Typed Memory Management in a Calculus of Capabilities*

David Walker
Cornell University

Karl Crary
Carnegie Mellon University

Greg Morrisett
Cornell University

February 2, 2000

Abstract

Region-based memory management is an alternative to standard tracing garbage collection that makes potentially dangerous operations such as memory deallocation explicit but verifiably safe. In this article, we present a new compiler intermediate language, called the Capability Calculus, that supports region-based memory management and enjoys a provably safe type system. Unlike previous region-based type systems, region lifetimes need not be lexically scoped and yet the language may be checked for safety without complex analyses. Therefore, our type system may be deployed in settings such as extensible operating systems where both the performance and safety of untrusted code is important.

The central novelty of the language is the use of static capabilities to specify the permissibility of various operations, such as memory access and deallocation. In order to ensure capabilities are relinquished properly, the type system tracks aliasing information using a form of bounded quantification. Moreover, unlike previous work on region-based type systems, the proof of soundness of our type system is relatively simple, employing only standard syntactic techniques.

In order to show our language may be used in practice, we show how to translate a variant of Tofte and Talpin's high-level type-and-effects system for region-based memory management into our language. When combined with known region inference algorithms, this translation provides a way to compile source-level languages to the Capability Calculus.

1 Motivation and Background

A current trend in systems software is to allow untrusted extensions to be installed in protected services, relying upon language technology to protect the integrity of the service instead of hardware-based protection mechanisms [24, 51, 4, 35, 33, 22, 19]. For example, the SPIN project [4] relies upon the Modula-3 type system to protect an operating system kernel from erroneous extensions. Similarly, web browsers rely upon the Java Virtual Machine byte-code verifier [24] to protect users from malicious applets. In both situations, the goal is to eliminate expensive communications or boundary crossings by allowing extensions to directly access the resources they require.

*This research was performed while the first author was at Cornell University. This material is based on work supported in part by the AFOSR grant F49620-97-1-0013, ARPA/RADC grant F30602-96-1-0317 and NSF grant No. EIA 97-03470. 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.

Recently, Necula and Lee [36, 35] have proposed Proof-Carrying Code (PCC) and Morrisett *et al.* [34, 33] have suggested Typed Assembly Language (TAL) as language technologies that provide the security advantages of high-level languages, but without the overheads of interpretation or just-in-time compilation. In both systems, low-level machine code can be heavily optimized, by hand or by compiler, and yet be automatically verified through proof- or type-checking.

However, in all of these systems (SPIN, JVM, TAL, and Touchstone [37], a compiler that generates PCC), there is one aspect over which programmers and optimizing compilers have little or no control: memory management. In particular, their soundness depends on memory being reclaimed by a trusted garbage collector. Hence, applets or kernel extensions may not perform their own optimized memory management. Furthermore, as garbage collectors tend to be large, complicated pieces of unverified software, the degree of trust in language-based protection mechanisms is diminished.

The goal of this work is to provide a high degree of control over memory management for programmers and compilers, but as in the PCC and TAL frameworks, make verification of the safety of programs a straightforward task.

1.1 Regions

Tofte and Talpin [47, 48] suggest a type and effects system for verifying the soundness of *region-based* memory management. In later work, Tofte and others show how to infer region types and lifetimes and how to implement their theory [46, 5, 6]. There are several advantages to region-based memory management; from our point of view, the two most important are:

1. Region-based memory management can be implemented using relatively simple constant-time routines.

2. All memory operations explicit in the program text, but safety is guaranteed.

The first advantage has several implications. If regions are used in a secure system then the simplicity of the implementation leads to a smaller trusted computed base. Moreover, it may be possible to formally verify that the region operations are implemented correctly. In contrast, a standard tracing garbage collector is a large and extremely complicated piece of code; a formal analysis of a garbage collector would be a much more onerous task than an analysis of a region-based system. Second, because region operations are constant-time and do not trace the structure of the heap, programs do not suffer from the pauses that are associated with conventional garbage collectors. Consequently, region-based memory management systems may be a practical alternative to real-time garbage collectors [3, 53].

The second advantage gives programmers greater control over memory use. By using a region-profiler [5], programmers can quickly identify the memory regions that are causing performance problems in their applications. Next, because allocation and deallocation operations are explicit in the program text, programmers can use the profiling data to accurately relate the run-time behaviour of programs to their static representation. In other words, given information about the ways regions are used at run time, it is often straight-forward to examine program code, identify memory-intensive routines, and reason about the lifetimes of the data structures allocated there. Once the trouble spots have been identified, programmers can concentrate their optimization efforts on a small portion of the code. Most importantly, throughout the programming process, a type checker guarantees that all memory

operations are safe so programmers do not have to worry about programs crashing due to memory faults.

In order to ensure that regions are used safely, the Tofte-Talpin language includes a lexically-scoped expression (`letregion r in e end`) that delimits the lifetime of a region `r`. A region is allocated when control enters the scope of the `letregion` construct and is deallocated when control leaves the scope. Programs may allocate values into live regions using the notation `v at r`. These values may be used until the region is deallocated. For example,

Region lifetime

```
⋮
letregion r in            % Allocate region r
    let x = v at r in     % Allocate value v in region r
    f (x)
end                       % Deallocate region r (and v)
⋮
```

Tofte and Talpin ensure that deallocated values are not accessed unsafely using a type and effects system. Informally, whenever an expression uses a value in region `r`, the type system expresses this fact using the effect *access*(`r`). However, effects occuring within the scope of the `letregion` construct are masked. More specifically, if the expression `e` has effects *access*(`r`) $\cup$ ψ (for some set of effects ψ) then the overall effect of the expression `letregion r in e end` is simply ψ. Hence, if there is no overall effect for an entire program then every region access must have occured within the scope of the corresponding `letregion` construct. In other words, values in region `r` are used only during the lifetime of `r` and not before or after. If this condition holds, we can conclude the program is safe.

The Tofte-Talpin language makes efficient use of memory provided that the lifetimes of values coincide with the lexical structure of the program. However, if lifetimes deviate from program structure then this style of region-based memory management may force programs to use considerably more memory than necessary. Consider the following program fragments.

```
% Scope 1:  The Call Site

let x = v at r in
⋮
let y = f (x) in
⋮
y is dead
```

```
% Scope 2:  The Function

fun f (x) =
    ⋮
    x is dead
    ⋮
    let y = v' at r' in
    ⋮
    return y
```

The value `v` is an argument to the function `f` and must be allocated in the scope of the function call. However, when `f` is executed, `v` dies quickly. The value `v'` exhibits the inverse behaviour. It is allocated inside `f` but is returned as the function result. Both `v` and `v'` have lifetimes that span two lexical scopes,

but neither is live for very long in either scope. Consequently, vanilla region inference does not perform well in this setting. The best it can do is wrap the function call in a pair of `letregion` commands.

```
% Scope 1:  The Call Site

letregion r in
    let x = v at r in
    ⋮
    letregion r′ in
        let y = f (x) in
        ⋮
        y is dead
    end (r′)
end (r)
⋮
```

```
% Scope 2:  The Function

fun f (x) =
    ⋮
    x is dead
    ⋮
    let y = v′ at r′ in
    ⋮
    return y
```

Here, the regions `r` and `r′` are live much longer than they need to be due to the inflexibility of the `letregion` construct. Both regions must be allocated and outside the function call. Notice also that even though `v` is dead when the function call returns, the outer region `r` cannot be deallocated until after the inner region `r′` has been deallocated. Lexical scoping enforces a stack-like, last-allocated/first-deallocated memory management discipline.

In this example, a much better solution to this memory management problem is to separate region allocation (`newregion`) from deallocation (`freeregion`). The following program takes this approach. In principle, since the lifetimes of regions `r` and `r′` do not overlap, the memory for these regions could be reused.

```
% Scope 1:  The Call Site

let newregion r in
let x = v at r in
⋮
let y = f (x) in
⋮
y is dead
let freeregion r′ in
⋮
```

```
% Scope 2:  The Function

fun f (x) =
    ⋮
    x is dead
    let freeregion r in
    ⋮
    let newregion r′ in
    let y = v′ at r′ in
    ⋮
    return y
```

Unfortunately, we cannot write this program in the Tofte-Talpin language because it is based on the idea of lexical scoping. Another consequence of this language design is that any program transformation

that alters program structure can affect memory management. One of the most devastating transformations for the Tofte-Talpin type system is the continuation-passing style (CPS) transformation; each successive computation is placed in the scope of all previous computations, with the result that no regions can be deallocated until the entire computation has been completed. In the following example, the CPS transformation prevents the region `r` from being deallocated until after `code` has been executed when it could be deallocated as soon as `f` has completed its computation.

```
letregion r in                    letregion r in
    f (v)                   ⇒         f (v, λ.code)
end;                              end
code
```

The observation that the Tofte-Talpin type system will make poor use of memory in such cases has been made before. Both Birkedal *et al.* [6] and Aiken *et al.* [1] have proposed optimizations that allow regions to be freed early. However, although their optimizations are safe, there is no simple proof- or type-checker that an untrusting client can use to check the output code. Therefore, in order to construct a verifyably safe, efficient region-based language, we must re-examine the fundamental question: *"When can a program access a value* `v`*?"*

1.2 Contributions

Our solution to the problem of provably safe yet efficient region-based memory management takes its inspiration from operating systems such as Hydra [55]. Hydra solves the access control problem by associating an unforgeable key or *capability* with every object and requiring that the user present this capability to gain access to the object. Furthermore, when the need arises, Hydra revokes capabilities, thereby preventing future access to the protected objects.

We define a new strongly-typed compiler intermediate language for region-based memory management that uses a compile-time notion of capability to ensure that region operations are performed safely. Unlike Tofte and Talpin's language, lexical scoping plays no part in the verification process. Instead, the type system threads static information in the form of capabilities along the control-flow path of a program. In order to use a value in region `r` at a particular control-flow point, the program must present the capability for that region. As in traditional capability systems, our type system keeps track of capability copies carefully so that it can determine when a capability has truly been revoked. Unlike traditional capability systems, our calculus supports only voluntary revocation. However, the capabilities in our calculus are a purely static concept and thus their implementation requires no run-time overhead. This mechanism provides an efficient way to check the safety of explicit, arbitrarily ordered region allocation and deallocation instructions.

We have a purely *syntactic* argument, based on Subject Reduction and Progress lemmas in the style of Felleisen and Wright [54], that the type system of the Capability Calculus is sound. In contrast, Tofte and Talpin formulate the soundness of their system using a more complicated greatest fixed point argument [48], and the soundness of Aiken *et al.*'s optimizations [1] depend upon this argument. Part of the reason for the extra complexity is that Tofte and Talpin simultaneously show that region *inference* translates lambda calculus terms into operationally equivalent region calculus terms, a stronger property than we prove. However, when system security is the main concern, soundness is the critical property.

The simplicity of our argument demonstrates the benefits of separating type soundness from type inference or optimization correctness.

We also have a formal translation of a variant of the Tofte-Talpin language into our calculus. Given a type-safe Tofte-Talpin program, the translation always produces a type-safe Capability Calculus program. Therefore, when the translation is combined with a region inference algorithm [46], it provides a way to compile source language programs into type-safe low-level code that can be used in secure extensible systems or mobile code.

The remaining sections of this article describe the Capability Calculus in greater detail. Section 2 presents the syntax and semantics of the language formally and motivates the design decisions that we made. At the end of this section, we present the type soundness theorem and discuss the most interesting parts of our proof. The complete proof may be found in Appendix A. Section 3 demonstrates that the Capability Calculus is at least as expressive as the Tofte-Talpin language. We define the semantics of a variant of the latter language and give a translation into the Capability Calculus. The translation preserves the type safety property and Appendix B proves this fact. We further demonstrate the expressiveness of the Capability Calculus by sketching a couple of optimizations that are not possible in the more restrictive language. Section 4 describes how to compile the relatively high-level Capability Calculus into a capability-based Typed Assembly Language. Most elements of this translation are orthogonal to the capability mechanism and therefore this section is mostly informal, although the capability-based Typed Assembly Language has been fully defined. We claim without proof that this target language is type-safe. We believe we could prove a type safety result, but the proof would be long and tedious and we doubt it would reveal any new insights. Finally, we believe that our notion of static capability is generally useful concept. Section 5 informally explores several other applications of capabilities. This section also explains further connections with related work. Finally, Section 6 concludes.

2 The Capability Calculus

The central technical contribution of this article is the Capability Calculus, a statically-typed intermediate language that supports the explicit allocation, freeing and accessing of memory regions.

As mentioned in the introduction, the type system for the language propagates static information (capabilities) along the control-flow path of a program. Therefore, the most elegant and natural form for the language is continuation-passing style (CPS) [40]. That is, functions in the Capability Calculus do not return values; instead, functions finish by calling a continuation function that is typically provided as an argument. The fact that there is only one means of transferring control in CPS—rather than the two means (call and return) in direct style—simplifies the tracking of capabilities. A direct style formulation is possible, but the complications involved obscure the central issues. In the remainder of this paper, we assume familiarity with CPS.

The syntax of the Capability Calculus appears in Figure 1. In the following sections, we explain and motivate the main constructs and typing rules of the language one by one.

kinds	κ	::=	$\mathtt{Type} \mid \mathtt{Rgn} \mid \mathtt{Cap}$
constructor vars	α, ρ, ϵ		
constructors	c	::=	$\alpha \mid \tau \mid r \mid C$
types	τ	::=	$\alpha \mid \mathtt{int} \mid r\ \mathtt{handle} \mid \forall[\Delta].(C, \tau_1, \ldots, \tau_n) \to 0\ \mathtt{at}\ r \mid \langle \tau_1, \ldots, \tau_n \rangle\ \mathtt{at}\ r$
regions	r	::=	$\rho \mid \nu$
capabilities	C	::=	$\epsilon \mid \emptyset \mid \{r^{\varphi}\} \mid C_1 \oplus C_2 \mid \overline{C}$
multiplicities	φ	::=	$1 \mid +$
constructor contexts	Δ	::=	$\cdot \mid \Delta, \alpha{:}\kappa \mid \Delta, \epsilon \leq C$
value contexts	Γ	::=	$\cdot \mid \Gamma, x{:}\tau$
region types	Υ	::=	$\{\ell_1 : \tau_1, \ldots, \ell_n : \tau_n\}$
memory types	Ψ	::=	$\{\nu_1 : \Upsilon_1, \ldots, \nu_n : \Upsilon_n\}$
word values	v	::=	$x \mid i \mid \nu.\ell \mid \mathtt{handle}(\nu) \mid v[c]$
heap values	h	::=	$\mathtt{fix}\, f[\Delta](C, x_1{:}\tau_1, \ldots, x_n{:}\tau_n).e \mid \langle v_1, \ldots, v_n \rangle$
arithmetic ops	p	::=	$+ \mid - \mid \times$
declarations	d	::=	$x = v \mid x = v_1\ p\ v_2 \mid x = h\ \mathtt{at}\ v \mid x = \pi_i v \mid \mathtt{newrgn}\, \rho, x \mid \mathtt{freergn}\, v$
terms	e	::=	$\mathtt{let}\, d\ \mathtt{in}\, e \mid \mathtt{if0}\, v\ \mathtt{then}\, e_2\ \mathtt{else}\, e_3 \mid v(v_1, \ldots, v_n) \mid \mathtt{halt}\ v$
memory regions	R	::=	$\{\ell_{\Gamma} \mapsto h_1, \ldots, \ell_{\overline{n}} \mapsto h_n\}$
memories	M	::=	$\{\nu_{\Gamma} \mapsto R_1, \ldots, \nu_{\overline{n}} \mapsto R_n\}$
machine states	P	::=	(M, e)

Figure 1: Capability Syntax

2.1 Preliminaries

We specify the operational behavior of the Capability Calculus using an allocation semantics [30, 31, 33], which makes the allocation of data in memory explicit. The semantics, which is specified in Figure 2, is given by a deterministic rewriting system $P \longmapsto P'$ mapping machine states to new machine states. A machine state consists of a pair (M, e) of a memory and a term being executed. A memory is a finite mapping of *region names* (ν) to *regions* where a region is a block of memory that holds a collection of heap-allocated objects. Regions are created at run time by the declaration $\texttt{newrgn}\,\rho, x$, which allocates a new region in the heap, binds ρ to the name of that region, and binds x to the *handle* ($\texttt{handle}(\nu)$) for that region.

Region names and handles are distinguished in order to maintain a phase distinction between compile-time and run-time expressions. Region names are significant at compile time: The type-checker identifies which region an object inhabits via a region name (see below). However, region names, like other type constructors, have no run-time significance and may be erased from executable code. In contrast, region handles hold the run-time data necessary to manipulate regions. In addition to accounting for a phase distinction, the separation of region names and handles also allows us to refine the contexts in which region handles are needed. Handles are needed when allocating objects within a region and when freeing a region, but are not needed when reading data from a region.

Regions are freed by the declaration $\texttt{freergn}\,v$, where v is the handle for the region to be freed. Objects h large enough to require heap allocation (*i.e.,* functions and tuples), called *heap values,* are allocated by the declaration $x = h\,\texttt{at}\,v$, where v is the handle for the region in which h is to be allocated. Data is read from a region in two ways: functions are read by a function call, and tuples are read by the declaration $x = \pi_i(v)$, which binds x to the data residing in the ith field of the object at address v. Each of these operations may be performed only when the region in question has not already been freed. Enforcing this restriction is the purpose of the capability mechanism discussed in Section 2.2.

A region maps locations (ℓ) to heap values. Thus, an address is given by a pair $\nu.\ell$ of a region name and a location. In the course of execution, word-sized values (v) will be substituted for value variables and type constructors for constructor variables, but heap values (h) are always allocated in memory and referred to indirectly by an address. Thus, when executing the declaration $x = h\ \texttt{at}\ r$ (where r is $\texttt{handle}(\nu)$, the handle for region ν), h is allocated in region ν (say at ℓ) and the address $\nu.\ell$ is substituted for x in the following code.

A term in the Capability Calculus consists of a series of declarations ending in either a branch or a function call (or a halt). The class of declarations includes those constructs discussed above, plus two standard constructs, $x = v$ for binding variables to values and $x = v_1\ p\ v_2$ (where p ranges over $+$, $-$ and $\times$) for arithmetic.

Types The types of the Capability Calculus include type constructor variables and integers, a type of region handles, as well as tuple and function types. If r is a region, then r `handle` is the type of r's region handle. The tuple type $\langle \tau_1, \ldots, \tau_n \rangle\ \texttt{at}\ r$ contains the usual n field tuples, but also specifies that such tuples are allocated in region r, where r is either a region name ν or, more frequently, a region variable ρ.

The function type $\forall[].(C, \tau_1, \ldots, \tau_n) \rightarrow \texttt{0}\ \texttt{at}\ r$ contains functions taking n arguments (with types τ_1 through τ_n) that may be called when capability C is satisfied (see the next section). The `0` return type is intended to suggest the fact that CPS functions invoke their continuations rather than returning as

$(M, e) \longmapsto P$	
If $e =$	then $P =$
$\texttt{let}\, x = v\, \texttt{in}\, e'$	$(M, e'[v/x])$
$\texttt{let}\, x = i\ p\ j\, \texttt{in}\, e'$	$(M, e'[(i\ p\ j)/x])$
$\texttt{let}\, x = h\, \texttt{at}\, (\texttt{handle}(\nu))\, \texttt{in}\, e'$ and $\nu \in Dom(M)$	$(M\{\nu.\ell \mapsto h\}, e'[\nu.\ell/x])$ where $\ell \notin Dom(M(\nu))$
$\texttt{let}\, x = \pi_i(\nu.\ell)\, \texttt{in}\, e'$ and $\nu \in Dom(M)$ and $\ell \in Dom(M(\nu))$	$(M, e'[v_i/x])$ where $M(\nu.\ell) = \langle v_0, \ldots, v_{n-1}\rangle \quad (0 \le i < n)$
$\texttt{let newrgn}\, \rho, x\, \texttt{in}\, e'$	$(M\{\nu \mapsto \{\}\}, e'[\nu, \texttt{handle}(\nu)/\rho, x])$ where $\nu \notin M$ and $\nu \notin e'$
$\texttt{let freergn}\, (\texttt{handle}(\nu))\, \texttt{in}\, e'$ and $\nu \in Dom(M)$	$(M \backslash \nu, e')$
$\texttt{if0}\, 0\, \texttt{then}\, e_2\, \texttt{else}\, e_3$	(M, e_2)
$\texttt{if0}\, i\, \texttt{then}\, e_2\, \texttt{else}\, e_3$ and $i \neq 0$	(M, e_3)
$v(v_1, \ldots, v_n)$	$(M, e[c_1, \ldots, c_m, \nu.\ell, v_1, \ldots, v_n/\alpha_1, \ldots, \alpha_m, f, x_1, \ldots, x_n])$ where $v = \nu.\ell[c_1, \ldots, c_m]$ and $M(\nu.\ell) = \texttt{fix}\, f[\Delta](C, x_1{:}\tau_1, \ldots, x_n{:}\tau_n).e$ and $Dom(\Delta) = \alpha_1, \ldots, \alpha_m$

Figure 2: Capability Operational Semantics

a direct-style function does. The suffix "$\texttt{at}\,r$", like the corresponding suffix for tuple types, indicates the region in which the function is allocated.

Functions may be made polymorphic over types, regions or capabilities by adding a constructor context Δ to the function type. For convenience, types, regions and capabilities are combined into a single syntactic class of "constructors" and are distinguished by kinds. Thus, a type is a constructor with kind `Type`, a region is a constructor with kind `Rgn`, and a capability is a constructor with kind `Cap`. We use the metavariable c to range over constructors, but use the metavariables τ, r and C when those constructors are types, regions and capabilities, respectively. We also use the metavariables ρ and ϵ for constructor variables of kind `Rgn` and `Cap`, and use the metavariable α for type variables and generic constructor variables. When Δ is empty, we abbreviate the function type $\forall[\Delta].(C,\vec{\tau}) \rightarrow \texttt{0}\ \texttt{at}\ r$ by $(C,\vec{\tau}) \rightarrow \texttt{0}\ \texttt{at}\ r$.

For example, a polymorphic identity function that is allocated in region r, but whose continuation function may be in any region, may be given type

$$\forall[\alpha{:}\texttt{Type},\rho{:}\texttt{Rgn}].(C,\alpha,(C,\alpha) \rightarrow \texttt{0}\ \texttt{at}\ \rho) \rightarrow \texttt{0}\ \texttt{at}\ r$$

for some appropriate C. Let f be such a function, let v be its argument with type τ, and let g be its continuation with type $(C,\tau) \rightarrow \texttt{0}\ \texttt{at}\ r$. Then f is called by $f[\tau][r](v,g)$.

Figure 3 specifies all well-formed constructors and constructor contexts. The two main judgments $\Delta \vdash \Delta'$ and $\Delta \vdash c : \kappa$ assume that the constructor context Δ is well-formed. The first judgement states that Δ' is a well-formed constructor context and the second judgement states c is a well-formed constructor with kind κ.

The typing rules also use region types (Υ), which assign a type to every location allocated in a region, and memory types (Ψ), which assign a region type to every region allocated in memory. However, it is not necessary to understand these constructs in the preliminary development, and therefore we will defer discussing them until we describe the static semantics of the abstract machine in formal detail (see Section 2.4).

2.2 Capabilities

The central problem is how to ensure statically that no region is used after it is freed. The typing rules enforce this with a system of capabilities that specify what operations are permitted. The main typing judgement is

$$\Psi;\Delta;\Gamma;C \vdash e$$

which states that (when memory has type Ψ, free constructor variables have kinds given by Δ and free value variables have types given by Γ) it is legal to execute the term e, *provided that the capability C is held.* A related typing judgement is

$$\Psi;\Delta;\Gamma;C \vdash d \Rightarrow \Delta';\Gamma';C'$$

which states that if the capability C is held, it is legal to execute the declaration d, which results in new constructor context Δ', new value context Γ' and new capability C'.

Capabilities indicate the set of regions that are presently valid to access, that is, those regions that have not been freed. Capabilities are formed by joining together a collection of singleton capabilities $\{r\}$

$\boxed{\Delta \vdash \Delta'}$

$$\frac{}{\Delta \vdash \cdot}\ \text{(ctxt-empty)} \qquad \frac{\Delta \vdash \Delta'}{\Delta \vdash \Delta', \alpha{:}\kappa}\ (\alpha \notin Dom(\Delta\Delta'))\ \text{(ctxt-var)}$$

$$\frac{\Delta \vdash \Delta' \quad \Delta\Delta' \vdash C : \texttt{Cap}}{\Delta \vdash \Delta', \epsilon \leq C}\ (\epsilon \notin Dom(\Delta\Delta'))\ \text{(ctxt-sub)}$$

$\boxed{\Delta \vdash c : \kappa}$

$$\frac{}{\Delta \vdash \alpha : \kappa}\ (\Delta(\alpha) = \kappa)\ \text{(type-var)} \qquad \frac{}{\Delta \vdash \epsilon : \texttt{Cap}}\ ((\epsilon \leq C) \in \Delta)\ \text{(type-sub)}$$

$$\frac{}{\Delta \vdash \texttt{int} : \texttt{Type}}\ \text{(type-int)} \qquad \frac{\Delta \vdash r : \texttt{Rgn}}{\Delta \vdash r\ \texttt{handle} : \texttt{Type}}\ \text{(type-handle)}$$

$$\frac{\Delta \vdash \tau_i : \texttt{Type} \quad (\text{for } 1 \leq i \leq n) \quad \Delta \vdash r : \texttt{Rgn}}{\Delta \vdash \langle \tau_1, \ldots, \tau_n \rangle\ \texttt{at}\ r : \texttt{Type}}\ \text{(type-tuple)}$$

$$\frac{\begin{array}{c}\Delta \vdash \Delta' \quad \Delta\Delta' \vdash \tau_i : \texttt{Type} \quad (\text{for } 1 \leq i \leq n) \\ \Delta\Delta' \vdash C : \texttt{Cap} \quad \Delta \vdash r : \texttt{Rgn}\end{array}}{\Delta \vdash \forall[\Delta'].(C, \tau_1, \ldots, \tau_n) \to 0\ \texttt{at}\ r : \texttt{Type}}\ \text{(type-arrow)}$$

$$\frac{}{\Delta \vdash \nu : \texttt{Rgn}}\ \text{(type-name)} \qquad \frac{}{\Delta \vdash \emptyset : \texttt{Cap}}\ \text{(type-}\emptyset\text{)} \qquad \frac{\Delta \vdash r : \texttt{Rgn}}{\Delta \vdash \{r^{\varphi}\} : \texttt{Cap}}\ \text{(type-single)}$$

$$\frac{\Delta \vdash C_1 : \texttt{Cap} \quad \Delta \vdash C_2 : \texttt{Cap}}{\Delta \vdash C_1 \oplus C_2 : \texttt{Cap}}\ \text{(type-plus)} \qquad \frac{\Delta \vdash C : \texttt{Cap}}{\Delta \vdash \overline{C} : \texttt{Cap}}\ \text{(type-bar)}$$

Figure 3: Capability Static Semantics: Type and Context Formation

that provide access to only one region, and capability variables ϵ that provide access to an unspecified set of regions. Capability joins, written $C_1 \oplus C_2$, are associative and commutative, but are not always idempotent; in Section 2.3 we will see examples where $C \oplus C$ is not equivalent to C. The empty capability, which provides access to no regions, is denoted by $\emptyset$. We will often abbreviate the capability $\{r_1\} \oplus \cdots \oplus \{r_n\}$ by $\{r_1, \ldots, r_n\}$.

In order to read a field from a tuple in region r, it is necessary to hold the capability to access r, as in the rule:

$$\frac{\begin{array}{c}\Delta \vdash C = C' \oplus \{r\} : \texttt{Cap} \\ \Psi; \Delta; \Gamma \vdash v : \langle \tau_1, \ldots, \tau_n \rangle \texttt{ at } r\end{array}}{\Psi; \Delta; \Gamma; C \vdash x = \pi_i(v) \Rightarrow \Delta; \Gamma\{x{:}\tau_i\}; C} \; (x \notin Dom(\Gamma))$$

The first subgoal indicates that the capability held (C) is equivalent to some capability that includes $\{r\}$.

A similar rule is used to allocate an object in a region. Since the type of a heap value reflects the region in which it is allocated, the heap value typing judgement (the second subgoal below) must be provided with that region.

$$\frac{\begin{array}{c}\Delta \vdash C = C' \oplus \{r\} : \texttt{Cap} \\ \Psi; \Delta; \Gamma \vdash h \texttt{ at } r : \tau \\ \Psi; \Delta; \Gamma \vdash v : r \texttt{ handle}\end{array}}{\Psi; \Delta; \Gamma; C \vdash x = h \texttt{ at } v \Rightarrow \Delta; \Gamma\{x{:}\tau\}; C} \; (x \notin Dom(\Gamma))$$

Functions Functions are defined by the form $\texttt{fix}\, f[\Delta](C, x_1{:}\tau_1, \ldots, x_n{:}\tau_n).e$, where f stands for the function itself and may appear free in the body, Δ specifies the function's constructor arguments, and C is the function's capability precondition. When Δ is empty and f does not appear free in the function body we abbreviate the `fix` form by $\lambda(C, x_1{:}\tau_1, \ldots, x_n{:}\tau_n).e$.

In order to call a function residing in region r, it is again necessary to hold the capability to access r, and also to hold a capability equivalent to the function's capability precondition:

$$\frac{\begin{array}{c}\Delta \vdash C = C'' \oplus \{r\} : \texttt{Cap} \quad \Delta \vdash C = C' : \texttt{Cap} \\ \Psi; \Delta; \Gamma \vdash v : (C', \tau_1, \ldots, \tau_n) \rightarrow \texttt{0 at } r \\ \Psi; \Delta; \Gamma \vdash v_i : \tau_i\end{array}}{\Psi; \Delta; \Gamma; C \vdash v(v_1, \ldots, v_n)}$$

The body of a function may then assume the function's capability precondition is satisfied, as indicated by the capability C in the premise of the rule:[1]

$$\frac{\Psi; \Delta; \Gamma\{x_1{:}\tau_1, \ldots, x_n{:}\tau_n\}; C \vdash e}{\Psi; \Delta; \Gamma \vdash \lambda(C, x_1{:}\tau_1, \ldots, x_n{:}\tau_n).e \texttt{ at } r : \tau_f} \; (x_i \notin Dom(\Gamma))$$

As might be expected, the annotation "`at` r" indicates that the closure value resides in region r. The resultant function type τ_f is $(C, \tau_1, \ldots, \tau_n) \rightarrow \texttt{0 at } r$.

Often, we will extend the required capability for a function with a quantified capability variable (similar to a *row variable*). This variable may be instantiated with whatever capabilities are leftover

[1] This rule specializes the full rule for *fix* to the case where the function is neither polymorphic nor recursive.

after satisfying the required capability. Consequently, the function may be used in a variety of contexts. For example, functions with type

$$\forall[\epsilon{:}\mathtt{Cap}].(\{r\} \oplus \epsilon, \ldots) \to 0 \ \mathtt{at}\ r$$

may be called with any capability that extends $\{r\}$.

Allocation and Deallocation The most delicate issue is the typing of region allocation and deallocation. Intuitively, the typing rules for the `newrgn` and `freergn` declarations should add and remove capabilities for the appropriate region. Naive typing rules could be:

$$\frac{}{\Psi;\Delta;\Gamma;C \vdash \mathtt{newrgn}\,\rho, x \Rightarrow \Delta\{\rho{:}\mathtt{Rgn}\};\Gamma\{x{:}\rho\ \mathtt{handle}\};C \oplus \{\rho\}} \text{ (wrong)}$$

$$\frac{\Psi;\Delta;\Gamma \vdash v : r\ \mathtt{handle} \quad C' = C \setminus \{r\}}{\Psi;\Delta;\Gamma;C \vdash \mathtt{freergn}\,v \Rightarrow \Delta;\Gamma;C'} \text{ (wrong)}$$

We will be able to use something much like the first rule for allocation, but the naive rule for freeing regions is fundamentally flawed. For example, consider the following function:

$$\begin{array}{l}\mathtt{fix}\, f[\rho_1{:}\mathtt{Rgn}, \rho_2{:}\mathtt{Rgn}](\{\rho_1, \rho_2\}, x{:}\rho_1\ \mathtt{handle}, y{:}\langle\mathtt{int}\rangle\ \mathtt{at}\ \rho_2).\\ \quad \mathtt{let}\ \mathtt{freergn}\, x\ \mathtt{in}\\ \quad \mathtt{let}\ z = \pi_0 y\ \mathtt{in} \cdots\end{array}$$

This function is well-formed according to the naive typing rule: The function begins with the capability $\{\rho_1, \rho_2\}$ and ρ_1 is removed by the `freergn` declaration, leaving $\{\rho_2\}$. The tuple y is allocated in ρ_2, so the projection is legal. However, this code is operationally incorrect if ρ_1 and ρ_2 are instantiated by the same region r. In that case, the first declaration frees r and the second attempts to read from r.

This problem is a familiar one. To free a region safely it is necessary to delete all copies of the capability. However, instantiating region variables can create aliases, making it impossible to tell by inspection whether any copies exist.

2.3 Alias Control

We desire a system for alias control that can easily be enforced by the type system, without expensive and complex program analyses. One possibility is a linear type system [16, 49, 50]. In a linear type system, aliasing would be trivially controlled; any use of a region name would consume that name, ensuring that it could not be used elsewhere. Thus, in a linear type system, the naive rules for allocating and deallocating regions would be sound. Unfortunately, a linear type system is too restrictive to permit many useful programs. For example, suppose f has type

$$\forall[\rho_1{:}\mathtt{Rgn}, \rho_2{:}\mathtt{Rgn}].(\{\rho_1, \rho_2\}, \langle\mathtt{int}\rangle\ \mathtt{at}\ \rho_1, \langle\mathtt{int}\rangle\ \mathtt{at}\ \rho_2, \ldots) \to 0\ \mathtt{at}\ r'$$

and v_1 and v_2 are integer tuples allocated in the same region r. Then f could not be called with v_1 and v_2 as arguments, because that would require instantiating ρ_1 and ρ_2 with the same region. More generally, one could not type any function that takes two arguments that might or might not be allocated in the same region.

Approaches based on syntactic control of interference [41, 42] are more permissive than a linear type system, but are still too restrictive for our purposes; it is still impossible to instantiate multiple arguments with the same region.

Uniqueness Our approach, instead of trying to *prevent* aliasing, is to use the type system to *track* aliasing. More precisely, we track *non*-aliasing, that is, uniqueness. We do this by tagging regions with one of two *multiplicities* when forming a capability. The first form, $\{r^+\}$, is the capability to access region r as it has been understood heretofore. The second form, $\{r^1\}$, also permits accessing region r, but adds the additional information that r is unique; that is, r represents a different region from any other region appearing in a capability formed using $\{r^1\}$. For example, the capability $\{r_1^+, r_2^1\}$ not only indicates that it is permissible to access r_1 and r_2, but also indicates that r_1 and r_2 represent distinct regions.

Since $\{r^1\}$ guarantees that r does not appear anywhere else in a capability formed using it, it is the capability, not just to access r, but also to free r. Thus we may type region deallocation with the rule:

$$\frac{\Delta;\Gamma \vdash v : r\ \texttt{handle} \quad \Delta \vdash C = C' \oplus \{r^1\} : \texttt{Cap}}{\Psi;\Delta;\Gamma;C \vdash \texttt{freergn}\, v \Rightarrow \Delta;\Gamma;C'}$$

Allocation of a region accordingly adds the new capability as unique:

$$\frac{}{\Psi;\Delta;\Gamma;C \vdash \texttt{newrgn}\, \rho, x \Rightarrow \Delta\{\rho{:}\texttt{Rgn}\};\Gamma\{x{:}\rho\ \texttt{handle}\};C \oplus \{\rho^1\}} \ (\rho \notin Dom(\Delta), x \notin Dom(\Gamma))$$

Note that joining capabilities is only idempotent when the capabilities in question contain no unique multiplicities. For instance, the capabilities $\{r^+\}$ and $\{r^+, r^+\}$ are equivalent, but the capabilities $\{r^1\}$ and $\{r^1, r^1\}$ are not; the latter capability ($\{r^1, r^1\}$) asserts that r is distinct from itself and consequently that latter capability can never be satisfied.

When C is equivalent to $C \oplus C$, we say that C is *duplicatable.* Note that capability variables are unduplicatable, since they can stand for any capability, including unduplicatable ones. Occasionally this prevents the typing of desired programs, so we provide a stripping operator $\overline{C}$ that replaces all 1 multiplicities in C with + multiplicities. For example, $\overline{\{r_1^1, r_2^+\}} = \{r_1^+, r_2^+\}$. For any capability C, the capability $\overline{C}$ is duplicatable. When programs need an unknown but duplicatable capability, they may use a stripped variable $\overline{\epsilon}$. As you will see in section 3, the stripping operator is essential in the translation of Tofte and Talpin's region-based language into the Capability Calculus.

The complete rules for equivalence of capabilities and other constructors appear in Figure 4. Notice that the single rule eq-flag equates the duplicatable capability $\{r^+\}$ with the barred capability $\overline{\{r^1\}}$. Consequently, the form $\{r^+\}$ is redundant given the presence of the bar operator. However, the + notation is a pleasing foil for the 1 notation and the two flags give us a convenient way to distinguish between regions that appear once and regions that potentially appear many times in a single capability.

Subcapabilities The capabilities $\{r^1\}$ and $\{r^+\}$ are not the same, but the former should provide all the privileges of the latter. We therefore say that the former is a subcapability of the latter and write $\{r^1\} \leq \{r^+\}$. In the complete system, the various rules from Section 2.2 are modified to account for subcapabilities. For example, the function call rule becomes:

$$\frac{\begin{array}{c}\Psi;\Delta;\Gamma \vdash v : (C', \tau_1, \ldots, \tau_n) \to 0\ \texttt{at}\ r \\ \Delta \vdash C \leq C'' \oplus \{r^+\} \quad \Delta \vdash C \leq C' \\ \Psi;\Delta;\Gamma \vdash v_i : \tau_i\end{array}}{\Psi;\Delta;\Gamma;C \vdash v(v_1, \ldots, v_n)}$$

$\boxed{\Delta \vdash \Delta_1 = \Delta_2}$

$$\frac{}{\Delta \vdash \cdot = \cdot}\ \text{(ctxt-eq-empty)} \qquad \frac{\Delta \vdash \Delta_1 = \Delta_2}{\Delta \vdash \Delta_1, \alpha{:}\kappa = \Delta_2, \alpha{:}\kappa}\ (\alpha \notin \Delta\Delta_1)\ \text{(ctxt-eq-kind)}$$

$$\frac{\Delta \vdash \Delta_1 = \Delta_2 \quad \Delta\Delta_1 \vdash C_1 = C_2 : \texttt{Cap}}{\Delta \vdash \Delta_1, \epsilon \leq C_1 = \Delta_2, \epsilon \leq C_2}\ (\epsilon \notin \Delta\Delta_1)\ \text{(ctxt-eq-bound)}$$

$\boxed{\Delta \vdash c_1 = c_2 : \kappa}$

$$\frac{\Delta \vdash c : \kappa}{\Delta \vdash c = c : \kappa}\ \text{(eq-reflex)} \qquad \frac{\Delta \vdash c_2 = c_1 : \kappa}{\Delta \vdash c_1 = c_2 : \kappa}\ \text{(eq-symm)}$$

$$\frac{\Delta \vdash c_1 = c_2 : \kappa \quad \Delta \vdash c_2 = c_3 : \kappa}{\Delta \vdash c_1 = c_3 : \kappa}\ \text{(eq-trans)}$$

$$\frac{\Delta \vdash C_1 = C_1' : \texttt{Cap} \quad \Delta \vdash C_2 = C_2' : \texttt{Cap}}{\Delta \vdash C_1 \oplus C_2 = C_1' \oplus C_2' : \texttt{Cap}}\ \text{(eq-congruence-plus)} \quad \frac{\Delta \vdash C = C' : \texttt{Cap}}{\Delta \vdash \overline{C} = \overline{C'} : \texttt{Cap}}\ \text{(eq-congruence-bar)}$$

$$\frac{\Delta \vdash C : \texttt{Cap}}{\Delta \vdash \emptyset \oplus C = C : \texttt{Cap}}\ \text{(eq-}\emptyset\text{)} \qquad \frac{\Delta \vdash C_1 : \texttt{Cap} \quad \Delta \vdash C_2 : \texttt{Cap}}{\Delta \vdash C_1 \oplus C_2 = C_2 \oplus C_1 : \texttt{Cap}}\ \text{(eq-comm)}$$

$$\frac{\Delta \vdash C_i : \texttt{Cap} \quad (\text{for } 1 \leq i \leq 3)}{\Delta \vdash (C_1 \oplus C_2) \oplus C_3 = C_1 \oplus (C_2 \oplus C_3) : \texttt{Cap}}\ \text{(eq-assoc)} \qquad \frac{\Delta \vdash C : \texttt{Cap}}{\Delta \vdash \overline{C} = \overline{C} \oplus \overline{C} : \texttt{Cap}}\ \text{(eq-dup)}$$

$$\frac{}{\Delta \vdash \overline{\emptyset} = \emptyset : \texttt{Cap}}\ \text{(eq-bar-}\emptyset\text{)} \qquad \frac{\Delta \vdash r : \texttt{Rgn}}{\Delta \vdash \overline{\{r^1\}} = \{r^+\} : \texttt{Cap}}\ \text{(eq-flag)}$$

$$\frac{\Delta \vdash C : \texttt{Cap}}{\Delta \vdash \overline{\overline{C}} = \overline{C} : \texttt{Cap}}\ \text{(eq-bar-idem)} \qquad \frac{\Delta \vdash C_1 : \texttt{Cap} \quad \Delta \vdash C_2 : \texttt{Cap}}{\Delta \vdash \overline{C_1 \oplus C_2} = \overline{C_1} \oplus \overline{C_2} : \texttt{Cap}}\ \text{(eq-distrib)}$$

Figure 4: Capability Static Semantics: Equality

$\boxed{\Delta \vdash C_1 \leq C_2}$

$$\frac{\Delta \vdash C_1 = C_2 : \texttt{Cap}}{\Delta \vdash C_1 \leq C_2}\ \text{(sub-eq)} \qquad \frac{\Delta \vdash C_1 \leq C_2 \quad \Delta \vdash C_2 \leq C_3}{\Delta \vdash C_1 \leq C_3}\ \text{(sub-trans)}$$

$$\frac{\Delta \vdash C_1 \leq C_1' \quad \Delta \vdash C_2 \leq C_2'}{\Delta \vdash C_1 \oplus C_2 \leq C_1' \oplus C_2'}\ \text{(sub-congruence-plus)} \qquad \frac{\Delta \vdash C \leq C'}{\Delta \vdash \overline{C} \leq \overline{C'}}\ \text{(sub-congruence-bar)}$$

$$\frac{}{\Delta \vdash \epsilon \leq C}\ ((\epsilon \leq C) \in \Delta)\ \text{(sub-var)} \qquad \frac{\Delta \vdash C : \texttt{Cap}}{\Delta \vdash C \leq \overline{C}}\ \text{(sub-bar)}$$

Figure 5: Capability Static Semantics: Equality and Subcapability Relations

Suppose f has type $\forall[\rho_1{:}\texttt{Rgn}, \rho_2{:}\texttt{Rgn}].(\{\rho_1^+, \rho_2^+\}, \ldots) \to \texttt{0 at } r$. If we hold capability $\{r^+\}$, we may call f by instantiating ρ_1 and ρ_2 with r, since $\{r^+\} = \{r^+, r^+\}$. Using the subcapability relation, we may also call f when we hold $\{r^1\}$, again by instantiating ρ_1 and ρ_2 with r, since $\{r^1\} \leq \{r^+\} = \{r^+, \overline{r^+}\}$.

Figure 5 contains the subcapability rules. When reading these rules, remember that $\Delta \vdash \overline{\{r^1\}} = \{r^+\} : \texttt{Cap}$. We use this fact to derive the judgement $\Delta \vdash \{r^1\} \leq \{r^+\} : \texttt{Cap}$ that we discussed informally above:

$$\dfrac{\dfrac{}{\Delta \vdash \{r^1\} \leq \overline{\{r^1\}} : \texttt{Cap}}\ \text{(sub-bar)} \quad \dfrac{\dfrac{}{\Delta \vdash \overline{\{r^1\}} = \{r^+\} : \texttt{Cap}}\ \text{(eq-flag)}}{\Delta \vdash \overline{\{r^1\}} \leq \{r^+\} : \texttt{Cap}}\ \text{(sub-eq)}}{\Delta \vdash \{r^1\} \leq \{r^+\} : \texttt{Cap}}\ \text{(sub-trans)}$$

The subcapability relation accounts only for the forgetting of uniqueness information. Intuitively there could be a second source of subcapabilities, those generated by forgetting an entire capability. For example, $\{r_1^+, r_2^+\}$ seems to provide all the privileges of $\{r_1^+\}$, so it is reasonable to suppose $\{r_1^+, r_2^+\}$ to be subcapability of $\{r_1^+\}$. Indeed, one can construct a sound Capability Calculus incorporating this axiom, but we omit it because doing so allows us to specify memory management obligations and to prove a stronger property about space usage. Notice also, that by omitting this axiom, we do not give up any flexibility; one may always write a function that can be called with extra capabilities by using a capability variable, as discussed in Section 2.2.

By omitting the axiom $C_1 \oplus C_2 \leq C_1$, our type system may formally specify who has responsibility for freeing a region. Failure to follow informal conventions is a common source of bugs in languages (such as C) that use manual memory management. Our type system rules out such bugs. For example, consider the type:

$$\forall[\rho{:}\texttt{Rgn}, \epsilon{:}\texttt{Cap}].(\epsilon \oplus \{r^+, \rho^1\}, \rho\ \texttt{handle}, (\epsilon \oplus \{r^+\}) \to \texttt{0 at } r) \to \texttt{0 at } r$$

In our system $\epsilon \oplus \{r^+, \rho^1\} \not\leq \epsilon \oplus \{r^+\}$. Consequently, before any function with this type can return

(*i.e.*, call the continuation of type $(\epsilon \oplus \{r^+\}) \to \texttt{0 at } r$), it must take action to satisfy the capability $\epsilon \oplus \{r^+\}$, that is, it must free ρ.

In general, our type system prevents "region leaks": Programs must deallocate all memory regions before they terminate (Theorem 2.5). Therefore, the operating system does not have to clean up after a program halts. The typing rule for `halt` states that no capabilities may be held, and since capabilities may not be forgotten, this means that all regions must have been freed.

$$\frac{\Psi;\Delta;\Gamma \vdash v : \texttt{int} \quad \Delta \vdash C = \emptyset : \texttt{Cap}}{\Psi;\Delta;\Gamma;C \vdash \texttt{halt } v}$$

Bounded Quantification The system presented to this point is sound, but it is not yet sufficient for compiling real source languages. We need to be able to recover uniqueness after a region name is duplicated. To see why, suppose we hold the capability $\{r^1\}$ and f has type:

$$\forall[\rho_1{:}\texttt{Rgn}, \rho_2{:}\texttt{Rgn}].(\{\rho_1^+, \rho_2^+\}, \ldots, (\{\rho_1^+, \rho_2^+\}, \ldots) \to \texttt{0 at } \rho_1) \to \texttt{0 at } r$$

We would like to be able to instantiate ρ_1 and ρ_2 with r (which we may do, since $\{r^1\} \leq \{r^+, r^+\}$), and then free r when f calls the continuation in its final argument. Unfortunately, the continuation only possesses the capability $\{r^+, r^+\} = \{r^+\}$, not the capability $\{r^1\}$ necessary to free r. It does not help to strengthen the capability of the continuation to (for example) $\{\rho_1^1\}$, because then f may not call it.

We *may* recover uniqueness information by quantifying a capability variable. Suppose we again hold capability $\{r^1\}$ and g has type:

$$\forall[\rho_1{:}\texttt{Rgn}, \rho_2{:}\texttt{Rgn}, \epsilon{:}\texttt{Cap}].(\epsilon, \ldots, (\epsilon, \ldots) \to \texttt{0 at } \rho_1) \to \texttt{0 at } r$$

We may instantiate ϵ with $\{r^1\}$ and then the continuation will possess that same capability, allowing it to free r. Unfortunately, the body of function g no longer has the capability to access ρ_1 and ρ_2, since its type draws no connection between them and ϵ.

We solve this problem by using bounded quantification to relate ρ_1, ρ_2 and ϵ. Suppose h has type:

$$\forall[\rho_1{:}\texttt{Rgn}, \rho_2{:}\texttt{Rgn}, \epsilon \leq \{\rho_1^+, \rho_2^+\}].(\epsilon, \ldots, (\epsilon, \ldots) \to \texttt{0 at } \rho_1) \to \texttt{0 at } r$$

If we hold capability $\{r^1\}$, we may call h by instantiating ρ_1 and ρ_2 with r and instantiating ϵ with $\{r^1\}$. This instantiation is permissible because $\{r^1\} \leq \{r^+, r^+\}$. As with g, the continuation will possess the capability $\{r^1\}$, allowing it to free r, but the body of h (like that of f) will have the capability to access ρ_1 and ρ_2, since $\epsilon \leq \{\rho_1^+, \rho_2^+\}$.

Bounded quantification solves the problem by revealing some information about a capability ϵ, while still requiring the function to be parametric over ϵ. Hence, when the function calls its continuation we regain the stronger capability (to free r), although that capability was temporarily hidden in order to duplicate r. More generally, bounded quantification allows us to hide some privileges when calling a function, and regain those privileges in its continuation. Thus, we support statically checkable attenuation and amplification of capabilities.

Static Semantics So Far The combination of parametric polymorphism, bounded parametric polymorphism, and notions of uniqueness and aliasing provide a flexible language for expressing the lifetimes

of regions. Figures 7 and 6 formally summarize the rules for type checking instructions and values that depend upon these concepts. We have already explained the majority of these rules in previous sections and the rules that we have not yet specified are the obvious ones (integers are given type `int`, etc.). Notice, however, that the form of the judgement for heap values h is slightly different from the judgements for instructions and small values v. The judgment $\Psi; \Delta; \Gamma \vdash h\,\mathtt{at}\,r : \tau$ states that when memory has type Ψ, free constructor variables have kinds given by Δ and free value variables have types given by Γ, the heap value h resides in region r and has type τ.

2.4 The Static Semantics of the Abstract Machine

We have described the type constructor language of the Capability Calculus and the typing rules for the main term-level constructs. In fact, the previous section contains all of the information programmers or compilers require to write type-safe programs in the Capability Calculus. However, in order to prove a type soundness result in the style of Wright and Felleisen [54], we must be able to type check programs at every step during their evaluation. In this section, we give the static semantics of the run-time values that are not normally manipulated by programmers, but are nevertheless necessary to prove our soundness result.

At first, the formal definition of the semantics may appear quite complex because we use a number of different judgment forms. However, most of these forms follow naturally from the development of previous sections and other work on type systems for allocation semantics [30, 31, 33]. The extra complexity in the definition of the language will pay off when we come to prove type soundness: All of the main invariants are expressed directly in the typing rules and therefore most of the proof follows from straightforward inductions over these rules.

Figure 8 specifies the rules for typing memory, most of which are straightforward. The judgments $\vdash \Psi$ and $\vdash \Upsilon$ specify when memory types and region types are well-formed. This is the case whenever their subcomponents are well-formed. The judgment $\vdash M : \Psi$ states that memory M is described by Ψ and the judgement $\Psi \vdash R\,\mathtt{at}\,\nu : \Upsilon$ states that region R with name ν is described by Υ. Informally, these judgements ensure that for addresses $\nu.\ell$, $\Psi(\nu.\ell)$ is type τ if and only if the memory M described by Ψ contains a value v at address $\nu.\ell$ that has type τ.

The next judgment, $\Psi \vdash C$ sat is called the satisfiability judgment and it formalizes the connection between the static capability and the run-time state of memory. Clearly, the current capability must not contain any regions that are not in memory; this could lead to a runtime error. However, it is equally important that memory not contain regions for which we have no capability; these regions can never be freed. Consequently, satisfiability ensures that at any time during execution of the abstract machine, our capability contains *exactly* the regions in memory. Furthermore, by virtue of the fact that $\cdot \vdash \{r^1\} \neq \{r^1, r^1\} : \mathtt{Cap}$, no unique regions may appear more than once in C. Each of these properties are essential to ensure that regions are used safely.

Figure 9 contains rules for small values that only appear at run time (addresses and region handles). The rules for typing an address $\nu.\ell$ are quite unusual, but crucial to the type soundness proof. The first rule, v-addr, is used during the lifetime of the region ν: If the region ν is in memory then ν will also be in the domain of the memory type Ψ. Therefore, rule v-addr applies and $\nu.\ell$ will have type $\Psi(\nu.\ell)$. Now consider some point in the computation after the region ν has been deallocated. The region ν is no longer in the memory, but the addresses $\nu.\ell$ may still appear embedded in tuples or closures allocated in other regions, and, therefore, it must be given a type. If a region ν does not appear in memory type

$\boxed{\Psi;\Delta;\Gamma \vdash h \,\mathtt{at}\, r : \tau}$

$$\frac{\begin{array}{c}\Delta \vdash \tau_f \\ \Psi;\Delta\Delta';\Gamma\{f{:}\tau_f, x_1{:}\tau_1, \ldots, x_n{:}\tau_n\};C \vdash e\end{array}}{\Psi;\Delta;\Gamma \vdash \mathtt{fix}\, f[\Delta'](C, x_1{:}\tau_1, \ldots, x_n{:}\tau_n).e \,\mathtt{at}\, r : \tau_f} \begin{pmatrix}\tau_f = \forall[\Delta'].(C, \tau_1, \ldots, \tau_n) \to \mathtt{0}\,\mathtt{at}\, r \\ f, x_1, \ldots, x_n \notin \mathit{Dom}(\Gamma)\end{pmatrix} \text{ (h-fix)}$$

$$\frac{\Psi;\Delta;\Gamma \vdash v_i : \tau_i \quad (\text{for } 1 \le i \le n) \quad \Delta \vdash r : \mathtt{Rgn}}{\Psi;\Delta;\Gamma \vdash \langle v_1, \ldots, v_n\rangle \,\mathtt{at}\, r : \langle \tau_1, \ldots, \tau_n\rangle \,\mathtt{at}\, r} \text{ (h-tuple)}$$

$$\frac{\Psi;\Delta;\Gamma \vdash h \,\mathtt{at}\, r : \tau' \qquad \Delta \vdash \tau' = \tau : \mathtt{Type}}{\Psi;\Delta;\Gamma \vdash h \,\mathtt{at}\, r : \tau} \text{ (h-eq)}$$

$\boxed{\Psi;\Delta;\Gamma \vdash v : \tau}$

$$\frac{}{\Psi;\Delta;\Gamma \vdash x : \tau} \; (\Gamma(x) = \tau) \text{ (v-var)}$$

$$\frac{}{\Psi;\Delta;\Gamma \vdash i : \mathtt{int}} \text{ (v-int)}$$

$$\frac{\Psi;\Delta;\Gamma \vdash v : \forall[\alpha{:}\kappa, \Delta'].(C, \tau_1, \ldots, \tau_n) \to \mathtt{0}\,\mathtt{at}\, r \qquad \Delta \vdash c : \kappa}{\Psi;\Delta;\Gamma \vdash v[c] : (\forall[\Delta'].(C, \tau_1, \ldots, \tau_n) \to \mathtt{0})[c/\alpha]\,\mathtt{at}\, r} \text{ (v-type)}$$

$$\frac{\Psi;\Delta;\Gamma \vdash v : \forall[\epsilon \le C'', \Delta'].(C', \tau_1, \ldots, \tau_n) \to \mathtt{0}\,\mathtt{at}\, r \qquad \Delta \vdash C \le C''}{\Psi;\Delta;\Gamma \vdash v[C] : (\forall[\Delta'].(C', \tau_1, \ldots, \tau_n) \to \mathtt{0})[C/\epsilon]\,\mathtt{at}\, r} \text{ (v-sub)}$$

$$\frac{\Psi;\Delta;\Gamma \vdash v : \tau' \quad \Delta \vdash \tau' = \tau : \mathtt{Type}}{\Psi;\Delta;\Gamma \vdash v : \tau} \text{ (v-eq)}$$

Figure 6: Capability Static Semantics: Heap and Word Values

$\boxed{\Psi;\Delta;\Gamma;C \vdash d \Rightarrow \Delta';\Gamma';C'}$

$$\frac{\Psi;\Delta;\Gamma \vdash v : \tau}{\Psi;\Delta;\Gamma;C \vdash x = v \Rightarrow \Delta;\Gamma\{x{:}\tau\};C}\ (x \notin Dom(\Gamma))\ \text{(val)}$$

$$\frac{\Psi;\Delta;\Gamma \vdash v_1 : \mathtt{int} \quad \Psi;\Delta;\Gamma \vdash v_2 : \mathtt{int}}{\Psi;\Delta;\Gamma;C \vdash x = v_1\ p\ v_2 \Rightarrow \Delta;\Gamma\{x{:}\mathtt{int}\};C}\ (x \notin Dom(\Gamma))\ \text{(prim)}$$

$$\frac{\Psi;\Delta;\Gamma \vdash v : r\ \mathtt{handle} \qquad \Psi;\Delta;\Gamma \vdash h\ \mathtt{at}\ r : \tau \qquad \Delta \vdash C \le C' \oplus \{r^+\}}{\Psi;\Delta;\Gamma;C \vdash x = h\ \mathtt{at}\ v \Rightarrow \Delta;\Gamma\{x{:}\tau\};C}\ (x \notin Dom(\Gamma))\ \text{(hval)}$$

$$\frac{\Psi;\Delta;\Gamma \vdash v : \langle \tau_0, \ldots, \tau_{n-1}\rangle\ \mathtt{at}\ r \qquad \Delta \vdash C \le C' \oplus \{r^+\}}{\Psi;\Delta;\Gamma;C \vdash x = \pi_i v \Rightarrow \Delta;\Gamma\{x{:}\tau_i\};C}\ (x \notin Dom(\Gamma) \wedge 0 \le i < n)\ \text{(proj)}$$

$$\frac{}{\Psi;\Delta;\Gamma;C \vdash \mathtt{newrgn}\,\rho, x \Rightarrow \Delta\{\rho{:}\mathtt{Rgn}\};\Gamma\{x{:}\rho\ \mathtt{handle}\};C \oplus \{\rho^1\}}\ \begin{pmatrix}\rho \notin Dom(\Delta)\\ x \notin Dom(\Gamma)\end{pmatrix}\ \text{(newrgn)}$$

$$\frac{\Psi;\Delta;\Gamma \vdash v : r\ \mathtt{handle} \quad \Delta \vdash C = C' \oplus \{r^1\} : \mathtt{Cap}}{\Psi;\Delta;\Gamma;C \vdash \mathtt{freergn}\,v \Rightarrow \Delta;\Gamma;C'}\ \text{(freergn)}$$

$\boxed{\Psi;\Delta;\Gamma;C \vdash e}$

$$\frac{\Psi;\Delta;\Gamma;C \vdash d \Rightarrow \Delta';\Gamma';C' \qquad \Psi;\Delta';\Gamma';C' \vdash e}{\Psi;\Delta;\Gamma;C \vdash \mathtt{let}\,d\,\mathtt{in}\,e}\ \text{(letdec)} \qquad \frac{\begin{array}{c}\Psi;\Delta;\Gamma \vdash v : \mathtt{int}\\ \Psi;\Delta;\Gamma;C \vdash e_2 \qquad \Psi;\Delta;\Gamma;C \vdash e_3\end{array}}{\Psi;\Delta;\Gamma;C \vdash \mathtt{if0}\,v\,\mathtt{then}\,e_2\,\mathtt{else}\,e_3}\ \text{(if)}$$

$$\frac{\begin{array}{c}\Psi;\Delta;\Gamma \vdash v : \forall[\,].(C', \tau_1, \ldots, \tau_n) \to 0\ \mathtt{at}\ r\\ \Psi;\Delta;\Gamma \vdash v_i : \tau_i \quad (\text{for } 1 \le i \le n)\\ \Delta \vdash C \le C'' \oplus \{r^+\} \qquad \Delta \vdash C \le C'\end{array}}{\Psi;\Delta;\Gamma;C \vdash v(v_1, \ldots, v_n)}\ \text{(app)} \qquad \frac{\Psi;\Delta;\Gamma \vdash v : \mathtt{int} \quad \Delta \vdash C = \emptyset : \mathtt{Cap}}{\Psi;\Delta;\Gamma;C \vdash \mathtt{halt}\ v}\ \text{(halt)}$$

Figure 7: Capability Static Semantics: Declarations and Expressions

$\boxed{\vdash \Upsilon \qquad \vdash \Psi}$

$$\frac{\cdot \vdash \tau_i \qquad (\text{for } 1 \leq i \leq n)}{\vdash \{\ell_1{:}\tau_1, \ldots, \ell_n{:}\tau_n\}} \text{ (region-type)}$$

$$\frac{\vdash \Upsilon_i \qquad (\text{for } i \leq i \leq n)}{\vdash \{\nu_1{:}\Upsilon_1, \ldots, \nu_n{:}\Upsilon_n\}} \text{ (memory-type)}$$

$\boxed{\Psi \vdash R \,\mathtt{at}\, \nu : \Upsilon \qquad \vdash M : \Psi}$

$$\frac{\Psi; \cdot; \cdot \vdash h_i \,\mathtt{at}\, \nu : \tau_i \quad (\text{for } 1 \leq i \leq n)}{\Psi \vdash \{\ell_1 \mapsto h_1, \ldots, \ell_n \mapsto h_n\} \,\mathtt{at}\, \nu : \{\ell_1 : \tau_1, \ldots, \ell_n : \tau_n\}} \text{ (region)}$$

$$\frac{\vdash \Psi \quad \Psi \vdash R_i \,\mathtt{at}\, \nu_i : \Upsilon_i \quad (\text{for } 1 \leq i \leq n)}{\vdash \{\nu_1 \rightarrowtail R_1, \ldots, \nu_n \rightarrowtail R_n\} : \Psi} (\Psi = \{\nu_1 : \Upsilon_1, \ldots, \nu_n : \Upsilon_n\}) \text{ (memory)}$$

$\boxed{\Psi \vdash C \text{ sat}}$

$$\frac{\cdot \vdash C = \{\nu_1^{\varphi_1}, \ldots, \nu_n^{\varphi_n}\} : \mathtt{Cap}}{\{\nu_1 : \Upsilon_1, \ldots, \nu_n : \Upsilon_n\} \vdash C \text{ sat}} \ (\nu_i \neq \nu_j \text{ for } 1 \leq i, j \leq n \text{ and } i \neq j) \text{ (sat)}$$

Figure 8: Capability Static Semantics: Memory

Ψ, the type system has the flexibility to give $\nu.\ell$ *any* function type (by rule v-addr-arrow) or tuple type (by rule v-addr-tuple).

At first glance, these rules would appear to lead to unsoundness: The address $\nu.\ell$ is a dangling pointer and it may be given a valid type. Fortunately, though, capabilities prevent anything from going wrong. The satisfiability judgment ensures that programs only ever possess capabilities for regions that appear in memory, and, as we explained earlier, programs can only access the regions they have capabilities for. Consequently, a dangling pointer may be given a valid tuple or function type, but capabilities prevent it from being accessed.

We now have all components necessary to define a well-formed machine state. The state (M, e) is well-formed if the memory M can be described by a well-formed heap type Ψ, there exists a capability C such that C *satisfies* the heap type Ψ, and finally, the expression e is well-formed with respect to Ψ and C:

$$\frac{\vdash M : \Psi \qquad \Psi \vdash C \text{ sat} \qquad \Psi; \cdot; \cdot; C \vdash e}{\vdash (M, e)} \text{ (program)}$$

$\boxed{\Psi;\Delta;\Gamma \vdash v : \tau}$

$$\frac{}{\Psi;\Delta;\Gamma \vdash \nu.\ell : \tau}\ (\Psi(\nu.\ell) = \tau)\ \text{(v-addr)}$$

$$\frac{\Delta \vdash \langle \tau_1, \ldots, \tau_n \rangle\ \texttt{at}\ \nu : \texttt{Type}}{\Psi;\Delta;\Gamma \vdash \nu.\ell : \langle \tau_1, \ldots, \tau_n \rangle\ \texttt{at}\ \nu}\ (\nu \notin \mathit{Dom}(\Psi))\ \text{(v-addr-tuple)}$$

$$\frac{\Delta \vdash \forall[\Delta'].(C, \tau_1, \ldots, \tau_n) \rightarrow 0\ \texttt{at}\ \nu : \texttt{Type}}{\Psi;\Delta;\Gamma \vdash \nu.\ell : \forall[\Delta'].(C, \tau_1, \ldots, \tau_n) \rightarrow 0\ \texttt{at}\ \nu}\ (\nu \notin \mathit{Dom}(\Psi))\ \text{(v-addr-arrow)}$$

$$\frac{}{\Psi;\Delta;\Gamma \vdash \texttt{handle}(\nu) : \nu\ \texttt{handle}}\ \text{(v-handle)}$$

Figure 9: Capability Static Semantics: Run-time Values

2.5 Formal Properties of the Calculus

The most important property of the Capability Calculus is *Type Soundness*. Type Soundness states that a program will never enter a *stuck state* during execution. A state (M, e) is *stuck* if there does not exist (M', e') such that $(M, e) \longmapsto (M', e')$ and e is not $\texttt{halt}\ i$. For example, a state that tries to project a value from a tuple that does not appear in memory is stuck.

Theorem 1 (Type Soundness)
If $\vdash (M, e)$ *and* $(M, e) \longmapsto^* (M', e')$ *then* (M', e') *is not stuck.*

In the previous sections of this article, we have explained how to type memory, how to relate the memory typing to static capabilities and finally, given a collection of capabilities, how the rules for typing expressions prevent unsafe accesses to the store. These invariants are the main elements in the formal proof of soundness. However, there are many details to fill in. The proof is in the style of Wright and Felleisen [54] and uses the standard Type Preservation and Progress lemmas. Progress states that well-typed states are not stuck, and Preservation states that evaluation steps preserve well-typedness.

Lemma 2 (Type Preservation)
If $\vdash (M, e)$ *and* $(M, e) \longmapsto (M', e')$ *then* $\vdash (M', e')$

Lemma 3 (Progress) *If* $\vdash (M, e)$ *then either:*

1. *There exists* (M', e') *such that* $(M, e) \longmapsto (M', e')$*, or*
2. $e = \texttt{halt}\ i$

kinds	κ	::=	$\mathtt{Type} \mid \mathtt{Rgn} \mid \mathtt{Eff}$
constructor variables	$\alpha, \beta, \rho, \epsilon$		
constructor contexts	Δ	::=	$\cdot \mid \Delta, \alpha{:}\kappa_1$
type schemes	σ	::=	$\tau \mid \forall[\Delta].\tau_1 \xrightarrow{\psi} \tau_2 \,\mathtt{at}\, r$
constructors	c, τ, r, ψ	::=	$\alpha \mid \mathtt{int} \mid r\ \mathtt{handle} \mid \langle \tau_1, \ldots, \tau_n \rangle \,\mathtt{at}\, r \mid \tau_1 \xrightarrow{\psi} \tau_2 \,\mathtt{at}\, r \mid$ $\emptyset \mid \{r\} \mid \psi_1 \cup \psi_2$
term variables	x, f		
term contexts	Γ	::=	$\cdot \mid \Gamma, x{:}\sigma$
terms	e	::=	$x \mid i \mid e_1\ p\ e_2 \mid \mathtt{if0}\, e_1\, \mathtt{then}\, e_2\, \mathtt{else}\, e_3 \mid \langle e_1, \ldots, e_n \rangle \,\mathtt{at}\, e_{n+1} \mid$ $\pi_i e \mid \mathtt{letrec} f[\Delta](x) : \sigma \,\mathtt{at}\, e_1 = e_2 \,\mathtt{in}\, e_3 \mid f[c_1, \ldots, c_n] \mid$ $e_1 e_2 \mid \mathtt{letregion}\, \rho, x_\rho \,\mathtt{in}\, e$

Figure 10: Region Syntax

Because of the length and tedium of the proofs of these lemmas, we have removed them, along with the proof of soundness itself, to Appendix A.

The second important property of the language is that well-typed terminating programs return all of their memory resources to the system before they halt. We call this property *Complete Collection*.

Theorem 4 (Complete Collection) *If* $\vdash (M, e)$ *then either* (M, e) *diverges or* $(M, e) \longmapsto^* (\{\}, \mathtt{halt}\ i)$.

By Subject Reduction and Progress, terminating programs end in well-formed machine states $(M, \mathtt{halt}\ i)$. The typing rule for the `halt` expression requires that the capability C be empty. Using this fact, we can infer that the memory M contains no regions. Appendix A also contains a formal proof of this theorem.

3 From Regions to Capabilities

The primary goal of this work is the development of a type-safe language that gives compilers control over the allocation and deallocation of data. In order to use this technology effectively, we need compiler support for generating the intermediate language code from source language programs. In this section, we define a high-level explicitly-typed variant of Tofte and Talpin's region-based language and show that it can be translated into the Capability Calculus. By composing this translation with region inference, we may obtain a type-preserving compiler front-end.

3.1 A Region-Based Calculus

Preliminaries The source language for our compiler is the region-based calculus shown in Figure 10. This language is an explicitly-typed variant of the calculus first presented by Tofte and Talpin [47]. Like the Capability Calculus, it contains integers, tuples and functions. However, unlike the Capability

Calculus, allocation and deallocation of regions is combined in a single construct: $\texttt{letregion}\,\rho, x_\rho\,\texttt{in}\,e$. This construct allocates a new region ρ and places the handle for that region in the term variable x_ρ. Next, it executes the expression e. Finally, the region ρ is deallocated. As discussed in the introduction, this lexically-scoped construct is not as flexible as the separate `newrgn` and `freergn` constructs provided by the Capability Calculus. The main goal of this section is to show how to compile `letregion` expressions into these lower-level primitives.

As in the original Tofte-Talpin calculus, the region language has prenex predicative polymorphism. The term $\texttt{letrec}f[\Delta](x) : \sigma\,\texttt{at}\,e_1 = e_2\,\texttt{in}\,e_3$ allocates a closure f of polytype σ. The closure is polymorphic over its type context Δ, which may contain ordinary type variables as well as region variables and effect variables (explained below). The closure is allocated in the region r if the expression e_1 evaluates to a region handle for r. The expression e_2 describes the body of the function.

Unlike previous work on region-based type systems, we treat all type constructors, including region constructors, as compile-time-only objects. Therefore, the term $f[c_1, \ldots, c_n]$, which denotes type application, has no runtime effect. During type checking, the type scheme for the polymorphic function f is instantiated with the types $c_1, \ldots, c_n$ to obtain the resultant type for the expression, but the dynamic semantics of the program (not shown here) do not depend upon these types. Hence, the types may be erased before the program is run without affecting the computation. As in the Capability Calculus, the data structures that are required to allocate and deallocate regions are treated as ordinarly values of type $\texttt{handle}(r)$.

Types and Effects The main interest of the type constructor portion of the region language is the presence of *effects*. Effects, like capabilities, are used to control a program's access to regions and, in particular, to prevent access to regions that have been deallocated. Intuitively, the effect of a term is the set of regions that the term accesses. Formally, an effect is either the empty effect ($\emptyset$), an effect variable (ϵ), a singleton set ($\{r\}$), or the union of two effects ($\psi_1 \cup \psi_2$). The $\cup$ operator is associative, commutative, and idempotent and $\emptyset$ is the unit for the union operator. We write $\Delta \vdash_R \psi_1 = \psi_2 : \texttt{Eff}$ for equality on effects and we use the abbreviation $\Delta \vdash_R \psi_1 \subseteq \psi_2$ when $\Delta \vdash_R \psi_1 \cup \psi_3 = \psi_2 : \texttt{Eff}$ for some effect ψ_3.

All functions have *latent effects* that are incurred when the function is called and its body is executed. The effect that appears on arrow types $(\forall[\Delta].\tau \xrightarrow{\psi} \tau')$[2] specifies the set of regions that a function of that type may access when it is invoked. The formal rules for type and effect formation and equality may be found in Figure 11.

Static Semantics The static semantics (Figure 12) for terms use a judgement of the form $\Delta; \Gamma \vdash_R e : \tau, \psi$ to track the effects produced by each expression. This judgement states that under the type context Δ and the value context Γ, a term e has type τ and produces effect ψ. For example, the rule for projection states that if e is an expression producing effect ψ then $\pi_i e$ produces the effect $\psi \cup \{r\}$. The projection operation reads from the region r and the subexpression e may read from or write to any of the regions in ψ. Hence the resulting effect must be the union of the two.

[2]Tofte and Talpin require each arrow type be annotated with an "arrow effect", which is constrained to have the form $\epsilon \cup \psi$. The type variable ϵ is used to name the effect and plays a role in their inference system. Because we are interested in type checking rather than type inference, we do not need to name the effects on arrows.

$\boxed{\Delta \vdash \Delta'}$

$$\frac{}{\Delta \vdash \cdot} \qquad \frac{\Delta \vdash \Delta'}{\Delta \vdash \Delta', \{\alpha{:}\kappa\}}\ (\alpha \notin Dom(\Delta))$$

$\boxed{\Delta \vdash \sigma}$

$$\frac{\Delta \vdash \tau : \mathtt{Type}}{\Delta \vdash \tau} \qquad \frac{\begin{array}{c}\vdash \Delta\Delta' \qquad \Delta\Delta' \vdash \tau_i : \mathtt{Type} \quad (\text{for } 1 \le i \le n+1) \\ \Delta\Delta' \vdash \psi : \mathtt{Eff} \qquad \Delta \vdash r : \mathtt{Rgn}\end{array}}{\Delta \vdash \forall[\Delta'].(\tau_1, \ldots, \tau_n) \xrightarrow{\psi} \tau_{n+1}\ \mathtt{at}\ r}$$

$\boxed{\Delta \vdash c : \kappa}$

$$\frac{}{\Delta \vdash \alpha : \kappa}\ (\Delta(\alpha) = \kappa) \qquad \frac{}{\Delta \vdash \mathtt{int} : \mathtt{Type}} \qquad \frac{\Delta \vdash r : \mathtt{Rgn}}{\Delta \vdash r\ \mathtt{handle} : \mathtt{Type}}$$

$$\frac{\Delta \vdash \tau_i : \mathtt{Type} \quad (\text{for } 1 \le i \le n) \quad \Delta \vdash r : \mathtt{Rgn}}{\Delta \vdash \langle \tau_1, \ldots, \tau_n \rangle\ \mathtt{at}\ r : \mathtt{Type}} \qquad \frac{\begin{array}{c}\Delta \vdash r : \mathtt{Rgn} \qquad \Delta \vdash \psi : \mathtt{Eff} \\ \Delta \vdash \tau_i : \mathtt{Type} \quad (\text{for } 1 \le i \le n+1)\end{array}}{\Delta \vdash (\tau_1, \ldots, \tau_n) \xrightarrow{\psi} \tau_{n+1}\ \mathtt{at}\ r : \mathtt{Type}}$$

$$\frac{}{\Delta \vdash \emptyset : \mathtt{Eff}} \qquad \frac{\Delta \vdash r : \mathtt{Rgn}}{\Delta \vdash \{r\} : \mathtt{Eff}} \qquad \frac{\Delta \vdash \psi_1 : \mathtt{Eff} \quad \Delta \vdash \psi_2 : \mathtt{Eff}}{\Delta \vdash \psi_1 \cup \psi_2 : \mathtt{Eff}}$$

$\boxed{\Delta \vdash \sigma_1 = \sigma_2 \quad \Delta \vdash c_1 = c_2 : \kappa \text{ (congruence rules omitted)}}$

$$\frac{\Delta \vdash c : \kappa}{\Delta \vdash c = c : \kappa} \qquad \frac{\Delta \vdash c_2 = c_1 : \kappa}{\Delta \vdash c_1 = c_2 : \kappa} \qquad \frac{\Delta \vdash c_1 = c_2 : \kappa \quad \Delta \vdash c_2 = c_3 : \kappa}{\Delta \vdash c_1 = c_3 : \kappa}$$

$$\frac{\Delta \vdash \psi : \mathtt{Eff}}{\Delta \vdash \emptyset \cup \psi = \psi : \mathtt{Eff}} \qquad \frac{\Delta \vdash \psi_1 : \mathtt{Eff} \quad \Delta \vdash \psi_2 : \mathtt{Eff}}{\Delta \vdash \psi_1 \cup \psi_2 = \psi_2 \cup \psi_1 : \mathtt{Eff}}$$

$$\frac{\Delta \vdash \psi_i : \mathtt{Eff} \quad (\text{for } 1 \le i \le 3)}{\Delta \vdash (\psi_1 \cup \psi_2) \cup \psi_3 = \psi_1 \cup (\psi_2 \cup \psi_3) : \mathtt{Eff}} \qquad \frac{\Delta \vdash \psi_1 : \mathtt{Eff} \quad \Delta \vdash \psi_2 : \mathtt{Eff}}{\Delta \vdash \psi_1 \cup \psi_2 = \psi_1 \cup (\psi_2 \cup \psi_2) : \mathtt{Eff}}$$

$$\frac{\Delta \vdash \psi_1 \cup \psi_3 = \psi_2 : \mathtt{Eff}}{\Delta \vdash \psi_1 \subseteq \psi_2}$$

Figure 11: Region Type Formation, Equality, and Subset

$$\frac{}{\Delta;\Gamma \vdash_R x : \tau, \emptyset}\ (\Gamma(x) = \tau) \qquad \frac{}{\Delta;\Gamma \vdash_R i : \mathtt{int}, \emptyset}$$

$$\frac{\Delta;\Gamma \vdash_R e_1 : \mathtt{int}, \psi_1 \quad \Delta;\Gamma \vdash_R e_2 : \mathtt{int}, \psi_2}{\Delta;\Gamma \vdash_R e_1\ p\ e_2 : \mathtt{int}, \psi_1 \cup \psi_2}$$

$$\frac{\Delta;\Gamma \vdash_R e_i : \tau_i, \psi_i \quad (\text{for } 1 \le i \le n) \quad \Delta;\Gamma \vdash_R e_{n+1} : r\ \mathtt{handle}, \psi_{n+1}}{\Delta;\Gamma \vdash_R \langle e_1, \ldots, e_n \rangle\ \mathtt{at}\ e_{n+1} : \tau, \psi_1 \cup \ldots \cup \psi_{n+1} \cup \{r\}}$$

$$\frac{\Delta;\Gamma \vdash_R e : \langle \tau_0, \ldots, \tau_{n-1} \rangle\ \mathtt{at}\ r, \psi}{\Delta;\Gamma \vdash_R \pi_i e : \tau_i, \psi \cup \{r\}}\ (0 \le i < n)$$

$$\frac{\Delta;\Gamma \vdash_R e_1 : \mathtt{int}, \psi_1 \quad \Delta;\Gamma \vdash_R e_2 : \tau, \psi_2 \quad \Delta;\Gamma \vdash_R e_3 : \tau, \psi_3}{\Delta;\Gamma \vdash_R \mathtt{if0}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3 : \tau, \psi_1 \cup \psi_2 \cup \psi_3}$$

$$\frac{\begin{array}{c}\Delta \vdash_R \sigma \qquad \Delta\Delta';\Gamma\{f{:}\sigma, x{:}\tau_1\} \vdash_R e_2 : \tau_2, \psi \\ \Delta;\Gamma \vdash_R e_1 : r\ \mathtt{handle}, \psi_1 \qquad \Delta;\Gamma\{f{:}\sigma\} \vdash_R e_3 : \tau_3, \psi_3\end{array}}{\Delta;\Gamma \vdash_R \mathtt{letrec} f[\Delta'](x) : \sigma\ \mathtt{at}\ e_1 = e_2\ \mathtt{in}\ e_3 : \tau_3, \psi_1 \cup \psi_3 \cup \{r\}}\ \left(\begin{array}{c} x, f \notin Dom(\Gamma) \\ \sigma = \forall[\Delta'].\tau_1 \xrightarrow{\psi} \tau_2\ \mathtt{at}\ r \end{array}\right)$$

$$\frac{\Delta \vdash_R c_i : \kappa_i \qquad (\text{for } 1 \le i \le n)}{\begin{array}{l}\Delta;\Gamma \vdash_R f[c_1, \ldots, c_n] : \\ \quad (\tau_1 \xrightarrow{\psi} \tau_2)[c_1, \ldots, c_n/\alpha_1, \ldots, \alpha_n]\ \mathtt{at}\ r, \emptyset\end{array}}\ (\Gamma(f) = \forall[\alpha_1{:}\kappa_1, \ldots, \alpha_n{:}\kappa_n].\tau_1 \xrightarrow{\psi} \tau_2\ \mathtt{at}\ r)$$

$$\frac{\Delta;\Gamma \vdash_R e_1 : \tau_1 \xrightarrow{\psi} \tau_2\ \mathtt{at}\ r, \psi_1 \quad \Delta;\Gamma \vdash_R e_2 : \tau_1, \psi_2}{\Delta;\Gamma \vdash_R e_1 e_2 : \tau_2, \psi_1 \cup \psi_2 \cup \psi \cup \{r\}}$$

$$\frac{\Delta\{\rho{:}\mathtt{Rgn}\};\Gamma\{x_\rho{:}\rho\ \mathtt{handle}\} \vdash_R e : \tau, \psi}{\Delta;\Gamma \vdash_R \mathtt{letregion}\,\rho, x_\rho\ \mathtt{in}\ e : \tau, \psi \backslash \{\rho\}}\ \left(\begin{array}{c} \rho \notin ftv(\tau) \cup Dom(\Delta) \\ x_\rho \notin Dom(\Gamma) \end{array}\right)$$

$$\frac{\Delta;\Gamma \vdash_R e : \tau, \psi \quad \Delta \vdash_R \tau = \tau' : \mathtt{Type} \quad \Delta \vdash_R \psi \subseteq \psi'}{\Delta;\Gamma \vdash_R e : \tau', \psi'}$$

Figure 12: Region Term Static Semantics

The rules involving functions are more complex. First, consider a function call $e_1 e_2$. Assume that e_1 generates the effect ψ_1 and evaluates to a closure of type $\forall[].(\tau) \xrightarrow{\psi} \tau'$ `at` r, and that e_2 produces the effect ψ_2 and has type τ. After both expressions have been evaluated, the code for the function is projected from a closure that resides in region r. Now, because the function itself produces the effect ψ, the overall effect of the call is the union of $\{r\}$ with ψ, ψ_1, and ψ_2. In contrast to the value application rule, the rule for type application produces no effect; remember, types are erased before an expression is executed. Finally, examine the rule for the `letrec` term. There are three components to the effect produced by this expression: ψ_1, the effect of evaluating handle expression; $\{r\}$, the effect of writing the closure data structure in region r; and ψ_3, the effect of the subsequent expression e_3. As well as checking that types match up properly, we must be sure that the effect produced by the body of the function is a subset of the declared effect. The type system would be unsound if functions could hide their effects.

The last rule checks the `letregion` construct. Here, we use the notation $\psi \backslash \{\rho\}$ to denote the effect ψ with all occurences of $\{\rho\}$ replaced by $\emptyset$. This rule discharges the effect $\{r\}$ from the effect produced by the subexpression e. Intuitively, because the `letregion` construct discharges effects whereas all other constructs pass on effect information from their subexpressions to their enclosing expressions, any access to a region outside of the scope of a `letregion` will be detected and the type checker will reject the expression as a whole.

Figure 13 shows an example program, a function `count` that counts down to zero. In order to have interesting allocation behavior the integers involved in the count are boxed, and hence are allocated in a region. The `count` function is stored in region ρ_1 and takes two arguments, a handle for region ρ and a boxed integer x allocated in region ρ. If x is nonzero, `count` decrements it, storing the result again in ρ, and recurses. The function has two effects: a read on ρ_1, resulting from the recursive call, and a read/write effect on ρ, resulting from line 1's read and line 2's store. Therefore, we give the function `count` the effect $\{\rho_1, \rho_2\}$. Overall, the code in Figure 13 allocates two regions (ρ_1 and ρ_2), stores the closure for `count` in ρ_1, stores a boxed integer in ρ_2, calls `count`, and then deallocates ρ_1 and ρ_2.

3.2 The Translation

In order to make a formal connection to region-based calculi and to corroborate our claims that we can use the region inference techniques developped by Tofte and others as a front-end for a capability-based compiler, we have defined a type-directed and type-preserving translation from the region calculus of the previous section to the Capability Calculus. Appendix B contains a proof that any well-formed source term is translated into a well-formed target term.

Kind and Type Translation The translation is a continuation-passing style transformation in which we simultaneously transform effects into capabilities. The kind and type transformation is presented in Figure 14. The kind translation is trivial; effects become capabilities and the other kinds are unchanged. The translation of most types is equally simple. The translation of base types is the identity and, in general, to translate other types we recursively translate their components and recombine using the corresponding capability constructor. Thus, tuples are mapped to tuples and handles are mapped to handles, etc.

```
letregion ρ1, x_ρ1 in
  letregion ρ2, x_ρ2 in
    letrec count [ρ] (x_ρ : ρ handle, x : ⟨int⟩ at ρ) at x_ρ1 : σ_count =
      let n = π0(x) in                          % (1)
        if0 n
          then ()
          else count [ρ] (x_ρ, ⟨n − 1⟩ at x_ρ)   % (2)
      end
    in
      count [ρ2] (x_ρ2, ⟨10⟩ at x_ρ2)
    end % letrec
  % end region ρ2 scope and deallocate
% end region ρ1 scope and deallocate

where σ_count = ∀[ρ].(ρ handle, ⟨int⟩ at ρ) --{ρ1,ρ}--> unit
```

Figure 13: Count in the Region Calculus

$$
\begin{array}{rcl}
\mathcal{K}[\![\texttt{Type}]\!] & = & \texttt{Type} \\
\mathcal{K}[\![\texttt{Rgn}]\!] & = & \texttt{Rgn} \\
\mathcal{K}[\![\texttt{Eff}]\!] & = & \texttt{Cap} \\
\mathcal{K}[\![\alpha_1{:}\kappa_1, \ldots, \alpha_n{:}\kappa_n]\!] & = & \alpha_1{:}\mathcal{K}[\![\kappa_1]\!], \ldots, \alpha_n{:}\mathcal{K}[\![\kappa_n]\!] \\
 & & \\
\mathcal{T}[\![\alpha]\!] & = & \alpha \\
\mathcal{T}[\![\texttt{int}]\!] & = & \texttt{int} \\
\mathcal{T}[\![\langle \tau_1, \ldots, \tau_n \rangle \texttt{ at } r]\!] & = & \langle \mathcal{T}[\![\tau_1]\!], \ldots, \mathcal{T}[\![\tau_n]\!] \rangle \texttt{ at } \mathcal{T}[\![r]\!] \\
\mathcal{T}[\![\tau_1 \xrightarrow{\psi} \tau_2 \texttt{ at } r]\!] & = & \forall[\rho'{:}\texttt{Rgn}, \epsilon{:}\texttt{Cap}, \epsilon' \leq \overline{\epsilon \oplus \mathcal{T}[\![\psi]\!] \oplus \{\rho'^1\}}].(\epsilon', \\
 & & \quad \mathcal{T}[\![\tau_1]\!], \forall[\,].(\epsilon', \mathcal{T}[\![\tau_2]\!]) \to 0 \texttt{ at } \rho') \to 0 \texttt{ at } \mathcal{T}[\![r]\!] \\
\mathcal{T}[\![r \texttt{ handle}]\!] & = & \mathcal{T}[\![r]\!] \texttt{ handle} \\
\mathcal{T}[\![\emptyset]\!] & = & \emptyset \\
\mathcal{T}[\![\{r\}]\!] & = & \{\mathcal{T}[\![r]\!]^1\} \\
\mathcal{T}[\![\psi_1 \cup \psi_2]\!] & = & \mathcal{T}[\![\psi_1]\!] \oplus \mathcal{T}[\![\psi_2]\!] \\
 & & \\
\mathcal{S}[\![\tau]\!] & = & \mathcal{T}[\![\tau]\!] \\
\mathcal{S}[\![\forall[\Delta].\tau_1 \xrightarrow{\psi} \tau_2 \texttt{ at } r]\!] & = & \forall[\mathcal{K}[\![\Delta]\!]].\mathcal{T}[\![\tau_1 \xrightarrow{\psi} \tau_2 \texttt{ at } r]\!] \\
 & & \\
\mathcal{S}[\![\{x_1{:}\sigma_1, \ldots, x_n{:}\sigma_n\}]\!] & = & \{x_1{:}\mathcal{S}[\![\sigma_1]\!], \ldots, x_n{:}\mathcal{S}[\![\sigma_n]\!]\}
\end{array}
$$

Figure 14: Region to Capability Kind and Type Translation

The translation of function types is more involved. Recall that in the usual CPS-translation, an arrow type $(\tau_1) \to \tau_2$ is transformed so that it accepts a translated τ_1 and a τ_2 continuation:

$$(\mathcal{T}[\![\tau_1]\!], \mathcal{T}[\![\tau_2]\!] \to 0) \to 0$$

The translation of region arrow types has the same structure, but there are several complications that arise as we transform effects into capabilities. The first complication involves finding a region for the continuation closure. We solve this problem by allocating a new region ρ' to hold the continuation; the translated function abstracts this region. The second complication is that an effect for a function may only mention a subset of the regions that are live at the call site. Nevertheless, the resulting Capability Calculus function must thread the capability describing all the live regions from the context where the function is called through the body of the function to the continuation. We accomplish this task by abstracting an additional capability variable ϵ that makes each function context sensitive. Using this mechanism, we can thread any capability in the calling context through the function to its continuation.

The third complication is that the type translation must ensure that equal types in the region calculus are translated to equal types in the Capability Calculus. For the most part, this obligation is satisfied trivially because the equality relation for most region types and their corresponding Capability Calculus analogues is simple syntactic equality up to α-conversion of bound variables. However, the equality relation for effects is set equality whereas the equality relation for *arbitrary* capabilities is not set equality ($\oplus$ is not necessarily idempotent). Fortunately, equality of duplicatable capabilities is exactly set equality. Therefore, the type translation carefully translates all arrow effects into duplicatable capabilities.

These three insights naturally lead us to translate a region function type $\tau_1 \xrightarrow{\psi} \tau_2 \texttt{ at } r$ into the Capability Calculus function type

$$\forall[\rho'{:}\texttt{Rgn}, \epsilon{:}\texttt{Cap}, \epsilon' \leq \overline{\epsilon \oplus \mathcal{T}[\![\psi]\!] \oplus \{\rho'^1\}}].(\epsilon', \mathcal{T}[\![\tau_1]\!], \tau_2 \text{ cont at } \rho') \to 0 \texttt{ at } \mathcal{T}[\![r]\!]$$

The capability for the translated function is ϵ' where ϵ' is a subtype of $\overline{\epsilon \oplus \mathcal{T}[\![\psi]\!] \oplus \{\rho'^1\}}$. This capability gives the translated function access to all the regions it requires: The regions in $\mathcal{T}[\![\psi]\!]$ are the regions accessed by the source language function; ρ' is the region containing the continuation; and ϵ contains the regions from the calling context that are threaded through the call to the continuation. Notice also that the capability that appears in this type is duplicatable, so we can prove that equal types are translated to equal types. Finally, as explained in Section 2.3, if we give our τ_2-continuation the correct type, namely

$$\forall[\,].(\epsilon', \mathcal{T}[\![\tau_2]\!]) \to 0 \texttt{ at } \rho'$$

then bounded quantification allows the continuation to recover the uniqueness information necessary to deallocate the regions used in the function.

Given these definitions, it is straightforward to prove that the essential properties of types (well-formedness, equality, and substitution) are preserved through the translation. Each of the following lemmas can be proven by a simple induction on the well-formedness or equality derivation.

Lemma 5 (Well-Formedness Preservation)

1. *If* $\Delta \vdash_R \Delta'$ *then* $\mathcal{K}[\![\Delta]\!] \vdash \mathcal{K}[\![\Delta']\!]$

2. *If* $\Delta \vdash_R c : \kappa$ *then* $\mathcal{K}[\![\Delta]\!] \vdash \mathcal{T}[\![c]\!] : \mathcal{K}[\![\kappa]\!]$

Lemma 6 (Equality Preservation)

1. *If* $\Delta \vdash_R \psi = \psi' : \texttt{Eff}$ *then* $\mathcal{K}[\![\Delta]\!] \vdash \overline{\mathcal{T}[\![\psi]\!]} = \overline{\mathcal{T}[\![\psi']\!]} : \texttt{Cap}$

2. *If* $\Delta \vdash_R c = c' : \kappa$ *and* κ *is not* `Eff` *then* $\mathcal{K}[\![\Delta]\!] \vdash \mathcal{T}[\![c]\!] = \mathcal{T}[\![c']\!] : \texttt{Cap}$

Lemma 7 (Substitution Preservation) *If* $\Delta, \alpha{:}\kappa \vdash_R \tau : \texttt{Type}$ *and* $\Delta \vdash_R c : \kappa$ *then* $\mathcal{K}[\![\Delta]\!] \vdash \mathcal{T}[\![\tau[c/\alpha]]\!] = \mathcal{T}[\![\tau]\!][\mathcal{T}[\![c]\!]/\alpha]$.

Term Translation The heart of the term translation is a continuation-passing style [12, 39] transformation. There are many variations of this transformation [10, 43, 17, 9], some of which produce more efficient code than others, and some of which lead to simpler correctness proofs. We have chosen a simple translation that is straightforward to prove type preserving so that we may focus on the details relevant to region-based memory management.

We begin with an informal description of the basic mechanics of the CPS term translation, ignoring all of the details relevant to regions or capabilities. There are three main arguments to the translation:

- a type-checking context Φ,
- a source-language term e,
- and a target-language continuation k.

If the source term e is well-formed under the context Φ with type τ, and k is a $\mathcal{T}[\![\tau]\!]$-continuation then the translation $\mathcal{C}_\Phi(e)k$ should produce a well-formed target term.

Operationally, the target term computes e, producing a value v as a result, and then calls the continuation k with v as its argument. Therefore, if the source term e is already a value v, such as an integer or a variable, then the translation is simply the function call $k(v)$. On the other hand, assuming a left-to-right evaluation order, if the source term e actually represents a computation, say the computation $\langle e_1, e_2 \rangle$, the CPS translation arranges to compute e_1 producing value v_1, compute e_2 producing value v_2, allocate the pair $\langle v_1, v_2 \rangle$ and finally pass the resulting pointer to the continuation k. We might write such a translation as follows.

$$\begin{array}{rcl} \mathcal{C}_\Phi(\langle e_1^{\tau_1}, e_2^{\tau_2} \rangle)k & = & \mathcal{C}_\Phi(e_1)(\lambda x_1. \\ & & \mathcal{C}_{\Phi, x_1:\mathcal{T}[\![\tau_1]\!]}(e_2)(\lambda x_2. \\ & & \texttt{let}\ x = \langle x_1, x_2 \rangle\ \texttt{in}\ k(x))) \end{array}$$

The translation of each subcomponent of e requires a continuation and that continuation contains code for all subsequent subcomponents. Finally, the primitive operation *op* is applied to the resulting values and the result is passed to k. The compilation of arithmetic operations and the projections have this form.

There are a couple of further details to notice about the translation. First, we have taken the liberty of annotating expressions with their types where necessary (*e.g.*, $e_1^{\tau_1}$). Second, when the translation

introduces new variables, such as x_1, we add those variables, with their translated types, to the context Φ. The latter decision has no influence on the behaviour of the translation, but it facilitates the statement and proof of the type correctness theorem.

The translation of function application $e_1\, e_2$ begins in the same way as other operations: Translate e_1, passing the resulting value to a continuation that contains the translation of e_2. The continuation for e_2 contains the function application itself. Because user-defined CPS functions (unlike the primitive operations) do not return, the continuation k is passed directly to the translated function.

$$\begin{array}{rcl} \mathcal{C}_\Phi(e_1^{\tau_1\to\tau_2}\, e_2)k & = & \mathcal{C}_\Phi(e_1)(\lambda x_1. \\ & & \mathcal{C}_{\Phi,x_1:\mathcal{T}[\![\tau_1\to\tau_2]\!]}(e_2)(\lambda x_2. \\ & & x_1(x_2,k))) \end{array}$$

Finally, expressions that declare functions must be translated so the result expects an extra continuation argument (x_{cont}) and calls that continuation to return.

$$\begin{array}{rcl} \mathcal{C}_\Phi(\mathtt{let}\, f:\tau_1\to\tau_2 = \lambda x.e\, \mathtt{in}\, e') & = & \mathtt{let}\, f:\tau_f = \lambda(x,x_{cont}).\mathcal{C}_{\Phi,x:\mathcal{T}[\![\tau_1]\!],x_{cont}:\tau_{cont}}(e)x_{cont}\, \mathtt{in} \\ & & \mathcal{C}_{\Phi,f:\mathcal{T}[\![\tau_1\to\tau_2]\!]}(e')k \end{array}$$

Here, the type of the function's continuation, τ_{cont}, is $(\mathcal{T}[\![\tau_2]\!])\to 0$. The type τ_f of the function itself is $(\mathcal{T}[\![\tau_1]\!],\tau_{cont})\to 0$.

This simple CPS translation provides the basic structure for the translation from the region language into the Capability Calculus. However, as many previous researchers have observed, this translation introduces unnecessary or *administrative* redexes. For example, under the scheme we have presented so far, the translation of a simple pair $\langle 2,3\rangle$ with respect to a continuation k is

$$(\lambda x_1.(\lambda x_2.\mathtt{let}\, x_3 = \langle x_1,x_2\rangle\, \mathtt{in}\, k(x_3))\,(3))\,(2)$$

instead of the much simpler term $\mathtt{let}\, x_3 = \langle 2,3\rangle\, \mathtt{in}\, k\,(x_3)$. While we are not concerned with the time required to execute the extra function applications, we are concerned about the space required by additional function closures. If we based our region translation directly on this naive translation, we would be forced to allocate additional regions for each of the λ-expressions above. Previous work has avoided these problems by defining the translation in terms of a two-level type system and passing the translation meta-level continuations instead of target-level continuations. However, because we are only interested in the space properties of our translation, we will use a simpler solution. Rather than allocating continuation closures, we will use a `let` expression to bind the result of a computation and pass it to a continuation. This solution avoids additional allocation and does not lead to the complexities of a two-level type system. Hence, the translation of the pair $\langle 2,3\rangle$ with respect to continuation k will be

$$\begin{array}{l} \mathtt{let}\, x_1 = 2\, \mathtt{in} \\ \mathtt{let}\, x_2 = 3\, \mathtt{in} \\ \mathtt{let}\, x_3 = \langle x_1,x_2\rangle\, \mathtt{in} \\ \mathcal{A}(k,x_3) \end{array}$$

The notation $\mathcal{A}(k,x_3)$ denotes the "application" of the continuation k to the value x_3. The continuation k is not represented as a target-language λ-expression, but, intuitively, this "application" is simply $k(x_3)$.

The continuation k is actually a pair $\langle x^k; e^k \rangle$. The variable x^k is the continuation's parameter and e^k is its body. Given this representation, it is natural to define $\mathcal{A}(\langle x^k; e^k \rangle, v)$ to be $\texttt{let}\, x^k = v\, \texttt{in}\, e^k$.

Using this notation, we can define a CPS translation for pairs in general as follows.

$$\begin{array}{rcl} \mathcal{C}_\Phi(\langle e_1^{\tau_1}, e_2 \rangle)k & = & \mathcal{C}_\Phi(e_1)\langle x_1; \\ & & \mathcal{C}_{\Phi, x_1:\mathcal{T}[\![\tau_1]\!]}(e_2)\langle x_2; \\ & & \texttt{let}\, x = \langle x_1, x_2 \rangle\, \texttt{in}\, \mathcal{A}(k, x) \rangle \rangle \end{array}$$

From Region Expressions to Capability Calculus With the basic CPS transformation in hand, we are ready to investigate the formal details translation of the region language expressions into Capability Calculus expressions. As discussed above, the translation, $\mathcal{C}$, has the following form.

$$\mathcal{C}_\Phi(e)k$$

The context Φ is actually $\Delta; \Gamma; \Theta$. The first two components are a region type context and a region value context. The third component, Θ, is a *translation environment*. This environment contains a Capability Calculus type context Δ^Θ, a Capability Calculus value context Γ^Θ, and a pair of capabilities C^Θ and B^Θ. The context Δ^Θ describes the kinds of the new type variables introduced by the translation and, if they are capability type variables, then possibly their bounds. The value context Γ^Θ describes the types of the new value variables introduced by the translation. Intuitively the capability C^Θ represents the current capability at a given point in the translation and B^Θ is a bound on C^Θ. If Θ is $\langle \Delta^\Theta; \Gamma^\Theta; C^\Theta; B^\Theta \rangle$, then we use the notation $\Theta, x{:}\tau$ to denote the translation environment $\langle \Delta^\Theta; \Gamma^\Theta, x{:}\tau; C^\Theta; B^\Theta \rangle$.

The formal translation is presented in Figures 15 and 16. In the translation, we make the assumption that all variables are unique and that when we introduce a variable in a term or in a continuation, it is "fresh" (*i.e.* it is does not conflict with any of the other variables in the source term, type-checking context, or continuation).

The invariant guiding the transformation has three main parts:

1. The region language term e is well-formed under the type and value contexts Δ and Γ. Formally, $\Delta; \Gamma \vdash_R e$.

2. The continuation $k = \langle x^k; e^k \rangle$ is well-formed in the current context. Formally, $\{\}; \mathcal{K}[\![\Delta]\!], \Delta^\Theta; \mathcal{S}[\![\Gamma]\!], \Gamma^\Theta, x^k{:}\mathcal{T}[\![\tau]\!]; C^\Theta \vdash e^k$.

3. Finally, the current capability C^Θ is a subcapability of $\overline{B^\Theta}$, and moreover, B^Θ contains a superset of the regions mentioned in the effect of e. Formally,

   - $\mathcal{K}[\![\Delta]\!], \Delta^\Theta \vdash C^\Theta \leq \overline{B^\Theta}$ and
   - $\mathcal{K}[\![\Delta]\!], \Delta^\Theta \vdash \overline{B^\Theta} = \overline{B^\Theta \oplus \mathcal{T}[\![\psi]\!]}$

As in the case for the simplified CPS translation, the translation of source-language values is the simplest. For example, to translate a variable x or the integer i, we simply apply the continuation k to x or i respectively. The type application expression $f[c_1, \ldots, c_n]$ is also a value because we take the interpretation that types are erased at run time. Therefore, we apply the continuation k directly to $f[\mathcal{T}[\![c_1]\!], \ldots, \mathcal{T}[\![c_n]\!]]$.

$$\mathcal{A}(\langle \Gamma; x; e\rangle, v) = (\texttt{let}\ \ x = v\ \texttt{in}\ e)$$

$$\mathcal{C}_{\Delta;\Gamma;\Theta}(x)k = \mathcal{A}(k, x)$$

$$\mathcal{C}_{\Delta;\Gamma;\Theta}(i)k = \mathcal{A}(k, i)$$

$$\mathcal{C}_{\Delta;\Gamma;\Theta}(f[c_1, \ldots, c_n])k = \mathcal{A}(k, f[\mathcal{T}[\![c_1]\!], \ldots, \mathcal{T}[\![c_n]\!]])$$

$$\begin{array}{l}
\mathcal{C}_{\Delta;\Gamma;\Theta}(e_1\ p\ e_2)k = \\
\quad \mathcal{C}_{\Delta;\Gamma;\Theta}(e_1)\langle x_1; \\
\quad \mathcal{C}_{\Delta;\Gamma;\Theta,x_1:\texttt{int}}(e_2)\langle x_2; \\
\quad \texttt{let}\ x = x_1\ p\ x_2\ \texttt{in} \\
\quad \mathcal{A}(k, x)\rangle\rangle
\end{array}$$

$$\begin{array}{l}
\mathcal{C}_{\Delta;\Gamma;\Theta}(\langle e_1^{\tau_1}, \ldots, e_n^{\tau_n}\rangle\ \texttt{at}\ e_{n+1}^{\tau_{n+1}})k = \\
\quad \mathcal{C}_{\Delta;\Gamma;\Theta}(e_1)\langle x_1; \\
\quad \vdots \\
\quad \mathcal{C}_{\Delta;\Gamma;\Theta,x_1:\mathcal{T}[\![\tau_1]\!],\ldots,x_{n-1}:\mathcal{T}[\![\tau_{n-1}]\!]}(e_n)\langle x_n; \\
\quad \mathcal{C}_{\Delta;\Gamma;\Theta,x_1:\mathcal{T}[\![\tau_1]\!],\ldots,x_n:\mathcal{T}[\![\tau_n]\!]}(e_{n+1})\langle x_{n+1}; \\
\quad \texttt{let}\ x = \langle x_1, \ldots, x_n\rangle\ \texttt{at}\ x_{n+1}\ \texttt{in}\ \mathcal{A}(k, x) \\
\quad \rangle\rangle \cdots \rangle
\end{array}$$

$$\mathcal{C}_{\Delta;\Gamma;\Theta}(\pi_i e)k = \mathcal{C}_{\Delta;\Gamma;\Theta}(e)\langle x_1; \texttt{let}\ x = \pi_i x_1\ \texttt{in}\ \mathcal{A}(k, x)\rangle$$

$$\begin{array}{l}
\mathcal{C}_{\Delta;\Gamma;\Theta}(\texttt{if0}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3)k = \\
\quad \mathcal{C}_{\Delta;\Gamma;\Theta}(e_1)\langle x_1; \\
\quad \texttt{if0}\ x_1\ \texttt{then}\ \mathcal{C}_{\Delta;\Gamma;\Theta,x_1:\texttt{int}}(e_2)k\ \texttt{else}\ \mathcal{C}_{\Delta;\Gamma;\Theta,x_1:\texttt{int}}(e_3)k\rangle
\end{array}$$

$$\begin{array}{l}
\mathcal{C}_{\Delta;\Gamma;\Theta}(\texttt{letregion}\,\rho, x_\rho\ \texttt{in}\ e)k = \\
\quad \texttt{let newrgn}\,\rho, x_\rho\ \texttt{in} \\
\quad \mathcal{C}_{\Delta\{\rho:\texttt{Rgn}\};\Gamma\{x_\rho:\rho\,\texttt{handle}\};\Theta'}(e)\langle x'; \texttt{freergn}\,x_\rho\ \texttt{in}\ \mathcal{A}(k, x')\rangle
\end{array}$$

where

$$\Theta' \quad = \quad \langle \Delta^\Theta; \Gamma^\Theta; C^\Theta \oplus \{\rho^1\}; B^\Theta \oplus \{\rho^1\}\rangle$$

Figure 15: Region to Capability Term Translation

$\mathcal{C}_{\Delta;\Gamma;\Theta}(\texttt{letrec}f[\Delta'](x) : (\forall[\Delta'].\tau_1 \xrightarrow{\psi_f} \tau_2 \texttt{ at } r) \texttt{ at } e_1 = e_2 \texttt{ in } e_3)k =$

$\quad \mathcal{C}_{\Delta;\Gamma;\Theta}(e_1)\langle x_1;$

$\quad \texttt{let } f = (\texttt{fix } f[\mathcal{K}[\![\Delta']\!], \rho{:}\texttt{Rgn}, \epsilon{:}\texttt{Cap}, \epsilon' \leq \overline{B^{\Theta'}}](\epsilon', x{:}\mathcal{T}[\![\tau_1]\!], x_{cont}{:}\tau_{cont}).$

$\qquad\qquad \mathcal{C}_{\Delta\Delta';\Gamma\{f:\sigma,x:\tau_1\};\Theta'}(e_2)\langle x_2; x_{cont}(x_2)\rangle) \texttt{ at } x_1$

$\quad \texttt{in}$

$\quad \mathcal{C}_{\Delta;\Gamma\{f:\forall[\Delta'].\tau_1 \xrightarrow{\psi_f} \tau_2 \texttt{at} r\};\Theta,x_1:\mathcal{T}[\![r \texttt{ handle}]\!]}(e_3)k\rangle$

where

$$\begin{array}{lcl}
\tau_{cont} & = & \forall[\,].(\epsilon', \mathcal{T}[\![\tau_2]\!]) \rightarrow 0 \texttt{ at } \rho \\
\Delta^{\Theta'} & = & \Delta^{\Theta}, \rho{:}\texttt{Rgn}, \epsilon{:}\texttt{Cap}, \epsilon' \leq \overline{B^{\Theta'}} \\
\Gamma^{\Theta'} & = & \Gamma^{\Theta}, x_1{:}r \texttt{ handle}, x_{cont}{:}\tau_{cont} \\
C^{\Theta'} & = & \epsilon' \\
B^{\Theta'} & = & \epsilon \oplus \mathcal{T}[\![\psi_f]\!] \oplus \{\rho^1\} \\
\Theta' & = & \langle \Delta^{\Theta'}; \Gamma^{\Theta'}; C^{\Theta'}; B^{\Theta'} \rangle
\end{array}$$

$\mathcal{C}_{\Delta;\Gamma;\Theta}(e_1^{\tau_f}\, e_2)k =$

$\quad \mathcal{C}_{\Delta;\Gamma;\Theta}(e_1)\langle x_1;$

$\quad \mathcal{C}_{\Delta;\Gamma;\Theta,x_1:\mathcal{T}[\![\tau_f]\!]}(e_2)\langle x_2;$

$\quad \texttt{let newrgn } \rho, x_\rho \texttt{ in}$

$\quad \texttt{let } f_{cont} = (\texttt{fix } f_{cont}[\,](C^{\Theta} \oplus \{\rho^1\}, x{:}\mathcal{T}[\![\tau_2]\!]).\texttt{let freergn } x_\rho \texttt{ in } \mathcal{A}(k, x)) \texttt{ at } x_\rho$

$\quad \texttt{in } x_1[\rho, B^{\Theta}, C^{\Theta} \oplus \{\rho^1\}](x_2, f_{cont})\rangle\rangle$

where

$$\tau_f \quad = \quad \tau_1 \xrightarrow{\psi_f} \tau_2 \texttt{ at } r$$

Figure 16: Region to Capability Term Translation (Functions)

The translation of tuples also follows our informal description closely. We translate each of the computations $e_1, \ldots, e_{n+1}$ that make up the tuple in sequence and bind the resulting values to $x_1, \ldots, x_{n+1}$. Once we have translated all expressions, we allocate the tuple $\langle x_1, \ldots, x_n \rangle$ using the region handle x_{n+1}.

In order for the tuple allocation operation to be safe, we must ensure the region (say, ρ) that corresponds to the region handle x_{n+1} is still live. In other words, we must be able to prove that the current capability C^Θ contains a capability for ρ. The invariants above provide us with the means to deduce this fact using the following informal reasoning. Invariant 1 states that the expression $\langle e_1, \ldots, e_n \rangle$ `at` e_{n+1} is well-formed and has effect ψ. Inspection of the region language typing rule for tuples reveals that ψ contains an effect on ρ. Now, the second part of invariant 3 states that the capability B^Θ contains capabilities on all regions that appear in the effect ψ, including, of course, ρ. Finally, using the first part of invariant 3, we know that the current capability C^Θ is subcapability of B^Θ, and, therefore, that it too contains ρ. Consequently, the tuple allocation operation is safe. Using similar reasoning, it is straightforward to verify informally that the translation of arithmetic operations, projections and if statements will not fail to type check because they lack sufficient capabilities.

The translation of the term `letregion` ρ, x_ρ `in` e is not much more difficult: `letregion` ρ, x_ρ `in` e is translated into a `newrgn` ρ, x_ρ declaration followed by the translation of the inner expression e and finally a `freergn` declaration to deallocate ρ. Once again, some simple reasoning allows us to check that the stated invariants hold throughout the transformation. In particular, the translation of the inner expression e reflects the fact that a new region ρ has just been allocated; the translation environment for that step contains capabilities $C^\Theta \oplus \{\rho^1\}$ and $B^\Theta \oplus \{\rho^1\}$. Since we know that C^Θ is a subcapability of $\overline{B^\Theta}$, we may conclude that $C^\Theta \oplus \{\rho^1\}$ is a subcapability of $\overline{B^\Theta \oplus \{\rho^1\}}$ and therefore that invariant 3, part 1 is satisfied. Next, inspection of the typing rule for `letregion` reveals that if ψ' is the effect of e then $\psi' \setminus \{\rho\}$ is the effect of the entire `letregion` expression. Since, B^Θ contains $\psi' \setminus \{\rho\}$, we know that $B^\Theta \oplus \{\rho^1\}$ contains capabilities for all regions in ψ', including, of course, ρ. Hence, we also satisfy invariant 3, part 2 during the translation of e. Finally, recall that the region type system ensures that the region ρ can only be used inside of e. Therefore, in the continuation for the translation of e, we safely free the region and proceed with k. Invariant 2 specified that k only expected the capability C^Θ and this is exactly the capability of the context after freeing the region ρ.

The most complex part of the translation involves functions. Fortunately, the type translation, which we have already explained, specifies the main invariants; the translation of functions terms follows directly from this specification. More specifically, we extend the function's type context Δ' with ρ, a region for the continuation's closure, ϵ, a capability for hiding extra regions in the calling context, and ϵ', the current capability, which is bounded by $\overline{\epsilon \oplus \mathcal{T}[\![\psi_f]\!] \oplus \{\rho^1\}}$. We also add a value argument to the translated function, the continuation x_{cont}. It is simple to verify that the translated function has the translated function type. The body of the function is translated under these assumptions. The continuation for this part of the translation does nothing but invoke the function's new argument x_{cont}.

Finally, we examine the translation of a function application. First, the translation allocates a new region ρ for the continuation closure. Next, the translation actually allocates the continuation in ρ. This continuation is defined to expect the capability $C^\Theta \oplus \{\rho^1\}$. This is the maximum capability at this point in the computation and it permits the continuation to deallocate the region containing its own closure. After allocating the continuation, the translation continues with the translation of the actual function application. The translated function value, x_1, will have the type

$$\forall[\rho' : \mathtt{Rgn}, \epsilon : \mathtt{Cap}, \epsilon' \leq \overline{\epsilon \oplus \mathcal{T}[\![\psi_f]\!] \oplus \{\rho'^1\}}].(\cdots) \rightarrow 0 \ \mathtt{at}\ r$$

Therefore, before calling the function, the code must instantiate the variables ρ', ϵ, and ϵ' properly. The code naturally instantiates ρ' with the region ρ that was just allocated. At this point in the program, the capability $C^\Theta \oplus \{\rho^1\}$ represents the current capability and $B^\Theta \oplus \{\rho^1\}$ is its upper bound. Therefore, the code instantiates ϵ with B^Θ and ϵ' with $C^\Theta \oplus \{\rho^1\}$, which is legal provided we can prove that

$$C^\Theta \oplus \{\rho^1\} \leq \overline{B^\Theta \oplus \psi_f \oplus \{\rho^1\}}$$

Given invariant 3, which states that $C^\Theta \leq \overline{B^\Theta}$ and that $\overline{B^\Theta} = \overline{B^\Theta \oplus \psi_f}$, it is easy to verify this fact. Now, we need only verify that the term arguments, x_2 and the continuation f_{cont}, have the types expected by the translated function and this too can be easily checked.

Properties of the Translation We have proven that our translation preserves types.

Theorem 8 (CPS Type Preservation)
If $\cdot;\cdot \vdash e : \texttt{int}, \emptyset$ *then* $\{\};\cdot;\cdot;\emptyset \vdash \mathcal{C}_{\cdot;\cdot;\Theta}(e)\langle\cdot; x; \texttt{halt}\ x\rangle$ *where* x *is fresh and* Θ *is the empty enviroment:* $\langle\cdot;\cdot;\emptyset;\emptyset\rangle$.

The proof proceeds by induction on the height typing derivation of the source language term with invariants 1, 2, and 3 forming the induction hypothesis. Appendix B contains a formal proof of the representative cases.

We would also like our translation to preserve the space used by the program. Recently, Minamide [26] has proven that a standard CPS translation preserves the maximum amount of reachable space within a constant factor. He defines a space-profiling semantics for the simply-typed lambda calculus that refines the work of Blelloch and Greiner [7]. If we were to augment our semantics with this sort of space-profiling information, we may be able to prove a similar result for our translation. An informal inspection of the translation indicates that the resultant term allocates no more data structures than the source term with the exception of the continuation closure that we require to return from a function call, and intuitively, this closure corresponds to the stack space that is required to save local variables across a function call. However, a formal investigation of the space properties of our translation is beyond the scope of this paper.

An Example The program below is the translation of the `count` function from the previous section. We have made several simplifications to the output of the formal translation to improve the readability of the program. In particular, we have optimized away many of the administrative redexes and performed a tail-call optimization on the recursive call to the count function. Rather than writing capabilities $\overline{\{\rho_1^1, \rho^1\}}$, we use the equivalent form $\{\rho_1^+, \rho^+\}$.

The program begins by allocating regions ρ_1 and ρ_2 using the `newrgn` declaration, and puts the closure for `count` into ρ_1. The `count` function requires a capability ϵ' at least as good as the capability $\{\rho_1^+, \rho^+, \rho_{cont}^+\}$ needed to access itself, its argument, and its continuation; and it passes on that same capability ϵ' to its continuation k. As we type check the body of the count function, we verify that we possess the capabilities necessary to make all data accesses legal. Comments in the code indicate where these checks occur. When calling `count`, we pass it the continuation `cont`. This continuation requires the capability $\{\rho_1^1, \rho_2^1, \rho_3^1\}$ in order to free the three regions. Hence, at the application site, `count`'s capability, ϵ', is instantiated with the stronger capability needed by the continuation.

```
let newrgn ρ1, xρ1 in
let newrgn ρ2, xρ2 in
% capability held is {ρ1^1, ρ2^1}
let count =
  (fix count
       [ρ:Rgn, ρcont:Rgn, ε:Cap, ε' ≤ ε̄ ⊕ {ρ1^+, ρ^+, ρcont^+}]
       (ε', xρ:ρ handle, x:⟨int⟩ at ρ, k:(ε') → 0 at ρcont) .
    % capability held is ε' ≤ ε̄ ⊕ {ρ1^+, ρ^+, ρcont^+}
    let n = π0(x) in                                        % ρ ok
      if0 n
        then k()                                            % ρcont ok
        else
          let n' = n − 1 in
          let x' = ⟨n'⟩ at xρ in                            % ρ ok
            count [ρ, ρcont, ε̄ ⊕ {ρ1^+, ρ^+, ρcont^+}, ε'] (xρ, x', k) % ρ1 ok
  ) at xρ1 in
let newrgn ρ3, xρ3 in
% capability held is {ρ1^1, ρ2^1, ρ3^1}
let ten = ⟨10⟩ at xρ2 in
let cont =
  (λ ({ρ1^1, ρ2^1, ρ3^1}) .
    % capability held is {ρ1^1, ρ2^1, ρ3^1}
    let freergn xρ3 in                                      % ρ3 unique
    let freergn xρ2 in                                      % ρ2 unique
    let freergn xρ1 in                                      % ρ1 unique
      halt 0
  ) at xρ3
in
  count [ρ2, ρ3, {ρ1^1, ρ2^1, ρ3^1}, {ρ1^1, ρ2^1, ρ3^1}] (xρ2, ten, cont)
```

Another Example In this context, the `count` function uses all of the regions that are currently allocated and the capability variable ϵ is redundant. When the code instantiates ϵ at the call site for `count`, it does so with exactly the regions ρ_1, ρ, and ρ_{cont} which already appear in the bound on ϵ'. However, in general, ϵ will hide some "left-over" capability. For example, if we had allocated a fourth region, ρ_4, we would need to instantiate ϵ with the capability $\{\rho_4^1\} \oplus \{\rho_1^1, \rho_2^1, \rho_3^1\}$ and make corresponding changes to the continuation. Now, ϵ would hide the capability on the fourth region from `count` but preserve it across the call so it could be deallocated in the continuation:

```
%%% count with ε hiding a left-over capability
let newrgn ρ1, xρ1 in
let newrgn ρ2, xρ2 in
let newrgn ρ3, xρ3 in
let newrgn ρ4, xρ4 in
let count = ... as before ...
```

```
% capability held is {ρ1^1, ρ2^1, ρ3^1, ρ4^1}
let ten = ⟨10⟩ at x_ρ2 in
let cont =
  (λ ({ρ1^1, ρ2^1, ρ3^1, ρ4^1}) ...)  at x_ρ2
in
  count [ρ2, ρ3, {ρ4^1} ⊕ {ρ1^1, ρ2^1, ρ3^1}, {ρ4^1} ⊕ {ρ1^1, ρ2^1, ρ3^1}] (x_ρ2, ten, cont)
```

The power of bounded quantification comes into play when a function is called with several regions, some of which may or may not be the same. For example, the original code could be rewritten to have `ten` and `cont` share a region, without changing the function `count` in any way:

```
%%% count with ten and cont sharing ρ2
let newrgn ρ1, x_ρ1 in
let newrgn ρ2, x_ρ2 in
let count = ... as before ...
% capability held is {ρ1^1, ρ2^1}
let ten = ⟨10⟩ at x_ρ2 in
let cont =
  (λ ({ρ1^1, ρ2^1}) ...)  at x_ρ2
in
  count [ρ2, ρ2, {ρ1^1, ρ2^1}, {ρ1^1, ρ2^1}] (x_ρ2, ten, cont)
```

In this example, ρ_{cont} is instantiated with ρ_2 and ϵ' is instantiated with $\{\rho_1^1, \rho_2^1\}$ (which is again the capability required by `cont`). However, `count` proceeds exactly as before because ϵ' is still as good as $\{\rho_1^+, \rho^+, \rho_{cont}^+\}$ since:

$$\begin{aligned} \{\rho_1^1, \rho_2^1\} &\leq \{\rho_1^+, \rho_2^+\} \\ &= \underline{\{\rho_1^+, \rho_2^+\}} \oplus \{\rho_1^+, \rho_2^+, \rho_2^+\} \\ &= \underline{\{\rho_1^1, \rho_2^1\}} \oplus \{\rho_1^+, \rho_2^+, \rho_2^+\} \\ &= (\overline{\epsilon} \oplus \{\rho_1^+, \rho_2^+, \rho_2^+\})[\{\rho_1^1, \rho_2^1\}/\epsilon] \end{aligned}$$

An Optimization In the examples above, even though `count` is tail-recursive, we allocate a new cell each time around the loop and we do not deallocate any of the cells until the count is complete. However, since ρ never contains any live values other than the current argument, it is safe to reduce the program's space usage by deallocating the argument's region each time around the loop, as shown in Figure 17. Note that this optimization is not possible when region lifetimes must be lexically scoped.

In order to deallocate its argument, the revised `count` requires a unique capability for its argument's region ρ. Note that if the program were again rewritten so that `ten` and `cont` shared a region (which would lead to a run-time error, since `ten` is deallocated early), the program would no longer typecheck, since $\{\rho_1^1, \rho_2^1\} \not\leq \{\rho_1^+, \rho_2^+, \rho_2^+\}$. However, the program rewritten so that `count` and `cont` share a region does not fail at run time, and does typecheck, since $\{\rho_1^1, \rho_2^1\} \leq \{\rho_1^+, \rho_1^+, \rho_2^1\}$.

```
let newrgn ρ1, xρ1 in
let newrgn ρ2, xρ2 in
% capability held is {ρ1^1, ρ2^1, ρ3^1}
let count =
  (fix count
       [ρ:Rgn, ρcont:Rgn, ε ≤ {ρ1^+, ρcont^+}]
       (ε ⊕ {ρ^1}, xρ:ρ handle, x:⟨int⟩ at ρ,
        k:∀(ε) → 0 at ρcont) .
    % capability held is ε ⊕ {ρ^1}
    let n = π0(x) in                        % ρ ok
    let freergn xρ in                       % ρ unique
      % capability held is ε
      if0 n
        then k()                            % ρcont ok
        else
          let n' = n − 1 in
          let newrgn ρ', xρ' in
          % capability held is ε ⊕ {ρ'^1}
          let x' = ⟨n'⟩ at xρ' in           % ρ' ok
            count [ρ', ρcont, ε] (xρ', x', k)   % ρ1 ok
  ) at xρ1 in
let ten = ⟨10⟩ at xρ2 in
let newrgn ρ3, xρ3 in
let cont =
  (λ ({ρ1^1, ρ3^1}) .
    % capability held is {ρ1^1, ρ3^1}
    let freergn xρ1 in                      % ρ1 unique
    let freergn xρ3 in                      % ρ3 unique
      halt 0
  ) at xρ3
in
  count [ρ2, ρ3, {ρ1^1, ρ3^1}] (xρ2, ten, cont)
```

Figure 17: Count with Efficient Memory Usage

4 From Capability Calculus to Typed Assembly Language

In this section, we informally describe how to construct the backend of a compiler that translates the Capability Calculus into a capability-based typed assembly language.

The continuation-passing style transformation of the previous section makes the order of evaluation of expressions explicit and names all of the intermediate computations. It also reduces all unconditional control-flow transfers to one uniform mechanism: the function call or "functional goto." Hence, this first transformation performs a significant portion of the compilation of a high-level language. However, we are still a long way from a strongly-typed assembly code.

Morrisett *et al.* [33] describe how to eliminate high-level language features such as closures and other data structures by compiling them into a strongly-typed RISC-like assembly language. Fortunately, the addition of capability constructs to our CPS-language does not interfere with the type-directed compilation as they describe it. In fact, the capability constructors are the same at the machine language level as they appear in the Capability Calculus that we have already described. Therefore, in this section, we briefly retrace the compilation strategy that has already been laid out for us by Morrisett *et al.* [33] and point out how it interacts with our capability framework.

4.1 Closure Conversion

Assembly languages do not contain the nested functions that are common in some high-level languages. Closure conversion is the process of transforming each function so that it only depends upon its explicitly declared parameters and does not depend upon values defined in an outer, non-global scope. Once all nested functions have been *closed* in this way, they may be lifted to the top level. This process usually results in pairing closed function code with an environment to create the closure data structure.

There are many typed closure conversion algorithms of various complexities [27, 33, 32]. Minamide *et al.* [27] show that closure converting a polymorphic type-passing language requires a very sophisticated target language that includes translucent sums and abstract kinds as well as existential types. Morrisett *et al.* [33], on the other hand, are able to define a much simpler closure translation because they give up the advantages of a type-passing semantics and assume a type-erasure semantics instead. More recently, Crary *et al.* [8] have discovered how to preserve the advantages of a type-passing language through closure conversion and yet retain the simplicity of the algorithm proposed by Morrisett *et al.*. Crary's approach achieves the expressiveness of type-passing with the simplicity of a type-erasure semantics by passing values that represent types at runtime instead of types themselves. Crary distinguishes between type constructors (erased at runtime) and type representations (values manipulated at runtime) just as we distinguish between region constructors (erased at runtime) and region handles (values manipulated at runtime). This decision allows us to use a much simpler closure conversion algorithm than would be possible in the type-passing region calculus of Tofte and Talpin.

There are many additional considerations in the choice of closure conversion algorithms involving how to represent environments and how to closure convert mutually recursive functions. For the purposes of this paper, we will assume the simple closure transformation described by Morrisett *et al.*. This algorithm only requires the addition of existential types ($\exists\alpha.\tau$) to the type language. The disadvantage of this algorithm is that it recreates the closure each time around the loop of a recursive function.[3] Morrisett and Harper [32] discuss translations that require either recursive types or a cyclic

[3] An observant reader will recognize that this extra allocation must show up as an extra effect in the high-level languages.

pack operation and do not incur the overhead of repeated closure creation. Any practical language implementation would choose one of these more efficient translations, but because the details of closure conversion are orthogonal to our central concerns, we assume the simplest alternative here and refer the interested reader to the literature for a deeper investigation of the tradeoffs.

4.2 Code Generation

Once closure conversion has been completed, all program memory requirements are explicit. Compiler writers have the opportunity to optimize memory usage further by eliminating closure creation for known functions, optimizing environment representations, and removing the resulting dead regions.

At this point, we begin code generation. Morrisett *et al.* [33] describe code generation in two phases. The first phase breaks the atomic data structure allocation and initialization of high-level languages into a series of lower level instructions that initialize the fields of the data structure one by one. In order to prevent access to uninitialized fields, they add a flag to each field that indicates whether or not it can be read. For example, the expression $\texttt{let}\, x = \langle 2, 3\rangle\, \texttt{at}\, x_r\, \texttt{in} \cdots$ becomes

$$
\begin{array}{ll}
\texttt{let}\, x_0 = \texttt{malloc}\ [\texttt{int}, \texttt{int}]\, \texttt{at}\, x_r\, \texttt{in} & \%\ x_0 : \langle \texttt{int}^0, \texttt{int}^0\rangle\, \texttt{at}\, r \\
\texttt{let}\, x_1 = x_0[0] \leftarrow 2\, \texttt{in} & \%\ x_1 : \langle \texttt{int}^1, \texttt{int}^0\rangle\, \texttt{at}\, r \\
\texttt{let}\, x = x_1[1] \leftarrow 3\, \texttt{in} & \%\ x : \langle \texttt{int}^1, \texttt{int}^1\rangle\, \texttt{at}\, r \\
\cdots &
\end{array}
$$

A 1-flag indicates the field has been initialized is therefore readable. A 0-flag indicates the field has not been initialized. In this case, the initialization steps appear one after another. However, in general, they may separated and if a field is unused, it need not ever be initialized. This flexibility allows an optimizer to schedule instructions as it chooses, without interference from the type system.

The major task of the second phase of code generation is to compile closed code and branching constructs such as our $if0$ statements into assembly language control-flow operations. At this point, all control transfers become jumps to typed assembly language code blocks. Each of these code blocks has a typing precondition and, for the most part, contains standard assembly language instructions with a few additional typing annotations. Because we have already eliminated high-level abstract data structures such as closures and tuples and named all our intermediate computations, there is almost a one-to-one correspondence between the intermediate language declarations and the assembly language instructions.

The next section describes these typed code blocks precisely and gives a complete description of a capability-based typed assembly language. Because code generation has little impact on the novel features of the Capability Calculus, we refer the reader to the work of Morrisett *et al.* [33], which gives a formal description of the details of code generation for a closed, continuation-passing style lambda calculus.

Indeed, to accomodate this translation, when checking a recursive region calculus function of type $\tau_1 \xrightarrow{\psi} \tau_2\, \texttt{at}\, r$, we must require that ψ contain an effect in region r. Similarly, the capability of a recursive capability calculus function must contain the region that function is allocated in.

4.3 Capability-Based Typed Assembly Language

The Capability-Based Typed Assembly Language is based on the generic Typed Assembly Language of Morrisett *et al.* [33]. Because of the great similarity, we only discuss the capability features in detail. The syntax for the language is given in Figure 18.

4.3.1 Types

As discussed above, there is little change to the type language. The kind structure is identical and only tuples and function types have changed. Tuple types have initialization flags on each field as discussed in the previous section. Continuation-passing style function types have been translated into code block types with the following form:

$$\forall[\Delta].C \Rightarrow \Gamma$$

As before, these types abstract a type context Δ and specify the capability C that the code requires. However, the typed assembly language does not have alpha-varying variables and uses a fixed set of non-varying registers instead. Therefore, the precondition for the code block is a finite map Γ from registers to types rather than a simple list of types. Furthermore, these types classify code not closures (a pair of code and data) and we assume all code is allocated in its own special region before execution of the program begins. Therefore, we omit the trailing region annotation that appears in the Capability Calculus closure types. Tuples are the only remaining heap-allocated data. The only addition to the type language is the existential type $\exists\alpha.\tau$, which we need to encode the results of closure conversion.

The separation between code and data is also reflected in the memory types, which are now split into a pair of a code region (Υ_{cd}) and the familiar mapping of region names to data regions (Υ).

Figure 19 contains the well-formedness rules for the new types. The rules for the well-formedness of capabilities and other types are identitical to the rules we gave for the Capability Calculus as are the rules governing the well-formedness of contexts and the type equality and subcapability relations. We refer the reader to previous sections for these rules. For technical reasons, the TAL type system requires a simple subtyping relation on tuple types and register file types. These subtyping relations do not interact with capability subtyping. See Morrisett *et al.* [34] for a discussion of these items or a more complete description of the TAL type structure in general.

4.3.2 Values

In TAL, values are split into three categories: word values, small values, and heap values. Heap values (h), code and tuples, are analogous to their Capability Calculus counterparts except as discussed above. Both small values (v) and word values (w), are "small." We distinguish between the two because the former contains registers whereas the latter does not. The register file ($\Re$) is a mapping from registers ($\mathtt{r}$) to word values. We do not map registers to other registers.

Word values contain the integers (i), data addresses ($\nu.\ell$), and region handles ($\mathtt{handle}(\nu)$) that we saw in the Capability Calculus. Code addresses ($\mathtt{cd}.\ell$) refer to code in the code region. The word value $?\tau$ is an uninitialized value of type τ. Thus, $\langle ?\mathtt{int}, ?\mathtt{int}\rangle$ is an uninitialized pair of integers. The values $w[c]$ and $\mathtt{pack}[\tau', w]$ *as* τ are both coercions that change the type of the word value w but have no runtime effect (for technical reasons, there are analogous small value forms $v[c]$ and $\mathtt{pack}[\tau', v]$ *as* τ).

kinds	κ	::=	$\mathtt{Type} \mid \mathtt{Rgn} \mid \mathtt{Cap}$
constructors	c	::=	$\alpha \mid \tau \mid r \mid C$
types	τ	::=	$\alpha \mid \mathtt{int} \mid r\ \mathtt{handle} \mid \forall[\Delta].C \Rightarrow \Gamma \mid \langle \tau_1^{\xi}, \ldots, \tau_n^{\xi} \rangle\ \mathtt{at}\ r \mid \exists\alpha.\tau$
regions	r	::=	$\rho \mid \nu$
capabilities	C	::=	$\epsilon \mid \emptyset \mid \{r^{\varphi}\} \mid C_1 \oplus C_2 \mid \overline{C}$
multiplicities	φ	::=	$1 \mid +$
init flags	ξ	::=	$0 \mid 1$
memory types	Ψ	::=	$\{\mathtt{cd} : \Upsilon_{cd}, \nu_1 : \Upsilon_1, \ldots, \nu_n : \Upsilon_n\}$
region types	Υ	::=	$\{\ell_1 : \tau_1, \ldots, \ell_n : \tau_n\}$
register file types	Γ	::=	$\{\mathtt{r}_1 : \tau_1, \ldots, \mathtt{r}_n : \tau_n\}$
contexts	Δ	::=	$\cdot \mid \Delta, \alpha{:}\kappa \mid \Delta, \epsilon \leq C$
registers	$\mathtt{r}$	$\in$	$\{\mathtt{r1}, \mathtt{r2}, \ldots\}$
word values	w	::=	$\nu.\ell \mid \mathtt{cd}.\ell \mid i \mid ?\tau \mid w[c] \mid \mathtt{pack}[\tau', w]\ as\ \tau \mid \mathtt{handle}(\nu)$
small values	v	::=	$\mathtt{r} \mid w \mid v[c] \mid \mathtt{pack}[\tau', v]\ as\ \tau$
heap values	h	::=	$\langle w_1, \ldots, w_n \rangle \mid \mathtt{code}[\Delta]C \Rightarrow \Gamma.I$
inst. sequences	I	::=	$\iota; S \mid \mathtt{jmp}\ v \mid \mathtt{halt}$
instructions	ι	::=	$\mathtt{add}\ \mathtt{r}_d, \mathtt{r}_s, v \mid \mathtt{bnz}\ \mathtt{r}, v \mid \mathtt{freergn}\ v \mid \mathtt{ld}\ \mathtt{r}_d, \mathtt{r}_s[i] \mid \mathtt{malloc}\ \mathtt{r}_d, v[\vec{\tau}] \mid$ $\mathtt{mov}\ \mathtt{r}_d, v \mid \mathtt{mul}\ \mathtt{r}_d, \mathtt{r}_s, v \mid \mathtt{newrgn}\ \rho, \mathtt{r}_d \mid \mathtt{st}\ \mathtt{r}_d[i], \mathtt{r}_s \mid$ $\mathtt{sub}\ \mathtt{r}_d, \mathtt{r}_s, v \mid \mathtt{unpack}\ [\alpha, \mathtt{r}_d], v$
memory	M	::=	$\{\mathtt{cd} \mapsto R_{cd}, \nu_1 \mapsto R_1, \ldots, \nu_n \mapsto R_n\}$
heaps	R	::=	$\{\ell_1 \mapsto h_1, \ldots, \ell_n \mapsto h_n\}$
register files	$\Re$	::=	$\{\mathtt{r}_1 \mapsto w_1, \ldots, \mathtt{r}_n \mapsto w_n\}$
programs	P	::=	$(M, \Re, I)$

Figure 18: Capability-Based TAL Syntax

The first is the type instantiation that we saw in the Capability Calculus. If τ is $\exists\alpha.\tau''$ then the second value form introduces an existential type by hiding τ' behind the abstract type variable α.

In general, the form and meaning of the static semantic judgements for values are similar to their counter-parts in the Capability Calculus. One addition is the judgement $\Psi;\Delta;\Gamma;C \vdash w : \tau^{\xi}$ fwval, which states that given a well-formed typing context the word value w can be given the flagged type τ^{ξ}. Figure 20 contains the rules for value well-formedness.

4.3.3 TAL Abstract Machine

In general, the TAL instruction set closely ressemble the instructions of a standard RISC instruction set and their semantics should be intuitive. For example, `add` $\mathtt{r}_d, \mathtt{r}_s, v$, adds the contents of register $\mathtt{r}_s$ and the value v, and places the result in register $\mathtt{r}_d$. The instructions `mul` and `sub`, multiply and subtract in a similar fashion. The `ld` $\mathtt{r}_d, \mathtt{r}_s[i]$ instruction loads the ith word from the data block indicated by the pointer in register $\mathtt{r}_s$ into the register $\mathtt{r}_d$. The `st` $\mathtt{r}_d[i], \mathtt{r}_s$ instruction writes the contents of $\mathtt{r}_s$ into the ith word of memory pointed to by $\mathtt{r}_d$. The `mov` $\mathtt{r}_d, v$ instruction, copies the source operand (v) to the destination register ($\mathtt{r}_d$). This simplified language also contains two control-flow instructions: `jmp` v, an unconditional jump to the address contained in v and `bnz` $\mathtt{r}, v$, a jump to v if the contents of the register $\mathtt{r}$ are non-zero and a fall-thru to the next instruction otherwise.

The other instructions cannot be found in a typical RISC instruction set. The commands `malloc`, `newrgn`, and `freergn` are calls to the runtime system. `newrgn` and `freergn` allocate and deallocate regions in a similar manner to their high-level counter-parts. `malloc` $\mathtt{r}_d, v[\vec{\tau}]$ allocates space equivalent to the sum of the sizes of the types $\vec{\tau}$ in the region indicated by the region handle v. The resulting address is placed in register $\mathtt{r}_d$. None of these instructions can be coded in our typed assembly language. Hence, their implementations, along with the implementation of the type checker forms our trusted computing base. Fortunately, these region primitives are much simpler to implement than a standard tracing garbage collector. Hence, we have acheived one of our goals, a reduction in the trusted computing base over previous low-level safe language efforts such as the orginal TAL and Proof-Carrying Code [35].

The last two instructions, `halt` and `unpack` are also somewhat special. The `unpack` instruction is the elimination form for existential types. An existential v of type $\exists\alpha.\tau$ is opened and a value of type τ is written to the register $\mathtt{r}_d$. The type variable α is introduced into the type context. At runtime, types are erased so this instruction reduces to either a move, or if v is the same register as $\mathtt{r}_d$ (the normal case), this instruction becomes a no-op. Intuitively, the `halt` instruction returns control to the operating system, which expects an integer in register $\mathtt{r}_1$.

An abstract machine state (P) contains three components: memory, a register file, and a list of instructions. As noted above, memory (M), contains both a code region and a set of data regions. Memory is described by the memory type Ψ. The register file ($\Re$), described by the register file type Γ, is a mapping from registers ($\mathtt{r}$) to word values. The typing rules for the abstract machine and its instruction set is presented in Figures 21 and 22. The operational semantics of the machine can be found in Figure 23. Again, given the previous development, the semantics is quite straightforward. In this semantics, we use one additional piece of notation: $[c_1,\ldots,c_n/\Delta]$ denotes the substitution of the constructors $\vec{c}$ for the variables in the domain of Δ.

We claim that the static semantics for the TAL abstract machine is sound with respect to the operational semantics:

$\boxed{\Delta \vdash c : \kappa \text{ (new rules)}}$

$$\frac{\Delta \vdash \Delta' \quad \Delta, \Delta' \vdash C : \texttt{Cap} \quad \Delta, \Delta' \vdash \Gamma}{\Delta \vdash \forall[\Delta'].C \Rightarrow \Gamma : \texttt{Type}} \qquad \frac{\Delta \vdash \tau_i : \texttt{Type} \quad (\text{for } 1 \leq i \leq n) \quad \Delta \vdash r : \texttt{Rgn}}{\Delta \vdash \langle \tau_1^{\xi_1}, \ldots, \tau_n^{\xi_n} \rangle \texttt{ at } r : \texttt{Type}}$$

$$\frac{\Delta, \alpha{:}\texttt{Type} \vdash \tau : \texttt{Type}}{\Delta \vdash \exists \alpha.\tau : \texttt{Type}}$$

$\boxed{\vdash \Psi \quad \vdash \Upsilon \quad \Delta \vdash \Gamma}$

$$\frac{\vdash \Upsilon_{cd} \quad \vdash \Upsilon_i \quad (\text{for } 1 \leq i \leq n)}{\vdash \{\texttt{cd} : \Upsilon_{cd}, \nu_1 : \Upsilon_1, \ldots, \Upsilon_n : \tau_n\}} \qquad \frac{\cdot \vdash \tau_i : \texttt{Type}}{\vdash \{\ell_1 : \tau_1, \ldots, \ell_n : \tau_n\}}$$

$$\frac{\Delta \vdash \tau_i : \texttt{Type} \quad (\text{for } 1 \leq i \leq n)}{\Delta \vdash \{\mathfrak{r}_1 : \tau_1, \ldots, \mathfrak{r}_n : \tau_n\}}$$

$\boxed{\Delta \vdash c_1 \leq c_2 : \kappa \text{ (new rules)} \quad \Delta \vdash \Gamma_1 \leq \Gamma_2}$

$$\frac{\Delta \vdash c_1 = c_2 : \kappa}{\Delta \vdash c_1 \leq c_2 : \kappa} \qquad \frac{\Delta \vdash c_1 \leq c_2 : \kappa \quad \Delta \vdash c_2 \leq c_3 : \kappa}{\Delta \vdash c_1 \leq c_3 : \kappa}$$

$$\frac{\Delta \vdash \tau_i : \texttt{Type} \quad (\text{for } 1 \leq i \leq n) \quad \Delta \vdash r : \texttt{Rgn}}{\Delta \vdash \langle \tau_1^{\xi_1}, \ldots, \tau_{j-1}^{\xi_{j-1}}, \tau_j^1, \tau_{j+1}^{\xi_{j+1}}, \ldots, \tau_n^{\xi_n} \rangle \texttt{ at } r \leq \langle \tau_1^{\xi_1}, \ldots, \tau_{j-1}^{\xi_{j-1}}, \tau_j^0, \tau_{j+1}^{\xi_{j+1}}, \ldots, \tau_n^{\xi_n} \rangle \texttt{ at } r : \texttt{Type}}$$

$$\frac{\Delta \vdash \tau_i = \tau_i' : \texttt{Type} \quad (\text{for } 1 \leq i \leq n) \quad \Delta \vdash \tau_i : \texttt{Type} \quad (\text{for } n+1 \leq i \leq m)}{\Delta \vdash \{\mathfrak{r}_1 : \tau_1, \ldots, \mathfrak{r}_m : \tau_m\} \leq \{\mathfrak{r}_1 : \tau_1', \ldots, \mathfrak{r}_n : \tau_n'\}} \; (m \geq n)$$

Figure 19: Capability-Based TAL: Abbreviated Type Well-Formedness

$\boxed{\Psi \vdash h\ \mathtt{at}\ \nu : \tau \qquad \Psi \vdash h : \tau\ \text{hcode} \qquad \Psi;\Delta \vdash w : \tau \qquad \Psi;\Delta \vdash w : \tau^{\xi}}$

$$\frac{\Psi;\cdot \vdash w_i : \tau_i^{\xi_i} \quad (\text{for } 1 \le i \le n)}{\Psi;\nu \vdash \langle w_1,\ldots,w_n\rangle : \langle \tau_1^{\xi_1},\ldots,\tau_n^{\xi_n}\rangle\ \mathtt{at}\ \nu} \qquad \frac{\cdot \vdash \Delta \quad \Delta \vdash C : \mathtt{Cap} \quad \Delta \vdash \Gamma \quad \Psi;\Delta;\Gamma;C \vdash I}{\Psi \vdash \mathtt{code}[\Delta]C{\Rightarrow}\Gamma.I : \forall[\Delta].C{\Rightarrow}\Gamma\ \text{hcode}}$$

$$\frac{\Delta \vdash \tau = \langle \tau_1^{\xi_1},\ldots,\tau_n^{\xi_n}\rangle\ \mathtt{at}\ \nu}{\Psi;\Delta \vdash \nu.\ell : \tau}\ (\nu \notin Dom(\Psi)) \qquad \frac{\Delta \vdash \tau_1 \le \tau_2}{\Psi;\Delta \vdash \nu.\ell : \tau_2}\ (\Psi(\nu.\ell) = \tau_1)$$

$$\frac{}{\Psi;\Delta \vdash \mathtt{cd}.\ell : \tau_1}\ (\Psi(\mathtt{cd}.\ell) = \tau_1) \qquad \frac{}{\Psi;\Delta \vdash i : \mathtt{int}} \qquad \frac{}{\Psi;\Delta \vdash \mathtt{handle}(\nu) : \nu\ \mathtt{handle}}$$

$$\frac{\Psi;\Delta \vdash w : \forall[\alpha{:}\kappa,\Delta].C{\Rightarrow}\Gamma \quad \Delta \vdash c : \kappa}{\Psi;\Delta \vdash w[c] : (\forall[\Delta].C{\Rightarrow}\Gamma)[c/\alpha]} \qquad \frac{\Psi;\Delta \vdash w : \forall[\epsilon{\le}C_1,\Delta].C_2{\Rightarrow}\Gamma \quad \Delta \vdash C \le C_1 : \mathtt{Cap}}{\Psi;\Delta \vdash w[C] : (\forall[\Delta].C_2{\Rightarrow}\Gamma)[C/\epsilon]}$$

$$\frac{\Delta \vdash c : \mathtt{Type} \quad \Psi;\Delta \vdash w : \tau[c/\alpha]}{\Psi;\Delta \vdash \mathtt{pack}[\tau',w]\ as\ \exists\alpha.\tau : \exists\alpha.\tau}$$

$$\frac{\Psi;\Delta \vdash w : \tau}{\Psi;\Delta \vdash w : \tau^{\xi}} \qquad \frac{\Delta \vdash \tau : \mathtt{Type}}{\Psi;\Delta \vdash ?\tau : \tau^{0}}$$

$\boxed{\Psi;\Delta;\Gamma \vdash v : \tau}$

$$\frac{}{\Psi;\Delta;\Gamma \vdash \mathfrak{r} : \tau}\ (\Gamma(r) = \tau) \qquad \frac{\Psi;\Delta \vdash w : \tau}{\Psi;\Delta;\Gamma \vdash w : \tau}$$

$$\frac{\Psi;\Delta;\Gamma \vdash v : \forall[\alpha{:}\kappa,\Delta].C{\Rightarrow}\Gamma \quad \Delta \vdash c : \kappa}{\Psi;\Delta;\Gamma \vdash v[c] : (\forall[\Delta].C{\Rightarrow}\Gamma)[c/\alpha]} \qquad \frac{\Psi;\Delta;\Gamma \vdash v : \forall[\epsilon{\le}C_1,\Delta].C_2{\Rightarrow}\Gamma \quad \Delta \vdash C \le C_1 : \mathtt{Cap}}{\Psi;\Delta;\Gamma \vdash v[C] : (\forall[\Delta].C_2{\Rightarrow}\Gamma)[C/\epsilon]}$$

$$\frac{\Delta \vdash c : \mathtt{Type} \quad \Psi;\Delta;\Gamma \vdash v : \tau[c/\alpha]}{\Psi;\Delta;\Gamma \vdash \mathtt{pack}[\tau',v]\ as\ \exists\alpha.\tau : \exists\alpha.\tau}$$

Figure 20: Capability-Based TAL: Value Formation

$\boxed{\Psi;\Delta;\Gamma;C \vdash I}$

$$\frac{\Psi;\Delta;\Gamma;C \vdash \iota \Rightarrow \Delta';\Gamma';C' \quad \Psi;\Delta';\Gamma';C' \vdash I}{\Psi;\Delta;\Gamma;C \vdash \iota;I} \qquad \frac{\Psi;\Delta;\Gamma \vdash \mathtt{r1} : \tau}{\Psi;\Delta;\Gamma;C \vdash \mathtt{halt}\ [\tau]}$$

$$\frac{\Psi;\Delta;\Gamma \vdash v : \forall[\,].\, C' \Rightarrow \Gamma' \quad \Delta \vdash C \leq C' : \mathtt{Cap} \quad \Delta \vdash \Gamma \leq \Gamma'}{\Psi;\Delta;\Gamma;C \vdash \mathtt{jmp}\ v}$$

$\boxed{\Psi;\Delta;\Gamma;C \vdash \iota \Rightarrow \Delta';\Gamma';C' \text{ (interesting rules)}}$

$$\frac{\Psi;\Delta;\Gamma \vdash v : r\ \mathtt{handle} \quad \Delta \vdash C \leq C' \oplus \{r\} : \mathtt{Cap}}{\Psi;\Delta;\Gamma;C \vdash \mathtt{freergn}\, v \Rightarrow \Delta;\Gamma;C'}$$

$$\frac{\Psi;\Delta;\Gamma \vdash \mathfrak{r}_s : \langle \tau_0^{\xi_0}, \ldots, \tau_{n-1}^{\xi_{n-1}} \rangle\ \mathtt{at}\ r \quad \Delta \vdash C \leq C' \oplus \overline{\{r\}} : \mathtt{Cap}}{\Psi;\Delta;\Gamma;C \vdash \mathtt{ld}\ \mathfrak{r}_d, \mathfrak{r}_s[i] \Rightarrow \Delta;\Gamma\{\mathfrak{r}_d : \tau_i\};C}\ (\xi_i = 1, 0 \leq i < n)$$

$$\frac{\Delta \vdash \tau_i : \mathtt{Type} \quad (\text{for } 1 \leq i \leq n) \quad \Psi;\Delta;\Gamma \vdash v : r\ \mathtt{handle} \quad \Delta \vdash C \leq C' \oplus \overline{\{r\}} : \mathtt{Cap}}{\Psi;\Delta;\Gamma;C \vdash \mathtt{malloc}\ \mathfrak{r}_d, v[\tau_1, \ldots, \tau_n] \Rightarrow \Delta;\Gamma\{\mathfrak{r}_d : \langle \tau_1^0, \ldots, \tau_n^0 \rangle\ \mathtt{at}\ r\};C}$$

$$\frac{}{\Psi;\Delta;\Gamma;C \vdash \mathtt{newrgn}\, \rho, \mathfrak{r}_d \Rightarrow (\Delta, \rho);\Gamma\{\mathfrak{r}_d : \rho\ \mathtt{handle}\};(C \oplus \{\rho\})}\ (\rho \notin \Delta)$$

$$\frac{\Psi;\Delta;\Gamma \vdash \mathfrak{r}_d : \langle \tau_0^{\xi_0}, \ldots, \tau_{n-1}^{\xi_{n-1}} \rangle\ \mathtt{at}\ r \quad \Psi;\Delta;\Gamma;C \vdash \mathfrak{r}_s : \tau_i \quad \Delta \vdash C \leq C' \oplus \overline{\{r\}} : \mathtt{Cap}}{\Psi;\Delta;\Gamma;C \vdash \mathtt{st}\ \mathfrak{r}_d[i], \mathfrak{r}_s \Rightarrow \Delta;\Gamma\{\mathfrak{r}_d : \langle \tau_0^{\xi_0}, \ldots, \tau_{i-1}^{\xi_{i-1}}, \tau_i^1, \tau_{i+1}^{\xi_{i+1}}, \ldots, \tau_{n-1}^{\xi_{n-1}} \rangle\ \mathtt{at}\ p\};C}\ (0 \leq i < n)$$

Figure 21: Capability-Based TAL: Instructions

$$\boxed{\vdash M : \Psi \quad \Psi \vdash R\,\mathtt{at}\,\nu : \Upsilon \quad \Psi \vdash R : \Upsilon \quad \Psi \vdash \Re : \Gamma \quad \Psi \vdash C \text{ sat} \quad \vdash P}$$

$$\frac{\vdash \Psi \qquad \Psi \vdash R_{cd} : \Upsilon_{cd} \text{ code} \qquad \Psi \vdash R_i\,\mathtt{at}\,\nu_i : \Upsilon_i \text{ val} \quad (\text{for } 1 \le i \le n)}{\vdash \{\mathtt{cd} : \Upsilon_{cd}, \nu_1 \mapsto R_1, \ldots, \nu_n \mapsto R_n\} : \Psi} \; (\Psi = \{\mathtt{cd} : \Upsilon_{cd}, \nu_1 : \Upsilon_1, \ldots, \nu_n : \Upsilon_n\})$$

$$\frac{\Psi \vdash h_i : \tau_i \text{ hcode} \quad (\text{for } 1 \le i \le n)}{\Psi \vdash \{\ell_1 \mapsto h_1, \ldots, \ell_n \mapsto h_n\} : \{\ell_1 : \tau_1, \ldots, \ell_n : \tau_n\} \text{ code}}$$

$$\frac{\Psi; \nu \vdash h_i : \tau_i \qquad (\text{for } 1 \le i \le n)}{\Psi \vdash \{\ell_1 \mapsto h_1, \ldots, \ell_n \mapsto h_n\} : \{\ell_1 : \tau_1, \ldots, \ell_n : \tau_n\}\,\mathtt{at}\,\nu \text{ val}}$$

$$\frac{\Psi; \cdot \vdash w_i : \tau_i \quad (\text{for } 1 \le i \le n)}{\Psi \vdash \{\mathfrak{r}_1 \mapsto w_1, \ldots, \mathfrak{r}_n \mapsto w_m\} : \{\mathfrak{r}_1 : \tau_1, \ldots, \mathfrak{r}_n : \tau_n\}} \; (m \ge n)$$

$$\frac{\cdot \vdash C = \{\nu_1^{\varphi_1}, \ldots, \nu_n^{\varphi_n}\} : \mathtt{Cap}}{\{\nu_1 : \Upsilon_1, \ldots, \nu_n : \Upsilon_n\} \vdash C \text{ sat}} \; (\nu_i \neq \nu_j \text{ for } 1 \le i, j \le n \text{ and } i \neq j)$$

$$\frac{\vdash M : \Psi \quad \Psi \vdash \Re : \Gamma \quad \Psi \vdash C \text{ sat} \quad \Psi; \cdot; \Gamma; C \vdash I}{\vdash (M, \Re, I)}$$

Figure 22: Capability-Based TAL: Abstract Machine

$(M, R, I) \longmapsto P$ where	
if $I =$	then $P =$
`add` $r_d, r_s, v; I'$	$(M, R\{r_d \mapsto R(r_s) + \hat{R}(v)\}, I')$ and similarly for `mul` and `sub`
`bnz` $r, v; I'$ when $R(r) = 0$	(M, R, I')
`bnz` $r, v; I'$ when $R(r) = i$ and $i \neq 0$	$(M, R, I''[\vec{c}/\Delta])$ where $\hat{R}(v) = \texttt{cd}.\ell[\vec{c}]$ and $M(\texttt{cd}.\ell) = \texttt{code}[\Delta]C\Rightarrow\Gamma.I''$
`freergn` v	(M', R, I'') where $\hat{R}(v) = \texttt{handle}(\nu)$ and $\nu \in Dom(M)$ and $M' = M\backslash\nu$
`jmp` v	$(M, R, I'[\vec{\tau}/\Delta])$ where $\hat{R}(v) = \texttt{cd}.\ell[\vec{\tau}]$ and $M(\texttt{cd}.\ell) = \texttt{code}[\Delta]\Gamma.I'$
`ld` $r_d, r_s[i]; I'$	$(M, R\{r_d \mapsto w_i\}, I')$ where $R(r_s) = \nu.\ell$ and $M(\nu.\ell) = \langle w_0, \ldots, w_{n-1}\rangle$ with $0 \leq i < n$
`malloc` $r_d, v[\tau_1, \ldots, \tau_n]; I'$	$(M\{\nu.\ell \mapsto \langle ?\tau_1, \ldots, ?\tau_n\rangle\}, R\{r_d \mapsto \nu.\ell\}, I')$ where $\hat{R}(v) = \texttt{handle}(\nu)$ and $\nu \in Dom(M)$ and $\ell \notin M(\nu)$
`mov` $r_d, v; I'$	$(M, R\{r_d \mapsto \hat{R}(v)\}, I')$
`newrgn` $\rho, r_d; I'$	$(M\{\nu \mapsto \{\}\}, R\{r_d \mapsto \texttt{handle}(\nu)\}, I'[\nu/\rho])$ where ν is new
`st` $r_d[i], r_s; I'$	$(M\{\nu.\ell \mapsto \langle w_0, \ldots, w_{i-1}, R(r_s), w_{i+1}, \ldots, w_{n-1}\rangle\}, R, I')$ where $R(r_d) = \nu.\ell$ and $M(\nu.\ell) = \langle w_0, \ldots, w_{n-1}\rangle$ with $0 \leq i < n$
`unpack` $[\alpha, r_d], v; I'$	$(M, R\{r_d \mapsto w\}, I'[\tau/\alpha])$ where $\hat{R}(v) = \texttt{pack}[\tau, w]$ *as* τ'

$$\text{Where } \hat{R}(v) = \begin{cases} R(r) & \text{when } v = r \\ w & \text{when } v = w \\ \hat{R}(v')[\tau] & \text{when } v = v'[\tau] \\ \texttt{pack}[\tau, \hat{R}(v')] \text{ as } \tau' & \text{when } v = \texttt{pack}[\tau, v'] \text{ as } \tau' \end{cases}$$

Figure 23: Operational Semantics of TAL

Claim 9 (Capability-Based TAL Type Safety) *If* $\vdash P$*, then there is no stuck state* P' *such that* $P \longmapsto^* P'$.

We believe the proof would be long and tedious but straightforward. Like the proof of type safety for the Capability Calculus, this proof is driven by the syntax of the language and does not involve a complicated greatest fixed-point construction.

5 Discussion

We believe the general framework of our capability system is quite robust. There are several ways to extend the language and a number of directions for future research.

5.1 Language Extensions

In this article, we have concentrated on using the Capability Calculus to implement safe region-based memory management, but with a few changes, we believe our capability apparatus may be used in a variety of other settings as well. One potential application involves reducing the overhead of communication across the user-kernel address space boundary in traditional operating systems. Typically, in such systems, when data in user space is presented to the kernel, the kernel must copy that data to ensure its integrity is preserved. However, if a user process hands off a unique capability for a region to the kernel, the kernel does not have to copy that region's data; without the capability, the user can no longer read or modify the contents of that region.

By handing off a user's capability to the kernel, we ensure that the kernel has exclusive access to the data governed by the capability. We can generalize this idea and use capabilities to ensure mutually-exclusive access to shared mutable data in a multi-threaded environment, by viewing locks as analogous to regions. If we associate each piece of sensitive data with a lock, we can statically check that every client of the data obtains the corresponding lock and its associated capability before attempting access. When the code releases the lock, the type system would revoke the capability on the data, just as it revokes a capability after a region is freed. Flanagan and Abadi [13] have investigated this idea in the context of a high-level lexically-scoped language. Just as we compiled Tofte and Talpin's high-level region language into the Capability Calculus, we conjecture we could compile Flanagan and Abadi's locking language into a variant of the Capability Calculus with locking primitives instead of allocation primitives.

A third application of capabilities is to control and reason about aliasing on a per-object basis rather than a per-region basis. Smith, Walker, and Morrisett [45] have investigated the idea of associating a different capability with each individual object and including the type of the object within the capability itself. When code possesses the unique capability for an object, it may deallocate the object, or, if it chooses, it may explicitly reuse the space for that object to store a value of a different type. This new design may be viewed as an extension to conventional linear type systems [16, 50] as it admits aliasing and yet allows safe deallocation of objects. Recently, these techniques have been used to extend the Typed Assembly Language implementation [29] with operations for explicit, but verifiably safe memory management.

In general, whenever a system wishes to restrict access to some data statically, and/or to ensure a certain sequence of operations are performed, it may consider using capabilities. In fact, Walker [52]

has shown that the combination of capabilities and a simple logic are sufficiently powerful to encode any property that can be enforced using a security automaton. Since security automata can specify any *safety property* [2, 44], the extended Capability Calculus can too.

5.2 Related Work

There are many formalisms for reasoning about computational effects in programming languages including type-and-effects systems [15, 25, 21, 47], monads [28, 38, 23, 11], linear types [16, 50, 49], and now capabilities. Many researchers are actively investigating the relationships between these different areas, but the overall picture is not yet fully understood. We are eager to continue this line of research and explore the formal links between our system and the others.

Our translation of Tofte and Talpin's region calculus into the Capability Calculus reveals that the relationship between effects and capabilities is quite close. A necessary prerequisite for the use of either system is type inference, performed by a programmer or compiler, and much of the research into effects systems has concentrated on this difficult task. However, because of the focus on inference, effect systems are usually formulated as a bottom-up synthesis of effects. Our work may viewed as producing verifiable evidence of the correctness of an inference. Hence, while effect systems typically work bottom-up, specifying the effects that might occur, we take a top-down approach, specifying by capabilities the effects that are *permitted to occur*. Moreover, unlike Tofte and Talpin's effect system, our capabilities are sensitive to control-flow. Rather than constructing the overall effect of an expression by taking the union of the effects of the subexpressions, and thereby losing information about the order of evaluation, we verify that programs are safe by checking one instruction after another and using the capability produced by previous instructions to verify that the instructions that follow are safe.

A connection can also be drawn between capabilities and monadic type systems. Work relating effects to monads has viewed effectful functions as pure functions that return state transformers. This might be called an *ex post* view: the effect takes place after the function's execution. In contrast, we take an *ex ante* view in which the capability to perform the relevant effect must be satisfied *before* the function's execution. Nevertheless, there is considerable similarity between the views; just as the monad laws ensure that the store is single-threaded through a computation, our typing rules thread a capability (which summarizes aspects of the store) along the execution path of a program.

Perhaps the closest relationship occurs between linear type systems and capabilities. An inspection of the axioms of capability equality reveals that they are very similar to the structural rules of linear type systems (See Wadler [50] for an introduction to linear logic). In particular, linear assumptions, like unique capabilities, do not allow contraction and weakening rules whereas non-linear assumptions, like duplicatable capabilities, do allow contraction and weakening rules.[4] One essential difference between the two formalisms is that the capability to access an object (say, $\{\rho^1\}$) is separated from the type of the object itself (say, $\langle \texttt{int} \rangle \texttt{ at } \rho$). This level of indirection makes it possible to allow aliasing and yet verify that deallocation is still safe, operations that are not permissible in conventional linear type systems.

There has also been a significant amount of prior research on the more specific topic of the theory and implementation of region-based memory management. On the implementation side, Birkedal *et*

[4]Many formulations of linear logic admit a weakening rule that allows an assumption to be completely forgotten. As explained earlier, we do not allow complete forgetting of capabilities because it leads to space leaks. Instead, we admit a more restrictive weakening rule that allows all but the last capability to be forgotten.

al. [6] describe several optimizations to the basic region-allocation scheme that are used in the ML Kit with Regions to improve space-efficiency. One of their observations is that functions can be used in two different contexts: one context in which no live object remains in a region after a function call and a second context in which there may be live objects remaining in a region after a call. In order to avoid code duplication and yet ensure efficient space usage, the call site passes information to the called function at run time. Using this information, the function may make dynamic decisions about region deallocation. The type system we present here is not powerful enough to encode these *storage-mode polymorphic* functions. However, we believe these dynamic tests may be viewed as form of intensional type analysis [18, 8], and, therefore, if we augment the Capability Calculus with a variant of Harper and Morrisett's typecase mechanism, we may be able to verify the results of storage-mode optimizations as well.

Aiken *et al.* [1] have also studied how to optimize the original Tofte-Talpin region framework. As in the Capability Calculus, they separate region allocation from region deallocation. However, they have not presented a technique for verifying that the results of their optimizations are safe. We conjecture, based on the soundness proof for Aiken *et al.*'s analyses, that the analysis could be used to produce typing annotations and that verification could take place using the Capability Calculus.

Gay and Aiken [14] have developed extensions to C that gives programmers complete control over region allocation and deallocation. They use reference counting to prevent programmers from accidentally accessing deallocated regions. Hawblitzel and von Eicken [20] have also used the notion of a region in their language Passport to support sharing and revocation between multiple protection domains. Both of these groups use run-time checking to ensure safety and it would be interesting to investigate hybrid systems that combine features of our static type system with more dynamic systems.

On the theory side, we believe that one of our most important contributions is the syntactic proof of the soundness of region deallocation. As mentioned earlier, our formulation of the Capability Calculus allows us to use the standard proof techniques popularized by Wright and Felleisen [54]. Although long, the proof requires very little creativity. In contrast, Tofte and Talpin [48] use a greatest fixed-point construction and a co-inductive argument to prove the correctness of their region-inference scheme. Despite these high-level differences between the proof techniques, there are illuminating similarities in some of the details of the proof. Most notably, Tofte and Talpin's proof involves a notion of *consistency* that relates source and target values in the region inference translation. Consistency is defined with respect to the effect (ψ) of the rest of the computation. Informally, one of the consistency conditions states that a source value is consistent with a target value in region ρ, with respect to effect ψ, if ρ does not appear in ψ. Hence, if ρ is not in the effect, or *capability*, of the rest of the computation, then we can deallocate that region because the rest of the computation cannot distinguish a dangling pointer into ρ from a value in the source language. Therefore, within the Tofte-Talpin proof, the effect of the rest of the computation plays a role very similar to the capability required by a continuation. So perhaps the key reason that our language admits a syntactic proof of soundness is that it makes continuations and their capabilities explicit in the language whereas Tofte and Talpin introduce this idea as a meta-level construction in their proof.

6 Conclusions

We have presented a new strongly typed language that admits operations for explicit allocation and deallocation of data structures. Furthermore, this language is expressive enough to serve as a target for region inference and admits a relatively straightforward proof of soundness. We believe that the notion of capabilities that support statically checkable attenuation, amplification, and revocation is an effective new tool for language designers.

7 Acknowledgements

We would like to thank Lars Birkedal, Martin Elsman, Dan Grossman, Chris Hawblitzel, Fred Smith, Mads Tofte, Stephanie Weirich, Steve Zdancewic, and the anonymous reviewers for their comments and suggestions on earlier versions of this work.

References

[1] Alexander Aiken, Manuel Fähndrich, and Raph 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, California, 1995.

[2] Bowen Alpern and Fred Schneider. Recognizing safety and liveness. *Distributed Computing*, 2:117–126, 1987.

[3] H. G. Baker. List processing in real-time on a serial computer. *Communications of the ACM*, 21(4):280–294, 1978.

[4] Brain Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Sirer, Marc Fiuczynski, David Becker, Craig Chambers, and Susan Eggers. Extensibility, safety and performance in the SPIN operating system. In *Fifteenth ACM Symposium on Operating Systems Principles*, pages 267–284, Copper Mountain, December 1995.

[5] Lars Birkedal, Nick Rothwell, Mads Tofte, and David N. Turner. The ML Kit (version 1). Technical Report 93/14, Department of Computer Science, University of Copenhagen, 1993.

[6] Lars Birkedal, Mads Tofte, and Magnus Vejlstrup. From region inference to von Neumann machines via region representation inference. In *Twenty-Third ACM Symposium on Principles of Programming Languages*, pages 171–183, St. Petersburg, January 1996.

[7] Guy Blelloch and John Greiner. A provably time and space efficient implementation of NESL. In *ACM International Conference on Functional Programming*, pages 213–225, June 1996.

[8] Karl Crary, Stephanie Weirich, and Greg Morrisett. Intensional polymorphism in type-erasure semantics. In *ACM International Conference on Functional Programming*, pages 301–312, Baltimore, September 1998.

[9] Olivier Danvy, Belmina Dzafic, and Frank Pfenning. On proving syntactic properties of CPS programs. In *Third International Workshop on Higher-Order Operational Techniques in Semantics*, September 1999. Published in Alan Gordon and Andrew Pitts, editors, *Electronic Notes in Computer Science*, volume 26, pages 19-31. 1999.

[10] Olivier Danvy and Andrzej Filinski. Representing control: a study of the CPS transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, December 1992.

[11] Andrzej Filinski. *Controlling Effects.* PhD thesis, Carnegie Mellon University, School of Computer Science, Pittsburgh, Pennsylvania, May 1996.

[12] M. J. Fischer. Lambda calculus schemata. In *Proceedings of the ACM Conference on Proving Assertions about Programs*, pages 104–109, 1972.

[13] Cormac Flanagan and Martin Abadi. Types for safe locking. In *Proceedings of the Eighth European Symposium on Programming*, pages 91–108, March 1999.

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

[15] D. K. Gifford and J. M. Lucassen. Integrating functional and imperative programming. In *ACM Conference on Lisp and Functional Programming*, Cambridge, Massachusetts, August 1986.

[16] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[17] Robert Harper and Mark Lillibridge. Explicit polymorphism and CPS conversion. In *Twentieth ACM Symposium on Principles of Programming Languages*, pages 206–219, Charleston, January 1993.

[18] Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *Twenty-Second ACM Symposium on Principles of Programming Languages*, pages 130–141, San Francisco, January 1995.

[19] Chris Hawblitzel, Chi-Chao Chang, Grzegorz Czajkowski, Deyu Hu, and Thorsten von Eicken. Implementing multiple protection domains in Java. In *1998 USENIX Annual Technical Conference*, New Orleans, June 1998.

[20] Chris Hawblitzel and Thorsten von Eicken. Sharing and revocation in a safe language. Unpublished manuscript., 1998.

[21] Pierre Jouvelot and D. K. Gifford. Algebraic reconstruction of types and effects. In *Eighteenth ACM Symposium on Principles of Programming Languages*, pages 303–310, January 1991.

[22] Dexter Kozen. Efficient code certification. Technical Report TR98-1661, Cornell University, January 1998.

[23] John Launchbury and Simon L. Peyton Jones. State in Haskell. *LISP and Symbolic Computation*, 8(4):293–341, December 1995.

[24] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.

[25] John M. Lucassen. *Types and Effects—Towards the Integration of Functional and Imperative Programming*. PhD thesis, MIT Laboratory for Computer Science, 1987.

[26] Y. Minamide. Space-profiling semantics of the call-by-value lambda calculus and the CPS transformation. In *Third International Workshop on Higher-Order Operational Techniques in Semantics*, September 1999. Published in Alan Gordon and Andrew Pitts, editors, *Electronic Notes in Computer Science*, volume 26, pages 103-118. 1999.

[27] 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, January 1996.

[28] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93:55–92, 1991.

[29] Greg Morrisett, Karl Crary, Neal Glew, Dan Grossman, Richard Samuels, Frederick Smith, David Walker, Stephanie Weirich, and Steve Zdancewic. Typed Assembly Language for the Intel IA32 architecture. See http://www.cs.cornell.edu/talc for the latest implementation.

[30] Greg Morrisett, Matthias Felleisen, and Robert Harper. Abstract models of memory management. In *ACM Conference on Functional Programming and Computer Architecture*, pages 66–77, La Jolla, June 1995.

[31] Greg Morrisett and Robert Harper. Semantics of memory management for polymorphic languages. In A.D. Gordon and A.M. Pitts, editors, *Higher Order Operational Techniques in Semantics*, Publications of the Newton Institute. Cambridge University Press, 1997.

[32] Greg Morrisett and Robert Harper. Typed closure conversion for recursively-defined functions (extended abstract). In Carolyn Talcott, editor, *Higher-Order Techniques in Operational Semantics*, volume 10 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1998.

[33] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to Typed Assembly Language. In *Twenty-Fifth ACM Symposium on Principles of Programming Languages*, pages 85–97, San Diego, January 1998.

[34] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to Typed Assembly Language. *ACM Transactions on Progamming Languages and Systems*, 21(3):528–569, May 1999.

[35] George Necula. Proof-carrying code. In *Twenty-Fourth ACM Symposium on Principles of Programming Languages*, pages 106–119, Paris, 1997.

[36] George Necula and Peter Lee. Safe kernel extensions without run-time checking. In *Proceedings of Operating System Design and Implementation*, pages 229–243, Seattle, October 1996.

[37] George Necula and Peter Lee. The design and implementation of a certifying compiler. In *ACM Conference on Programming Language Design and Implementation*, pages 333 – 344, Montreal, June 1998.

[38] Simon L. Peyton Jones and Philip Wadler. Imperative functional programming. In *Twentieth ACM Symposium on Principles of Programming Languages*, Charleston, South Carolina, January 1993.

[39] G. D. Plotkin. Call-by-name, call-by-value, and the lambda calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[40] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Conference Record of the 25th National ACM Conference*, pages 717–740, Boston, August 1972.

[41] John C. Reynolds. Syntactic control of interference. In *Fifth ACM Symposium on Principles of Programming Languages*, pages 39–46, Tucson, Arizona, 1978.

[42] John C. Reynolds. Syntactic control of interference, part 2. In *Sixteenth International Colloquium on Automata, Languages, and Programming*, July 1989.

[43] Amr Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. *LISP and Symbolic Computation*, 6(3/4):289–360, 1993.

[44] Fred Schneider. Enforceable security policies. Technical Report TR98-1664, Cornell University, January 1998.

[45] Frederick Smith, David Walker, and Greg Morrisett. Alias types. Technical Report TR99-1773, Cornell University, October 1999.

[46] Mads Tofte and Lars Birkedal. A region inference algorithm. *Transactions on Programming Languages and Systems*, November 1998. To appear.

[47] Mads Tofte and Jean-Pierre Talpin. Implementation of the typed call-by-value λ-calculus using a stack of regions. In *Twenty-First ACM Symposium on Principles of Programming Languages*, pages 188–201, Portland, Oregon, January 1994.

[48] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.

[49] Philip Wadler. Linear types can change the world! In M. Broy and C. Jones, editors, *Programming Concepts and Methods*, Sea of Galilee, Israel, April 1990. North Holland. IFIP TC 2 Working Conference.

[50] Philip Wadler. A taste of linear logic. In *Mathematical Foundations of Computer Science*, volume 711 of *LNCS*, Gdansk, Poland, August 1993. Springer-Verlag.

[51] Robert Wahbe, Steven Lucco, Thomas Anderson, and Susan Graham. Efficient software-based fault isolation. In *Fourteenth ACM Symposium on Operating Systems Principles*, pages 203–216, Asheville, December 1993.

[52] David Walker. A type system for expressive security policies. In *Twenty-Seventh ACM Symposium on Principles of Programming Languages*, January 2000. To appear.

[53] Paul R. Wilson. Uniprocessor garbage collection techniques. In Yves Bekkers and Jacques Cohen, editors, *International Workshop on Memory Management*, number 637 in Lecture Notes in Computer Science, pages 1–42, St. Malo, September 1992. Springer-Verlag.

[54] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

[55] W. A. Wulf, R. Levin, and S. P. Harbison. *Hydra/C.mmp: An Experimental Computer System*. McGraw-Hill, New York, NY, 1981.

A Soundness of The Capability Calculus

Notation The capability $\{r^+\}$ is a derived form that we used for expository purposes in the article. It is equivalent to $\overline{\{r^1\}}$. For the sake of simplicity, the proof operates on a new language that does not include capabilities of the form $\{r^+\}$. The syntax of capabilities is:

$$C \quad ::= \quad \epsilon \mid \emptyset \mid \{r\} \mid C_1 \oplus C_2 \mid \overline{C}$$

The form $\{r\}$ is the new syntax for unique capabilities. The only way to form a duplicatable capability is to use the bar operator as in $\overline{\{r\}}$, hence multiplicity annotations are unnecessary. The rule eq-flag is a derived rule. If the abbreviation $\{r^+\}$ is replaced by its definition, it is clear the rule is simply a special case of reflexivity, and therefore, we do not need it in our system. Where convenient, we continue to use $\{r^+\}$ as an abbreviation for $\overline{\{r\}}$.

We also use the following notational conventions:

- Alpha-equivalent expressions are considered identical.
- Memories, memory regions, memory types, and region types that differ only in the order of their fields are considered identical.
- The expression $E[E'/X]$ denotes the capture-avoiding substitution of E' for X in E.
- Updates of finite maps M are denoted by $M\{X \mapsto E\}$ or $M\{X{:}E\}$.
- Juxtaposition of two maps M and MN as in MN denotes an update of the first with the elements of the second.
- The notation $M \backslash X$ excludes X from the domain of map M.
- We abbreviate $M(\nu)(\ell)$ by $M(\nu.\ell)$.
- We abbreviate $M\{\nu \mapsto M(\nu)\{\ell \mapsto E\}\}$ by $M\{\nu.\ell \mapsto E\}$.
- We abbreviate $(\cdots((\emptyset \oplus \{r_1^{\varphi_1}\}) \oplus \{r_2^{\varphi_2}\})\cdots) \oplus \{r_n^{\varphi_n}\}$ by $\{r_1^{\varphi_1}, \ldots, r_n^{\varphi_n}\}$.
- We abbreviate $(\cdots((\emptyset \oplus C_1) \oplus C_2)\cdots) \oplus C_n$ by $C_1 \oplus \cdots \oplus C_n$.
- We use the notation $C \in C'$ to denote the fact that C appears in the syntax of C'.

Overview The proof is broken down into a series of lemmas, most of which are proven by induction on the typing derivations or by induction on the syntax of the language. The proof culminates in a proof Type Soundness and Complete Collection. The supporting lemmas are grouped as follows:

- Lemmas 10 to 12 describe when extensions to type contexts or exchanges of elements within a type context are permissible.
- Lemmas 13 to 15 state that constructors involved in equality and subtyping judgements are well-formed and that all free variables of well-formed constructors are bound by the type context.
- Definitions and lemmas 16 to 23 describe which capabilities are equal to one another and which capabilities are subtypes of one another. They provide a higher level of abstraction than the rules for equality and subtyping and are used frequently in the rest of the proof.
- Lemmas 24 and 25 are substitution lemmas for types and values respectively.
- Lemma 26 states that well-formed small values, heap values, and declarations have well-formed types.
- Lemmas 27 to 29 are Canonical Forms lemmas. Given a type, these lemmas describe the shape of memory or of values.
- Lemmas 30 to 32 describe the conditions under which you can add labels or regions to the memory type and preserve typing.

- Lemma 34 states that satisfiability is preserved across equality and subtyping (under the empty context).
- Lemma 35 states that satisfiability is preserved when a region and the corresponding unique capability are removed both from memory and the current capability simulataneously.
- Lemmas 36 and 37 are the Preservation and Progress lemmas respectively. They are used directly in the proof of Type Soundness.

Lemma 10 *If* $\Delta \vdash \Delta'$ *then* $Dom(\Delta) \cap Dom(\Delta') = \emptyset$.

Proof:

By induction on the derivation.

□

Lemma 11 (Type Context Exchange) *If* $Dom(\Delta_1) \cap Dom(\Delta_2) = \emptyset$ *then*

1. *If* $\Delta_0\Delta_1\Delta_2\Delta_3 \vdash \Delta$ *then* $\Delta_0\Delta_2\Delta_1\Delta_3 \vdash \Delta$
2. *If* $\Delta_0\Delta_1\Delta_2\Delta_3 \vdash c : \kappa$ *then* $\Delta_0\Delta_2\Delta_1\Delta_3 \vdash c : \kappa$

Proof:

By induction on the derivations. In the rule type-var:

$$\frac{}{\Delta_0\Delta_1\Delta_2\Delta_3 \vdash \alpha : \kappa}\ (\Delta_0\Delta_1\Delta_2\Delta_3(\alpha) = \kappa)$$

we know $\Delta_0\Delta_1\Delta_2\Delta_3(\alpha) = \Delta_0\Delta_2\Delta_1\Delta_3(\alpha)$ because the domains of Δ_1 and Δ_2 are disjoint. Consequently, $\Delta_0\Delta_2\Delta_1\Delta_3 \vdash \alpha : \kappa$.

□

Lemma 12 (Type Context Extension) *If* $\Delta \vdash \Delta'$ *then*

1. *If* $\Delta \vdash \Delta''$ *and* $Dom(\Delta'') \cap Dom(\Delta') = \emptyset$ *then* $\Delta\Delta' \vdash \Delta''$
2. *If* $\Delta \vdash c : \kappa$ *then* $\Delta\Delta' \vdash c : \kappa$
3. *If* $\Delta \vdash c_1 = c_2 : \kappa$ *then* $\Delta\Delta' \vdash c_1 = c_2 : \kappa$
4. *If* $\Delta \vdash c_1 \leq c_2 : \kappa$ *then* $\Delta\Delta' \vdash c_1 \leq c_2 : \kappa$

Proof:

By induction on the derivation. Almost all cases follow directly from the inductive hypothesis. Rules ctxt-sub and type-arrow require Type Context Exchange where Δ_3 is $\cdot$. In part 2, the rules type-var and type-sub, and in part 3, the congruence rules for type variables α, and in part 4, the rule sub-var follow because, by reflemma:disjoint-domains, $Dom(\Delta) \cap Dom(\Delta') = \emptyset$ and thus $\Delta(\alpha) = \Delta\Delta'(\alpha)$.

□

Lemma 13 *If $\Delta \vdash c : \kappa$ then $ftv(c) \subseteq Dom(\Delta)$.*

Proof: By induction on the derivation.

Lemma 14 (Equality Regularity) *If $\Delta \vdash C = C' : \kappa$ then $\Delta \vdash C : \kappa$ and $\Delta \vdash C' : \kappa$.*

Proof: By induction on the derivation.

Lemma 15 (Subtyping Regularity) *If $\cdot \vdash \Delta$ and $\Delta \vdash C \leq C' : \kappa$ then $\Delta \vdash C : \kappa$ and $\Delta \vdash C' : \kappa$.*

Proof:

By induction on the derivation. In the rule sub-var, we show by induction on the derivation $\cdot \vdash \Delta$, that if $\Delta(\epsilon) = \leq C$ then $\Delta \vdash \Delta(\epsilon) : \mathtt{Cap}$.

□

Definition 16 (Atomic Element) *An atomic element, a, is a type variable ϵ of kind* Cap*, a singleton capability $\{r\}$, or a barred capability $\overline{\epsilon}$ or $\overline{\{r\}}$. The meta-variable a ranges over atomic elements.*

Definition 17 *$E(C)$ is the set of elements ϵ or $\{r\}$ that appear in C (where $\{\!|x_1, \ldots, x_n|\!\}$ is notation for the* set *of elements $x_1, \ldots, x_n$):*

$$\begin{array}{rcl} E(\emptyset) & = & \emptyset \\ E(\{r\}) & = & \{\!|\{r\}|\!\} \\ E(\epsilon) & = & \{\!|\epsilon|\!\} \\ E(C_1 \oplus C_2) & = & E(C_1) \cup E(C_2) \\ E(\overline{C}) & = & E(C) \end{array}$$

Lemma 18 (Equality) *If $\Delta \vdash C : \mathtt{Cap}$ then*

1. *$\Delta \vdash C = a_1 \oplus \cdots \oplus a_n : \mathtt{Cap}$ for some atomic capabilities $a_1, \ldots, a_n$*
2. *$\Delta \vdash a_1 \oplus \cdots \oplus a_{i-1} \oplus a_i \oplus a_{i+1} \oplus \cdots \oplus a_n = a'_1 \oplus \cdots \oplus a_{i-1} \oplus a_{i+1} \oplus \cdots \oplus a_n \oplus a_i : \mathtt{Cap}$*
3. *$\Delta \vdash a_1 \oplus \cdots \oplus a_n = a'_1 \oplus \cdots \oplus a'_n : \mathtt{Cap}$ where $a'_1, \ldots, a'_n$ is any permutation of $a_1, \ldots, a_n$.*
4. *$\Delta \vdash a_1 \oplus \cdots \oplus a_n = a'_1 \oplus \cdots \oplus a'_m : \mathtt{Cap}$ where $a'_1, \ldots, a'_m$ is a subsequence of $a_1, \ldots, a_n$ with all duplicate barred elements removed.*
5. *If $\Delta \vdash C = C' : \mathtt{Cap}$ then the sets $E(C)$ and $E(C')$ are equal.*
6. *If $E(C) = E(C')$ and $\Delta \vdash C' : \mathtt{Cap}$ then $\Delta \vdash \overline{C} = \overline{C'} : \mathtt{Cap}$.*
7. *If $\Delta \vdash C \oplus \{r\} = C' \oplus \{r\} : \mathtt{Cap}$ then $\Delta \vdash C = C' : \mathtt{Cap}$.*

Proof:

Part 1 follows by induction on the derivation $\Delta \vdash C : \texttt{Cap}$. Case type-$\emptyset$ is immediate. Case type-single, follows from application of the equality rules eq-symm and eq-$\emptyset$. Case type-plus is more intricate. The inductive hypothesis gives us:

$$\Delta \vdash C_1 = a_1 \oplus \cdots \oplus a_n : \texttt{Cap}$$
$$\Delta \vdash C_2 = a'_1 \oplus \cdots \oplus a'_m : \texttt{Cap}$$

By induction on m and using the rules eq-$\emptyset$, eq-assoc, and eq-trans

$$\Delta \vdash a'_1 \oplus \cdots \oplus a'_m = a'_1 \oplus (a'_2 \oplus \cdots \oplus (a'_{m-1} \oplus a'_m) \cdots) : \texttt{Cap}$$

By equality congruence and eq-trans,

$$\Delta \vdash C_1 \oplus C_2 = (a_1 \oplus \cdots \oplus a_n) \oplus a'_1 \oplus (a'_2 \oplus \cdots \oplus (a'_{m-1} \oplus a'_m) \cdots) : \texttt{Cap}$$

By induction on m again and using eq-assoc, eq-symm, and eq-trans,

$$\Delta \vdash C_1 \oplus C_2 = a_1 \oplus \cdots \oplus a_n \oplus a'_1 \oplus \cdots \oplus a'_m : \texttt{Cap}$$

For the case $\overline{C}$, we have $\Delta \vdash C = a_1 \oplus \cdots \oplus a_n : \texttt{Cap}$ by IH. By congruence, $\Delta \vdash \overline{C} = \overline{a_1 \oplus \cdots \oplus a_n} : \texttt{Cap}$. By induction on n, $\Delta \vdash \overline{C} = \overline{a_1} \oplus \cdots \oplus \overline{a_n} : \texttt{Cap}$. For each a_i, either $\overline{a_i}$ is an atomic element or a_i is already barred and we use the eq-bar-idem rule to show that $\Delta \vdash \overline{a_i} = a_i : \texttt{Cap}$. In either case, by induction on n again and use of the congruence rules, we are done.

Part 2 follows by induction on $m - i$ using eq-assoc, eq-comm as well as the transitivity and symmetry of equality. Part 3 is a corollary of part 2. Part 4 follows by induction on the number of barred duplicates and uses part 3, transitivity, symmetry, and eq-dup rules. Part 5 follows by induction on the equality judgement.

Part 6 may be proven as follows:
$\Delta \vdash C = a_1 \oplus \cdots \oplus a_n : \texttt{Cap}$ where $E(C) = E(a_1 \oplus \cdots \oplus a_n)$ by parts 1 and 5.
$\Delta \vdash C' = a'_1 \oplus \cdots \oplus a'_m : \texttt{Cap}$ where $E(C') = E(a'_1 \oplus \cdots \oplus a'_m)$ by parts 1 and 5.
By parts 3 and 4 and congruence of equality: $\Delta \vdash \overline{C} = \overline{a_1 \oplus \cdots \oplus a_n} = a_{j_1} \oplus a_{j_n} : \texttt{Cap}$
$\Delta \vdash \overline{C'} = \overline{a'_1 \oplus \cdots \oplus a'_m} = a'_{j_1} \oplus a'_{j_m} : \texttt{Cap}$
where the a_{j_i} and a'_{j_i} contain no duplicates and are ordered according to some canonical ordering. If $E(C) = E(C')$ then the a_{j_i} and the a'_{j_i} are the same and are in the same order. Hence, the constructors are syntactically equal and thus definitionally equal.

Part 7 follows by induction on the typing derivation.

□

Definition 19 (Unique/Duplicatable Capabilities) *A capability C is* unique *in C' if there does not exist C'' such that $\overline{C''} \in C'$ and $C \in C''$. A capability C is* duplicatable *in C' if $\overline{C''} \in C'$ and $C \in \overline{C''}$.*

Lemma 20 *If $\Delta \vdash C' : \texttt{Cap}$ and C is duplicatable in C' then $\Delta \vdash C' = C'' \oplus \overline{C} : \texttt{Cap}$.*

Proof: By induction on the typing derivation.

Lemma 21 *If* $\Delta \vdash C' : \texttt{Cap}$ *and* C *is unique in* C' *then* $\Delta \vdash C' = C'' \oplus C : \texttt{Cap}$.

Proof: By induction on the typing derivation.

Lemma 22 (Capability Equality Cardinality Preservation(CECP)) *If* $\Delta \vdash C_1 = C_2 : \texttt{Cap}$ *and* $\Delta \vdash a : \texttt{Cap}$ *and* $a = \epsilon$ *or* $\{r\}$ *then*

1. *a is unique (duplicatable) in* C_1 *iff a is unique (duplicatable) in* C_2.
2. *The number of unique occurences of a is the same in* C_1 *and* C_2.

Proof: By induction on the derivation.

Lemma 23 (Capability Subtyping Cardinality Preservation(CSCP)) *If* $\cdot \vdash C_1 \leq C_2 : \texttt{Cap}$ *then*

1. *For all region names* ν, $\{\nu\} \in C_1$ *iff* $\{\nu\} \in C_2$.
2. *For all region names* ν, *if* $\{\nu\}$ *is not unique in* C_1 *then* $\{\nu\}$ *is not unique in* C_2.

Proof:

By induction on the derivation and Capability Equality Cardinality Preservation. Note that by Subtyping Regularity and Lemma 13, no type variables ϵ appear in C_1 and consequently, the rule sub-var never appears in the derivation.

□

Lemma 24 (Type Substitution)
If Δ' *is* $b_1, \ldots, b_n$ *where for* $1 \leq i \leq n$:

A1. if b_i *is* $\alpha_i{:}\kappa_i$ *then* $\cdot \vdash c_i : \kappa_i$

A2. if b_i *is* $\epsilon_i \leq C_i$ *then* $\cdot \vdash c_i \leq C_i : \texttt{Cap}$

Then when Δ_0 *is* $\Delta[\vec{\tau}/\Delta']$ Γ_0 *is* $\Gamma[\vec{\tau}/\Delta']$ C_0 *is* $C[\vec{\tau}/\Delta']$ r_0 *is* $r[\vec{\tau}/\Delta']$ τ_0 *is* $\tau[\vec{\tau}/\Delta']$:

1. *If* $\Delta', \Delta \vdash \Delta''$ *then* $\Delta_0 \vdash \Delta''[\vec{\tau}/\Delta']$
2. *If* $\Delta', \Delta \vdash c : \kappa$ *then* $\Delta_0 \vdash c[\vec{\tau}/\Delta'] : \kappa$
3. *If* $\Delta', \Delta \vdash c_1 = c_2 : \kappa$ *then* $\Delta_0 \vdash c_1[\vec{\tau}/\Delta'] = c_2[\vec{\tau}/\Delta'] : \kappa$
4. *If* $\Delta', \Delta \vdash c_1 \leq c_2 : \kappa$ *then* $\Delta_0 \vdash c_1[\vec{\tau}/\Delta'] \leq c_2[\vec{\tau}/\Delta'] : \kappa$
5. *If* $\Delta', \Delta \vdash \Delta_1 = \Delta_2$ *then* $\Delta_0 \vdash \Delta_1[\vec{\tau}/\Delta'] = \Delta_2[\vec{\tau}/\Delta']$
6. *If* $\Psi; \Delta', \Delta; \Gamma; r \vdash h : \tau$ *then* $\Psi; \Delta_0; \Gamma_0; r_0 \vdash h[\vec{\tau}/\Delta'] : \tau_0$
7. *If* $\Psi; \Delta', \Delta; \Gamma \vdash v : \tau$ *then* $\Psi; \Delta_0; \Gamma_0 \vdash v[\vec{\tau}/\Delta'] : \tau_0$

8. *If* $\Psi; \Delta', \Delta; \Gamma; C \vdash d \Rightarrow \Delta', \Delta''; \Gamma''; C''$ *then*
   $\Psi; \Delta_0; \Gamma_0; C_0 \vdash d[\vec{\tau}/\Delta'] \Rightarrow (\Delta''; \Gamma''; C'')[\vec{\tau}/\Delta']$.

9. *If* $\Psi; \Delta', \Delta; \Gamma; C \vdash e$ *then* $\Psi; \Delta_0; \Gamma_0; C_0 \vdash e[\vec{\tau}/\Delta']$

Proof:

By induction on the derivations. Almost all cases follow directly from the IH. In part 2, we must prove our lemma for the rules:

$$\frac{}{\Delta', \Delta \vdash \alpha : \kappa}\ (\Delta', \Delta(\alpha) = \kappa) \qquad \frac{}{\Delta', \Delta \vdash \epsilon : \texttt{Cap}}\ ((\epsilon \leq C) \in \Delta', \Delta)$$

In the first case, we have our result by A1 and Type Context Extension. In the second case, assume ϵ is ϵ_i. By A2, we have $\cdot \vdash c_i \leq C_i : \texttt{Cap}$. Because $\cdot \vdash \cdot$, Subtyping Regularity tells us that $\cdot \vdash c_i : \texttt{Cap}$. By Type Context Extension $\Delta_0 \vdash c_i : \texttt{Cap}$. In part 4, for the rule:

$$\frac{}{\Delta', \Delta \vdash \epsilon \leq C : \texttt{Cap}}\ ((\epsilon \leq C) \in \Delta', \Delta)$$

our result follows by A2 and Type Context Extension. In part 9, the case for `let`, we can apply the induction hypothesis because inspection of the rules for declarations show that $\Psi; \Delta; \Gamma; C \vdash d \Rightarrow \Delta, \Delta''; \Gamma''; C'$ instead of the more general $\Psi; \Delta; \Gamma; C \vdash d \Rightarrow \Delta''; \Gamma''; C'$.

□

Lemma 25 (Value Substitution) *If* Γ *is* $\{x_1{:}\tau_1, \ldots, x_n{:}\tau_n\}$, $\cdot \vdash \Gamma$ *and for* $1 \leq i \leq n$, $\Psi; \cdot; \cdot \vdash v_i : \tau_i$ *then*

1. *If* $\Psi; \Delta; \Gamma, \Gamma' \vdash h\ \texttt{at}\ r : \tau$ *then* $\Psi; \Delta; \Gamma' \vdash h[v_1, \ldots, v_n/x_1, \ldots, x_n]\ \texttt{at}\ r : \tau$

2. *If* $\Psi; \Delta; \Gamma, \Gamma' \vdash v : \tau$ *then* $\Psi; \Delta; \Gamma' \vdash v[v_1, \ldots, v_n/x_1, \ldots, x_n] : \tau$

3. *If* $\Psi; \Delta; \Gamma, \Gamma'; C \vdash d \Rightarrow \Delta'; \Gamma, \Gamma''; C'$ *then* $\Psi; \Delta; \Gamma'; C \vdash d[v_1, \ldots, v_n/x_1, \ldots, x_n] \Rightarrow \Delta'; \Gamma''; C'$

4. *If* $\Psi; \Delta; \Gamma, \Gamma'; C \vdash e$ *then* $\Psi; \Delta; \Gamma'; C \vdash e[v_1, \ldots, v_n/x_1, \ldots, x_n]$

Proof:

By induction on the typing derivations. In part 4, the case for `let`, we can use the induction hypothesis because inspection of the typing rules for declarations reveals that $\Psi; \Delta; \Gamma; C \vdash d \Rightarrow \Delta'; \Gamma, \Gamma'; C'$ instead of the more general $\Psi; \Delta; \Gamma; C \vdash d \Rightarrow \Delta'; \Gamma'; C'$.

□

Lemma 26 (Term Judgement Regularity)
If

A1 $\vdash \Psi$

A2 $\cdot \vdash C : \mathtt{Cap}$

A3 $\cdot \vdash r : \mathtt{Rgn}$

then

1. *If* $\Psi;\cdot;\cdot \vdash v : \tau$ *then* $\cdot \vdash \tau : \mathtt{Type}$
2. *If* $\Psi;\cdot;\cdot \vdash h\ \mathtt{at}\ r : \tau$ *then* $\cdot \vdash \tau : \mathtt{Type}$
3. *If* $\Psi;\cdot;\cdot;C \vdash d \Rightarrow \Delta';\Gamma';C'$ *then* $\cdot \vdash \Delta'$ *and* $\Delta' \vdash \Gamma'$ *and* $\Delta' \vdash C' : \mathtt{Cap}$

Proof:

By induction on the typing derivations. Almost all cases follow directly from the induction hypothesis and Equality Regularity or Subtyping Regularity. In part 1, consider the case for type application:

$$\frac{\Psi;\cdot;\cdot \vdash v : \forall[\alpha{:}\kappa,\Delta].(C',\tau_1,\ldots,\tau_n) \to \mathtt{0}\ \mathtt{at}\ r \quad \cdot \vdash c : \kappa}{\Psi;\cdot;\cdot \vdash v[c] : (\forall[\Delta].(C',\tau_1,\ldots,\tau_n) \to \mathtt{0})[c/\alpha]\ \mathtt{at}\ r}$$

By the induction hypothesis, and then inspection of the typing rules for arrow types, we can deduce a judgement of the following form:

$$\frac{\dfrac{\alpha{:}\kappa \vdash \Delta}{\cdot \vdash \alpha{:}\kappa,\Delta} \quad \alpha{:}\kappa,\Delta \vdash C' : \mathtt{Cap} \quad \alpha{:}\kappa,\Delta \vdash \tau_i : \mathtt{Type} \quad (\text{for } 1 \le i \le n)}{\cdot \vdash \forall[\alpha{:}\kappa,\Delta].(C',\tau_1,\ldots,\tau_n) \to \mathtt{0}\ \mathtt{at}\ r : \mathtt{Type}}$$

By Type Substitution, we may deduce that $\cdot \vdash (\forall[\Delta'].(C',\tau_1,\ldots,\tau_n) \to \mathtt{0})[c/\alpha]\ \mathtt{at}\ r$. The second type application rule follows similarly.

□

Lemma 27 (Canonical Memory Forms) *If* $\vdash \{\nu_1 \mapsto R_1,\ldots,\nu_n \mapsto R_n\} : \{\nu_1{:}\Upsilon_1\ldots,\nu_n{:}\Upsilon_n\}$ *then For all* $1 \le i \le n$ *and for all* $\ell \in Dom(\Upsilon_i)$*, either*

1. $\Upsilon_i(\ell)$ *is* $\langle\tau_1,\ldots,\tau_m\rangle\ \mathtt{at}\ \nu_i$ *and* $R_i(\ell) = \langle v_1,\ldots,v_m\rangle$ *and for* $1 \le j \le m$, $\Psi;\cdot;\cdot \vdash v_j : \tau_j$ *or*
2. $\Upsilon_i(\ell)$ *is* $\forall[\Delta].(C,\tau_1,\ldots,\tau_n) \to \mathtt{0}\ \mathtt{at}\ \nu_i$ *and* $R_i(\ell) = \mathtt{fix}\, f[\Delta](C,x_1{:}\tau_1,\ldots,x_n{:}\tau_n).e$ *and* $\Psi;\Delta;\{f{:}\forall[\Delta].(C,\tau_1,\ldots,\tau_n) \to \mathtt{0}\ \mathtt{at}\ \nu_i, x_1{:}\tau_1,\ldots,x_n{:}\tau_n\};C \vdash e$

Proof: By inspection of the typing judgements for memory, regions, and heap values.

Lemma 28 (Canonical Memory Forms II)

1. *If* $\vdash M : \Psi$ *and* $\nu \notin M$ *then* $\nu \notin \Psi$
2. *If* $\Psi \vdash R\ \mathtt{at}\ \nu' : \Upsilon$ *and* $\nu \notin R$ *and* ν' *is not* ν *then* $\nu \notin \Upsilon$
3. *If* $\Psi;\cdot;\cdot \vdash h\ \mathtt{at}\ \nu' : \tau$ *and* ν' *is not* ν *and* $\nu \notin h$ *then* $\nu \notin \tau$

4. *If* $\Psi;\cdot;\cdot \vdash v : \tau$ *and* $\nu \notin v$ *then* $\nu \notin \tau$

Proof: By induction on the typing derivations.

Lemma 29 (Canonical Forms) *If* $\vdash M : \Psi$ *and* $\Psi;\cdot;\cdot \vdash v : \tau$ *then*

1. *If* τ *is* `int` *then* $v = i$.
2. *If* τ *is* ν `handle` *then* $v = \texttt{handle}(\nu)$.
3. *If* τ *is* $\forall[\Delta].(C, \tau_1, \ldots, \tau_n) \to 0 \texttt{ at } \nu$ *then*
   (a) $v = \nu.\ell[c_1, \ldots, c_m]$
   (b) $M(\nu.\ell) = \texttt{fix}\, f[\Delta', \Delta''](C', x_1{:}\tau_1', \ldots, x_n{:}\tau_n').e$
   (c) Δ' *is* $b_1, \ldots, b_m$ *and for* $0 \le i \le m$, *either* b_i *is* $\alpha_i{:}\kappa_i$ *and* $\cdot \vdash c_i{:}\kappa_i$, *or* b_i *is* $\epsilon_i \le C_i$ *and* $\cdot \vdash c_i \le C_i : \texttt{Cap}$.
   (d) $\cdot \vdash \Delta = \Delta''[c_1, \ldots, c_m/\Delta']$, *and* $\Delta \vdash C = C'[c_1, \ldots, c_m/\Delta']$, *and for* $1 \le i \le n$, $\Delta \vdash \tau_i = \tau_i'[c_1, \ldots, c_m/\Delta'] : \texttt{Type}$
   (e) $\Psi; \Delta', \Delta''; \{f{:}\forall[\Delta', \Delta''].(C', \tau_1', \ldots, \tau_n') \to 0 \texttt{ at } \nu, x_1{:}\tau_1', \ldots, x_n{:}\tau_n'\}; C' \vdash e$

   or instead of (b),(c),(d), and (e): $\nu \notin \Psi$.
4. *If* τ *is* $\langle \tau_1, \ldots, \tau_n \rangle \texttt{ at } \nu$ *then*
   (a) $v = \nu.\ell$
   (b) $M(\nu.\ell) = \langle v_1, \ldots, v_n \rangle$
   (c) $\Psi;\cdot \vdash v_i : \tau_i wval$

   or instead of (b),(c): $\nu \notin \Psi$.

Proof:

Part 1, 2 follow by inspection of the typing rules for word values.
Part 3 follows by induction on the derivation, $\Psi;\cdot;\cdot \vdash v : \tau$. By Canonical Memory Forms and inspection of the typing rules for word values, either $\nu.\ell$ or one of the type application rules are last:

case $\nu.\ell$ where $\nu \notin \Psi$:
(a) Trivial.

case $\nu.\ell$ where $\nu \in \Psi$:
(a) Trivial.
(b) By Canonical Memory Forms where Δ' is $\cdot$, Δ'' is Δ, C' is C, and for $1 \le i \le n$, τ_i' is τ_i.
(c) Trivial.
(d) Trivial.
(e) By inspection of judgement.

case $v[C]$

$$\frac{\Psi;\cdot;\cdot \vdash v : \forall[\epsilon \leq C_a, \Delta].(C_b, \tau_1, \ldots, \tau_n) \to 0\ \mathtt{at}\ r \qquad \cdot \vdash C \leq C_a : \mathtt{Cap}}{\Psi;\cdot;\cdot \vdash v[C] : (\forall[\Delta].(C_b, \tau_1, \ldots, \tau_n) \to 0)[C/\epsilon]\ \mathtt{at}\ r}$$

By Term Judgement Regularity and Lemma 13, r is ν. The inductive hypothesis is as follows:

(a) $v = \nu.\ell[c_1, \ldots, c_n]$

(b) $M(\nu.\ell) = \mathtt{fix}\, f[\Delta', \epsilon \leq C'_a, \Delta''](C'_b, x_1{:}\tau'_1, \ldots, x_n{:}\tau'_n).e$

(c) Δ' is $b_1, \ldots, b_m$ and for $0 \leq i \leq m$, either b_i is $\alpha_i{:}\kappa_i$ and $\cdot \vdash c_i{:}\kappa_i$, or b_i is $\epsilon_i \leq C_i$ and $\cdot \vdash c_i \leq C_i : \mathtt{Cap}$.

(d) $\cdot \vdash \epsilon \leq C_a, \Delta = (\epsilon \leq C'_a, \Delta'')[c_1, \ldots, c_m/\Delta']$
and $\epsilon \leq C_a, \Delta \vdash C_b = C'_b[c_1, \ldots, c_m/\Delta']$
and for $1 \leq i \leq n$, $\epsilon \leq C_a, \Delta \vdash \tau_i = \tau'_i[c_1, \ldots, c_m] : \mathtt{Type}$

(e) $\Psi; \Delta', \epsilon \leq C_a, \Delta''; \Gamma; C'_b \vdash e$
where $\Gamma = \{f{:}\forall[\Delta', \epsilon \leq C_a, \Delta''].(C'_b, \tau'_1, \ldots, \tau'_n) \to 0\ \mathtt{at}\ \nu, x_1{:}\tau'_1, \ldots, x_n{:}\tau'_n\}$

or instead of (b),(c),(d), and (e), $\nu \notin \Psi$. Thus,

(a) $v[C] = \nu.\ell[c_1, \ldots, c_n, C]$ from IH.
If $\nu \notin \Psi$ then the result follows trivially. Thus assume $\nu \in \Psi$.

(b) By IH.

(c) By IH and the typing judgement which states $\cdot \vdash C \leq C_a : \mathtt{Cap}$.

(d) By IH and Type Substitution.

(e) By IH.

case $v[c]$ Similar.

Part 4 follows by inspection of the typing rules for word values. Notice only the $\nu.\ell$ rule when $\nu \in \Psi$, or the rule for tuples when $\nu \notin \Psi$ can apply. Assuming the former (the latter is trivial), then (a) is immediate and (b), (c) follow by Canonical Memory Forms.

□

Lemma 30 (Memory Type GC) *If* $\vdash \Psi$ *and* Ψ' *is* $\Psi \backslash \nu$ *then*

1. *If* $\Psi; \Delta; \Gamma \vdash h\ \mathtt{at}\ r : \tau$ *then* $\Psi'; \Delta; \Gamma \vdash h\ \mathtt{at}\ r : \tau$
2. *If* $\Psi; \Delta; \Gamma \vdash v : \tau$ *then* $\Psi'; \Delta; \Gamma \vdash v : \tau$
3. *If* $\Psi; \Delta; \Gamma; C \vdash d \Rightarrow \Delta'; \Gamma'; C'$ *then* $\Psi'; \Delta; \Gamma; C \vdash d \Rightarrow \Delta'; \Gamma'; C'$
4. *If* $\Psi; \Delta; \Gamma; C \vdash e$ *then* $\Psi'; \Delta; \Gamma; C \vdash e$
5. *If* $\Psi \vdash R\ \mathtt{at}\ \nu : \Upsilon$ *then* $\Psi' \vdash R\ \mathtt{at}\ \nu : \Upsilon$

Proof:

By induction on the typing derivation. All cases follow directly from IH except the rule:

$$\frac{}{\Psi;\Delta;\Gamma \vdash \nu'.\ell : \tau}\ (\Psi(\nu'.\ell) = \tau)$$

When ν is not ν', this case is trivial so assume ν is ν'. By Canonical Memory Forms, τ is either $\forall[\Delta'].(\epsilon, \tau_1, \ldots, \tau_n) \to 0\ \mathtt{at}\ \nu$ or $\langle \tau_1, \ldots, \tau_n \rangle\ \mathtt{at}\ \nu$. Because $\vdash \Psi$, we have $\cdot \vdash \tau : \mathtt{Type}$. By Type Context Extension, $\Delta \vdash \tau : \mathtt{Type}$. Thus, in either of the above cases, $\Psi;\Delta;\Gamma \vdash \nu'.\ell : \tau$ via one of the two rules for $\nu \notin Dom(\Psi)$.

□

Lemma 31 (Memory Type Extension) *If ν does not appear in Ψ, Δ, Γ, r, h, v, d, e, or R, and Ψ' is $\Psi\{\nu{:}\{\,\}\}$ then*

1. *If $\Psi;\Delta;\Gamma \vdash h\ \mathtt{at}\ r : \tau$ then $\Psi';\Delta;\Gamma \vdash h\ \mathtt{at}\ r : \tau$*
2. *If $\Psi;\Delta;\Gamma \vdash v : \tau$ then $\Psi';\Delta;\Gamma \vdash v : \tau$*
3. *If $\Psi;\Delta;\Gamma;C \vdash d \Rightarrow \Delta';\Gamma';C'$ then $\Psi';\Delta;\Gamma;C \vdash d \Rightarrow \Delta';\Gamma';C'$*
4. *If $\Psi;\Delta;\Gamma;C \vdash e$ then $\Psi';\Delta;\Gamma;C \vdash e$*
5. *If $\Psi \vdash R\ \mathtt{at}\ \nu' : \Upsilon$ then $\Psi' \vdash R\ \mathtt{at}\ \nu' : \Upsilon$*

Proof:

By induction on the typing derivation. In part 2, for the rule:

$$\frac{\Delta \vdash \langle \tau_1, \ldots, \tau_n \rangle\ \mathtt{at}\ \nu' : \mathtt{Type}}{\Psi;\Delta;\Gamma \vdash \nu'.\ell : \langle \tau_1, \ldots, \tau_n \rangle\ \mathtt{at}\ \nu'}\ (\nu' \notin \Psi)$$

ν' is not ν by assumption and thus the result holds and similarly for the analogous rule for arrow types.

□

Lemma 32 (Region Type Extension) *If $\nu \in Dom(\Psi)$, $\ell \notin Dom(\Psi(\nu))$, and Ψ' is $\Psi\{\nu.\ell{:}\tau\}$ then*

1. *If $\Psi;\Delta;\Gamma \vdash h\ \mathtt{at}\ r : \tau$ then $\Psi';\Delta;\Gamma \vdash h\ \mathtt{at}\ r : \tau$*
2. *If $\Psi;\Delta;\Gamma \vdash v : \tau$ then $\Psi';\Delta;\Gamma \vdash v : \tau$*
3. *If $\Psi;\Delta;\Gamma;C \vdash d \Rightarrow \Delta';\Gamma';C'$ then $\Psi';\Delta;\Gamma;C \vdash d \Rightarrow \Delta';\Gamma';C'$*
4. *If $\Psi;\Delta;\Gamma;C \vdash e$ then $\Psi';\Delta;\Gamma;C \vdash e$*
5. *If $\Psi \vdash R\ \mathtt{at}\ \nu' : \Upsilon$ then $\Psi' \vdash R\ \mathtt{at}\ \nu' : \Upsilon$*
6. *If $\Psi \vdash C$ sat then $\Psi' \vdash C$ sat*

Proof:

By induction on the typing derivation.

□

Lemma 33 *If* $\Delta \vdash C_1 \oplus C_2 : \texttt{Cap}$ *and* $\Delta \vdash C_1 \oplus C_2 = a_1 \oplus \cdots \oplus a_n : \texttt{Cap}$ *then* $\Delta \vdash C_1 = a'_1 \oplus \cdots \oplus a'_m : \texttt{Cap}$ *and* $a'_1, \ldots, a'_m$ *is a subset of* $a_1, \ldots, a_n$.

Proof:

By Lemma 18 (1), $\Delta \vdash C_1 = a'_1 \oplus \cdots \oplus a'_m : \texttt{Cap}$. By Lemma 18 (5), $E(a'_1 \oplus \cdots \oplus a'_m) = E(C_1) \subseteq E(C_1 \oplus C_2) = E(a_1 \oplus \cdots \oplus a_n)$. By CECP, a_i is unique (duplicatable) in $a'_1 \oplus \cdots \oplus a'_m$ if a_i is unique (duplicatable) in $a_1 \oplus \cdots \oplus a_n$.

□

Lemma 34 (Capability Satisfiability Preservation)

1. *If* $\Psi \vdash C$ sat *and* $\cdot \vdash C = C' : \texttt{Cap}$ *then* $\Psi \vdash C'$ sat.
2. *If* $\Psi \vdash C$ sat *and* $\cdot \vdash C \leq C' : \texttt{Cap}$ *then* $\Psi \vdash C'$ sat.

Proof:

1. By symmetry and transitivity of equality and inspection of the sat derivation.
2. By induction on the subtyping derivation.

   case equality: From Part 1.

   case transitivity: By IH.

   case $\cdot \vdash \epsilon \leq C$: Does not apply because the context $(\cdot)$ is empty and therefore, by Lemma 13, the two capabilities must not contain any free variables.

   case $\cdot \vdash C \leq \overline{C}$:
   1. Assume $\{\nu_1{:}\Upsilon_1, \ldots, \nu_n{:}\Upsilon_n\} \vdash C$ sat
   2. Hence $\cdot \vdash C = \{\nu_1^{\varphi_1}, \ldots, \nu_n^{\varphi_n}\} : \texttt{Cap}$
   3. and $\nu_i \neq \nu_j$ for $1 \leq i, j \leq n$ and $i \neq j$ from 1
   4. From 2, using rule eq-congruence-bar, $\cdot \vdash \overline{C} = \overline{\{\nu_1^{\varphi_1}, \ldots, \nu_n^{\varphi_n}\}} : \texttt{Cap}$
   5. From 4, by constructor equality rules, $\cdot \vdash \overline{C} = \{\nu_1^{+}, \ldots, \nu_n^{+}\} : \texttt{Cap}$
   6. Hence, from 5,3 we have $\{\nu_1{:}\Upsilon_1, \ldots, \nu_n{:}\Upsilon_n\} \vdash \overline{C}$ sat

   case

$$\frac{\cdot \vdash C_1 \leq C'_1 \ (1) \quad \cdot \vdash C_2 \leq C'_2 \ (2)}{\cdot \vdash C_1 \oplus C_2 \leq C'_1 \oplus C'_2}$$

   3. Assume $\{\nu_1{:}\Upsilon_1, \ldots, \nu_n{:}\Upsilon_n\} \vdash C_1 \oplus C_2$ sat
   4. From 3, $\cdot \vdash C_1 \oplus C_2 = \{\nu_1^{\varphi_1}, \ldots, \nu_n^{\varphi_n}\} : \texttt{Cap}$

5. and $\nu_i \neq \nu_j$ for $1 \leq i, j \leq n$ and $i \neq j$ from 1
6. By 4 and Equality Regularity, C_1,C_2,C_1',C_2' are all well-formed under the empty context $\cdot$.
7. By 6 and Lemma 18(1), $\cdot \vdash C_1 = \{\nu_{1,1}^{\varphi_{1,1}}, \ldots, \nu_{1,n_1}^{\varphi_{1,n_1}}\} : \texttt{Cap}$
8. From 7 and CECP, for $1 \leq i \leq n_1$, $\nu_{1,i}^{\varphi_{1,i}} \in C_1$
9. Hence, we have $\nu_{1,i}^{\varphi_{1,i}} \in C_1 \oplus C_2$
10. and by 4 and CECP again, we can conclude $\nu_{1,i}^{\varphi_{1,i}} \in \{\nu_1^{\varphi_1}, \ldots, \nu_n^{\varphi_n}\}$
11. For $1 \leq i \leq n_1$ and $i \neq j$, if $\nu_{1,i} = \nu_{1,j}$ and $\varphi_{1,i} = \varphi_{1,j} = +$ then by Lemma 18(4), we can eliminate the duplicates and assume without loss of generality that $\{\nu_{1,1}^{\varphi_{1,1}}, \ldots, \nu_{1,n_1}^{\varphi_{1,n_1}}\}$ does not contain duplicates of this form.
12. For some $1 \leq i \leq n_1$ and $i \neq j$, assume (anticipating a contradiction) that $\nu_{1,i} = \nu_{1,j}$ and one of $\varphi_{1,i}$ or $\varphi_{1,j}$ is 1 then by CECP, $\{\nu_{1,i}^{\varphi_{1,i}}\}$ and $\{\nu_{1,j}^{\varphi_{1,j}}\}$ both appear in C_1
13. From 12, we have $\{\nu_{1,i}^{\varphi_{1,i}}\}$ and $\{\nu_{1,j}^{\varphi_{1,j}}\}$ both appear in $C_1 \oplus C_2$
14. and from 13, and CECP, $\{\nu_{1,i}^{\varphi_{1,i}}\}$ and $\{\nu_{1,j}^{\varphi_{1,j}}\}$ both appear in $\{\nu_1^{\varphi_1}, \ldots, \nu_n^{\varphi_n}\}$
15. However, 14 and 12 ($\nu_{1,i} = \nu_{1,j}$) contradict 5, indicating our assumption 12 was false
16. By 15 and 11, we may assume without loss of generality that for $1 \leq i \leq n_1$ and $i \neq j$, $\nu_{1,i} \neq \nu_{1,j}$
17. By 7, 11, and 16, we can conclude $\Psi_1 \vdash C_1$ sat where Ψ_1 is Ψ with domain restricted to $\{\nu_{1,1} \ldots, \nu_{1,n_1}\}$
18. Analogous reasoning and definitions for Ψ_2 yields $\Psi_2 \vdash C_2$ sat
19. From 17 and 1, using the inductive hypothesis, we have $\Psi_1 \vdash C_1'$ sat
20. From 18 and 2, using the inductive hypothesis, we have $\Psi_2 \vdash C_2'$ sat
21. From 19 (20) and inspection of the sat judgement, we know $\cdot \vdash C_1' = \{\nu_{1',1}^{\varphi_{1',1}}, \ldots, \nu_{1',n_1'}^{\varphi_{1',n_1'}}\} :$ $\texttt{Cap}$ for some $\nu_{1',1}^{\varphi_{1',1}}, \ldots, \nu_{1',n_1'}^{\varphi_{1',n_1'}}$ (and we know $\cdot \vdash C_2' = \{\nu_{2',1}^{\varphi_{2',1}}, \ldots, \nu_{2',n_1'}^{\varphi_{2',n_1'}}\} : \texttt{Cap}$ for some $\{\nu_{2',1}^{\varphi_{2',1}}, \ldots, \nu_{2',n_1'}^{\varphi_{2',n_1'}}\}$)
22. Now, by CSCP, 19 (20), and 1, we have that for $1 \leq i \leq n_1'$, $\{\nu_{1',i}^{\varphi_{1',i}}\} \in C_1$ (and that for $1 \leq i \leq n_2'$, $\{\nu_{2',i}^{\varphi_{2',i}}\} \in C_2$)
23. Consequently, from 22 using CECP, $\nu_{1',1}, \ldots, \nu_{1',n_1'} \in \{\nu_1, \ldots \nu_n\}$
and similarly $\nu_{2',1}, \ldots, \nu_{2',n_2'} \in \{\nu_1, \ldots \nu_n\}$.
24. Conversely, also by CECP, $\nu_1, \ldots, \nu_n \in \{\nu_{1',1}, \ldots, \nu_{1',n_1'}, \nu_{2',1}, \ldots, \nu_{2',n_2'}\}$
25. To summarize, 23 and 24 state that region names $\{\nu_{1',1}^{\varphi_{1',1}}, \ldots, \nu_{1',n_1'}^{\varphi_{1',n_1'}}, \nu_{2',1}^{\varphi_{2',1}}, \ldots, \nu_{2',n_1'}^{\varphi_{2',n_1'}}\}$ are an exact cover of $\{\nu_1, \ldots, \nu_n\}$
26. Now, reasoning analogously to steps 12 through 16, we can deduce that $\{\nu_{1',1}^{\varphi_{1',1}}, \ldots, \nu_{1',n_1'}^{\varphi_{1',n_1'}}, \nu_{2',1}^{\varphi_{2',1}}, \ldots, \nu_{2',n_1'}^{\varphi_{2',n_1'}}\}$ contains no duplicate region names aside from those which both have multiplicity flag $+$.
27. By 21 and rule eq-congruence-plus, we have $\cdot \vdash C_1' \oplus C_2' = \{\nu_{1',1}^{\varphi_{1',1}}, \ldots, \nu_{1',n_1'}^{\varphi_{1',n_1'}}\} \oplus \{\nu_{2',1}^{\varphi_{2',1}}, \ldots, \nu_{2',n_1'}^{\varphi_{2',n_1'}}\} : \texttt{Cap}$
28. By Lemma 18(4) we may eliminate duplicate region names with flag $+$ on the right-hand side of equation 27.
29. By Lemma 18(3), we can reorder the elements of the right-hand side of 28 which by 25

and 26 are exactly $\{\nu_1, \ldots, \nu_n\}$ giving us $\cdot \vdash C_1' \oplus C_2' = \{\nu_1^{\varphi_1'}, \ldots, \nu_n^{\varphi_n'}\} : \texttt{Cap}$
32. Consequently, we have $\{\nu_1, \ldots, \nu_n\} \vdash C_1' \oplus C_2'$ sat

□

Lemma 35 *If* $\Psi \vdash C \oplus \{\nu\}$ sat *then* $\Psi \backslash \nu \vdash C$ sat.

Proof:

1. Assume $\Psi \vdash C \oplus \{\nu\}$ sat
2. and $\Psi = \{\nu_1{:}\Upsilon_1, \ldots, \nu_n{:}\Upsilon_n\}$
3. From 1, we know $\cdot \vdash C \oplus \{\nu\} = \{\nu_1^{\varphi_1}, \ldots, \nu_n^{\varphi_n}\} : \texttt{Cap}$
4. and $\nu_i \neq \nu_j$, for $1 \leq i, j \leq n$ and $i \neq j$
5. From 3, 4, and CECP, ν appears once in $\{\nu_1^{\varphi_1}, \ldots, \nu_n^{\varphi_n}\}$ and once in $C \oplus \{\nu\}$.
6. From 5, and Equality (3),
$\cdot \vdash \{\nu_1^{\varphi_1}, \ldots, \nu_{i-1}^{\varphi_{i-1}}, \nu^1, \nu_{i+1}^{\varphi_{i+1}}, \ldots, \nu_n^{\varphi_n}\} = \{\nu_1^{\varphi_1}, \ldots, \nu_{i-1}^{\varphi_{i-1}}, \nu_{i+1}^{\varphi_{i+1}}, \ldots, \nu_n^{\varphi_n}\} \oplus \{\nu\} : \texttt{Cap}$
7. From 3, 6, transitivity of equality, and Equality (7),
$\cdot \vdash C = \{\nu_1^{\varphi_1}, \ldots, \nu_{i-1}^{\varphi_{i-1}}, \nu_{i+1}^{\varphi_{i+1}}, \ldots, \nu_n^{\varphi_n}\} : \texttt{Cap}$
8. Hence, from 2,4,7 we have $\Psi \backslash \nu \vdash C$ sat

□

Lemma 36 (Preservation) *If* $\vdash (M, e)$ *and* $(M, e) \longmapsto (M', e')$ *then* $\vdash (M', e')$

Proof:

The proof proceeds by cases on the structure of e. In each case, we show the form of the typing judgement that can be inferred by inspection of the typing rules and refer to it throughout the case as "the typing judgement". Then we give the transition specified by the operational semantics. Using these two facts, we derive the result $\vdash (M', e')$.

- $\texttt{let}\, v$

$$\dfrac{\vdash M : \Psi \quad \Psi \vdash C \text{ sat} \qquad \dfrac{\dfrac{\Psi; \cdot; \cdot \vdash v : \tau}{\Psi; \cdot; \cdot; C \vdash x = v \Rightarrow \cdot; \{x{:}\tau\}; C} \quad \Psi; \cdot; \{x{:}\tau\}; C \vdash e}{\Psi; \cdot; \cdot; C \vdash \texttt{let}\, x = v\, \texttt{in}\, e}}{\vdash (M, \texttt{let}\, x = v\, \texttt{in}\, e)}$$

  and $(M, \texttt{let}\, x = v\, \texttt{in}\, e) \longmapsto (M, e[v/x])$. By the typing judgement and Value Substitution, $\vdash (M, e[v/x])$.

- $\texttt{let}\, h$

$$\dfrac{\vdash M : \Psi \quad \Psi \vdash C \text{ sat} \qquad \dfrac{\dfrac{\dfrac{\Psi; \cdot; \cdot \vdash v : \nu\ \texttt{handle}}{\Psi; \cdot; \cdot \vdash h\, \texttt{at}\, \nu : \tau \qquad \cdot \vdash C \leq C' \oplus \overline{\{\nu\}}} \quad \dfrac{\vdots}{\Psi; \cdot; \{x{:}\tau\}; C \vdash e}}{\Psi; \cdot; \cdot; C \vdash x = h\, \texttt{at}\, v \Rightarrow \cdot; \{x{:}\tau\}; C}}{\Psi; \cdot; \cdot; C \vdash \texttt{let}\, x = h\, \texttt{at}\, v\, \texttt{in}\, e}}{\vdash (M, \texttt{let}\, x = h\, \texttt{at}\, v\, \texttt{in}\, e)}$$

where $v = \texttt{handle}(\nu)$
and $\nu \in Dom(M)$ and $\ell \notin Dom(M(\nu))$
and $M' = M\{\nu.\ell \mapsto h\}$
and $(M, \texttt{let}\, x = h\, \texttt{at}\, v\, \texttt{in}\, e) \longmapsto (M', e[\nu.\ell/x])$
and let $\Psi' = \Psi\{\nu.\ell{:}\tau\}$

1. (a) τ is $\langle \tau_1, \ldots, \tau_n \rangle\, \texttt{at}\, \nu$ or $\forall[\,].(C'', \tau_1, \ldots, \tau_n) \to 0\, \texttt{at}\, \nu$ by inspection of the heap value typing rules and the typing judgement.
   (b) $\vdash M' : \Psi'$ by the typing judgement and inspection of the memory typing rule.
2. $\Psi' \vdash C$ sat by Region Type Extension.
3. (a) $\Psi'; \cdot; \cdot \vdash \nu.\ell : \tau$ by the typing rules for word values.
   (b) $\Psi'; \cdot; \{\}; C \vdash e[\nu.\ell/x]$ by (a) and Value Substitution.

By 1(b), 2, and 3(b), we have $\vdash (M', e[\nu.\ell/x])$.

- $\pi_i v$

$$\dfrac{\vdash M : \Psi \quad \Psi \vdash C \text{ sat} \qquad \dfrac{\dfrac{\Psi; \cdot; \cdot \vdash v : \langle \tau_1, \ldots, \tau_n \rangle\, \texttt{at}\, \nu \qquad \cdot \vdash C \leq C' \oplus \overline{\{\nu\}}}{\Psi; \cdot; \cdot; C \vdash x = \pi_i v \Rightarrow \cdot; \{x{:}\tau_i\}; C} \quad \Psi; \cdot; \{x{:}\tau_i\}; C \vdash e}{\Psi; \cdot; \cdot; C \vdash \texttt{let}\, x = \pi_i v\, \texttt{in}\, e}}{\vdash (M, \texttt{let}\, x = \pi_i v\, \texttt{in}\, e)}$$

where $v = \nu.\ell$
and $M(\nu.\ell) = \langle v_0, \ldots, v_n \rangle$
and $(M, \texttt{let}\, x = \pi_i v\, \texttt{in}\, e) \longmapsto (M, e[v_i/x])$

1. $\vdash M : \Psi$ by the typing judgement.
2. $\Psi \vdash C$ sat by the typing judgement.
3. (a) $\Psi; \cdot; \cdot \vdash v_i : \tau_i$ by Canonical Forms and the typing judgement.
   (b) $\Psi; \cdot; \cdot C \vdash e[v_i/x]$ by Value Substitution, (a), and the typing judgement.

By 1,2,and 3(b), $\vdash (M, e[v_i/x])$.

- `freergn`

$$\dfrac{\vdash M : \Psi \quad \Psi \vdash C \text{ sat} \qquad \dfrac{\dfrac{\Psi; \cdot; \cdot \vdash v : \nu\ \texttt{handle} \quad \cdot \vdash C \leq C' \oplus \{\nu\}}{\Psi; \cdot; \cdot; C \vdash \texttt{freergn}\, v \Rightarrow \cdot; \cdot; C'} \quad \Psi; \cdot; \cdot; C' \vdash e}{\Psi; \cdot; \cdot; C \vdash \texttt{freergn}\, v\, \texttt{in}\, e}}{\vdash (M, \texttt{freergn}\, v\, \texttt{in}\, e)}$$

where v is $\texttt{handle}(\nu)$ and $(M, \texttt{freergn}\, v\, \texttt{in}\, e) \longmapsto (M \backslash \nu, e)$. Let Ψ' be $\Psi \backslash \nu$.

1. $\vdash M' : \Psi'$ by Memory Type GC and the typing judgement.
2. (a) $\Psi \vdash C$ sat and $\cdot \vdash C \leq C' \oplus \{\nu\}$ by the typing judgement.
   (b) $\Psi \vdash C' \oplus \{\nu\}$ sat by Capability Satisfiability Preservation and (a)

(c) $\Psi' \vdash C'$ sat by Lemma 35 and (b)

3. $\Psi'; \cdot; \cdot; C' \vdash e$ by the typing judgement and Memory Type GC.

By 1, 2(e), and 3, $\vdash (M \backslash \nu, e)$.

- `newrgn`

$$\frac{\vdash M : \Psi \qquad \dfrac{\cdot \vdash C = C' : \mathtt{Cap}}{\Psi \vdash C \text{ sat}}\ (...) \qquad \dfrac{(A) \quad (B)}{\Psi; \cdot; \cdot; C \vdash \mathtt{newrgn}\,\rho, x_\rho\ \mathtt{in}\, e}}{\vdash (M, \mathtt{newrgn}\,\rho, x_\rho\ \mathtt{in}\, e)}$$

$$\frac{}{\Psi; \cdot; \cdot; C \vdash \mathtt{newrgn}\,\rho, x_\rho \Rightarrow \rho{:}\mathtt{Rgn}; \{x_\rho{:}\rho\ \mathtt{handle}\}; C \oplus \{\rho\}}\ (A)$$

$$\frac{\vdots}{\Psi; \rho{:}\mathtt{Rgn}; \{x_\rho{:}\rho\ \mathtt{handle}\}; C \oplus \{\rho\} \vdash e}\ (B)$$

The operational rule is

$$(M, \mathtt{newrgn}\,\rho, x_\rho\ \mathtt{in}\, e) \longmapsto (M', e'[\nu, \mathtt{handle}(\nu)/\rho, x_\rho])$$

where $M' = M\{\nu \mapsto \{\,\}\}$ and $\nu \notin M$ and $\nu \notin e$.
In what follows, let $\Psi' = \Psi\{\nu{:}\{\,\}\}$.

1. $\vdash \Psi'$ by Memory Type Extension and the typing judgement.
2. Since $\nu \notin \Psi$ by assumption in the operational semantics, we can satisfy the side condition on the sat judgement. We can also prove $\cdot \vdash C \oplus \{\nu\} = C' \oplus \{\nu\} : \mathtt{Cap}$ by the congruence rule for equality. Consequently, $\Psi' \vdash C \oplus \{\nu\}$ sat.
3. $\Psi'; \cdot; \cdot; C \oplus \{\nu\} \vdash e[\nu, \mathtt{handle}(\nu)/\rho, x_\rho]$ from the typing judgement and application of Type and Value Substitution and then Memory Type Extension Lemmas.

By 1, 2(e), and 3, $\vdash (M', e[\nu, \mathtt{handle}(\nu)/\rho, x_\rho])$.

- `if0`

$$\frac{\vdash M : \Psi \quad \Psi \vdash C \text{ sat} \qquad \dfrac{\Psi; \cdot; \cdot \vdash i : \mathtt{int} \quad \Psi; \cdot; \cdot; C \vdash e_2 \quad \Psi; \cdot; \cdot; C \vdash e_3}{\Psi; \cdot; \cdot; C \vdash \mathtt{if0}\, i\, \mathtt{then}\, e_2\, \mathtt{else}\, e_3}}{\vdash (M, \mathtt{if0}\, i\, \mathtt{then}\, e_2\, \mathtt{else}\, e_3)}$$

$(M, e) \longmapsto (M, e_2)$ if $i = 0$ and $(M, e) \longmapsto (M, e_3)$ otherwise. By the typing judgement, $\vdash (M, e_2)$, or $\vdash (M, e_3)$.

- $v_0(v_1, \ldots, v_n)$

$$\frac{\vdash M : \Psi \quad \Psi \vdash C \text{ sat} \qquad \dfrac{\begin{array}{c}\Psi; \cdot; \cdot \vdash v_i : \tau_i \quad (\text{for } 0 \le i \le n) \\ \cdot \vdash C \le C'' \oplus \overline{\{\nu\}} : \mathtt{Cap} \\ \cdot \vdash \tau_0 = \forall[\,].(C', \tau_1, \ldots, \tau_n) \to \mathtt{0}\ \mathtt{at}\ \nu \\ \cdot \vdash C \le C' : \mathtt{Cap}\end{array}}{\Psi; \cdot; \cdot; C \vdash v_0(v_1, \ldots, v_n)}}{\vdash (M, v_0(v_1, \ldots, v_n))}$$

$(M, v_0(v_1, \ldots, v_n)) \longmapsto (M, S(e))$
where $w = \nu.\ell[c_1, \ldots, c_m]$
and $M(\nu.\ell) = \texttt{fix}\, f[b_1, \ldots, b_m](C'', x_1 : \tau_1, \ldots, x_n{:}\tau_n).e$
and for $1 \leq i \leq m$, $b_i = \alpha_i{:}\kappa_i$ or $b_i = \alpha_i \leq C_i$
and $S = [c_1, \ldots, c_m, \nu.\ell, v_1, \ldots, v_n/\alpha_1, \ldots, \alpha_m, f, x_1, \ldots, x_n]$

1. $\vdash M : \Psi$ by the typing judgement.
2. $\Psi \vdash C'$ sat by Capability Satisfiability Preservation.
3. (a) $\Psi \vdash C'' \oplus \overline{\{\nu\}}$ sat by Capability Satisfiability Preservation and the typing judgement
   (b) $\nu \in Dom(\Psi)$ by CECP and (a).
   (c) $\Psi; \cdot; \cdot \vdash v_0 : \forall[\,].(C''', \tau_1''', \ldots, \tau_n''') \to \mathtt{0}$ and
   $\cdot \vdash \forall[\,].(C''', \tau_1''', \ldots, \tau_n''') \to \mathtt{0} = \forall[\,].(C', \tau_1, \ldots, \tau_n) \to \mathtt{0}$ by the typing judgement
   (d) $\Psi; b_1, \ldots, b_m; \{x_1{:}\tau_1, \ldots, x_n{:}\tau_n\}; C'' \vdash e$ by Canonical Forms 3(e), (b), and (c).
   (e) $\cdot \vdash C' = C''[c_1, \ldots, c_m/\alpha_1, \ldots, \alpha_m] : \mathtt{Cap}$ by the transitivity of equality, Canonical Forms 3(d), (b), and (c)
   (f) $\Psi; \cdot; \cdot; C' \vdash S(e)$ by Type and Value Substitution, and (e).

By 1, 2, and 3(f), $\vdash (M, S(e))$

□

Lemma 37 (Progress) *If* $\vdash (M, e)$ *then either:*

1. *There exists* (M', e') *such that* $(M, e) \longmapsto (M', e')$, *or*
2. $e = \texttt{halt}\ v$ *and* $\Psi; \cdot; \cdot \vdash v : \texttt{int}$.

Proof:

The proof proceeds by cases on the structure of e and makes heavy use of the Canonical Forms lemma.

- $\texttt{let}\, v$ Trivial.
- $\texttt{let}\, h$

$$\dfrac{\vdash M : \Psi \quad \Psi \vdash C \text{ sat} \qquad \dfrac{\begin{array}{c}\Psi; \cdot; \cdot \vdash v : r\ \texttt{handle} \\ \Psi; \cdot; \cdot \vdash h\ \texttt{at}\ r : \tau \\ \cdot \vdash C \leq C' \oplus \overline{\{r\}}\end{array}}{\Psi; \cdot; \cdot; C \vdash x = h\ \texttt{at}\ v \Rightarrow \cdot; \{x{:}\tau\}; C} \quad \cdots}{\Psi; \cdot; \cdot; C \vdash \texttt{let}\, x = h\ \texttt{at}\ v\ \texttt{in}\, e}}{\vdash (M, \texttt{let}\, x = h\ \texttt{at}\ v\ \texttt{in}\, e)}$$

  $\Psi; \cdot; \cdot \vdash v : r\ \texttt{handle}$ directly from the typing judgement. By Term Judgement Regularity and Lemma 13, r is ν, and by Canonical Forms, v is $\texttt{handle}(\nu)$. By Capability Satisfiability Preservation, $\Psi \vdash C' \oplus \overline{\{\nu\}}$ sat and thus $\nu \in Dom(\Psi)$. By inspection of the memory typing rules, $\nu \in Dom(M)$. Thus $(M, e) \longmapsto (M\{\nu.\ell \mapsto h\}, e[\nu.\ell/x])$.

- $\pi_i v$

$$\dfrac{\vdash M : \Psi \quad \Psi \vdash C \text{ sat} \qquad \dfrac{\dfrac{\Psi;\cdot;\cdot \vdash v : \langle \tau_1, \ldots, \tau_n \rangle \,\mathtt{at}\, r \quad \cdot \vdash C \leq C' \oplus \overline{\{r\}}}{\Psi;\cdot;\cdot; C \vdash x = \pi_i v \Rightarrow \cdot;\{x{:}\tau_i\}; C} \quad \cdots}{\Psi;\cdot;\cdot; C \vdash \mathtt{let}\, x = \pi_i v \,\mathtt{in}\, e}}{\vdash (M, \mathtt{let}\, x = \pi_i v \,\mathtt{in}\, e)}$$

  By Capability Satisfiability Preservation, $\Psi \vdash C' \oplus \overline{\{\nu\}}$ sat. By CECP, $\nu \in Dom(\Psi)$ and by Canonical Forms, $M(\nu.\ell) = \langle v_1, \ldots, v_n \rangle$. By the typing judgement, $0 \leq i \leq n-1$. Thus $(M, \mathtt{let}\, x = \pi_i v \,\mathtt{in}\, e) \longmapsto (M, e[v_i/x])$.

- `newrgn` Trivial.

- `freergn`

$$\dfrac{\vdash M : \Psi \quad \Psi \vdash C \text{ sat} \qquad \dfrac{\dfrac{\Psi;\cdot;\cdot \vdash v : r\ \mathtt{handle} \quad \cdot \vdash C \leq C' \oplus \{r\} : \mathtt{Cap}}{\Psi;\cdot;\cdot; C \vdash \mathtt{freergn}\, v \Rightarrow \cdot;\cdot; C'} \quad \cdots}{\Psi;\cdot;\cdot; C \vdash \mathtt{freergn}\, v \,\mathtt{in}\, e}}{\vdash (M, \mathtt{freergn}\, v \,\mathtt{in}\, e)}$$

  By Term Judgement Regularity and Lemma 13, r is ν, and by Canonical Forms, v is $\mathtt{handle}(\nu)$. By Capability Satisfiability Preservation, $\Psi \vdash C \oplus \{\nu\}$ sat. Thus, by CECP, $\nu \in Dom(\Psi)$, and by inspection of the memory typing rules, $\nu \in Dom(M)$. Consequently, $(M, \mathtt{freergn}\, v \,\mathtt{in}\, e) \longmapsto (M \backslash \nu, e)$.

- `if0`

$$\dfrac{\Psi;\Delta;\Gamma \vdash v : \mathtt{int} \qquad \Psi;\Delta;\Gamma;C \vdash e_2 \qquad \Psi;\Delta;\Gamma;C \vdash e_3}{\Psi;\Delta;\Gamma;C \vdash \mathtt{if0}\, v \,\mathtt{then}\, e_2 \,\mathtt{else}\, e_3}$$

  By Canonical Forms, v must be integer. Therefore, one of the two operational rules for `if0` applies.

- $v_0(v_1, \ldots, v_n)$

$$\dfrac{\vdash M : \Psi \quad \Psi \vdash C \text{ sat} \qquad \dfrac{\begin{array}{c}\Psi;\cdot;\cdot \vdash v_i : \tau_i \quad (\text{for } 0 \leq i \leq n) \\ \cdot \vdash C \leq C'' \oplus \overline{\{r\}} : \mathtt{Cap} \\ \cdot \vdash \tau_0 = \forall[].(C', \tau_1, \ldots, \tau_n) \to 0\,\mathtt{at}\, r \\ \cdot \vdash C \leq C' : \mathtt{Cap}\end{array}}{\Psi;\cdot;\cdot; C \vdash v_0(v_1, \ldots, v_n)}}{\vdash (M, v_0(v_1, \ldots, v_n))}$$

  By Subtyping Regularity and Lemma 13, r is ν. By Capability Satisfiability Preservation, $\Psi \vdash C'' \oplus \overline{\{\nu\}}$ sat, and by CECP, $\nu \in Dom(\Psi)$. Thus, by Canonical Forms, $v = \nu.\ell[c_1, \ldots, c_m]$, $M(\nu.\ell) = \mathtt{fix}\, f[b_1, \ldots, b_m](C, x_1{:}\tau_1, \ldots, x_n{:}\tau_n).e$ and for $1 \leq i \leq n$, $b_i = \alpha_i{:}\kappa_i$ or $b_i = \alpha_i \leq C_i$. If S is $[c_1, \ldots, c_m, \nu.\ell, v_1, \ldots, v_n / \alpha_1, \ldots, \alpha_m, f, x_1, \ldots, x_n]$ then $(M, v_0(v_1, \ldots, v_n)) \longmapsto (M, S(e))$

- `halt`

$$\frac{\vdash M : \Psi \quad \Psi \vdash C \text{ sat} \qquad \dfrac{\Psi;\cdot;\cdot \vdash v : \texttt{int} \quad \cdot \vdash C = \emptyset : \texttt{Cap}}{\Psi;\cdot;\cdot;C \vdash \texttt{halt } v}}{\vdash (M, \texttt{halt } v)}$$

  Part 2 holds by inspection of the typing judgement.

□

Definition 38 (Stuck State) *An abstract machine state* (M, e) *is* stuck *if* e *is not* `halt` v *and there does not exist* (M', e') *such that* $(M, e) \longmapsto (M', e')$.

Theorem 39 (Type Soundness) *If* $\vdash (M, e)$ *and* $(M, e) \longmapsto^* (M', e')$ *then* (M', e') *is not stuck.*

Proof:

By induction on the number of steps taken in the operational semantics and Preservation, if $\vdash (M, e)$ and $(M, e) \longmapsto^* (M', e')$ then $\vdash (M', e')$. By Progress, no well-typed state (M', e') is stuck: either e' is `halt` v or $(M', e') \longmapsto (M'', e'')$.

□

Theorem 40 (Complete Collection) *If* $\vdash (M, e)$ *then either* $(M, e) \Uparrow$ *or* $(M, e) \longmapsto^* (M', \texttt{halt } v)$ *and* $M' = \{\,\}$.

Proof:

Assume $\vdash (M, e)$ and $(M, e) \longmapsto^* (M', e')$ and there is no (M'', e'') such that $(M', e') \longmapsto (M'', e'')$. By Preservation and Progress, $e' = \texttt{halt } v$ and

$$\frac{\vdash M' : \Psi \quad \Psi \vdash C \text{ sat} \qquad \dfrac{\Psi;\cdot;\cdot \vdash v : \texttt{int} \quad \cdot \vdash C = \emptyset : \texttt{Cap}}{\Psi;\cdot;\cdot;C \vdash \texttt{halt } v}}{\vdash (M', \texttt{halt } v)}$$

By CECP and the sat judgement, $\nu \in Dom(\Psi)$ if and only if $\nu \in \emptyset$. Consequently, $\Psi = \{\,\}$. By inspection of the judgement for memory types, $M' = \{\,\}$.

□

B Region Translation Type Preservation

In this section, we prove our translation of the region calculus into the Capability Calculus is type preserving. In other words, given a well-formed source language term, the result of the translation is a well-formed Capability Calculus term. Section 3 describes the syntax and static semantics of the term constructs.

Notation In this proof, we use several abbreviations and conventions. Equality and subtyping judgements for capabilities are written in the form:

$$\begin{array}{rcl} \Delta \vdash C_1 & = & C_2 \\ & \leq & C_3 \\ & = & C_4 \\ & = & \cdots \end{array}$$

Transitivity rules link each of the equations together. We use associativity and commutativity rules without mentioning them and we will substitute a subcomponent of a capability for an equal capability without mentioning the use of the congruence rules.

We use the notation $\Delta \vdash_R \Gamma$ to indicate that all the types in Γ are well-formed under type context Δ. Formally:

$$\frac{}{\Delta \vdash_R \cdot} \qquad \frac{\Delta \vdash_R \Gamma \quad \Delta \vdash_R \tau : \mathtt{Type}}{\Delta \vdash_R \Gamma, x{:}\tau} \; (x \notin Dom(\Gamma))$$

Overview The theorem borrows several lemmas from the proof of soundness including lemmas for manipulating capabilities (Lemma 18) and asserting well-formedness (Lemma 14). We also require a number of suplementary lemmas:

- Lemmas 41 and 42 describe additional well-formedness constraints on the types and effects that appear in region calculus judgements.
- Lemmas 43, 44, and 45 state that well-formedness of constructors, constructor equality, and substitution is preserved across the translation.
- Lemma 48 describes the way the subset relation is preserved during the translation of effects (sets) to capabilities (not sets). This lemma makes use of Lemmas 46 and 47 in its proof.
- Lemma 49 is a miscellaneous lemma required in the proof of the letregion construct.
- Lemma 50 gives the conditions under which the static application of a continuation closure to its arguments (*i.e.* $\mathcal{A}(k, v)$) is well-formed.

Lemma 41

1. *If* $\Delta \vdash_R c_1 = c_2 : \kappa$ *then* $\Delta \vdash_R c_1 : \kappa$ *and* $\Delta \vdash_R c_2 : \kappa$.
2. *If* $\Delta \vdash_R \psi_1 \subseteq \psi_2$ *then* $\Delta \vdash_R \psi_1 : \mathtt{Eff}$ *and* $\Delta \vdash_R \psi_2 : \mathtt{Eff}$

Proof: By induction on the derivations.

Lemma 42 *If* $\Delta \vdash_R \Gamma$ *and* $\Delta; \Gamma \vdash_R e : \tau, \psi$ *then* $\Delta \vdash_R \tau : \mathtt{Type}$ *and* $\Delta \vdash_R \psi : \mathtt{Eff}$

Proof: By induction on the derivation $\Delta; \Gamma \vdash_R e : \tau, \psi$.

Lemma 43 (Well-Formedness Preservation)

1. *If* $\Delta \vdash_R \Delta'$ *then* $\mathcal{K}[\![\Delta]\!] \vdash \mathcal{K}[\![\Delta']\!]$

2. *If* $\Delta \vdash_R c : \kappa$ *then* $\mathcal{K}[\![\Delta]\!] \vdash \mathcal{T}[\![c]\!] : \mathcal{K}[\![\kappa]\!]$

Proof: By induction on the derivations.

Lemma 44 (Equality Preservation)

1. *If* $\Delta \vdash_R \psi = \psi' : \mathtt{Eff}$ *then* $\mathcal{K}[\![\Delta]\!] \vdash \overline{\mathcal{T}[\![\psi]\!]} = \overline{\mathcal{T}[\![\psi']\!]} : \mathtt{Cap}$

2. *If* $\Delta \vdash_R c = c' : \kappa$ *and* $\kappa \neq \mathtt{Eff}$ *then* $\mathcal{K}[\![\Delta]\!] \vdash \mathcal{T}[\![c]\!] = \mathcal{T}[\![c']\!] : \mathtt{Cap}$

Proof: By induction on the equality derivations.

Lemma 45 (Substitution Preservation) *If* $\Delta, \alpha{:}\kappa \vdash_R \tau : \mathtt{Type}$ *and* $\Delta \vdash_R c : \kappa$ *then* $\mathcal{K}[\![\Delta]\!] \vdash \mathcal{T}[\![\tau[c/\alpha]]\!] = \mathcal{T}[\![\tau]\!][\mathcal{T}[\![c]\!]/\alpha]$.

Proof: By induction on the typing derivation.

Lemma 46 *If* $\Delta \vdash \psi = \psi' : \mathtt{Eff}$ *then* $E(\mathcal{T}[\![\psi]\!]) = E(\mathcal{T}[\![\psi']\!])$.

Proof: By induction on the derivation.

Lemma 47 *If* $\Delta \vdash \psi \subseteq \psi'$ *then* $E(\mathcal{T}[\![\psi]\!]) \subseteq E(\mathcal{T}[\![\psi']\!])$.

Proof: By inspection of the sub-effecting rule, $\Delta \vdash \psi \cup \psi'' = \psi' : \mathtt{Eff}$ for some effect ψ''.
By Lemma 46, $E(\mathcal{T}[\![\psi \cup \psi'']\!]) = E(\mathcal{T}[\![\psi']\!])$.
By definition of E and the type translation, we have $E(\mathcal{T}[\![\psi]\!]) \cup E(\mathcal{T}[\![\psi'']\!]) = E(\mathcal{T}[\![\psi']\!])$.
Hence, $E(\mathcal{T}[\![\psi]\!]) \subseteq E(\mathcal{T}[\![\psi']\!])$.

Lemma 48 *If* $\Delta \vdash \psi \subseteq \psi'$ *and* $\Delta \vdash \overline{C} = \overline{C \oplus \mathcal{T}[\![\psi']\!]} : \mathtt{Cap}$ *then* $\Delta \vdash \overline{C} = \overline{C \oplus \mathcal{T}[\![\psi]\!]} : \mathtt{Cap}$.

Proof:

1. By $\Delta \vdash \overline{C} = \overline{C \oplus \mathcal{T}[\![\psi']\!]} : \mathtt{Cap}$ and Equality (5), $E(C) = E(C \oplus \mathcal{T}[\![\psi']\!])$.
2. Thus, by definition of E, $E(\mathcal{T}[\![\psi']\!]) \subseteq E(C)$.
3. By $\Delta \vdash \psi \subseteq \psi'$ and Lemma 47, $E(\mathcal{T}[\![\psi]\!]) \subseteq E(\mathcal{T}[\![\psi']\!])$.
4. By 2 and 3, $E(\mathcal{T}[\![\psi]\!]) \subseteq E(C)$.
5. Hence, by definition of E, $E(C) = E(C \oplus \mathcal{T}[\![\psi]\!])$.
6. By 5 and Equality (6), $\Delta \vdash \overline{C} = \overline{C \oplus \mathcal{T}[\![\psi]\!]} : \mathtt{Cap}$.

□

Lemma 49 *If* $\Delta \vdash_R \psi$ *then* $\mathcal{K}[\![\Delta]\!] \vdash \overline{\mathcal{T}[\![\psi \backslash \{\rho\}]\!] \oplus \{\rho^1\}} = \overline{\mathcal{T}[\![\psi]\!] \oplus \{\rho^1\}} : \mathtt{Cap}$

Proof:

The proof is by induction on the structure of ψ.

- $\psi = \emptyset$. By definition, $\mathcal{T}[\![\emptyset \backslash \{\rho\}]\!] = \emptyset$. The result follow immediately from the reflexivity of equality.
- $\psi = \epsilon$ or $\psi = \{\rho'\}$ and $\rho \neq \rho'$. Similar.
- $\psi = \{\rho\}$. The following reasoning provides the result:

$$\begin{array}{rcll} \mathcal{K}[\![\Delta]\!] \vdash \overline{\mathcal{T}[\![\{\rho\}\backslash\{\rho\}]\!] \oplus \{\rho^1\}} & = & \overline{\emptyset \oplus \{\rho^1\}} & \text{(By definition)} \\ & = & \overline{\{\rho^1\}} & \text{(By rule eq-}\emptyset\text{)} \\ & = & \overline{\{\rho^1\} \oplus \{\rho^1\}} & \text{(By rule eq-dup)} \end{array}$$

- $\psi = \psi_1 \cup \psi_2$. By induction, we know that (1) $\mathcal{K}[\![\Delta]\!] \vdash \overline{\mathcal{T}[\![\psi_i \backslash \{\rho\}]\!] \oplus \{\rho^1\}} = \overline{\mathcal{T}[\![\psi_i]\!] \oplus \{\rho^1\}}$ for $i = 1, 2$. Now, the following reasoning provides the result:

$$\begin{array}{rcll} \mathcal{K}[\![\Delta]\!] \vdash \overline{\mathcal{T}[\![(\psi_1 \cup \psi_2)\backslash\{\rho\}]\!] \oplus \{\rho^1\}} & = & \overline{\mathcal{T}[\![\psi_1\backslash\{\rho\}]\!] \oplus \mathcal{T}[\![(\psi_2\backslash\{\rho\})]\!] \oplus \{\rho^1\}} & \text{(By def.)} \\ & = & \overline{\mathcal{T}[\![\psi_1\backslash\{\rho\}]\!] \oplus \mathcal{T}[\![(\psi_2\backslash\{\rho\})]\!] \oplus \{\rho^1\} \oplus \{\rho^1\}} & \text{(By eq-dup)} \\ & = & \overline{\mathcal{T}[\![\psi_1\backslash\{\rho\}]\!] \oplus \{\rho^1\} \oplus \mathcal{T}[\![(\psi_2\backslash\{\rho\})]\!] \oplus \{\rho^1\}} & \text{(By *)} \\ & = & \overline{\mathcal{T}[\![\psi_1]\!] \oplus \{\rho^1\} \oplus \mathcal{T}[\![\psi_2]\!] \oplus \{\rho^1\}} & \text{(By 1)} \\ & = & \overline{\mathcal{T}[\![\psi_1]\!] \oplus \mathcal{T}[\![\psi_2]\!] \oplus \{\rho^1\} \oplus \{\rho^1\}} & \text{(By *)} \\ & = & \overline{\mathcal{T}[\![\psi_1]\!] \oplus \mathcal{T}[\![\psi_2]\!] \oplus \{\rho^1\}} & \text{(By eq-dup)} \end{array}$$

  where * is rule eq-comm.

□

Lemma 50 *Let* $k = \langle x^k; e^k \rangle$. *If* $\Psi; \Delta; \Gamma, x^k{:}\tau; C \vdash e^k$ *and* $\Psi; \Delta; \Gamma \vdash v : \tau$ *then* $\Psi; \Delta; \Gamma; C \vdash \mathcal{A}(k, v)$.

Proof:

The term $\mathcal{A}(k, v)$ is defined to be $\texttt{let}\, x^k = v\, \texttt{in}\, e^k$. The following derivation proves the lemma:

$$\frac{\Psi; \Delta; \Gamma \vdash v : \tau \quad \Psi; \Delta; \Gamma, x^k{:}\tau; C \vdash e^k}{\Psi; \Delta; \Gamma; C \vdash \texttt{let}\, x^k = v\, \texttt{in}\, e^k}$$

□

Lemma 51 *If* $\{\}; \Delta; \Gamma, x{:}\tau; C \vdash e$ *and* $\Delta \vdash \tau = \tau' : \texttt{Type}$ *then* $\{\}; \Delta; \Gamma, x{:}\tau'; C \vdash e$.

Proof:

The proof is by induction on the typing derivation for expressions.

□

Theorem 52 (CPS Type Preservation)
If $\cdot; \cdot \vdash e : \texttt{int}, \emptyset$ *then* $\{\}; \cdot; \cdot; \emptyset \vdash \mathcal{C}_{\cdot;\cdot;\Theta}(e)\langle x; \texttt{halt}\ x\rangle$ *where* x *is fresh and* Θ *is the empty environment,* $\langle \cdot; \cdot; \emptyset; \emptyset \rangle$.

Proof:

The proof is by induction on the typing derivation of the expression using the following inductive hypothesis:

Given Δ, Γ, Θ, e, and k where $\Theta = \langle \Delta^\Theta; \Gamma^\Theta; C^\Theta; B^\Theta \rangle$ and $k = \langle x^k; e^k \rangle$. If:

A. $\Delta; \Gamma \vdash_R e : \tau, \psi$

B. $\{\}; \mathcal{K}[\![\Delta]\!], \Delta^\Theta; \mathcal{S}[\![\Gamma]\!], \Gamma^\Theta, x^k{:}\mathcal{T}[\![\tau]\!]; C^\Theta \vdash e^k$

C. $\mathcal{K}[\![\Delta]\!], \Delta^\Theta \vdash C^\Theta \leq \overline{B^\Theta}$

D. $\mathcal{K}[\![\Delta]\!], \Delta^\Theta \vdash \overline{B^\Theta} = \overline{B^\Theta \oplus \mathcal{T}[\![\psi]\!]}$

Then:

E. $\{\}; \mathcal{K}[\![\Delta]\!], \Delta^\Theta; \mathcal{S}[\![\Gamma]\!], \Gamma^\Theta; C^\Theta \vdash \mathcal{C}_{\Delta;\Gamma;\Theta}(e)k$

In this proof, we will use several abbreviations and conventions in order to make the derivations so that we can focus on the important elements in the proof. First, we will often use the meta-variable Φ to range over type-checking contexts of the form $\Psi; \Delta; \Gamma$ or $\Psi; \Delta; \Gamma; C$.

We also abbreviate derivations involving let. A derivation:

$$\dfrac{\dfrac{\mathcal{D}_1 \quad \cdots \quad \mathcal{D}_n}{\Psi; \Delta; \Gamma; C \vdash d \Rightarrow \Delta'; \Gamma'; C'} \quad \Psi; \Delta'; \Gamma'; C' \vdash e}{\Psi; \Delta; \Gamma; C \vdash \texttt{let}\, d\, \texttt{in}\, e}$$

is abbreviated by:

$$\dfrac{\mathcal{D}_1 \quad \cdots \quad \mathcal{D}_n \quad \Psi; \Delta'; \Gamma'; C' \vdash e}{\Psi; \Delta; \Gamma; C \vdash \texttt{let}\, d\, \texttt{in}\, e}$$

Many of the typing rules contain the side condition that a Capability Calculus variable ρ, ϵ, or x not be contained in the context Δ or Γ for that judgement. We have assumed that all the variables in the translated term have been generated fresh so that this will be the case. For the sake of brevity, we do not mention this side condition each time it occurs in the proof. Many of the rules also contain well-formedness constraints on types or capabilities (ie: $\Delta \vdash \tau$ or $\Delta \vdash C$). These well-formedness constraints always follow directly from the source typing judgement and Lemma 43. However, in order to concentrate on the more important aspects of the proof, we do not mention these conditions each time they appear in a derivation.

In the following, we prove the result for the more difficult cases: `letrec`, type application, value application, `letregion`, and equality. The other cases follow a similar, but simpler pattern.

- The case for `letrec`. The translation is:

$$
\begin{array}{l}
\mathcal{C}_{\Delta;\Gamma;\Theta}(\texttt{letrec}f[\Delta'](x):\sigma\ \texttt{at}\ e_1 = e_2\ \texttt{in}\, e_3)k = \\
\quad \mathcal{C}_{\Delta;\Gamma;\Theta}(e_1)\langle x_1; \\
\quad \texttt{let}\, f = (\texttt{fix}\, f[\mathcal{K}[\![\Delta']\!], \Delta''](\epsilon', x{:}\mathcal{T}[\![\tau_1]\!], x_{cont}{:}\tau_{cont}). \\
\qquad\qquad \mathcal{C}_{\Delta\Delta';\Gamma\{f:\sigma,x:\tau_1\};\Theta'}(e_2)\langle x_2; x_{cont}(x_2)\rangle)\ \texttt{at}\ x_1 \\
\quad \texttt{in}\, \mathcal{C}_{\Delta;\Gamma\{f:\sigma\};\Theta}(e_3)k\rangle
\end{array}
$$

where

$$
\begin{array}{lcl}
\sigma & = & \forall[\Delta'].\tau_1 \xrightarrow{\psi_f} \tau_2\ \texttt{at}\ r \\
\tau_{cont} & = & \forall[\,].(\epsilon', \mathcal{T}[\![\tau_2]\!]) \to 0\ \texttt{at}\ \rho \\
B^{\Theta'} & = & \epsilon \oplus \mathcal{T}[\![\psi_f]\!] \oplus \{\rho^1\} \\
\Delta'' & = & \rho{:}\texttt{Rgn}, \epsilon{:}\texttt{Cap}, \epsilon' \leq \overline{B^{\Theta'}} \\
\Delta^{\Theta'} & = & \Delta^{\Theta}, \Delta'' \\
\Gamma^{\Theta'} & = & \Gamma^{\Theta}, x_1{:}r\ \texttt{handle}, x_{cont}{:}\tau_{cont} \\
C^{\Theta'} & = & \epsilon' \\
\Theta' & = & \langle \Delta^{\Theta'}; \Gamma^{\Theta'}; C^{\Theta'}; B^{\Theta'} \rangle
\end{array}
$$

The source typing derivation A:

$$
\frac{\begin{array}{c}(\mathrm{A}_1)\ \Delta;\Gamma \vdash_R e_1 : r\ \texttt{handle}, \psi_1 \\ (\mathrm{A}_2)\ \Delta\Delta';\Gamma\{f{:}\sigma, x{:}\tau_1\} \vdash_R e_2 : \tau_2, \psi_f \\ (\mathrm{A}_3)\ \Delta;\Gamma\{f{:}\sigma\} \vdash_R e_3 : \tau_3, \psi_3\end{array}}{\Delta;\Gamma \vdash_R \texttt{letrec}f[\Delta'](x):\sigma\ \texttt{at}\ e_1 = e_2\ \texttt{in}\, e_3 : \tau_3, \psi_1 \cup \psi_3 \cup \{r\}}\ (x, f \notin Dom(\Gamma))
$$

In order to make the derivations for this case more manageable, we will use the following abbreviations:

$$
\begin{array}{lcl}
\Phi_1 & = & \{\,\}; \mathcal{K}[\![\Delta]\!], \Delta^{\Theta}; \mathcal{S}[\![\Gamma]\!], \Gamma^{\Theta}, x_1{:}r\ \texttt{handle}; C^{\Theta} \\
\Phi_2 & = & \{\,\}; \mathcal{K}[\![\Delta, \Delta']\!], \Delta^{\Theta'}; \mathcal{S}[\![\Gamma, f{:}\sigma, x{:}\tau_1]\!], \Gamma^{\Theta'}, x_1{:}\mathcal{T}[\![r\ \texttt{handle}]\!], x_2{:}\mathcal{T}[\![\tau_1]\!]; C^{\Theta'} \\
\Phi_3 & = & \{\,\}; \mathcal{K}[\![\Delta, \Delta']\!], \Delta^{\Theta'}; \mathcal{S}[\![\Gamma, f{:}\sigma, x{:}\tau_1]\!], \Gamma^{\Theta'} x_1{:}\mathcal{T}[\![r\ \texttt{handle}]\!]; C^{\Theta'} \\
\Phi_4 & = & \{\,\}; \mathcal{K}[\![\Delta]\!], \Delta^{\Theta}; \mathcal{S}[\![\Gamma, f{:}\sigma]\!], \Gamma^{\Theta}; C^{\Theta} \\
\Phi_5 & = & \{\,\}; \mathcal{K}[\![\Delta]\!], \Delta^{\Theta}; \mathcal{S}[\![\Gamma, f{:}\sigma]\!], \Gamma^{\Theta}, x_1{:}\mathcal{T}[\![r\ \texttt{handle}]\!]; C^{\Theta}
\end{array}
$$

We begin by showing the continuation used in the translation of e_2 is well-formed under the appropriate context (call this fact B_2):

$$
\frac{\dfrac{\text{(By v-var)}}{\Phi_2 \vdash x_{cont} : \mathcal{T}[\![\tau_{cont}]\!]} \quad \dfrac{\text{(By v-var)}}{\Phi_2 \vdash x_2 : \mathcal{T}[\![\tau_1]\!]} \quad \dfrac{\text{(By eq-reflex)}}{\mathcal{K}[\![\Delta, \Delta']\!], \Delta^{\Theta'} \vdash \epsilon' \leq \epsilon'} \quad \mathcal{D}_{\epsilon'}}{\Phi_2 \vdash x_{cont}(x_2)}
$$

The derivation $\mathcal{D}_{\epsilon'}$ is the following (recall that $\Delta^{\Theta'}$ is $\Delta^{\Theta}, \rho{:}\texttt{Rgn}, \epsilon{:}\texttt{Cap}, \epsilon' \leq \overline{\epsilon \oplus \mathcal{T}[\![\psi]\!] \oplus \{\rho^1\}}$):

$$
\begin{array}{lcll}
\mathcal{K}[\![\Delta, \Delta']\!], \Delta^{\Theta'} \vdash \epsilon' & \leq & \overline{\epsilon \oplus \mathcal{T}[\![\psi]\!] \oplus \{\rho^1\}} & \text{(By sub-var)} \\
& = & \overline{\epsilon \oplus \mathcal{T}[\![\psi]\!]} \oplus \{\rho^+\} & \text{(By eq-distrib)}
\end{array}
$$

We have now shown B_2. By use of rule sub-var, we can conclude that (C_2):

$$\mathcal{K}[\![\Delta, \Delta']\!], \Delta^{\Theta'} \vdash C^{\Theta'} \leq \overline{B^{\Theta'}}$$

We also have (D_2):

$$\begin{array}{rcll} \mathcal{K}[\![\Delta, \Delta']\!], \Delta^{\Theta'} \vdash \overline{B^{\Theta'}} & = & \overline{\epsilon \oplus \mathcal{T}[\![\psi_f]\!] \oplus \{\rho^1\}} & \text{(expand abbreviation)} \\ & = & \overline{\epsilon \oplus \{\rho^1\} \oplus \mathcal{T}[\![\psi_f]\!]} & \text{(By eq-comm)} \\ & = & \overline{\epsilon \oplus \{\rho^1\} \oplus \mathcal{T}[\![\psi_f]\!] \oplus \mathcal{T}[\![\psi_f]\!]} & \text{(By eq-dup)} \end{array}$$

Using A_2, B_2, C_2 and D_2, we can apply the induction hypothesis and obtain E_2:

$$\Phi_3 \vdash \mathcal{C}_{\Delta\Delta';\Gamma\{f:\sigma, x:\tau_1\};\Theta'}(e_2)\langle x_2; x_{cont}(x_2)\rangle$$

Now, from A_3, B, C, and D, we can apply the induction hypothesis and obtain E_3:

$$\Phi_4 \vdash \mathcal{C}_{\Delta;\Gamma\{f:\sigma\};\Theta, x_1:\mathcal{T}[\![r\ \texttt{handle}]\!]}(e_3)k$$

Using E_2 and E_3, we can build the typing derivation for the code in the continuation for translating e_1 (call this fact B_1):

$$\frac{\dfrac{\text{(By v-var)}}{\Phi_1 \vdash x_1 : r\ \texttt{handle}} \quad \dfrac{E_2}{\Phi \vdash \texttt{fix}\, f[\cdots](\cdots).\mathcal{C}_{\Delta\Delta';\Gamma\{f:\sigma, x:\tau_1\};\Theta'}(e_2)\langle x_2; x_{cont}(x_2)\rangle\ \texttt{at}\ r} \quad \mathcal{D}_r \quad E_3}{\Phi_1 \vdash \texttt{let}\, f = (\cdots)\ \texttt{at}\ x_1\ \texttt{in}\, \mathcal{C}_{\Delta;\Gamma\{f:\sigma\};\Theta, x_1:r\ \texttt{handle}}(e_3)k}$$

where the derivation $\mathcal{D}_r$ is:

$$\begin{array}{rcll} \mathcal{K}[\![\Delta]\!]\Delta^{\Theta} \vdash C^{\Theta} & \leq & \overline{B^{\Theta}} & \text{(By assumption C)} \\ & = & \overline{B^{\Theta} \oplus \{r^1\}} & \text{(By assumption D, Lemma 48)} \\ & = & \overline{B^{\Theta}} \oplus \{r^+\} & \text{(By eq-distrib)} \end{array}$$

Finally, using A_1, B_1, C, and D, we can apply the induction hypothesis to the translation of e_1, giving us the final result:

$$\{\,\}; \mathcal{K}[\![\Delta]\!], \Delta^{\Theta}; \mathcal{S}[\![\Gamma]\!], \Gamma^{\Theta}; C^{\Theta} \vdash \mathcal{C}_{\Delta;\Gamma;\Theta}(\texttt{letrec}\, f[\Delta'](x) : \sigma\ \texttt{at}\ e_1 = e_2\ \texttt{in}\, e_3)k$$

- The case for type application. The translation is:

$$\mathcal{C}_{\Delta;\Gamma;\Theta}(f[c_1, \ldots, c_n])k = \mathcal{A}(k, f[\mathcal{T}[\![c_1]\!], \ldots, \mathcal{T}[\![c_n]\!]])$$

The source typing derivation A, where σ is $\forall[\alpha_1{:}\kappa_1, \ldots, \alpha_n{:}\kappa_n].\tau_1 \xrightarrow{\psi} \tau_2\ \texttt{at}\ r$:

$$\frac{\Delta \vdash_R \sigma \qquad \Delta \vdash_R c_i : \kappa_i}{\Delta; \Gamma \vdash_R f[c_1, \ldots, c_n] : (\tau_1 \xrightarrow{\psi} \tau_2)[c_1, \ldots, c_n/\alpha_1, \ldots, \alpha_n]\ \texttt{at}\ r, \emptyset}\ (\Gamma(f) = \sigma)$$

First, we must show that the value $f[\mathcal{T}[\![c_1]\!], \ldots, \mathcal{T}[\![c_n]\!]]$ is well-formed under the context

$$\Phi_1 = \{\,\}; \mathcal{K}[\![\Delta]\!], \Delta^\Theta; \mathcal{S}[\![\Gamma]\!], \Gamma^\Theta$$

where $\Theta = \langle \Delta^\Theta; \Gamma^\Theta; C^\Theta; B^\Theta \rangle$
First, we can conclude that

$$\frac{\text{(By A and rule v-var)}}{\Phi_1 \vdash f : \mathcal{T}[\![\forall[\alpha_1{:}\kappa_1, \ldots, \alpha_n{:}\kappa_n].\tau_1 \xrightarrow{\psi} \tau_2 \,\mathtt{at}\, r]\!]}$$

Now, by induction on the number of constructors c_i applied to f, and use of the rule v-type, we can conclude that

$$\Phi_1 \vdash f[\mathcal{T}[\![c_1]\!], \ldots, \mathcal{T}[\![c_n]\!]] : \mathcal{T}[\![\tau_1 \xrightarrow{\psi} \tau_2 \,\mathtt{at}\, r]\!][\mathcal{T}[\![c_1]\!], \ldots, \mathcal{T}[\![c_n]\!]/\alpha_1, \ldots, \alpha_n]$$

By the Substitution Lemma, Lemma 45, we have

$$\Phi_1 \vdash f[\mathcal{T}[\![c_1]\!], \ldots, \mathcal{T}[\![c_n]\!]] : \mathcal{T}[\![(\tau_1 \xrightarrow{\psi} \tau_2 \,\mathtt{at}\, r)[c_1, \ldots, c_n/\alpha_1, \ldots, \alpha_n]]\!]$$

Using this fact, assumption B, and Lemma 50, we obtain our final result:

$$\Phi_1; C^\Theta \vdash \mathcal{A}(k, f[\mathcal{T}[\![c_1]\!], \ldots, \mathcal{T}[\![c_n]\!]])$$

- The case for application. The translation is:

$$\begin{array}{l}
\mathcal{C}_{\Delta;\Gamma;\Theta}(e_1^{\tau_f}\ e_2)k = \\
\quad \mathcal{C}_{\Delta;\Gamma;\Theta}(e_1)\langle x_1; \\
\quad \mathcal{C}_{\Delta;\Gamma;\Theta,x_1:\mathcal{T}[\![\tau_f]\!]}(e_2)\langle x_2; \\
\quad \mathtt{let\ newrgn}\,\rho, x_\rho\ \mathtt{in} \\
\quad \mathtt{let}\ f_{cont} = (\mathtt{fix}\ f_{cont}[\,](C^\Theta \oplus \{\rho^1\}, x{:}\mathcal{T}[\![\tau_2]\!]).\mathtt{let\ freergn}\,x_\rho\ \mathtt{in}\,\mathcal{A}(k,x))\ \mathtt{at}\ x_\rho \\
\quad \mathtt{in}\ x_1[\rho, B^\Theta, C^\Theta \oplus \{\rho^1\}](x_2, f_{cont})\rangle\rangle
\end{array}$$

where

$$\tau_f \quad = \quad \tau_1 \xrightarrow{\psi_f} \tau_2 \,\mathtt{at}\, r$$

The source typing derivation for the term is (A):

$$\frac{(\mathrm{A}_1)\ \Delta; \Gamma \vdash_R e_1 : \tau_1 \xrightarrow{\psi_3} \tau_2 \,\mathtt{at}\, r, \psi_1 \quad (\mathrm{A}_2)\ \Delta; \Gamma \vdash_R e_2 : \tau_1, \psi_2}{\Delta; \Gamma \vdash_R e_1 e_2 : \tau_2, \psi_1 \cup \psi_2 \cup \psi_3 \cup \{r\}}$$

We begin showing that the result of the translation is type-correct by showing that the body of the innermost continuation ($\mathtt{let\,newrgn}\,\rho, x_\rho\ \mathtt{in}\ \cdots$) is well-formed under the appropriate context. In order to make the derivation more manageable, we will use the following abbreviations:

$$
\begin{array}{lcl}
\Phi_1 & = & \{\,\};\mathcal{K}[\![\Delta]\!],\Delta^\Theta;\mathcal{S}[\![\Gamma]\!],\Gamma^\Theta,x_1{:}\mathcal{T}[\![\tau_f]\!],x_2{:}\mathcal{T}[\![\tau_2]\!];C^\Theta \\
\Phi_2 & = & \{\,\};\mathcal{K}[\![\Delta]\!],\Delta^\Theta,\rho{:}\mathtt{Rgn};\mathcal{S}[\![\Gamma]\!],\Gamma^\Theta,x_1{:}\mathcal{T}[\![\tau_f]\!],x_2{:}\mathcal{T}[\![\tau_2]\!],x_\rho{:}\rho\ \mathtt{handle};C^\Theta \oplus \{\rho^1\} \\
\Phi_3 & = & \{\,\};\mathcal{K}[\![\Delta]\!],\Delta^\Theta,\rho{:}\mathtt{Rgn};\mathcal{S}[\![\Gamma]\!],\Gamma^\Theta,x_1{:}\mathcal{T}[\![\tau_f]\!],x_2{:}\mathcal{T}[\![\tau_2]\!],x_\rho{:}\rho\ \mathtt{handle},f_{cont}{:}\tau_{cont} \\
\Phi_4 & = & \{\,\};\mathcal{K}[\![\Delta]\!],\Delta^\Theta,\rho{:}\mathtt{Rgn};\mathcal{S}[\![\Gamma]\!],\Gamma^\Theta,x_1{:}\mathcal{T}[\![\tau_f]\!],x_2{:}\mathcal{T}[\![\tau_2]\!],x_\rho{:}\rho\ \mathtt{handle} \\
\Phi_5 & = & \{\,\};\mathcal{K}[\![\Delta]\!],\Delta^\Theta,\rho{:}\mathtt{Rgn};\mathcal{S}[\![\Gamma]\!],\Gamma^\Theta,x_1{:}\mathcal{T}[\![\tau_f]\!],x_2{:}\mathcal{T}[\![\tau_2]\!],x_\rho{:}\rho\ \mathtt{handle},f_{cont}{:}\tau_{cont},x{:}\mathcal{T}[\![\tau_2]\!] \\
e_1' & = & \mathtt{let}\ f_{cont} = h_{cont}\ \mathtt{at}\ x_\rho\ \mathtt{in}\ e_2' \\
e_2' & = & x_1[\rho,B^\Theta,C^\Theta \oplus \{\rho^1\}](x_2,f_{cont}) \\
h_{cont} & = & \mathtt{fix}\ f_{cont}[\,](C^\Theta \oplus \{\rho^1\},x{:}\mathcal{T}[\![\tau_2]\!]).\mathtt{let}\ \mathtt{freergn}\ x_\rho\ \mathtt{in}\ \mathcal{A}(k,x)) \\
\tau_{app} & = & (C^\Theta \oplus \{\rho^1\},\mathcal{T}[\![\tau_1]\!],\tau_{cont}) \to 0\ \mathtt{at}\ r \\
\tau_{cont} & = & (C^\Theta \oplus \{\rho^1\},\mathcal{T}[\![\tau_2]\!]) \to 0\ \mathtt{at}\ \rho
\end{array}
$$

The derivation is as follows:

$$
\dfrac{\dfrac{\dfrac{\mathcal{D}_{app}}{\Phi_3 \vdash x_1[\rho,B^\Theta,C^\Theta \oplus \{\rho^1\}] : \tau_{app}} \quad \dfrac{\text{(by v-var)}}{\Phi_3 \vdash x_2 : \mathcal{T}[\![\tau_1]\!]} \quad \dfrac{\text{(by v-var)}}{\Phi_3 \vdash f_{cont} : \tau_{cont}} \quad \mathcal{D}_r \quad \mathcal{D}_{C^\Theta}}{\Phi_2,f_{cont}{:}\tau_{cont} \vdash x_1[\rho,B^\Theta,C^\Theta \oplus \{\rho^1\}](x_2,f_{cont})} \quad \mathcal{D}_{x_\rho} \quad \mathcal{D}_h \quad \mathcal{D}_\rho}{\Phi_2 \vdash \mathtt{let}\ f_{cont} = h_{cont}\ \mathtt{at}\ x_\rho\ \mathtt{in}\ e_2'}}{\Phi_1 \vdash \mathtt{let}\ \mathtt{newrgn}\ \rho,x_\rho\ \mathtt{in}\ e_1'}
$$

The derivation $\mathcal{D}_{app}$ can be proven as follows. First, by rule v-var, we can deduce

$$
\Phi_3 \vdash x_1 : \mathcal{T}[\![\tau_f]\!] = \forall[\rho'{:}\mathtt{Rgn},\epsilon{:}\mathtt{Cap},\epsilon' \le \overline{\epsilon \oplus \mathcal{T}[\![\psi_3]\!] \oplus \{\rho'^1\}}].(\epsilon',\mathcal{T}[\![\tau_1]\!],\tau_{cont}) \to 0\ \mathtt{at}\ r
$$

From this judgement, two applications of the rule v-type give us:

$$
\Phi_3 \vdash x_1[\rho,B^\Theta] : \forall[\epsilon' \le \overline{B^\Theta \oplus \mathcal{T}[\![\psi_3]\!] \oplus \{\rho^1\}}].(\epsilon',\mathcal{T}[\![\tau_1]\!],\tau_{cont}) \to 0\ \mathtt{at}\ r
$$

Finally, by rule v-sub, we can conclude:

$$
\Phi_3 \vdash x_1[\rho,B^\Theta,C^\Theta \oplus \{\rho^1\}] : \forall[].(C^\Theta \oplus \{\rho^1\},\mathcal{T}[\![\tau_1]\!],\tau_{cont}) \to 0\ \mathtt{at}\ r
$$

because the required sub-capability relation holds:

$$
\begin{array}{lcll}
\mathcal{K}[\![\Delta]\!],\Delta^\Theta,\rho{:}\mathtt{Rgn} \vdash C^\Theta \oplus \{\rho^1\} & \le & \overline{B^\Theta} \oplus \{\rho^1\} & \text{(By assumption C)} \\
& = & \overline{B^\Theta \oplus \mathcal{T}[\![\psi_3]\!]} \oplus \{\rho^1\} & \text{(By assumption D, Lemma 48)} \\
& \le & \overline{B^\Theta \oplus \mathcal{T}[\![\psi_3]\!] \oplus \{\rho^1\}} & \text{(By rule sub-bar)}
\end{array}
$$

Next, we consider the derivation $\mathcal{D}_r$. Here we must show that

$$
\mathcal{K}[\![\Delta]\!],\Delta^\Theta,\rho{:}\mathtt{Rgn} \vdash C^\Theta \oplus \{\rho^1\} \le C'' \oplus \{r^+\}
$$

for some capability C''. The reasoning is straightforward:

$$
\begin{array}{rcll}
\mathcal{K}[\![\Delta]\!], \Delta^\Theta, \rho{:}\mathtt{Rgn} \vdash C^\Theta \oplus \{\rho^1\} & = & \{\rho^1\} \oplus \overline{C^\Theta} & \text{(By rule eq-comm)} \\
& \leq & \{\rho^1\} \oplus \overline{B^\Theta} & \text{(By assumption C)} \\
& = & \{\rho^1\} \oplus \overline{B^\Theta \oplus \mathcal{T}[\![\{r\}]\!]} & \text{(By assumption D, Lemma 48)} \\
& = & \{\rho^1\} \oplus \overline{B^\Theta \oplus \{r^1\}} & \text{(By def. of translation)} \\
& = & \{\rho^1\} \oplus \overline{B^\Theta} \oplus \{r^+\} & \text{(By rule eq-distrib)}
\end{array}
$$

Next, we consider $\mathcal{D}_{C^\Theta}$. The judgement we must prove is $\mathcal{K}[\![\Delta]\!], \Delta^\Theta \vdash C^\Theta \oplus \{\rho^1\} \leq C^\Theta \oplus \{\rho^1\}$. This follows by rules sub-eq and eq-reflex.

Next, we consider $\mathcal{D}_{x_\rho}$. The judgement we must prove is: $\Phi_4 \vdash x_\rho : \rho$ `handle`. The judgement follows by rule v-var.

Next, we must prove $\mathcal{D}_h$:

$$
\dfrac{\dfrac{\text{(by v-var)}}{\Phi_5 \vdash x_\rho : \rho\ \mathtt{handle}} \quad \dfrac{\text{(by eq-reflex)}}{\mathcal{K}[\![\Delta]\!], \Delta^\Theta, \rho{:}\mathtt{Rgn} \vdash C^\Theta \oplus \{\rho^1\} = C^\Theta \oplus \{\rho^1\} : \mathtt{Cap}} \quad \mathcal{D}_{\mathcal{A}(k,x)}}{\dfrac{\Phi_5; C^\Theta \oplus \{\rho^1\} \vdash \mathtt{let\,freergn}\, x_\rho\, \mathtt{in}\, \mathcal{A}(k,x)}{\Phi_4 \vdash h_{cont}\, \mathtt{at}\, \rho : \mathcal{T}[\![\tau_f]\!]}}
$$

The judgement we are trying to prove in the derivation $\mathcal{D}_{\mathcal{A}(k,x)}$ is $\Phi_5; C^\Theta \vdash \mathcal{A}(k,x)$. Using rule v-var, we can conclude that $\Phi_5 \vdash x : \mathcal{T}[\![\tau_2]\!]$. Assumption B tells us that $\Phi_5; C^\Theta \vdash e^k$. Hence, by Lemma 50, we have the result.

Finally, we show $\mathcal{D}_\rho$. We must prove $\mathcal{K}[\![\Delta]\!], \Delta^\Theta, \rho{:}\mathtt{Rgn} \vdash C^\Theta \oplus \{\rho^1\} \leq C'' \oplus \{\rho^+\}$ for some capability C''. This fact follows using rule sub-dup (C'' is C^Θ).

We have now satisfied all of the requirements necessary to show that the body of the innermost continuation is well-formed:

$$(\mathrm{B}_2)\ \Phi_1 \vdash \mathtt{letregion}\, \rho, x_\rho\, \mathtt{in}\, e_1'$$

We have A_2 from the source typing derivation. If we let C_2 be C and D_2 be D, then we fulfill all of the requirements for the induction hypothesis and we may conclude E_2:

$$\{\,\}; \mathcal{K}[\![\Delta]\!], \Delta^\Theta; \mathcal{S}[\![\Gamma]\!], \Gamma^\Theta, x_1{:}\mathcal{T}[\![\tau_f]\!]; C^\Theta \vdash \mathcal{C}_{\Delta;\Gamma;\Theta,x_1:\mathcal{T}[\![\tau_f]\!]}(e_2)\langle x_2; \mathtt{letregion}\, \rho, x_\rho\, \mathtt{in}\, e_1'\rangle$$

Now, using this fact, A_1 from the source typing derivation, C, and D, we can apply the induction hypothesis and conclude E_1:

$$\{\,\}; \mathcal{K}[\![\Delta]\!], \Delta^\Theta; \mathcal{S}[\![\Gamma]\!], \Gamma^\Theta; C^\Theta \vdash \mathcal{C}_{\Delta;\Gamma;\Theta}(e_1)\langle x_1; \mathcal{C}_{\Delta;\Gamma;\Theta,x_1:\mathcal{T}[\![\tau_f]\!]}(e_2)\langle\cdots\rangle\rangle$$

which is equivalent to the result E that we were trying to prove.

- The case for `letregion`. The translation is:

$$\begin{array}{l}\mathcal{C}_{\Delta;\Gamma;\Theta}(\texttt{letregion}\,\rho, x_\rho\ \texttt{in}\, e)k = \\ \quad \texttt{let newrgn}\,\rho, x_\rho\ \texttt{in} \\ \quad \mathcal{C}_{\Delta\{\rho:\texttt{Rgn}\};\Gamma\{x_\rho:\rho\ \texttt{handle}\};\Theta'}(e)\langle x'; \texttt{freergn}\, x_\rho\ \texttt{in}\, \mathcal{A}(k, x')\rangle \\ \text{where} \\ \quad \Theta' \quad = \quad \langle \Delta^\Theta; \Gamma^\Theta; C^\Theta \oplus \{\rho^1\}; B^\Theta \oplus \{\rho^1\}\rangle\end{array}$$

The source typing derivation A:

$$\frac{(\mathrm{A}_1)\ \Delta\{\rho{:}\texttt{Rgn}\}; \Gamma\{x_\rho{:}\rho\ \texttt{handle}\} \vdash_R e : \tau, \psi}{\Delta; \Gamma \vdash_R \texttt{letregion}\,\rho, x_\rho\ \texttt{in}\, e : \tau, \psi\backslash\{\rho\}} \left(\begin{array}{c}\rho \notin ftv(\tau) \cup Dom(\Delta) \\ x_\rho \notin Dom(\Gamma)\end{array}\right)$$

First, we show that the continuation for the translation of e is well-formed under an appropriate context:

$$\frac{\mathcal{D}_{\mathcal{A}(k,x')} \quad \mathcal{D}_{C^\Theta} \quad \dfrac{}{\mathcal{K}[\![\Delta, \rho{:}\texttt{Rgn}]\!], \Delta^\Theta; \mathcal{S}[\![\Gamma, x_\rho{:}\rho\ \texttt{handle}]\!], \Gamma^\Theta, \Gamma^k, x'{:}\mathcal{T}[\![\tau]\!] \vdash x_\rho : \rho\ \texttt{handle}}\ (\text{by v-var})}{(\mathrm{B}_1)\ \mathcal{K}[\![\Delta, \rho{:}\texttt{Rgn}]\!], \Delta^\Theta; \mathcal{S}[\![\Gamma, x_\rho{:}\rho\ \texttt{handle}]\!], \Gamma^\Theta, x'{:}\mathcal{T}[\![\tau]\!]; C^\Theta \oplus \{\rho^1\} \vdash \texttt{let freergn}\, x_\rho\ \texttt{in}\, \mathcal{A}(k, x')}$$

The judgement we must prove in $\mathcal{D}_{\mathcal{A}(k,x')}$ is:

$$(J)\ \mathcal{K}[\![\Delta, \rho{:}\texttt{Rgn}]\!], \Delta^\Theta; \mathcal{S}[\![\Gamma, x_\rho{:}\rho\ \texttt{handle}]\!], \Gamma^\Theta, x'{:}\mathcal{T}[\![\tau]\!]; C^\Theta \vdash \mathcal{A}(k, x')$$

By rule v-var, we can conclude:

$$\mathcal{K}[\![\Delta, \rho{:}\texttt{Rgn}]\!], \Delta^\Theta; \mathcal{S}[\![\Gamma, x_\rho{:}\rho\ \texttt{handle}]\!], \Gamma^\Theta, x'{:}\mathcal{T}[\![\tau]\!] \vdash x' : \mathcal{T}[\![\tau]\!]$$

Using this fact, assumption B and Lemma 50, we can conclude (J).
The judgement we must prove in $\mathcal{D}_{C^\Theta}$ is:

$$\mathcal{K}[\![\Delta, \rho{:}\texttt{Rgn}]\!], \Delta^\Theta \vdash C^\Theta \oplus \{\rho^1\} = C^\Theta \oplus \{\rho^1\}$$

which follows by the rule eq-reflex.
Now, we have fulfilled all the requirements necessary to show that the body of the innermost continuation is well-formed (call this fact B_1). We have A_1 from the typing derivation. In order to apply the induction hypothesis, we must show C_1:

$$\begin{array}{rcll}\mathcal{K}[\![\Delta, \rho{:}\texttt{Rgn}]\!], \Delta^\Theta \vdash C^\Theta \oplus \{\rho^1\} & \leq & \overline{B^\Theta} \oplus \{\rho^1\} & (\text{By assumption C}) \\ & \leq & \overline{B^\Theta} \oplus \overline{\{\rho^1\}} & (\text{By rule sub-bar}) \\ & = & \overline{B^\Theta \oplus \{\rho^1\}} & (\text{By rule eq-distrib})\end{array}$$

and also D_1:

$$\begin{array}{rcll}
\mathcal{K}[\![\Delta, \rho{:}\mathtt{Rgn}]\!], \Delta^\Theta \vdash \overline{B^\Theta \oplus \{\rho^1\}} & = & \overline{B^\Theta \oplus \mathcal{T}[\![\psi \backslash \{\rho\}]\!] \oplus \{\rho^1\}} & \text{(By assumption D)} \\
& = & \overline{B^\Theta \oplus \mathcal{T}[\![\psi]\!] \oplus \{\rho^1\}} & \text{(By Lemma 49)} \\
& = & \overline{B^\Theta \oplus \{\rho^1\} \oplus \mathcal{T}[\![\psi]\!]} & \text{(By rule eq-comm)}
\end{array}$$

Together A_1, B_1, C_1 and D_1 satisfy the preconditions for applying the induction hypothesis. The result is E_1:

$$\mathcal{K}[\![\Delta, \rho{:}\mathtt{Rgn}]\!], \Delta^\Theta; \mathcal{S}[\![\Gamma, x_\rho{:}\rho\ \mathtt{handle}]\!], \Gamma^\Theta; C^\Theta \oplus \{\rho^1\} \vdash \mathcal{C}_{\mathcal{K}[\![\Delta,\rho:\mathtt{Rgn}]\!],\Delta^\Theta;\mathcal{S}[\![\Gamma,x_\rho:\rho\,\mathtt{handle}]\!],\Theta'}(e)\langle\cdots\rangle$$

Now, we can show the result of the translation type-checks:

$$\frac{\mathrm{E}_1}{\mathcal{K}[\![\Delta]\!], \Delta^\Theta; \mathcal{S}[\![\Gamma]\!], \Gamma^\Theta; C^\Theta \vdash \mathtt{let}\,\mathtt{newrgn}\,\rho, x_\rho\,\mathtt{in}\,\mathcal{C}_{\mathcal{K}[\![\Delta,\rho:\mathtt{Rgn}]\!],\Delta^\Theta;\mathcal{S}[\![\Gamma,x_\rho:\rho\,\mathtt{handle}]\!],\Theta'}(e)\langle\cdots\rangle}$$

- The equality rule. The source typing derivation A is

$$\frac{\Delta; \Gamma \vdash_R e : \tau, \psi \quad \Delta \vdash_R \tau = \tau' : \mathtt{Type} \quad \Delta \vdash_R \psi \subseteq \psi'}{(\mathrm{A}_1)\,\Delta; \Gamma \vdash_R e : \tau', \psi'}$$

and the continuation $k = \langle x; e^k \rangle$ is well-formed under the appropriate context (B):

$$\{\,\}; \mathcal{K}[\![\Delta]\!], \Delta^\Theta; \mathcal{S}[\![\Gamma]\!], \Gamma^\Theta, x{:}\mathcal{T}[\![\tau]\!]; C^\Theta \vdash e^k$$

By Lemma 44 and the equality judgement $\Delta \vdash \tau = \tau' : \mathtt{Type}$ we can deduce that

$$\mathcal{K}[\![\Delta]\!] \vdash \mathcal{T}[\![\tau]\!] = \mathcal{T}[\![\tau']\!] : \mathtt{Type}$$

Therefore, by Lemma 51, we can deduce (B_1).

$$\{\,\}; \mathcal{K}[\![\Delta]\!], \Delta^\Theta; \mathcal{S}[\![\Gamma]\!], \Gamma^\Theta, x{:}\mathcal{T}[\![\tau']\!]; C^\Theta \vdash e^k$$

Now, recall Assumption D states that

$$\mathcal{K}[\![\Delta]\!], \Delta \vdash \overline{B^\Theta} = \overline{B^\Theta \oplus \mathcal{T}[\![\psi']\!]} : \mathtt{Cap}$$

Using this fact, and the source typing derivation (A), which states that $\Delta \vdash \psi \subseteq \psi'$, and Lemma 48, we can deduce that (D_1)

$$\mathcal{K}[\![\Delta]\!], \Delta \vdash \overline{B^\Theta} = \overline{B^\Theta \oplus \mathcal{T}[\![\psi]\!]} : \mathtt{Cap}$$

Using A_1, B_1, C, and D_1, we can apply the induction hypothesis and obtain

$$\{\,\}; \mathcal{K}[\![\Delta]\!], \Delta^\Theta; \mathcal{S}[\![\Gamma]\!], \Gamma^\Theta; C^\Theta \vdash \mathcal{C}_{\Delta;\Gamma;\Theta}(e)k$$

and we are done.

We have completed the proof that the induction hypothesis is preserved by the translation. In order to obtain the proof of the CPS translation theorem, we simply instantiate the induction hypothesis. We are given part A: $\cdot;\cdot \vdash e : \mathtt{int}, \emptyset$.

The continuation k is $\langle x; \mathtt{halt}\ x\rangle$. Using the halt rule, we have part B,

$$\frac{\{\,\};\cdot;x{:}\mathtt{int} \vdash x : \mathtt{int} \qquad \cdot \vdash \emptyset = \emptyset : \mathtt{Cap}}{\{\,\};\cdot;x{:}\mathtt{int};\emptyset \vdash \mathtt{halt}\ x}$$

Part C is trivial: $\cdot \vdash \emptyset \leq \emptyset$.

Part D is also straightforward,

$$\begin{array}{rcll} \cdot \vdash \overline{\emptyset} & \leq & \overline{\emptyset} & \text{(By rule sub-eq)} \\ & = & \overline{\emptyset \oplus \emptyset} & \text{(By rule eq-dup)} \\ & = & \overline{\emptyset \oplus \mathcal{T}[\![\emptyset]\!]} & \text{(By definition of } \mathcal{T}[\![\,\cdot\,]\!]) \end{array}$$

Therefore, we can conclude E, $\{\,\};\cdot;\cdot;\emptyset \vdash \mathcal{C}_\Theta(e)k$ where Θ is the empty translation environment $\langle\cdot;\cdot;\emptyset;\emptyset\rangle$ and k is the trivial continuation $\langle x; \mathtt{halt}\ x\rangle$.

□