text{rhs}} e : \tau$*, then* $\Delta\Delta'; \Gamma\Gamma'; \gamma'; \epsilon' \vdash_{\text{rhs}} e : \tau$.
3. *If* $\Delta; \Gamma; \gamma; \epsilon; \tau \vdash_{\text{stmt}} s$*, then* $\Delta\Delta'; \Gamma\Gamma'; \gamma'; \epsilon'; \tau \vdash_{\text{stmt}} s$.
4. *If* $\Delta; \Gamma; \gamma; \epsilon \vdash_{\text{rgn}} R : \Gamma''$*, then* $\Delta\Delta'; \Gamma\Gamma'; \gamma'; \epsilon'; \tau \vdash_{\text{rgn}} R : \Gamma''$.
5. *If* $\Delta; \Gamma; \gamma; \epsilon \vdash_{\text{stack}} S : \Gamma''$*, then* $\Delta\Delta'; \Gamma\Gamma'; \gamma'; \epsilon'; \tau \vdash_{\text{stack}} S : \Gamma''$.

**Proof:**

In each case, by induction on the derivation of the first assumption, appealing to Context Weakening as necessary. (The first three cases are proved simultaneously.)

**Lemma 10.4 (Regions-Of Lemmas)**

1. *If* $\Delta \vdash_{\text{con}} \tau : \kappa$*, then* $\Delta \vdash_{\text{rset}} \text{regions}(\tau)$ *and* $\text{regions}(\tau) \subseteq \text{Dom}(\Delta)$.
2. *If* $\Delta \vdash_{\text{con}} \tau : \mathcal{R}$*, then* $\text{regions}(\tau) = \rho$ *for some* $\rho$.

**Proof:**

Part (1) is by induction on the derivation of $\Delta \vdash_{\text{con}} \tau : \kappa$. The only interesting case is **ST8**. If $\tau'$ is the type under the quantifier, then the induction hypothesis provides $\Delta, \alpha{:}\kappa \vdash_{\text{rset}} \text{regions}(\tau')$ and $\text{regions}(\tau') \subseteq \text{Dom}(\Delta)$. We can write $\text{regions}(\tau')$ as either $\epsilon$ or $\epsilon \cup \alpha$ where $\alpha \notin \epsilon$. So inversion on **SC5** and the definition of regions give the desired result. Part (2) is by inspection of the $\vdash_{\text{con}}$ judgment.

**Lemma 10.5 (Substitution)**

1. *If* $\Delta, \alpha{:}\kappa \vdash_{\text{rset}} \epsilon$ *and* $\Delta \vdash_{\text{rset}} \epsilon'$*, then* $\Delta \vdash_{\text{rset}} \epsilon[\epsilon'/\alpha]$.

   **Proof:**

   The proof is by induction on the derivation of $\Delta, \alpha{:}\kappa \vdash_{\text{rset}} \epsilon$. There are only three cases: **SC3**, **SC4**, and **SC5**.

2. *If* $\Delta, \alpha{:}\mathcal{T} \vdash_{\text{rc}} \gamma$ *and* $\Delta \vdash_{\text{rset}} \epsilon$*, then* $\Delta \vdash_{\text{rc}} \gamma[\epsilon/\alpha]$*.*

   **Proof:**

   The proof is by induction on the derivation of $\Delta, \alpha{:}\mathcal{T} \vdash_{\text{rc}} \gamma$. There are only two cases: **SC6** and **SC7**.

3. *If* $\Delta, \rho{:}\mathcal{R} \vdash_{\text{rc}} \gamma$ *and* $\Delta \vdash_{\text{con}} \rho'{:}\mathcal{R}$*, then* $\Delta \vdash_{\text{rc}} \gamma[\rho'/\rho]$*.*

   **Proof:**

   This proof is similar to the previous one.

4. *If* $\Delta, \alpha{:}\kappa' \vdash_{\text{con}} \tau : \kappa$ *and* $\Delta \vdash_{\text{con}} \tau' : \kappa'$*, then* $\Delta \vdash_{\text{con}} \tau[\tau'/\alpha] : \kappa$*.*

   **Proof:**

   The proof is by induction on the derivation of $\Delta, \alpha{:}\kappa' \vdash_{\text{con}} \tau : \kappa$ and by inspecting rules **ST1** to **ST8**. For **ST5** and **ST7**, we need the Regions-Of Lemma to establish that the substitution is defined and the result is derivable.

5. *If* $\Delta, \alpha{:}\kappa \vdash_{\text{vctxt}} \Gamma$ *and* $\Delta \vdash_{\text{con}} \tau : \kappa$*, then* $\Delta \vdash_{\text{vctxt}} \Gamma[\tau/\alpha]$*.*

   **Proof:**

   The proof is by induction on the derivation of $\Delta, \alpha : \kappa \vdash_{\text{vctxt}} \Gamma$. There are only two cases: **SC8** and **SC9**.

6. *If* $\Delta \vdash_{\text{rc}} \gamma$*,* $\gamma \vdash_{\text{vi}} \alpha_1 \Rightarrow \alpha_2$*, and* $\Delta \vdash_{\text{con}} \alpha_3 : \mathcal{T}$*, then* $\gamma[\epsilon/\alpha_3] \vdash_{\text{ei}} \alpha_1[\epsilon/\alpha_3] \Rightarrow \alpha_2[\epsilon/\alpha_3]$*.*

   **Proof:**

   The proof is by induction on the derivation of $\gamma \vdash_{\text{vi}} \alpha_1 \Rightarrow \alpha_2$.

   **Case SC11**: By the definition of this rule, $\alpha_1 = \alpha_2$. Then $\alpha_1[\epsilon/\alpha_3] = \alpha_2[\epsilon/\alpha_3]$. Clearly, for all $\alpha \in \alpha_2[\epsilon/\alpha_3]$, $\alpha \in \alpha_1[\epsilon/\alpha_3]$, so $\gamma[\epsilon/\alpha_3] \vdash_{\text{vi}} \alpha \Rightarrow \alpha$. By **SC14**, $\gamma[\epsilon/\alpha_3] \vdash_{\text{ei}} \alpha_1[\epsilon/\alpha_3] \Rightarrow \alpha_2[\epsilon/\alpha_3]$.

   **Case SC12**: By definition of this rule,

   $$\frac{\alpha \in \epsilon_0}{\gamma_1, \epsilon_0 <: \rho, \gamma_2 \vdash_{\text{vi}} \rho \Rightarrow \alpha}$$

   In this rule, $\alpha_1 = \rho$, $\alpha_2 = \alpha$. Since $\Delta \vdash_{\text{rc}} \gamma$, $\alpha_1 = \rho : \mathcal{R}$ and $\alpha_1 \neq \alpha_3$. If $\alpha_2 \neq \alpha_3$, we can derive $\alpha_2 \in \epsilon_0[\epsilon/\alpha_3]$ from $\alpha_2 \in \epsilon_0$. By applying **SC12** again we get $\gamma[\epsilon/\alpha_3] \vdash_{\text{vi}} \alpha_1 \Rightarrow \alpha_2$. If $\alpha_2 = \alpha_3$, $\alpha_2[\epsilon/\alpha_3] = \epsilon$. Since $\alpha_2 \in \epsilon_0$, $\epsilon \subseteq \epsilon_0[\epsilon/\alpha_3]$. $\forall \alpha_0 \in \epsilon$, $\alpha_0 \in \epsilon_0[\epsilon/\alpha_3]$. By **SC12** $\rho \Rightarrow \alpha_0$. By **SC14**, $\rho \Rightarrow \epsilon$. So $\gamma[\epsilon/\alpha_3] \vdash_{\text{ei}} \alpha_1 \Rightarrow \alpha_2[\epsilon/\alpha_3]$.

   **Case SC13**:

   $$\frac{\gamma \vdash_{\text{vi}} \alpha_1 \Rightarrow \alpha_0 \quad \gamma \vdash_{\text{vi}} \alpha_0 \Rightarrow \alpha_2}{\gamma \vdash_{\text{vi}} \alpha_1 \Rightarrow \alpha_2}$$

   By induction hypothesis, $\gamma[\epsilon/\alpha_3] \vdash_{\text{ei}} \alpha_1[\epsilon/\alpha_3] \Rightarrow \alpha_0[\epsilon/\alpha_3]$, and $\gamma[\epsilon/\alpha_3] \vdash_{\text{ei}} \alpha_0[\epsilon/\alpha_3] \Rightarrow \alpha_2[\epsilon/\alpha_3]$. By inversion of **SC14**, we know that $\forall \alpha \in \alpha_2[\epsilon/\alpha_3]$, there exists $\alpha' \in \alpha_0[\epsilon/\alpha_3]$, such that $\gamma[\epsilon/\alpha_3] \vdash_{\text{vi}} \alpha' \Rightarrow \alpha$. Similarly, $\forall \alpha' \in \alpha_0[\epsilon/\alpha_3]$, there exists $\alpha'' \in \alpha_1[\epsilon/\alpha_3]$, such that $\gamma[\epsilon/\alpha_3] \vdash_{\text{vi}} \alpha'' \Rightarrow \alpha'$. By applying **SC13** again, we have $\gamma[\epsilon/\alpha_3] \vdash_{\text{vi}} \alpha'' \Rightarrow \alpha$. By applying **SC14** again, we have $\gamma[\epsilon/\alpha_3] \vdash_{\text{ei}} \alpha_1[\epsilon/\alpha_3] \Rightarrow \alpha_2[\epsilon/\alpha_3]$

7. *If* $\Delta \vdash_{\text{rc}} \gamma$*,* $\gamma \vdash_{\text{vi}} \alpha_1 \Rightarrow \alpha_2$*, and* $\Delta \vdash_{\text{con}} \rho : \mathcal{R}$*, then* $\gamma[\rho'/\rho] \vdash_{\text{ei}} \alpha_1[\rho'/\rho] \Rightarrow \alpha_2[\rho'/\rho]$*.*

   **Proof:**

The proof for this lemma is similar to the previous one. The only difference is for **SC12**: We need to consider the case where $\rho$ is substituted.

8. *If* $\Delta \vdash_{\text{rc}} \gamma$, $\gamma \vdash_{\text{ei}} \epsilon_1 \Rightarrow \epsilon_2$, *and* $\Delta \vdash_{\text{con}} \alpha_3 : \mathcal{T}$, *then* $\gamma[\epsilon/\alpha_3] \vdash_{\text{ei}} \epsilon_1[\epsilon/\alpha_3] \Rightarrow \epsilon_2[\epsilon/\alpha_3]$.

   **Proof:**

   We can only derive $\gamma \vdash_{\text{ei}} \epsilon_1 \Rightarrow \epsilon_2$ from rule **SC14**. By inversion of this rule, we have $\forall \alpha_2 \in \epsilon_2$, there exists $\alpha_1 \in \epsilon_1$ such that $\gamma \vdash_{\text{vi}} \alpha_1 \Rightarrow \alpha_2$. Our proof is by induction on the structure of the region set $\epsilon_2$.

   **Case 1:** $\epsilon_2 = \alpha_2$. $\gamma \vdash_{\text{vi}} \alpha_1 \Rightarrow \alpha_2$. So by Lemma 6, $\gamma[\epsilon/\alpha_3] \vdash_{\text{ei}} \alpha_1[\epsilon/\alpha_3] \Rightarrow \alpha_2[\epsilon/\alpha_3]$. Since $\alpha_1 \in \epsilon_1$, $\alpha_1[\epsilon/\alpha_3] \subseteq \epsilon_1[\epsilon/\alpha_3]$. By rule **SC14**, $\gamma[\epsilon/\alpha_3] \vdash_{\text{ei}} \epsilon_1[\epsilon/\alpha_3] \Rightarrow \epsilon_2[\epsilon/\alpha_3]$.

   **Case 2:** $\epsilon_2 = \epsilon_3 \cup \epsilon_4$. By rule **SC14**, $\gamma \vdash_{\text{ei}} \epsilon_1 \Rightarrow \epsilon_2$ means $\gamma \vdash_{\text{ei}} \epsilon_1 \Rightarrow \epsilon_3$ and $\gamma \vdash_{\text{ei}} \epsilon_1 \Rightarrow \epsilon_4$. By induction hypothesis, $\gamma[\epsilon/\alpha_3] \vdash_{\text{ei}} \epsilon_1[\epsilon/\alpha_3] \Rightarrow \epsilon_3[\epsilon/\alpha_3]$ and $\gamma[\epsilon/\alpha_3] \vdash_{\text{ei}} \epsilon_1[\epsilon/\alpha_3] \Rightarrow \epsilon_4[\epsilon/\alpha_3]$. By rule **SC14**, it is obvious that $\gamma[\epsilon/\alpha_3] \vdash_{\text{ei}} \epsilon_1[\epsilon/\alpha_3] \Rightarrow (\epsilon_3 \cup \epsilon_4)[\epsilon/\alpha_3]$.

9. *If* $\Delta \vdash_{\text{rc}} \gamma$, $\gamma \vdash_{\text{ei}} \epsilon_1 \Rightarrow \epsilon_2$, *and* $\Delta \vdash_{\text{con}} \rho : \mathcal{R}$, *then* $\gamma[\rho'/\rho] \vdash_{\text{ei}} \epsilon_1[\rho'/\rho] \Rightarrow \epsilon_2[\rho'/\rho]$.

   **Proof:**

   This proof is similar to the previous one.

10. *If* $\Delta, \alpha{:}\mathcal{T} \vdash_{\text{rc}} \gamma$, $\Delta \vdash_{\text{rset}} \epsilon$, *and* $\gamma \vdash_{\text{ord}} \gamma'$, *then* $\gamma[\epsilon/\alpha] \vdash_{\text{ord}} \gamma'[\epsilon/\alpha]$.
    *If* $\Delta, \rho{:}\mathcal{R} \vdash_{\text{rc}} \gamma$, $\Delta \vdash_{\text{con}} \rho' : \mathcal{R}$, *and* $\gamma \vdash_{\text{ord}} \gamma'$, *then* $\gamma[\rho'/\rho] \vdash_{\text{ord}} \gamma'[\rho'/\rho]$.

    **Proof:**

    The proof is by induction on the derivation of $\gamma \vdash_{\text{ord}} \gamma'$. There are only two cases: **SC15** and **SC16**.

11. *If* $\Delta, \alpha{:}\kappa; \Gamma; \gamma; \epsilon \vdash_{\text{rhs}} e : \tau$ *and* $\Delta \vdash_{\text{con}} \tau' : \kappa$, *then* $\Delta; \Gamma[\tau'/\alpha]; \gamma[\text{regions}(\tau')/\alpha]; \epsilon[\text{regions}(\tau')/\alpha] \vdash_{\text{rhs}} e[\tau'/\alpha] : \tau[\tau'/\alpha]$.
    *If* $\Delta, \alpha{:}\kappa; \Gamma; \gamma; \epsilon \vdash_{\text{lhs}} e : \tau$ *and* $\Delta \vdash_{\text{con}} \tau' : \kappa$, *then* $\Delta; \Gamma[\tau'/\alpha]; \gamma[\text{regions}(\tau')/\alpha]; \epsilon[\text{regions}(\tau')/\alpha] \vdash_{\text{lhs}} e[\tau'/\alpha] : \tau[\tau'/\alpha]$.
    *If* $\Delta, \alpha{:}\kappa; \Gamma; \gamma; \epsilon; \tau \vdash_{\text{stmt}} s$ *and* $\Delta \vdash_{\text{con}} \tau' : \kappa$, *then* $\Delta; \Gamma[\tau'/\alpha]; \gamma[\text{regions}(\tau')/\alpha]; \epsilon[\text{regions}(\tau')/\alpha]; \tau[\tau'/\alpha] \vdash_{\text{stmt}} s[\tau'/\alpha]$.

    Note that substitution of $\tau'$ through $e$, by definition, uses regions($\tau'$) whenever a region-constraint or region-set is encountered.

    **Proof:**

    By simultaneous induction on the derivations of $\Delta, \alpha{:}\kappa; \Gamma; \gamma; \epsilon \vdash_{\text{rhs}} e : \tau$, $\Delta, \alpha{:}\kappa; \Gamma; \gamma; \epsilon \vdash_{\text{lhs}} e : \tau$, and $\Delta, \alpha{:}\kappa; \Gamma; \gamma; \epsilon; \tau \vdash_{\text{stmt}} s$. Proceed by cases on the last rule of the derivation.

    **Cases SS1–SS5**: These cases are just proof by induction.

    **Case SS6**: This is a simple proof by induction. Notice that $\rho$ in `pop`$[\rho]$ is always a region name, but not a region type variable.

    **Cases SS7–SS9**: These are all simple proofs by induction. Notice that $\rho \notin \text{Dom}(\Delta)$, so $\rho \neq \alpha_3$, so the substitution of $\alpha_3$ will not affect the occurrences of $\rho$.

    **Cases SR1–SR16**: The proof for these cases is by induction. We need to use other substitution lemmas throughout the proof.

**Cases SL1–SL3**: The proof for these cases is by induction.

12. *If* $S \vdash_{\text{epop}} e$*, then* $S \vdash_{\text{epop}} e[\tau/\alpha]$*.*
*If* $S \vdash_{\text{spop}} s$*, then* $S \vdash_{\text{spop}} s[\tau/\alpha]$*.*
**Proof:**

The proof is trivial. Clearly expression and statement pop rules do not refer to type variables.

13. *If* $\vdash_{\text{ret}} s$*, then* $\vdash_{\text{ret}} s[\tau/\alpha]$*.*
**Proof:**

The proof is trivial. Clearly statement must-return rules do not refer to type variables.

**Lemma 10.6 (Useless Substitution)**

1. *If* $\alpha \notin \text{Dom}(\Delta)$ *and* $\Delta \vdash_{\text{rset}} \epsilon$*, then* $\epsilon[\text{regions}(\tau)/\alpha] = \epsilon$*.*

2. *If* $\alpha \notin \text{Dom}(\Delta)$ *and* $\Delta \vdash_{\text{rc}} \gamma$*, then* $\gamma[\text{regions}(\tau)/\alpha] = \gamma$*.*

3. *If* $\alpha \notin \text{Dom}(\Delta)$ *and* $\Delta \vdash_{\text{con}} \tau'{:}\kappa$*, then* $\tau'[\tau/\alpha] = \tau'$*.*

4. *If* $\alpha \notin \text{Dom}(\Delta)$ *and* $\Delta \vdash_{\text{vctxt}} \Gamma$*, then* $\Gamma[\tau/\alpha] = \Gamma$*.*

**Proof:**

In each case, by induction on the derivation of the second assumption, using the definition of substitution.

**Lemma 10.7 (Path Substitution)**
*Let* $C = \Delta; \Gamma; \gamma; \epsilon$*. If* $C \vdash_{\text{lhs}} x_\rho.i_1.i_2.\cdots.i_n : \tau@\rho$*, then there exists a* $\tau'$ *such that* $\Gamma(x_\rho) = \tau'$ *and for all* $e$ *such that* $C \vdash_{\text{rhs}} e : \tau'$*,* $C \vdash_{\text{rhs}} e.i_1.i_2.\cdots.i_n : \tau$*.*

**Proof:**

The proof is by induction on $n$. When $n = 0$, we have $C \vdash_{\text{lhs}} x_\rho : \tau@\rho$, which must follow from rule **SL1**. Thus, $\tau' = \tau = \Gamma(x_\rho)$ and from the assumptions, $C \vdash_{\text{rhs}} e : \tau$.

Now assume that the lemma holds for all values up to $n - 1$. We have as an assumption $C \vdash_{\text{lhs}} x_\rho.i_1.i_2.\cdots.i_{n-1}.i_n : \tau@\rho$. This can only hold via **SL3** and thus $C \vdash_{\text{lhs}} x_\rho.i_1.i_2.\cdots.i_{n-1} : (\tau_1 \times \tau_2)@\rho$ such that $\tau = \tau_{i_n}$. From the induction hypothesis, we can therefore conclude that there exists a $\tau'$ such that $\Gamma(x_\rho) = \tau'$ and that for all $e$ such that $C \vdash_{\text{rhs}} e : \tau'$, $C \vdash_{\text{rhs}} e.i_1.i_2.\cdots.i_{n-1} : \tau_1 \times \tau_2$. Then using **SR6**, we can conclude $C \vdash_{\text{rhs}} e.i_1.i_2.\cdots.i_{n-1}.i_n : \tau$.

**Lemma 10.8 (Values Effectless)**

1. $S \vdash_{\text{epop}} p$ *implies* $S = \emptyset$*.*

2. $S \vdash_{\text{epop}} v$ *implies* $S = \emptyset$*.*

3. *If* $\Delta; \Gamma; \gamma; \epsilon \vdash_{\text{lhs}} p : \tau$*, then* $\Delta; \Gamma; \gamma; \emptyset \vdash_{\text{lhs}} p : \tau$*.*

4. *If* $\Delta; \Gamma; \gamma; \epsilon \vdash_{\text{rhs}} v : \tau$*, then* $\Delta; \Gamma; \gamma; \emptyset \vdash_{\text{rhs}} v : \tau$*.*

5. *If* $\Delta; \Gamma; \gamma; \epsilon \vdash_{\text{lhs}} p : \tau$ *and* $\Delta \vdash \epsilon'$*, then* $\Delta; \Gamma; \gamma; \epsilon' \vdash_{\text{lhs}} p : \tau$*.*

*6. If $\Delta;\Gamma;\gamma;\epsilon \vdash_{\text{rhs}} v : \tau$ and $\Delta \vdash \epsilon'$, then $\Delta;\Gamma;\gamma;\epsilon' \vdash_{\text{rhs}} v : \tau$.*

**Proof:**

Each of the parts follows by a straightforward induction. Note that only elimination or allocation forms actually use $\epsilon$ to show that a region or set of regions is accessible and that these are not values (or paths). Also note that verifying the constraints on an existential package does not depend upon $\epsilon$. Part (5) is a corollary to part (3) using Weakening (since $\gamma \vdash_{\text{ei}} \epsilon' \Rightarrow \emptyset$). Similarly, part (6) is a corollary to part (4).

**Lemma 10.9 (Return Preservation)**
*If $\vdash_{\text{ret}} s$ and $(G, S, s) \overset{\text{stmt}}{\longrightarrow} (G', S', s')$, then $\vdash_{\text{ret}} s'$.*

**Proof:**

By induction on the derivation of $(G, S, s) \overset{\text{stmt}}{\longrightarrow} (G', S', s')$.

**case DS1**: $(G, S, v; s_1) \overset{\text{stmt}}{\longrightarrow} (G', S', s_1)$. Now $\vdash_{\text{ret}} v; s_1$ holds must hold via **SRet2**, for we cannot derive $\vdash_{\text{ret}} v$. Therefore, we have $\vdash_{\text{ret}} s_1$.

**cases DS2,DS3**: $(G, S, \texttt{if}\ (0)\ s_1\ \texttt{else}\ s_2) \overset{\text{stmt}}{\longrightarrow} (G, S, s_2)$. Now $\vdash_{\text{ret}}$ $\texttt{if}\ (0)\ s_1\ \texttt{else}\ s_2$ must follow from **SRet4**. Therefore, $\vdash_{\text{ret}} s_2$. The **DS3** case is similar.

**case DS4**: $(G, S, \texttt{while}\ (e)\ s_1) \overset{\text{stmt}}{\longrightarrow} (G, S, \texttt{if}\ (e)\ \{s_1; s\}\ \texttt{else}\ 0)$. This case is trivial because we cannot derive $\vdash_{\text{ret}} s$.

**case DS5, DS6**:
$(G, S, \texttt{return}\, v; s_1 \overset{\text{stmt}}{\longrightarrow} (G, S, \texttt{return}\, v)$ or $(G, S, \texttt{return}\, v\, \texttt{pop}[\rho]) \overset{\text{stmt}}{\longrightarrow} (G', S', \texttt{return}\, v)$. These cases follow trivially since **SRet1** allows us to conclude $\vdash_{\text{ret}} \texttt{return}\, v$.

**case DS7**: We cannot have $\vdash_{\text{ret}} v\, \texttt{pop}[\rho]$ for this requires $\vdash_{\text{ret}} v$.

**case DS8**: These cases follow directly from inversion of **SRet5**, and then using **SRet5** with the $\vdash_{\text{ret}} s\, \texttt{pop}[\rho]$ conclusion to establish the result.

**case DS9**: These cases follow trivially as changes to a sub-expression do not impact whether or not $\vdash_{\text{ret}}$ holds.

**case DS10**: These cases follow directly from the induction hypothesis.

**Lemma 10.10 (New-Region Preservation)**
*Let $S = [\rho_1 \mapsto R_1, \ldots \rho_n \mapsto R_n]$. Suppose:*

- $\vdash_{\text{heap}} (G, S) : \Delta; \Gamma; \gamma$
- $\vdash_{\text{heap}} (G, S[\rho \mapsto R]) : \Delta'; \Gamma'; \gamma, \rho_n <: \rho$

*Then $\gamma, \rho_n <: \rho \vdash_{\text{ord}} \gamma, \text{Dom}(S) <: \rho$.*

**Proof:**

By (**SC16**), we must show $\gamma, \rho_n <: \rho \vdash_{\text{ord}} \gamma$ and $\gamma, \rho_n <: \rho \vdash_{\text{ei}} \text{Dom}(S) <: \rho$. The former is a trivial induction over the size of $\gamma$ using **SC15** and **SC14**, which in turn uses (**SC12**) for each element in a constraint's effect. For the latter, by **SC16** and **SC14**, it suffices to show that $\gamma, \rho_n <: \rho \vdash_{\text{vi}} \rho \Rightarrow \rho_{n-i}$ for $0 \leq i \leq n-1$. We use induction on $i$. For $i = 0$, the constraint $\rho_n <: \rho$ and rule (**SC12**) suffices. For $i > 0$, we know by induction that $\gamma, \rho_n <: \rho \vdash_{\text{vi}} \rho \Rightarrow \rho_{n-(i-1)}$. Furthermore, **SH** and the assumptions ensure that $\gamma$ contains the constraint $\rho_{n-i} <: \rho_{n-(i-1)}$, so by **SC12**, $\gamma, \rho_n <: \rho \vdash_{\text{vi}} \rho_{n-(i-1)} \Rightarrow \rho_{n-i}$. So by **SC13**, we are done.

**Lemma 10.11 (Projection)**
*Suppose* $C \vdash_{\text{rhs}} v.i.i_1.\cdots.i_n : \tau$. *Then* $v = (v_1, v_2)$ *for some* $v_1$ *and* $v_2$, *and* $C \vdash_{\text{rhs}} v_i.i_1.\cdots.i_n : \tau$.

**Proof:**

We argue by induction on $n$. When $n = 0$ we have $C \vdash_{\text{rhs}} v.i : \tau$. We now argue by an inner induction on the derivation of this fact that $v = (v_1, v_2)$ and $C \vdash_{\text{rhs}} v_i : \tau_i$. There are two cases to consider:

**case SR6**: We have:

$$\frac{C \vdash_{\text{rhs}} v : \tau_1 \times \tau_2}{C \vdash_{\text{rhs}} v.i : \tau_i}$$

From the Canonical Forms lemma, we can therefore conclude $v = (v_1, v_2)$. By inversion, we then have $C \vdash v_i : \tau_i$.

**case SR2**: We have:

$$\frac{\Delta; \Gamma; \gamma; \epsilon \vdash_{\text{rhs}} v.i : \tau'@\rho_1 \quad \gamma \vdash_{\text{ei}} \rho_2 \Rightarrow \rho_1}{\Delta; \Gamma; \gamma; \epsilon \vdash_{\text{rhs}} v.i : \tau'@\rho_2}$$

where $\tau_i = \tau'@\rho_2$. From the first premise and the inner induction hypothesis, $v = (v_1, v_2)$ and $\Delta; \Gamma; \gamma; \epsilon \vdash_{\text{rhs}} v_i : \tau'@\rho_1$. Therefore, using the second premise and **SR6** we can conclude $\Delta; \Gamma; \gamma; \epsilon \vdash_{\text{rhs}} v_i : \tau'@\rho_2$. This completes the inner induction and the case for $n = 0$.

Suppose the lemma holds up through $n-1$ and let $C \vdash_{\text{rhs}} v.i.i_1.\cdots.i_{n-1}.i_n : \tau$. Again, we argue by an inner induction on the derivation of this fact. Again, the derivation can end with one of two cases:

**case SR6**: We have:

$$\frac{C \vdash_{\text{rhs}} v.i.i_1.\cdots.i_{n-1} : \tau_1 \times \tau_2}{C \vdash_{\text{rhs}} v.i.i_1.\cdots.i_{n-1}.i_n : \tau_{i_n}}$$

The result follows from the premise and the outer induction hypothesis.

**case SR2**: We have:

$$\frac{\Delta; \Gamma; \gamma; \epsilon \vdash_{\text{rhs}} v.i.i_1.\cdots.i_{n-1}.i_n : \tau'@\rho_1 \quad \gamma \vdash_{\text{ei}} \rho_2 \Rightarrow \rho_1}{\Delta; \Gamma; \gamma; \epsilon \vdash_{\text{rhs}} v.i.i_1.\cdots.i_{n-1}.i_n : \tau'@\rho_2}$$

where $\tau_{i_n} = \tau'@\rho_2$. From the first premise and the inner induction hypothesis, we have $v = (v_1, v_2)$, and $\Delta; \Gamma; \gamma; \epsilon \vdash_{\text{rhs}} v_i.i_1.\cdots.i_{n-1}.i_n : \tau'@\rho_1$. Therefore, using the second premise and **SR6** we can conclude $\Delta; \Gamma; \gamma; \epsilon \vdash_{\text{rhs}} v_i.i_1.\cdots.i_{n-1}.i_n : \tau'@\rho_2$.

**Lemma 10.12 (Lookup Preservation)**
*Suppose:*

1. $\vdash_{\text{heap}} (G, S_1 S_2) : \Delta; \Gamma; \gamma$

*2.* $\gamma \vdash_{\text{ei}} \text{Dom}(S_1) \Rightarrow \rho$, *and*

*3.* $\Delta; \Gamma; \gamma; \text{Dom}(S_1) \vdash_{\text{rhs}} x_\rho : \tau$.

*Then* $S_1 S_2 = S_a[\rho' \mapsto R[x \mapsto v]S_b$ *for some* $S_a$, $\rho'$, $R$, $v$, *and* $S_b$ *and* $\Delta; \Gamma; \gamma; \text{Dom}(S_1) \vdash_{\text{rhs}} v : \tau$.

**Proof:**

By induction on the derivation $\Delta; \Gamma; \gamma; \text{Dom}(S_1) \vdash_{\text{rhs}} x_\rho : \tau$. The proof must end with an application of **SR1** or **SR2**.

**case SR1**: Then $\Gamma(x_\rho) = \tau$ and we can show $\Delta; \Gamma; \gamma; \text{Dom}(S_1) \vdash_{\text{lhs}} x : \tau@\rho$. From this, and assumptions (1) and (2), the result follows using the Canonical Paths lemma and weakening.

**case SR2**: The derivation ends with:

$$\frac{\Delta; \Gamma; \gamma; \text{Dom}(S_1) \vdash_{\text{rhs}} e : \tau'@\rho_1 \quad \gamma \vdash_{\text{ei}} \rho_2 \Rightarrow \rho_1}{\Delta; \Gamma; \gamma; \text{Dom}(S_1) \vdash_{\text{rhs}} x : \tau'@\rho_2}$$

where $\tau = \tau'@\rho_2$. The result then follows from the first premise and the induction hypothesis, using the second premise and **SR2** to establish that the value has type $\tau'@\rho_2$.

**Lemma 10.13 (Update Preservation)**
*Suppose*

*1.* $\text{update}(v_a, [i_n, i_{n-1}, \ldots, i_2, i_1], v_b) = v$

*2.* $\Delta; \Gamma; \gamma; \emptyset \vdash_{\text{rhs}} v_a : \tau$

*3.* $\Delta; \Gamma; \gamma; \epsilon \vdash_{\text{rhs}} v_a.i_n.i_{n-1}.\cdots.i_2.i_1 : \tau'$

*4.* $\Delta; \Gamma; \gamma; \epsilon \vdash_{\text{rhs}} v_b : \tau'$

*Then* $\Delta; \Gamma; \gamma; \epsilon \vdash_{\text{rhs}} v : \tau$.

**Proof:**

By induction on $n$. For $n = 0$, we have from (2) and (3) that $\tau = \tau'$. Furthermore, $\text{update}(v_a, [], v_b) = v_b$. The result then follows from (4).

Suppose the lemma holds for all values up through $n - 1$. Without loss of generality, assume that $i_n = 1$. Then from the definition of update, it must be that $v_a = (v_1, v_2)$ for some values $v_1$ and $v_2$ and $v = (\text{update}(v_1, [i_{n-1}, \ldots, i_2, i_1], v_b), v_2)$. From (2) and inversion of the **SR** rules, we know that $\tau = \tau_1 \times \tau_2$ and that $\Delta; \Gamma; \gamma; \emptyset \vdash_{\text{rhs}} v_i : \tau_i$. By Projection, we know $\Delta; \Gamma; \gamma; \emptyset \vdash_{\text{rhs}} v_1.i_{n-1}, \ldots, i_1 : \tau'$. So by the induction hypothesis, we then have $\Delta; \Gamma; \gamma; \epsilon \vdash_{\text{rhs}} \text{update}(v_1, [i_{n-1}, \ldots, i_2, i_1], v_b) : \tau_1$. Therefore, by **SR5** we have $\Delta; \Gamma; \gamma; \epsilon \vdash_{\text{rhs}} v : \tau_1 \times \tau_2$.

**Definition 10.14 (Extensions)**

*1. If* $\text{Dom}(\Delta') \supseteq \text{Dom}(\Delta)$ *and* $\Delta$ *and* $\Delta'$ *agree on* $\text{Dom}(\Delta)$*, then* $\Delta'$ *extends* $\Delta$.

*2. If* $\text{Dom}(\Gamma') \supseteq \text{Dom}(\Gamma)$ *and* $\Gamma$ *and* $\Gamma'$ *agree on* $\text{Dom}(\Gamma)$*, then* $\Gamma'$ *extends* $\Gamma$.

*3. If* $\gamma' = \gamma, \gamma''$*, then* $\gamma'$ *extends* $\gamma$.

4. *If $S = [\rho_1 \mapsto R_1, \ldots, \rho_n \mapsto R_n]$ and $S' = [\rho_1 \mapsto R'_1, \ldots, \rho_n \mapsto R'_n]$, then $S'$ extends $S$. (We could require each $R'_i$ to have a larger domain than $R_i$, but this fact will follow from the heap type-checking under an extended $\Gamma$.)*

**Lemma 10.15 (Type and Pop Preservation)**

1. *If*
   - $\vdash_{\text{heap}} (G, S_1S_2) : \Delta; \Gamma; \gamma$
   - $\Delta; \Gamma; \gamma; \text{Dom}(S_1); \tau \vdash_{\text{stmt}} s$
   - $S_2 \vdash_{\text{spop}} s$
   - $(G, S_1S_2, s) \xrightarrow{\text{stmt}} (G', S', s')$

   *Then there exists $S'$ and $\Delta', \Gamma', \gamma'$ extending $\Delta, \Gamma, \gamma$ respectively such that*
   - $S' = S'_1S'_2$ *and* $S'_1$ *extends* $S_1$
   - $\vdash_{\text{heap}} (G', S') : \Delta'; \Gamma'; \gamma'$
   - $\Delta'; \Gamma'; \gamma'; \text{Dom}(S'_1); \tau \vdash_{\text{stmt}} s'$
   - $S'_2 \vdash_{\text{spop}} s'$
2. *If*
   - $\vdash_{\text{heap}} (G, S_1S_2) : \Delta; \Gamma; \gamma$
   - $\Delta; \Gamma; \gamma; \text{Dom}(S_1) \vdash_{\text{rhs}} e : \tau$
   - $S_2 \vdash_{\text{epop}} e$
   - $(G, S_1S_2, e) \xrightarrow{\text{rhs}} (G', S', e')$

   *Then there exists $S'$ and $\Delta', \Gamma', \gamma'$ extending $\Delta, \Gamma, \gamma$ respectively such that*
   - $S' = S'_1S'_2$ *and* $S'_1$ *extends* $S_1$
   - $\vdash_{\text{heap}} (G', S') : \Delta'; \Gamma'; \gamma'$
   - $\Delta'; \Gamma'; \gamma'; \text{Dom}(S'_1) \vdash_{\text{rhs}} e' : \tau$
   - $S'_2 \vdash_{\text{epop}} e'$
3. *If*
   - $\vdash_{\text{heap}} (G, S_1S_2) : \Delta; \Gamma; \gamma$
   - $\Delta; \Gamma; \gamma; \text{Dom}(S_1) \vdash_{\text{lhs}} e : \tau$
   - $S_2 \vdash_{\text{epop}} e$
   - $(G, S_1S_2, e) \xrightarrow{\text{lhs}} (G', S', e')$

   *Then there exists $S'$ and $\Delta', \Gamma', \gamma'$ extending $\Delta, \Gamma, \gamma$ respectively such that*
   - $S' = S'_1S'_2$ *and* $S'_1$ *extends* $S_1$
   - $\vdash_{\text{heap}} (G', S') : \Delta'; \Gamma'; \gamma'$
   - $\Delta'; \Gamma'; \gamma'; \text{Dom}(S'_1) \vdash_{\text{lhs}} e' : \tau$

- $S_2' \vdash_{\text{epop}} e'$

**Proof:**

By simultaneous induction on the derivations of $\Delta; \Gamma; \gamma; \text{Dom}(S_1); \tau \vdash_{\text{stmt}} s$, $\Delta; \Gamma; \gamma; \text{Dom}(S_1) \vdash_{\text{rhs}} e : \tau$, and $\Delta; \Gamma; \gamma; \text{Dom}(S_1) \vdash_{\text{lhs}} e : \tau@\rho$, proceeding by cases on the last rule in the derivation.

**case SS1**: The derivation ends with:

$$\frac{\Delta; \Gamma; \gamma; \text{Dom}(S_1) \vdash_{\text{rhs}} e : \tau' \quad \Delta \vdash_{\text{con}} \tau{:}\mathcal{T}}{\Delta; \Gamma; \gamma; \text{Dom}(S_1); \tau \vdash_{\text{stmt}} e}$$

The only dynamic rule that applies is **DS9**. Thus, $(G, S_1S_2, e) \xrightarrow{\text{rhs}} (G', S', e')$. Inversion on $S_2 \vdash_{\text{spop}} e$ (rule **SP1**) provides $S_2 \vdash_{\text{epop}} e$. So by induction (the right-hand-side expression part), **SS1** (with Context Weakening to show $\Delta' \vdash_{\text{con}} \tau{:}\mathcal{T}$), and **SP1**, we can derive all the results we need.

**case SS2**: The derivation ends with:

$$\frac{\Delta; \Gamma; \gamma; \text{Dom}(S_1) \vdash_{\text{rhs}} e : \tau}{\Delta; \Gamma; \gamma; \text{Dom}(S_1); \tau \vdash_{\text{stmt}} \texttt{return}\, e}$$

The only dynamic rule that applies is **DS9**. The argument is analogous to case **SS1**, using **SS2** and **SP2** in place of **SS1** and **SP1**.

**case SS3**: The derivation ends with:

$$\frac{\Delta; \Gamma; \gamma; \text{Dom}(S_1); \tau \vdash_{\text{stmt}} s_1 \quad \Delta; \Gamma; \gamma; \text{Dom}(S_1); \tau \vdash_{\text{stmt}} s_2}{\Delta; \Gamma; \gamma; \text{Dom}(S_1); \tau \vdash_{\text{stmt}} s_1; s_2}$$

Inversion on $S_2 \vdash_{\text{spop}} s_1; s_2$ (rule **SP3**) provides $S_2 \vdash_{\text{spop}} s_1$, and $\emptyset \vdash_{\text{spop}} s_2$.

If $s_1 = v$, then only dynamic rule **DS1** applies. By Values Effectless and inversion on $S_2 \vdash_{\text{spop}} v$ (rule **SP1**), we know $S_2 = \emptyset$, so $S_2 \vdash_{\text{spop}} s_2$. Along with $\Delta; \Gamma; \gamma; \text{Dom}(S_1); \tau \vdash_{\text{stmt}} s_2$ and the original assumptions, we can conclude everything we need by letting $G' = G$, $S' = S_1S_2$, $\Delta = \Delta'$, $\Gamma' = \Gamma$, $\gamma' = \gamma$, and $s' = s_2$.

If $s_1 = \texttt{return}\, v$, then only dynamic rule **DS5** applies. From $\Delta; \Gamma; \gamma; \text{Dom}(S_1); \tau \vdash_{\text{stmt}} s_1$, $S_2 \vdash_{\text{spop}} s_1$, and the original assumptions, we can conclude everything we need by letting $G' = G$, $S' = S_1S_2$, $\Delta = \Delta'$, $\Gamma' = \Gamma$, $\gamma' = \gamma$, and $s' = s_1$.

Else $s_1$ is not terminal, in which case only dynamic rule **DS10** applies. Thus, $(G, S_1S_2, s_1; s_2) \xrightarrow{\text{stmt}} (G', S', s_1'; s_2)$. So by induction (the statement part), **SS3** (with Term Weakening and $\text{Dom}(S_1') = \text{Dom}(S_1)$ to type-check $s_2$ under the new context), and **SP3** (using $\emptyset \vdash_{\text{spop}} s_2$), we can derive all the results we need.

**case SS4**: The derivation ends with

$$\frac{\Delta; \Gamma; \gamma; \text{Dom}(S_1) \vdash_{\text{rhs}} e : \text{int} \quad \Delta; \Gamma; \gamma; \text{Dom}(S_1); \tau \vdash_{\text{stmt}} s_1 \quad \Delta; \Gamma; \gamma; \text{Dom}(S_1); \tau \vdash_{\text{stmt}} s_2}{\Delta; \Gamma; \gamma; \text{Dom}(S_1); \tau \vdash_{\text{stmt}} \texttt{if}\ (e)\ s_1\ \texttt{else}\ s_2}$$

Inversion on $S_2 \vdash_{\text{spop}} \texttt{if}\ (e)\ s_1\ \texttt{else}\ s_2$ (rule **SP4**) provides $S_2 \vdash_{\text{epop}} e$, $\emptyset \vdash_{\text{spop}} s_1$, and $\emptyset \vdash_{\text{spop}} s_2$.

If $e = 0$, then only dynamic rule **DS2** applies. By Values Effectless and $S_2 \vdash_{\text{epop}} 0$, we know $S_2 = \emptyset$, so $S_2 \vdash_{\text{spop}} s_2$. Along with $\Delta; \Gamma; \gamma; \text{Dom}(S_1); \tau \vdash_{\text{stmt}} s_2$ and the original assumptions, we can conclude everything we need by letting $G' = G$, $S' = S_1S_2$, $\Delta = \Delta'$, $\Gamma' = \Gamma$, $\gamma' = \gamma$, and $s' = s_2$.

If $e = i \neq 0$, then only dynamic rule **DS3** applies. The argument is analogous to when $e = 0$, replacing $s_2$ with $s_1$.

Else only dynamic rule **DS9** applies, in which case we know $(G, S_1S_2, \texttt{if}\ (e)\ s_1\ \texttt{else}\ s_2) \overset{\text{stmt}}{\longrightarrow} (G', S', \texttt{if}\ (e')\ s_1\ \texttt{else}\ s_2)$. So by induction (the right-hand expression part), **SS4** (with Term Weakening and $\mathrm{Dom}(S_1') = \mathrm{Dom}(S_1)$ to type-check $s_1$ and $s_2$ under the new context), and **SP3** (using $\emptyset \vdash_{\text{spop}} s_1$ and $\emptyset \vdash_{\text{spop}} s_2$), we can derive all the results we need.

**case SS5:** The derivation ends with

$$\frac{\Delta;\Gamma;\gamma;\mathrm{Dom}(S_1) \vdash_{\text{rhs}} e : \mathrm{int} \quad \Delta;\Gamma;\gamma;\mathrm{Dom}(S_1);\tau \vdash_{\text{stmt}} s_1}{\Delta;\Gamma;\gamma;\mathrm{Dom}(S_1);\tau \vdash_{\text{stmt}} \texttt{while}\ (e)\ s_1}$$

Inversion on $S_2 \vdash_{\text{spop}} \texttt{while}\ (e)\ s$ (rule **SP5**) provides $S_2 = \emptyset$, $\emptyset \vdash_{\text{epop}} e$, and $\emptyset \vdash_{\text{spop}} s_1$. Only dynamic rule **DS4** applies. Letting $G' = G$, $S' = S_1S_2$, $\Delta = \Delta'$, $\Gamma' = \Gamma$, $\gamma' = \gamma$, and $s' = \texttt{if}\ (e)\ \{s_1;\texttt{while}\ (e)\ s_1\}\ \texttt{else}\ 0$, we need to show $C;\tau \vdash_{\text{stmt}} \texttt{if}\ (e)\ \{s_1;\texttt{while}\ (e)\ s_1\}\ \texttt{else}\ 0$ where $C = \Delta;\Gamma;\gamma;\mathrm{Dom}(S_1)$. We do so as follows:

$$\frac{C \vdash_{\text{rhs}} e : \mathrm{int} \quad \dfrac{C;\tau \vdash_{\text{stmt}} s_1 \quad C;\tau \vdash_{\text{stmt}} \texttt{while}\ (e)\ s_1}{C;\tau \vdash_{\text{stmt}} s_1;\texttt{while}\ (e)\ s_1} \quad \dfrac{C \vdash_{\text{rhs}} 0 : \mathrm{int}}{C;\tau \vdash_{\text{stmt}} 0}}{C;\tau \vdash_{\text{stmt}} \texttt{if}\ (e)\ \{s_1;\texttt{while}\ (e)\ s_1\}\ \texttt{else}\ 0}$$

We also must show $\emptyset \vdash_{\text{spop}} \texttt{if}\ (e)\ \{s_1;\texttt{while}\ (e)\ s_1\}\ \texttt{else}\ 0$:

$$\frac{\emptyset \vdash_{\text{epop}} e \quad \dfrac{\emptyset \vdash_{\text{spop}} s_1 \quad \emptyset \vdash_{\text{spop}} \texttt{while}\ (e)\ s_1}{\emptyset \vdash_{\text{spop}} s_1;\texttt{while}\ (e)\ s_1} \quad \dfrac{\emptyset \vdash_{\text{epop}} 0}{\emptyset \vdash_{\text{spop}} 0}}{\emptyset \vdash_{\text{spop}} \texttt{if}\ (e)\ \{s_1;\texttt{while}\ (e)\ s_1\}\ \texttt{else}\ 0}$$

**case SS6:** The derivation ends with:

$$\frac{\Delta;\Gamma;\gamma;\mathrm{Dom}(S_1) \uplus \rho;\tau \vdash_{\text{stmt}} s_1 \quad \Delta \vdash_{\text{con}} \rho{:}\mathcal{R}}{\Delta;\Gamma;\gamma;\mathrm{Dom}(S_1);\tau \vdash_{\text{stmt}} s_1\,\texttt{pop}[\rho]}$$

Inversion on $S_2 \vdash_{\text{spop}} s_1\,\texttt{pop}[\rho]$ (rule **SP9**) provides $S_2 = [\rho \mapsto R]S_3$ for some $R$ and $S_3 \vdash_{\text{spop}} s_1$.

If $s_1 = v$, then only dynamic rule **DS7** applies. By Values Effectless and inversion on $S_3 \vdash_{\text{spop}} v$ (rule **SP1**), we know $S_3 = \emptyset$. By inversion on $\Delta;\Gamma;\gamma;\mathrm{Dom}(S_1) \uplus \rho;\tau \vdash_{\text{stmt}} v$ (rule **SS1**), we know $\Delta;\Gamma;\gamma;\mathrm{Dom}(S_1) \uplus \rho \vdash_{\text{rhs}} v : \tau'$. So by Values Effectless, $\Delta;\Gamma;\gamma;\mathrm{Dom}(S_1) \vdash_{\text{rhs}} v : \tau'$, and then by **SS1**, $\Delta;\Gamma;\gamma;\mathrm{Dom}(S_1);\tau \vdash_{\text{stmt}} v$. Letting $G' = G[\rho \mapsto R]$, $S_1' = S_1$, and $S_2' = \emptyset$, all that remains is showing $\vdash_{\text{heap}} (G', S_1\emptyset) : \Delta;\Gamma;\gamma$. Note that we do not change the typing context. From inversion (**SH**) on the assumption $\vdash_{\text{heap}} (G, S_1S_2) : \Delta;\Gamma;\gamma$, we know we have a derivation of the form:

$$\frac{\begin{array}{c} G = [\rho_1' \mapsto R_1', \ldots, \rho_m' \mapsto R_m'] \qquad S_1[\rho \mapsto R] = [\rho_1 \mapsto R_1, \ldots, \rho_n \mapsto R_n][\rho \mapsto R] \\ \Delta = \rho_1'{:}\mathcal{R}, \ldots, \rho_m'{:}\mathcal{R}, \rho_1{:}\mathcal{R}, \ldots, \rho_n{:}\mathcal{R}, \rho{:}\mathcal{R} \\ \gamma_G = \epsilon_1 <: \rho_1', \ldots, \epsilon_m <: \rho_m' \qquad \gamma_S = \rho_1 <: \rho_2, \rho_2 <: \rho_3, \ldots, \rho_{n-1} <: \rho_n, \rho_n <: \rho \\ \gamma = \gamma_G, \gamma_S \qquad \Delta \vdash_{\text{rc}} \gamma \\ \Gamma = \Gamma_G \uplus \Gamma_S \qquad \Delta \vdash_{\text{vctxt}} \Gamma \\ \Delta;\Gamma;\gamma \vdash_{\text{stack}} G : \Gamma_G \qquad \Delta;\Gamma;\gamma \vdash_{\text{stack}} S_1[\rho \mapsto R] : \Gamma_S \end{array}}{\vdash_{\text{heap}} (G, S_1[\rho \mapsto R]) : \Delta;\Gamma;\gamma}$$

By inversion (**SStk2**) on $\Delta;\Gamma;\gamma \vdash_{\text{stack}} S_1[\rho \mapsto R] : \Gamma_S$, we know $\Delta;\Gamma;\gamma \vdash_{\text{stack}} S_1 : \Gamma_S'$, $\Delta;\Gamma;\gamma \vdash_{\text{rgn}} \rho \mapsto R : \Gamma_\rho$, and $\Gamma_S = \Gamma_S' \uplus \Gamma_\rho$. From the second fact and the assumption $\Delta;\Gamma;\gamma \vdash_{\text{stack}} G : \Gamma_G$,

**SStk2** lets us conclude $\Delta;\Gamma;\gamma \vdash_{\text{stack}} G[\rho \mapsto R] : \Gamma_G \uplus \Gamma_\rho$. By reordering on variable contexts, $\Gamma = (\Gamma_G \uplus \Gamma_\rho) \uplus \Gamma'_S$. Letting $\gamma'_G = \gamma_G, \rho_n <: \rho$ (or $\gamma_G$ if $S_1 = \emptyset$) and $\gamma'_S = \rho_1 <: \rho_2, \rho_2 <: \rho_3, \ldots, \rho_{n-1} <: \rho_n$, reordering on constraints gives us $\gamma = \gamma'_G, \gamma'_S$. Finally, reordering on type contexts gives us $\Delta = \rho'_1{:}\mathcal{R}, \ldots, \rho'_m{:}\mathcal{R}, \rho{:}\mathcal{R}, \rho_1{:}\mathcal{R}, \ldots, \rho_n{:}\mathcal{R}$. Putting all this together, we can derive what we want:

$$\frac{\begin{array}{c} G[\rho \mapsto R] = [\rho'_1 \mapsto R'_1, \ldots, \rho'_m \mapsto R'_m][\rho \mapsto R] \qquad S_1 = [\rho_1 \mapsto R_1, \ldots, \rho_n \mapsto R_n] \\ \Delta = \rho'_1{:}\mathcal{R}, \ldots, \rho'_m{:}\mathcal{R}, \rho{:}\mathcal{R}, \rho_1{:}\mathcal{R}, \ldots, \rho_n{:}\mathcal{R} \\ \gamma'_G = \epsilon_1 <: \rho'_1, \ldots, \epsilon_m <: \rho'_m, \rho_n <: \rho \qquad \gamma'_S = \rho_1 <: \rho_2, \rho_2 <: \rho_3, \ldots, \rho_{n-1} <: \rho_n \\ \gamma = \gamma'_G, \gamma'_S \qquad \Delta \vdash_{\text{rc}} \gamma \\ \Gamma = (\Gamma_G \uplus \Gamma_\rho) \uplus \Gamma'_S \qquad \Delta \vdash_{\text{vctxt}} \Gamma \\ \Delta;\Gamma;\gamma \vdash_{\text{stack}} G[\rho \mapsto R] : \Gamma_G \uplus \Gamma_\rho \qquad \Delta;\Gamma;\gamma \vdash_{\text{stack}} S_1 : \Gamma'_S \end{array}}{\vdash_{\text{heap}} (G[\rho \mapsto R], S_1) : \Delta;\Gamma;\gamma}$$

If $s_1 = \texttt{return}\, v$, then the argument is analogous to the previous one, using **SS2** and **SP2** in place of **SS1** and **SP1**.

Else $s_1$ is not terminal, in which case only dynamic rule **DS10** applies. Thus, $(G, S_1S_2, s_1\, \texttt{pop}[\rho] \xrightarrow{\text{stmt}} (G', S', s'_1\, \texttt{pop}[\rho])$. Let $S_4 = S_1[\rho \mapsto R]$. Then by the induction hypothesis, there exist $G'$, $S'_4$ extending $S_4$, $S'_3$, and $\Delta'$, $\Gamma'$, $\gamma$ extending $\Delta$, $\Gamma$, $\gamma$ respectively such that

- $S' = S'_4 S'_3$
- $\vdash_{\text{heap}} (G', S') : \Delta';\Gamma';\gamma'$
- $\Delta';\Gamma';\gamma;\text{Dom}(S'_4);\tau \vdash_{\text{stmt}} s'_1$
- $S'_3 \vdash_{\text{spop}} s'_1$

By the definition of extension, $S'_4 = S'_1[\rho \mapsto R']$ for some $S'_1$ and $R'$ and, furthermore, $S'_1$ extends $S_1$. Letting $S'_2 = [\rho \mapsto R']S'_3$, we have $\vdash_{\text{heap}} (G', S'_1S'_2) : \Delta';\Gamma';\gamma'$. By **SP9**, $S'_2 \vdash_{\text{spop}} s'_1\, \texttt{pop}[\rho]$. Finally, $\text{Dom}(S'_4) = \text{Dom}(S'_1) \uplus \rho$, so by **SS6** (using Context Weakening to show $\Delta' \vdash_{\text{con}} \rho{:}\mathcal{R}$), we have
$\Delta';\Gamma';\gamma;\text{Dom}(S'_1);\tau \vdash_{\text{stmt}} s'_1\, \texttt{pop}[\rho]$.

**case SS7**: The derivation ends with

$$\frac{\begin{array}{c} \Delta;\Gamma;\gamma;\text{Dom}(S_1) \vdash_{\text{rhs}} e : \tau' \quad \Delta \vdash_{\text{con}} \tau{:}\mathcal{T} \\ (\Delta, \rho{:}\mathcal{R}); (\Gamma, x_\rho{:}\tau'); (\gamma, \text{Dom}(S_1) <: \rho); (\text{Dom}(S_1) \cup \rho); \tau \vdash_{\text{stmt}} s_1 \end{array}}{\Delta;\Gamma;\gamma;\text{Dom}(S_1);\tau \vdash_{\text{stmt}} \rho{:}\{\tau'\, x_\rho = e;\ s_1\}} \quad (\rho \notin \text{Dom}(\Delta), x_\rho \notin \text{Dom}(\Gamma))$$

Inversion on $S_2 \vdash_{\text{spop}} \rho{:}\{\tau'\, x_\rho = e;\ s_1\}$ (rule **SP6**) provides $S_2 \vdash_{\text{epop}} e$ and $\emptyset \vdash_{\text{epop}} s_1$.

If $e = v$, then only dynamic rule **DS8a** applies. Let $G' = G$, $S'_1 = S_1$, $S'_2 = [\rho \mapsto \{x \mapsto v\}]$, $\Delta' = \Delta, \rho{:}\mathcal{R}$, and $\Gamma' = \Gamma, x_\rho{:}\tau'$. If $S_1S_2 = [\rho_1 \mapsto R_1, \ldots, \rho_n \mapsto R_n]$, then let $\gamma' = \gamma, \rho_n <: \rho$. (If $S_1S_2 = \emptyset$, which wouldn't actually happen, let $\gamma' = \gamma$.) By Values Effectless, $S_2 \vdash_{\text{epop}} v$ means $S_2 = \emptyset$, so $S_2 \vdash_{\text{spop}} s_1$. So by **SP9**, $S'_2 \vdash_{\text{spop}} s_1\, \texttt{pop}[\rho]$, as needed. Next we show $\vdash_{\text{heap}} (G', S_1S'_2) : \Delta';\Gamma';\gamma'$. By assumption, there must be a derivation of the form:

$$\frac{\begin{array}{c} G = [\rho'_1 \mapsto R'_1, \ldots, \rho'_m \mapsto R'_m] \qquad S_1 = [\rho_1 \mapsto R_1, \ldots, \rho_n \mapsto R_n] \\ \Delta = \rho'_1{:}\mathcal{R}, \ldots, \rho'_m{:}\mathcal{R}, \rho_1{:}\mathcal{R}, \ldots, \rho_n{:}\mathcal{R} \\ \gamma_G = \epsilon_1 <: \rho'_1, \ldots, \epsilon_m <: \rho'_m \qquad \gamma_S = \rho_1 <: \rho_2, \rho_2 <: \rho_3, \ldots, \rho_{n-1} <: \rho_n \\ \gamma = \gamma_G, \gamma_S \qquad \Delta \vdash_{\text{rc}} \gamma \\ \Gamma = \Gamma_G \uplus \Gamma_S \qquad \Delta \vdash_{\text{vctxt}} \Gamma \\ \Delta;\Gamma;\gamma \vdash_{\text{stack}} G : \Gamma_G \qquad \Delta;\Gamma;\gamma \vdash_{\text{stack}} S_1 : \Gamma_S \end{array}}{\vdash_{\text{heap}} (G, S_1) : \Delta;\Gamma;\gamma}$$

Furthermore, by Term Weakening, our original inversion, and Values Effectless, we know that $\Delta';\Gamma';\gamma';\emptyset \vdash_{\text{rhs}} v : \tau'$. We also know $\emptyset \vdash_{\text{epop}} v$. So by **SRgn2** and **SRgn1**, we know $\Delta';\Gamma';\gamma' \vdash_{\text{rgn}} \rho \mapsto \{x \mapsto v\} : x_\rho{:}\tau'$. By Term Weakening, we also have $\Delta';\Gamma';\gamma' \vdash_{\text{stack}} G : \Gamma_G$ and $\Delta';\Gamma';\gamma' \vdash_{\text{stack}} S_1 : \Gamma_S$. So by **Sstk2**, we have $\Delta';\Gamma';\gamma' \vdash_{\text{stack}} S_1 S_2' : \Gamma_S, x_\rho{:}\tau'$. From Weakening and our original inversion, we know $\Delta' \vdash_{\text{con}} \tau'{:}\mathcal{T}$. Because $\Delta \vdash_{\text{vctxt}} \Gamma$, Weakening and **SC9** ensures $\Delta' \vdash_{\text{vctxt}} \Gamma'$. Similarly, we can show that $\Delta' \vdash_{\text{rc}} \gamma'$. Putting all this together, **SH** gives us that $\vdash_{\text{heap}} (G', S') : \Delta';\Gamma';\gamma'$. Furthermore, that means New-Region Preservation Lemma applies, so $\gamma' \vdash_{\text{ord}} \gamma, \text{Dom}(S_1) <: \epsilon$. So by Term Weakening and our original inversion, $\Delta';\Gamma';\gamma';(\text{Dom}(S_1) \cup \rho \vdash_{\text{stmt}} s_1$. So by **SS6**, $\Delta';\Gamma';\gamma';\text{Dom}(S_1) \vdash_{\text{stmt}} s_1\,\texttt{pop}[\rho]$, which is our last obligation.

Else $e$ is not a value, in which case only dynamic rule **DS9** applies. By induction (the right-hand-side expression part), **SS7** (with Term Weakening to show $(\Delta', \rho{:}\mathcal{R});(\Gamma', x_\rho{:}\tau');(\gamma', \text{Dom}(S_1') <: \rho);(\text{Dom}(S_1') \cup \rho);\tau \vdash_{\text{stmt}} s_1$ and Context Weakening to show $\Delta' \vdash_{\text{con}} \tau{:}\mathcal{T}$), and **SP6** (using $\emptyset \vdash_{\text{epop}} s_1$), we can derive all the results we need.

**case SS8**: We use almost the same argument as in the previous case when $e$ was a value $v$. We use $\texttt{region}(\rho)$ for $v$ and handle$(\rho)$ for $\tau'$. We know $S_2 = \emptyset$ directly from **SP7**. Because the inversion of the typing judgment uses **SS8** and **SP7**, we cannot use the same argument to show $\Delta';\Gamma'\gamma' \vdash_{\text{rhs}} v : \tau'$ and $\emptyset \vdash_{\text{epop}} \texttt{region}(\rho)$. However, these both follow immediately: The former follows from $\Delta' = \Delta, \rho{:}\mathcal{R}$, **SR4**, **ST1**, and Context Wellformedness. The latter follows immediately from **SE3**. The rest of the argument is the same.

**case SS9**: The derivation ends with:

$$\frac{\begin{array}{c}\Delta;\Gamma;\gamma;\text{Dom}(S_1) \vdash_{\text{rhs}} e : \exists\alpha{:}\kappa \triangleright \gamma_1.\tau_2 \quad \Delta \vdash_{\text{con}} \tau{:}\mathcal{T} \\ (\Delta, \rho{:}\mathcal{R}, \alpha{:}\kappa);(\Gamma, x_\rho{:}\tau_2);(\gamma, \text{Dom}(S_1) <: \rho, \gamma_1);(\text{Dom}(S_1) \cup \rho);\tau \vdash_{\text{stmt}} s_1\end{array}}{\Delta;\Gamma;\gamma;\text{Dom}(S_1);\tau \vdash_{\text{stmt}} \rho{:}\{\texttt{open}\,[\alpha, x_\rho] = e;\ s_1\}} \quad (\rho, \alpha \notin \text{Dom}(\Delta), x_\rho \notin \text{Dom}(\Gamma))$$

Inversion of $S_2 \vdash_{\text{spop}} \rho{:}\{\texttt{open}\,[\alpha, x_\rho] = e;\ s_1\}$ (rule **SP8**) provides $S_2 \vdash_{\text{epop}} e$ and $\emptyset \vdash_{\text{spop}} s$.

If $e = v_1$, then only dynamic rule **DS8c** applies, so $v_1 = \texttt{pack}\,[\tau_1, v]\,\texttt{as}\,\tau_p$. By inversion (rule **SR13**), we know $\tau_p = \exists\alpha{:}\kappa \triangleright \gamma_1.\tau_2$, $\Delta;\Gamma;\gamma;\text{Dom}(S) \vdash_{\text{rhs}} v : \tau_2[\tau_1/\alpha]$, $\gamma \vdash_{\text{ord}} \gamma_1[\text{regions}(\tau_1)/\alpha]$, and $\Delta \vdash_{\text{con}} \tau_1{:}\kappa$. Weakening this last fact gives $\Delta, \rho{:}\mathcal{R} \vdash_{\text{con}} \tau_1{:}\kappa$. So by Substitution, we have

$$\begin{array}{l}(\Delta, \rho{:}\mathcal{R});(\Gamma, x_\rho{:}\tau_2)[\tau_1/\alpha];(\gamma, \text{Dom}(S) <: \rho, \gamma_1)[\text{regions}(\tau_1)/\alpha]; \\ (\text{Dom}(S) \cup \rho)[\text{regions}(\tau_1)/\alpha];\tau[\tau_1/\alpha] \vdash_{\text{stmt}} s_1[\tau_1/\alpha]\end{array}$$

By $\vdash_{\text{heap}} (G, S) : \Delta;\Gamma;\gamma$ and inversion on **SH**, we know $\alpha \notin \text{Dom}(S)$, $\Delta \vdash_{\text{rc}} \gamma$, and $\Delta \vdash_{\text{vctxt}} \Gamma$. Furthermore, Context Well-Formedness ensures that $\Delta \vdash_{\text{con}} \tau{:}\mathcal{T}$. Hence Useless Substitution lets us simplify to

$$\begin{array}{l}(\Delta, \rho{:}\mathcal{R});(\Gamma, x_\rho{:}\tau_2[\tau_1/\alpha]);(\gamma, \text{Dom}(S) <: \rho, \gamma_1[\text{regions}(\tau_1)/\alpha]); \\ (\text{Dom}(S) \cup \rho);\tau \vdash_{\text{stmt}} s_1[\tau_1/\alpha]\end{array}$$

Moreover, we already claimed $\gamma \vdash_{\text{ord}} \gamma_1[\text{regions}(\tau_1)/\alpha]$, so an induction on the size of $\gamma$ can show $\gamma, \text{Dom}(S) <: \rho \vdash_{\text{ord}} \gamma, \text{Dom}(S) <: \rho, \gamma_1[\text{regions}(\tau_1)/\alpha]$. So by Term Weakening, we have:

$$(\Delta, \rho{:}\mathcal{R});(\Gamma, x_\rho{:}\tau_2[\tau_1/\alpha]);(\gamma, \text{Dom}(S) <: \rho);(\text{Dom}(S) \cup \rho);\tau \vdash_{\text{stmt}} s_1[\tau_1/\alpha]$$

So by **SS7**, we can derive: $\Delta;\Gamma;\gamma;\epsilon;\tau \vdash_{\text{stmt}} \rho{:}\{\tau_2[\tau_1/\alpha]\,x_\rho = v; s_1\}$

Now, by inversion on $S_2 \vdash_{\text{epop}} e$ (rule **SE6**), we have $S_2 \vdash_{\text{epop}} v$. So by Substitution, $\emptyset \vdash_{\text{spop}} s_1[\tau_1/\alpha]$. So by **SP8**, $S_2 \vdash_{\text{spop}} \rho{:}\{\tau_2[\tau_1/\alpha]\,x_\rho = v; s_1\}$.

Letting $G' = G$, $S' = S_1S_2$ and leaving typing context unchanged, we are done because $(G, S_1S_2, \rho{:}\{\texttt{open}\,[\alpha, x_\rho] = \texttt{pack}\,[\tau_1, v]\,\texttt{as}\,\exists\alpha{:}\kappa \triangleright \gamma.\tau_2; s_1\}) \xrightarrow{\text{stmt}}$ $(G, S_1S_2, \rho{:}\{\tau_2[\tau_1/\alpha]\,x_\rho = v;\, s_1\})$.

Finally, if $e$ is not a value then only dynamic rule **DS9** applies. By induction (the right-hand-side expression part), **SS9** (with Term Weakening to show $(\Delta', \rho{:}\mathcal{R}, \alpha{:}\kappa); (\Gamma', x_\rho{:}\tau_1); (\gamma', \mathrm{Dom}(S_1') <: \rho, \gamma_1); (\mathrm{Dom}(S_1') \cup \rho); \tau \vdash_{\text{stmt}} s_1$ and Context Weakening to show $\Delta' \vdash_{\text{con}} \tau{:}\mathcal{T}$), and **SP8** (using $\emptyset \vdash_{\text{epop}} s_1$), we can derive all the results we need.

**case SR1**: The derivation ends with:

$$\frac{\gamma \vdash_{\text{ei}} \mathrm{Dom}(S_1) \Rightarrow \rho \quad \vdash_{\text{ctxt}} \Delta; \Gamma; \gamma; \mathrm{Dom}(S_1)}{\Delta; \Gamma; \gamma; \mathrm{Dom}(S_1) \vdash_{\text{rhs}} x_\rho : \Gamma(x_\rho)}$$

The only dynamic rule that applies is **DR1**. Thus, $(G, S_a[\rho \mapsto R]S_b, x_\rho) \xrightarrow{\text{rhs}} (G, S_a[\rho \mapsto R]S_b, v)$ where $R(x) = v$ and $S_1S_2 = S_a[\rho \mapsto R]S_b$. From the assumptions, we can use the Lookup Preservation lemma to show $\Delta; \Gamma; \gamma; \mathrm{Dom}(S_1) \vdash_{\text{rhs}} v : \tau$. Using $S_2 \vdash_{\text{epop}} x$ and inversion, we know that $S_2 = \emptyset$. From the Values Effectless lemma part 3, $\emptyset \vdash_{\text{epop}} v$.

**case SR2**: The derivation ends with:

$$\frac{\Delta; \Gamma; \gamma; \mathrm{Dom}(S_1) \vdash_{\text{rhs}} e : \tau@\rho_1 \quad \gamma \vdash_{\text{ei}} \rho_2 \Rightarrow \rho_1}{\Delta; \Gamma; \gamma; \mathrm{Dom}(S_1) \vdash_{\text{rhs}} e : \tau@\rho_2}$$

From the first premise and the induction hypothesis, if $(G, S_1S_2, e) \xrightarrow{\text{rhs}} (G', S_1'S_2', e')$ then (a) $\vdash_{\text{heap}} (G, S_1'S_2') : \Delta'; \Gamma'; \gamma'$, (b) $\Delta'; \Gamma'; \gamma'; \mathrm{Dom}(S_1') \vdash_{\text{rhs}} e' : \tau@\rho_1$, and (c) $S_2' \vdash_{\text{epop}} e'$ where $\Delta'$, $\Gamma'$, and $\gamma'$ extend $\Delta$, $\Gamma$, and $\gamma$ respectively and $S_1'$ extends $S_1$. It remains to show that $\gamma' \vdash_{\text{ei}} \rho_2 \Rightarrow \rho_1$ for then using **SR2** we have $\Delta'; \Gamma'; \gamma'; \mathrm{Dom}(S_1) \vdash_{\text{rhs}} e' : \tau@\rho_2$. This follows since $\gamma'$ extends $\gamma$ and $\gamma \vdash_{\text{ei}} \rho_2 \Rightarrow \rho_1$.

**cases SR3, SR4**: Trivial since $(G, S, e)$ is terminal.

**case SR5**: The derivation ends with:

$$\frac{\Delta; \Gamma; \gamma; \mathrm{Dom}(S_1) \vdash_{\text{rhs}} e_i : \tau_i}{\Delta; \Gamma; \gamma; \mathrm{Dom}(S_1) \vdash_{\text{rhs}} (e_1, e_2) : \tau_1 \times \tau_2}$$

There are two dynamic rules that might apply depending upon whether or not $e_1$ is a value.

Suppose $e_1$ is not a value. Then we have $(G, S_1S_2, (e_1, e_2)) \xrightarrow{\text{rhs}} (G', S_1'S_2', (e_1', e_2))$ via **DR11**, and thus $(G, S_1S_2, e_1) \xrightarrow{\text{rhs}} (G', S_1'S_2', e_1')$. By the induction hypothesis, we have $\Delta'$, $\Gamma'$, and $\gamma'$ which extend $\Delta$, $\Gamma$, and $\gamma$ respectively such that (a) $\vdash_{\text{heap}} (G', S_1'S_2') : \Delta'; \Gamma'; \gamma'$, (b) $\Delta'; \Gamma'; \gamma'; \mathrm{Dom}(S_1') \vdash_{\text{rhs}} e_1' : \tau_1$, and (c) $S_2' \vdash_{\text{epop}} e_1'$. By Term Weakening, we then have $\Delta'; \Gamma'; \gamma'; \mathrm{Dom}(S_1') \vdash_{\text{rhs}} e_2 : \tau_2$ (since $\mathrm{Dom}(S_1') = \mathrm{Dom}(S_1)$) and then by **SR5** we have $\Delta'; \Gamma'; \gamma'; \mathrm{Dom}(S_1') \vdash_{\text{rhs}} (e_1', e_2) : \tau_1 \times \tau_2$. Since $S_2 \vdash_{\text{epop}} (e_1, e_2)$, by inversion, we have $\emptyset \vdash_{\text{epop}} e_2$. Thus, by **SE7** $S_2' \vdash_{\text{epop}} (e_1', e_2)$.

Suppose $e_1$ is a value $v$. Then we have $(G, S_1S_2, (v, e_2)) \xrightarrow{\text{rhs}} (G', S_1'S_2', (v, e_2'))$ via **DR11**, and thus $(G, S_1S_2, e_2) \xrightarrow{\text{rhs}} (G', S_1'S_2', e_2')$. By the induction hypothesis, we have $\Delta'$, $\Gamma'$, and $\gamma'$ which extend $\Delta$, $\Gamma$, and $\gamma$ respectively such that (a) $\vdash_{\text{heap}} (G', S_1'S_2') : \Delta'; \Gamma'; \gamma'$, (b) $\Delta'; \Gamma'; \gamma'; \mathrm{Dom}(S_1') \vdash_{\text{rhs}} e_2' : \tau_2$, and (c) $S_2' \vdash_{\text{epop}} e_2'$. By Term Weakening, we then have $\Delta'; \Gamma'; \gamma'; \mathrm{Dom}(S_1') \vdash_{\text{rhs}} v : \tau_1$ (since $\mathrm{Dom}(S_1') = \mathrm{Dom}(S_1)$) and then by **SR5** we have $\Delta'; \Gamma'; \gamma'; \mathrm{Dom}(S_1) \vdash_{\text{rhs}} (v, e_2') : \tau_1 \times \tau_2$. By the Values Effectless Lemma, $\emptyset \vdash_{\text{epop}} v$ and thus using **SE8** we have $S_2' \vdash_{\text{epop}} (v, e_2')$.

**case SR6**: The derivation ends with:

$$\frac{\Delta; \Gamma; \gamma; \mathrm{Dom}(S_1) \vdash_{\mathrm{rhs}} e : \tau_1 \times \tau_2}{\Delta; \Gamma; \gamma; \mathrm{Dom}(S_1) \vdash_{\mathrm{rhs}} e.i : \tau_i}$$

There are two cases depending upon whether or not $e$ is a value. If $e$ is not a value, then $(G, S_1 S_2, e.i) \xrightarrow{\mathrm{rhs}} (G', S_1 S_2', e'.i)$ via **DR11**. In this case, the result follows from the premise, induction hypothesis and **SR6**.

Suppose $e = v$. Then $(G, S_1 S_2, v.i) \xrightarrow{\mathrm{rhs}} (G, S_1 S_2, v_i)$ via **DR3** where $v = (v_1, v_2)$ for some values $v_1$ and $v_2$. By inversion and the fact that $\Delta; \Gamma; \gamma; \mathrm{Dom}(S_1) \vdash_{\mathrm{rhs}} (v_1, v_2) : \tau_1 \times \tau_2$, we have $\Delta; \Gamma; \gamma; \mathrm{Dom}(S_1) \vdash_{\mathrm{rhs}} v_i : \tau_i$. By the Values Effectless Lemma and the fact that $S_2 \vdash_{\mathrm{epop}} (v_1, v_2)$, we know that $S_2 = \emptyset$. Again, from the Values Effectless Lemma, we then have $S_2 \vdash_{\mathrm{epop}} v_i$.

**case SR7**: The derivation ends with:

$$\frac{\Delta; \Gamma; \gamma; \mathrm{Dom}(S_1) \vdash_{\mathrm{rhs}} e : \tau@\rho \quad \gamma \vdash_{\mathrm{ei}} \mathrm{Dom}(S_1) \Rightarrow \rho}{\Delta; \Gamma; \gamma; \mathrm{Dom}(S_1) \vdash_{\mathrm{rhs}} *e : \tau}$$

There are two cases depending upon whether or not $e$ is a value. If $e$ is not a value, then $(G, S_1 S_2, *e) \xrightarrow{\mathrm{rhs}} (G', S_1' S_2', *e')$ via **DR11**. In this case, the result follows from the premise, induction hypothesis, Context Weakening (to show $\gamma' \vdash_{\mathrm{ei}} \mathrm{Dom}(S_1') \Rightarrow \rho$), and **SR7**.

Suppose $e$ is a value. Then $(G, S_1 S_2, *e) \xrightarrow{\mathrm{rhs}} (G, S_1 S_2, p)$ where $e = \&p$. From the Values Effectless lemma, it is clear that $S_2 = \emptyset$ and $S_2 \vdash_{\mathrm{epop}} p$. From the premise, we have $\Delta; \Gamma; \gamma; \mathrm{Dom}(S_1) \vdash_{\mathrm{rhs}} \&p : \tau@\rho$. We argue by an inner induction on this derivation that $\Delta; \Gamma; \gamma; \mathrm{Dom}(S_1) \vdash_{\mathrm{rhs}} p : \tau$. The derivation must end with either **SR2** or **SR9**. In the former case, the result follows from the inner induction hypothesis and the **SR2** rule. In the latter case, we have:

$$\frac{\Delta; \Gamma; \gamma; \mathrm{Dom}(S_1) \vdash_{\mathrm{lhs}} p : \tau@\rho}{\Delta; \Gamma; \gamma; \mathrm{Dom}(S_1) \vdash_{\mathrm{rhs}} \&p : \tau@\rho}$$

Now from Canonical Forms, we know $p = x_\rho.i_1.i_2.\cdots.i_n$ for some $x_\rho$, and $i_1, i_2, \ldots, i_n$. From the Path Substitution lemma, we know there exists a $\tau'$ such that $\Gamma(x_\rho) = \tau'$. From the second premise of the original conclusion, we have $\gamma \vdash_{\mathrm{ei}} \mathrm{Dom}(S_1) \Rightarrow \rho$. Therefore, from **SR1** we have $\Delta; \Gamma; \gamma; \mathrm{Dom}(S_1) \vdash_{\mathrm{rhs}} x_\rho : \tau'$. Then from the Path Substitution lemma, we know that $\Delta; \Gamma; \gamma; \mathrm{Dom}(S_1) \vdash_{\mathrm{rhs}} p : \tau@\rho$.

**case SR8**: The derivation ends with:

$$\frac{\begin{array}{c}\gamma \vdash_{\mathrm{ei}} \mathrm{Dom}(S_1) \Rightarrow \rho \\ \Delta; \Gamma; \gamma; \mathrm{Dom}(S_1) \vdash_{\mathrm{rhs}} e_1 : \mathrm{handle}(\rho) \qquad \Delta; \Gamma; \gamma; \mathrm{Dom}(S_1) \vdash_{\mathrm{rhs}} e_2 : \tau\end{array}}{\Delta; \Gamma; \gamma; \mathrm{Dom}(S_1) \vdash_{\mathrm{rhs}} \texttt{new}(e_1)\, e_2 : \tau@\rho}$$

If $e_1$ is not a value, then only dynamic rule **DR11** applies. Furthermore, inverting $S_2 \vdash_{\mathrm{epop}} \texttt{new}(e_1)\, e_2$ (rule **SE7**) provides $S_2 \vdash_{\mathrm{epop}} e_1$ and $\emptyset \vdash_{\mathrm{epop}} e_2$. So by induction, **SR8** (with Term Weakening to show $\Delta'; \Gamma'; \gamma'; \mathrm{Dom}(S_1') \vdash_{\mathrm{rhs}} e_2 : \tau$ and Constraint Weakening to show $\gamma' \vdash_{\mathrm{ei}} \mathrm{Dom}(S_1') \Rightarrow \rho$), and **SE7** (using $\emptyset \vdash_{\mathrm{epop}} e_2$), we can derive all the results we need.

Similarly, if $e_1$ is a value but $e_2$ is not a value, then only dynamic rule **DR11** applies. Furthermore, inverting $S_2 \vdash_{\mathrm{epop}} \texttt{new}(e_1)\, e_2$ (rule **SE8**) provides $\emptyset \vdash_{\mathrm{epop}} e_1$ and $\emptyset \vdash_{\mathrm{epop}} e_2$. So by induction, **SR8** (with Term Weakening to show $\Delta'; \Gamma'; \gamma'; \mathrm{Dom}(S_1') \vdash_{\mathrm{rhs}} e_1 : \mathrm{handle}(\rho)$ and Constraint Weakening to show $\gamma' \vdash_{\mathrm{ei}} \mathrm{Dom}(S_1') \Rightarrow \rho$), and **SE8** (using $\emptyset \vdash_{\mathrm{epop}} e_2$), we can derive all the results we need.

Finally, if $e_1 = v_1$ and $e_2 = v_2$, then only dynamic rule **DR5** applies. So $v_1 = \texttt{region}(\rho)$, $S_1S_2 = S_a[\rho \mapsto R]S_b$, and $(G, S_1S_2, e) \xrightarrow{\text{rhs}} (G, S_a[\rho \mapsto R[x \mapsto v]]S_b, \&x_\rho)$. Inversion of $S_2 \vdash_{\text{epop}} e$ provides (via rule **SE8**) that $S_2 \vdash_{\text{epop}} v_2$, so by Values Effectless, $S_2 = \emptyset$. Let $G' = G$, $S_1' = S_a[\rho \mapsto R[x \mapsto v]]S_b$, $S_2' = \emptyset$. $\Delta' = \Delta$, $\Gamma' = (\Gamma, x_\rho \mapsto \tau)$, and $\gamma' = \gamma$. Note that Context Well-formedness and our original derivation ensure $\Delta' \vdash_{\text{vctxt}} \Gamma'$, so $\vdash_{\text{ctxt}} \Delta'; \Gamma'; \gamma'; \text{Dom}(S_1')$. Therefore, rules **SL1** and **SR9** suffice to show $\Delta'; \Gamma'; \gamma; \text{Dom}(S_1') \vdash_{\text{rhs}} \&x_\rho : \tau$. Rules **SE1** and **SE6** suffice to show $\emptyset \vdash_{\text{epop}} \&x_\rho$. All that remains is showing $\vdash_{\text{heap}} (G, S_1') : \Delta'; \Gamma'; \gamma'$. This follows from inversion of $\vdash_{\text{heap}} (G, S_1) : \Delta; \Gamma; \gamma$ and a tedious inversion of its $\Delta; \Gamma; \gamma \vdash_{\text{stack}} S : \Gamma_S$ premise, showing that adding $[x \mapsto v]$ to $R$ makes the heap well-typed under $\Gamma'$. (For all other heap elements, the original derivation and Term Weakening suffice.) We omit the uninteresting details.

**case SR9**: The derivation ends with:

$$\frac{\Delta; \Gamma; \gamma; \text{Dom}(S_1) \vdash_{\text{lhs}} e : \tau@\rho}{\Delta; \Gamma; \gamma; \text{Dom}(S_1) \vdash_{\text{rhs}} \&e_1 : \tau@\rho}$$

Then the only dynamic rule that applies is **DR9**. And by inversion of $S_2 \vdash_{\text{epop}} e_1$ (rule **SE6**), $S_2 \vdash_{\text{epop}} e_1$. So by the induction hypothesis (the left-hand-side part) and **SR9**, we can derive $\Delta'; \Gamma'; \gamma'; \text{Dom}(S_1') \vdash_{\text{rhs}} e' : \tau@\rho$. Similarly, the induction hypothesis and **SE6** ensure that $\S_2' \vdash_{\text{epop}} e'$.

**case SR10**: The derivation ends with:

$$\frac{\Delta; \Gamma; \gamma; \text{Dom}(S_1) \vdash_{\text{lhs}} e_1 : \tau@\rho \quad \Delta; \Gamma; \gamma; \text{Dom}(S_1) \vdash_{\text{rhs}} e_2 : \tau \quad \gamma \vdash_{\text{ei}} \text{Dom}(S_1) \Rightarrow \rho}{\Delta; \Gamma; \gamma; \text{Dom}(S_1) \vdash_{\text{rhs}} e_1 = e_2 : \tau}$$

If $e_1$ is not a value, then only dynamic rule **DR9** applies. By inversion of $S_2 \vdash_{\text{epop}} e$ (rule **SE7**), we know $S \vdash_{\text{epop}} e_1$ and $\emptyset \vdash_{\text{epop}} e_2$. So by induction (the left-hand-side part), **SR10** (with Term Weakening and $\text{Dom}(S_1') = \text{Dom}(S_1)$ to type-check $e_2$ and Context Weakening to ensure $\gamma \vdash_{\text{ei}} \text{Dom}(S_1') \Rightarrow \rho$), and **SE7** (using $\emptyset \vdash_{\text{epop}} e_2$), we can derive all the results we need.

Similarly, if $e_1$ is a value and $e_2$ is not a value, then only **DR11** applies. By inversion of $S_2 \vdash_{\text{epop}} e$ (rule **SE8**, we know $\emptyset \vdash_{\text{epop}} e_1$ and $S_2 \vdash_{\text{epop}} e_2$. So by induction (the right-hand-side part), **SR10** (with Term Weakening and $\text{Dom}(S_1') = \text{Dom}(S_1)$ to type-check $e_1$ and Context Weakening to ensure $\gamma \vdash_{\text{ei}} \text{Dom}(S_1') \Rightarrow \rho$), and **SE8** (using $\emptyset \vdash_{\text{epop}} e_1$), we can derive all the results we need.

Finally, if $e_1 = p = x_\rho.i_1.\cdots.i_n$ and $e_2 = v$, then only rule **DR8** applies. Then by inversion of $S_2 \vdash_{\text{epop}} p = v$ (rule **SE8**) and Values Effectless, we know $S_2 = \emptyset$. So $S_1 = S_a[\rho \mapsto R[x \mapsto v_a]]S_b$, which by inversion of the heap typing (using stack and region rules appropriately) ensures $\Delta; \Gamma; \gamma; \emptyset \vdash_{\text{rhs}} v_a : \tau'$. So by Update Preservation, $\Delta; \Gamma; \gamma; \emptyset \vdash_{\text{rhs}} \text{update}(v_a, [i_1, \ldots, i_n], v) : \tau'$. Hence the the derivation we used to show $\Delta; \Gamma; \gamma; \emptyset \vdash_{\text{rhs}} v_a : \tau'$ can be "redone" with $\Delta; \Gamma; \gamma; \emptyset \vdash_{\text{rhs}} \text{update}(v_a, [i_1, \ldots, i_n], v) : \tau'$ in its place to type-check the new heap with the same $\Delta$, $\Gamma$, and $\gamma$. Our other obligations: $\Delta; \Gamma; \gamma; \text{Dom}(S_1') \vdash_{\text{rhs}} v : \tau$ and $\emptyset \vdash_{\text{epop}} v$ are immediate from earlier conclusions.

**case SR11**: The derivation ends with:

$$\frac{\Delta; \Gamma; \gamma; \text{Dom}(S_1); \tau \vdash_{\text{stmt}} s \quad \vdash_{\text{ret}} s}{\Delta; \Gamma; \gamma; \text{Dom}(S_1) \vdash_{\text{rhs}} \texttt{call}\{s\} : \tau}$$

If $s$ is terminal, then only dynamic rule **DR7** applies, so $s = \texttt{return}\, v$ for some $v$. By inversion of $\Delta; \Gamma; \gamma; \text{Dom}(S_1); \tau \vdash_{\text{stmt}} s$ (rule **SS2**), $\Delta; \Gamma; \gamma; \text{Dom}(S_1) \vdash_{\text{rhs}} v : \tau$. By inversion of $S_2 \vdash_{\text{epop}}$ $\texttt{call}\{\texttt{return}\, v\}$ (rules **SE9** and **SP2**) we know $S_2 \vdash_{\text{epop}} v$. So leaving the heap and context unchanged, we can derive all the results we need.

If $s$ is not terminal, then only dynamic rule **DR10** applies. Then by induction (the statement part), Return Preservation, and **SR11**, we can derive $\Delta'; \Gamma'; \gamma'; \mathrm{Dom}(S_1') \vdash_{\mathrm{rhs}} \mathtt{call}\{s'\} : \tau$. Similarly, by induction and **SE9**, we can derive $S_2' \vdash_{\mathrm{epop}} \mathtt{call}\{s'\}$.

**case SR12**: The derivation ends with:

$$\frac{\Delta; \Gamma; \gamma; \mathrm{Dom}(S_1) \vdash_{\mathrm{rhs}} e_1 \tau_2 \xrightarrow{\epsilon_1} \tau \quad \Delta; \Gamma; \gamma; \mathrm{Dom}(S_1) \vdash_{\mathrm{rhs}} e_2 : \tau_2 \quad \gamma \vdash_{\mathrm{ei}} \mathrm{Dom}(S_1) \Rightarrow \epsilon_1}{\Delta; \Gamma; \gamma; \mathrm{Dom}(S_1) \vdash_{\mathrm{rhs}} e_1(e_2) : \tau}$$

If $e_1$ or $e_2$ is not a value, the argument is very much like the analogous cases in the proof of case **SR10**; we omit the details.

So assume $e_1$ and $e_2$ are values. Then only dynamic rule **DR6** applies. So $e_1 = \rho{:}(\tau'\, x_\rho) \xrightarrow{\epsilon_1} \tau = \{s\}$ (where inversion on the typing of $e_1$ (rule **SR15** ensures the effect is $\epsilon_1$) and $e_2 = v$). Then we can derive:

$$\frac{\dfrac{\begin{array}{c}\Delta; \Gamma; \gamma; \mathrm{Dom}(S_1) \vdash_{\mathrm{rhs}} v : \tau' \quad \Delta \vdash_{\mathrm{con}} \tau{:}\mathcal{T} \\ (\Delta, \rho{:}\mathcal{R}); (\Gamma, x_\rho{:}\tau'); (\gamma, \mathrm{Dom}(S_1) <: \rho); (\mathrm{Dom}(S_1) \cup \rho); \tau \quad \vdash_{\mathrm{stmt}} s\end{array}}{\Delta; \Gamma; \gamma; \mathrm{Dom}(S_1); \tau \vdash_{\mathrm{stmt}} \rho : \{\tau'\, x_\rho = v; s\}} \quad \dfrac{\vdash_{\mathrm{ret}} s}{\vdash_{\mathrm{ret}} \rho : \{\tau'\, x_\rho = v; s\}}}{\Delta; \Gamma; \gamma; \mathrm{Dom}(S_1) \vdash_{\mathrm{rhs}} \mathtt{call}\{\rho : \{\tau'\, x_\rho = v; s\}\} : \tau}$$

We can discharge all the assumptions:

- $\Delta; \Gamma; \gamma; \mathrm{Dom}(S_1) \vdash_{\mathrm{rhs}} v : \tau'$ is from our original inversion.
- $\Delta \vdash_{\mathrm{con}} \tau{:}\mathcal{T}$ is from our original inversion and Context Well-formedness.
- $(\Delta, \rho{:}\mathcal{R}); (\Gamma, x_\rho{:}\tau'); (\gamma, \mathrm{Dom}(S_1) <: \rho); (\mathrm{Dom}(S_1) \cup \rho); \tau \vdash_{\mathrm{stmt}} s$ is from inversion of $\Delta; \Gamma; \gamma; \mathrm{Dom}(S_1) \vdash_{\mathrm{rhs}} e_1 : \tau_2 \xrightarrow{\epsilon_1} \tau$ (rule **SR15**), the assumption $\gamma \vdash_{\mathrm{ei}} \mathrm{Dom}(S_1) \Rightarrow \epsilon_1$ from our original inversion, and Term Weakening.
- $\vdash_{\mathrm{ret}} s$ is from inversion of $\Delta; \Gamma; \gamma; \mathrm{Dom}(S_1) \vdash_{\mathrm{rhs}} e_1 : \tau_2 \xrightarrow{\epsilon_1} \tau$ (rule **SR15**).

Inversion of $S_2 \vdash_{\mathrm{epop}} e_1(e_2)$ provides (using rules **SE8** and **SE4** that $S_2 \vdash_{\mathrm{spop}} e_2$ and $\emptyset \vdash_{\mathrm{spop}} s$. By Values Effectless, $S_2 = \emptyset$. So leaving the heap and context unchanged, we can prove all the desired results.

**case SR13**: The derivation ends with:

$$\frac{\begin{array}{c}\Delta \vdash_{\mathrm{con}} \tau_1 : \kappa \qquad \gamma \vdash_{\mathrm{ord}} \gamma_1[\mathrm{regions}(\tau_1)/\alpha] \\ \Delta; \Gamma; \gamma; \mathrm{Dom}(S_1) \vdash_{\mathrm{rhs}} e_1 : \tau_2[\tau_1/\alpha]\end{array}}{\Delta; \Gamma; \gamma; \mathrm{Dom}(S_1) \vdash_{\mathrm{rhs}} \mathtt{pack}\,[\tau_1, e_1]\,\mathtt{as}\, \exists\alpha{:}\kappa \triangleright \gamma_1.\tau_2 : \exists\alpha{:}\kappa \triangleright \gamma_1.\tau_2} \qquad (\alpha \notin \mathrm{Dom}(\Delta), \kappa \neq \mathcal{T})$$

The only dynamic rule that applies is **DR11**. Inversion of $S_2 \vdash_{\mathrm{epop}} e$ provides (via rule **SE6**) that $S_2 \vdash_{\mathrm{epop}} e_1$. So by induction (the right-hand-side part), Context Weakening (to show $\Delta' \vdash_{\mathrm{con}} \tau_1{:}\kappa$ and $\gamma \vdash_{\mathrm{ord}} \gamma_1[\mathrm{regions}(\tau_1)/\alpha]$), and **SR13**, we can show $\Delta'; \Gamma'; \gamma'; \mathrm{Dom}(S_1') \vdash_{\mathrm{rhs}} e' : \exists\alpha{:}\kappa \triangleright \gamma_1.\tau_2$. And induction and **SE6** shows that $S_2' \vdash_{\mathrm{epop}} e'$.

**case SR14**: The derivation ends with:

$$\frac{\begin{array}{c}\Delta; \Gamma; \gamma; \mathrm{Dom}(S_1) \vdash_{\mathrm{rhs}} \vdash_{\mathrm{rhs}} e_1 : \forall\alpha{:}\kappa \triangleright \gamma_1.\tau_2 \\ \Delta \vdash_{\mathrm{con}} \tau_1 : \kappa \qquad \gamma \vdash_{\mathrm{ord}} \gamma_1[\mathrm{regions}(\tau_1)/\alpha]\end{array}}{\Delta; \Gamma; \gamma; \mathrm{Dom}(S_1) \vdash_{\mathrm{rhs}} e_1\langle\tau_1\rangle : \tau_2[\tau_1/\alpha]}$$

Inversion on $S_2 \vdash_{\mathrm{epop}} e$ (rule **SE6**) provides $S_2 \vdash_{\mathrm{epop}} e_1$.

If $e_1$ is not a value then only dynamic rule **DR11** applies. By the induction hypothesis, **SR14**, Context Weakening (to show $\Delta' \vdash_{\text{con}} \tau_1{:}\kappa$ and $\gamma' \vdash_{\text{ord}} \gamma_1[\text{regions}(\tau_1)/\alpha]$, and **SE6**, we can derive all the results that we need.

If $e_1$ is a value, then only **DR2** applies, so $v = \Lambda\alpha{:}\kappa \triangleright \gamma_1.f$ (where inversion on type-checking $e_1$ (rule **SR16**) requires $\kappa$ and $\gamma_1$). Furthermore, the inversion provides:

- $(\Delta, \alpha{:}\kappa); \Gamma; (\gamma, \gamma_1); \text{Dom}(S_1) \vdash_{\text{rhs}} f : \tau_2$
- $\Delta, \alpha{:}\kappa \vdash_{\text{rc}} \gamma_1$
- $\vdash_{\text{ctxt}} \Delta; \Gamma; \gamma; \text{Dom}(S_1)$

So by Substitution (noting that $\Delta \vdash_{\text{con}} \tau_1{:}\kappa$), we know

$$\Delta; \Gamma[\tau_1/\alpha]; (\gamma, \gamma_1)[\text{regions}(\tau_1)/\alpha]; \text{Dom}(S_1)[\text{regions}(\tau_1)/\alpha] \vdash_{\text{rhs}} f[\tau_1/\alpha] : \tau_2[\tau_1/\alpha]$$

By inversion of $\vdash_{\text{ctxt}} \Delta; \Gamma; \gamma; \text{Dom}(S_1)$ and Useless Substitution, we can simplify to

$$\Delta; \Gamma; (\gamma, \gamma_1[\text{regions}(\tau_1)/\alpha]); \text{Dom}(S_1) \vdash_{\text{rhs}} f[\tau_1/\alpha] : \tau_2[\tau_1/\alpha]$$

And by $\gamma \vdash_{\text{ord}} \gamma_1[\text{regions}(\tau_1)/\alpha]$ we can show $\gamma \vdash_{\text{ord}} \gamma, \gamma_1[\text{regions}(\tau_1)/\alpha]$, so by Term Weakening,

$$\Delta; \Gamma; \gamma; \text{Dom}(S_1) \vdash_{\text{rhs}} f[\tau_1/\alpha] : \tau_2[\tau_1/\alpha]$$

Now, by inversion on $S_2 \vdash_{\text{epop}} e_1$ (rule **SE5**), we know $S_2 \vdash_{\text{epop}} f$. So by Substitution, $S_2 \vdash_{\text{epop}} f[\tau_1/\alpha]$. Leaving the heap and typing context unchanged, we are done because $e' = f[\tau_1/\alpha]$.

**case SR15, SR16, SL1**: Trivial because $(G, S, e)$ is terminal

**case SL2**: The derivation ends with:

$$\frac{\Delta; \Gamma; \gamma; \text{Dom}(S_1) \vdash_{\text{rhs}} e : \tau@\rho}{\Delta; \Gamma; \gamma; \text{Dom}(S_1) \vdash_{\text{lhs}} e_1 : \tau@\rho}$$

If $e_1 = v$, then only dynamic rule **DL1** applies. That means $v = \&p$, so by inversion (rule **SR9**), $\Delta; \Gamma; \gamma; \text{Dom}(S_1) \vdash_{\text{lhs}} p : \tau@\rho$. By inversion of $S_2 \vdash_{\text{epop}} e$ (two uses of **SE6**), we know $S_2 \vdash_{\text{epop}} p$. Leaving the heap and context unchanged, we are done.

If $e_1$ is not a value, then only dynamic rule **DL3** applies. By inversion of $S_2 \vdash_{\text{epop}} e$ (rule **SE6**), $S_2 \vdash_{\text{epop}} e_1$. So by the induction hypothesis (the right-hand-side part) and **SL9**, we can derive $\Delta'; \Gamma'; \gamma'; \text{Dom}(S_1') \vdash_{\text{rhs}} e' : \tau@\rho$. Similarly, the induction hypothesis and **SE6** ensure that $S_2' \vdash_{\text{epop}} e'$.

**case SL3**: The derivation ends with

$$\frac{\Delta; \Gamma; \gamma; \text{Dom}(S_1) \vdash_{\text{rhs}} e_1 : (\tau_1 \times \tau_2)@\rho}{\Delta; \Gamma; \gamma; \text{Dom}(S_1) \vdash_{\text{lhs}} e_1.i : \tau_i@\rho}$$

Only dynamic rule **DL2** can apply. By inversion of $S_2 \vdash_{\text{epop}} e$ (rule **SE6**), $S_2 \vdash_{\text{epop}} e_1$. So by the induction hypothesis and **SL3**, we can derive $\Delta'; \Gamma'; \gamma'; \text{Dom}(S_1') \vdash_{\text{rhs}} e' : \tau@\rho$. Similarly, the induction hypothesis and **SE6** ensure that $S_2' \vdash_{\text{epop}} e'$.

**Lemma 10.16 (Constraint Progress)**
*Suppose:*

1. $\vdash_{\text{heap}} (G, S) : \Delta; \Gamma; \gamma$

*2.* $\gamma \vdash_{\text{vi}} \rho_1 \Rightarrow \rho_2$

*3.* $\rho_1 \in \text{Dom}(S)$

*Then* $\rho_2 \in \text{Dom}(S)$*.*

**Proof:**

By induction on the derivation of $\gamma \vdash_{\text{vi}} \rho_1 \Rightarrow \rho_2$.

**case SC11**: The derivation ends with $\gamma \vdash_{\text{vi}} \rho \Rightarrow \rho$ (i.e., $\rho_1 = \rho_2 = \rho$). From (3), it follows that $\rho \in \text{Dom}(S)$.

**case SC13**: The derivation ends with:

$$\frac{\gamma \vdash_{\text{vi}} \rho_1 \Rightarrow \rho \quad \gamma \vdash_{\text{vi}} \rho \Rightarrow \rho_2}{\gamma \vdash_{\text{vi}} \rho_1 \Rightarrow \rho_2}$$

By induction on the first premise, we have $\rho \in \text{Dom}(S)$. Then by induction on the second premise, we have $\rho_2 \in \text{Dom}(S)$.

**case SC12**: The derivation ends with:

$$\frac{\rho_2 \in \epsilon}{\gamma_1, \epsilon <: \rho_1, \gamma_2 \vdash_{\text{vi}} \rho_1 \Rightarrow \rho_2}$$

Therefore, $\gamma = \gamma_1, \epsilon <: \rho_1, \gamma_2$ for some $\gamma_1$, $\gamma_2$, and $\epsilon$. By (1) and inversion of the **SH** rule, it then follows that either (a) $\epsilon <: \rho_1 \in \gamma_S$ where the constraints in $\gamma_S$ are all of the form $\rho_a <: \rho_b$ such that $\rho_a, \rho_b \in \text{Dom}(S)$, or else (b) $\epsilon <: \rho_1 \in \gamma_G$ where the constraints in $\gamma_G$ are all of the form $\epsilon_a <: \rho_b$ and $\rho_b \in \text{Dom}(G)$. But since $\text{Dom}(G) \cap \text{Dom}(S) = \emptyset$ and (3), $\epsilon <: \rho_1 \notin \gamma_G$. Therefore, $\epsilon <: \rho_1 \in \gamma_S$ and we can conclude that $\epsilon = \rho_2$ and $\rho_2 \in \text{Dom}(S)$.

**Lemma 10.17 (Canonical Forms)**
*Suppose* $\vdash_{\text{heap}} (G, S) : \Delta; \Gamma; \gamma$*,* $\text{Dom}(S) = \epsilon_1 \uplus \epsilon_2$*, and* $\Delta; \Gamma; \gamma; \epsilon_1 \vdash_{\text{rhs}} v : \tau$*. Then:*

1. *if* $\tau = int$ *then* $v = i$ *for some integer* $i$*.*
2. *if* $\tau = \tau_1 \xrightarrow{\epsilon} \tau_2$ *then* $v = \rho{:}(\tau_1\, x_\rho) \xrightarrow{\tau_2} s = \{\}$ *for some* $\rho$*,* $x_\rho$*, and* $s$*.*
3. *if* $\tau = \tau_1 \times \tau_2$ *then* $v = (v_1, v_2)$ *for some values* $v_1$ *and* $v_2$*.*
4. *if* $\tau = \tau@\rho$ *then* $v = \&p$ *for some path* $p$*.*
5. *if* $\tau = \text{handle}(\rho)$ *then* $v = \texttt{region}(\rho)$*.*
6. *if* $\tau = \forall\alpha{:}\kappa \triangleright \gamma.\tau$*, then* $v = \Lambda\alpha{:}\kappa \triangleright \gamma.f$ *for some function* $f$*.*
7. *if* $\tau = \exists\alpha{:}\kappa \triangleright \gamma.\tau$*, then* $v = \texttt{pack}\,[\tau', v]\,\texttt{as}\,\exists\alpha{:}\kappa \triangleright \gamma.\tau$ *for some* $\tau'$ *and* $v$*.*

**Proof:**

By the definition of values and inspection of the **SR** rules.

**Lemma 10.18 (Canonical Paths)**
*Suppose:*

*1.* $\vdash_{\text{heap}} (G, S) : \Delta; \Gamma; \gamma$

*2.* $\mathrm{Dom}(S) = \epsilon_1 \uplus \epsilon_2$,

*3.* $\gamma \vdash_{\mathrm{ei}} \epsilon_1 \Rightarrow \rho$, *and*

*4.* $\Delta; \Gamma; \gamma; \epsilon_1 \vdash_{\mathrm{lhs}} p : \tau@\rho$.

*Then:*

*1.* $p$ *is of the form* $x_\rho.i_1.i_2.\cdots.i_n$ *for some* $i_1, i_2, \ldots, i_n$ *where* $n \geq 0$,

*2.* $S = S_1[\rho \mapsto R[x \mapsto v]]S_2$ *for some* $S_1$, $S_2$, $R$, *and* $v$,

*3.* $\Delta; \Gamma; \gamma; \emptyset \vdash_{\mathrm{rhs}} v : \tau'$ *for some* $\tau'$, *and*

*4.* $\Delta; \Gamma; \gamma; \emptyset \vdash_{\mathrm{rhs}} v.i_1.i_2.\cdots.i_n : \tau$.

**Proof:**

By inspection of the **SL** rules, $p$ must be of the form $x_\rho.i_1.i_2.\cdots.i_n$ for some $n \geq 0$. From (3) and inversion of rule **SC14**, we know that there exists a $\rho'$ in $\epsilon_1$ such that $\gamma \vdash_{\mathrm{vi}} \rho' \Rightarrow \rho$. From this fact and assumptions (1) and (2), we can use the Constraint Progress lemma to conclude that $\rho \in \mathrm{Dom}(S)$. Therefore, $S = S_1[\rho \mapsto R]S_2$ for some $S_1$, $S_2$, and $R$.

Using the Path Substitution lemma, we know that there exists a $\tau'$ such that $\Gamma(x_\rho) = \tau'$. From assumption (1) and inversion of the **SH** rule, we have $\Delta; \Gamma; \gamma \vdash_{\mathrm{stack}} S_1[\rho \mapsto R]S_2 : \Gamma_S$ and that $\Gamma = \Gamma_G \uplus \Gamma_S$. By induction on the length of $S_2$ using inversion of the **SStk2** rule, we can show that $\Gamma_S = \Gamma_{S_1} \uplus \Gamma_R \uplus \Gamma_{S_2}$ and that $\Delta; \Gamma; \gamma \vdash_{\mathrm{rgn}} \rho \mapsto R : \Gamma_R$. By induction on the length of $R$ using inversion of the **SRgn2** rule, we can show that $R = R_1[x \mapsto v]R_2$, $\Gamma_R(x_\rho) = \tau'$, and $\Delta; \Gamma; \gamma; \emptyset \vdash_{\mathrm{rhs}} v : \tau'$. Using Weakening, we can conclude that $\Delta; \Gamma; \gamma; \epsilon_1 \vdash_{\mathrm{rhs}} v : \tau'$. From this and the fact that $\Delta; \Gamma; \gamma; \epsilon_1 \vdash_{\mathrm{lhs}} p : \tau@\rho$, we can use the Path Substitution lemma to conclude $\Delta; \Gamma; \gamma; \emptyset \vdash_{\mathrm{rhs}} v.i_1.i_2.\cdots.i_n : \tau$.

**Lemma 10.19 (Update Progress)**
*Suppose*

*1.* $\Delta; \Gamma; \gamma; \emptyset \vdash_{\mathrm{rhs}} v_a : \tau$

*2.* $\Delta; \Gamma; \gamma; \epsilon \vdash_{\mathrm{rhs}} v_a.i_n.i_{n-1}.\cdots.i_2.i_1 : \tau'$

*Then* $\mathrm{update}(v_a, [i_n, i_{n-1}, \ldots, i_2, i_1], v_b) = v$ *for some value* $v$.

**Proof:**

The proof is by induction on $n$. When $n = 0$ we have $\mathrm{update}(v_a, [], v_b) = v_b$. Suppose the lemma holds for all values up to $n-1$. By the Projection lemma, we have $v_a = (v_1, v_2)$ and that $\Delta; \Gamma; \gamma; \epsilon \vdash_{\mathrm{rhs}} v_{i_n}.i_{n-1}.\cdots.i_2.i_1 : \tau$. So by the induction hypothesis, we have $\mathrm{update}(v_{i_n}, [i_{n-1}, \ldots, i_2, i_1], v_b) = v$ for some value $v$. Thus, if $i_n = 1$ we have $\mathrm{update}((v_1, v_2), [1, i_{n-1}, \ldots, i_2, i_1], v_b) = (v, v_2)$ and if $i_n = 2$ we have $\mathrm{update}((v_1, v_2), [2, i_{n-1}, \ldots, i_2, i_1], v_b) = (v_1, v)$.

**Lemma 10.20 (Progress)**
*Suppose* $\vdash_{\mathrm{heap}} (G, S_1S_2) : \Delta; \Gamma; \gamma$. *Then:*

*1. If* $\Delta; \Gamma; \gamma; \mathrm{Dom}(S_1); \tau \vdash_{\mathrm{stmt}} s$ *and* $S_2 \vdash_{\mathrm{spop}} s$, *then either* $s$ *is terminal* ($s = v$ *or* $s = \texttt{return}\, v$ *for some* $v$) *or there exist* $G', S', s'$ *such that* $(G, S_1S_2, s) \overset{\mathrm{stmt}}{\longrightarrow} (G', S', s')$.

2. *If* $\Delta; \Gamma; \gamma; \text{Dom}(S_1) \vdash_{\text{rhs}} e : \tau$ *and* $S_2 \vdash_{\text{epop}} e$*, then either* $e = v$ *for some* $v$ *or there exist* $G', S', e'$ *such that* $(G, S_1S_2, e) \xrightarrow{\text{rhs}} (G', S', e')$*.*
3. *If* $\Delta; \Gamma; \gamma; \text{Dom}(S_1) \vdash_{\text{lhs}} e : \tau$ *and* $S_2 \vdash_{\text{epop}} e$*, then either* $e = p$ *for some* $p$ *or there exist* $G', S', e'$ *such that* $(G, S_1S_2, e) \xrightarrow{\text{lhs}} (G', S', e')$*.*

**Proof:**

By simultaneous induction on the derivations of $\Delta; \Gamma; \gamma; \text{Dom}(S_1); \tau \vdash_{\text{stmt}} s$, $\Delta; \Gamma; \gamma; \text{Dom}(S_1) \vdash_{\text{rhs}} e : \tau$, and $\Delta; \Gamma; \gamma; \text{Dom}(S_1) \vdash_{\text{lhs}} e : \tau@\rho$. Most of the cases are straightforward applications of the induction hypotheses or follow from the Canonical Forms lemma. The interesting cases, shown below, involve situations where we must show that a needed region has not yet been deallocated, or where we use the "$\vdash_{\text{ret}}$" judgment to ensure that functions eventually return.

**case SS6**: The derivation ends with:

$$\frac{\Delta; \Gamma; \text{Dom}(S_1) \uplus \rho; \tau \vdash_{\text{stmt}} s}{\Delta; \Gamma; \gamma; \text{Dom}(S_1); \tau \vdash_{\text{stmt}} s\ \mathtt{pop}[\rho]}$$

Using the assumption $S_2 \vdash_{\text{spop}} s\ \mathtt{pop}[\rho]$ and inversion, we know that $S_2 = [\rho \mapsto R]S_2'$ for some $S_2'$ and $S_2' \vdash_{\text{spop}} s$. Therefore, $S_1S_2 = (S_1[\rho \mapsto R])S_2'$ and the induction hypothesis applies to $s$. If $(G, S_1S_2, s) \xrightarrow{\text{stmt}} (G, S_1S_2', s')$, then using **DS10**, we can derive $(G, S_1S_2', s\,\mathtt{pop}[\rho]) \xrightarrow{\text{stmt}} (G', S_1S_2', s'\,\mathtt{pop}[\rho])$. Suppose, therefore, $s$ is terminal. Since $S_1S_2$ is of the form $S_1[\rho \mapsto R]S_2'$, if we can show $S_2'$ is empty we can use **DS6** or **DS7** to show that $(G, S_1[\rho \mapsto R], s\ \mathtt{pop}[\rho]) \xrightarrow{\text{stmt}} (G[\rho \mapsto R], S_1, s)$. But since $S_2' \vdash_{\text{spop}} s$ and $s$ is terminal, Values Effectless ensures that $S_2' = \emptyset$.

**case SR1**: The derivation ends with:

$$\frac{\gamma \vdash_{\text{ei}} \text{Dom}(S_1) \Rightarrow \rho \quad \vdash_{\text{ctxt}} \Delta; \Gamma; \gamma; \epsilon_1}{\Delta; \Gamma; \gamma; \text{Dom}(S_1) \vdash_{\text{rhs}} x_\rho : \Gamma(x_\rho)}$$

It suffices to show that $S_1S_2 = S_a[\rho \mapsto R[x \mapsto v]]S_b$ for some $S_a$, $S_b$, $R$, and $v$, for then rule **DR1** applies. This follows from the assumptions and the Canonical Paths lemma.

**case SR7**: The derivation ends with:

$$\frac{\Delta; \Gamma; \gamma \text{Dom}(S_1) \vdash_{\text{rhs}} e : \tau@\rho \quad \gamma \vdash_{\text{ei}} \text{Dom}(S_1) \Rightarrow \rho}{\Delta; \Gamma; \gamma; \text{Dom}(S_1) \vdash_{\text{rhs}} *e : \tau}$$

By the induction hypothesis, $e = v$ for some value $v$ or else $(G, S_1S_2, e) \xrightarrow{\text{rhs}} (G', S', e')$. In the latter case, we can use rule **DR11** to show $(G, S_1S_2, *e) \xrightarrow{\text{rhs}} (G', S', *e')$. So suppose $e = v$. Then by the Canonical Forms lemma, $v = \&p$ for some path $p$. Then by rule **DR4**, we have $(G, S_1S_2, *(\&p)) \xrightarrow{\text{rhs}} (G, S_1S_2, p)$.

**case SR10**: The derivation ends with:

$$\frac{\Delta; \Gamma; \gamma; \text{Dom}(S_1) \vdash_{\text{lhs}} e_1 : \tau@\rho \quad \Delta; \Gamma; \gamma; \text{Dom}(S_1) \vdash_{\text{rhs}} e_2 : \tau \quad \gamma \vdash_{\text{ei}} \text{Dom}(S_1) \Rightarrow \rho}{\Delta; \Gamma; \gamma; \text{Dom}(S_1) \vdash_{\text{rhs}} e_1 = e_2 : \tau}$$

By the induction hypothesis either $e_1$ is a path $p$ or else $(G, S_1S_2, e_1) \xrightarrow{\text{lhs}} (G', S', e_1')$. In the latter case, we can use rule **DR9** to show $(G, S_1S_2, e_1 = e_2) \xrightarrow{\text{rhs}} (G', S', e_1' = e_2)$. So suppose $e_1 = p$.

Then using the induction hypothesis, either $e_2$ is a value $v_2$ or else $(G, S_1S_2, e_2) \xrightarrow{\text{rhs}} (G', S', e_2')$. Again, in the latter case we can use rule **DR11** to show $(G, S_1S_2, p = e_2) \xrightarrow{\text{rhs}} (G', S', p = e_2')$. So suppose $e_2 = v_2$. By the Canonical Paths lemma, we know that $p = x_\rho.i_1.i_2.\cdots.i_n$, $S_1S_2 = S_a[\rho \mapsto R[x \mapsto v_1]]S_b$, $\Delta; \Gamma; \gamma; \emptyset \vdash_{\text{rhs}} v_1 : \tau'$ for some $\tau'$, and $\Delta; \Gamma; \gamma; \emptyset \vdash_{\text{rhs}} v_1.i_1.i_2.\cdots.i_n : \tau$. Then from the Update Progress lemma, we have that update$(v_1, [i_1, i_2, \ldots, i_n], v_2)$ is defined. Consequently we can use rule **DR8** to show:

$$(G, S_1S_2, e_1 = e_2) \xrightarrow{\text{rhs}} \quad (G, S_a[\rho \mapsto R[x \mapsto \text{update}(v_1, [i_1, i_2, \ldots, i_n], v_2)]]S_b, v_2)$$

**case SR11**: The derivation ends with:

$$\frac{\Delta; \Gamma; \gamma; \text{Dom}(S_1); \tau \vdash s \quad \vdash_{\text{ret}} s}{\Delta; \Gamma; \gamma; \text{Dom}(S_1) \vdash \texttt{call}\{s\} : \tau}$$

By the induction hypothesis, $s$ is either terminal or else $(G, S_1S_2, s) \xrightarrow{\text{stmt}} (G', S', s')$. In the latter case, we can use rule **DR10** to show $(G, S_1S_2, \texttt{call}\{s\}) \xrightarrow{\text{rhs}} (G', S', \texttt{call}\{s'\})$. So suppose $s$ is terminal. Then using inversion and $\vdash_{\text{ret}} s$, we can conclude that $s$ must be of the form $\texttt{return}\, v$. Therefore, we can use **DR7** to conclude $(G, S_1S_2, \texttt{call}\{\texttt{return}\, v\}) \xrightarrow{\text{rhs}} (G, S_1S_2, v)$.

As usual, Preservation and Progress suffice to prove syntactic type-soundness. In order to allow (mutually) recursive functions, we state the theorem in terms of a program $s$ executing under an initial region $\rho_H$:

**Theorem 10.21 (Soundness)** *If:*

1. $\vdash_{\text{heap}} (\emptyset, [\rho_H \mapsto R]) : \Delta; \Gamma; \gamma$,
2. $\vdash_{\text{ret}} s$,
3. $\Delta; \Gamma; \gamma; \{\rho_H\}; \text{int} \vdash_{\text{stmt}} s$, *and*
4. *$s$ contains no* `pop` *statements*

*then either $(G, [\rho_H \mapsto R], s)$ runs forever or there exist $G'$, $R'$ and $i$ such that $(G, [\rho_H \mapsto R], s) \rightarrow^* (G', [\rho_H \mapsto R'], \texttt{return}(i))$*

**Proof:**

First we notice that part (4) implies (or is just the informal version of) $\emptyset \vdash_{\text{spop}} s$. So letting $S_1 = [\rho_H \mapsto R]$ and $S_2 = \emptyset$, Preservation and Progress apply. By induction on the number of evaluation steps, using Preservation and Progress at each step (both will apply because of Preservation), either $(G, [\rho_H \mapsto R], s)$ runs forever or there exists a $G'$, $S'$ and $i$ such that $(G, [\rho_H \mapsto R], s) \rightarrow^* (G', S', \texttt{return}(i))$. Furthermore, $S' = S_1'S_2'$ where $S_1'$ extends (again by induction, since the definition implies transitivity) $[\rho_H \mapsto R]$. Finally, $S_2' \vdash_{\text{spop}} \texttt{return}(i)$, so $S_2' = \emptyset$. The definition of extends for stacks yields the desired result.