Typed Memory Management in a Calculus of Capabilities*

Karl Crary
Carnegie Mellon University

David Walker
Cornell University

Greg Morrisett
Cornell University

Abstract

An increasing number of systems rely on programming language technology to ensure safety and security of low-level code. Unfortunately, these systems typically rely on a complex, trusted garbage collector. Region-based type systems provide an alternative to garbage collection by making memory management explicit but verifiably safe. However, it has not been clear how to use regions in low-level, type-safe code.

We present a compiler intermediate language, called the Capability Calculus, that supports region-based memory management, enjoys a provably safe type system, and is straightforward to compile to a typed assembly language. Source languages may be compiled to our language using known region inference algorithms. Furthermore, region lifetimes need not be lexically scoped in our language, yet the language may be checked for safety without complex analyses. Finally, our soundness proof is relatively simple, employing only standard techniques.

The central novelty 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.

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 [19, 39, 2, 25, 24, 17, 14]. For example, the SPIN project [2] 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 [19] 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.

Recently, Necula and Lee [26, 25] have proposed Proof-Carrying Code (PCC) and Morrisett *et al.* [24] 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 [27], 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 [35, 36] 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 [34, 3, 4]. There are several advantages to region-based memory management; from our point of view, the two most important are:

1. Deallocation is explicit but provably safe.
2. The run-time routines necessary for region-based memory management are relatively simple constant-time operations and, in principle, could be formally verified.

The Tofte-Talpin calculus uses a lexically scoped expression (`letregion r in e end`) to delimit the lifetime of a region `r`. Memory for the region is allocated when control enters the scope of the `letregion` construct and is deallocated when control leaves the scope. This mechanism results in a strictly LIFO (last-in, first-out) ordering of region lifetimes.

*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 and ARPA/RADC grant F30602-96-1-0317. 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.

To appear in the 1999 Symposium on the Principles of Programming Languages, San Antonio, Texas, January 1999.

Both Birkedal *et al.* [4] and Aiken *et al.* [1] observed that a completely LIFO (de)allocation of regions would make poor use of memory in many cases. They proposed a series of optimizations that often alleviate efficiency concerns and even improve upon traditional tracing garbage collection in some cases. Although their optimizations are safe, there is no simple proof- or type-checker that an untrusting client can use to check the output code. Similarly, even the most straightforward code-generation requires that we stray from the Tofte-Talpin framework and allow arbitrary separation of allocation and deallocation points. Therefore, while region inference brings us half way to our goal, in order to construct a strongly-typed region-based assembly language we must re-examine the fundamental question: *"When can we free an object* `x`*?"*

One solution is to free `x` when you can guarantee that it will no longer be accessed. Operating systems such as Hydra [41] have solved the access control problem before 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, these systems revoke capabilities, thereby preventing future access.

1.2 Overview of Contributions

In the rest of this paper, we describe a strongly typed language called the Capability Calculus. Our language's type system provides an efficient way to check the safety of explicit, arbitrarily ordered region allocation and deallocation instructions using a notion of capability. As in traditional capability systems, our type system keeps track of capability copies carefully in order to 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.

We have a purely *syntactic* argument, based on Subject Reduction and Progress lemmas in the style of Felleisen and Wright [40], 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 [36], 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 have a formal translation of a variant of the Tofte-Talpin language into our calculus. We describe the translation in this paper by example; the full details appear in the companion technical report [5]. We also illustrate how some region-based optimizations may be coded in our language, taking advantage of our extra freedom to place allocation and deallocation points.

We have adapted the type system for the Capability Calculus to the setting of Typed Assembly Language, providing for the first time the ability for "applets" to explicitly control their memory management without sacrificing memory- or type-safety. As the typing constructs at the intermediate and assembly levels are so similar, we discuss here only the intermediate language and refer the interested reader to the companion technical report [5] for details on the assembly level.

2 A Calculus of Capabilities

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

Programs in the Capability Calculus are written in continuation-passing style (CPS) [29]. That is, functions 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 in our type system. 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. The complete static and operational semantics are specified in Appendix A.

2.1 Preliminaries

We specify the operational behavior of the Capability Calculus using an allocation semantics [22, 23, 24], which makes the allocation of data in memory explicit. The semantics 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

kinds	κ	::=	$\texttt{Type} \mid \texttt{Rgn} \mid \texttt{Cap}$
constructor vars	α, ρ, ϵ		
constructors	c	::=	$\alpha \mid \tau \mid r \mid C$
types	τ	::=	$\alpha \mid \texttt{int} \mid r\ \texttt{handle} \mid \forall[\Delta].(C, \tau_1, \ldots, \tau_n) \to \texttt{0 at}\ r \mid \langle \tau_1, \ldots, \tau_n \rangle\ \texttt{at}\ r$
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	::=	$x \mid i \mid \nu.\ell \mid \texttt{handle}(\nu) \mid v[c]$
heap values	h	::=	$\texttt{fix}\, f[\Delta](C, x_1{:}\tau_1, \ldots, x_n{:}\tau_n).e \mid \langle v_1, \ldots, v_n \rangle$
arithmetic ops	p	::=	$+ \mid - \mid \times$
declarations	d	::=	$x = v \mid x = v_1\ p\ v_2 \mid x = h\ \texttt{at}\ v \mid x = \pi_i v \mid \texttt{newrgn}\,\rho, x \mid \texttt{freergn}\, v$
terms	e	::=	$\texttt{let}\, d\ \texttt{in}\, e \mid \texttt{if0}\, v\ \texttt{then}\, e_2\ \texttt{else}\, e_3 \mid v(v_1, \ldots, v_n) \mid \texttt{halt}\ v$
memory regions	R	::=	$\{\ell_1 \mapsto h_1, \ldots, \ell_n \mapsto h_n\}$
memories	M	::=	$\{\nu_1 \mapsto R_1, \ldots, \nu_n \mapsto R_n\}$
machine states	P	::=	(M, e)

Figure 1: Capability Syntax

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 $\forall[\,].(C, \tau_1, \ldots, \tau_n) \to \texttt{0 at}\ r$ 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 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 $\forall[\Delta].(C, \vec{\tau}) \to \texttt{0 at}\, r$ by $(C, \vec{\tau}) \to \texttt{0 at}\, r$.

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

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

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 $(C, \tau) \to \texttt{0 at}\ r$. Then f is called by $f[\tau][r](v, g)$.

The typing rules also make use of 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.

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 $\{r\}$ 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 $\texttt{fix}$ 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]

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

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

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

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

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:

$$\frac{}{\begin{array}{l}\Psi; \Delta; \Gamma; C \vdash \texttt{newrgn}\, \rho, x \Rightarrow \\ \qquad \Delta\{\rho{:}\texttt{Rgn}\}; \Gamma\{x{:}\rho \texttt{ handle}\}; C \oplus \{\rho\}\end{array}} \ \text{(wrong)}$$

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

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:

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

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 [12, 37, 38]. 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

$$\begin{array}{l}\forall[\rho_1{:}\texttt{Rgn}, \rho_2{:}\texttt{Rgn}]. \\ \quad (\{\rho_1, \rho_2\}, \langle \texttt{int} \rangle \texttt{ at } \rho_1, \langle \texttt{int} \rangle \texttt{ at } \rho_2, \ldots) \rightarrow \texttt{0 at } r'\end{array}$$

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 [30, 31] 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:

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

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. The rules for capability equivalence appear in Appendix A.

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 $\overline{\epsilon}$.

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:

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

Suppose f has type $\forall[\rho_1{:}\texttt{Rgn}, \rho_2{:}\texttt{Rgn}].(\{\rho_1^+, \rho_2^+\}, \ldots) \to 0\,\texttt{at}\,r'$. 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 $\{r^1\} \leq \{r^+\} = \{r^+, r^+\}$.

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. One may write a function that can be called with extra capabilities 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:

$$\begin{array}{l}\forall[\rho{:}\texttt{Rgn}, \epsilon{:}\texttt{Cap}]. \\ \quad (\{\rho^1\} \oplus \epsilon, \rho\ \texttt{handle}, (\epsilon) \to 0 \ \texttt{at}\ r') \to 0 \ \texttt{at}\ r\end{array}$$

In our system $\epsilon \oplus \{\rho^1\} \not\leq \epsilon$. Consequently, before any function with this type can return (*i.e.*, call the continuation of type $(\epsilon) \to 0 \ \texttt{at}\ r'$), it must take action to satisfy the capability ϵ, that is, it must free ρ.

In general, our type system prevents "region leaks": programs must return all memory resources to the operating system before they terminate (Theorem 2.4). 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:

$$\begin{array}{l}\forall[\rho_1{:}\texttt{Rgn}, \rho_2{:}\texttt{Rgn}]. \\ \quad (\{\rho_1^+, \rho_2^+\}, \ldots, (\{\rho_1^+, \rho_2^+\}, \ldots) \to 0 \ \texttt{at}\ r') \to 0 \ \texttt{at}\ r''\end{array}$$

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:

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

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:

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

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.

2.4 Formal Properties of the Calculus

The most important properties of the Capability Calculus are Type Soundness and Complete Collection. Each can be proven from the formal semantics in Appendix A.

Type Soundness states that we will never enter a *stuck state* during the execution of a well-typed program. A state (M, e) is *stuck* if there does not exist (M', e') such that $(M, e) \longmapsto (M', e')$ and e is not `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.*

The proof of soundness is straightforward, making use of the standard Subject Reduction and Progress lemmas. Progress states that well-typed states are not stuck, and Subject Reduction states that evaluation steps preserve well-typedness.

Lemma 2 (Subject Reduction)
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. $e =$ `halt` i

The Complete Collection property guarantees that well-typed terminating programs return all of their memory resources to the system before they halt.

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.

3 Expressiveness

Our work provides a type system in which to check region-annotated programs for safety, but we do not provide any new techniques for determining those annotations. Instead, we rely on existing region inference strategies [34, 1] to infer appropriate region annotations. In this section we illustrate how a variant of the Tofte-Talpin region language can be translated into the Capability Calculus. The translation is formalized in the companion technical report [5]. By composing Tofte and Birkedal's region inference algorithm [34] with this translation, we have a way to compile high-level languages into the Capability Calculus.

Our example considers 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.

```
%%% count in a Tofte-Talpin calculus variant
letregion ρ1, x_ρ1 in
letregion ρ2, x_ρ2 in
  letrec count [ρ] (x_ρ : ρ handle, x : ⟨int⟩ at ρ) at x_ρ1 =
    % count :
    %   ∀[ρ].(ρ handle, ⟨int⟩ at ρ) --{access(ρ1),access(ρ)}--> unit
    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
```

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 $\{access(\rho_1), access(\rho)\}$.

Operationally, the `letregion` command serves a purpose similar to a pair of `newrgn` and `freergn` declarations. A new region is allocated at the beginning of the `letregion` block and is automatically deallocated at the end of the block, resulting in a stack-like (LIFO) allocation pattern. Hence, the code above allocates two regions (ρ_1 and ρ_2), stores `count` in ρ_1, stores a boxed integer in ρ_2, calls `count`, and then deallocates ρ_1 and ρ_2.

The translation of this program into the Capability Calculus rewrites it in continuation-passing style, and converts effects information into capability requirements. One of the main tasks of the translation is the compilation of `letregion` blocks into `newrgn` and `freergn` declarations. The resulting program appears below. In the interest of clarity, we have simplified the actual output of the formal translation in a few ways.

```
%%% count in the Capability Calculus
let newrgn ρ1, x_ρ1 in
let newrgn ρ2, x_ρ2 in
let newrgn ρ3, x_ρ3 in
% capability held is {ρ1^1, ρ2^1, ρ3^1}
let count =
  (fix count
       [ρ:Rgn, ρ_cont:Rgn, ε ≤ {ρ1^+, ρ^+, ρ_cont^+}]
       (ε, x_ρ:ρ handle, x:⟨int⟩ at ρ, k:(ε) → 0 at ρ_cont) .
    % capability held is ε ≤ {ρ1^+, ρ^+, ρ_cont^+}
    let n = π0(x) in                    % ρ ok
      if0 n
        then k()                        % ρ_cont ok
        else
          let n' = n − 1 in
          let x' = ⟨n'⟩ at x_ρ in       % ρ ok
            count [ρ, ρ_cont, ε] (x_ρ, x', k)   % ρ1 ok
  ) at x_ρ1 in
let ten = ⟨10⟩ at x_ρ2 in
let cont =
  (λ ({ρ1^1, ρ2^1, ρ3^1}) .
    % capability held is {ρ1^1, ρ2^1, ρ3^1}
    let freergn x_ρ1 in                 % ρ1 unique
    let freergn x_ρ2 in                 % ρ2 unique
    let freergn x_ρ3 in                 % ρ3 unique
      halt 0
  ) at x_ρ3
in
  count [ρ2, ρ3, {ρ1^1, ρ2^1, ρ3^1}] (x_ρ2, ten, cont)
```

The translated program begins by allocating regions ρ_1 and ρ_2, and also allocates a third region ρ_3 to hold `count`'s continuation. 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. The continuation requires the capability $\{\rho_1^1, \rho_2^1, \rho_3^1\}$ in order to free the three regions. Hence ϵ is instantiated with the stronger capability needed by the continuation.

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 above example could be rewritten to have `ten` and `cont` share a region, without changing the code of `count` in any way:

```
%%% count with ten and cont sharing ρ2
let newrgn ρ1, x_ρ1 in
let newrgn ρ2, x_ρ2 in
% capability held is {ρ1^1, ρ2^1}
let count = ... as before ...
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}] (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 $\{\rho_1^1, \rho_2^1\} \leq \{\rho_1^+, \rho_2^+\} = \{\rho_1^+, \rho_2^+, \rho_2^+\}$).

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 below. Note that this optimization is not possible when region lifetimes must be lexically scoped.

```
%%% count with efficient memory usage
let newrgn ρ1, x_ρ1 in
let newrgn ρ2, x_ρ2 in
let newrgn ρ3, x_ρ3 in
% capability held is {ρ1^1, ρ2^1, ρ3^1}
let count =
  (fix count
       [ρ:Rgn, ρ_cont:Rgn, ε ≤ {ρ1^+, ρ_cont^+}]
       (ε ⊕ {ρ^1}, x_ρ:ρ handle, x:⟨int⟩ at ρ,
        k:∀(ε) → 0 at ρ_cont) .
    % capability held is ε ⊕ {ρ^1}
    let n = π0(x) in                    % ρ ok
    let freergn x_ρ in                  % ρ unique
      % capability held is ε
      if0 n
        then k()                        % ρ_cont ok
        else
          let n' = n − 1 in
          let newrgn ρ', x_ρ' in
          % capability held is ε ⊕ {ρ'^1}
          let x' = ⟨n'⟩ at x_ρ' in      % ρ' ok
            count [ρ', ρ_cont, ε] (x_ρ', x', k)   % ρ1 ok
  ) at x_ρ1 in
let ten = ⟨10⟩ at x_ρ2 in
let cont =
  (λ ({ρ1^1, ρ3^1}) .
    % capability held is {ρ1^1, ρ3^1}
    let freergn x_ρ1 in                 % ρ1 unique
    let freergn x_ρ3 in                 % ρ3 unique
      halt 0
  ) at x_ρ3
in
  count [ρ2, ρ3, {ρ1^1, ρ3^1}] (x_ρ2, ten, cont)
```

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 $\{\rho_1^1, \rho_2^1\} \not\leq \{\rho_1^+, \rho_2^+, \rho_2^1\}$. However, the program rewritten so that `count` and `cont` share a region does not fail at run time, and does typecheck, since $\{\rho_1^1, \rho_2^1\} \leq \{\rho_1^+, \rho_1^+, \rho_2^1\}$.

4 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.

4.1 Language Extensions

The primary goal of this work was the development of a low-level, type-safe language that gives compilers and programmers control over the allocation and deallocation of data. The language that we have described so far is relatively high-level as it includes abstract closures and high-

level operations such as the atomic allocation and initialization of data. In the companion technical report [5], we show that the capability constructs interact benignly with the process of type-preserving compilation described by Morrisett *et al.* [24] and we use the techniques described in this paper to modify their typed assembly language to allow explicit deallocation of data structures.

In this paper, we have concentrated on using the Capability Calculus to implement safe deallocation of memory, 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.

Capabilities can also be used 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 to sensitive data obtains the corresponding lock and its associated capability before accessing that data. When the code releases the lock, we revoke the capability on the data, just as we revoke a capability when we free a region.

In general, whenever a system wishes statically to restrict access to some data, and/or to ensure a certain sequence of operations are performed, it may consider using capabilities to check that the appropriate invariants are maintained.

4.2 Related Work

The Capability Calculus derives its lineage from the work of Gifford and Lucassen on type and effect systems [11, 20] and the subsequent study by many others [16, 33, 36, 34]. 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. 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*.

The addition of aliasing information to our capabilities also separates them from earlier work on effects systems. However, capabilities only express the simplest aliasing relationships: a region is either completely unaliased or it may alias any other region. Our capabilities reveal very little of the structure of the store. A number of other researchers [10, 7, 32] have studied static analyses that infer the shapes of data structures and the aliasing relationships between them. We plan to investigate how to use these finer-grained memory models to increase the flexibility of our type system.

A connection can also be drawn between capabilities and monadic type systems. Work relating effects to monads [21, 28, 18, 8] 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.

The experience of Birkedal *et al.* [4] with the ML Kit region compiler shows that there are many refinements to the basic system that will be necessary to make our Capability Calculus a practical intermediate language. In particular, Birkedal found that allocation often occurs in two different contexts: one context in which no live object remains in the region and a second context in which there may be live objects remaining in the region. In order to avoid code duplication and yet ensure efficient space usage, they check at run time to find out which situation has occurred. In the former case, they reset the region (deallocate and reallocate in our formalism) and in the latter case, they do not reset but continue allocating at the top of the region. The type system we present here is not powerful enough to encode these storage-mode polymorphic operations. In fact, it must be refined in two ways. First, this optimization demands finer-grained aliasing specifications that declare a region ρ does not alias some particular region ρ' but may alias other regions. Second, after we dynamically check which of the two contexts above we are in, we must refine the type of our capability. Harper and Morrisett's typecase [13] mechanism developed for the TIL compiler and further refined by Crary *et al.* [6] allows the sort of type refinement required here.

Aiken *et al.* [1] have also studied how to optimize the initial Tofte-Talpin region framework and they also allow regions to be independently deallocated. Furthermore, their system separates the naming of a region from its allocation. Our language, as presented, does not make such a distinction, but it is straightforward to add one. With such a mechanism in place, we conjecture, based on the soundness proof for Aiken *et al.*'s analyses, that those analyses may be used to produce type correct code in the Capability Calculus.

Gay and Aiken [9] have developed a safe region implementation that gives programmers control over region allocation and deallocation. They use reference counting to ensure safety. Hawblitzel and von Eicken [15] 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.

5 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 can be compiled to a typed assembly language. We believe that the notion of capabilities that support statically checkable attenuation, amplification, and revocation is an effective new tool for language designers.

6 Acknowledgements

We would like to thank Lars Birkedal, Martin Elsman, Dan Grossman, Chris Hawblitzel, Fred Smith, Stephanie Weirich, Steve Zdancewic, and the anonymous reviewers for their comments and suggestions.

References

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

[2] 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.

[3] 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.

[4] 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.

[5] Karl Crary, David Walker, and Greg Morrisett. Typed memory management in a calculus of capabilities. Technical report, Cornell University, 1999.

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

[7] Alain Deutsch. Interprocedural may-alias analysis for pointers: Beyond k-limiting. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 230–241, Orlando, June 1994.

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

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

[10] Rakesh Ghiya and Laurie J. Hendren. Is it a tree, a DAG, or a cyclic graph? A shape analysis for heap-directed pointers in C. In *Twenty-Third ACM Symposium on Principles of Programming Languages*, pages 1–15, St. Petersburg Beach, Florida, January 1996.

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

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

[13] 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.

[14] 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.

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

[16] 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.

[17] Dexter Kozen. Efficient code certification. Technical Report 98-1661, Cornell University, January 1998.

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

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

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

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

[22] 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.

[23] 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.

[24] 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*, San Diego, January 1998.

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

[26] 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.

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

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

[29] 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.

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

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

[32] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Transactions on Programming Languages and Systems*, 20(1):1–50, January 1996.

[33] J.-P. Talpin and P. Jouvelot. Polymorphic type, region, and effect inference. *Journal of Functional Programming*, 2(3):245–271, July 1992.

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

[35] 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.

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

[37] 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.

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

[39] 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.

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

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

A Formal Semantics of the Capability Calculus

We 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 $M\{\nu.\ell \mapsto E\}$.

$(M, e) \longmapsto P$	
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 $M(\nu.\ell) = \langle v_0, \ldots, v_{n-1} \rangle \quad (0 \leq i < n)$
$\texttt{let}\, \texttt{newrgn}\, \rho, x\, \texttt{in}\, e'$	$(M\{\nu \mapsto \{\}\}, e'[\nu, \texttt{handle}(\nu)/\rho, x])$ where $\nu \notin M$ and $\nu \notin e'$
$\texttt{let}\, \texttt{freergn}\, (\texttt{handle}(\nu))\, \texttt{in}\, e'$ 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

Judgement	Meaning
$\Delta \vdash \Delta'$	Constructor context Δ' is well-formed.
$\Delta \vdash c : \kappa$	Constructor c has kind κ.
$\Delta \vdash \Delta_1 = \Delta_2$	Constructor contexts Δ_1 and Δ_2 are equal.
$\Delta \vdash c_1 = c_2 : \kappa$	Constructors c_1 and c_2 are equal in kind κ.
$\Delta \vdash C_1 \leq C_2$	Capability C_1 is a subcapability of C_2.
$\vdash \Psi$	Memory type Ψ is well-formed.
$\vdash \Upsilon$	Region type Υ is well-formed.
$\Psi; \Delta; \Gamma \vdash v : \tau$	Word value v has type τ.
$\Psi; \Delta; \Gamma \vdash h\, \texttt{at}\, r : \tau$	Heap value h (residing in region r) has type τ.
$\Psi; \Delta; \Gamma; C \vdash d \Rightarrow \Delta'; \Gamma'; C'$	Declaration d is well-formed and produces constructor context Δ', value context Γ' and capability C'.
$\Psi; \Delta; \Gamma; C \vdash e$	Expression e is well-formed.
$\Psi \vdash C$ sat	Memories with type Ψ satisfy the capability C. That is, capability C contains exactly the regions in the domain of Ψ, and unique capabilities appear exactly once in C.
$\Psi \vdash R\, \texttt{at}\, \nu : \Upsilon$	Region R (named ν) has region type Υ.
$\vdash M : \Psi$	Memory M has memory type Ψ.
$\vdash P$	Machine state P is well-formed.

Figure 3: Capability Static Semantics: Judgements

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

$$\frac{}{\Delta \vdash \cdot} \qquad \frac{\Delta \vdash \Delta'}{\Delta \vdash \Delta', \alpha{:}\kappa}\ (\alpha \notin Dom(\Delta\Delta')) \qquad \frac{\Delta \vdash \Delta' \quad \Delta\Delta' \vdash C : \texttt{Cap}}{\Delta \vdash \Delta', \epsilon \leq C}\ (\epsilon \notin Dom(\Delta\Delta'))$$

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

$$\frac{}{\Delta \vdash \alpha : \kappa}\ (\Delta(\alpha) = \kappa) \qquad \frac{}{\Delta \vdash \epsilon : \texttt{Cap}}\ ((\epsilon \leq C) \in \Delta) \qquad \frac{}{\Delta \vdash \texttt{int} : \texttt{Type}} \qquad \frac{\Delta \vdash r : \texttt{Rgn}}{\Delta \vdash r\ \texttt{handle} : \texttt{Type}}$$

$$\frac{\Delta \vdash \tau_i : \texttt{Type} \quad (\text{for } 1 \leq i \leq n) \quad \Delta \vdash r : \texttt{Rgn}}{\Delta \vdash \langle \tau_1, \ldots, \tau_n \rangle\ \texttt{at}\ r : \texttt{Type}} \qquad \frac{\Delta \vdash \Delta' \quad \Delta\Delta' \vdash \tau_i : \texttt{Type} \quad (\text{for } 1 \leq i \leq n) \quad \Delta\Delta' \vdash C : \texttt{Cap} \quad \Delta \vdash r : \texttt{Rgn}}{\Delta \vdash \forall[\Delta'].(C, \tau_1, \ldots, \tau_n) \to \texttt{0}\ \texttt{at}\ r : \texttt{Type}}$$

$$\frac{}{\Delta \vdash \nu : \texttt{Rgn}} \qquad \frac{}{\Delta \vdash \emptyset : \texttt{Cap}} \qquad \frac{\Delta \vdash r : \texttt{Rgn}}{\Delta \vdash \{r^{\varphi}\} : \texttt{Cap}} \qquad \frac{\Delta \vdash C_1 : \texttt{Cap} \quad \Delta \vdash C_2 : \texttt{Cap}}{\Delta \vdash C_1 \oplus C_2 : \texttt{Cap}} \qquad \frac{\Delta \vdash C : \texttt{Cap}}{\Delta \vdash \overline{C} : \texttt{Cap}}$$

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

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

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

$$\frac{}{\Delta \vdash \cdot = \cdot} \qquad \frac{\Delta \vdash \Delta_1 = \Delta_2}{\Delta \vdash \Delta_1, \alpha{:}\kappa = \Delta_2, \alpha{:}\kappa}\ (\alpha \notin Dom(\Delta\Delta_1)) \qquad \frac{\Delta \vdash \Delta_1 = \Delta_2 \quad \Delta\Delta_1 \vdash C_1 = C_2 : \texttt{Cap}}{\Delta \vdash \Delta_1, \epsilon \leq C_1 = \Delta_2, \epsilon \leq C_2}\ (\epsilon \notin Dom(\Delta\Delta_1))$$

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

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

$$\frac{\Delta \vdash C : \texttt{Cap}}{\Delta \vdash \emptyset \oplus C = C : \texttt{Cap}} \qquad \frac{\Delta \vdash C_1 : \texttt{Cap} \quad \Delta \vdash C_2 : \texttt{Cap}}{\Delta \vdash C_1 \oplus C_2 = C_2 \oplus C_1 : \texttt{Cap}}$$

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

$$\frac{}{\Delta \vdash \overline{\emptyset} = \emptyset : \texttt{Cap}} \qquad \frac{\Delta \vdash r : \texttt{Rgn}}{\Delta \vdash \overline{\{r^1\}} = \{r^+\} : \texttt{Cap}} \qquad \frac{\Delta \vdash r : \texttt{Rgn}}{\Delta \vdash \overline{\{r^+\}} = \{r^+\} : \texttt{Cap}}$$

$$\frac{\Delta \vdash C : \texttt{Cap}}{\Delta \vdash \overline{\overline{C}} = \overline{C} : \texttt{Cap}} \qquad \frac{\Delta \vdash C_1 : \texttt{Cap} \quad \Delta \vdash C_2 : \texttt{Cap}}{\Delta \vdash \overline{C_1 \oplus C_2} = \overline{C_1} \oplus \overline{C_2} : \texttt{Cap}}$$

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

$$\frac{\Delta \vdash C_1 = C_2 : \texttt{Cap}}{\Delta \vdash C_1 \leq C_2} \qquad \frac{\Delta \vdash C_1 \leq C_2 \quad \Delta \vdash C_2 \leq C_3}{\Delta \vdash C_1 \leq C_3} \qquad \frac{}{\Delta \vdash \epsilon \leq C}\ ((\epsilon \leq C) \in \Delta)$$

$$\frac{\Delta \vdash C_1 \leq C_1' \quad \Delta \vdash C_2 \leq C_2'}{\Delta \vdash C_1 \oplus C_2 \leq C_1' \oplus C_2'} \qquad \frac{\Delta \vdash C \leq C'}{\Delta \vdash \overline{C} \leq \overline{C'}} \qquad \frac{\Delta \vdash C : \texttt{Cap}}{\Delta \vdash C \leq \overline{C}}$$

Figure 4: Capability Static Semantics: Type and Context Formation

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

$$\frac{\begin{array}{c}\Delta \vdash \Delta' \qquad \Delta\Delta' \vdash C : \texttt{Cap} \\ \Delta\Delta' \vdash \tau_i : \texttt{Type} \quad (\text{for } 1 \leq i \leq n) \qquad \Delta \vdash r : \texttt{Rgn} \\ \Psi; \Delta\Delta'; \Gamma\{f{:}\tau_f, x_1{:}\tau_1, \ldots, x_n{:}\tau_n\}; C \vdash e\end{array}}{\Psi; \Delta; \Gamma \vdash \texttt{fix}\, f[\Delta'](C, x_1{:}\tau_1, \ldots, x_n{:}\tau_n).e \texttt{ at } r : \tau_f} \quad \left(\begin{array}{l}\tau_f = \forall[\Delta'].(C, \tau_1, \ldots, \tau_n) \to \texttt{0 at } r \\ f, x_1, \ldots, x_n \notin Dom(\Gamma)\end{array}\right)$$

$$\frac{\Psi; \Delta; \Gamma \vdash v_i : \tau_i \quad (\text{for } 1 \leq i \leq n) \quad \Delta \vdash r : \texttt{Rgn}}{\Psi; \Delta; \Gamma \vdash \langle v_1, \ldots, v_n \rangle \texttt{ at } r : \langle \tau_1, \ldots, \tau_n \rangle \texttt{ at } r} \qquad \frac{\Psi; \Delta; \Gamma \vdash h \texttt{ at } r : \tau' \qquad \Delta \vdash \tau' = \tau : \texttt{Type}}{\Psi; \Delta; \Gamma \vdash h \texttt{ at } r : \tau}$$

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

$$\frac{}{\Psi; \Delta; \Gamma \vdash x : \tau} \ (\Gamma(x) = \tau) \qquad \frac{}{\Psi; \Delta; \Gamma \vdash i : \texttt{int}}$$

$$\frac{\Delta \vdash \langle \tau_1, \ldots, \tau_n \rangle \texttt{ at } \nu : \texttt{Type}}{\Psi; \Delta; \Gamma \vdash \nu.\ell : \langle \tau_1, \ldots, \tau_n \rangle \texttt{ at } \nu} \ (\nu \notin Dom(\Psi)) \qquad \frac{\Delta \vdash \forall[\Delta'].(C, \tau_1, \ldots, \tau_n) \to \texttt{0 at } \nu : \texttt{Type}}{\Psi; \Delta; \Gamma \vdash \nu.\ell : \forall[\Delta'].(C, \tau_1, \ldots, \tau_n) \to \texttt{0 at } \nu} \ (\nu \notin Dom(\Psi))$$

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

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

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

$$\frac{\Psi; \Delta; \Gamma \vdash v : \tau' \quad \Delta \vdash \tau' = \tau : \texttt{Type}}{\Psi; \Delta; \Gamma \vdash v : \tau}$$

Figure 5: Capability Static Semantics: Heap and Word Values

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

$$\frac{\Psi;\Delta;\Gamma \vdash v : \tau}{\Psi;\Delta;\Gamma;C \vdash x = v \Rightarrow \Delta;\Gamma\{x{:}\tau\};C}\ (x \notin Dom(\Gamma)) \qquad \frac{\Psi;\Delta;\Gamma \vdash v_1 : \texttt{int} \quad \Psi;\Delta;\Gamma \vdash v_2 : \texttt{int}}{\Psi;\Delta;\Gamma;C \vdash x = v_1\ p\ v_2 \Rightarrow \Delta;\Gamma\{x{:}\texttt{int}\};C}\ (x \notin Dom(\Gamma))$$

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

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

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

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

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

$$\frac{\Psi;\Delta;\Gamma;C \vdash d \Rightarrow \Delta';\Gamma';C' \qquad \Psi;\Delta';\Gamma';C' \vdash e}{\Psi;\Delta;\Gamma;C \vdash \texttt{let}\,d\,\texttt{in}\,e} \qquad \frac{\Psi;\Delta;\Gamma \vdash v : \texttt{int} \quad \Psi;\Delta;\Gamma;C \vdash e_2 \quad \Psi;\Delta;\Gamma;C \vdash e_3}{\Psi;\Delta;\Gamma;C \vdash \texttt{if0}\,v\,\texttt{then}\,e_2\,\texttt{else}\,e_3}$$

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

Figure 6: Capability Static Semantics: Declarations and Expressions

$\boxed{\Psi \vdash C\ \text{sat} \qquad \Psi \vdash R\ \texttt{at}\ \nu : \Upsilon \qquad \vdash M : \Psi \qquad \vdash P}$

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

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

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

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

Figure 7: Capability Static Semantics: Memory