Typed Memory Management in a Calculus of Capabilities

Karl Crary David Walker Greg Morrisett

July 23, 1998

Abstract

An increasing number of systems rely on programming language technology to ensure safety and security of low-level code. Unfortunately, these systems typically prevent manual management of memory, even by their resulting executable programs, and instead rely crucially on a complex but trusted garbage collector. Static memory management using regions provides a safe alternative to garbage collection by making memory allocation and deallocation explicit. 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 static 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. Unlike previous work, our language both does not require that region lifetimes be lexically scoped, and may be checked for safety without complex analyses. Furthermore, our safety 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. These capability mechanisms may also be used in a variety of other applications in which some operations are not always permissible, such as mutually exclusive data access in concurrent systems.

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 [13, 29, 2, 17, 16, 12, 11]. 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 [13] 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 [18, 17] have proposed Proof-Carrying Code (PCC) and Morrisett *et al.* [16] 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, PCC, and TAL), 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 [23, 24] 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 automatically region types and insert allocation and deallocation points into ML source code [22] as well as implement their theory [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 runtime 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. Both Birkedal *et al.* [4] and Aiken *et al.* [1] realized that a completely LIFO [de]allocation of regions would make poor use of memory in many cases. They proposed a series of optimizations that usually alleviate efficiency concerns and even improve upon traditional tracing garbage collection in some cases. 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. 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 you free* x*?"*

One solution is to free x when you can guarantee that it will no longer be accessed. Operating systems such as Hydra [30] 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. 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 runtime overhead.

We have a purely *syntactic* argument, based on Subject Reduction and Progress lemmas in the style of Felleisen and Wright, 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 [24], and the soundness of Aiken *et al.*'s optimizations [1] depend upon this argument.

We have a formal translation of a variant of the Tofte-Talpin language into our calculus [5], which we describe here via example. 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 concentrate here on an intermediate language and refer the reader interested in the TAL details to the companion technical report [5].

Finally, we have found that the typing mechanisms in the language can serve in other settings where there is a need to revoke access to objects. In particular, we outline how capabilities can be used to ensure mutually exclusive data access in concurrent systems.

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. The syntax of the language appears in Figure 1. Although direct-style formulations of the language are possible, they are slightly less elegant than a continuation-passing style (CPS) language and further removed from machine language. In the remainder of this section, we assume familiarity with CPS as we explain and motivate the calculus's constructs and typing rules.

2.1 Preliminaries

We specify the operational behavior of the capability calculus using an allocation semantics [14, 15, 16], 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

kinds	κ	::=	$\mathtt{Type} \mid \mathtt{Rgn} \mid \mathtt{Cap}$
constructor vars	α, ρ, ϵ		
uniqueness flags	μ	::=	$\cdot \mid *$
constructors	c	::=	$\alpha \mid \tau \mid r \mid C$
types	τ	::=	$\alpha \mid \mathtt{int} \mid r\ \mathtt{handle} \mid \forall[\Delta].(C, \tau_1, \ldots, \tau_n) \to \mathbf{0}\ \mathtt{at}\ r \mid \langle \tau_1, \ldots, \tau_n \rangle\ \mathtt{at}\ r$
regions	r	::=	$\rho \mid \nu$
capabilities	C	::=	$\epsilon \mid \emptyset \mid \{r^\mu\} \mid C_1 \oplus C_2$
constructor ctxts	Δ	::=	$\cdot \mid \Delta, \alpha{:}\kappa \mid \Delta, \epsilon \leq C$
value ctxts	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	::=	$\{\ell_1 \mapsto h_1, \ldots, \ell_n \mapsto h_n\}$
memory	M	::=	$\{\nu_1 \mapsto R_1, \ldots, \nu_n \mapsto R_n\}$
machine states	P	::=	(M, e)

Figure 1: Capability Syntax

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 values (v) will be substituted for value variables[1] 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, and variants of tuple and function types. If r is a region, then r $\texttt{handle}$ 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 $\forall[\,].(C, \tau_1, \ldots, \tau_n) \rightarrow \mathbf{0}\,\texttt{at}\,r$ 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 $\mathbf{0}$ 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.

[1] This semantics implicitly assumes that all values stored in a function closure are projected when the function is called.

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 $\forall[\Delta].(C, \vec{\tau}) \to \mathbf{0}\ \mathtt{at}\ r$ by $(C, \vec{\tau}) \to \mathbf{0}\ \mathtt{at}\ r$.

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

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

The second 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.

$$\frac{\Psi;\Delta;\Gamma \vdash v : r\ \texttt{handle} \quad \Psi;\Delta;\Gamma \vdash h\,\texttt{at}\,r : \tau\ \text{hval} \quad \Delta \vdash C = \{r\} \oplus C' : \texttt{Cap}}{\Psi;\Delta;\Gamma;C \vdash x = h\,\texttt{at}\,v \Rightarrow \Delta;\Gamma\{x:\tau\};C}\ (x \notin \mathit{Dom}(\Gamma))$$

Functions In order to call a function 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:

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

The body of a function may then assume the function's capability precondition is satisfied:[2]

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

In order to use a function in a variety of contexts, the required capability will usually be extended by a quantified capability variable. This variable serves as a row variable and is instantiated with the left-over capability. For example, functions with type

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

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

Allocation and Deallocation The most delicate issue is the typing of region allocation and deallocation. The naive typing rules would simply add and delete appropriate capabilities:

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

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

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

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

This function is well-typed 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] This rule simplifies the full rule for *fix* to the case where the function is neither polymorphic nor recursive.

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 [8, 27, 28]. 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

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

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 r. 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 [20, 21] 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 adding a second way to form a capability from a region. As before, $\{r\}$ is the capability to access region r. The new capability $\{r^*\}$ again permits accessing region r, but adds the additional information that r is unique; it represents a different region from any other region appearing in a capability formed using $\{r^*\}$. For example, the capability $\{r_1, r_2^*\}$ 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 $\{r^*\}$ guarantees that r does not appear anywhere else in the capability, it is the capability, not just to access r, but also to free r. Thus, we may type region deallocation with the rule:

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

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

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

Note that joining ordinary capabilities is idempotent, but joining capabilities containing uniqueness flags is not. The capabilities $\{r\}$ and $\{r, r\}$ are equivalent, but the capabilities $\{r^*\}$ and $\{r^*, r^*\}$ are not. The rules for capability equivalence appear in Appendix A.

Subcapabilities The capabilities $\{r^*\}$ 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 $\{r^*\} \leq \{r\}$. The various rules from Section 2.2 are modified to account for subcapabilities. For example, the function call rule becomes:

$$\frac{\Psi; \Delta; \Gamma \vdash v : (C', \tau_1, \ldots, \tau_n) \to \mathbf{0} \,\mathtt{at}\, r \quad \Delta \vdash C \leq \{r\} \oplus C'' \quad \Delta \vdash C \leq C' \quad \Psi; \Delta; \Gamma \vdash v_i : \tau_i}{\Psi; \Delta; \Gamma; C \vdash v(v_1, \ldots, v_n)}$$

Suppose f has type $\forall[\rho_1{:}\mathtt{Rgn}, \rho_2{:}\mathtt{Rgn}].(\{\rho_1, \rho_2\}, \ldots) \to \mathbf{0}\ \mathtt{at}\ r'$. 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 capability $\{r^*\}$, again by instantiating ρ_1 and ρ_2 with r, since $\{r^*\} \leq \{r\} = \{r, r\}$.

The subcapability relation accounts only for the forgetting of uniqueness information. Intuitively there should 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 that the $\{r_1, r_2\}$ should 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 state more expressive calling conventions and to prove a stronger property about space usage.

By omitting the axiom $C_1 \oplus C_2 \leq C_1$, our type system may express calling conventions that state who has responsibility for freeing a region. Failure to follow informal calling conventions on memory management responsibility 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:

$$\forall[\rho{:}\mathtt{Rgn}, \epsilon{:}\mathtt{Cap}].(\{\rho^*\} \oplus \epsilon, \rho\ \mathtt{handle}, (\epsilon) \to \mathbf{0}\ \mathtt{at}\ r') \to \mathbf{0}\ \mathtt{at}\ r$$

In our system $\{\rho^*\} \oplus \epsilon \not\leq \epsilon$. Consequently, before any function with this type can return, it must take action to satisfy the capability ϵ, that is, it must free ρ.

A related point is that we can prove that well-typed terminating programs cannot leak space (Theorem 2.4). 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.

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

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 capability $\{r^*\}$ and f has type:

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

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

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

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

We may instantiate ϵ with $\{r^*\}$ and then the return function 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:

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

If we hold capability $\{r^*\}$, we may call h by instantiating ρ_1 and ρ_2 with r and instantiating ϵ with $\{r^*\}$. This instantiation is permissible because $\{r^*\} \leq \{r, r\}$. As with g, the return function will possess the capability $\{r^*\}$, 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 returns we regain the stronger capability (to free r), although that capability was necessarily 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 when the function returns. Thus, we support statically checkable attenuation and amplification of capabilities.

2.4 Formal Properties of the Calculus

There are two important properties of the calculus: Type Soundness and Complete Collection. Both can be proven from the formal semantics given 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 $\mathtt{halt}\ w$. Complete Collection guarantees that well-typed programs do not have any memory leaks.

Theorem 2.1 (Type Soundness) *If* $\vdash (M, e)$ *and* $(M, e) \longmapsto^* (M', e')$ *then* (M', e') *is not stuck.*

Theorem 2.2 (Complete Collection) *If* $\vdash (M, e)$ *then either* (M, e) *diverges or* $(M, e) \longmapsto^* (\{\}, \mathtt{halt}\ w)$.

The proofs are straightforward and depend upon the usual Progress, Subject Reduction, and Type Substitution lemmas. In order to use the type system in practice, type checking should also be decidable. This property follows straightforwardly from the decidability of capability equality and subcapability judgements.

2.5 Expressiveness

In order to begin to demonstrate the expressiveness of our type system, we have defined an embedding of an explicitly-typed variant of the original Tofte-Talpin system into our calculus [5]. By composing Tofte's region inference algorithm with this translation, we have a way to compile high-level languages into our Capability Calculus. We demonstrate the most important aspects of the translation using the the following simple example:

```
letrgn ρ1, x_ρ1 in
  letrec count[ρ2:Rgn]({ρ1, ρ2}, x_ρ2:ρ2 handle, x:⟨int⟩ at ρ2) at x_ρ1=
    let n = π0 x in                                  % (1)
      if0 n then x
            else count[ρ2](x_ρ2, ⟨n − 1⟩ at x_ρ2)     % (2)
  in
  ⋮
  end % letrec
end % region scope and deallocate
```

The `count` function above takes a boxed integer as its argument as well as the handle for its argument's region. On each iteration, it allocates a new cell in region ρ_2 and decrements the argument. The `letrgn` ρ_1, x_{ρ_1} `in` e `end` construct allocates a new region ρ_1 when its scope begins and deallocates ρ_1 when its scope ends. The annotation $\{\rho_1, \rho_2\}$ on the `count` function serves a purpose similar to our capabilities as it describes the *effect* of the function. The projection at `(1)` produces a read effect on region ρ_2 while the allocation at `(2)` produces a write effect on ρ_2. The call to `count` produces a read effect on the function's region (ρ_1) in addition to the effect annotating the function. In this setting, there is no need to distinguish between read and write effects and thus we let the union of the regions ρ_1 and ρ_2 be the total effect of the function.

The translation into the Capability Calculus is a type-directed, type-preserving CPS transform [19, 7, 9] during which we transform effects into capabilities. We have simplified the formal translation of the `count` function here to illustrate a few key points:

```
let newrgn ρ1, x_ρ1 in
let count =
  fix count[ρ2:Rgn, ρ3:Rgn, ε:Cap, ε' ≤ ε ⊕ {ρ1, ρ2, ρ3}]
        (ε', x_ρ2:ρ2 handle, x:⟨int⟩ at ρ2, c:τ_cont) at x_ρ1 =
    let n = π0 x in
    if0 n then c(x)
    else
      let n' = n − 1 in
      let x' = ⟨n'⟩ at x_ρ2 in
      count[ρ2, ρ3, ε'](x_ρ2, x', c)
in
⋮
where τ_cont = (ε', ⟨int⟩ at ρ2) → 0 at ρ3
```

In continuation-passing style, count takes another argument: a continuation c that resides in some unknown region ρ_3. Since `count` may call its continuation, it requires the region ρ_3 in its capability. Because ρ_1 and ρ_2 appear in `count`'s effect, they must also appear in its capability. Finally, the continuation may need access to some other regions; these regions are hidden by the abstract variable ϵ.

Since `count` will not deallocate any of these regions itself, it has no need to know about any aliasing relationships. The continuation c, on the other hand, will deallocate all three regions and

does require this aliasing information. Bounded quantification is the mechanism we use to hide uniqueness information in `count` but reveal it in its continuation. A call site for `count` might look like:

```
% Current Capability: ε'' ⊕ {ρ1*, ρ4*, ρ5*}
% Continuation c' : (ε'' ⊕ {ρ1*, ρ4*, ρ5*}, ⟨int⟩ at ρ4) → 0 at ρ5
let n = ⟨6⟩ at x_ρ4 in
count[ρ4, ρ5, ε'', ε'' ⊕ {ρ1*, ρ4*, ρ5*}](x_ρ4, n, c')
```

In the examples above, even though `count` is tail-recursive, we allocate a new cell each time around the loop and do not deallocate any of the cells until it calls its continuation. If an aliasing and liveness analysis can determine that the region containing the initial argument to `count` has no other live values in it and is not aliased, we can safely deallocate the region containing the argument each time around the loop, reducing the linear space requirements of the function to a constant. In order to deallocate its argument, `count` requires the unique capability for region ρ_2, a fact we express in its type:

```
let newrgn ρ1, x_ρ1 in
let count =
  fix count[ρ2:Rgn, ρ3:Rgn, ε:Cap, ε' ≤ ε ⊕ {ρ1, ρ3}]
        (ε' ⊕ {ρ2*}, x_ρ2:ρ2 handle, x:⟨int⟩ at ρ2, c:τ_cont) at x_ρ1 =      % Note cap:  ε' ⊕ {ρ2*}
    let n = π0 x in
    if0 n then c[ρ2](x)
    else
      let freergn x_ρ2 in
      let n' = n − 1 in
      let allocrgn ρ6, x_ρ6 in
      let x' = ⟨n'⟩ at x_ρ6 in
      count[ρ6, ρ3, ε'](x_ρ6, x', c)
in
⋮
where τ_cont = ∀[ρ2:Rgn].(ε' ⊕ {ρ2*}, ρ2 handle, ⟨int⟩ at ρ2) → 0 at ρ3
```

3 Capabilities in Other Contexts

The design space for the Capability Calculus is quite large and variants of the simple calculus described in the previous section may be useful in other settings.

3.1 Capability-Based Typed Assembly Language

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 describe here is relatively high-level as it includes abstract closures and high-level operations

such as the atomic allocation and initialization of data. Nevertheless, the essential typing constructs can be modified for use in a typed assembly language in the style of Morrisett *et al.* [16]. More specifically, the existential types needed for closure conversion can be added without endangering the soundness of the type system, as can sums, arrays, floats, recursive types and other type constructors required to scale this theoretical calculus up to a practical compiler intermediate or target language.

3.2 Safe Refs

Concurrent programmers often use locks to ensure mutually exclusive access to shared mutable data. In order to access a shared object, a thread must first acquire the appropriate lock. When it has finished its computation, a thread relinquishes the lock so that others may access the object. Programmers must ensure that threads acquire and release locks appropriately lest they violate the mutual exclusion principle which the lock attempts to enforce. In their work on debugging Standard ML, Tolmach and Appel [26, 25] implemented an ML threads library which associates every shared reference with a lock and dynamically checks for possession of the lock at every reference access. With a few minor changes to the Capability Calculus, we can *statically* check that programmers obey a similar locking discipline. Static checking moves error detection to compilation phase of the software development cycle and eliminates the cost of repeated runtime checks. On the other hand, it is generally less flexible and requires that either the compiler or programmer annotate programs with typing information. A simple example shows how checking these *safe refs* falls into the framework of the Capability Calculus:

```
newlock ρ, x_ρ in        % create lock; capability: C ⊕ {ρ*}
let shared =
  sref 0 with lock ρ     % shared: (int, ρ) sref
in
release x_ρ in           % release lock to other threads; capability: C
⋮
acquire x_ρ in           % reacquire lock; capability: C' ⊕ {ρ*}
shared := !shared + 1    % checks for capability {ρ} and mutates shared
release x_ρ in           % release again
⋮
```

Creation and deletion of locks is very similar to the creation and deletion of regions and occurs via the commands `newlock` ρ, x_ρ, which binds the value `lock`(ρ) (the `handle` counterpart) to x_ρ, and `freelock` v, which given a `lock`(ρ) value, destroys the lock. These operations also add or remove $\{\rho^*\}$ from the capability in an identical manner to their analogous region operations. As in the region setting, capabilities mediate access to locked objects. There are two additional operations on locks: `acquire` x_ρ and `release` x_ρ. `acquire` waits on a value of type `lock`(ρ) until the thread obtains the unique capability associated with the lock. `release` gives up the unique capability $\{\rho^*\}$ to another waiting thread. Just as our region scheme ensured we could not terminate without first collecting all our garbage, no thread will exit without first returning all its locks.

4 Directions for Future Research

Aiken *et al.* [1] 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.

In practice, Birkedal *et al.* [4] 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 runtime 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, in our formalism, regions are either completely unaliased or may alias any other (non-unique) region. 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 [10] mechanism developed for the TIL compiler and further refined by Crary, Weirich, and Morrisett [6] allows the sort of type refinement required here. We plan to explore extensions of the Capability Calculus which include these features.

References

[1] A. Aiken, M. Fähndrich, and R. 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] B. Bershad, S. Savage, P. Pardyak, E. Sirer, M. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility, safety and performance in the SPIN operating system. In *Fifteenth ACM Symposium on Operating Systems Principles*, pages 267–284, Copper Mountain, Dec. 1995.

[3] L. Birkedal, N. Rothwell, M. Tofte, and D. N. Turner. The ML Kit (version 1). Technical Report 93/14, Department of Computer Science, University of Copenhagen, 1993.

[4] L. Birkedal, M. Tofte, and M. 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, Jan. 1996.

[5] K. Crary, D. Walker, and G. Morrisett. Typed memory management in a calculus of capabilities. Technical report, Cornell University, 1998. Forthcoming.

[6] K. Crary, S. Weirich, and G. Morrisett. Intensional polymorphism in type-erasure semantics. In *ACM SIGPLAN International Conference on Functional Programming*, Baltimore, Sept. 1998. To appear.

[7] O. Danvy and A. Filinski. Representing control: a study of the CPS transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, Dec. 1992.

[8] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[9] R. Harper and M. Lillibridge. Explicit polymorphism and CPS conversion. In *Twentieth ACM Symposium on Principles of Programming Languages*, pages 206–219, Charleston, Jan. 1993.

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

[11] C. Hawblitzel, C.-C. Chang, G. Czajkowski, D. Hu, and T. von Eicken. Implementing multiple protection domains in Java. In *1998 USENIX Annual Technical Conference*, New Orleans, June 1998.

[12] D. Kozen. Efficient code certification. Technical Report 98-1661, Cornell University, Jan. 1998.

[13] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.

[14] G. Morrisett, M. Felleisen, and R. Harper. Abstract models of memory management. In *ACM Conference on Functional Programming and Computer Architecture*, pages 66–77, La Jolla, June 1995.

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

[16] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. In *Twenty-Fifth ACM Symposium on Principles of Programming Languages*, San Diego, Jan. 1998.

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

[18] G. Necula and P. Lee. Safe kernel extensions without run-time checking. In *Proceedings of Operating System Design and Implementation*, pages 229–243, Seattle, Oct. 1996.

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

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

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

[22] M. Tofte and L. Birkedal. A region inference algorithm. *Transactions on Programming Languages and Systems*, Nov. 1998. To appear.

[23] M. Tofte and J.-P. 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, Jan. 1994.

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

[25] A. Tolmach. *Debugging Standard ML*. PhD thesis, Princeton University, 1992. Published as Princeton University Dept. of Computer Science Techical Report CS-TR-378-92.

[26] A. Tolmach and A. Appel. Debuggable concurrency extensions for Standard ML. In *Proceedings of ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 120–131, May 1991. Published as SIGPLAN Notices 26(12), December 1991.

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

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

[29] R. Wahbe, S. Lucco, T. Anderson, and S. Graham. Efficient software-based fault isolation. In *Fourteenth ACM Symposium on Operating Systems Principles*, pages 203–216, Asheville, Dec. 1993.

[30] 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:

- Bound variables alpha-vary.
- The expression $e[y/x]$ denotes the capture-avoiding substitution of y for x in e.
- Update of finite map X is denoted $X\{x \mapsto y\}$ or $X\{x{:}y\}$.
- Juxtaposition of two maps X and Y as in XY denotes an update of the first with the elements of the second.
- The notation $X \setminus x$ excludes x from the domain of map X.
- We abbreviate $X(\nu)(\ell)$ by $X(\nu.\ell)$.
- We abbreviate $X\{\nu \mapsto X(\nu)\{\ell \mapsto y\}\}$ by $X\{\nu.\ell \mapsto y\}$.

$(M, e) \longmapsto P$	
If $e =$	then $P =$
$\texttt{let}\, x = v\, \texttt{in}\, e'$	$(M, e'[v/x])$
$\texttt{let}\, x = v_1\ p\ v_2$	$(M, e'[k/x])$ where $v_1 = i$ and $v_2 = j$ and $k = i\ p\ j$
$\texttt{let}\, x = h\, \texttt{at}\, v\, \texttt{in}\, e'$	$(M', e'[\nu.\ell/x])$ where $v = \texttt{handle}(\nu)$ and $\nu \in Dom(M)$ and $\ell \notin Dom(M(\nu))$ and $M' = M\{\nu.\ell \mapsto h\}$
$\texttt{let}\, x = \pi_i v\, \texttt{in}\, e'$	$(M, e'[v_i/x])$ where $v = \nu.\ell$ and $\nu \in Dom(M)$ and $\ell \in Dom(M(\nu))$ and $M(\nu.\ell) = \langle v_0, \ldots, v_{n-1}\rangle \quad (0 \le i < n)$
$\texttt{let}\, \texttt{newrgn}\, \rho, x\, \texttt{in}\, e'$	$(M', e'[\nu, \texttt{handle}(\nu)/\rho, x])$ where $M' = M\{\nu \mapsto \{\}\} \quad (\nu \notin M$ and $\nu \notin e')$
$\texttt{let}\, \texttt{freergn}\, v\, \texttt{in}\, e'$	(M', e') where $v = \texttt{handle}(\nu)$ and $\nu \in Dom(M)$ and $M' = M \setminus \nu$
$\texttt{if0}\, 0\, \texttt{then}\, e_2\, \texttt{else}\, e_3$	(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 (assuming the well-formedness of left-hand side)
$\Delta \vdash \Delta'$	Δ' is a well-formed constructor context
$\Delta \vdash c : \kappa$	c is a well-formed constructor with kind κ
$\Delta \vdash \Delta_1 = \Delta_2$	Δ_1 and Δ_2 are equal constructor contexts
$\Delta \vdash c_1 = c_2 : \kappa$	c_1 and c_2 are equal constructors of kind κ
$\Delta \vdash C_1 \leq C_2$	C_1 is a subcapability of C_2
$\vdash \Psi$	Ψ is a well-formed memory type
$\vdash \Upsilon$	Υ is a well-formed region type
$\Psi; \Delta; \Gamma \vdash h\ \mathtt{at}\ r : \tau$ hval	h is a well-formed heap value of type τ in region r
$\Psi; \Delta; \Gamma \vdash v : \tau$	v is a well-formed word value of type τ
$\Psi; \Delta; \Gamma; C \vdash d \Rightarrow \Delta'; \Gamma'; C'$	d is a well-formed declaration producing constructor context Δ', value context Γ' and capability C'
$\Psi; \Delta; \Gamma; C \vdash e$	e is a well-formed expression
$\Psi \vdash C$ sat	Capability C contains exactly the regions in the domain of Ψ. Unique capabilities ($\{\nu^*\}$) appear at most once in C.
$\Psi \vdash R\ \mathtt{at}\ \nu : \Upsilon$	Region R with name ν is described by region type Υ
$\vdash M : \Psi$	Memory M is described by memory type Ψ.
$\vdash P$	Machine state P is well-formed.

Figure 3: Capability Static Semantics: Judgements

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

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

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

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

$$\frac{}{\Delta \vdash \nu : \mathtt{Rgn}} \qquad \frac{}{\Delta \vdash \emptyset : \mathtt{Cap}} \qquad \frac{\Delta \vdash r : \mathtt{Rgn}}{\Delta \vdash \{r^{\mu}\} : \mathtt{Cap}} \qquad \frac{\Delta \vdash C_1 : \mathtt{Cap} \quad \Delta \vdash C_2 : \mathtt{Cap}}{\Delta \vdash C_1 \oplus C_2 : \mathtt{Cap}}$$

$\boxed{\Delta \vdash \Delta_1 = \Delta_2 \quad \text{(congruence rules only)}}$

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

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

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

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

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

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

$$\frac{\Delta \vdash C_1 \leq C_1' \quad \Delta \vdash C_2 \leq C_2'}{\Delta \vdash C_1 \oplus C_2 \leq C_1' \oplus C_2'} \qquad \frac{\Delta \vdash r : \mathtt{Rgn}}{\Delta \vdash \{r^*\} \leq \{r\}}$$

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

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

Figure 4: Capability Static Semantics: Type Formation, Equality and Subtyping

$\boxed{\Psi;\Delta;\Gamma \vdash h\ \mathtt{at}\ r : \tau\ \text{hval} \qquad \Psi;\Delta;\Gamma \vdash v : \tau}$

$$\frac{\Delta \vdash \Delta' \quad \Delta\Delta' \vdash C : \mathtt{Cap} \quad \Delta\Delta' \vdash \tau_i : \mathtt{Type} \quad \Delta \vdash r : \mathtt{Rgn} \quad \Psi;\Delta\Delta';\Gamma\{f{:}\forall[\Delta'].(C,\tau_1,\ldots,\tau_n)\to \mathbf{0}\ \mathtt{at}\ r, x_1{:}\tau_1,\ldots,x_n{:}\tau_n\};C \vdash e}{\Psi;\Delta;\Gamma \vdash \mathtt{fix}\, f[\Delta'](C,x_1{:}\tau_1,\ldots,x_n{:}\tau_n).e\ \mathtt{at}\ r : \forall[\Delta'].(C,\tau_1,\ldots,\tau_n)\to \mathbf{0}\ \mathtt{at}\ r\ \text{hval}}\ (f, x_i \notin Dom(\Gamma))$$

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

$$\frac{\Psi;\Delta;\Gamma \vdash h\ \mathtt{at}\ r : \tau'\ \text{hval} \quad \Delta \vdash \tau' = \tau : \mathtt{Type}}{\Psi;\Delta;\Gamma \vdash h\ \mathtt{at}\ r : \tau\ \text{hval}}$$

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

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

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

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

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

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

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

Figure 5: Capability Static Semantics: Heap and Word Values

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

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

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

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

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

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

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

$$\frac{\Psi;\Delta;\Gamma;C \vdash d \Rightarrow \Delta';\Gamma';C' \qquad \Psi;\Delta';\Gamma';C' \vdash e}{\Psi;\Delta;\Gamma;C \vdash \mathtt{let}\,d\,\mathtt{in}\,e}$$

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

$$\frac{\begin{array}{c}\Psi;\Delta;\Gamma \vdash v : \forall[\,].(C', \tau_1, \ldots, \tau_n) \to \mathbf{0}\ \mathtt{at}\ r \qquad \Psi;\Delta;\Gamma \vdash v_i : \tau_i \\ \Delta \vdash C \le C'' \oplus \{r\} \qquad \Delta \vdash C \le C'\end{array}}{\Psi;\Delta;\Gamma;C \vdash v(v_1, \ldots, v_n)}$$

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

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

$$\frac{\cdot \vdash C = \{{\nu_1}^*, \ldots, {\nu_m}^*, \nu_{m+1}, \ldots, \nu_n\} : \mathtt{Cap}}{\Psi \vdash C\ \mathrm{sat}} \left(\begin{array}{l}\forall \nu.\nu \in Dom(\Psi) \Leftrightarrow (\exists i.1 \le i \le n \wedge \nu = \nu_i) \\ \forall i,j.(1 \le i \le m \wedge 1 \le j \le n \wedge i \ne j) \Rightarrow \nu_i \ne \nu_j\end{array}\right)$$

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

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

Figure 6: Capability Static Semantics: Terms and Programs